A Parallel Implementation of the Cylindrical Algebraic Decomposition Algorithm

B. David Saunders^{*} Hong R. Lee^{*} Department of Computer & Information Sciences University of Delaware Newark, Delaware 19711

S. Kamal Abdali[†] Division of Computer & Computation Research National Science Foundation Washington, DC 20550

Abstract

In this paper, we describe a parallelization scheme for Collins' cylindrical algebraic decomposition algorithm for quantifier elimination in the theory of real closed fields. We first discuss a parallel implementation of the computer algebra system SAC2 in which a complete sequential implementation of Collins' algorithm already exists. We report some initial results on the speedup obtained, drawing on a suite of examples previously given by Arnon.

1 Introduction

The elementary theory of Algebra and Geometry or theory of real closed fields is in essence the matter of deciding the validity of statements which can contain quantifiers and logical connectives and in which the atomic formulae consist of polynomial equations and inequalities. For example, $(\exists x)(\forall y)[(x^2 + y^2 > 1)\&(xy \ge 1)]$ is such a statement. The more general problem is that of quantifier elimination. Here the given formula may contain free variables, and the problem is to find a quantifierfree formula logically equivalent to the given one. For example, $(\forall x)[x^2 + bx + c \ge 0]$ contains free variables b and c, and a logically equivalent quantifier-free formula is $b^2 - 4c \ge 0$. If the given formula has no free variables, then the result of eliminating quantifiers is a formula which is just TRUE or FALSE.

The first algorithm for quantifier elimination was given by Tarski [?] in 1940. Although important theoretically for establishing that the theory of real closed fields is decidable, this algorithm turns out to be too inefficient to be of any practical use. The cylindrical algebraic decomposition (CAD) algorithm was invented by Collins [?] in 1973. A full implementation of the CAD algorithm was completed in 1981 by Arnon in the computer algebra system SAC2. A variant of the algorithm using *clustering* was later implemented in SAC2 also [?].

The CAD algorithm has found use in several areas other than quantifier elimination, e.g. robot motion planning [?], term-rewriting systems [?], algebraic topology [?], and computer graphics [?]. Algorithms other than Collins' have been proposed, e.g. [?, ?, ?, ?] and important improvements in the CAD approach have been offered, e.g. [?]. But at present no other algorithms have been implemented, and no complete implementation of even the CAD algorithm is available in any computer algebra system other than SAC2.

The CAD algorithm is doubly exponential in the number of real variables involved, and, in practice, can only solve rather small problems in a reasonable time. Nonetheless, many interesting and, indeed, unsolved problems can be expressed by a reasonably sized statement in the theory, involving only a few variables. This

 $^{^{*}{\}rm This}$ work has been supported by the National Science Foundation under Grant No. CDA-8805353.

[†]The statements contributed by this author to this paper represent his personal opinions, and should not be construed as the official viewpoint of the National Science Foundation.

pilot project explores the potential to increase the universe of solvable problems in the theory by exploiting multiprocessing.

Parallel processing architectures can provide dramatic speedup in the time needed to perform many computations, including algebraic ones. Here we obtain a speedup by using coarse grained parallelism on a multiprocessor with tens of processors and shared memory. Examples of such machines are the Sequent Balance and Symmetry models, the Encore Multimax, and the BBN Butterfly. The computations reported here were performed on a Sequent Symmetry with 8 processors.

In contrast to MIMD machines and the coarse grained approach taken here, one might consider SIMD designs and fine grained parallelism. Such an approach works best on homogeneously structured data, so that the synchronous processing can be effective. However, most data in algebraic computation is highly heterogeneous. For example, consider matrices whose entries are polynomials in several variables of differing degrees and term lengths. Computations on such matrices will call for subcomputations on the entries which differ dramatically in detail of instruction sequence and in time required. This makes it difficult to obtain good speedup on SIMD machines, with important exceptions for specific problems (see for example Johnson [?]).

On the other hand, with appropriate dynamic scheduling of tasks and load balancing good speedup may be obtainable using coarse grained parallelism on MIMD designs. In our experiments the processors operate asynchronously and communicate through shared memory. The risk exists that communication costs may override the benefits of the multiprocessing. However we have experienced approximately 50% efficiency in this initial effort (see Section 5 for details). From this modest experience we offer the modest conjecture that a factor of 10 speedup can be obtained on a wide range of computer algebra computations from a 20 processor machine.

