Using Parallelism to Compensate for Extended Precision in Path Tracking for Polynomial System Solving

BY

GENADY YOFFE

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Mathematics
in the Graduate College of the
University of Illinois at Chicago, 2012

Chicago, Illinois

Defense Committee:

Jan Verschelde, Chair and Advisor
Shmuel Friedland
Anton Leykin, Georgia Institute of Technology
David Nicholls
Gyorgy Turan

TABLE OF CONTENTS

CHAPTER			PAGE
1	**INTRODUCTION**		1
	1.1	Problem Statement	1
	1.1.1	Context	1
	1.1.2	Thesis Main Goal	1
	1.2	Tool Selection: Multicore Processing, pthreads, and QD library	2
	1.2.1	Hardware/Interaction Model	2
	1.2.2	Multiprecision Library	3
	1.2.3	Programming Tools	4
	1.3	Contributions	5
	1.3.1	Cores Compensate for Cost of Multiprecision	5
	1.3.2	Software	10
	1.3.3	Towards Efficiency	10
	1.3.4	Graphics Processing Unit Prospects	14
2	**CONTINUING THREADS**		16
	2.1	Necessity	16
	2.2	Debugging Overhead	17
	2.3	Synchronization and Avoiding Mutexes	18
3	**MULTITASKING COMPONENTS OF NEWTON'S METHOD**		21
	3.1	MT Polynomial Evaluation	22
	3.1.1	Algorithm	22
	3.1.2	Parallelization	22
	3.2	MT Linear Systems Solver	25
	3.2.1	MT Gaussian Elimination	25
	3.2.2	MT Back Substitution	26
4	**MT NEWTON'S METHOD AND MT PATH TRACKER**		31
	4.1	MT Path Tracker	31
	4.1.1	Algorithm	31
	4.1.2	Computational Results and Testing Homotopies	33
	4.2	Multithreaded Newton's Method	35
5	**AD-LIKE POLYNOMIAL EVALUATION AND DIFFERENTIATION**		39
	5.1	Terminology and the Serial Algorithm	40
	5.2	Precomputing Powers	42

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

5.3 Multithreaded Computing Monomials and Monomial Derivatives 43
5.4 Homotopy Parameter Multiplications 45
5.5 Multithreaded Coefficient Product (CP) 46

6 TOWARDS FURTHER EFFICIENCY IN WORKING DIMENSIONS 48
6.1 Input Processing 48
6.2 Quadratic Predictor Versus Secant Predictor 50
6.3 Computational Results and Conclusions 53

7 GRAPHICS PROCESSING UNIT PROSPECTS 56
7.1 Monomial Prototype Calculation 56
7.2 Monomial Evaluation and Differentiation of Products of Variables 62
7.3 Summation of Terms 68
7.4 Computational Experiments 72

8 CONCLUSIONS 74

CITED LITERATURE 76

VITA 81

LIST OF FIGURES

FIGURE		PAGE
1	Threads synchronization illustration	19
2	Use of constant memory in the GPU polynomial evaluation	58

LIST OF TABLES

TABLE		PAGE
I	Effect Of Continuing Threads	16
II	Multithreaded Polynomial Evaluation Timings And Speedups . .	24
III	Multithreaded Gaussian Elimination Timings And Speedups . . .	29
IV	Multithreaded Back Substitution Timings And Speedups	30
V	Performance Of The Preliminary Version Of MT Path Tracker . .	34
VI	Illustration Of The Efficiency Of The Quadratic Predictor	53
VII	Timings And Speedups For More Efficient MT Path Tracker 1 . .	54
VIII	Timings And Speedups For More Efficient MT Path Tracker 2 . .	55
IX	Work Of The Second Kernel Of The GPU Polynomial Evaluation	64
X	Timings And Speedups For The GPU Polynomial Evaluation . . .	73

ACKNOWLEDGMENTS

I would like to express my deep gratitude to my adviser Jan Verschelde for introducing me to the area of polynomial homotopy continuation and parallel computing, for fruitful cooperation, and for encouraging and thoughtful supervision of my work.

I would like to thank my other thesis committee members Shmuel Friedland, Anton Leykin, David Nicholls, and Gyorgy Turan for valuable comments and stimulating questions on my work.

I am also very much grateful to the Applied Mathematics faculty of the MSCS department for developing a broad and interesting program, which helped me to prepare for conducting research in the field.

Different stages of my research were supported by the National Science Foundation under grants No. 0713018 and No. 1115777. I would like to thank NSF for this support.

LIST OF ABBREVIATIONS

AD	Automatic Differentiation
GPU	Graphics Processing Unit
CUDA	Compute Unified Device Architecture
MPI	Message Passing Interface
MT	Multithreaded (an attribute)
QD	Quad Double
DD	Double Double
PE	Polynomial Evaluation
ME	Monomial Evaluation
GE	Gaussian Elimination
BS	Back Substitution
STL	Standard Template Library
AMS	American Mathematical Society
UIC	University of Illinois at Chicago

SUMMARY

Employing extended precision in polynomial homotopy continuation prompts a substantial time overhead. First we have shown that for sparse polynomial systems of moderate sizes it is possible with multiple cores to compensate to a great extent for the additional cost of double double (DD) and of quad double (QD) precision in path tracking. That was done by obtaining a scalable multithreaded DD/QD version of Newton's method and subsequently of a path tracker. We achieved on eight cores for moderate systems good speedups in double double path tracking and close to maximal speedups in quad double path tracking.

Later we have greatly increased the efficiency of our multithreaded implementation in working dimensions by improvements in three different directions. First we suggested a new input processing procedure based on functionality of a Standard Template Library (STL) sorted associative container. Secondly we came up with a choice of much more suitable predictor. Thirdly we integrated into our multithreaded implementation a prudent algorithm for polynomial evaluation and differentiation, which is based on the ideas of reverse mode Automatic Differentiation (AD).

Recently we efficiently accelerated with general purpose graphics unit the same AD-like algorithm for polynomial evaluation and differentiation. We obtained two digit speedups for moderate systems as hardware double arithmetic is used.

CHAPTER 1

INTRODUCTION

1.1 Problem Statement

1.1.1 Context

Homotopy continuation methods have led to efficient numerical solvers of polynomial systems (see e.g. (1), (2), (3), (4)) and constitute an essential part in the emerging area of numerical algebraic geometry ((5), (6), (7)). If one is interested in speeding up the finding *all* isolated solutions of polynomial systems, which might become a computationally involved task for systems of already intermediate dimensions and degrees, then distributing homotopy solution path tracking jobs in a manager/worker paradigm using message passing (8) leads to very good speedups. Such parallel implementations are in `Bertini` (9), `HOM4PS-2.0para` (10), `PHoMpara` (11), `POLSYS_GLP` (12), and `PHCpack` (13), documented in (14), (15; 16), (17), (18), and (19).

1.1.2 Thesis Main Goal

At the same time for large polynomial systems in many variables and of high degrees the double precision in standard hardware is often insufficient to guarantee accurate results. When running many path tracking jobs, just one solution path may require extended multiprecision arithmetic. Our ultimate goal is to offset the extra cost of software driven arithmetic in such a situation by multitasking computationally intensive stages of tracking the harder to follow

solution path in question. In particular we would like to determine for which dimensions and degrees our goal could be achieved on existing parallel technologies. It seems appealing to refer to any realizing our purpose software product as to a *multitasked path tracker*. In this thesis we are concerned with developing a multitasked path tracker for *sparse polynomial systems*, at which even as monomials could appear to higher total degrees, only relatively few monomials appear with nonzero coefficients. Typically we consider systems with the number of monomials per polynomial $\Theta(n)$, where n is the dimension of the system. Sparse systems appear in the most applications. We refer to their antipodes, the systems at which almost all monomials have nonzero coefficients, as to *dense polynomial systems*. The class of polynomial homotopies, which helps naturally to solve sparse polynomial systems is the class of polyhedral homotopies described in (6).

1.2 Tool Selection: Multicore Processing, pthreads, and QD library

1.2.1 Hardware/Interaction Model

We have chosen shared memory multicore processing to be a formative framework for our parallel implementation. Most computers are multicore nowadays, thus for the majority of users it would be at least as feasible (or at most as expensive) to organize a run of an accordingly implemented multitasked path tracker on a multicore workstation compared to running an accordingly implemented multitasked path tracker on a computer cluster.

Also a parallel technology, with communicating among its computing units by message passing, would not constitute a better alternative tool to implement for running on it a multitasked path tracker for systems of moderate dimensions. Continuation methods apply Newton's

method as corrector in predictor-corrector algorithms to track paths of solutions defined by a homotopy. The computationally intensive stages of Newton's iteration are polynomial evaluation and differentiation and solving a linear system of equations. The nature of linear system solving is so, that while working with moderate dimensions, whatever parallel implementation is used for its realization, the chunks of work which are done independently by different nodes, are not enough computationally heavy to compensate for the message passing overhead to follow, see (20) and (21) for granularity issues. Working in shared memory environment excludes any communication overhead. Threads, the lightweight processes, share the same memory space of a multicore workstation for reading input from and storing output to as they proceed with assigned to them computations. Thus the results of parallel computations can be immediately used by all working threads. We refer in this thesis to a multitasked path tracker implemented with threads for running on a multicore workstation as to a *multithreaded path tracker* or an *MT path tracker*.

1.2.2 Multiprecision Library

The ideas to achieve extended precision using hardware doubles originate in (22), see also (23), (24) and (25). We have chosen to work with fixed multiple precision as its memory management is simpler in shared memory multicore processing versus the situation when arbitrary multiple precision is used. We integrated the `QD 2.3.9` (26) library into our multithreaded path tracker. The library's arithmetic is implemented based on the idea of cited above papers to keep roundoff errors of the algebraic operations. In (24) this idea is described in terms of error-free transformations. The QD library provides arithmetic in two extended

levels of precision: double double with 32 significant digits and quad double with 64 significant digits. The use of these two extended precision levels significantly widens the variety of systems for which path tracking attains correct results. We determined experimentally that the cost factor in the overhead of using double double arithmetic in path tracking is around 8, see (27), coinciding with the number of cores on the workstation we run our experiments on - Mac Pro with 2 Quad-Core Intel Xeons at 3.2 Ghz. Thus ideally the cost of tracking one solution path in double double arithmetic can be fully compensated in a parallel multicore implementation on our workstation.

1.2.3 Programming Tools

Working with pthreads provides to a parallel algorithm designer finer control over workload distribution among threads, comparing for instance to the level of control which working with OpenMP provides. With pthreads, at the expense of more work input, we presumably can hope to achieve close to maximal speedups for systems of appropriate dimensions and degrees. The other advantage of pthreads is that they constitute a universal standard for the majority of today's mainstream operating systems.

We have chosen to work with C++ as a programming language, in essence following the choice of the QD library developers, to benefit on one side from high performance of the language and on the other side to use its object oriented features to work more conveniently with different levels of precision and complex arithmetic.

1.3 Contributions

1.3.1 Cores Compensate for Cost of Multiprecision

At the beginning we had to establish some suitable multithreaded routines for implementation of computationally involved stages of path tracking and appropriate principles of threads management, which, as combined together, would allow as low as possible thresholds on systems dimensions and degrees for reasonable speedups. Since of absence of long latency operations, we work with the number of threads equal to the number of available cores. We try to achieve good speedups with some static assignments on threads for each of the computationally involved stages of the path tracking. To achieve good speedups we need to equalize workload among computational chunks executed in parallel before a synchronization point is met, and also we need to make sure that such chunks would be enough computationally heavy. The first prevents threads, and thus cores from being idle, the second compensates passing synchronization points, which is also referred to as synchronization overhead. Often it is hard to achieve both goals simultaneously, since as we assign a bigger workload to threads between synchronization points, it is harder to make it balanced. Employing software driven multi-precision arithmetic helps very much to increase workload of threads before synchronization points for a given balanced parallel algorithm, comparing the situation when just hardware double arithmetic is used. Nevertheless the nature of liner system solving (one of the computationally involved stages of Newton's method) is so that for each thread to perform before a synchronization point a number of operations big enough to compensate synchronization overhead, and to have a balanced workload distribution among threads, we would need some minimal threshold on the ratio of

system dimension to the number of threads, whatever parallel implementation for solving linear system would be used. This threshold is less than message passing accordingly would imply to compensate the communication overhead between nodes of computer cluster, in an analogous MPI implementation, but yet not negligible. We first generalized and implemented outlined in (28, §5.3.4) parallel equalizing workload routines for Gaussian elimination (GE) and back substitution (BS) - the chosen stages for linear system solving. To compensate synchronization overhead and to have balanced workload distribution in polynomial evaluation, the other computationally intensive stage of Newton's method, we largely needed to impose some minimal reasonable threshold on the amount of monomials in the system and some very light regularity assumptions on variables degrees, see sections 5.3 - 5.5. Multitasking sparse polynomial evaluation (and differentiation) did not prescribe additional significant restrictions on considered systems. That was true for the Algorithm 3.1.1 we used in the beginning, and that remained true for more efficient Algorithm 5.1.2 described in 1.3.3 and 5.1.

Continuing Threads

To imitate the use of chosen multithreaded routines for GE, BS, and for polynomial evaluation (PE) in path tracking and thus in particular for establishing threshold dimensions for good speedups, we were running the same routine on multiple cores thousands times in a row. First extremely important for our purposes founding was establishing that for multiple runs, as the system dimensions are not that big, much better speedups are achieved when threads are not created and destroyed for each repetition of a routine, but as working threads remain

active along all of a couple of thousand of prescribed repetitions. For correspondingly small and yet very much relevant dimensions the granularity of the chosen procedures remains too fine to afford destroying and creating of threads at synchronization points even when double double and quad double arithmetic is in use. The threads, being light weight processes, require much less time to be created and destroyed than usual operating systems processes, however the time for destroying, creating, and scheduling them to cores too often accumulates and significantly diminishes the gain of parallelism. The finding about the preference of unique creating and destroying of threads for repeated runs of multithreaded versions of individual stages of the path tracker, as keeping a constant threads-to-cores one-to-one mapping all along, naturally led us to the idea to develop an entire multithreaded path tracker based on active the entire duration of tracking a solution path the same collection of threads, each of which runs on the same core from the beginning till the end of the path tracking. After all path tracker largely consists of repeated interlaced instances of its computationally intense routines.

