Polynomial Homotopies on Multicore Workstations*

Jan Verschelde[†]

Genady Yoffe[‡]

1 July 2010

Abstract

Homotopy continuation methods to solve polynomial systems scale very well on parallel machines. In this paper we examine its parallel implementation on multiprocessor multicore workstations using threads. With more cores we can speed up pleasingly parallel path tracking jobs. In addition, we can compute solutions more accurately in the same amount of time with threads, and thus achieve quality up. Focusing on polynomial evaluation and linear system solving (the key ingredients of Newton's method) we can double the accuracy of the results with the quad doubles of QD-2.3.9 in less than double the time, if we use all available eight cores on our workstation.

1 Introduction

PHCpack is an open source software package for homotopy continuation methods to solve polynomial systems. For surveys and general introductions on homotopy continuation methods, see [32], [36], and [44].

The first public release of PHCpack is archived in [46]. Parallel implementations of various homotopy algorithms in PHCpack have been developed jointly with Yusong Wang [47, 49], Yan Zhuang [31, 48, 51], Yun Guan [23], and Anton Leykin [29, 30, 31]. All parallel homotopy algorithms described in [23, 29, 30, 31, 47, 48, 49, 51] use the message passing library MPI [42] for interprocess communication on clusters of Linux nodes, developed with personal clusters and tested on the NCSA supercomputers. Also other continuation software for polynomial systems (Bertini [8], HOM4PS-2.0para [34], PHoMpara [24], and POLSYS-GLP [45]) use MPI in their parallel implementations.

The current paper describes the use of multithreading in PHCpack, using tasks in Ada [13]. PHCpack is developed with the aid of the GNU-Ada compiler which maps tasks to kernel threads. The multitasking implementation of homotopies could complement the existing parallel versions in a multi-tiered approach: each node could run a more efficient multithreaded finer-grained

---

*Date: 17 June 2010. This material is based upon work supported by the National Science Foundation under Grant No. 0713018.

[†]Department of Mathematics, Statistics, and Computer Science, University of Illinois at Chicago, 851 South Morgan (M/C 249), Chicago, IL 60607-7045, USA. *Email:* jan@math.uic.edu. *URL:* <http://www.math.uic.edu/~jan>.

[‡]Department of Mathematics, Statistics, and Computer Science, University of Illinois at Chicago, 851 South Morgan (M/C 249), Chicago, IL 60607-7045, USA. *Email:* gyoffe2@uic.edu.

homotopy algorithm, while the communication between the nodes happens with message passing in a coarser grain. Granularity issues in homotopy algorithms are discussed in [3].

We first describe multithreaded versions of the most commonly used homotopy continuations in PHCpack. As expected from pleasingly parallel computations, the speedup of the multithreaded code is close to optimal. The second part of this paper is devoted to a “quality up” (defined in [2]) using multiprecision arithmetic instead of the standard hardware floats. Because of locks placed on memory allocations of the existing multiprecision of PHCpack, we have started to use the quad doubles as available in the software QD-2.3.9 [25]. A quad double is an unevaluated sum of four doubles and quadruples the working precision from $\epsilon = 2.204 \times 10^{-16}$ to $\epsilon^4 = 2.430 \times 10^{-63}$. Extending the precision in this manner was introduced in [17], see also [39] and [41].

Performing arithmetic with operations defined by software instead of hardware multiplies the cost with a certain factor. Experimentally we report in this paper that the multiplication factors in raising the level of precision from double to double double, and from double to quad double are close to the number eight, which is the number of available cores on our computer. Using all available cores properly, we can compensate for the extra cost of working with extended precision and thus obtain a quality up.

Our computations happened on a on a Mac OS X Pro with 2 Quad-Core Intel Xeons at 3.2 Ghz, bus speed at 1.6 Ghz, 12 Mb L2 cache per processor and 8Gb memory. We used the GNAT GPL 2009 edition of the GNU-Ada compiler. The code is available in version 2.3.56 of PHCpack. We also use QD-2.3.9 via its C interface, compiled with the g++ compiler (gcc version 4.0.1), including the pthread library.

Acknowledgements. Preliminary versions of this work were presented by the first author at a session of ACA 2009 and at a minisymposium of the SIAM PP10 conference. The first author is grateful to the organizers of these meetings. We thank the reviewers for their comments.

2 Speeding Up Path Trackers

In this section we describe the use of threads to speedup three different types of homotopies: cheater [33] [37], polyhedral [26], and monodromy breakup [43]. First we outline the manager/worker paradigm applied to multithreading.

2.1 Job Scheduling Algorithms

When we use the MPI library, we typically apply the manager/worker paradigm. One manager node (or process) maintains a job queue. Worker nodes (or processes) send messages to request jobs. If the received job is not the termination signal, then after computing the job, the results of the job are sent back to the manager. This paradigm scales very well for thousands of nodes and for systems of millions of solutions.

On smaller multiprocessor multicore computers, one could of course also run MPI, but in that case one may lose one entire process to the manager. Because of the irregularities of the path tracking jobs – it is hard to predict in advance how much work one job will take – the manager must always be available to serve the worker processes with a new job. Therefore one of the

available cores must be assigned to the manager and is therefore unavailable for computational jobs. On a system with 8 cores, this leaves only 7/8 or at most 87.5% of the total capacity available for computation.

The multithreaded version of the manager/worker paradigm uses one main thread to launch the worker threads. The main thread initializes the job queue but leaves the management of the pointer to the current job to the worker threads. If a worker needs a new job, it will request a lock (or semaphore) and adjust the pointer to the current job in a critical section. After adjusting the pointer, the lock is released [13]. Semaphores are supported in the GNU-Ada compiler [1].

Starting worker tasks in Ada is relatively simple and illustrated with code in Figure 1. The procedure `Workers` calls the `Job` procedure, which executes code based on the `id` number.

```

procedure Workers ( n : in natural ) is
  task type Worker ( id,n : natural );
  task body Worker is
  begin
    Job(id,n);
  end Worker;
  procedure Launch_Workers ( i,n : in natural ) is
    w : Worker(i,n);
  begin
    if i < n
    then Launch_Workers(i+1,n);
    end if;
  end Launch_Workers;
begin
  Launch_Workers(1,n);
end Workers;

```

Figure 1: Launching n workers. Each worker is an Ada task and calls the procedure `Job` with its identification number `id`.