The rest of the paper is organized as follows: Section 2 outlines the changes made to the SAC2 library to support parallel computation. Section 3 gives a brief overview of the CAD algorithm with references to the literature, and Section 4 describes the modifications made to the algorithm in order to parallelize the most time consuming portion, the extension phase. Section 5 offers the results of some timing experiments and preliminary analysis of the data. Finally, some conclusions and observations on these experiments are drawn in Section 6.

2 Parallelizing SAC2

The CAD algorithm, implemented by Collins and his students, was written in ALDES and depends on his

extensive library, SAC2, of algebraic procedures.

The first step in this effort was to adapt the SAC2 library for parallel execution on multiple processors. They are written in ALDES, which is translated into FORTRAN. The FORTRAN versions are then compiled and archived appropriately on a given machine and file system. At run time the routines call each other in various patterns, including recursive calls. Parameters and other variables manipulated for the most part are list structures representing algebraic entities. To handle the situation, variables are of type integer from FOR-TRAN's point of view. They serve as pointers to list cells. In SAC2 this means they are indices for the global shared array SPACE.

The other fundamental data structure of the system is the STACK array. The STACK, for each routine in the library, holds variables which must be subject to garbage collection. In SAC2 parlance, these variables are "unsafe". The SPACE array holds the list cells which are the basic units of all the data manipulated by procedures in the system. The basic need to adapt to the multiprocessor environment is to arrange for a stack for each process, while maintaining a common SPACE in shared memory. The stack for each process could be a private local entity, but we chose to make the STACK a global common entity like SPACE. For multiprocessing it is partitioned into segments, one for each process. If the STACK array has n elements unused at the time of multiprocess forking, and the machine has p processors, then each processor operates with its stack being a segment of length n/p in the global STACK array. There are two reasons for this choice. For one, a side effect of the scheme is that any process can access any other processor's stack. This capability is used in some versions of garbage collection with which we have been experimenting. Secondly, if each processor is to have a private local stack, there must be an effective way to share that stack with subroutines. A subroutine must share P's stack when called by P and Q's stack when called by Q. We didn't feel we had an appropriate (relatively machine independent) mechanism in the FORTRAN available to us on the Sequent to handle this situation.

An additional consideration in the stack implementation is the handling of the current stack pointer, INDEX. In the parallel version there must be an INDEX for each processor. We handle this by creating an array PINDX of *p* stack pointers, one for each process. Subroutines in the library call IUP when they start. This updates the INDEX appropriately. We replace this with PIUP, which uses the processor id to access and update the appropriate entry in PINDX and also set a *local* variable INDEX. Then the references to INDEX in the remainder of the code are valid and the number of modifications needed is kept to a minimum.

Since a global SPACE array is constantly accessed and

modified by all processes, there are, of course, some subroutines where greater adjustment is required to handle the mutual exclusion requirements and avoid racing conditions. In particular, routines which modify list structure (routines which write on SPACE) must be written carefully. Chief among these is COMP (the SAC2 equivalent of Lisp CONS). Mutual exclusion must be assured in the section in which COMP detaches a cell from the available cell list. Secondly, if no cells are available, garbage collection must be initiated, a process which affects the whole computation and involves all processors. At least, this is the case for SAC2's simple mark and sweep garbage collector where we make no attempt to do incremental or local garbage collection.

Our approach to garbage collection depends on the assumption that all processes will access the available cell list often (usually through COMP). There is a barrier at the beginning of garbage collection. The assumption is that processors can afford to spin at this barrier because other processors will soon also need a new cell, discover there are none, and join the earlier arrived ones at the barrier. When all processors have arrived, garbage collection commences. Our first implementation ignored one important situation in which this assumption (that all processes will seek new cells) is invalid. This situation occurs when, near the end of the parallel computation, some processes find no work left to do (see the section below on the CAD implementation) and exit. It is not hard to see that such a process will indeed not seek new cells. Our solution to this is greedy. We have finished processors spin on a GC flag. When an unfinished processor discovers the need for garbage collection, it raises the flag, and finished processors join the active ones at the barrier and participate in garbage collection. Our data indicates that this solution works well (see Section 5).

Garbage collection itself is done in two phases separated by a barrier. First each processor marks cells reachable from its STACK segment, then spins at a barrier until all processes are through marking. Next, each processor sweeps an assigned region in SPACE, and links the available cells it obtains into the global available cell list. The synchronization for the latter is managed by a lock.