Avoiding Mutexes

The other question which immediately arose as we were running repeated multithreaded instances for polynomial evaluation, Gaussian elimination, and back substitution was how to efficiently organize a synchronization barrier as threads proceed form one instance of a routine to another. In fact parallel computations in Gaussian elimination and back substitution have to be synchronized multiple times within single instances of those routines, and again that should be done efficiently. In (28) the synchronization techniques for threads are not considered. It is

rather suggested that, as threads are light weight processes, each time once computations need to be synchronized, the current working threads need to be destroyed, and a new collection of threads should be created for the future work and scheduled in some way to cores. In our situation, with *excessively* many synchronization points and thus necessary elaborating of continuing threads, for threads to pass properly synchronization points of computations, we conveniently adopt for shared memory model the barrier mechanisms in message passing multiprocessing, described in (28, §6.1.1). We implement these barriers with counters and flags. While each counter or flag is subject updating by only one thread at any given time, we avoid use of mutexes - locking protecting variables. The use of locking variables to protect synchronization counters from updating not properly by concurrent threads appeared to slow down parallel computations with threads significantly as it is again associated with rescheduling threads to cores.

Our work on multithreaded implementation of involved routines and accepted principles of thread management was published as work in progress towards implementing a multithreaded path tracker in PASCO 2010 proceedings, see (27).

Gluing MT Newton Method and Path Tracker

Our results presented at PASCO 2010, largely provided proof of concept that the goal we were pursuing - implementing and developing a multithreaded path tracker for compensating multiprecision - is achievable. Ahead was yet a long journey to complete obtaining the multithreaded routines realizing Newton's method and the entire path tracker. All the involved

subroutines had to be glued together. Keeping fixed number of threads working along the entire path tracking had its cost. The closer we were to obtaining the entire path tracker, the harder was becoming debugging the gluing step of the next in line subroutine to the gradually developing path tracker prototype. To find the cause of a deadlock or of wrong results we had to establish which of threads caused the inconsistency, and at what stage of the program, which was becoming more challenging as the logic of the program was becoming more sophisticated. Some interesting principles for the threads management were incorporated on the journey of composing MT Newton's method and MT path tracker from interlaced instances of multithreaded computationally intensive subroutines, while integrating the upper level logic of the path tracking into the procedure.

We achieve close to equal workload among threads for all multithreaded subroutines, as the ratio of the dimension of the system to the number of threads is big enough. Generally the thresholds on dimension to number of threads ratio to achieve good speedups are higher for linear algebra subroutines, than for polynomial evaluation subroutines. The working dimensions for the entire path tracker, for which good speedups were achieved for complex QD arithmetic on the number of cores up to eight were twenty or higher. For systems of dimensions between 20 and 30 we needed, though, to add some light requirements on lower bounds for average total degrees of monomials of the system, in addition to the requirement of having $\Omega(n)$ monomials per polynomial in the system, to make the later true, so the better gain of multithreading polynomial evaluation than the gain of multithreading linear system solving in these dimensions, would have more impact on the overall path tracking speedup. The algorithms for MT Newton's method

and MT path tracker were presented at the 17th International Conference on Applications of Computer Algebra, June 27-30, 2011 and described at (29).

1.3.2 Software

We are planning to release our MT Newton's Method and MT path tracker under GNU GPL license.

1.3.3 Towards Efficiency

While our basic multithreaded path tracker provides compensatory speedups with enough cores for use of double double and quad double precision, the path tracking in higher dimensions meets new challenges, as some traditionally used, successful for lower dimensions, algorithms for its ingredients no longer efficient in these dimensions. The later is conditioned by both: more sophisticated geometry of higher dimensions and by nonlinear dependence of the involved algorithms on dimension. Thus the first draft version of our multithreaded path tracker, even when run on 8 cores with complex quad double arithmetic, while providing very satisfactory speedups of about 6 and 7 times for dimensions 20 and 40 correspondingly, would run hours for systems of dimension 20 and days for systems of dimension 40. Processing input for the Newton homotopies, with which we are testing our implementation, also was taking not adequately long time for these dimensions. Subsequently we made the application greatly more efficient by improvements in the three next described directions.

Use of Balancing Search Trees in Input Processing

The input processing with Maple was a computational bottleneck. We came up with a quick input processing procedure, which creates the set of unique monomials of the system and suitably arranges it for further path tracking. Our expeditious procedure is based on the Standard Template Library (STL) container set functionality, promptness of which is provided by efficient balancing search trees algorithms. The scheme we suggest is applicable beyond the scope of our stated purposes, and might be used as a general input scheme for any polyhedral homotopies for sparse systems with a big enough amount of monomials.

Automatic Differentiation Advantage

Secondly we integrated into our path tracker an algorithm for polynomial evaluation and differentiation, the Algorithm 5.1.2 , which incorporates ideas used in the reverse mode Automatic Differentiation. The gain of use of the algorithm grows linearly with the dimension, and thus the employment of the algorithm in our application becomes extremely beneficial, since we work with higher dimensions.

We will illustrate how the Algorithm 5.1.2 computes the monomial $x_1^4x_3^2x_7^3x_9^5$ and its monomial derivatives $x_1^3x_3^2x_7^3x_9^5$, $x_1^4x_3x_7^3x_9^5$, $x_1^4x_3^2x_7^2x_9^5$, and $x_1^4x_3^2x_7^3x_9^4$.

First the algorithm computes in a *minimal* possible number of multiplications all partial derivatives $x_3x_7x_9, x_1x_7x_9, x_1x_3x_9$, and $x_1x_3x_7$ of $x_1x_3x_7x_9$ - the product of participating in the monomial variables - which is referred to as the Speelpenning example in (30, §3.6). The Algorithm 5.1.3 prescribes the subroutine for evaluating derivatives of the Speelpenning

example. The subroutine largely repeats steps of the reverse mode Automatic Differentiation applied to a corresponding way of evaluating the product of variables.

Then we multiply the common factor $x_1^3 x_3 x_7^2 x_9^4$ of the monomial and of its monomial derivatives by the derivatives of the Speelpenning example to obtain the values of the monomial derivatives $x_1^3 x_3^2 x_7^3 x_9^5$, $x_1^4 x_3 x_7^3 x_9^5$, $x_1^4 x_3^2 x_7^2 x_9^5$, and $x_1^4 x_3^2 x_7^3 x_9^4$.

Lastly we multiply one of the values of the monomial derivatives by the value of the corresponding variable to obtain the value of the monomial itself. Thus for instance we can multiply $x_1^3 x_3^2 x_7^3 x_9^5$ by x_1 to obtain the value of the monomial $x_1^4 x_3^2 x_7^3 x_9^5$.

The number of multiplications in which the derivatives of Speelpenning example of k variables are computed by the Algorithm 5.1.3 is $3k - 6$. Thus for our example, as $k = 4$, the needed number of multiplications is $3 * 4 - 6$=6, not $3 * 4$=12 multiplications, which we would perform if we would evaluate each derivative of the Speelpenning example from scratch.

We successfully scaled Algorithm 5.1.2 on multiple cores, as described in sections 5.3 - 5.5. To distribute evenly among threads the workload of the monomial evaluation, the most intensive computational part of the algorithm, we estimate the overall complexity of the procedure and assign to each thread such a number of monomials with their monomial derivatives to compute, that workload of a thread would be roughly equal to the overall complexity of the monomial evaluation divided by the number of threads.

Effect of a Quadratic Predictor

Lastly we selected so called quadratic predictor for our path tracker, with a complexity depending linearly on the dimension of the system on one hand, and at the same time which dramatically reduces the number of needed corrections to track a homotopy path for a system of our working dimensions. The use of the quadratic predictor is less original in path tracking, than the use of automatic differentiation techniques. However this quasi optimal choice of predictor seems to deserve an additional credit, as it proved to consistently reduce the absolute path tracking time 200-300 times in our working dimensions.

Summary

Thanks to incorporating quadratic predictor and the AD-like Algorithm 5.1.2 the over all complex QD path tracking time on eight cores was reduced on average 1000 - 4000 times in our working dimensions. In particular for dimensions 20 the new version would run seconds instead hours, and for dimension 40 it would run minutes instead days. The execution of the new, based on use of balancing search trees, input processing routine would take few portions of a second, while the execution of the Maple worksheet, previously used to process the input, would take half an hour for the Newton homotopy for system of dimension 20, and could crash an operating system run while an input for dimension 40 was assigned to be processed. Thus newly integrated techniques and new findings turned our multithreaded path tracker into an efficient original application.

1.3.4 Graphics Processing Unit Prospects

GPU processors present modern coprocessor massively parallel technology, which potentially capable to provide two digit speedups, and yet represent relatively very inexpensive hardware. At the same time the technology has very sophisticated hierarchy of memories and other specifics in its architecture, which together cause many limitations in obtaining good speedups. First it is difficult to organize efficiently irregular computations for large input data volumes on GPU processors, see (31). Also, our computations of interest are not only irregular, they consist of many iterations, which forces writing intermediate computational results into the slower memory of the device from faster ones and reading them back many times during the run of the application. That is implied as faster memories physically divided into sectors, which are managed and accessible by blocks of threads, which constitute a partition of a bigger collection of threads, given the fact that only distributing a workload of one iteration (or of a stage of an iteration) among a bigger collection of threads, provides a chance of obtaining good speedups on a GPU card. Thus going through an iterative algorithm blocks of threads need to transfer computed data from their individual sectors of faster memories into the slower memory, which is shared by the entire collection of threads, each time as this data would need be used by other blocks of threads for the next stage of the current iteration or for the next iteration. The next step would be extracting the computed data by blocks of threads, which need it for their very next computations, from the slower memory into their individual sectors of faster memories. The goal of a GPU programmer in such situation is to organize computations so that blocks of threads would do as much as possible work independently of each other, efficiently using only

their individual fast memories sectors of very limited capacities, so to minimize the volumes of the described above memory transfers and the very number of occurrences of such transfers. If this task would not be properly addressed, the frequent transfers between faster and slower memories might bring to a situation when no speedup would be achieved by a GPU application, or it would run even slower, than its sequential counterpart.

Recently we accelerated the AD-like Algorithm 5.1.2 for polynomial evaluation and differentiation on GPU processors using CUDA computing architecture. Our massively parallel implementation is described in detail in Chapter 7. A paper on this implementation will be presented at the 13th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC-12) to be held May 21-25, 2012, Shanghai, China and will be published in the IPDPS-12 proceedings by IEEE CS.

The NVIDIA Tesla 2050/2070 card provides very satisfactory speedups of 10-20 times for our implementation for systems of dimensions 30-50, comparing sequential run on HP Z800 workstation with Intel Xeon X5690 CPU at 3.47 Ghz, as double arithmetic is used.

The computational complexity of polynomial evaluation and differentiation in most cases dominates complexities of other path tracking stages all together. At the same time polynomial evaluation and differentiation represent the most irregular part of path tracking computations. Thus our acceleration and efficient adopting the QD library for GPU processors by Lu, He, and Luo, see (32), strongly encourage setting a long term prospect for developing a multitasked path tracker, compensating use of extended arithmetic on a Graphics Processing Unit.

CHAPTER 2

CONTINUING THREADS

2.1 Necessity

The necessity of continuing threads is well demonstrated by the results in Table I. In the table the timings and speedups for repeated runs of a multithreaded version of matrix vector product are presented. Matrix vector product essentially simulates coefficient product - the second stage of the polynomial evaluation and differentiation algorithm 3.1.1. The dimensions chosen for demonstration in Table I make the matrix vector product simulate the coefficient product of a system of dimension 20 with 20 monomials in each of the polynomials of the system and of its Jacobian. In particular the number of rows of the considered matrix A, $420 = 20^2+20$, is equal to the total number of polynomials in a system of dimension 20 and in its Jacobian.

`Product of a matrix A of size` 420×20
`with a vector B of dim=20, 100,000 times`

#threads	multiple creation	speedup	unique creation	speedup
1	`1m 34.287s`	1	`1m24.937s`	1
2	`1m0.860s`	1.55	`0m42.951s`	1.978
4	`0m40.817s`	2.31	`0m21.748s`	3.906
8	`0m40.502s`	2.33	`0m11.370s`	7.47

TABLE I

complex QD arithmetic

Note that as unique creation of threads is employed, meaning that threads are created once, and destroyed only after 100,000 instances of the matrix vector product are computed, speedups grow almost linearly with the number of threads, and are close to maximal ones. On the other hand when threads are created and destroyed for each of the 100,000 instances of the product, the speedups are much worse. In particular there is almost no difference in the running time, as four threads compute the product versus the set up when eight threads are assigned to accomplish this task. Creation of threads and scheduling them to cores appears to be too expensive for the performed within one iteration number of quad double complex multiplications and summations by a thread.

As unique creation of threads is employed, the scheme 2.3.1 is used to synchronize the computations. Nearness of the obtained speedups to maximal ones also shows the efficiency of the scheme.

2.2 Debugging Overhead

The major obstacles in debugging as continuing threads are employed are mentioned in section 1.3.1. One should be aware though, as developing an iterative application with pthreads, that as the logic of the application grows, use of continuing threads brings an additional very significant difficulty in debugging. Use of continuing threads inevitably considerably extends the time and effort needed for obtaining a multilevel application, but as in our case might provide much better speedups.

2.3 Synchronization and Avoiding Mutexes

To synchronize repeated runs of a multithreaded routine A which does not have additional synchronization points within each its iteration we use the following scheme, which employs an integer variable `counter` and p boolean flags $st[i], i \in 0, \ldots, p-1$.

