

Quality Up in Polynomial Homotopy Continuation by Multithreaded Path Tracking*

Jan Verschelde and Genady Yoffe
Department of Mathematics, Statistics, and Computer Science
University of Illinois at Chicago
851 South Morgan (M/C 249)
Chicago, IL 60607-7045, USA
jan@math.uic.edu, gyoffe2@uic.edu
www.math.uic.edu/~jan

2 September 2011

Abstract

Speedup measures how much faster we can solve the same problem using many cores. If we can afford to keep the execution time fixed, then quality up measures how much better the solution will be computed using many cores. In this paper we describe our multithreaded implementation to track one solution path defined by a polynomial homotopy. Limiting quality to accuracy and confusing accuracy with precision, we strive to offset the cost of multiprecision arithmetic running multithreaded code on many cores.

1 Introduction

Solving polynomial systems by homotopy continuation proceeds in two stages: we first define a family of systems (the homotopy) and then we track the solution paths defined by the homotopy. Tracking all paths is a pleasingly parallel computation. The problem we consider in this paper is to track one solution path. While tracking only one solution path could occur for huge problems (for which it is no longer feasible to compute all solutions), or the need to track one difficult solution path for which multiprecision arithmetic is required often arises for larger systems.

On a multicore workstation, we experimentally determined in [10] thresholds on the dimension of the problem to achieve a good speedup for the components of Newton method, using the quad double software library QD-2.3.9 [5]. While polynomial evaluation often dominates the computational cost, it pays off to run also multithreaded versions of the Gaussian elimination stage of Newton's method. In this paper we describe our multithreaded path tracker.

The idea to use floating-point arithmetic as implemented in QD-2.3.9 to increase the working precision dates back to [3]. In [8], this idea is described as an error-free transformations because

*This material is based upon work supported by the National Science Foundation under Grant No. 0713018 and Grant No. 1115777.

with double doubles we can calculate the error of a floating-point operation. From a computational complexity point of view, double doubles are attractive because the cost overhead is similar to working with complex arithmetic. For multithreading – using concurrent tasks accessing shared memory – double doubles are very attractive because blocking memory allocations and deallocations do not occur.

As in [10] we continue the development of double double and quad double arithmetic in our path trackers and experimentally determine thresholds on the dimensions and degrees to achieve a good speedup. In the next section we relate our calculations with the less commonly used notion of quality up [1]. As double floating-point arithmetic has become the norm, we can ask how much faster our hardware should become for double double arithmetic to become our default precision? We illustrate the computation of quality up factors in the next section.

In [10] we experienced that for homotopy continuation methods, polynomial evaluation is the dominating cost factor, exceeding the cost of the linear system solving as required in Newton’s method – although it still pays off to multithread Gaussian elimination. For Newton’s method we not only need to evaluate polynomials but also all derivatives with respect to all unknowns are needed in the Jacobian matrix. Using ideas from algorithmic differentiation [4] we have been able to reduce the dominating cost factor.

Another application area for the techniques of this paper is the deflation of isolated singularities [7]. The deflation method accurately locates singular solutions at the expense of adding higher derivatives to the original system, essentially doubling the dimension in every stage. Any implementation of this deflation will benefit from increased precision and efficient evaluation of polynomials and all their derivatives. In this setting the granularity of the parallelism must be fine and the use of multithreading is needed.

Acknowledgements. The ideas for the quality up section below were developed while preparing for an invited talk of the second author at the workshop on Hybrid Methodologies for Symbolic-Numeric Computation, held at MSRI from 17 to 19 November 2010. The second author is grateful to the organizers of this MSRI workshop for their invitation.

2 Speedup and Quality Up

When using multiple cores, we commonly ask how much faster we can solve a problem when using p cores. Denoting T_p the time on p cores, then the speedup is defined as T_1/T_p , i.e.: the time on 1 core divided by the time on p cores. In the optimal case, the speedup will converge to p : with p cores we can solve the same problem p times faster.

In addition to speedup, we like to know how much better we can solve the problem when using p cores? Our notion of quality up as an analogue to speedup is inspired by Selim Akl’s paper [1]. We define quality as the number of correct decimal places in the computed solution. Denoting Q_p as the quality obtained using p cores, we define quality up as Q_p/Q_1 , keeping the time fixed. As with speedup, we could also hope for a quality up factor of p in the optimal case.