Parallel performance can be heavily dependent on communication costs. In our situation the most uncertain and difficult to analyze or predict are the effects of SPACE array accesses. Each processor has a local cache. A high proportion of cache hits among the memory accesses is very valuable. In addition, there can be considerable contention on the bus for some memory access patterns. The data below, though not extensive, fails to show any serious problem in this area, at least for the CAD algorithm.

3 Cylindrical Algebraic Decomposition

Below we give a very brief description of the CAD algorithm, referring the reader to [?] for a thorough discussion of the algorithm and to [?] for a comprehensive bibliography of CAD theory and applications and other related work.

Let A be a finite set of polynomials in r variables with integer or rational coefficients. Let E^r denote the r-dimensional Euclidean space. The CAD algorithm decomposes E^r into a finite set of *cells* (disjoint, connected sets), in which at every point each polynomial in A has the same sign. This decomposition has two important properties:

- 1. It is cylindrical in the following sense: Given a region (a nonempty connected subset) R of E^k , the set $R \times E$ is called a cylinder (over R). The cells obtained in the decomposition of E^r can be joined into cylinders over certain regions of E^{r-1} , which can, in turn, be joined into cylinders over certain regions of E^{r-2} , and so on.
- 2. The decomposition is also *algebraic* in the sense that the cell boundaries are the zeros of certain polynomials obtained from the polynomials of A.

The CAD algorithm works in three phases:

- 1. Projection. From the given polynomials in r variables, polynomial resultant and discriminant operations are used to construct new polynomials containing, successively, r-1, r-2, ..., 1 variable(s). The so obtained k-variate polynomials have the property that their zeros are projections of certain critical aspects of the (k + 1)-variate polynomials, such as intersections and tangents to the direction of projection.
- 2. Base. The real zeros of the univariate polynomials obtained in the last step of the projection phase are used to decompose E^1 into cells which are either points (corresponding to polynomial zeros) or open intervals (between the zeros). Since each polynomial in A has the same sign throughout each of these intervals, a convenient rational sample point is chosen for each interval.
- 3. Extension. From the base phase decomposition of E^1 , successive decompositions of E^2 , E^3 , ..., E^r are derived. This is done by erecting cylinders over cells of lower dimensions, and partitioning these cells into regions throughout which appropriate polynomials from the projection phase have the same sign. During the extension from E^k to E^{k+1} , the (k + 1)-variate polynomials (obtained during

projection) are evaluated at the sample points in the cells of E^k , giving just univariate polynomials. The zeros of these polynomials provide the (k+1)st coordinates that, combined with the k coordinates of the cells over E^k , determine the decomposition of E^{k+1} .

The application of the CAD algorithm to the quantifier elimination problem is briefly as follows: Consider a logical formula which is comprised of logical connectives and quantifiers and in which the atomic formulae are equations and inequalities involving polynomials in r variables. The variables are assumed to range over real numbers. First consider the case in which the formula contains no free variables. In this decision problem what matters is the *sign pattern*— that is, the sequence of signs (positive, zero, or negative)— of the values of the polynomials (in some order) at all r-tuples of real numbers. The CAD algorithm decomposes the entire r-dimensional real space into a finite number of cells which are invariant as to sign pattern of the polynomials. Hence, one can determine whether the given formula is true or false from the values of the polynomials at the sample point of each cell. In the case that there are free variables in the formula, the algorithm uses the describing polynomials for each cell to construct a quantifier formula equivalent to the original formula.

4 Parallelizing Cylindrical Algebraic Decomposition

We now describe how we have modified the CAD algorithm to run it on the Sequent Symmetry computer. Sequent provides a FORTRAN preprocessor based on parallelizing DO loops with the DOACROSS directive [?], essentially a stripmining technique. This is of no use to us, since SAC2 is completely devoid of DO loops. Sequent also provides a suite of multitasking primitives, centered around a routine, mfork, which forks a specified number of identical processes. We used this suite for our parallelization. Other somewhat more machine independent approaches are possible, but Sequent's multitasking system was quite suitable for this pilot project.

The extension phase of the CAD algorithm was chosen for parallelization for two reasons. First, it dominates the time of computation in most instances. Second, it consists of completely independent work building a cylinder over each cell. The clustering versions of CAD do not have such complete independence, and clustering was not attempted here. However, we do not think that clustering is a significant impediment to parallelization.