The GNU-Ada compiler maps the Ada tasks to kernel threads. The implementation of the task scheduler is described in [40]. While our development platform is a Mac Pro with 2 Quad-Core processors, our multicore implementations have also been tested on dual core computers running Linux and Windows (win32 thread model). Also there – using the same source code – we observed good speedups.

2.2 Multitasking Polynomial Continuation

Table 1 lists preliminary timings on tracking all 924 paths in a cheater homotopy for the cyclic 7-roots problem [4]. Although the system overhead increases as we go from one to 8 threads, it constitutes only about 1% of the total wall clock time. The timings were obtained using version 2.3.45 of PHCpack, typing at the command prompt `$`

```
$ time phc -p -t8 < /tmp/input8
```

where the 8 was replaced subsequently by 4, 2, and 1.

#workers	real	user	sys	speedup
1	15.478s	15.457s	0.010s	1
2	7.790s	15.483s	0.010s	1.987
4	3.926s	15.445s	0.011s	3.942
8	1.992s	15.424s	0.015s	7.770

Table 1: Real, user, and system time in seconds (s) for tracking 924 paths of the cyclic 7-roots system using 1, 2, 4, and 8 worker threads. The speedup is the real time for one worker thread divided by the real time for 2, 4, and 8 workers.

Since version 2.3.46 of PHCpack, the blackbox solver called as `phc -b -t8` will use 8 tasks to run the polyhedral homotopies [26]. To run MPI programs, the environment of the user must be configured for `mpirun` or `mpiexec`. In contrast, using threads is done just by adding the `-t8` option to the `phc` executable.

2.3 Multitasking Polyhedral Continuation

To approximate all isolated solutions of a given system $f(\mathbf{x}) = \mathbf{0}$ with as many equations as unknowns, using polyhedral homotopies [26] we distinguish three stages:

1. Compute the mixed volume MV (aka the BKK bound) of the Newton polytopes spanned by the supports A of f via a regular mixed-cell configuration Δ_ω .
2. Given Δ_ω , solve a generic system $g(\mathbf{x}) = \mathbf{0}$, using polyhedral homotopies [26]. Every cell $C \in \Delta_\omega$ defines one homotopy

$$h_C(\mathbf{x}, s) = \sum_{\mathbf{a} \in C} c_{\mathbf{a}} \mathbf{x}^{\mathbf{a}} + \sum_{\mathbf{a} \in A \setminus C} c_{\mathbf{a}} \mathbf{x}^{\mathbf{a}} s^{\nu_{\mathbf{a}}}, \quad \nu_{\mathbf{a}} > 0, \quad (1)$$

tracking as many paths as the mixed volume of the cell C , as s goes from 0 to 1.

3. Use $(1 - t)g(\mathbf{x}) + tf(\mathbf{x}) = \mathbf{0}$ to solve $f(\mathbf{x}) = \mathbf{0}$.

Stages 2 and 3 are computationally most intensive. For example, for cyclic 10-roots ($MV = 35940$): stage 1 takes 21 seconds (using the version of MixedVol [22] as integrated in PHCpack) whereas stage 2 lasts 39 minutes.

We point out that PHoMpara [24] provides a parallel computation of the mixed volume.

A static distribution of the workload (as used in `mpi2cell.s` developed with Yan Zhuang [48]) is shown in Figure 2.

In a static workload distribution, we assume that every path takes about the same amount of work. Because polyhedral homotopies solve a generic system $g(\mathbf{x}) = \mathbf{0}$, this assumption may hold, although we experienced better performance with a dynamic load balancing [48]. Table 2 shows timings for running polyhedral homotopies on a random coefficient system, distributing mixed cells, for the cyclic n -roots problems.

manager	worker 1	worker 2	worker 3
$V(c_1) = 5$	$\#p(c_1) : 5$		
$V(c_2) = 4$	$\#p(c_2) : 4$		
$V(c_3) = 4$	$\#p(c_3) : 4$		
$V(c_4) = 6$	$\#p(c_4) : 1$	$\#p(c_4) : 5$	
$V(c_5) = 7$		$\#p(c_5) : 7$	
$V(c_6) = 3$		$\#p(c_6) : 2$	$\#p(c_6) : 1$
$V(c_7) = 4$			$\#p(c_7) : 4$
$V(c_8) = 8$			$\#p(c_8) : 8$
$\#p : 41$	$\#p : 14$	$\#p : 14$	$\#p : 13$

Figure 2: Static workload distribution of polyhedral continuation, for 8 cells $c_1, c_2, \dots, c_8$, where $V()$ is the mixed-volume function and $\#p$ stands for the number of paths.

n	MV	#tasks, times in seconds			
		1	2	4	8
7	924	12	6	3	2
8	2560	58	29	15	8
9	11016	417	209	104	52
10	35940	2156	1068	534	270

Table 2: Running polyhedral homotopies on cyclic n -roots, with 1, 2, 4, and 8 tasks, tracking a total of MV (mixed volume) many paths.

On the same random coefficient system $g(\mathbf{x}) = \mathbf{0}$ used to solve the cyclic 10-roots problem, we compared the MPI implementation with the new multitasked code, on our 8-core Mac Pro:

- `mpirun -n 9 mpi2cell_d`: total wall time = 270.5 seconds,
- `phc -m -t8`: elapsed wall clock time is 233 seconds.

Both implementations use the same tolerances and the same numerical settings for the parameters.

In defense of the implementation using MPI, one must point out the differences in managing the list of solutions. Because the MPI implementation was set up to work for millions of solutions, the complete list of solutions is not kept in memory: start solutions are read from a file or computed when needed and after tracking a path, the end solution is also directly written to a file. This jumpstarting mechanism is described in greater detail in [31].

Rather than advocating for the exclusive use of either MPI or threads, MPI and threads should be combined in a multi-tiered implementation on supercomputers with multicore nodes.

2.4 Multitasking Monodromy Breakup

Many polynomial system arising in practical applications have positive dimensional solution sets. Keeping to our running examples, the cyclic 8-roots problem [11] has a curve of degree 144.

This curve factors into 16 irreducible components, 8 factors have degree 16 and the other 8 are quadrics.

The application of monodromy to factor polynomials first appeared in [6], mainly to derive a complexity result. Actual computations were described in [15, 16] and [43]. See [14] for a nice introduction, [20], [21], [38] for recent developments, and to [28] for an application.