Because the number of correct decimal places in a numerical solution depends on the sensitivity of the solution to perturbations in the input and is (bounded by condition numbers, we assume our problems are well-conditioned deliberately confusing working precision with accuracy. Taking

a narrow view on quality, we define

$$\text{quality up} = \frac{Q_p}{Q_1} = \frac{\# \text{ decimal places with } p \text{ cores}}{\# \text{ decimal places with 1 core}}.$$

Often multiprecision arithmetic is necessary to obtain meaningful answers and then we want to know how many cores we need to compensate for the overhead caused by software driven arithmetic. Using the quad double software library QD-2.3.9 [5], we experimentally determined in [10] that the computational overhead of using double double arithmetic over hardware double arithmetic on solving linear systems with LU factorization averaged around eight. This experimental factor of eight is about the same overhead factor of using complex arithmetic compared to real arithmetic. The number eight also equals the number of cores on our Mac OS X 3.2 Ghz Intel Xeon workstation.

To estimate the quality up factors, we assume an optimal (or constant) speedup. Moreover, we assume that the ratio Q_p/Q_1 is linear in p so we can apply linear extrapolation. To illustrate the estimation of the quality up factor, consider the refinement of the 1,747 generating cyclic 10-roots. The cyclic 10-roots problem belongs to a well known family of benchmark polynomial systems, see for instance [2], [6], or [9]. To compare quality up, we compare the 4.818 seconds of real time with one core using double double complex arithmetic to the 8.076 seconds of real time using quad double arithmetic using 8 cores. With 8 cores we double the accuracy in less than double the time. As this refinement is a pleasingly parallel calculation, the assumption that the speedup is optimal is natural. The concept of quality up requires a constant time, so we ask: how many cores do we need to reduce the calculation with quad doubles to 4.818s?

$$\frac{8.076}{4.818} \times 8 = 13.410 \Rightarrow 14 \text{ cores}$$

Denoting $y(p) = Q_p/Q_1$ and assuming $y(p)$ is linear in p , we have $y(1) = 1$ and $y(14) = 2$, so we interpolate:

$$y(p) - y(1) = \frac{y(14) - y(1)}{14 - 1}(p - 1).$$

and the quality up factor is $y(8) = 1 + \frac{7}{13} \approx 1.538$. The interpretation for the factor 1.538 is as follows: in keeping the total time fixed, we can increase the working precision with about 50% using 8 cores.

3 Multithreaded Path Tracking

Given a homotopy $h(\mathbf{x}, t) = \mathbf{0}$ and a start solution at $t = 0$, the path tracker returns the solution at the path at $t = 1$. A path tracker has three ingredients: a predictor for the next value of t and the extrapolated corresponding values for \mathbf{x} ; Newton's method as a corrector, keeping the predicted value for t fixed; and a step size control algorithm.

Algorithm 3.1 provides pseudo code for the multithreaded version of a path tracker. Every thread executes the same code. Variables that start with `my_` are local to the thread. The first thread manages the flags used for synchronization. Because threads are created once and remain

allocated to the path tracking process till the end, idle threads are not released to the operating system, but run a busy waiting loop.

Algorithm 3.1 (Multithreaded Path Tracking).

Input: $h(\mathbf{x}, t) = \mathbf{0}$, \mathbf{z} .	<i>homotopy and start solution</i>
Output: \mathbf{z} : $h(\mathbf{z}, 1) = \mathbf{0}$ or fail.	<i>solution at end of path or failure</i>
stop := false;	<i>initializations</i>
λ := initial step size;	<i>all other variables are set to 0</i>
while ($t < 1$ and not stop) do	
while (corr_Ind < my_corr_Ind) wait;	<i>wait till previous post correction</i>
if (my_ID = 1) then	<i>prediction done by thread 1</i>
predict(\mathbf{z}, t, λ);	<i>new \mathbf{z} and t</i>
pred_Ind := pred_Ind+1;	<i>signal that prediction done</i>
end if;	
while (pred_Ind < corr_Ind+1) wait;	<i>wait till prediction done</i>
Newton(my_ID, h , \mathbf{z} , ϵ , Max_It, success);	<i>run multithreaded Newton</i>
Newton_Ind[my_ID] := Newton_Ind[my_ID] + 1;	<i>thread my_ID is done</i>
while (\exists ID: Newton_Ind[ID] < corr_Ind+1) wait;	<i>wait till correction terminates</i>
if (my_ID = 1) then	<i>step size control by thread 1</i>
step_size_control(λ , success);	<i>adjust step size</i>
step_back(\mathbf{z} , t , success);	<i>step back if no success</i>
stop := stop_criterion(λ , corr_Ind);	<i>λ too small or corr_Ind too large</i>
corr_Ind := corr_Ind + 1;	<i>step size control is done</i>
end if;	
my_corr_Ind := my_corr_Ind + 1;	<i>continue to next step in while</i>
end while;	
fail := not stop.	<i>failure if stopped with $t < 1$</i>