The sequential CAD algorithm does the extension phase in a loop controlled by a structured list S of cells. Each loop iteration takes one cell and produces in its place the list of cells one dimension higher in the cylinder over the original cell. We use dynamic scheduling to parallelize this loop. First a copy S' of the structured list, with entries all zero, is created. Then a subroutine is mforked to run on p processors. The subroutine on each processor repeats the following

- 1. Take one cell from S (under lock),
- 2. extend the cell, creating a list s of higher dimension cells, and
- Replace by s the zero in S' in the position corresponding to the cell initially taken from S. Note that this does not have to be synchronized in any way.

When no cells remain in S, processors spin at a barrier until all are done. Then a single processor replaces Sby S' and makes a copy as before in preparation for extension to the next dimension.

The process is repeated r-1 times as we extend from 1 dimension to r dimensions. When no dimensions remain, finished processors spin on a GC flag as mentioned in the discussion of GC in Section 2.

The cell lists could very naturally be implemented as a shared database, in the style of the Linda [?] parallel programming system.

5 Timing Data and their Interpretation

We ran the sequential as well as the parallel version of the CAD algorithm on a number of problems taken from Arnon [?]. The problems attempted are the following:

• Ellipse. This example originates in the problem to determine whether the ellipse defined by the equation

$$\frac{(x-c)^2}{a^2} + \frac{(y-d)^2}{b^2} = 1$$

lies (without touching) wholly inside the circle defined by $x^2 + y^2 = 1$. Actually, at present the CAD implementations have only succeeded in solving the special case when d = 0. The input formula for quantifier elimination is therefore

$$(\forall x)(\forall y)((ab \neq 0 \& b^2(x-c)^2 + a^2y^2 - a^2b^2 = 0)$$

 $\Rightarrow x^2 + y^2 - 1 \le 0)$

The input presented to the CAD algorithm for the timings below consists of the following polynomials obtained by some manual projection and simplification [?]:

$$\begin{aligned} &a,a-1,b,b-1,b-a,c,c-1,c+1,c+a+1,c+a-1,\\ &c-a+1,c-a-1,b^2c^2+b^4-a^2b^2-b^2+a^2. \end{aligned}$$

	Problem	Time	No. of processors						
No.	Description		1	2	3	4	5	6	7
1	Ellipse,	Total	316.01	196.33	153.81	132.20	118.83	109.59	105.26
	1 Mw space,	GC	18.73	17.08	12.59	9.93	8.63	7.66	7.30
	4 GCs	Net	297.28	179.25	141.22	122.27	110.20	101.93	97.96
2	Ellipse,	Total	314.92	201.26	158.20	137.72	124.01	114.46	110.47
	2 Mw space,	GC	18.08	16.57	12.00	9.35	8.03	7.07	6.71
	2 GCs	Net	296.84	184.69	146.20	128.37	115.98	107.39	103.76
3	Ellipse,	Total	415.71	352.35	336.54	267.05	255.46	259.62	263.73
	4 Mw space,	GC	47.61	82.65	103.33	63.35	64.66	72.94	66.68
	1 GC	Net	368.10	269.70	233.21	203.70	190.81	186.68	197.05
4	Quartic,	Total	40.63	33.72	30.04	28.11	26.98	25.98	25.76
	2 Mw space,								
	$0 \mathrm{GCs}$								
5	Quartic,	Total	42.69	29.72	25.68	23.75	22.46	21.59	21.07
	0.2 Mw space,	GC	1.77	1.62	1.15	0.88	0.75	0.65	0.61
	2 GCs	Net	40.92	28.10	24.53	22.87	21.71	20.94	20.46
6	SIAM,	Total	14.69	14.68	12.95	12.63	12.10	10.97	11.74
	2 Mw space,								
	$0 \mathrm{GCs}$								
7	SIAM,	Total	15.01	9.50	7.77	6.95	6.83	6.13	6.13
	0.1 Mw space,	GC	0.45	0.40	0.29	0.23	0.19	0.16	0.15
	1 GCs	Net	14.56	9.10	7.48	6.72	6.64	5.97	5.98
8	Pair-5,	Total	250.76	150.70	126.93	98.06	96.08	94.69	95.39
	2 Mw space,	GC	8.52	7.82	5.47	4.12	3.48	3.03	2.85
	1 GC	Net	242.24	142.88	121.46	93.94	92.60	91.66	92.54
9	Tacnode,	Total	1607.06	897.58	865.15	847.00	842.81	835.44	837.01
	2 Mw space,	GC	94.50	86.36	60.62	45.77	38.66	33.37	31.53
	11 GC	Net	1512.56	811.22	804.53	801.23	804.15	802.07	805.48
10	Implicit,	Total	12976	7021	5110	4936	4883	4810	4781
	2 Mw space,	GC	629	553	405	322	280	295	286
	$70 \ \mathrm{GC}$	Net	12346	6467	4704	4613	4603	4514	4494
11	Pair2,	Total	34985	23657	13636	13335	13241	13141	13096
		# GCs	(194)	(196)	(199)	(199)	(199)	(199)	(199)
	2 Mw space,	GC	1751	1639	1125	864	763	677	633
		Net	33234	22018	12511	12470	12478	12463	12462