Algorithm 2.3.1. *at the k-th iteration a thread with id=My_ID performs:*

Input: My_ID	*maps a portion of A to the thread*
$st[i] = 0\ \forall i$	all flags set to zero
Output: `counter` $= k+1$,	the k-th iteration of A is completed
while `counter` $< k$ {}	*wait the iteration begins*
do A(My_ID);	*complete a portion of A*
set st[My_ID]=1	*signal that a portion is complete*
if (My_ID=0) {	
1. while $st[i] = 1\ \forall i$ {}	*wait all done*
2. set $st[i] = 0\ \forall i$	*update flags*
3. `counter` =`counter`+1	*signal all to start a new iteration*
}	

Notice that in scheme 2.3.1 the values of each flag and of the counter are modified by only one thread at any given time, while state or value of these synchronization variables can be read by several threads concurrently. This helps to avoid use of locking variables, which again would

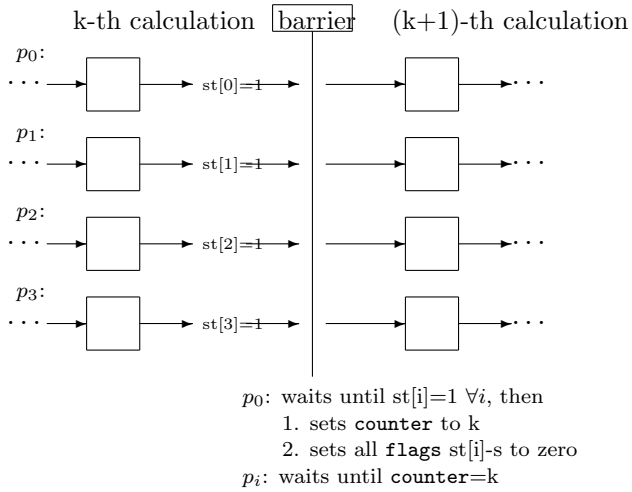

Figure 1. Synchronization with a `counter` and p `flags`

prompt an expensive remapping of threads to cores . The scheme 2.3.1 is further illustrated in Figure 1.

In multithreaded polynomial evaluation in section 3.1.1 we have two different routines glued together. We synchronize repeated runs of a hybrid of two routines A and B with the following synchronization scheme which employs two counters and p flags.

Algorithm 2.3.2. *at the k-th iteration a thread with id=My_ID performs:*

Input: My_ID — *maps portions of A and B to the thread*

$st[i] = 0\ \forall i$ — all flags set to zero

Output: `counter1` $= k + 1$, — the k-th iteration of A is completed

`counter2` $= k + 1$ — the k-th iteration of B is completed

while `counter2` $< k$ { } — *wait the iteration of A begins*

do A(My_ID); — *Complete a portion of A*

set st[My_ID]=1; — *Signal a portion of A is complete*

if (My_ID=0) {

1. *while* $st[i] = 1\ \forall i$ {} — *wait all portions of A are complete*
2. *set* $st[i] = 0\ \forall i$ — *update flags*
3. `counter1=counter1+1` — *signal all to start B*

}

while `counter1` $< k + 1$ { } — *wait the iteration of B begins*

do B(My_ID); — *Complete a portion of B*

set st[My_ID]=1; — *Signal a portion of B is compete*

if (My_ID=0) {

1. *while* $st[i] = 1\ \forall i$ {} — *wait all portions of B are complete*
2. *set* $st[i] = 0\ \forall i$ — *update flags*
3. `counter2=counter2+1` — *signal all to start the new iteration of A*

}

CHAPTER 3

MULTITASKING COMPONENTS OF NEWTON'S METHOD

At the beginning of each Newton's iteration we need to evaluate the system and its Jacobian matrix. We assume throughout this work that polynomials of the considered systems and of their Jacobians generally do not factor. We start polynomial evaluation and differentiation from evaluation of all monomials appearing in the system and its Jacobian. In this chapter we consider a pleasingly scalable, relevant for sparse polynomial systems, basic two-stages algorithm for polynomial evaluation and differentiation. More advanced Automatic Differentiation-like algorithm for accomplishing the same task is considered in Chapter 5.

The second stage of Newton's iteration is solving of linear systems of equations. We accomplish this in our path tracker with a multithreaded version of Gaussian Elimination (GE) with partial pivoting followed by a multithreaded back substitution (BS). Multithreaded QR decomposition to treat overdetermined systems is yet to be integrated into the MT path tracker. Dealing in this work with intermediate dimensions, we found the best for our purposes to implement parallel algorithms for GE and BS which are based on algorithms for these tasks presented in (28, §5.3.4). The benefits of blocked Linear Algebra, see (34), and tiled Linear Algebra, see (33), show in higher dimensions.

3.1 MT Polynomial Evaluation

3.1.1 Algorithm

The serial version of the presented in this chapter algorithm for polynomial evaluation and differentiation follows.

Algorithm 3.1.1.

- *1. We first evaluate the set of all participating in the system and its Jacobian unique monomials and store the obtained values in an array MONs in total degree (for instance) monomial order. The computation of each monomial value consists of first computing by binary exponentiation of all participating in the monomial variables exponents and subsequent computing the product of obtained exponents.*

- *2. Each polynomial is represented with two arrays of equal length: the array COEFFs, which consists of nonzero coefficients of the polynomial, and the array INDs, the elements of which indicate to which monomials in MONs the coefficients in COEFFs correspond. To evaluate the polynomial we compute the inner product of the array COEFFs with the corresponding sub array of MONs, which is retrieved with mapping provided by elements of INDs.*

3.1.2 Parallelization

We distribute the workload of each of the two steps of the algorithm 3.1.1 as following:

Algorithm 3.1.2.

- *1: We assign to each of the threads to compute a* ***continuous block*** *of values in the array MONs of the length approximately equal to*

$$\frac{\textit{total number of monomials}}{p}.$$

- *2: The values of the polynomials and their derivatives are stored in an array of the length* $n^2 + n$. *We assign to each of the threads to compute a* ***continuous block*** *of this array of the length approximately equal to*

$$\frac{n^2 + n}{p}.$$

Synchronization for repeated runs of the above multithreaded polynomial evaluation routine is done as the synchronization Algorithm 2.3.2 suggests. The speedups for the Algorithm 3.1.2 for randomly generated systems of dimensions 20 and 40 are summarized in parts (a) and (b) of Table II correspondingly. In particular the speedups are high since for randomly generated systems

1. it is required almost the same amount of work for evaluation of each of participating in the system or in its Jacobian monomials;
2. the number of terms is roughly an invariant of all polynomials of the system and of all polynomials of its Jacobian.

420 polynomials, with 20 monomials in average
of degree 20 in average, in 20 variables, 1000 times.

#threads	real	user	sys	speedup
1	0m7.058s	0m7.053s	0m0.004s	1
2	0m3.861s	0m7.711s	0m0.005s	1.828
4	0m1.875s	0m7.480s	0m0.006s	3.764
8	0m0.993s	0m7.903s	0m0.007s	7.108

(a)

1620 polynomials, with 40 monomials in average
of degree 40 in average, in 40 variables, 1000 times.

#threads	real	user	sys	speedup
1	0m55.497s	0m55.475s	0m0.019s	1
2	0m28.818s	0m57.597s	0m0.022s	1.926
4	0m14.600s	0m58.309s	0m0.036s	3.801
8	0m7.376s	0m58.855s	0m0.031s	7.524

(b)

TABLE II

Polynomial Evaluation, complex QD arithmetic

However to achieve good speedups, as applying multithreading Algorithm 3.1.1, we do not have to impose such strong regularity assumptions on polynomials of the system. In sections 5.3 - 5.5 we efficiently multithread more favorable AD-like Algorithm 5.1.2, which also evaluates a polynomial system and its Jacobian matrix, for reasonably *irregular* systems. We will favor the Algorithm 5.1.2, as it is much more prudent for sparse polynomial systems than the algorithm 3.1.1, computing much less multiplications in its monomial evaluation part.

3.2 MT Linear Systems Solver

3.2.1 MT Gaussian Elimination

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

$$a_{j,k} := a_{j,k} - \frac{a_{j,i}}{a_{i,i}} a_{i,k}, \quad k = j, j+1, \ldots, n, \tag{3.1}$$

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 before the row modifications for a given pivot row, we perform necessary row interchanges (partial pivoting) to increase the numerical stability.

To achieve an equal workload among p threads, we assign rows to threads as follows: the first thread will work on rows $1, p+1, 2p+1, \ldots$, the second thread will work on rows $2, p+2, 2p+2, \ldots$, in general: the ith 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 row. It is possible to show that as $\frac{n}{p} \to \infty$, the ideal speed up $\to p$.

For correctness, threads are synchronized. Thread number 1, which performs partial pivoting row interchanges, to start the row interchange for the current pivot row, waits until row modifications are not finished by all threads for the previous pivot row. Also after it begins the row interchange for the current pivot row, all other threads wait until the row interchange is finished to start their row modifications for the current pivot row. The procedure is syn-

chronized to accomplish this with p counters (by the number of threads). The counter which is maintained by the thread number one shows for how many pivot rows the row interchange was performed. The counters maintained by the other $p-1$ threads are to show for how many pivot rows they done assigned to them row modifications.

In parts (a) and (b) of Table III we see that intermediate speedups are obtained on 8 cores for dimensions 20-40; the part (c) of Table III shows that much closer to maximal speedups are obtained on 8 cores as the ratio $\frac{n}{p}$ is bigger.

3.2.2 MT Back Substitution

Our parallel implementation of the algorithm for solving a triangular system is inspired by an analogous algorithm in (28, §5.3.4).

Let a lower triangular n-by-n matrix L with entries $\ell_{k,m}$ and an n-vector $\mathbf{b}$ define the system L $\mathbf{x} = \mathbf{b}$. The solution vector $\mathbf{x}$ is computed via the formulas

$$x_k := \frac{1}{\ell_{k,k}} \left(b_k - \sum_{m=1}^{k-1} \ell_{k,m} x_m \right), \quad k = 1, 2, \ldots, n. \tag{3.2}$$

The calculation of x_k needs the values for all previous components x_m, for $m = 1, 2, \ldots, k-1$.

In the parallel implementation suggested in (28, §5.3.4) in a pipeline of n processors, each processor computes a value of one variable. In our implementation p threads compute the values of all participating variables in a circular manner as following. Labeling threads by $1, 2, \ldots, p$, the ith thread computes first the value x_i and then computes successively all values of x_{i+jp} for all j: $i + jp \leq n$.

Starting to calculate x_{i+jp}, the ith thread does not wait until all values of x_k, for all k: $k < i + jp$ are computed. At first, without waiting, the ith thread computes the partial sum

$$\sum_{m=1}^{i+(j-1)p} \ell_{i+jp,m} x_m \quad of \quad \sum_{m=1}^{i+jp-1} \ell_{i+jp,m} x_m, \tag{3.3}$$

since at least $i + (j-1)p$ of x_k values computed by the time it starts its 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. Note that the i-th thread proceeds to computing of the $i+jp$ variable only after it itself finishes computing the value of the $i+(j-1)p$ variable, that is why it is guaranteed that at least $i + (j-1)p$ variables are computed before computations for $i + jp$ variable begin. Thus larger values of n/p lead to larger speedups, since for large indexes $i + jp$ by the time the ith thread starts computing x_{i+jp} almost all x_k with $k < i + jp$, namely $i + (j-1)p$ of them, are already computed, therefore the computations of x_{i+jp} with large indexes are done almost uninterruptedly, with at most $p - 1$ short breaks for the ith thread to wait until calculations for less then p preceding values of x_k to x_{i+jp} are completed by the other threads.

To synchronize the calculations within back substitution, 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. The thread, which computes the last n-th variable, after it completes calculation

of this last variable, augments the counter of completed back substitutions by one to synchronize repeated runs of the routine in the over all path tracking context.

Parts (a) and (b) of Table IV indicate in particular that we obtain intermediate speedups on 8 cores, as QD arithmetic is used, for dimensions 20-40. These intermediate dimensions will be eventually working dimensions for our path tracker. The part (c) of Table IV shows that when the ratio $n/p = 200/8$ is big enough we obtain close to maximal speedups on all available 8 cores.

20-by-20 matrix, 1000 times

#threads	real	user	sys	speedup
1	0m3.105s	0m3.103s	0m0.003s	1
2	0m1.871s	0m3.732s	0m0.005s	1.660
4	0m1.279s	0m5.099s	0m0.005s	2.428
8	0m0.956s	0m7.614s	0m0.007s	3.248

(a)

40-by-40 matrix, 1000 times

#threads	real	user	sys	speedup
1	0m21.424s	0m21.407s	0m0.014s	1
2	0m11.990s	0m23.951s	0m0.018s	1.787
4	0m7.354s	0m29.390s	0m0.011s	2.913
8	0m4.892s	0m38.956s	0m0.024s	4.379

(b)

200-by-200 matrix, 10 times

#threads	real	user	sys	speedup
1	0m23.227s	0m23.213s	0m0.012s	1
2	0m11.918s	0m23.758s	0m0.014s	1.949
4	0m6.245s	0m24.779s	0m0.014s	3.719
8	0m3.493s	0m27.378s	0m0.022s	6.650

(c)

TABLE III

GE with partial pivoting, complex QD arithmetic

20-by-20 matrix, 10000 times

#threads	real	user	sys	speedup
1	0m2.056s	0m2.054s	0m0.002s	1
2	0m1.131s	0m2.258s	0m0.003s	1.818
4	0m0.760s	0m3.031s	0m0.003s	2.705
8	0m0.730s	0m5.814s	0m0.005s	2.816

(a)

40-by-40 matrix, 10000 times

#threads	real	user	sys	speedup
1	0m7.457s	0m7.453s	0m0.003s	1
2	0m3.905s	0m7.802s	0m0.004s	1.910
4	0m2.197s	0m8.777s	0m0.005s	3.394
8	0m1.517s	0m12.099s	0m0.008s	4.916

(b)

200-by-200 matrix, 1000 times

#threads	real	user	sys	speedup
1	0m17.045s	0m17.037s	0m0.007s	1
2	0m8.661s	0m17.306s	0m0.008s	1.968
4	0m4.398s	0m17.561s	0m0.009s	3.876
8	0m2.281s	0m18.184s	0m0.012s	7.473

(c)

TABLE IV

Back Substitution with complex QD arithmetic

CHAPTER 4

MT NEWTON'S METHOD AND MT PATH TRACKER

4.1 MT Path Tracker

4.1.1 Algorithm

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 4.1.1 provides pseudo code for the multithreaded version of a path tracker. Every thread executes the same code. We synchronize the work of threads as they proceed from one stage of the path tracking to another using principals similar to those presented at section 2.3. In particular the values of each synchronization counters and each of synchronization flags can be updated by only one relevant thread at any given time. In addition to a variety of global counters which indicate how many instances of smaller path tracking stages were completed, each thread uses a local counter `my_corr_Ind` which assists it to keep up with the other threads as they start a new correction.