In Table 3 we show the outcome of running the real, user, and system timing on running ten monodromy loops. The fluctuations in the times are due to taking different random slices between the runs.

#workers	real	user	sys	speedup
1	61.203s	61.137s	0.046s	1
2	30.179s	59.888s	0.044s	2.028
4	15.472s	60.551s	0.050s	3.958
8	7.874s	59.469s	0.070s	7.773

Table 3: Real, user, and system time in seconds (s) for running 10 random loops, tracking a total of 20×144 paths to factor the solution curve of the cyclic 8-roots system, using 1, 2, 4, and 8 worker threads. The speedup is the real time for one worker thread divided by the real time for 2, 4, and 8 workers.

Using a basic parallel implementation as in [29], the experiments in Table 3 show a very good speedup. For larger number of workers, the overhead caused by scheduling and managing the irreducible decomposition leads to performance losses, remedied by the scheduling algorithms in [30].

Although we have no formal complexity bounds on the number of loops in the monodromy breakup algorithm seems to scale very well. One limiting factor — relevant for the second part of this paper — is that factoring polynomials of high degrees requires multiprecision arithmetic.

2.5 Granularity Issues

All parallel computations described in this section are coarse grained. We have many more paths to track than the available cores. Every task is in charge of multiple paths and every path can be tracked independently.

So far we obtained nice speedups with relatively little effort on multicore workstations. In addition, because the number of cores is limited and the jobs remain pleasingly parallel, there is no significant job scheduling overhead. If we can compute benchmark problems faster, then we can of course claim that we can solve more problems, but can we improve the quality of the solutions as well? Questions like these are addressed in [2] which defines “quality up”.

Shifting our attention from speedup to quality up, what if we have to track, instead of many solution paths, only a few, or even only one path? Often it occurs that there are a couple of solution paths that require extra care, for which the use multiprecision arithmetic is necessary [7]. The paper [5] surveys applications of high-precision computations, The use of double doubles in numerical linear algebra is considered in [35]. As software driven arithmetic is more expensive than hardware arithmetic, we want to offset this expense using multiple cores.

3 Quality Up

In this section we examine quality up: given 8 cores and roughly the same amount of time, can we increase the quality of our computations? Applied to polynomial system solving, we interpret quality as accuracy. We investigate the cost of obtaining more accurate solutions using Newton’s method in quad double complex arithmetic. Experiments in this section were done with an Ada translation of QD-2.3.9 [25], available in PHCpack since version 2.3.55.

3.1 Cost Overhead of Arithmetic

Complex arithmetic is standard practice in all homotopy solvers for polynomial systems and the overhead compared to real arithmetic is taken for granted. Since Ada 95, complex arithmetic is part of the language. However, PHCpack has its own packages for complex arithmetic and does *not* use the arithmetic provided by the language. The complex arithmetic in PHCpack happens via a generic package (generic is the equivalent to template in C++), suitable to work over any real number field. The experiment below applies the same code for all complex operations, over hardware doubles, double doubles, and quad doubles.

Fully optimized code on one core of a 3.2 Ghz Intel Xeon, performed one thousand times the following steps:

1. generate a 100-by-100 random matrix A and corresponding right hand side random vector $\mathbf{b}$;
2. solve $A\mathbf{x} = \mathbf{b}$ via LU factorization and substitutions $L\mathbf{y} = \mathbf{b}$, $U\mathbf{x} = \mathbf{y}$;
3. print $\|\mathbf{b} - A\mathbf{x}\|_\infty$ with 3 decimal places.

User CPU times are recorded in Table 4.

type of arithmetic	user CPU seconds
double real	2.026s
double complex	16.042s
double double real	20.192s
double double complex	140.352s
quad double real	173.769s
quad double complex	1281.934s

Table 4: CPU user time for solving a random linear system one thousand times, using double real and complex, double double real and complex, quad double real and complex arithmetic on one core.

Even as we take complex arithmetic for granted, we first measure the overhead factor as $16.042/2.026 = 7.918$, $140.352/20.192 = 6.951$, and $1281.934/173.769 = 7.377$. The three factors average to 7.415.

Going from double complex to double double complex gives a factor of $140.352/16.042 = 8.749$ and the multiplication factor in the cost of quad double complex over double double complex arithmetic is $1281.934/140.352 = 9.134$.

Similar factors to experiment the slowdown in going from standard double to software driven double doubles are reported in [35].

Although fluctuations in these computational experiments happen, we observe that the multiplication factors in the cost of using double double and quad double are of the same magnitude as the multiplication factor in the cost of complex arithmetic. In addition, as we have 8 cores at our disposal, we could offset the extra cost of more accurate arithmetic mapping threads of execution to all available 8 cores.

3.2 Newton’s Method with QD

To illustrate quality up, we consider a pleasingly parallel computation. Given a polynomial system and a list of isolated solution, we apply Newton’s method till either a specified tolerance on the size of the residual is achieved, or till the maximum number of iterations is exhausted.

As an example system, we take the cyclic 10-roots system. Because of symmetry, we refine only the generating solutions, starting at 1747 solutions accurate up to double precision. With double double complex arithmetic we set the tolerance to $1.0\text{E-}30$ and we allow at most 3 iterations of Newton’s method. With quad double complex arithmetic, the tolerance is set to $1.0\text{E-}60$ and we allow at most 5 iterations. The results are shown in Table 5.

double double complex				
#workers	real	user	sys	speedup
1	4.818s	4.790s	0.015s	1
2	2.493s	4.781s	0.013s	1.933
4	1.338s	4.783s	0.015s	3.601
8	0.764s	4.785s	0.016s	6.306
quad double complex				
#workers	real	user	sys	speedup
1	58.593s	58.542s	0.037s	1
2	29.709s	58.548s	0.054s	1.972
4	15.249s	58.508s	0.053s	3.842
8	8.076s	58.557s	0.058s	7.255

Table 5: Real, user, and system time in seconds (s) for refining 1747 solutions with Newton’s method in double double and quad double complex arithmetic, using 1, 2, 4, and 8 threads.

Entries in speedup columns of Table 5 are the real time on the same row divided by the real time for one worker. We observe an improved speedup as we move from double double to quad double.