Figure 1: Execution times for parallel extension and parallel GC

• Quartic. The input polynomials for CAD are

$$p, 8pr - 9q^2 - 2p^3,$$

$$256r^3 - 128p^2r^2 + 144pq^2r + 16p^4r - 27q^4 - 4p^3q^2.$$

• SIAM. The input polynomials for CAD are

$$144y^{2} + 96x^{2}y + 9x^{4} + 105x^{2} + 70x - 98,$$
$$xy^{2} + 6xy + x^{3} + 9x.$$

• Pair-5. The input polynomials for CAD are

$$27xy + 9x^2 - 31x + 4,$$

$$5y^3 - 14xy^2 + 15y^2 + 13x^2y + 2xy + 14y - 7x^3 - 3x.$$

• Tacnode. The input polynomial for CAD is

$$y^4 - 2y^3 + y^2 - 3x^2y + 2x^4.$$

• Implicit The input polynomials for CAD are

$$505t^{3} - 864t^{2} + 570t + x - 343,$$

$$211t^{3} - 276t^{2} - 90t - y + 345.$$

• **Pair-2** The input polynomials for CAD are

$$9y^{2} + 30xy - 22x^{2} + 21,$$

$$2y^{3} - 12x^{2}y - 12xy - 8y + 11x^{2} - 2x - 2.$$

Problem	Exe	cution Ti	Fit & Error			
Number	C	oefficient	Coefficients			
	a	b	u	v		
1	77	241	-1	2.4	.008	
2	87	230	-2	3.6	.01	
3	297	133	-10	43.7	.10	
4	27	14	-1	.9	.02	
5	17	26	0	.2	.005	
6	15	1	-1	1.2	.08	
7	4	11	0	0.3	.02	
8	40	208	3	11.0	.04	
9	176	1333	73	120.5	.07	
10	-453	12897	509	405.2	.03	
11	3479	31494	682	4041.1	.12	

Figure 2: Timing model for parallel execution

We ran the parallel version of the CAD program by varying the number of processors from 1 through 7. Although our Sequent Symmetry configuration has eight processors, there is a system constraint that only seven processors can be taken over for exclusive use of an application program. For the Ellipse problem, we also varied the size of the SPACE array of SAC2 from 1 to 4 Megawords (Mw). When garbage collection occured in any run, we recorded the garbage collection time also. For these runs, we thus have both the total CPU time and the net time not including garbage collections. The results are shown in Figure 1 where the times are given in seconds.

Except where indicated, the GC times are measured from the time the *first* processor detects the need for GC. We also have measured the time the *last* processor encounters lack of available list cells, and arrives at GC. The resulting times are insignificantly lower, indicating that the approach taken to parallel GC works fine in this context.

In the Ellipse problem, we note that as the SPACE array is increased from 1 to 2, then to 4 Mw, the number of garbage collections decreases from 4 to 2, then to 1. The net CPU time in significantly higher for the 4 Mw SPACE array. Since this means that the memory demand of the program is above the 16 Mb physical memory, the penalty is probably due to increased memory swapping to disk. The improvement when we go from 2 Mw to 1 Mw is less pronounced and we conjecture that it is accounted for by the hardware design. Each processor has a 64 kb cache. Memory access time is fastest when (the valid instance of) the data addressed is in one's own cache, slower if it is in another processor's cache, and still slower when it is in main memory. When the SPACE array is smaller, a greater portion tends to reside in the caches. To further test the increased performance with smaller SPACE, We tried smaller sizes. The Ellipse problem runs out space when the size is much below 1 Mw. However, Figure 1 shows that the benefit of smaller space is even more pronounced when the size 0.1 Mw is used for the problems Quartic and SIAM.