Prediction and step size control are relatively inexpensive operations and are performed entirely by the first thread. Newton's method is computationally more involved and is executed in a multithreaded fashion.

4 Multithreaded Newton Method

In this section we focus on our multithreaded version of Newton's method using multithreaded polynomial evaluation and linear system solving described in [10]. Following the same notational conventions as in Algorithm 3.1, pseudo code is described in Algorithm 4.1 below.

Algorithm 4.1 (Multithreaded Newton's Method).

<p>Input: $h(\mathbf{x}, t) = \mathbf{0}$, \mathbf{z}; ϵ, Max.It. Output: \mathbf{z}: $\ h(\mathbf{z}, t)\ < \epsilon$ or fail.</p>	<p style="text-align: right;"><i>homotopy and initial solution</i> <i>tolerance and maximal #iterations</i> <i>corrected solution or failure</i></p>
<p>$i := 0$; $\ h(\mathbf{z}, t)\ := 1$; while ($\ h(\mathbf{z}, t)\ > \epsilon$) and ($i < \text{Max_It}$) do $V := \text{Monomial_Evaluation}(\text{my_ID}, h, \mathbf{z})$; $\text{Status_MonVal}[\text{my_ID}] := 1$; if ($\text{my_ID} = 1$) then while ($\exists \text{ID}: \text{Status_MonVal}[\text{ID}] = 0$) wait; for all ID do $\text{Status_MonVal}[\text{ID}] := 0$; $\text{Mon_Ind} := \text{Mon_Ind} + 1$; end if; while ($\text{Mon_Ind} < \text{my_Iter}+1$) wait; $Y := \text{Coefficient_Product}(\text{my_ID}, V, h)$; $\text{Status_Coeff}[\text{my_ID}] := 1$; if ($\text{my_ID} = 1$) then while ($\exists \text{ID}: \text{Status_Coeff}[\text{ID}] = 0$) wait; for all ID do $\text{Status_Coeff}[\text{ID}] := 0$; $\ h(\mathbf{z}, t)\ := \text{Residual}(Y)$; $\text{Coeff_Ind} := \text{Coeff_Ind} + 1$; end if; while ($\text{Coeff_Ind} < \text{my_Iter}+1$) wait; $Ab := \text{GE}(\text{my_ID}, \text{my_Iter}, Y, \text{pivots})$; $m := (n - 1)(\text{my_Iter}+1)$; while ($\exists \text{ID}: \text{pivots}[\text{ID}] < m$) wait; $\text{Back_Subs}(\text{my_ID}, \text{my_Iter}, Ab, \Delta\mathbf{z}, \text{BS_Ind})$; while ($\text{BS_Ind} < \text{my_Iter}+1$) wait; if ($\text{my_ID} = 1$) then $\mathbf{z} := \mathbf{z} + \Delta\mathbf{z}$; $i := i + 1$; $\mathbf{z_Ind} := \mathbf{z_Ind} + 1$; end if; while ($\mathbf{z_Ind} < \text{my_Iter}+1$) wait; $\text{my_Iter} := \text{my_Iter} + 1$; end while; fail := $\ h(\mathbf{z}, t)\ \geq \epsilon$.</p>	<p style="text-align: right;"><i>count #iterations</i> <i>initialize residual</i></p> <p style="text-align: right;"><i>multithreaded monomial evaluation</i> <i>thread done with monomial evaluation</i> <i>flag adjustments for next stage</i> <i>thread 1 waits</i> <i>flags reset for next stage</i> <i>update monomial counter</i></p> <p style="text-align: right;"><i>wait till all monomials are evaluated</i> <i>multiply monomials with coefficients</i> <i>thread done with coefficient product</i> <i>flag adjustments for next stage</i> <i>thread 1 waits</i> <i>flags reset for next stage</i> <i>calculate residual</i> <i>update coefficient counter</i></p> <p style="text-align: right;"><i>wait till all polynomials are evaluated</i> <i>row reduction on Jacobi matrix</i> <i>used for synchronization</i> <i>wait for row reduction to finish</i> <i>multithreaded back substitution</i> <i>wait till back substitution done</i></p> <p style="text-align: right;"><i>update solution</i> <i>counter to update \mathbf{z}</i></p> <p style="text-align: right;"><i>wait till solution is updated</i></p>