Comparing the cost of quad double over double double arithmetic, we compute the factor $58.593/4.818 = 12.161$. As 12.161 is larger than the multiplication factor of 9.134 (based on the data in Table 4) we point out that to achieve an accuracy of $1.0\text{E-}60$, typically one extra

iteration of Newton’s method will be required, explaining the higher multiplication factor.

Considering quality up, we compare the first real time of Table 5 with the last real time: 4.818s with 8.076s. We conclude: if we can afford to wait twice as long, we can double the accuracy of our solutions from double double to quad double precision. As the cost overhead of using quad double complex arithmetic is compensated by the 8 cores, we have achieved quality up.

3.3 Granularity and Memory

Although threads were used in the previous section, the granularity is coarse, as the 1747 solutions were mapped to the threads (8 at most). Because the layout of quad doubles is restricted in size, one quad double is regarded as four local variables of type double and there is no memory allocation and deallocation of arrays of variable size like in general types of multiprecision. Memory (de)allocations impose locks on threads, preventing speedups.

4 Tracking one Path Faster and more accurately

In this section we focus on tracking one path with threads using double double or quad double complex arithmetic.

4.1 Multitasking Newton’s Method

The computational effort to execute one step of Newton’s method can be broken up into two main parts:

1. evaluate the system and the Jacobian matrix;
2. solve a linear system to update the solution.

There is more to path tracking than Newton’s method, although Newton’s method is the most computationally intensive ingredient. Complexity questions often reduce to estimating bounds on the domains of quadratic convergence [12]. A report on practical work of certified homotopies appears in [9]. Parallel algorithms for linear algebra are explained in [10] and [18].

Sparse and low degree polynomials can be evaluated fast and the cost of solving the linear system will dominate the total cost of Newton’s method. The situation may be reversed for large degree polynomials. From the perspective of achieving good speedups, we better assume we are dealing with either dense and/or large degree polynomial systems.

A detailed study of the efficient evaluation of polynomials for homotopy methods appeared in [27].

4.2 Polynomial System Evaluation

If we store all monomials of a polynomial system in one vector and its coefficients in a matrix (with proper correspondence to the monomial vector), then polynomial system evaluation is

turned into a matrix-vector product. While effective for dense polynomials (all monomials up to a certain degree appear with nonzero coefficient), sparse polynomials will generate sparse coefficient matrices. Our running example starts with 30 polynomials, each with 30 monomials with nonzero random complex coefficients in 30 variables. Along with the Jacobian matrix, this leads to a system of 930 polynomials with a total of 11540 distinct monomials.

We represent a sparse polynomial

$$f(\mathbf{x}) = \sum_{\mathbf{a} \in A} c_{\mathbf{a}} \mathbf{x}^{\mathbf{a}}, \quad c_{\mathbf{a}} \in \mathbb{C} \setminus \{0\}, \quad \mathbf{x}^{\mathbf{a}} = x_1^{a_1} x_2^{a_2} \cdots x_n^{a_n}, \quad (2)$$

collecting the exponents in the support A in a matrix E , as

$$F(\mathbf{x}) = \sum_{i=1}^m c_i \mathbf{x}^{E[k_i, :]}, \quad c_i = c_{\mathbf{a}}, \quad \mathbf{a} = E[k_i, :] \quad (3)$$

where k is an m -vector linking exponents to rows in E : $E[k_i, :]$ denotes all elements on the k_i th row of E . Storing all values of the monomials in a vector V , evaluating F (and f) is equivalent to making an inner product:

$$F(\mathbf{x}) = \sum_{i=1}^m c_i V_{k_i}, \quad V = \mathbf{x}^E. \quad (4)$$

Because we consider also all derivatives of all polynomials, we could exploit relations between the monomials and put evaluated powers in cache. In our first parallel evaluation algorithm, all monomials are evaluated independently, potentially by different tasks. In our running example, evaluating 11540 monomials of degree 30 requires about 346200 multiplications, more than ten times the inner products of the 930 coefficient vectors with the corresponding 30 values of the monomials. Since the evaluation of the monomials dominates the entire calculation, in our first parallel evaluation algorithm, we do not interlace the computation of the inner products with the evaluation of the monomials.

If p threads (or tasks) are labeled as $0, 1, \dots, p-1$, then the i th entry in the monomial vector is computed by the thread t for which $i \bmod p = t$.

In Table 6, we summarize the computational results.

Our first observation in Table 6 is that the system time grows as the number of tasks increase. Threads were created anew and destroyed twice for every evaluation as the evaluation of the monomial vectors was done separately from the computation of the inner products. While for double double complex arithmetic, the speedup is not so good, it is acceptable for quad double arithmetic.

Concerning quality up, we compare the sequential time of 69.536 seconds (first real time in Table 6) for double doubles with the 81.220 seconds (last real time) for quad double computations using 8 cores. Without parallel algorithms, the doubling of the accuracy takes an eightfold increase of computing time, as $559.085/69.536 = 8.040$. With 8 cores, the cost of extra accuracy is reduced to an increase of 17% of the real time, as $81.220/69.536 = 1.168$.

The system times of Table 6 are cut in half if we do not destroy and create threads between the evaluation of the monomial vector and the multiplication with the coefficient vectors. Rather than giving updated tables of timings, we describe how we avoid thread destruction and creation.

double double complex						
#tasks	real		user		sys	speedup
1	1m	9.536s	1m	9.359s	0.252s	1
2	0m	37.691s	1m	10.126s	0.417s	1.845
4	0m	21.634s	1m	10.466s	0.753s	3.214
8	0m	14.930s	1m	12.120s	1.711s	4.657

quad double complex						
#tasks	real		user		sys	speedup
1	9m	19.085s	9m	18.552s	0.563s	1
2	4m	43.005s	9m	19.402s	0.679s	1.976
4	2m	24.669s	9m	20.635s	1.023s	3.865
8	1m	21.220s	9m	26.120s	2.809s	6.884

Table 6: Real, user, and system time in minutes (m) and seconds (s) for evaluating a random system of 930 polynomials in 30 variables (at most 30 monomials of degree at most 30 in each polynomial), with double double and quad double complex arithmetic, at one thousand randomly generated points, using 1, 2, 4, and 8 threads.