To analyze the timing data obtained in our experiments, we looked for a fit to the following model. Suppose there are p processors working in parallel. Then the time for executing a task should be a function of the form

$$a + \frac{b}{p} + cp,$$

where a, b and c are constants that depend on the task. Here, a represents the part of the task that cannot be distributed among the processors, while b represents the parts that can be simultaneously performed by parallel processors. Even if the third term is deleted from the above formula, a non-zero value of a means that the speed up in performing the task is not perfectly efficient. But, in fact, matters are worse because there are parts in most tasks for which adding more processors can actually increase the exceution time. The purpose of the coefficient c is to account for such parts. For example, some processor initialization has to be done sequentially, once for each processor. More seriously, since in a shared-memory machine such as the Sequent, all processor/memory communication takes place over a

Time	seq.	No. of processors(parallel)						
		1	2	3	4	5	6	7
T_1	23.13	26.5	26.66	26.26	26.38	26.21	26.35	26.33
T_2	46.93	53.90	48.38	44.24	40.94	41.48	40.79	39.36
T_3	47.66	54.66	49.16	45.02	41.72	42.26	41.56	40.14
T_4	259.44	301.20	188.16	145.63	122.14	110.69	101.76	96.22
GC time	14.94	18.08	16.57	11.99	9.34	8.01	7.05	6.69
T_5	271.99	314.59	201.55	159.05	135.55	124.12	115.19	109.66
Speedup		.84	1.34	1.71	2.00	2.19	2.36	2.48
Efficiency		.84	.67	.57	.50	.44	.39	.35
$T_5 - GC$	257.05	296.51	184.98	147.06	126.21	116.11	108.14	102.97
$T_2 - T_1$	23.80	27.04	21.72	17.98	14.56	15.27	14.44	13.03
$T_4 - T_3$	211.78	246.54	139.00	100.61	80.42	68.43	60.20	56.08
$w/o \ GC$	196.84	228.46	122.43	88.62	71.08	60.42	53.15	49.39
T_p	235.58	273.58	160.72	118.59	94.98	83.70	74.64	69.11
Speedup		.86	1.47	1.99	2.48	2.81	3.16	3.41
Efficiency		.86	.73	.67	.62	.56	.53	.49

Figure 3: Timing of program segments

common bus, increasing the number of processors creates bottlenecks in the bus, and increases processing time. In the above model, we assume that such overheads are just proportional to the number of processors used, ignoring any quadratic or higher-order effects.

Figure 2 shows the values of a, b, and c that have been derived using a least-squares fit over the data in Figure 1. The values u, and v provide a measure of the absolute and relative accuracy of the fit. Specifically, the derivation of the quantities in Figure 2 is done as follows: Let **A** be the 7×3 matrix whose *i*th row is the vector [1, 1/i, i]. Then $\mathbf{x} = [a, b, c]^T$ is the best least squares fit to a solution of $\mathbf{A}\mathbf{x} = \mathbf{k}$, where **k** is the vector of the last seven numbers in each row of Figure 1. Furthermore, u is the 2-norm of $(\mathbf{A}\mathbf{x} - \mathbf{k})$, the absolute error of the best solution, and v is u/(2-norm of **k**).

Note that according to the above formula, the larger the value of b is compared to a and c for a program, the closer the program comes to achieving efficient linear speedup.

Below we compare one of these fits with detailed measurements of parallel and sequential segments of the computation. Just from the overall timings given in Figure 1, one can draw some conclusions about this model. It has some validity when the fit is extremely good, v < .01 say. But in problem 8 (Pair-5), for example, the coefficients suggest a successful parallelism; b = 208 is substantially larger than a = 40 and c is negligible. However, Figure 1 shows that the performance is essentially uninproved after p = 4, probably due to the computation time of two or three individual cell extensions.

We then recorded timings for individual sections of the computation to reveal the extent to which the model suggested above describes the sequential and parallel portions of the computation. In the table below, T_1 through T_5 denote clock readings at various points of the computation. The clock reading at the beginning is zero.