The array Y contains the evaluated polynomials of the polynomial system as defined by the homotopy $h(\mathbf{x}, t) = \mathbf{0}$ along with all partial derivatives as needed in the Jacobian matrix. The evaluation of the all polynomials as described in [10] occurs in two stages: first the values of all monomials are stored in V and then we multiply with the coefficients to obtain Y . The partitioning of the work load between the threads is such that no synchronization within the procedures `Monomial_Evaluation` and `Coefficient_Product` is needed.

The array Y contains then all the information needed to set up the linear system $A\mathbf{x} = \mathbf{b}$. The row reduction with pivoting is performed on the augmented matrix $[A \ \mathbf{b}]$, denoted in the algorithm by Ab . For numerical stability, we apply pivoting in the routine GE of Algorithm 4.1. The pivoting implies that within the procedure GE synchronization is needed. For synchronization in GE, we follow the same protocol: the first thread selects the pivot element (the largest number in the current column). After the selection of the pivot row, all threads can update their preassigned part of the matrix. The assignment of rows relates the row number of the identification number of the thread. The selection of the pivot row must wait till all threads have finished modifying their rows.

The output of the procedure GE is passed to the back substitution procedure Back_Subs. As the back substitution solves a triangular system, inside the routine synchronization is necessary for correct results.

To project the speedup for path tracking a system, we generate a system of 40 variables with a common support of 200 monomials. Every monomial has degree 40 on average with 80 as largest degree. We simulated 1,000 Newton iterations and results are reported in Table 1.

40-by-40 system, 1000 times					
#threads	Pol.Ev.	Gauss.El.	Back Subs.	Total	speedup
1	35.732s	4.849s	0.197s	40.778s	1
2	17.932s	3.113s	0.100s	21.145s	1.928
4	9.248s	1.824s	0.062s	11.134s	3.662
8	4.775s	1.349s	0.053s	6.177s	6.602

Table 1: Elapsed wall clock time for increasing number of threads for polynomial evaluation, Gaussian elimination and back substitution. The speedup is calculated for the total time.

For the generated problem the polynomial evaluation dominates the total cost. In Table 1 we see that once we reduced the cost of polynomial evaluation using 8 cores, the wall clock time becomes less than the total time spent on Gaussian elimination with one core. While multicore row reduction has a less favorable speedup compared to polynomial evaluation, we see that the multithreaded version is beneficial for the total speedup.

5 Effect of a Quadratic Predictor

Our multithreaded implementation achieves the better speedups the bigger is the ratio of dimension of the system to the number of engaged cores. Thus we work with larger dimensions. Secant predictor, which is merely efficient for systems of smaller dimensions becomes extremely unefficient for larger dimensions. We need to come up with a predictor, which would better approximate local intricate behaviour of a curve in multidimensional space. A suitable option for this proved to be the following quadratic predictor:

- Tracking a path, we keep approximate solutions x^* , x_{prev}^* , and x_{prev1}^* of intermediate systems, corresponding to the three most recent values of the homotopy parameter $t_{prev1} < t_{prev} < t$ respectively.

- For each index i , the coordinate $x^*[i]$ of the new initial guess x^* for the solution on the path of the intermediate system associated with the value of the homotopy parameter ($t +$ current step size), is computed independently of other coordinates as following:
 1. We interpolate points $(t_{prev1}, x_{prev1}^*[i])$, $(t_{prev}, x_{prev}^*[i])$, and $(t, x^*[i])$ by a parabola.
 2. The new $x^*[i]$ is then the value of this parabola at the point $(t +$ current step size).