To synchronize jobs performed by p threads we maintain p flags b_i of boolean values, for $i = 0, 1, \dots, p-1$. The i th flag b_i is true if and only if the i th thread is busy with a computational job. Before starting the next job, threads must wait till all threads are finished with their current job. The first thread manages the job counter k . When thread i finishes its job k , it sets its flag b_i to false and then waits to continue to the next job till k is increased. The first thread will increase k only when no jobs are busy. After increasing k , the first thread sets all busy flags to true.

4.3 Multithreaded Linear Algebra with real Quad Doubles

In this section we consider *real* quad doubles and work with the original QD library. Runs are done on the same computer, but the software environment is different. We use QD-2.3.9 via its C interface, compiled with the g++ compiler (gcc version 4.0.1), including the pthread library. Optimization flags were left off.

The operations we consider are (1) matrix-vector product; (2) solving a triangular linear system (back substitution); and (3) Gaussian elimination.

The setup for the matrix-vector product is as follows. We launch a number of threads and then divide the work to do *one* matrix-vector product among the threads. Because doing only one product goes too fast, we run multiple instances of multiple threaded matrix-vector products. As the matrix-vector products simulate polynomial evaluations, during the tracking of one solution path in our simulation we assume that one thousand evaluations of the system and all its partial derivatives are performed.

In Table 7 we report on timings with a multithreaded implementation of a matrix-vector product. The threads are created only once, when performing multiple jobs by many threads, but they wait on each other before moving on to the next job. While not optimal, the speedup

#threads	real	user	sys	speedup
1	28.990s	28.444s	0.022s	1
2	14.598s	28.205s	0.021s	1.986
4	7.537s	28.197s	0.017s	3.846
8	4.231s	28.394s	0.026s	6.852

Table 7: Real, user, and system time in seconds (s) to multiply a 700-by-200 matrix of quad doubles with a vector 1000 times, using 1, 2, 4, and 8 threads.

attained in Table 7 is still acceptable. In Table 8, we consider matrices of smaller sizes. We observe decreasing speedups as the dimensions of the matrices decrease.

100-by-100 matrix				
#threads	real	user	sys	speedup
1	2.031s	2.028s	0.003s	1
2	1.020s	2.030s	0.004s	1.991
4	0.514s	2.031s	0.004s	3.951
8	0.272s	2.129s	0.005s	7.467

50-by-50 matrix				
#threads	real	user	sys	speedup
1	0.510s	0.508s	0.002s	1
2	0.257s	0.508s	0.003s	1.984
4	0.137s	0.532s	0.003s	3.723
8	0.078s	0.590s	0.004s	6.538

20-by-20 matrix				
#threads	real	user	sys	speedup
1	0.085s	0.083s	0.002s	1
2	0.045s	0.085s	0.002s	1.889
4	0.025s	0.086s	0.002s	3.340
8	0.016s	0.104s	0.003s	5.313

Table 8: Real, user, and system time in seconds (s) for 1000 matrix-vector multiplications with matrices of quad doubles, using 1, 2, 4, and 8 threads.

The sequel to Table 7 in our simulated path tracking with multiple threads is the solution of a triangular linear system, with times reported in Table 9. As the linear system is already in triangular form, it can be solved via back substitutions. Our parallel implementation of the algorithm for solving a triangular system is inspired by [50, §5.3.4].

Let a lower triangular n -by- n matrix L with entries $\ell_{i,j}$ and an n -vector $\mathbf{b}$ define the system

#threads	real	user	sys	speedup
1	41.877s	41.770s	0.018s	1
2	21.365s	42.631s	0.026s	1.960
4	10.986s	43.376s	0.026s	3.812
8	5.984s	47.097s	0.030s	6.998

Table 9: Real, user, and system time in seconds (s) for 10,000 on solving a 200-by-200 triangular matrix of quad doubles, using 1, 2, 4, and 8 threads.

$L\mathbf{x} = \mathbf{b}$. The solution vector $\mathbf{x}$ is computed via the formulas

$$x_i := \frac{1}{\ell_{i,i}} \left(b_i - \sum_{j=1}^{i-1} \ell_{i,j} x_j \right), \quad i = 1, 2, \dots, n. \quad (5)$$

The calculation of x_i needs the values for all previous components x_j , for $j = 1, 2, \dots, i-1$.

Labeling p threads by $0, 1, \dots, p-1$, the i th thread computes first the value x_i and then computes successively all values of x_{i+jp} for all j : $i+jp \leq n-1$. Starting to calculate x_{i+jp} , the i th thread does not wait until all values of x_k , for all k : $k < i+jp$ are computed. At first, without waiting, the i th thread computes the partial sum

$$\sum_{j=1}^{m-1} \ell_{i+jp,j} x_j \quad \text{of} \quad \sum_{j=1}^{i+jp-1} \ell_{i+jp,j} x_j, \quad (6)$$

where m is the number of x_k values computed by the time it starts calculation of x_{i+jp} . Then the thread merely proceeds with the computing the remaining part of the sum for x_{i+jp} as long as each next x_k appearing in it is computed.

To synchronize the calculations, we keep an array of status flags associated to the variables. The status flag of a variable is updated by the processor which computes that particular variable after the calculation of the variable is complete. Other threads which need the value of the variable must wait till the status flag of the variable has been updated.

Adjusting the algorithm described above to solve upper triangular linear systems happens by reversing the indices of $\mathbf{x}$, i.e.: $x_i = x_{n-i}$, for $i = 1, 2, \dots, n$.

Larger values of n/p lead to larger speedups, since then for large indexes $i+jp$ by the time the i th thread starts computing x_{i+jp} almost all x_k with $k < i+jp$ are already computed, thus the computations of x_{i+jp} with large indexes are done almost uninterruptedly with relatively very few short breaks for the i th thread to wait until several preceding values of x_k to x_{i+jp} are calculated.

The analogue to Table 8 for back substitutions is Table 10. In Table 10 we display times for back substitutions of various dimensions. As before, we observe decreasing speedups in Table 10 as the dimensions of the matrices decrease. However, it seems that the threshold dimension for achieving good speedups lies higher with multithreaded back substitution than with matrix-vector products. Note that path tracking typically uses complex arithmetic, whereas the computations with quad double are fully real: the threshold dimension for good speedups is lower for complex arithmetic.