Algorithm 4.1.1 (Multithreaded Path Tracker)**.**

Input: $h(\boldsymbol{x},t) = \boldsymbol{0}$, $\boldsymbol{z}$.	homotopy and start solution
Output: $\boldsymbol{z} : h(\boldsymbol{z}, 1) = \boldsymbol{0}$ *or fail.*	solution at end of path or failure

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

As we will see in the next section each thread also uses a local counter, which assists it to keep up with the other threads as a new Newton's iteration begins.

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.

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. The parallel implementations of its stages are described in previous chapters.

4.1.2 Computational Results and Testing Homotopies

Computational Results

As Table V shows, if we allow total degrees of monomials to be as high as the system dimension, the multithreaded path tracker provides good speedups even for relatively small dimensions as QD complex arithmetic is used. By assigning such high degrees to monomials of the system, we make polynomial evaluation and differentiation to computationally dominate linear system solving in Newton's iteration to that extent, that lack of good speedups of linear system solving multithreaded routines for these small dimensions is not longer noticeable. On the other hand we would like to avoid considering systems of not naturally high degrees. In applications degrees of monomials rarely exceed half of the system's dimension.

As we choose total degrees of participating monomials to be around half of dimension of the system, we obtain good speedups for systems of dimensions of about forty and higher if

Tracking Newton homotopy path for a system of 10 polynomials
in 10 variables,10 monomials in each polynomial, with monomials of total degree 10

#threads	real	user	sys	speedup
1	14.064s	14.042s	0.006s	1
2	7.548s	15.060s	0.008s	1.863
4	4.074s	16.207s	0.009s	3.452
8	2.280s	17.960s	0.015s	6.168

TABLE V

Complex QD arithmetic

double double complex arithmetic is used, and of about twenty and higher if quad double complex arithmetic is used. On the other side it appeared that with methods considered so far the obtained efficient multithreaded double double and quad double implementations run unacceptably long time in these working dimensions. The two next chapters of the thesis consider improvements which make our path tracker multithreaded double double and quad double implementations to track solution paths of a polynomial homotopy $H(x,t) = (1-t)^2 * G(x) + \gamma t^2 F(x)$ greatly faster in their working dimensions.

Testing Systems and Homotopies

For testing our path tracker we consider target systems $F(x)$ of dimension n with randomly generated coefficients and degrees while : number of monomials in polynomials is equal to n, and total degree of monomials is equal to $\frac{n}{2}$. We also choose $G(x) \equiv F(x) - F(1,\ldots,1)$ and choose $x^* = (1,\ldots,1)$ to be starting solution for $G(x)$. All timings provided in section 6.3 are for randomly generated target systems , F(x), of the above characteristics and $G(x) \equiv F(x) -$

$F(1)$. However the multithreaded version of the path tracker, with improvements described in Chapters 5 and 6 remains efficient when almost no regular assumptions are imposed on $F(x)$, and also apparently the obtained multithreaded path tracker could be efficiently applied not only for the Newton homotopy but for any polyhedral homotopy which helps to solve $F(x) = 0$.

4.2 Multithreaded Newton's Method

In this section we focus on our multithreaded version of Newton's method using multithreaded polynomial evaluation and linear system solving described in chapter 3. Following the same notational conventions as in Algorithm 4.1.1, pseudo code is described in Algorithm 4.2.1 below.

As we consider Newton's method in context of path tracking, we do not provide here timings for running Newton's method on its own. As Newton's method bears almost all computational workload of the path tracker, the speedups for it might be estimated using speedups obtained for multithreaded path tracking. We can say that the speedups of the multithreaded Newton's method for systems of those characteristics as intermediate systems prescribed by the homotopies, with which we are testing the multithreaded path tracker, are the same as the speedups obtained for the path tracker at the corresponding test runs.

At the same time for big enough dimensions, for which applying Newton's method with extended precision would take a long enough time, our multithreaded implementation of the method might be of independent of path tracking context use.

Algorithm 4.2.1 (Multithreaded Newton's Method)**.**

Input: $h(\boldsymbol{x},t) = \boldsymbol{0}$, $\boldsymbol{z}$; — homotopy and initial solution

ϵ, *Max_It.* — tolerance and maximal #iterations

Output: $\boldsymbol{z} : ||h(\boldsymbol{z},t)|| < \epsilon$ *or fail.* — corrected solution or failure

$i := 0$; — count #iterations

$||h(\boldsymbol{z},t)|| := 1$; — initialize residual

while $(||h(\boldsymbol{z},t)|| > \epsilon)$ *and* $(i <$ *Max_It) do*

  V := Monomial_Evaluation(`my_ID`*,h,*$\boldsymbol{z}$*);* — multithreaded monomial evaluation

  Status_MonVal[`my_ID`*] := 1;* — thread done with monomial evaluation

  if (`my_ID` *= 1) then* — flag adjustments for next stage

    while $(\exists$ *ID: Status_MonVal[ID] = 0) wait;* — thread 1 waits

    for all ID do Status_MonVal[ID] := 0; — flags reset for next stage

    Mon_Ind := Mon_Ind + 1; — update monomial counter

  end if;

  while (Mon_Ind < my_Iter+1) wait; — wait till all monomials are evaluated

  Y := Coefficient_Product(`my_ID`*,V,h);* — multiply monomials with coefficients

  Status_Coeff[`my_ID`*] := 1;* — thread done with coefficient product

if (my_ID = 1) then — flag adjustments for next stage

while (∃ ID: Status_Coeff[ID] = 0) wait; — thread 1 waits

for all ID do Status_Coeff[ID] := 0; — flags reset for next stage

$||h(\mathbf{z},t)||$ *:= Residual(Y);* — calculate residual

Coeff_Ind := Coeff_Ind + 1; — update coefficient counter

end if;

while (Coeff_Ind < my_Iter+1) wait; — wait till all polynomials are evaluated

Ab := GE(my_ID,my_Iter,Y ,pivots); — row reduction on Jacobian matrix

$m := (n-1)$*(my_Iter+1);* — used for synchronization

while (∃ ID: pivots[ID] < m*) wait;* — wait for row reduction to finish

Back_Subs(my_Id,my_Iter,Ab,Δ $\mathbf{z}$*,BS_Ind);* — multithreaded back substitution

while (BS_Ind < my_Iter+1) wait; — wait till back substitution done

if (my_ID = 1) then

$\mathbf{z}$*:=* $\mathbf{z}$*+* Δ $\mathbf{z}$*;* $i := i+1$*;* — update solution

$\mathbf{z}$*_Ind := z_Ind + 1;* — counter to update **z**

end if;

while (z_Ind < my_Iter+1) wait; — wait till solution is updated

my_Iter := my_Iter + 1;

end while;

fail := $||h(\mathbf{z},t)|| \geq \epsilon$.

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 3.1.1 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 in accordance with the scheme 3.1.2 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 4.2.1. The threads are synchronized within GE after each pivot row interchange, and as well after related row modifications for each pivot row, as described in 3.2.1.

The output of the procedure GE is passed to the back substitution procedure Back_Subs. Distributing workload among threads in Back_Subs and synchronization within the routine described in 3.2.2.

CHAPTER 5

AD-LIKE POLYNOMIAL EVALUATION AND DIFFERENTIATION

Here we present an algorithm for sparse polynomial evaluation and differentiation, which is much more efficient than the algorithm 3.1.1. A major disadvantage of the algorithm 3.1.1 is computing each participating monomial from scratch, thus excessively many times repetitively evaluating existing common factors. One of the improvements of the algorithm described in this section is that it does compute the greatest common factor of a monomial and of all its partial derivatives, to which we refer below as to a *monomial prototype* of a given monomial, only once. For sparse polynomial systems computing for each monomial a monomial prototype only ones, versus of computing it repetitively as evaluating individually the monomial and each of its partial derivatives, already provides a significant reduction in the number of needed multiplications as we evaluate monomials of the system and of its Jacobian. The other advantage of great importance of the presented here algorithm is that it completes evaluation of a monomial and its partial derivatives, after the monomial prototype is already computed, in a *minimal* possible number of multiplications. The presented algorithm uses ideas of the reverse mode AD, see (30, §3.2). As we will see *individual* evaluation and differentiation of each of the monomials of the system, prescribed by the algorithm, provides us finer control on distributing the computational workload of the evaluation of the system and of its Jacobian among threads.

5.1 Terminology and the Serial Algorithm

To avoid confusion we accept the following terminology.

Definition 5.1.1. For a *term* $\beta = c_a\, x_{i_1}^{a_{i_1}} x_{i_2}^{a_{i_2}} \cdots x_{i_k}^{a_{i_k}}$ in a polynomial f of the system, we refer to $\gamma := x_{i_1}^{a_{i_1}} x_{i_2}^{a_{i_2}} \cdots x_{i_k}^{a_{i_k}}$ as to a *monomial* of β. Since a partial derivative of a monomial is actually not a monomial, but rather a term, we introduce our concept of a *monomial derivative* as the monomial of a partial derivative of a monomial. Thus we refer to monomials $x_{i_1}^{a_{i_1}-1} x_{i_2}^{a_{i_2}} \cdots x_{i_k}^{a_{i_k}}, \ldots, x_{i_1}^{a_{i_1}} x_{i_2}^{a_{i_2}} \cdots x_{i_k}^{a_{i_k}-1}$ as to monomial derivatives of γ with respect to $x_{i_1}, \ldots, x_{i_k}$ correspondingly. Note that the coefficients $(c_a a_{i_1}), \ldots, (c_a a_{i_k})$ of the corresponding to these monomials terms $\frac{\partial \beta}{\partial x_{i_1}}, \ldots, \frac{\partial \beta}{\partial x_{i_k}}$ in $\frac{\partial f}{\partial x_{i_1}}, \ldots, \frac{\partial f}{\partial x_{i_k}}$ are precomputed before the actual path tracking begins. Finally we refer to f as to a *hosting polynomial* of the term β (and of the monomial γ) in the system, and to $\frac{\partial f}{\partial x_{i_1}}, \ldots, \frac{\partial f}{\partial x_{i_k}}$ as to hosting polynomials of the terms $\frac{\partial \beta}{\partial x_{i_1}}, \ldots, \frac{\partial \beta}{\partial x_{i_k}}$ (and of the corresponding monomial derivatives of γ) in the Jacobian of the system.

The announced AD-like algorithm for evaluation of an intermediate system of a homotopy $H(x,t) = (1-t)^2 * G(x) + \alpha t^2 F(x)$ and of its Jacobian matrix is presented next. Without loss of generality, for simplicity of exposition or the step 3 of the algorithm, we assume that **mon**(F,G) = **mon** F.

Algorithm 5.1.2. *1. Compute all powers of x_i^j, for $i \in \{1, 2, \ldots, n\}$ and $j \in \{2, \ldots, d_i - 1\}$, where d_i stands for the maximal power to which x_i appears in **mon** (F, G).*

2. *for each monomial* $\gamma \in \boldsymbol{mon}\ (F,G)$ *compute* γ *and its monomial derivatives as following: first compute the monomial prototype* $x_{i_1}^{a_{i_1}-1}x_{i_2}^{a_{i_2}-1}\cdots x_{i_k}^{a_{i_k}-1}$ *of* γ *as a product of* k *quantities computed at step 1 of the algorithm; secondly compute in* $3k-6$ *multiplications monomial derivatives of Speelpenning example* $x_{i_1}x_{i_2}\cdots x_{i_k}$ *as described below in algorithm 5.1.3; thirdly multiply the monomial prototype by the derivatives of Speelpenning example to obtain monomial derivatives of* γ*; lastly obtain the value of* γ *by multiplying its monomial derivative with respect to* x_{i_1} *by the value of* x_{i_1}*.*

3. *Multiply the value of each* $\gamma \in \boldsymbol{mon}\ (F,G)$ *and the values of its monomial derivatives by the (real double) current values of the homotopy parameter* t *and of* t^2*;*

4. *multiply the coefficients with the corresponding evaluated monomials, and perform summations of obtained products within hosting polynomials of the system and the Jacobian.*

Algorithm 5.1.3. *We obtain the monomial derivatives* $\omega_m =: \prod_{j=1,\ j\neq m}^{j=k} x_{i_j}\ ,\ m \in 1,\ldots,k$ *of Speelpenning example* $\prod_{j=1}^{j=k} x_{i_j}$ *in* $3*k-6$ *multiplications as following:*

1. we get recursively all forward products $\psi_m = x_{i_1} * x_{i_2} * \cdots * x_{i_m}, m \in 1,\ldots,k-1$ *by* $\psi_m = \psi_{m-1} * x_{i_m}$*,* $\psi_1 = x_{i_1}$*.*

2. similarly we obtain all backward products $\varphi_m = x_{i_k} * x_{i_{k-1}} * \cdots * x_{i_{k-m+1}}, m \in 1,\ldots,k-1$ *by* $\varphi_m = \varphi_{m-1} * x_{i_{k-m+1}}$*,* $\varphi_1 = x_{i_k}$*.*

3. finally we obtain $\omega_m,\ \ m \in 1,\ldots,k$ *as* $\omega_1 = \psi_{k-1}$*,* $\omega_k = \varphi_{k-1}$*,* $\omega_m = \psi_{m-1} * \varphi_{k-m+2}, m \in 2,\ldots,k-1$*.*

The four steps in the above presentation of the serial Algorithm 5.1.2 correspond to four stages in our multithreaded implementation of the algorithm. Each of the subsequent sections of the chapter is devoted to one of the stages of our implementation. Threads are synchronized accordingly after each of the four stages.