Since each coordinate of such guess for a new intermediate system is computed independently of all the others, and all what it requires is computing just one value of interpolating three points parabola, which is done by a finite fixed number of algebraic operations, the complexity of such predictor depends linearly on the dimension of the system. Thus the portion of quadratic predictor computation in the entire path tracker computation is negligible. There is no reason to multitask quadratic predictor therefore, despite apparently it could be effectively done with a minimal effort. On the other hand, despite its very low computational time cost, use of the described above quadratic predictor instead of the secant predictor brings dramatic gain in the number of needed corrections to track a path. In our experiments we tracked on 8 cores a solution path for a system of dimension 20, with 20 monomials in each polynomial, with each monomial of maximal degree 2 using both predictors. When using the secant predictor, it required 113623 succesful corrections , with a running time 26m53.008s, minimal step size 6.10352e-07, and average step size 9.0404e-06, and when using the quadractic predictor, it required 572 succesful corrections , with a running time 8.863s, minimal step size 0.00016, and average step size 0.00019. In paricular the running time, when using quadratic predictor was about 180 less than when using the secant predictor. In all our other numerous experiments with systems of big enough various dimensions the gain of using the quadratic predictor kept to be of the same oreder. For a system of dimension 40 a run on 8 cores with a use of the quadratic predictor may take several minutes while a run with a use of the secant predictor may take several days.

The quadratic predictor provides very suitable balance between its low computational complexity and reduction in number of corrections it brings, thus ensuring a considerable, and probably one of the best possible, gain in absolute running time when tracking a path. The tables below show timings that illustrate the beneficial effect of using a quadratic predictor. On systems of the same dimension and degrees, Table 2 shows experimental results of runs with a secant predictor. Comparing the data of Table 2 with Table 3, we observe significant differences in the average and minimal step sizes along a path. Timings in Table 2 and 3 are for runs on eight cores.

6 Faster Evaluation of Polynomials and their Derivatives

In this section, we describe our application of techniques of algorithmic differentiation [4].

Along with each normalized monomial $x_{i_1}^{a_1} x_{i_2}^{a_2} \dots x_{i_k}^{a_k}$ with $1 \leq i_1 < i_2 < \dots < i_k \leq n$ and $a_1, \dots, a_k \geq 1$, which appears in the original homotopy $H(\mathbf{x}, t) = \mathbf{0}$, there appear associated to it monomials $x_{i_1}^{a_1-1} x_{i_2}^{a_2} \dots x_{i_k}^{a_k}$, $x_{i_1}^{a_1} x_{i_2}^{a_2-1} \dots x_{i_k}^{a_k}$, \dots , $x_{i_1}^{a_1} x_{i_2}^{a_2} \dots x_{i_k}^{a_k-1}$ in $\frac{\partial H}{\partial x_{i_1}}$, $\frac{\partial H}{\partial x_{i_2}}$, \dots , $\frac{\partial H}{\partial x_{i_k}}$ respectively. Employing the fact that the exponents of the monomial partial derivatives do not differ much from the exponents of the original monomial, we compute the values of the original monomial and of the k associated to it monomials in partial derivatives as follows:

20-by-20 systems, monomials of degree 10, secant predictor

	#succ. corrs	#corrs	time	avg step	min step
Syst. 1	141244	142657	3h55m3.559s	7.28E-06	1.22E-06
Syst. 2	176512	178274	8h34m5.082s	5.83E-06	3.05E-07
Syst. 3	150112	151612	7h5m29.649s	6.85E-06	3.05E-07
Syst. 4	125231	126483	7h26m11.352s	8.21E-06	1.22E-06
Syst. 5	187772	189645	9h19m29.869s	5.48E-06	3.05E-07
Average	156174.2	157734.2	7h16m7.900s	6.73E-06	6.71E-07
St. Dev.	25637.81	25892.2	2h4m31.303s	1.11E-06	5.01E-07

Table 2: For five differently generated systems, we respectively report the number of successful corrector stages, the total number of corrections, the total time, the average and minimal step size along a solution path, using a secant predictor.