150-by-150 matrix				
#threads	real	user	sys	speedup
1	23.661s	23.650s	0.009s	1
2	12.167s	24.311s	0.014s	1.945
4	6.370s	24.974s	0.014s	3.714
8	3.530s	27.590s	0.022s	6.703

100-by-100 matrix				
#threads	real	user	sys	speedup
1	10.752s	10.743s	0.009s	1
2	5.588s	11.069s	0.007s	1.924
4	2.868s	11.451s	0.007s	3.749
8	1.866s	14.624s	0.012s	5.762

50-by-50 matrix				
#threads	real	user	sys	speedup
1	2.802s	2.799s	0.003s	1
2	1.487s	2.963s	0.006s	1.884
4	0.859s	3.420s	0.004s	3.262
8	0.640s	5.088s	0.006s	4.378

20-by-20 matrix				
#threads	real	user	sys	speedup
1	0.510s	0.509s	0.002s	1
2	0.310s	0.614s	0.002s	1.645
4	0.220s	0.866s	0.003s	2.318
8	0.264s	2.066s	0.005s	1.932

Table 10: Real, user, and system time in seconds (s) for solving 10,000 triangular systems of quad doubles, using 1, 2, 4, and 8 threads.

To solve a linear system $A\mathbf{x} = \mathbf{b}$, we apply row reduction on the augmented matrix $[A \ \mathbf{b}]$. Denoting the entries of A by $a_{i,j}$, formulas using pivot row i , for i ranging from 1 to $n - 1$ are

$$a_{j,k} := a_{j,k} - \frac{a_{j,i}}{a_{i,i}}a_{i,k}, \quad k = j, j + 1, \dots, n, \quad (7)$$

and on $\mathbf{b}$: $b_j := b_j - (a_{j,i}/a_{i,i})b_i$, for j ranging from i to n . Note that these formulas do not perform row interchanges (partial pivoting) to increase the numerical stability. Partial pivoting is done by the algorithms in the next section.

To achieve an equal workload among p threads, we assign rows to threads as follows: the first thread will work on rows $1, p+1, 2p+1, \dots$, the second thread will work on rows $2, p+2, 2p+2, \dots$, in general: the i th thread works on rows $i + pj$ for all natural values of j starting at 0 and increasing as long as $i + pj \leq n$. As the pivot row increases, the difference between workloads among the threads is never more than one.

100-by-100 matrix				
#threads	real	user	sys	speedup
1	71.464s	71.436s	0.023s	1
2	37.129s	74.222s	0.026s	1.925
4	18.903s	75.557s	0.031s	3.781
8	10.721s	84.655s	0.116s	6.666

50-by-50 matrix				
#threads	real	user	sys	speedup
1	9.354s	9.345s	0.009s	1
2	5.049s	10.086s	0.007s	1.853
4	2.583s	10.314s	0.008s	3.621
8	1.522s	12.001s	0.026s	6.146

20-by-20 matrix				
#threads	real	user	sys	speedup
1	0.655s	0.653s	0.002s	1
2	0.368s	0.731s	0.003s	1.780
4	0.219s	0.864s	0.003s	2.991
8	0.166s	1.290s	0.006s	3.946

Table 11: Real, user, and system time in seconds (s) for performing 1000 times Gaussian elimination, using 1, 2, 4, and 8 threads.

For correctness, threads are synchronized so no threads starts updating rows for the next pivot until all threads have finished updating their rows for the current pivot row. This synchronization is the same as described at the end of section 4.2.

Timings on our multithreaded code for Gaussian elimination are reported in Table 11. We observe a very good speedup for dimensions 50 and 100. Even with a small number of variables such as 20, the speedup is still acceptable. Compared to the timings for the back substitutions in Table 10 (where we had to do ten thousand runs), it is definitely worthwhile to use multiple threads for the row reduction stage of Newton’s method.

4.4 Multithreaded Linear Algebra with Complex Quad Doubles

The routines to solve linear systems in PHCpack are based on LINPACK [19], and in particular on ZGEFA and ZGESL to solve linear systems via an LU factorization.

In this section we report on experiments with a basic multithreaded version of our LU factorization routines. Following the same synchronization mechanism as described with polynomial evaluation, the first thread is in charge of updating the column counter. The first thread also takes care of the pivoting, i.e.: the selection of the largest element in the current column used as the denominator in formulas (7). The computational results are summarized in Table 12.

In Table 12 we observe a significant increase in the user and system time for 8 tasks. The

double double complex						
#tasks	real		user		sys	speedup
1	1m	8.173s	1m	8.074s	0.131s	1
2	0m	36.712s	1m	13.061s	0.249s	1.857
4	0m	21.565s	1m	25.035s	0.455s	3.161
8	0m	20.986s	1m	42.156s	2.270s	3.248

quad double complex						
#tasks	real		user		sys	speedup
1	10m	12.216s	10m	11.900s	0.311s	1
2	5m	12.753s	10m	24.774s	0.477s	1.958
4	2m	42.653s	10m	48.795s	0.699s	3.764
8	1m	33.234s	12m	17.653s	1.930s	6.566

Table 12: Real, user, and system time in minutes (m) and seconds (s) for computing one thousand times the LU factorization of an 80-by-80 matrix of random double double and quad double complex numbers, using 1, 2, 4, and 8 threads.

increase could be due to threads spending a significant amount of time in busy waiting loops, waiting for the pivoting.

Looking at speedups, we see that for double double complex arithmetic, results start to deteriorate once we go past four threads. The speedups are better with quad double complex arithmetic. Comparing the first real time with the last one in Table 12, we see that the expense of doubling the precision from double double to quad double is compensated by the 8 threads.

4.5 Newton’s Method with Threads

Comparing Tables 6 and 12 we see that a LU factorization of an 80-by-80 matrix takes only slightly more time than the evaluation of a sparse system in 30 variables with all its 900 partial derivatives. If we perform the operations for the same number of variables, e.g.: at 30, then the time for linear algebra shrinks significantly, or e.g.: at 80, then the time for polynomial evaluation will increase significantly. In either case, the time for polynomial evaluation dominates and leads already in relatively low number of variables to good speedups.

Because the cost of LU factorization and triangular linear system solvers is respectively $O(n^3)$ and $O(n^2)$, for n variables, and because the number of threads is typically much smaller than n , the cost of the LU factorization will remain dominant, even if we would not run the back substitution on multiple cores.