- 1. T_1 is the elapsed time to the beginning of extension. this first phase is on a single processor.
- 2. T_2 is the end of univariate extension into E^2 , which is done in parallel.
- 3. T_3 is the end of a brief single processor stage to initialize the structured list of cells for bivariate extension.
- 4. T_4 is the end of the second extension phase into E^3 , done in parallel. All garbage collection occurs during this extension.
- 5. T_5 is the end of the computation, after a final single processor phase.
- 6. T_p is $(T_2 T_1) + (T_4 T_3)$, the total time spent in the parallel phase.
- 7. For T_5 and T_p the speedups, (sequential time)/(*p*-processor time), and the efficiencies of the speedups, speedup/*p*, are also shown.

The times for the parallel version of the program on one processor are about 16% greater than the times for the sequential algorithm. This applies to the uniprocessor sections as well as the (potential) multiprocessor sections of the parallel algorithm. We suspect that this may be explained almost entirely by the cost of the lock

Problem		Time	No. of processors						
No.	Description		1	2	3	4	5	6	7
1	Ellipse	Expansion	247.26	143.10	104.19	80.31	68.37	61.56	56.70
		Idle	0	1.47	1.20	1.62	.85	1.53	1.73
		times		0	.43	.33	.65	1.44	1.30
					0	.25	.46	.87	.90
						0	.04	.58	.79
							0	.23	.37
								0	.27
									0
		total	0	1.47	1.63	2.20	2.0	4.65	5.36
2	Tacnode	Expansion	1601.49	891.93	859.52	841.42	837.20	829.15	831.41
		Idle	0	3.23	576.41	827.22	825.93	819.81	823.65
		times		0	7.28	821.35	823.61	819.63	822.70
					0	4.51	820.00	815.75	821.90
						0	3.82	813.77	817.71
							0	3.24	816.35
								0	4.36
									0
		total	0	3.23	583.68	1653.08	2473.36	3272.20	4106.67

Figure 4: Processor idling after exit from extension phase

on AVAIL, the available space list, which is invoked at every use of COMP.

One may compare the time spent in the nonparallelized sections of the program, approximately 40 seconds, with the value a = 101 from the previous table. This shows that 60% of the "sequential" time from the model T = a + b/p + cp is due to costs of synchronization and memory contention in the parallel phase. Indeed, as calculated, the efficiency of the speedup is around 40% and degrading slightly as p increases. We believe substantial improvements can be made in these figures by arranging that more of the computation be in local memory instead of the almost complete dependence on the global SPACE array as done here. Because of the highly independent nature of the extension of each cell in the CAD algorithm this localization should be very successful. It is less clear how well it will work when one includes adjacency and clustering calculations, which also should be done.

To further study the nature of efficiency losses, we measured the variation in the idle time of the processors at the end of each extension phase. Extension involves the construction of cylinders over n cells, requiring times $t_1, ..., t_n$. The times may be quite variable and the mapping onto p processors may be far from ideal. At the worst, the difference between the time the first processor exits (for lack of more cells to extend) and the time the last processor exits is the maximum of $t_1, ..., t_n$. Figure 4 shows the idle times that occur as the processors exit the extension phase. The two problems shown reveal the extremes. For Ellipse the load is quite well balanced, whereas for Tacnode, there are two processors busy for the duration, and p-2 idle for most of the time. Thus for Tacnode, at the granularity we are exploiting, the benefits of parallelism are limited by the two largest t_i . To overcome this would require exploitation of parallelism at a finer granularity, namely in the individual cell extensions, i.e., in the real root isolation algorithm.

Also note for Ellipse that since there is little idle time as the processors exit the extension phase, the efficiency losses (around 50%) are due to costs spread throughout the extension computation. They are due to locking and unlocking, references based on a processor id variable and the like. Parallelism at a finer granularity will not help here. It will only add overhead costs. For this type of problem, subprocesses need greater independence and lower overhead. The challenge seems to be to achieve greater independence of the subprocesses (less dependence on the single shared SPACE) and adaptive granularity.

