from *SIAM News*, Volume 33, Number 8

Elimination of Variables In Parallel

By Hoon Hong, Richard Liska, Nicolas Robidoux, and Stanly Steinberg

Elimination of variables is a fundamental task that arises frequently in the solution of scientific and engineering problems. Given an expression, the task is to find an equivalent expression that involves fewer variables.

As a trivial example, suppose that we want to solve the following system of equations:

$$x + y = 3$$
$$x - y = 1$$

We begin by adding the first equation to the second, obtaining an equation in x alone:

$$2x = 4$$

This, of course, is the well-known Gaussian elimination procedure, which has been studied extensively by numerical analysts.

Gaussian elimination is essentially a variable-elimination method. To make this point more clearly, we restate our example, using the language of elementary logic:

$$(\exists y)\ x + y = 3 \wedge x - y = 1 \Leftrightarrow 2x = 4$$

APPLICATIONS ON ADVANCED ARCHITECTURE COMPUTERS

Greg Astfalk, Editor

where $\exists$, the existential quantifier, denotes "there exists'' and $\wedge$ is the conjunction that denotes "and.'' Notice that while both sides of the equivalence, $\Leftrightarrow$, are in fact equivalent, the right hand side does not involve y, i.e., the variable y has been eliminated. In fact, the quantifier $\exists$ has been also eliminated; for this reason, the procedure is also called "quantifier elimination.''

A serious deficiency of Gaussian elimination is that it works only for an existentially quantified conjunction of linear equations. However, many real-life scientific and engineering problems deal with nonlinear equations, inequalities ($\geq, >, \cdots$), disjunctions ($\vee$, which stands for "or''), implications ($\Rightarrow$), and the universal quantifier ($\forall$, which denotes "for all").

In studying the stability of a numerical method for solving a certain PDE, we encounter the following expression:

$$\begin{aligned}&(\forall s)(\forall t)\ (c > 0) \wedge \\ &[\ (0 \leq s \leq 1 \wedge 0 \leq t \leq 1) \Rightarrow \\ &\qquad -2\,s\,t\,c^3 + 3\,s\,t\,c^2 + s\,c^2 \\ &\qquad -2\,s\,c + t\,c^2 - 2\,t\,c + 1 \geq 0\ ]\end{aligned}$$

For this problem we are interested in eliminating both the quantified variables s and t and the universal quantifiers, obtaining an equivalent expression that involves only the "free'' variable c. Other nontrivial examples of such problems can be found in [6, 7].

Clearly, problems of this type cannot be solved by Gaussian elimination. Fortunately, new methods, much more power-ful than Gaussian elimination, have been developed in the past several decades.

State of the Art

Around 1930, the renowned logician and mathematician Alfred Tarski [8] proved that all quantifier-elimination problems can be solved by a single method. In 1951, with the assistance of J.C.C. McKinsey (at the RAND Corporation), Tarski gave a rigorous and algorithmic description of the proof of this fundamental result in mathematical logic [9], in the hope that the algorithm might be programmed on a computer. The complexity of the algorithm, however, was prohibitive; the running time could not be bound by any finite tower of exponential functions.

Another breakthrough, a completely new method whose time complexity was only doubly exponential [2], reduced the height of the tower of exponential functions from infinity to just two. The method was based on a geometric construction called "cylindrical algebraic decomposition'' (CAD). This algorithm renewed hopes that a mechanical elimination procedure would be devised and inspired many further improvements, as well as several new methods.

Resulting from the reduced complexity of the CAD method were several software packages that can mechanically solve many problems of moderate size [1,3]. One such package, QEPCAD [4], solves the example given in the previous section in less than a second, producing:

$$0 < c \leq 1/2 \vee c = 1$$

This would not be an easy result to obtain by hand, or even with a general-purpose computer algebra system!

Because of excessive computation times and memory requirements, current quantifier-elimination software systems cannot solve large practical problems. For this reason, we were led to consider the possibility of utilizing parallel computers, and in particular to experiment with parallelization of the CAD method. Because quantifier elimination is not well known, we begin the discussion here with a brief and intuitive explanation of the sequential CAD method.

The CAD Method

Two general polynomials, $P_1(x,y)$ and $P_2(x, y)$, in the two real variables x and y can be used to illustrate the CAD algorithm. Figure 1 shows the plotted curves $P_1(x,y) = 0$ and $P_2(x,y) = 0$, the *cylinders* determined by these polynomials, and the division of these cylinders into *cells*.