5.2 Precomputing Powers

Let $d_1, \ldots, d_n$ denote the maximal in **mon** (F,G) degrees of variables $x_1, \ldots, x_n$ correspondingly. The first stage of the polynomial evaluation and differentiation algorithm 5.1.2 is precomputing of powers of variables of the system $x_1^2, \ldots, x_1^{d_1-1}, \quad x_2^2, \ldots, x_2^{d_2-1}, \quad \ldots \quad , x_n^2, \ldots, x_n^{d_n-1}$. This procedure has relatively low computational complexity $\omega_{pre} = \sum_{i=1}^{n}(d_i - 2)$ multiplications. For instance, if the degrees of all variables do not exceed n, which still allows a system to have monomials of very high total degrees (up to n^2), precomputing powers would require no more than n^2 multiplications. Note that the set of powers of variables is computed only once serving the subsequent computation of monomial prototypes of *all* monomials of the system. That is the reason we add the prefix "pre" to the word "computing" as we refer here to computation of powers.

In the implementation of the Algorithm 5.1.2, which was used to obtain timings and speedups for path tracking provided in section 6.3, precomputing powers is not parallelized, since the number of multiplications it requires for systems we consider, see section 4.1.2, is less than $\frac{n^2}{4}$. Obtaining close to maximal speedups for path tracking for dimension 40 confirms that multithreading precomputing powers is not necessary for considered systems. As during

any other stage of the path tracking, which is computationally light and thus not parallelized in our implementation, the idle threads run busy waiting loops, as one of the threads precomputes the powers of variables.

On the other hand precomputing powers is easy to efficiently multithread in a vast variety of settings. In most cases it is possible to partition variables in a way that while each thread works on powers of all variables of a partition member, the workload per thread is approximately equal to $\frac{\omega_{pre}}{p}$. When systems are generated randomly, to achieve this, any partition with members of cardinality $\frac{n}{p}$ works. In less regular settings it is often enough to sort variables so that $d_1 \geq d_2 \geq \ldots \geq d_n$ and then let the thread 0 to work on variables $x_1, x_{p+1}, \ldots$, thread 1 on $x_2, x_{p+2}, \ldots$ and so on.

Alternatively to precomputing powers exponentiation by squaring for each of appearing exponents in monomials is used in Algorithm 3.1.1. While which of the two alternatives works better depends on the subject to evaluation and differentiation polynomial system, employing precomputing powers in Algorithm 5.1.2 makes it much easier to estimate the over all complexity of the algorithm, which helps to efficiently distribute the workload of the algorithm among threads.

5.3 Multithreaded Computing Monomials and Monomial Derivatives

When powers are precomputed, for a monomial $\gamma \in \mathbf{mon}\ (F, G)$ of $k \leq n$ variables, to accomplish all the work prescribed in the second step of the Algorithm 5.1.2, we need to perform

$$(k-1)+(3k-6)+k+1=5k-6 \quad \text{DD/QD multiplications.}$$

Thus to accomplish the first two steps of the algorithm for all monomials of the system we need over all

$$\omega_{pre}+\sum_{\alpha\in\mathbf{mon}(F,G)}(5k-6) \quad \text{DD/QD multiplications.}$$

We assign to each thread to compute such a sub collection of monomials of **mon**(F,G) (with their monomial derivatives) that the number of multiplications per thread

$$\approx \frac{\sum_{\alpha\in\mathbf{mon}(F,G)}(5k-6)}{p}.$$

Since in our settings, see section 4.1.2, in our working dimensions, e.g. n=40, we have thousands of monomials in **mon**(F,G) and $k \leq n/2 = 20$, the needed partition of the set **mon**(F,G) among threads is easy to organize.

Remark 5.3.1. *Almost no regular assumptions on a system are required for obtaining a good speedup in our multithreaded version of the second step of the Algorithm 5.1.2.*

Note that the workload for each of monomials of the second step of the algorithm depends only on the number of variables in the monomial.

Consider dimension $n = 40$ - a working dimension for our path tracker, prescribed by linear system solving speedups, for both (DD and QD) precision levels. If we keep the requirement

of n monomials in each polynomial , and the requirement that total degrees of monomials do not exceed half of the dimension, thus less or equal than 20, we would have about 1600 monomials in the system, with only 20 possible workloads for them. Since each of monomials of the system imposes a small amount of work, the later implies that whatever would be the distribution of the amount of variables in the monomials of the system, we always would be able to distribute the overall workload equally enough among 8 threads. That is why we state that the described above parallelizing principle for the second step of Algorithm 5.1.2 remains efficient as almost no regularity assumptions on monomials of the system are made. The only necessary assumption is that the system should not have too few monomials. As argued above having $\Theta(n)$, where n is the dimension of the system, monomials in each polynomial of the system, i.e. $\Theta(n^2)$ monomials overall in the system, is definitely enough for good speedups.

5.4 Homotopy Parameter Multiplications

If **mon**(F,G) = **mon** F, it is required to multiply a complex DD/QD value of each monomial from **mon** F and of values of all of its monomial derivatives by the real double value of t twice. Thus the overall workload of this step would be $\sum_{\alpha \in \mathbf{mon}_F}(2k+2)$ complex DD/QD - real double multiplications. We multithread this step similarly to how the second step of the algorithm is multithreaded. Namely we assign to each thread to perform homotopy parameter multiplications for such sub collection of monomials that the number of performed multiplications performed by a thread is approximately equal to the overall number of multiplications of the step, $\sum_{\alpha \in \mathbf{mon}_F}(2k+2)$, divided by p.

Note that we do not combine the second and the third steps of the algorithm in one multithreaded procedure. Our work partitioning of each of these two steps of the algorithm is based on estimating overall complexity of the step. Multiplications in the second and the third steps of the algorithm are of different complexities. Thus it would be not that easy to measure the overall complexity of a combined procedure, and to divide it equally among threads. Observe that the quantity $\sum_{\alpha \in \mathbf{mon}_F}[(5k-6)+(2k+2)]$ of conventional units would not constitute a proper estimate for the combined complexity of the second and third steps. Indeed multiplications of different complexities are included in this sum with the same weights.

5.5 Multithreaded Coefficient Product (CP)

The number of polynomials and derivatives of a system is n^2+n, thus e.g. for $n=40$ we need to compute 1640 polynomials. An approach for multithreading CP we used in algorithm 3.1.2 was to assign to each thread to compute about $\frac{n^2+n}{p}$ polynomials and derivatives. Such assignment to threads, despite a big $\frac{\text{number of polynomials}}{p}$ ratio in our working dimensions, could provide not a balanced workload for systems in applications since:

1. polynomials might consist of very different numbers of monomials;
2. derivatives generally have less terms than the original polynomials.

Here we describe how we subsequently organized efficient multithreading of coefficient product for systems for which the numbers of terms in polynomials of the system and of the Jacobian vary significantly. Regularity which random generation of systems provides, and which eventually largely equalizes the number of terms in partial derivatives of a given polynomial of the

system is not a requirement for possibility of efficient partitioning of the involved workload. Having the same number of terms in polynomials of the system is not a requirement for this either.

The following natural observation helps to equalize workload between threads as we partition the total collection of the polynomials and polynomials derivatives into p subsets, for assigning to each of threads to work on polynomials and derivatives of one element of the partition. We can largely associate each non-zero coefficient in a polynomial of the system or its Jacobian with one "+" and one " * " extended precision operation. To equalize workload we assign to each thread to perform CP for such sub collection of polynomials and derivatives that the total number of non-zero coefficients in all polynomials and derivatives assigned to one thread is

$$\approx \frac{\text{total \# of non-zero coefficients in all polynomials and derivatives}}{p}.$$

Remark 5.5.1. *Almost no regular assumptions on a system are required for obtaining a good speedup in our multithreaded version of the fourth step - CP - of the Algorithm 5.1.2.*

Indeed, for dimension 40, thus having up to 1640 polynomials in the system and its Jacobian, with the described above principle it would be not possible to equalize workload among 8 threads only in some situations at which there will appear polynomials in the system with excessively many terms or/and when the most of the 1600 polynomials of the Jacobian are identical zeros.

CHAPTER 6

TOWARDS FURTHER EFFICIENCY IN WORKING DIMENSIONS

6.1 Input Processing

The same monomials in F(x) and G(x) might appear numerous times. It is desirable to keep the set of all distinct supports of monomials, **mon**(F,G), to avoid repeated evaluations of the same monomials during path tracking. Having n monomials in **mon**(F,G), and all together m monomials in the polynomials of the homotopy , possibly including duplicates, so $n \leq m$, we construct such a set of supports and the corresponding indexing structure in $O(m \log n)$ time as following:

Algorithm 6.1.1. *1. use STL container **set** to create a binary tree T of unique monomial supports in $O(m \log n)$ time.*

2. construct a sorted array A of unique monomial supports in $O(n \log n)$ time.

3. In $O(m \log n)$ time: for all of m monomials in the homotopy, find indexes of their supports in A to associate the monomials to their coefficients (for further constant access during path tracking).

The main advantage of STL container set is that insertion, retrieval, and deleting an element to/from the container with k elements in the worst case is done asymptotically in $O(\log k)$ time. This is achieved as the implementation of the container along with the other data structures uses balancing binary search trees.

At the first step of the above input processing algorithm, a binary tree T, containing unique copies of monomial supports, is created. A lexicographic order is used on monomial supports to compare two monomials. As m elements are being added to the set, and one element is added in $O(\log n)$ time, the whole set is created in $O(m \log n)$ time.

Once the set of unique monomial supports is created and stored in the binary tree T, we have to set up an array, consisting of these supports, as we need to provide their constant time retrieval during path tracking. If the supports will remain stored in a binary search tree during the path tracking, it will require $O(\log n)$ time to retrieve one support. In that case the achieved time gain in the homotopy input processing by using STL, while creating a set of unique monomials, will be compromised by slower retrieval of degrees, to which variables should be raised at the polynomial evaluation step of each Newton iteration. To create the sorted array A of supports: at each step $i \in 1 \dots n$ we locate and delete the smallest element from T in $O(\log n)$ time, and store it as the next element of the array A.

Lastly we need to create a structure which corresponds to every polynomial of the homotopy an array of indexes of its monomials in the array A. As traversing a j-th polynomial we meet an i-th monomial in it, we find its support in A in $O(\log n)$ time, and associate the index of the found support in A with the monomial. As there is originally m monomials in the homotopy, this is completed in $O(m \log n)$ time.

Input for a homotopy with F(x) of dimension 40 with polynomials of degree 20 is processed in 0.2 seconds by this scheme.

6.2 Quadratic Predictor Versus Secant Predictor

Our multithreaded implementation achieves the better speedups the bigger is the ratio of the 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 inefficient for larger dimensions. We need to come up with a predictor, which would be able to better extrapolate the local intricate behavior of a curve in a multidimensional space. A suitable option for this proved to be the following quadratic predictor:

- Tracking a path, we keep approximate solutions x^*_{prev1}, x^*_{prev}, and x^* 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 quadratic 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 its other coordinates, and all what this computation requires is just computing one value of interpolating three points quadratic polynomial, 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 parallelize quadratic predictor therefore, despite apparently it could be efficiently done with a minimal effort. Fortunately, at the same time as quadratic predictor bears very low computational cost, its use in path tracking for our working dimensions instead of the secant predictor brings dramatic gain in the number of needed corrections. In our experiments we tracked on 8 cores, with complex QD arithmetic, 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. We kept the number of iterations in the Newton correction no more than four, and residual values of magnitudes about 10^{-4}, 10^{-8}, 10^{-16}, and 10^{-32} along each correction. When using the secant predictor, it required 113623 successful corrections , with a running time 26m53.008s, minimal step size 6.10352e-07, and average step size 9.0404e-06, and when using the quadratic predictor, it required 572 successful corrections , with a running time 8.863s, minimal step size 0.00016, and average step size 0.00019. In particular the overall path tracking running time, when using quadratic predictor was about 180 times less than when using the secant predictor. Our numerous experiments with systems of dimensions 20 and higher with monomials of various total degrees had persistently shown that the gain of using the quadratic predictor in the number of corrections and in absolute running time kept to be of the same order or higher in complex QD path tracking. 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.

Remark 6.2.1. *The quadratic predictor provides in our working dimensions very suitable balance between its low computational complexity and reduction in number of corrections it brings, thus ensuring a considerable gain in absolute running time when tracking a path.*

Note that for instance if we would use Euler predictor, as another higher order predictor, we would need to perform an additional work for each prediction which is comparable to the amount of work needed to be done in order to perform one Newton's iteration. At the same time, if the degrees of the system are high, it might be necessary to accomplish all the necessary work, as using Euler predictor, in multiprecision arithmetic. Thus for us one prediction by the quadratic predictor appears to be much less expensive than one prediction by Euler predictor. However to firmly establish which of these two higher order predictors is preferable in path tracking in our working dimensions, for each of the two extended precision levels (DD and QD), we need to establish experimentally in addition what reductions in the number of needed corrections the use of Euler predictor would provide.

The substantial preference of the quadratic predictor over the secant predictor in complex QD path tracking for systems of intermediate dimensions is further illustrated in parts (a) and (b) of Table VI.