To avoid the overhead from thread destruction and creation, threads will stay alive for all iterations of Newton’s method along a path. The synchronization algorithm is the same as described above, with one thread managing a job counter. There is a second level of job counters to coordinate the stages inside a Newton iteration.

5 Conclusions

As expected, using threads for pleasingly parallel computations leads rather directly to good speedups. Because the number of cores remains limited on workstations, we do not encounter problems to scale the calculations to hundreds or thousands of processors.

The more convenient thread model offers the opportunity for quality up: can we compute the solutions more accurately with multiple cores in roughly the same amount of time? Arbitrary multiprecision arithmetic with dynamic memory allocation imposes locks, preventing speedups, so we turned to the quad double arithmetic implemented by the software library `QD-2.3.9`.

Experimental results on solving linear systems, showed that going from double complex to double double complex increased the computation time by a factor of about 8.7. Using quad double complex arithmetic over double double arithmetic multiplied the user CPU times by a factor of 9.1. These experimental factors are slightly above eight, the number of available cores on our workstation, so we may potentially offset these factors using all available cores.

Placing the focus on polynomial evaluation and linear system solving — the computational ingredients of Newton’s method — we experienced good speedups using quad double arithmetic. In particular, doubling the accuracy from double double to quad double can be done in less than double the time if we use all eight cores.

References

- [1] AdaCore. The GNAT Reference Manual. At http://gcc.gnu.org/onlinedocs/gnat_rm.
- [2] S.G. Akl. Superlinear performance in real-time parallel computation. *The Journal of Supercomputing*, 29(1):89–111, 2004.
- [3] D.C.S. Allison, A. Chakraborty, and L.T. Watson. Granularity issues for solving polynomial systems via globally convergent algorithms on a hypercube. *J. of Supercomputing*, 3:5–20, 1989.
- [4] J. Backelin and R. Fröberg. How we proved that there are exactly 924 cyclic 7-roots. In *Proceedings of the 1991 International Symposium on Symbolic and Algebraic Computation (ISSAC’91)*, pages 101–111. ACM, 1991.
- [5] D.H. Bailey, R. Barrio, and J.H. Borwein. High precision computation: Mathematical physics and dynamics. Joint SIAM-RSME-SCM-SEMA Meeting on Emerging Topics in Dynamical Systems and Partial Differential Equations (DSPDEs’10), 31 May 2010, Barcelona, Spain.
- [6] C. Bajaj, J. Canny, T. Garritty, and J. Warren. Factoring rational polynomials over the complex numbers. *SIAM J. Comput.*, 22(2):318–331, 1993.
- [7] D. J. Bates, J.D. Hauenstein, A.J. Sommese, and C.W. Wampler. Adaptive multiprecision path tracking. *SIAM J. Numer. Anal.*, 46(2):722–746, 2008.
- [8] D.J. Bates, J.D. Hauenstein, A.J. Sommese, and C.W. Wampler. Bertini: Software for numerical algebraic geometry. Available at <http://www.nd.edu/~sommese/bertini/>.

- [9] C. Beltran and Leykin. A. Certified numerical homotopy tracking. Preprint [arXiv:0912.0920v1 \[math.NA\]](#).
- [10] R.H. Bisseling. *Parallel Scientific Computation. A structured approach using BSP and MPI*. Oxford University Press, 2004.
- [11] G. Björck and R. Fröberg. Methods to “divide out” certain solutions from systems of algebraic equations, applied to find all cyclic 8-roots. In M. Gyllenberg and L.E. Persson, editors, *Analysis, Algebra and Computers in Math. research*, volume 564 of *Lecture Notes in Mathematics*, pages 57–70. Dekker, 1994.
- [12] L. Blum, F. Cucker, M. Shub, and S. Smale. *Complexity and Real Computation*. Springer-Verlag, 1998.
- [13] A. Burns and A. Wellings. *Concurrent and Real-Time Programming in Ada*. Cambridge University Press, 2007.
- [14] G. Chèze and A. Galligo. Four lectures on polynomial absolute factorization. In *Solving Polynomial Equations. Foundations, Algorithms and Applications*, volume 14 of *Algorithms and Computation in Mathematics*, pages 339–394. Springer-Verlag, 2005.
- [15] R.M. Corless, A. Galligo, I.S. Kotsireas, and S.M. Watt. A geometric-numeric algorithm for factoring multivariate polynomials. In T. Mora, editor, *Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation (ISSAC 2002)*, pages 37–45. ACM, 2002.
- [16] R.M. Corless, M.W. Giesbrecht, M. van Hoeij, I.S. Kotsireas, and S.M. Watt. Towards factoring bivariate approximate polynomials. In B. Mourrain, editor, *Proceedings of the 2001 International Symposium on Symbolic and Algebraic Computation (ISSAC 2001)*, pages 85–92. ACM, 2001.
- [17] T.J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971.
- [18] J.J. Dongarra, I.S. Duff, D.C. Sorensen, and H.A. van der Vorst. *Numerical Linear Algebra for High-Performance Computers*. SIAM, 1998.
- [19] J.J. Dongarra, C.B. Moler, J.R. Bunch, and G.W. Stewart. *LINPACK Users’ Guide*. SIAM, 1979.
- [20] A. Galligo and A. Poteaux. Continuations and monodromy on random Riemann surfaces. In H. Kai and H. Sekigawa, editors, *SNC’09: Proceedings of the 2009 conference on Symbolic-Numeric Computation*, pages 115–124. ACM, 2009.
- [21] A. Galligo and M. van Hoeij. Approximate bivariate factorization: a geometric viewpoint. In J. Verschelde and S.M. Watt, editors, *SNC’07: Proceedings of the 2007 international workshop on Symbolic-Numeric Computation*, pages 1–10. ACM, 2007.
- [22] T. Gao, T. Y. Li, and M. Wu. Algorithm 846: MixedVol: a software package for mixed-volume computation. *ACM Trans. Math. Softw.*, 31(4):555–560, 2005.