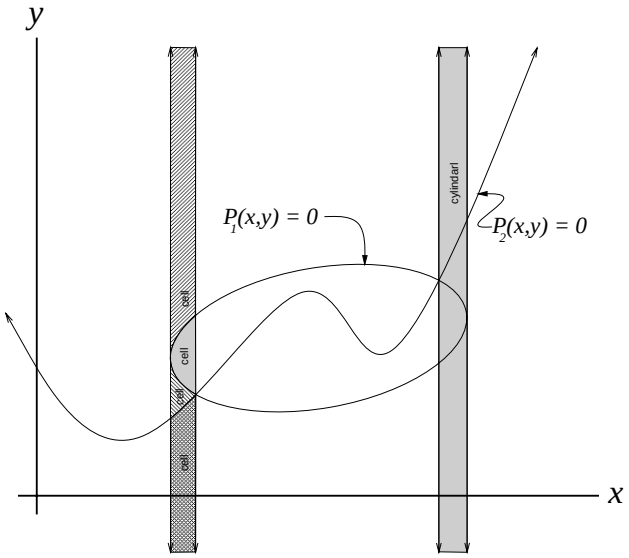

Figure 1. *Cylindrical decomposition of two polynomials.*

The first part of the CAD algorithm projects out the y variable and is therefore referred to as the *projection phase*. This requires that we look for all points along the x-axis where the polynomial curves intersect—that is, where there are simultaneous solutions of $P_1(x,y) = 0$ and $P_2(x,y) = 0$, or where one of the polynomial curves is vertical, i.e., where $P(x,y) = 0$ and $\partial P(x,y)/\partial y = 0$ (with P being one of the polynomials).

We actually proceed by using resultant and discriminant calculations to eliminate the y variable from the system of polynomial equations and thus to produce polynomial equations in only the x variable. The roots of these polynomials, marked along the x-axis, are given by algebraic numbers.

The next phase is the construction of a *stack* of cells in each cylinder. Through each of the special points on the x-axis, we draw a vertical line. These lines represent one-dimensional cylinders, and the regions between the lines are two-dimensional cylinders, as illustrated in the right-hand portion of Figure 1. The one-dimensional cylinders intersect the polynomial curves at points that are zero-dimensional cells. The regions on the one-dimensional cylinders between these points are one-dimensional cells. The two-dimensional cylinders are constructed so that they intersect the polynomial curves in pieces that can be described as graphs of functions. These graphs are one-dimensional cells, and the regions in the cylinders between the curves are two-dimensional cells. This is illustrated in the left-hand part of Figure 1.

Based on this description, we can illustrate the CAD algorithm for a simple example. Consider

$$\forall y \, \{x + y^2 \geq 0\}$$

whose CAD is illustrated in Figure 2. We find the point where the parabola becomes vertical by eliminating y from the system equations

$$\begin{aligned} P(x,y) &= x + y^2 = 0 \\ \partial/\partial y \, P(x,y) &= 2y = 0 \end{aligned}$$

which gives $x = 0$. The cylinders are thus given by $-\infty < x < 0$, $x = 0$, and $0 < x < \infty$. For the cylinder that is the left half of the x-axis, we chose a point in each of five cells, as shown in Figure 2, and then evaluated the truth of $x + y^2 > 0$ at each of those points. The cylinder $x = 0$ has three cells, and, again as shown in Figure 2, we chose a point in each cell and evaluated the truth of the polynomial at each of the points. For the cylinder on the right, which has only one cell, we chose a point in that cell. Because the quantifier is universal, all points in the cells of a cylinder must be true for the statement to be true in that cylinder. The quantifier-free formula, then, is

$$x = 0 \vee x > 0$$

which can be "simplified" to $x \geq 0$.

In higher-dimensional examples, both the projection and the stack constructions become much more complicated. The projection phase must be repeated, with the variables eliminated one at a time, until only one is left. The stack construction starts with the one-dimensional projection to build cylinders and cells in two dimensions. Over each cell in two dimensions, a cylinder of cells is then built in three dimensions. It is not difficult to see that the complexity of the algorithm grows very rapidly with increased dimensionality.

Many additional, important ideas are used in this algorithm, of course, but their explanation, involving ideas from modern algebra, logic, and algebraic geometry, is beyond the scope of this article. All these ideas together yield provably correct, finite, and imple-mentable algorithms for solving all quantifier-elimination problems.