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

	# succ.corrs	# corrs	time	aver. 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

(a) Quadratic Predictor

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

	# succ.corrs	# corrs	time	aver. 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	7h16m7.9s	6.73E-06	6.71E-07
St. Dev.	25637.81	25892.2	2h4m31.303s	1.11E-06	5.01E-07

(b) Secant Predictor

TABLE VI

6.3 Computational Results and Conclusions

The testing systems for provided timings are as prescribed in section 4.1.2. The timings in Table VII and Table VIII show reductions in absolute path tracking time first after employing quadratic predictor, and secondly after employing the multithreaded version of the reverse mode

secant predictor, original monomial evaluation

#threads	real	user	sys	speedup
8	4h48m19.6s	38h5m54.5s	1m48.7s	≈ 7

quadratic predictor, original monomial derivatives evaluation

#threads	real	user	sys	speedup
1	7m55.100s	7m54.730s	0m0.354s	1
2	4m1.577s	8m2.678s	0m0.157s	1.967
4	2m3.544s	8m13.064s	0m0.195s	3.846
8	1m6.824s	8m47.341s	0m0.462s	7.11

quadratic predictor, faster monomial derivatives evaluation

#threads	real	user	sys	speedup
1	1m10.244s	1m10.157s	0m0.082s	1
2	0m37.842s	1m15.465s	0m0.145s	1.856
4	0m20.449s	1m21.261s	0m0.262s	3.435
8	0m13.000s	1m41.088s	0m0.618s	5.403

TABLE VII

Tracking a path of Newton homotopy, Complex QD Arithmetic, Dim=20, monomials of degree 10

AD - like algorithm 5.1.2 for evaluating polynomials of intermediate systems and their partial derivatives. Note that the execution time for dimension 20 reduced 4h48m19.6s/13s=1330 times.

Observe that in the Table VIII the speedup on 8 cores, 6.684, as multithreaded version of the Algorithm 5.1.2 is used, is less than speedup, 7.567, on 8 cores when the multithreaded Algorithm 3.1.2 is used. The reason for this reduction is that the speedup for Gaussian

quadratic predictor, original monomial derivatives evaluation

#threads	real	user	sys	speedup
1	244m55.691s	244m48.501s	0m6.621s	1
2	123m1.536s	245m53.987s	0m3.838s	1.991
4	61m53.447s	247m14.921s	0m4.181s	3.958
8	32m22.671s	256m27.142s	0m11.541s	7.567

quadratic predictor, faster monomial derivatives evaluation

#threads	real	user	sys	speedup
1	22m23.220s	22m22.317s	0m0.842s	1
2	11m25.414s	22m49.375s	0m1.012s	1.960
4	5m58.487s	23m50.374s	0m1.941s	3.747
8	3m20.964s	26m23.740s	0m3.836s	6.684

TABLE VIII

Tracking a path of Newton homotopy, Complex QD Arithmetic, Dim=40, monomials of degree 20

Elimination is not good enough yet for dim = 40 on 8 cores. This impacts more the over all speedup as we get better timings for polynomial evaluation.

CHAPTER 7

GRAPHICS PROCESSING UNIT PROSPECTS

We accelerate in this chapter the algorithm 5.1.2 for polynomial evaluation and differentiation on a graphics processing unit. We describe in 1.3.4 the importance of this acceleration. For establishing benchmarks we consider here systems with a fixed number k of variables in monomials, a fixed maximal degree d up to which any of variables can appear in monomials of the system, and a fixed number m of monomials in all polynomials.

7.1 Monomial Prototype Calculation

Algorithm 7.1.1. *The kernel to compute monomial prototypes operates in two stages:*

1. *each of the first n threads of a thread block computes sequentially powers from the 2nd to the $(d-1)$th of one of the n variables;*
2. *each of the threads of a block computes a monomial prototype for one of the monomials of the system, as a product of k quantities computed at the first stage of the kernel.*

Fast Input and SIMT Computations

Storing the values of the successive variables of the system in the successive locations of the global memory enables their coalesced reading into the shared memory by the threads of a warp, thus providing a fast input for the first stage of the kernel.

Both stages of the kernel are largely SIMT (Single Instruction Multiple Thread) routines since at the first stage each busy thread performs the same, $d-1$, number of multiplications,

and at the second stage each thread in each warp also performs the same, $k-1$, number of multiplications.

Shared Memory Use

The precomputed powers of variables are stored at the shared memory of the blocks, since these powers essentially constitute shared input data for the threads of the block while the threads are working on the second stage of the kernel. The powers are stored in shared memory in a two dimensional array POWERS of complex numbers, where the (i,j)th element represents the ith power of the jth variable. Such indexing is aimed at minimizing the number of shared memory bank conflicts at least during the first stage of the kernel, as different threads in a warp, after computing the current power of the variables designated to them, will be writing the power values into different banks of the shared memory.

Constant Memory Use

As the threads of a block perform the second stage of the kernel, each thread computes a product of k quantities. These k quantities were computed at the first stage. As a thread proceeds to the next element in a product, to know what element to access in the shared memory array POWERS, it needs to know which variable and what exponent appears next in the monomial prototype it is computing. The information about positions of variables and their exponents does not change during path tracking and is thus stored in the constant memory of the card. We reserve two arrays of unsigned chars POSITIONS and EXPONENTS in the constant memory to represent this information. Each element in POSITIONS represents a position of a

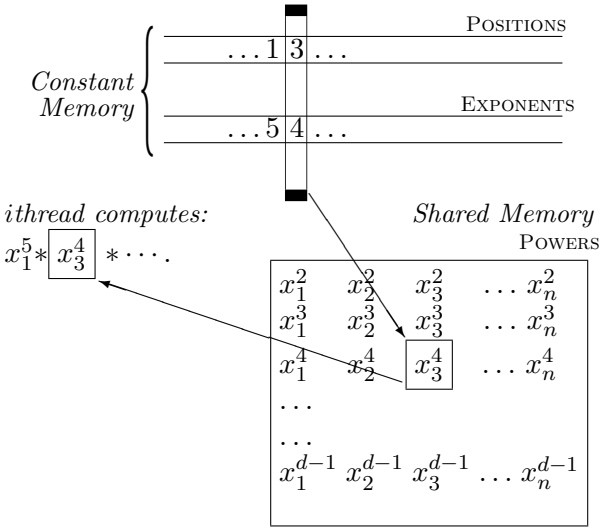

Figure 2. During the second stage of the first kernel, as the i-th thread of the grid computes the monomial prototype of the i-th monomial of the system, it retrieves from the constant memory the variable position and exponent of each next variable power in the product. Thus it knows which entry from the two dimensional array Powers in shared memory to use next in the multiplication. The powers of the variables are computed during the first stage of the kernel by the threads of the block.

variable from 0 to 255 in one of monomials of the system, and the element with the same index in Exponents represents the degree of this variable decreased by one in the same monomial, giving us opportunity to work with variables appearing in degrees up to 255.

Working Dimensions

We need at least about 1,000 monomials to keep all 14 multiprocessors of our card well occupied for the algorithms we consider here, so several warps work on each multiprocessor

simultaneously to hide long latency operations. This and the capacity of the constant memory, 65,536 bytes, prescribes working dimensions for our polynomial evaluation. Those dimensions are ranging from 30 to 40, if we want to keep m, the number of monomials in the polynomial, to be equal roughly to the dimension of the system, and k, the number of variables in the monomial, about half of the dimension. Indeed: for dimension 30 we would have 900 monomials, with a need of $900 \times 2 \times 15 \leq 30,000$ bytes; for dimension 40 we would have 1,600 monomials, with a need of $1,600 \times 2 \times 20 = 64,000$ bytes.

Extensions

We are planning to introduce more compact encodings for storing the positions and exponents of the variables in the constant memory so to be working with higher dimensions. The more compact encodings might introduce some branching for the threads of a warp, after the decoded indexing information would be read from the constant memory into the registers of the block, while each thread in a warp would be encoding the actual position and exponent of the next variable power, which it needs to use for its computations. However the computations, which would follow encodings (the multiplications), where the threads of a warp will join again one path of execution, are supposed to dominate encodings in time, especially if higher precision multiplications would be used. Thus with new ways of decoding, incorporated to store more efficiently monomial information in the constant memory, and employed multiprecision, we hope increase working dimensions for our implementation.

Coalesced Output

After each thread of a block computes its monomial prototype, the successive threads of the block conveniently write their output values (one value per thread) into successive locations of the global memory, thus providing a coalesced output for the kernel.

Alternatives

As an alternative to computing monomial prototypes with the two above stages, one can skip precomputing powers, and assign to each thread all work, which is necessary for computing of assigned to it prototype, to do by itself from scratch. This could be done entirely in registers assigned to a block, without any use of shared memory. However this would introduce branching in execution of threads of a warp when monomials would have different tuples of exponents, and if one would choose that each thread would compute all powers up to $d-1$ for participating in its monomial variables, it would most likely cause extensive repeated exponentiation of the same variables by threads within warps.

The Rationale for Multiple Precomputing of Powers

In our algorithm powers of variables are also computed multiple times – each block of threads computes its own copy of the set of powers from 2 to $d-1$ for all n variables of the system. This might look as a drawback of the algorithm. However, for our working dimensions ranging from 30 to 40, and the number of cores for one multiprocessor, 32, we would need to assign at most two blocks to work on precomputing powers if we want to do it only once, in this case 12 of 14 multiprocessors would be idle during precomputing powers of variables. Also to start using the other 12 multiprocessors for the second stage of computing monomial prototypes, we would

need to write the precomputed powers into the global memory, then to invoke a separate kernel with enough blocks to occupy all multiprocessors, and then threads of each block of the new kernel will access the global memory again for reading the powers of variables stored there. Our algorithm, as an alternative to prompted by the just described two kernels scheme additional time cost for global memory reading and writing, introduces the additional time cost, which is illustrated well by the following example.

Illustration

Consider a system of dimension 32 with 28 monomials in each polynomial. If we will work with blocks of 32 threads, 28 blocks of threads will be launched. Then, in the worst case, if only one block will be occupying one multiprocessor at a time, the execution time for our two-stages kernel will be the same as if one block of 32 threads would be launched two times in a row to compute altogether 64 monomial prototypes. In particular, precomputing powers, despite in fact it would be done 28 times, time-wise would take the same amount of time as it would be done twice. Thus, as within one thread block powers of all variables are computed in parallel, for our example precomputing degrees would take in the worst case the same time as is needed for one core to compute $2(d-2)$ complex multiplications (variables for the monomial prototypes need to be raised up to the power $d-1$, which requires $d-2$ multiplications). The degree d is in most cases not that high (while still allowing high total monomial degrees). Thus multiple precomputing powers of variables in our two stages one kernel algorithm in most cases would compensate for the additional necessary global memory accesses as the powers are precomputed only once, and most likely, even reduce the computational time for precomputing powers.

7.2 Monomial Evaluation and Differentiation of Products of Variables

In this section we describe the implementation of the algorithm to evaluate the example of Speelpenning $x_{i_1}x_{i_2}\cdots x_{i_k}$ and all its derivatives.

Second Kernel Assignment

In our second kernel each thread first computes one monomial and its monomial derivatives. Secondly it multiplies the computed value of the monomial by its coefficient in the hosting that monomial polynomial of the system, as well as it multiplies the values of the computed derivatives of the monomial by their coefficients in the hosting those monomial derivatives polynomials of the Jacobian. Thus this kernel completes computing additive terms of the polynomials of the system and the Jacobian, and the third last kernel only adds appearing in each polynomial terms to finish evaluating polynomials (of the system and of the Jacobian).

An efficient Use of Shared Memory

A thread of the second kernel performs only $5k-4$ multiplications and uses $k+1$ complex double locations of shared memory $L_1, L_2, \ldots, L_{k+1}$ and one variable in registers to perform all the announced above work. The work of a thread of the second kernel is illustrated in Table IX.

To obtain derivatives of Speelpenning product a thread first stores x_{i_1} in the location L_2. Then it computes sequentially, by $k-2$ multiplications, the $k-2$ forward products $x_{i_1}x_{i_2}$,

$x_{i_1}x_{i_2}x_{i_3}$, ... , $x_{i_1}x_{i_2}x_{i_3}\cdots x_{i_{k-1}}$, for each new r ranging from 1 to $k-2$ obtaining the product $x_{i_1}x_{i_2}x_{i_3}\cdots x_{i_{r+1}}$ as $(x_{i_1}x_{i_2}x_{i_3}\cdots x_{i_r})x_{i_{r+1}}$ and storing the newly obtained forward product into location L_{r+2}. Eventually the locations $L_3,\ldots,L_k$ are filled with the $k-2$ obtained forward products. Note that at this point the location L_k contains the derivative of the Speelpenning product with respect to x_{i_k}. In registers of the block we keep the only complex double variable Q to store the current backward product. We initialize Q with x_{i_k}. A thread computes the derivative of the Speelpenning product with respect to $x_{i_{k-1}}$ at L_{k-1} by multiplying stored in that location the forward product $x_{i_1}x_{i_2}x_{i_3}\cdots x_{i_{k-2}}$ by the current value of Q, which is x_{i_k}.

In the next $k-3$ steps, each of which consists of two multiplications, we compute partial derivatives of the Speelpenning product with respect to $x_{i_2}, x_{i_3},\ldots,x_{i_{k-2}}$, and store the computed values in locations $L_2, L_3,\ldots,L_{k-2}$. At the rth step, as r ranges from 1 to $k-3$, the Q represents the backward product $x_{i_k}x_{i_{k-1}}\cdots x_{i_{k-r}}$. At the rth step we first update the value of Q, accordingly to its above definition, by one multiplication as $Q = Q \times x_{i_{k-r}}$. The second multiplication updates the shared memory location L_{k-r-1} as $L_{k-r-1} = L_{k-r-1} \times Q$, so to obtain in this location the partial derivative of Speelpenning product with respect to $x_{i_{k-r-1}}$ as a product of previously stored there forward product $x_{i_1}x_{i_2}x_{i_3}\cdots x_{i_{k-r-2}}$ times the current backward product $x_{i_k}x_{i_{k-1}}\cdots x_{i_{k-r}}$.

Finally we obtain the last yet not obtained partial derivative of Speelpenning product with respect to x_{i_1} at Q, by the product $Q = Q \times x_{i_2}$ and store the obtained value at the shared memory location L_1.