20-by-20 systems, monomials of degree 10, quadratic predictor

	#succ.corrs	#corrs	time	avg step	min step
Syst. 1	571	624	0m59.552s	1.91E-03	3.13E-04
Syst. 2	791	864	2m34.505s	1.37E-03	7.81E-05
Syst. 3	668	730	2m4.725s	1.62E-03	3.91E-05
Syst. 4	528	578	1m39.336s	2.06E-03	1.56E-04
Syst. 5	848	924	2m39.015s	1.27E-03	7.81E-05
Average	681.2	744	1m59.427s	1.65E-03	1.33E-04
St. Dev.	137.538	149.124	0m41.275s	3.39E-04	1.09E-04

Table 3: For five differently generated systems, we respectively report the number of successful corrector stages, the total number of corrections, the total time, the average and minimal step size along a solution path, using a quadratic predictor.

1. We first compute the common factor $x_{i_1}^{a_1-1} x_{i_2}^{a_2-1} \dots x_{i_k}^{a_k-1}$ of the monomial and its derivatives.

2. We multiply the value of the common factor by $\omega_m = \prod_{\substack{j=1 \\ j \neq m}}^{j=k} x_{i_j}$, $m = 1, 2, \dots, k$, to get the values of the monomial partial derivatives.

3. We multiply $x_{i_1}^{a_1-1} x_{i_2}^{a_2} \dots x_{i_k}^{a_k}$ by x_{i_1} to obtain the original monomial.

The products ω_m , for $m = 1, 2, \dots, k$ we obtain in $3k - 6$ multiplications in the following fashion:

1. Recursively we get all products $\psi_m = x_{i_1} x_{i_2} \dots x_{i_m}$, $m = 1, 2, \dots, k - 1$ by $\psi_m = \psi_{m-1} x_{i_m}$, $\psi_1 = x_{i_1}$.
2. Similarly we obtain all products $\varphi_m = x_{i_k} x_{i_{k-1}} \dots x_{i_{k-m+1}}$, $m = 1, 2, \dots, k - 1$ by $\varphi_m = \varphi_{m-1} x_{i_{k-m+1}}$, $\varphi_1 = x_{i_k}$.

3. Finally we get the products ω_m $m = 1, 2, \dots, k$ as $\omega_1 = \psi_{k-1}$, $\omega_k = \varphi_{k-1}$, $\omega_m = \psi_{m-1}\varphi_{k-m+2}$, $m = 2, \dots, k-1$.

In [4], the evaluation of all derivatives of a product of variables is known as Speelpenning's example. Because we assume that our polynomials are sparse, we may focus on the individual monomials. For dense polynomials, a nested Horner scheme would be more appropriate.

7 Computational Experiments

The code was developed on a Mac OS X computer with two 3.2Ghz quad core Intel Xeon processors. For multithreading, we use the standard `pthread`s library and QD-2.3.9 for the quad double arithmetic.

In Table 4 we list one generated example for the complete integrated multithreaded version of the path tracker, for a system of dimension 40, once with polynomials of degree 2 and once with polynomial of degree 20. In the latter case, we get a close to optimal speedup and also for quadratic polynomials, the speedup is acceptable. Comparing with Table 5, we see that the degree of the polynomials are the determining factor in achieving a good speedup.

Dim=40, 40 monomials of degree 2 in a polynomial				
#threads	real	user	sys	speedup
1	5m25.509s	5m25.240s	0m0.254s	1
2	2m54.098s	5m47.506s	0m0.186s	1.870
4	1m38.316s	6m31.580s	0m0.206s	3.312
8	1m 2.257s	8m11.130s	0m0.352s	5.226
Dim=40, 40 monomials of degree 20 in a polynomial				
#threads	real	user	sys	speedup
1	244m55.691s	244m48.501s	0m 6.621s	1
2	123m 1.536s	245m53.987s	0m 3.838s	1.991
4	61m53.447s	247m14.921s	0m 4.181s	3.958
8	32m22.671s	256m27.142s	0m11.541s	7.567

Table 4: Elapsed real, user, system time, and speedup for tracking one path in complex quad double arithmetic on a system of dimension 40, once with quadrics, and once with polynomials of degree 20.

The results in Tables 4 and 5 are done without the faster evaluation of polynomials and their derivatives, so the degrees matter most in the speedup. With faster evaluation routines, the threshold on the dimension for the speedup will have to be higher.