The main difficulty for applications is computational complexity—the algorithm is doubly exponential in the number of input

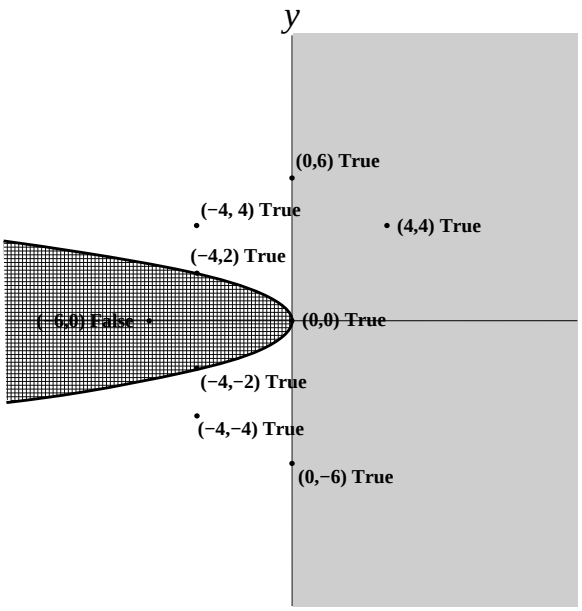

Figure 2. *Simple CAD.*

variables. Therefore, even for some modest problems, the algorithm does not finish in a reasonable amount of computer time—within, say, a day. This is the situation that led us to consider parallelism.

Going Parallel

Hong [5] had previously parallelized the stack-construction phase of the QEPCAD program on a workstation cluster by using Unix sockets. The use of sockets severely limits the number of processors that can be used. We re-implemented the code using MPI on the IBM SP2.

For many problems the projection phase is very fast, and we decided not to consider parallelizing this phase in the first round of our effort. The construction of the one-dimensional cells, i.e., breaking the real axis into cells, is also very fast; again, we decided not to parallelize this portion of the algorithm. The remaining cell-construction steps—building the stack of cells over one-dimensional cells—take most of the com-puter time. The statistics for our examples show that the number of cells can be quite large. Even in simple problems, development of approximately 10,000 cells is common. This is the focus of our parallelization effort.

The "P'' in QEPCAD, which stands for "partial,'' refers to a strategy for reducing the number of cells that need to be constructed. This strategy is dynamic and is one of the things that requires an interrupt in parallel implementations; this is discussed later.

Our parallelization strategy uses a manager node, which is responsible for sending tasks to the remaining nodes, the worker nodes. In this case the tasks serve to expand a particular cell—that is, to break the cylinder over that cell into a stack of higher-dimensional cells. The worker returns the constructed stack to the manager. The parallelization strategy, then, is as follows:

- Manager sends a task to a worker.
- Worker completes the task and returns its results to the manager.
- Manager includes the results in the global result.

Our strategy has two important features: (1) To avoid communication overhead, a worker, after expanding a given cell, continues to expand its subcells until reaching a limit based on the number of constructed stacks and on processor time. (2) The manager, after including a result returned by a worker, may find that some workers are expanding cells that are no longer needed; the manager must interrupt those workers.

The good news is that this approach produced speedups when the number of nodes did not exceed approximately five. Essentially no speedup was achieved for larger numbers of nodes. The reason is that the stack construction for most cells was very fast, with only a few com-putationally hard cells taking up most of the time. The approach had little scalability but was able to solve at least a few problems quickly. To gain scalability, we needed to move toward a finer-grain parallelization while, very importantly, not losing what had already been gained.

Many computer algebra algorithms are employed in stack construction. Fine-grained profiling revealed that most of the computer time used for the hard cells is spent in the computation of one, or just a few, greatest common divisors (GCDs) of polynomials with algebraic number coefficients. This algorithm was thus worth parallelizing. Before we discuss the parallelization of the GCD computation, a slight diversion is needed for some background information.

Fast Computer Algebra

In exact algebraic calculations with exact integers and rational numbers of arbitrary sizes, the expressions being manipulated grow large. A skilled user of a computer algebra system can control, but not eliminate, this growth. One of the most important tools for bounding expression growth is modular arithmetic. Use of this tool can be illustrated by an example: Choose a prime number, say 7. Modular arithmetic on integers is defined as performing arithmetic operations in the usual way and then dividing the answer by the chosen prime and keeping the remainder. In the case of 7, we see that we keep only the numbers 0, 1, 2, 3, 4, 5, 6 (the possible remainders on division by 7). Notice that $-1 = 6, -2 = 5, \cdots$. For arithmetic modulo 7, notice that

$$2 + 3 = 5, \quad 4 + 5 = 2, \quad \cdots$$
$$2 * 3 = 6, \quad 4 * 6 = 3, \quad \cdots$$

Because 7 is prime, we can always find a multiplicative inverse: $2 * 4 = 1, 3 * 5 = 1, \cdots$. That is, $1/2 = 2^{-1} = 4$, $1/3 = 3^{-1} = 5, \cdots$. The integer Euclidean algorithm is used to find multiplicative inverses.