L_1	L_2	L_3	L_4	Q
	x_1			
	x_1	$x_1 * x_2$		
	x_1	x_1x_2	$(x_1x_2) * x_3$	
	x_1	$(x_1x_2) * x_4$	$x_1x_2x_3$	x_4
	$x_1 * (x_3x_4)$	$x_1x_2x_4$	$x_1x_2x_3$	$x_4 * x_3$
$x_2x_3x_4$	$x_1x_3x_4$	$x_1x_2x_4$	$x_1x_2x_3$	$(x_4x_3) * x_2$
$\frac{\partial s}{\partial x_1}$	$\frac{\partial s}{\partial x_2}$	$\frac{\partial s}{\partial x_3}$	$\frac{\partial s}{\partial x_4}$	

L_1	L_2	L_3	L_4	L_5
$\frac{\partial s}{\partial x_1} * \alpha$	$\frac{\partial s}{\partial x_2} * \alpha$	$\frac{\partial s}{\partial x_3} * \alpha$	$\frac{\partial s}{\partial x_4} * \alpha$	
$\frac{1}{3}\frac{\partial \gamma}{\partial x_1}$	$\frac{1}{7}\frac{\partial \gamma}{\partial x_2}$	$\frac{1}{4}\frac{\partial \gamma}{\partial x_3}$	$\frac{1}{5}\frac{\partial \gamma}{\partial x_4}$	
$\frac{1}{3}\frac{\partial \gamma}{\partial x_1}$	$\frac{1}{7}\frac{\partial \gamma}{\partial x_2}$	$\frac{1}{4}\frac{\partial \gamma}{\partial x_3}$	$\frac{1}{5}\frac{\partial \gamma}{\partial x_4}$	$\frac{1}{5}\frac{\partial \gamma}{\partial x_4} * x_4$
$\frac{1}{3}\frac{\partial \gamma}{\partial x_1}$	$\frac{1}{7}\frac{\partial \gamma}{\partial x_2}$	$\frac{1}{4}\frac{\partial \gamma}{\partial x_3}$	$\frac{1}{5}\frac{\partial \gamma}{\partial x_4}$	γ
$\frac{1}{3}\frac{\partial \gamma}{\partial x_1} * (3c)$	$\frac{1}{7}\frac{\partial \gamma}{\partial x_2} * (7c)$	$\frac{1}{4}\frac{\partial \gamma}{\partial x_3} * (4c)$	$\frac{1}{5}\frac{\partial \gamma}{\partial x_4} * (5c)$	$\gamma * c$
$\frac{\partial \beta}{\partial x_1}$	$\frac{\partial \beta}{\partial x_2}$	$\frac{\partial \beta}{\partial x_3}$	$\frac{\partial \beta}{\partial x_4}$	β

TABLE IX

Example: A thread of the second kernel computes the term $\beta = c\,x_1^3x_2^7x_3^4x_4^5$ and its derivatives by $5k - 4 = 5 * 4 - 4 = 16$ multiplications, using $k + 1 = 5$ shared memory locations and one local variable. Here $\gamma := \frac{1}{c}\beta = x_1^3x_2^7x_3^4x_4^5$ is the corresponding monomial, $\alpha := x_1^2x_2^6x_3^3x_4^4$ (the monomial prototype), and $s := x_1x_2x_3x_4$ (the Speelpenning product). Note that the coefficients $(3c), (7c), (4c), (5c)$ are precomputed, only explicit $*$-s stay for performed multiplications.

Now a thread computes monomial derivatives in locations $L_1, L_2, \ldots, L_k$ by multiplying stored in these locations values of derivatives of Speelpenning product by the monomial prototype computed in the first kernel. Then it computes the value of the monomial itself as the product of its monomial derivative with respect to x_{i_k}, stored in L_k times the value of x_{i_k}. It stores the computed monomial value at L_{k+1}. Finally it multiplies each of the values stored in $L_1, L_2, \ldots, L_{k+1}$, i.e.: the values of the monomial and its monomial derivatives, by the corresponding coefficients.

SIMT Computations

As we take the same k — the number of variables in a monomial — for all monomials of the system, each thread of the second kernel will go through the same path of execution for the entire list of instructions of the kernel, which largely amounts to $5k - 4$ complex double multiplications. Thus all 32 threads within each warp will be indeed doing all the prescribed work for the assigned to them 32 monomials in a parallel fashion on an available multiprocessor.

Shared Memory Capacity Considerations

Let B denote the number of threads in a block. In addition to the fast access space of $B(k + 1)$ locations equally divided between threads of a block for storing their intermediate results, as the threads proceed along the kernel, we reserve in shared memory of a block a space for values of all variables of the system. The values of the variables are subject shared use of the threads of each block, as the same variables appear in different monomials. Thus, provided values of successive variables are stored in successive locations of global memory, and working

with $n = 32$, $k = 16$, $B = 32$, we would need to access global memory only once by all threads of a block simultaneously, as each thread would request a value of one variable, to download the values of all 32 variables into the shared memory of the block for their further common use by all threads of the block.

At the same time, if shared memory would not be used for storing values of variables, each thread would need to access global memory at least 16 times to get the values of all appearing in its monomial variables. The shared memory capacity allows us to apply the above algorithm of the second kernel for our working dimensions ranging between 30 and 40 as well as for some larger dimensions. We also could increase precision from double to double double and still work with dimensions up to 70, as long as k is less or equal than a half of dimension. Indeed, each thread would need for treating its monomial $k + 1$ complex double double locations, thus

$$(n/2 + 1) \times 2 \times \texttt{sizeof(double double)}$$

$$\leq (70/2 + 1) \times 2 \times 16 = 1,152$$

bytes in shared memory. To treat 32 monomials by a block of 32 threads we would need then at most $32 \times 1,152 = 36,864$ bytes of shared memory. Adding to this

$$n \times \texttt{sizeof(complex double double)}$$

$$\leq 70 \times 2 \times \texttt{sizeof(double double)}$$

$$= 70 \times 2 \times 16 = 2,240$$

bytes in shared memory for storing values of the variables, we are still $(49,152 - (36,864 + 2,240)) > 10,000$ bytes below the capacity of the shared memory of a block.

Constant Memory Use

Another important note about the memory management is that the array POSITIONS in constant memory, which contains positions indexes of variables in the monomials, and used in the first kernel, is used in this kernel as well, as threads are determining what variable in the shared memory to access as they need to perform each new multiplication while updating their forward and backward products.

Storing Coefficients in the Global Memory

The coefficients are stored in the global memory, since the capacity of the constant memory is exhausted by the variables positions indexes and variables exponents information. As we multiply monomials and their derivatives by the coefficients, we need to read the values of coefficients from the global memory fast. The total number of monomials in the system is $n \times m$. For mapping purposes all the monomials are ordered in a sequence S_m of length $n \times m$. For instance the monomials in S_m might be ordered as following: first m elements of the sequence are the monomials of the first polynomial, the next m elements are the monomials of the second polynomial, and so on. The coefficients are stored during entire path tracking in an array COEFFS of length $n \times m \times (k+1)$, which is the total number of monomials in the system and its Jacobian. The coefficients in COEFFS are stored in the following order:

- The first element of COEFFS is the coefficient of the derivative of the first monomial in S_m with respect to its first variable;
- the second element of Coeffs is the coefficient of the derivative of the second monomial in S_m with respect to its first variable, and so on until
- the nmth element of COEFFS, which is the coefficient of the derivative of the last monomial in S_m with respect to its first variable.
- The next $n \times m$ elements of COEFFS are the coefficients of the derivatives of monomials from S_m, with respect to the monomials second variables, also listed in accordance with order in S_m.

The portions of nm coefficients come in a similar manner until the kth portion of nm coefficients, in which are stored, in order inherited from S_m, the coefficients of monomial derivatives with respect to the monomials kth (last) variables. The last $(k+1)$th portion of the nm coefficients contains actually the coefficients of the system in order prescribed by order in S_m. With this way of storing coefficients, if ith thread of the second kernel is in charge of ith monomial in S_m for each $i = 1, 2, \ldots, nm$, we largely obtain a coalesced access within warps, as threads of a warp prescribed simultaneously to access the coefficients of their monomials or the jth, $j \in \{1, 2, \ldots, k\}$, derivative's coefficients of their monomials in COEFFS.

7.3 Summation of Terms

The Rationale for Introducing the Third Kernel

After multiplying the monomial and their derivatives values by coefficients, which is the last computational step of the second kernel, it is just left to add the corresponding computed additive terms to obtain the values of the polynomials of the system and of the Jacobian. If the size of a thread block used for the execution of the second kernel is smaller than m, then monomials of each polynomial of the system are treated by multiple blocks of the second kernel. In this case, even if some of the involved summations are done yet by the threads of the second kernel, it is necessary to launch another kernel to combine partial sums which are obtained by different blocks of the second kernel, which are working on monomials of the same polynomials. The situation, when the size of a thread block of the second kernel is less than m, is very common for our working dimensions: we try to keep the block size of the second kernel equal to 32, because of described above shared memory limited capacity considerations, on the other hand, we are willing to work with higher dimensions, ranging from 50 to 70, while we want to keep $m \approx n$. Also, computing partial sums for polynomials of the Jacobian by threads of the second kernel would involve branching in execution paths of the threads within warps, as different subsets of variables appear in monomials treated by different threads within a warp. Because of the above reasons we decided to introduce a third kernel, which would perform all involved summations, so to complete obtaining the values of the polynomials, as all multiplicative operations are done by the first two kernels.

SIMT Computations

Each thread of the third kernel sums additive terms of one of $n^2 + n$ polynomials of the combined set of polynomials of the system and the Jacobian matrix. To make each thread to go

through the same execution path, all what we assign to each thread to do during the execution of the kernel is to add exactly m terms. Thus, if a thread computes the value of the derivative of the pth polynomial with respect to x_i, and a jth monomial in the pth polynomial does not contain x_i, the thread which computes the derivative of the pth polynomial with respect to x_i, at the jth step does add to its current partial sum zero – the zero monomial derivative, which we probably never would add in a CPU execution. To ensure this, without introducing any if statements, the output array of the second kernel in the global memory along with its meaningful $nm(k+1)$ locations (the number of monomials and monomial derivatives of the system) contains also $(n^2+n)m - nm(k+1)$ locations, the values at which are originally set and kept to store zero values along the entire path tracking. These zero locations represent the zero monomial derivatives as in the described above situation. We also wish that the threads within warps of the third kernel for each step j, $j = 1, 2 \ldots m$ would perform a coalesced reading of the input data entries. To allow coalesced reading of the values of monomials and their derivatives by the threads of the third kernel, and to introduce the $(n^2+n)m - nm(k+1)$ zero monomial derivatives, the output of the second kernel is stored in the global memory in array MONS in the format we explain next.

The Output of the Second Kernel and Coalescing

The size of the array MONS is $(n^2+n)m$, representing the terms in n^2+n summations, m terms each. The first n^2+n elements of the array represent the first terms in each of n^2+n summations (polynomials). In particular, these first n^2+n elements are: the first n elements are the first monomials of the polynomials of the system, the second n elements are the derivatives

of the first monomials with respect to x_1, the third n elements are the derivatives of the first monomials with respect to x_2, and so on until the $(n+1)$th n elements, which are the derivatives of the first monomials with respect to x_n. The second n^2+n elements represent the second terms in each of n^2+n summations, and again the first n elements of them represent the second monomials of the polynomials of the system, and the next n^2 elements represent the partial derivatives of the second monomials of the system, listed in the same order as are listed the derivatives of the first monomials. In general the jth n^2+n elements represent jth monomials of the polynomials of the system and their partial derivatives listed in the same order as listed the first monomials of the system and their partial derivatives at the first n^2+n elements of the array.

For simplicity in this description we assumed that the number B of threads in a block, the block size, divides n^2+n. Now if we launch $(n^2+n)/B$ blocks, with a thread $t = BlockId \times B + ThreadId$ computing the sum: $\sum_{j=0}^{m-1} \textsc{Mons}[t + j(n^2+n)]$, the obtained sums will represent the values of polynomials of the system and of the Jacobian, while access to the elements of Mons will be coalesced within warps at each step $j = 0, 1, \ldots, m-1$ of the summation. To create the array Mons in such a format, we had to make the threads of the second kernel to output the values of monomials and their derivatives not in a coalesced way. However there was a tradeoff:

- either to make the output of the second kernel coalesced and then the input of the third kernel could not be accessed in a coalesced way,

- or as we chose to provide ability for the threads of the third kernel to read the input data in a coalesced way, and paid the price of not coalesced writing of the output of the second kernel.

7.4 Computational Experiments

Our computations are done on a HP Z800 workstation, running Red Hat Enterprise Linux Workstation release 6.1. The CPU is an Intel Xeon X5690 at 3.47 Ghz. The processor clock of the NVIDIA Tesla C2050 Computing Processor runs at 1147 Mhz. The graphics card has 14 multiprocessors, each with 32 cores, for a total of 448 cores. As the clock speed of the GPU is a third of the clock speed of the CPU, we hope to achieve a double digit speedup. We used the NVIDIA CUDA compiler driver `nvcc`, release 4.0, V0.2.1221.

In parts (a) and (b) of Table X we list results of our preliminary implementation. The number of threads in each block was 32 for all three kernels to evaluate a system and its Jacobian matrix of dimension 32. Generating 32 monomials per polynomial leads to 1,024 monomial in total. Increasing the number of monomials to 2,048 in the part (b) of Table X would have yielded a speedup of more than 20, but the capacity of the constant memory was not sufficient to hold the exponents and positions of all 2,048 monomials. For larger systems, we are planning to upgrade our preliminary implementation with a better compression strategy (instead of the current `char` used for each exponent).

We obtained good speedups for randomly generated polynomial systems of dimension 32 (the warp size) and fixed number of monomials per polynomial. For a double digit speedup, we need to have at least 1,000 monomials. The speedups improve for higher degree monomials.

#monomials	Tesla C2050	1 CPU core	speedup
704	14.514 sec	1min 50.9 sec	7.60
1024	15.265 sec	2min 39.3 sec	10.44
1536	17.000 sec	3min 58.7 sec	14.04

(a) Wall clock times and speedups for 100,000 evaluations of a polynomial system and its Jacobian matrix of dimension 32. Each monomial has 9 variables occurring with nonzero power. of at most 2.