We end this paper with some preliminary sequential timings on using faster evaluation and differentiation schemes. In our previous implementation, the continuation parameter t was treated as just another variable and this led to an overhead of a factor 3. Table 6 contains experimental results.

Dim=20, 20 monomials of degree 2 in a polynomial				
#threads	real	user	sys	speedup
1	0m37.853s	0m37.795s	0m0.037s	1
2	0m21.094s	0m42.011s	0m0.063s	1.794
4	0m12.804s	0m50.812s	0m0.061s	2.956
8	0m 8.721s	1m 8.646s	0m0.097s	4.340
Dim=20, 20 monomials of degree 10 in a polynomial				
#threads	real	user	sys	speedup
1	7m17.758s	7m17.617s	0m0.123s	1
2	3m42.742s	7m24.813s	0m0.206s	1.965
4	1m53.972s	7m34.386s	0m0.150s	3.841
8	0m59.742s	7m53.469s	0m0.279s	7.327

Table 5: Elapsed real, user, system time, and speedup for tracking one path in complex quad double arithmetic on a system of dimension 20, once with quadrics, and once with polynomials of degree 10.

8 Conclusions

For polynomial systems where the computational cost is dominated by the evaluation of polynomials, the multithreaded version of our path tracker performs already well in modest dimensions. For systems of lower degrees, the threshold dimension for good speedup will need to be higher because then the cost of Gaussian elimination becomes more important.

References

- [1] S.G. Akl. Superlinear performance in real-time parallel computation. *The Journal of Supercomputing*, 29(1):89–111, 2004.
- [2] Y. Dai, S. Kim, and M. Kojima. Computing all nonsingular solutions of cyclic-n polynomial using polyhedral homotopy continuation methods. *J. Comput. Appl. Math.*, 152(1-2):83–97, 2003.
- [3] T.J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971.
- [4] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, second edition, 2008.
- [5] Y. Hida, X.S. Li, and D.H. Bailey. Algorithms for quad-double precision floating point arithmetic. In *15th IEEE Symposium on Computer Arithmetic (Arith-15 2001), 11-17 June 2001, Vail, CO, USA*, pages 155–162. IEEE Computer Society, 2001. Shortened version of Technical Report LBNL-46996, software at <http://crd.lbl.gov/~dhbailey/mpdist/qd-2.3.9.tar.gz>.

20-by-20 systems, polynomial evaluation, 400 times, sequential execution				
degrees	new alg.time	old alg.time	speedup	pure speedup
5	4.267s	36.521s	8.59	2.85
10	7.280s	1m7.114s	9.22	3.07
20	11.304s	2m3.122s	10.89	3.63
40-by-40 systems, polynomial evaluation, 400 times, sequential execution				
degrees	new alg.time	old alg.time	speedup	pure speedup
5	19.855s	4m58.162s	15.02	5.01
10	36.737s	8m48.765s	14.39	4.80
20	1m1.980ss	16m4.541s	15.56	5.19

Table 6: For a system of dimension 20 and 40, for increasing degrees, we list the times of the new algorithm, the old algorithm and the speedup. The pure speedup is the speedup divided by 3, to account for treating the continuation parameter t differently.

- [6] T.L. Lee, T.Y. Li, and C.H. Tsai. HOM4PS-2.0: a software package for solving polynomial systems by the polyhedral homotopy continuation method. *Computing*, 83(2-3):109–133, 2008.
- [7] A. Leykin, J. Verschelde, and A. Zhao. Newton’s method with deflation for isolated singularities of polynomial systems. *Theoretical Computer Science*, 359(1-3):111–122, 2006.
- [8] S.M. Rump. Verification methods: Rigorous results using floating-point arithmetic. *Acta Numerica*, 19:287449, 2010.
- [9] J. Verschelde. Algorithm 795: PHCpack: A general-purpose solver for polynomial systems by homotopy continuation. *ACM Trans. Math. Softw.*, 25(2):251–276, 1999. Software available at <http://www.math.uic.edu/~jan/download.html>.
- [10] J. Verschelde and G. Yoffe. Polynomial homotopies on multicore workstations. In M.M. Maza and J.-L. Roch, editors, *Proceedings of the 2010 International Workshop on Parallel Symbolic Computation (PASCO 2010), July 21-23 2010, Grenoble, France*, pages 131–140. ACM, 2010.