A critical point is the classical Chinese remainder theorem. Suppose that we want to find an integer I that has given remainders when divided by two different primes:

$$I = 3 \bmod 5$$

$$I = 4 \bmod 7$$

It is easy to check that $I = 18$ plus some multiple of $5 * 7 = 35$. The Chinese remainder theorem provides an algorithm for answering such questions, where the remainder is given for any number of primes. The answer is given to a precision that is the product of the primes. Modular arithmetic and the Chinese remainder theorem form the core of many computer algebra algorithms. We could not factor large polynomials in computer algebra systems without this or closely related techniques. The primes used are typically much larger than 7, approximately half the integer word length or larger.

Wall-clock time (s)	10	20	30	40	50	60
Workers	1	2	4	8	12	12
Primes	6	18	42	90	162	234

Table 1. *Behavior of the parallel GCD algorithm.*

The advantage of modular arithmetic is that the arithmetic operations of addition, subtraction, multiplication, and division of integers take fixed amounts of time, just as for floating-point arithmetic. The disadvantage is that the Chinese remainder theorem must be used to construct the answer. In fact however, this technique makes many computer algebra calculations significantly faster than the direct use of large integers.

We now return to our parallelization problem. The Euclidean algorithm used by QEPCAD to compute the GCD of two polynomials with algebraic number coefficients uses the modular techniques just described. It has a main loop consisting of five basic steps:

1. Choose a prime p.
2. Compute the GCD modulo p.
3. Apply the Chinese remainder theorem.
4. Compute the rational coefficients for the polynomial.
5. Check the answer using a trial division.

Only the first two steps are independent. The others depend on results for all the previous primes. For hard GCDs a large number of primes, up to several hundred, must be used.

An obvious parallelization strategy is to choose several groups of primes and then pass each group to a different worker node. The difficulty is that we do not know, a priori, the total number of primes needed. It is only as the manager node sees the results of the modular GCDs that it can tell whether the full GCD computation is done. A better approach to parallelization is required.

Profiling has shown that the computation of one GCD modulo p takes only a short time. This time is constant for a particular problem because of the use of modular arithmetic. To reduce communication costs, we want each processor (i.e., worker node) to work for a time Δt before communicating with the manager node; Δt is an adjustable parameter. A node working on a GCD measures the time t_p needed to compute one GCD modulo p and then sets $n = \Delta t/t_p$. The strategy now is that if a worker node spends more than Δt time, it requests another worker node to help and sends a set of n primes to that worker node. That worker node then returns the set of n GCDs modulo those primes. When k nodes are working on a GCD and they are not finished at the end of Δt time, another k nodes will be requested. Because the number of primes needed cannot be effectively estimated, the node working on a GCD requests help only after spending Δt time computing.

Table 1 presents timing results for our algorithm with a GCD that needed 234 primes; in this case $\Delta t = 10$ seconds. The times in the table are wall-clock seconds. In the first 10 seconds, one worker finished six primes; it then asked for an additional worker. In the second 10 seconds, the two workers each computed $6 = (18 - 6)/2$ primes; both workers then requested additional helper nodes, and so forth. In serial, this computation would take approximately $10 * 234/6 = 390$ seconds. In parallel, it completed in 60 seconds—a speedup of more than 6. The number of workers used in achieving this speedup varied with time.

A complete profile for the GCD manager node with this example is shown in Table 2. These results confirm the estimate and show a speedup of approximately 8 when 12 processors are used. Further examples show similar parallel speedup results.

The GCD workers are only computing GCDs mod p, while the GCD manager is also performing other steps of the algorithm, such as checking whether enough primes have been chosen. At later times, after 20 seconds for example, the GCD manager is getting results from the GCD workers and processing those results, while the GCD workers continue. Table 2 indicates that the GCD manager spends 50 seconds on the GCD mod p computation. Table 1 shows that 60 seconds of wall-clock time were used to finish all the necessary GCD mod p computations. From the execution profile, it is obvious that parallelization of trial division is needed. Here, most of the time is spent in algebraic number arithmetic.