6 Conclusions

We have implemented a parallel version of the SAC2 library, and have successfully parallelized the CAD algorithm. To our knowledge, this is the first parallel implementation of an important, very large computer algebra program. To many computer algebra professionals, SAC2 is a bit unattractive because of its FORTRAN base and primitive interface. But the FORTRAN code of SAC2 turned out to be a blessing in disguise for us, because FORTRAN fits quite naturally in the Sequent parallel programming environment. Of course, a Cbased computer algebra system, such as Maple, would also be relatively easy to parallelize. But as yet the CAD algorithm has not been written in any C-based system. To parallelize a Lisp-based system (e.g. RE-DUCE or Macsyma) would be rather difficult at present. Although there exist certain parallel versions of Lisp, the differences in constructs between sequential and parallel Lisps would require one to essentially recode major parts of the system in order to parallelize it.

When new procedures are written in ALDES, our approach requires some manual modification to the intermediate FORTRAN code. It would be desirable, and not terribly involved, to modify the ALDES translator to handle the needed variant forms for parallelism.

At the present state of parallel programming tools and environments, the parallelization of a large program takes much effort. Required is a thorough analysis of the sequential code in order to determine parallelizable parts of algorithms, critical sections, variable classification with respect to their locality to processes and their different modes of access by processes, etc. Moreover, each new architecture and operating environment seems to require all this work to be redone. For this reason, we are quite intrigued by, and plan to experiment with, the Linda system [?] which promises to be an architectureindependent environment for writing parallel programs.

Our implementation seems to have a 40–50% efficiency (the ratio of speed-up to the number of processors used). Although we would certainly like to improve it, we feel that an order of magnitude speed-up is possible even with the current implementation on a fully configured machine.

7 Acknowledgement

Ruth Shtokhamer initiated the project to parallelize SAC2. Dennis Arnon explained and clarified several aspects of the CAD algorithm to us. He (with further improvements by Jeremy Johnson) is also the author of the QE-interface program which makes working with the CAD algorithm infinitely simpler than in the raw SAC2 version.

References

- Arnon, D. S.: "Topologically reliable display of algebraic curves", *Proc. ACM SIGGRAPH '83*, Detroit, MI, 219–227. (1983)
- [2] Arnon, D. S.: "A bibliography of quantifier elimination for real closed fields", J. Symbolic Computation, 5, 267–274. (1988)

- [3] Arnon, D. S.: "A cluster-based cylindrical algebraic decomposition algorithm", J. Symbolic Computation, 5, 189–212. (1988)
- [4] Arnon, D. S., Collins, G. E., McCallum, S.: "Cylindrical algebraic decomposition I: the basic algorithms", SIAM J. Comp., 13, #4, 865–877. (Nov. 1984)
- [5] Arnon, D. S., Mignotte, M.: "On mechanical quantifier elimination for elementary algebra and geometry", J. Symbolic Computation, 5, 237–259. (1988)
- [6] Ben-Or, M., Kozen, D., Reif, R.: "The complexity of elementary algebra and geometry", *JCSS*, **32**, 251–264. (1986)
- [7] Canny, J.: "Some algebraic and geometric computations in PSPACE", STOC 88, 460–467. (May 1988)
- [8] Carriero, N., Gelernter, D.: "How to write parallel programs—A guide to the perplexed", Comp. Sci. Tech Report No. DCS/RR-628, Yale University (May 1988)
- [9] Collins, G.: "Quantifier elimination for real closed fields by cylindrical algebraic decomposition", *Lecture Notes in Computer Science*, **33**, Springerverlag, 134–183. (1975)
- [10] Dershowitz, N.: "A note on simplification orderings", Info. Proc. Letters, 9, 212–215. (1979)
- [11] Johnson, J.: Some Issues in Designing Alegbraic Algorithms for the CRAY X-MP, Master's thesis, Univ. Delaware, Newark, DE. (1987)
- [12] Kahn, P. J.: "Counting types of rigid frameworks", Inventiones Math., 55, 297–308. (1979)
- [13] Kozen, D., Yap, C-K.: "Algebraic cell decomposition", FOCS 87, 515–521. (October 1987)
- [14] Renegar, J.: "A faster PSPACE algorithm for deciding the existential theory of the reals", FOCS 88, 291–295. (October 1988)
- [15] Schwartz, J., Sharir, M.: "On the 'Piano Movers' problem II: General techniques for computing topological properties of real algebraic manifolds", Advances in Appl. Math., 4, 298–351. (1983)",
- [16] Sequent Computer Co.: A Guide to Parallel Programming, 2nd ed., Beaverton, OR. (1987)
- [17] Tarski, A.: A decision method for elementary algebra and geometry (2nd ed.), UC Berkeley Press, Berkeley (1951)