- [23] Y. Guan and J. Verschelde. Parallel implementation of a subsystem-by-subsystem solver. In *Proceedings of the 22th High Performance Computing Symposium, Quebec City, 9-11 June 2008*, pages 117–123. IEEE Computer Society, 2008.
- [24] T. Gunji, S. Kim, K. Fujisawa, and M. Kojima. PHoMpara – parallel implementation of the Polyhedral Homotopy continuation Method for polynomial systems. *Computing*, 77(4):387–411, 2006.
- [25] 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>.
- [26] B. Huber and B. Sturmfels. A polyhedral method for solving sparse polynomial systems. *Math. Comp.*, 64(212):1541–1555, 1995.
- [27] M. Kojima. Efficient evaluation of polynomials and their partial derivatives in homotopy continuation methods. *Journal of the Operations Research Society of Japan*, 51(1):29–54, 2008.
- [28] A. Leykin and F. Sottile. Galois groups of Schubert problems via homotopy continuation. *Mathematics of Computation*, 78(267):1749–1765, 2009.
- [29] A. Leykin and J. Verschelde. Factoring solution sets of polynomial systems in parallel. In T. Skeie and C.-S. Yang, editors, *Proceedings of the 2005 International Conference on Parallel Processing Workshops. 14-17 June 2005. Oslo, Norway. High Performance Scientific and Engineering Computing*, pages 173–180. IEEE Computer Society, 2005.
- [30] A. Leykin and J. Verschelde. Decomposing solution sets of polynomial systems: a new parallel monodromy breakup algorithm. *The International Journal of Computational Science and Engineering*, 4(2):94–101, 2009.
- [31] A. Leykin, J. Verschelde, and Y. Zhuang. Parallel homotopy algorithms to solve polynomial systems. In N. Takayama and A. Iglesias, editors, *Proceedings of ICMS 2006*, volume 4151 of *Lecture Notes in Computer Science*, pages 225–234. Springer-Verlag, 2006.
- [32] T.Y. Li. Numerical solution of polynomial systems by homotopy continuation methods. In F. Cucker, editor, *Handbook of Numerical Analysis. Volume XI. Special Volume: Foundations of Computational Mathematics*, pages 209–304. North-Holland, 2003.
- [33] T.Y. Li, T. Sauer, and J.A. Yorke. The cheater’s homotopy: an efficient procedure for solving systems of polynomial equations. *SIAM J. Numer. Anal.*, 26(5):1241–1251, 1989.
- [34] T.Y. Li and C.-H. Tsai. HOM4PS-2.0para: Parallelization of HOM4PS-2.0 for solving polynomial systems. *Parallel Computing*, 35(4):226–238, 2009.
- [35] X. Li, J. Demmel, D. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Kang, A. Kapur, M. Martin, B. Thompson, T. Tung, and Yoo D. Design, implementation and testing of

- extended and mixed precision BLAS. *ACM Trans. Math. Softw.*, 28(2):152–205, 2002. This is a shortened version of Technical Report LBNL-45991.
- [36] A. Morgan. *Solving polynomial systems using continuation for engineering and scientific problems*. Prentice-Hall, 1987. Volume 57 of Classics in Applied Mathematics Series, SIAM 2009.
  - [37] A.P. Morgan and A.J. Sommese. Coefficient-parameter polynomial continuation. *Appl. Math. Comput.*, 29(2):123–160, 1989. Errata: *Appl. Math. Comput.* 51:207(1992).
  - [38] A. Poteaux. Computing monodromy groups defined by plane curves. In J. Verschelde and S.M. Watt, editors, *SNC'07. Proceedings of the 2007 International Workshop on Symbolic-Numeric Computation*, pages 239–246. ACM, 2007.
  - [39] D.N. Priest. *On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations*. PhD thesis, University of California at Berkeley, 1992. <ftp://ftp.icsi.berkeley.edu/pub/theory/priest-thesis.ps.Z>.
  - [40] J.F. Ruiz. GNAT Pro for on-board mission-critical space applications. In T. Vardanega and A. Wellings, editors, *Reliable Software Technology – Ada-Europe 2005. 10th Ada-Europe International Conference on Reliable Software Technologies. York, UK, June 20-24, 2005. Proceedings*, volume 3555 of *Lecture Notes in Computer Science*, pages 248–259, 2005.
  - [41] J.R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete Comput. Geom.*, 18(3):305–363, 1997.
  - [42] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI - The Complete Reference Volume 1, The MPI Core*. Massachusetts Institute of Technology, second edition, 1998.
  - [43] A.J. Sommese, J. Verschelde, and C.W. Wampler. Using monodromy to decompose solution sets of polynomial systems into irreducible components. In C. Ciliberto, F. Hirzebruch, R. Miranda, and M. Teicher, editors, *Application of Algebraic Geometry to Coding Theory, Physics and Computation*, pages 297–315. Kluwer Academic Publishers, 2001. Proceedings of a NATO Conference, February 25 - March 1, 2001, Eilat, Israel.
  - [44] A.J. Sommese and C.W. Wampler. *The Numerical solution of systems of polynomials arising in engineering and science*. World Scientific, 2005.
  - [45] H.-J. Su, J.M. McCarthy, M. Sosonkina, and L.T. Watson. Algorithm 857: POLSYS_GLP: A parallel general linear product homotopy code for solving polynomial systems of equations. *ACM Trans. Math. Softw.*, 32(4):561–579, 2006.
  - [46] 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>.
  - [47] J. Verschelde and Y. Wang. Computing feedback laws for linear systems with a parallel Pieri homotopy. In Y. Yang, editor, *Proceedings of the 2004 International Conference on Parallel*

- Processing Workshops, 15-18 August 2004, Montreal, Quebec, Canada. High Performance Scientific and Engineering Computing*, pages 222–229. IEEE Computer Society, 2004.
- [48] J. Verschelde and Y. Zhuang. Parallel implementation of the polyhedral homotopy method. In T.M. Pinkston and F. Ozguner, editors, *Proceedings of the 2006 International Conference on Parallel Processing Workshops. 14-18 Augustus 2006. Columbus, Ohio. High Performance Scientific and Engineering Computing*, pages 481–488. IEEE Computer Society, 2006.
  - [49] Y. Wang. *Computing Dynamic Output Feedback Laws with Pieri Homotopies on a Parallel Computer*. PhD thesis, University of Illinois at Chicago, 2005.
  - [50] B. Wilkinson and M. Allen. *Parallel Programming. Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice Hall, 2nd edition, 2005.
  - [51] Y. Zhuang. *Parallel Implementation of Polyhedral Homotopy Methods*. PhD thesis, University of Illinois at Chicago, 2007.