Implementation Technique

Implementation of this parallelization scheme required a rather complicated paradigm. We have a manager node and a pool of worker nodes. The manager has overall control of the computation and the pool of workers. In particular, the manager assigns tasks to the workers. A worker can request help from the manager, and the manager may or may not give a new worker to the requesting worker. The parallelization of the task has the following general structure:

- Worker asks manager for helper nodes.
- Manager gives, or does not give, helpers to a worker.
- Worker sends a subtask to a helper.

Computation phase	Time (s) Parallel	Sequential
GCD mod p	50	400
Chinese remainder	10	10
Coefficients	70	70
Trial divisions	500	500
Total	630	980

Table 2. *Execution profile for the GCD manager node.*

- Helper returns result to worker.
- Worker returns helpers to manager.

The QEPCAD algorithm is specifically designed to detect the possibility of early termination of tasks. For example, when the algorithm is working on an existence quantifier and finds a true case, or on a for-all quantifier and finds a false case, it terminates the computations. This is a dynamic process, however, and it must be possible to interrupt the workers.

Neither MPI nor, to our knowledge, any other message-passing library provides a method for "servicing" an interrupt. Because we are using MPI, we needed to simulate an interrupt, which we accomplished by picking a suitably frequent function call, the analog of the Lisp CONS function (a pointer allocation), and a parameter *n*. Every *n* times this function is called, a node checks for a reassignment message. If there is such a message, the node goes to the restart point and follows the instructions in the message; otherwise, it continues the current computation.

All this may seem easy, but there are quite a few possibilities for deadlock, some of which we discovered only via testing. When interrupting a worker, for example, it is important to interrupt its helpers as well. Additionally, we found that the performance depends critically on the manager's worker-allocation strategy.

Our strategy combining parallel stack construction and parallel GCD computations has been implemented in MPI and is currently running on an IBM SP2 computer. In the context of computer algebra, with the goal being the ability to tackle significantly harder problems, we believe that this work has been beneficial. There remain areas for significant improvement in performance, trial division in particular.

Acknowledgments

This research was supported in part by National Science Foundation grants INT–9212433 and CCR–9531828 and by the Czech Ministry of Education, grant Kontakt ME 050 (1997). The authors also thank the Maui High Performance Computing Center, the Albuquerque Resource Center and the Joint Supercomputing Center (of Czech Technical University, University of Chemical Technology, and IBM) for providing significant computing resources and assistance. The research was sponsored in part by the Phillips Laboratory, Air Force Materiel Command, USAF, through the use of the MHPCC under cooperative agreement F29601–93–2–0001. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Phillips Laboratory or the U.S. Government.

References

[1] B. Caviness and J. Johnson (eds.), *Quantifier elimination and cylindrical algebraic decomposition (Collins' 65th Birthday)*, in *Texts and Monographs in Symbolic Computation*, Springer-Verlag, 1996.

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

[3] H. Hong (ed.), *Computational Quantifier Elimination*, Comput. J., 36:5 (1993).

[4] H. Hong, *Improvements in CAD-based Quantifier Elimination*, PhD thesis, Ohio State University, 1990.

[5] H. Hong, *Parallelization of quantifier elimination on a workstation network*, in *Lecture Notes in Computer Science*, G. Cohen, T. Mora, and O. Moreno (eds.), AAECC–10, 673, Springer-Verlag, 1993, 170–179.

[6] H. Hong and R. Liska (eds.), *Applications of quantifier elimination*, special issue of J. Symb. Comput., 24 (1997).

[7] R. Liska and S. Steinberg, *Applying quantifier elimination to stability analysis of difference schemes*, Comput. J., 36:5 (1993), 497–503.

[8] A. Tarski, *The completeness of elementary algebra and geometry*, 1930, reprinted, Institute Blaise Pascal, 1967.

[9] A. Tarski, *A Decision Method for Elementary Algebra and Geometry*, 2nd ed., University of California Press, Berkeley, 1951.

Hoon Hong (hong@math.ncsu.edu) is in the Department of Mathematics at North Carolina State University. Richard Liska (liska@siduri.fjfi.cvut.cz) is with the Faculty of Nuclear Sciences and Physical Engineering, Czech Technical University in Prague. Nicolas Robidoux (n.robidoux@massey.ac.nz) is with the Mathematics Department at Massey University. Stanly Steinberg (stanly@math.unm.edu) is in the Department of Mathematics and Statistics at the University of New Mexico.