#monomials	Tesla C2050	1 CPU core	speedup
704	19.068 sec	3min 16.9 sec	10.33
1024	20.800 sec	4min 43.3 sec	13.62
1536	21.763 sec	7min 05.8 sec	19.56

(b) Wall clock times and speedups for 100,000 evaluations of a polynomial system and its Jacobian matrix of dimension 32. Each monomial has 16 variables occurring with nonzero power of at most 10.

TABLE X

CHAPTER 8

CONCLUSIONS

The improvements of the Chapters 5 and 6 turned our scalable multithreaded DD/QD path tracker into an efficient, ready for use for needs of Algebraic Geometry, application.

Both multicore shared memory processing and GPU are capable to speed up the AD-like Algorithm 5.1.2 for polynomial evaluation and differentiation for systems of intermediate dimensions and degrees with reasonably irregular data. Each of the two technologies provides such capability by its own means.

For multicore processors it appears possible to estimate overall relatively big workload of the involved tasks, and roughly equally distribute the workloads between relatively small amount of computing units. As illustrated in sections 5.3 - 5.5, that is possible to do for systems of intermediate sizes with no strict regularity conditions imposed. As at the same time multicore technology related interaction overhead is compensated by complexity of the obtained subproblems, especially as multiprecision is used, and virtually no memory capacity limitations were met, multiple cores definitely appear to be a suitable technology for parallelizing on it this AD-like algorithm.

On GPU we have shown that for systems of intermediate sizes it is possible to divide the stages of the Algorithm 5.1.2 to that many small subproblems, that first: all cores of the card are kept busy, secondly: the workloads of the subproblems are not too big so the involved memory

capacities are not exceeded as these subproblems are executed; and thirdly the workloads are not too small so occurrences of expensive data transfers between different levels of memory hierarchy of the card are minimized.

For establishing benchmarks for our GPU version of the Algorithm 5.1.2, we imposed some regularity assumptions on considered systems. However we believe that we can significantly weaken them. Its memory limitations and preference to work with structured data, GPU compensates with excessive amount of cores ready to work on parts of a problem simultaneously. Thus, as many small subtasks are assigned to many cores for execution, it is not that important that their workload will be indeed well balanced. What is important is that the workload between big collection of subtasks will be balanced to the extent that the complexity of the biggest of subtasks would be yet much smaller than the complexity of the original entire task. The involved workload of the AD-like polynomial evaluation and differentiation for systems of intermediate sizes is possible to partition this way in a vast variety of settings. Thus we assume that GPU will be capable to accelerate efficiently polynomial evaluation and differentiation for much less regular systems than those considered in Chapter 7.

CITED LITERATURE

1. Allgower, E. and Georg, K.: Introduction to Numerical Continuation Methods, volume 45 of Classics in Applied Mathematics. SIAM, 2003.

2. Li, T.: Numerical solution of polynomial systems by homotopy continuation methods. In Handbook of Numerical Analysis. Volume XI. Special Volume: Foundations of Computational Mathematics, ed. F. Cucker, pages 209–304. North-Holland, 2003.

3. Morgan, A.: Solving polynomial systems using continuation for engineering and scientific problems. Prentice-Hall, 1987. Volume 57 of Classics in Applied Mathematics Series, SIAM 2009.

4. Watson, L.: Probability-one homotopies in computational science. Journal of Computational and Applied Mathematics, 140(1&2):785–807, 2002.

5. Leykin, A.: Numerical algebraic geometry. The Journal of Software for Algebra and Geometry: Macaulay2, 3:5–10, 2011.

6. Sommese, A., Verschelde, J., and Wampler, C.: Introduction to numerical algebraic geometry. In Solving Polynomial Equations. Foundations, Algorithms and Applications, volume 14 of Algorithms and Computation in Mathematics, pages 301–337. Springer-Verlag, 2005.

7. Sommese, A. and Wampler, C.: The Numerical solution of systems of polynomials arising in engineering and science. World Scientific, 2005.

8. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongarra, J.: MPI - The Complete Reference Volume 1, The MPI Core. Massachusetts Institute of Technology, second edition, 1998.

9. Bates, D., Hauenstein, J., and Sommese, A.: A parallel endgame. In Randomization, Relaxation, and Complexity in Polynomial Equation Solving, eds. L. Gurvits, P. Pébay, J. Rojas, and D. Thompson, volume 556 of Contemporary Mathematics, pages 25–35. AMS, 2011.

10. Li, T. and Tsai, C.-H.: HOM4PS-2.0para: Parallelization of HOM4PS-2.0 for solving polynomial systems. Parallel Computing, 35(4):226–238, 2009.

11. Gunji, T., Kim, S., Fujisawa, K., and Kojima, M.: PHoMpara – parallel implementation of the Polyhedral Homotopy continuation Method for polynomial systems. Computing, 77(4):387–411, 2006.

12. Su, H.-J., McCarthy, J., Sosonkina, M., and Watson, L.: Algorithm 857: POLSYS_GLP: A parallel general linear product homotopy code for solving polynomial systems of equations. ACM Transactions on Mathematical Software, 32(4):561–579, 2006.

13. Verschelde, J.: Algorithm 795: PHCpack: A general-purpose solver for polynomial systems by homotopy continuation. ACM Transactions on Mathematical Software, 25(2):251–276, 1999. Software available at `http://www.math.uic.edu/~jan/download.html`.

14. Guan, Y. and Verschelde, J.: 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.

15. Leykin, A. and Verschelde, J.: Factoring solution sets of polynomial systems in parallel. In Proceedings of the 2005 International Conference on Parallel Processing Workshops. 14-17 June 2005. Oslo, Norway. High Performance Scientific and Engineering Computing, eds. T. Skeie and C.-S. Yang, pages 173–180. IEEE Computer Society, 2005.

16. Leykin, A. and Verschelde, J.: 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.

17. Leykin, A., Verschelde, J., and Zhuang, Y.: Parallel homotopy algorithms to solve polynomial systems. In Proceedings of ICMS 2006, eds. N. Takayama and A. Iglesias, volume 4151 of Lecture Notes in Computer Science, pages 225–234. Springer-Verlag, 2006.

18. Verschelde, J. and Wang, Y.: Computing feedback laws for linear systems with a parallel Pieri homotopy. In Proceedings of the 2004 International Conference on Parallel Processing Workshops, 15-18 August 2004, Montreal, Quebec, Canada. High Performance Scientific and Engineering Computing, ed. Y. Yang, pages 222–229. IEEE Computer Society, 2004.

19. Verschelde, J. and Zhuang, Y.: Parallel implementation of the polyhedral homotopy method. In Proceedings of the 2006 International Conference on Parallel Processing Workshops. 14-18 Augustus 2006. Columbus, Ohio. High Performance Scientific and Engineering Computing, eds. T. Pinkston and F. Ozguner, pages 481–488. IEEE Computer Society, 2006.

20. Allison, D., Chakraborty, A., and Watson, L.: Granularity issues for solving polynomial systems via globally convergent algorithms on a hypercube. Journal of Supercomputing, 3(1):5–20, 1989.

21. Chakraborty, A., Allison, D., Ribbens, C., and Watson, L.: The parallel complexity of embedding algorithms for the solution of systems of nonlinear equations. IEEE Transactions on Parallel and Distributed Systems, 4(4):458–465, 1993.

22. Dekker, T.: A floating-point technique for extending the available precision. Numerische Mathematik, 18(3):224–242, 1971.

23. Priest, D.: On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations. Doctoral dissertation, University of California at Berkeley, 1992. `ftp://ftp.icsi.berkeley.edu/pub/theory/ priest-thesis.ps.Z`.

24. Rump, S.: Verification methods: Rigorous results using floating-point arithmetic. Acta Numerica, 19:287449, 2010.

25. Shewchuk, J.: Adaptive precision floating-point arithmetic and fast robust geometric predicates. Discrete and Computational Geometry, 18(3):305–363, 1997.

26. Hida, Y., Li, X., and Bailey, D.: 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`.

27. Verschelde, J. and Yoffe, G.: Polynomial homotopies on multicore workstations. In Proceedings of the 4th International Workshop on Parallel Symbolic Computation (PASCO 2010), July 21-23 2010, Grenoble, France, eds. M. Maza and J.-L. Roch, pages 131–140. ACM, 2010.

28. Wilkinson, B. and Allen, M.: Parallel Programming. Techniques and Applications Using Networked Workstations and Parallel Computers. Prentice Hall, 2 edition, 2005.

29. Verschelde, J. and Yoffe, G.: Quality up in polynomial homotopy continuation by multithreaded path tracking. Preprint `arXiv:1109.0545v1 [cs.DC] 2 Sep 2011`.

30. Griewank, A. and Walther, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. SIAM, second edition, 2008.

31. Kirk, D. and Hwu, W.: Programming Massively Parallel Processors. A Hands-on Approach. Morgan Kaufmann, 2010.

32. Lu, M., He, B., and Luo, Q.: Supporting extended precision on graphics processors. In Proceedings of the Sixth International Workshop on Data Management on New Hardware (DaMoN 2010), June 7, 2010, Indianapolis, Indiana, eds. A. Ailamaki and P. Boncz, pages 19–26, 2010. Software at `http://code.google.com/p/gpugrpec/`.

33. Buttari, A., Langou, J., Kurzak, J., and Dongarra, J.: A class of parallel tiled linear algebra algorithms for multicore architectures. Parallel Computing, 35:38–53, 2009.

34. Dongarra, J., Duff, I. Sorensen, D., and Van Der Vorst, H.: Nimerical Linear Algebra for High-Performance Computers. SIAM, 1998.

35. Akl, S.: Superlinear performance in real-time parallel computation. The Journal of Supercomputing, 29(1):89–111, 2004.

36. Bischof, C., Guertler, N., Kowartz, A., and Walther, A.: Parallel reverse mode automatic differentiation for OpenMP programs with ADOL-C. In Advances in Automatic Differentiation, eds. C. Bischof, H. Bücker, P. Hovland, U. Naumann, and J. Utke, pages 163–173. Springer-Verlag, 2008.

37. Chow, S.-N., Ni, L., and Shen, Y.-Q.: A parallel homotopy method for solving a system of polynomial equations. In Parallel processing for scientific computing, ed. G. Rodrigue, pages 121–125. SIAM, 1989.

38. Emeliyanenko, P.: Efficient multiplication of polynomials on graphics hardware. In Advanced Parallel Processing Technologies. 8th International Symposium, APPT 2009, Rapperswil, Switzerland, August 2009, eds. Y. Dou, R. Gruber, and J. Joller, volume 5737 of Lecture Notes in Computer Science, pages 134–149. Springer-Verlag, 2009.

39. Emeliyanenko, P.: A complete modular resultant algorithm targeted for realization on graphics hardware. In Proceedings of the 4th International Workshop on Parallel Symbolic Computation (PASCO 2010), July 21-23 2010, Grenoble, France, eds. M. Maza and J.-L. Roch, pages 35–43. ACM, 2010.

40. Emeliyanenko, P.: High-performance polynomial GCD computations on graphics processors. In Proceedings of the 2011 International Conference on High Performance Computing & Simulation (HPCS 2011), eds. W. Smari and J. McIntire, pages 215–224. IEEE, 2011.

41. Grabner, M., Pock, T., Gross, T., and Kainz, B.: Automatic differentiation for GPU-accelerated 2D/3D registration. In Advances in Automatic Differentiation, eds. C. Bischof, H. Bücker, P. Hovland, U. Naumann, and J. Utke, pages 259–269. Springer-Verlag, 2008.

42. Harimoto, S. and Watson, L.: The granularity of homotopy algorithms for polynomial systems of equations. In Parallel Processing for Scientific Computing, ed. G. Rodrigue, pages 115–120. SIAM, 1989.

43. Hwu (editor), W.: GPU Computing Gems: Emerald Edition. Morgan Kaufmann, 2011.

44. Kojima, M.: 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.

45. Maza, M. and Pan, W.: Fast polynomial multiplication on a GPU. Journal of Physics: Conference Series, 256, 2010. High Performance Computing Symposium (HPCS2010), 5-9 June 2010, Victoria College, University of Toronto, Canada.

46. Maza, M. and Pan, W.: Solving bivariate polynomial systems on a GPU. In HPCS 2011, Montreal, 15-17 June 2011, to appear in Journal of Physics: Conference Series.

47. NVIDIA: NVIDIA CUDA Programming Guide. Version 3.0. 2010.

VITA

Education

- Ph.D. Applied Mathematics, University of Illinois at Chicago, 2007-2012

- M.S. Mathematics, Weizmann Institute of Science 2002
- B.S. Mathematics, Bar Ilan University, Israel 2000

Employment

- **NAVTEQ**| February 2011- August 2011
  Consultant | Data Mining: Cluster Analysis

- **Argonne National Laboratory, MCS**|May 2010-August 2010
  Intern — Applied Numerical Analysis
  Adviser: Barry Smith
  Research: Multi-grid methods for solving partial differential equations which produce algebraic systems subject of finite difference discretization

- **University of Illinois at Chicago** | June 2007- May 2012
  Research Assistant, Teaching Assistant | Math and Computer Science Department
  Thesis Adviser: Jan Verschelde
  Research: Development of a parallel version of PHCpack (an open source software for solving systems of polynomial equations)
- **Syracuse University**|June 2005-May 2007
  TA | Math Department
  Lecturer for Multivariate Calculus,
  Teaching assistant for mathematical courses

Publications

- Verschelde, J. and Yoffe, G.: Evaluating polynomials in several variables and their derivatives on a GPU computing processor. To appear in Proceeding of the13th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC-12) to be held May 21-25, 2012, Shanghai, China
- Verschelde, J. and Yoffe, G.: Quality up in polynomial homotopy continuation by multithreaded path tracking. Preprint `arXiv:1109.0545v1 [cs.DC] 2 Sep 2011`.

- Verschelde, J. and Yoffe, G.: Polynomial homotopies on multicore workstations. In Proceedings of the 4th International Workshop on Parallel Symbolic Computation (PASCO 2010), July 21-23 2010, Grenoble, France, eds. M. Maza and J.-L. Roch, pages 131-140. ACM, 2010.