A Parallel-Computing Solution for Optimization of Polynomials

Matthew M. Peet[1] and Yulia V. Peet[2]

Abstract

In this paper, we consider optimization of polynomials in a parallel computing environment. Algorithms for optimization of polynomials can be used to solve NP-hard control problems such as stability of nonlinear and delayed systems. Unfortunately, the high computational costs of current algorithms such as sum-of-squares has limited its use to relatively small problems. In this paper we review several results on polynomial representation which and show that these results can be used to develop an algorithm for polynomial optimization with a naturally parallel structure. In particular, we design and implement a massively parallel algorithm in MPI which tests positivity of polynomials. Test results confirm that the implementation has high efficiency with relatively low overhead.

1 Introduction

Polynomial computing is the manipulation of polynomials by computational algorithms. One aspect of polynomial computing - optimization of polynomials (typically positive polynomials) - is a rapidly developing field of research. The interest in optimization of polynomials has been driven primarily by the ability of these algorithms to effectively handle difficult problems in control of nonlinear, delayed and partial-differential systems [16, 17, 15]. The most serious problem with optimization using polynomials as opposed to matrices is that there is no efficient way of telling when a polynomial is positive. Computationally, the question of polynomial positivity is intractable (NP-hard) [3]. However, there are several classical alternatives that we can use. The alternative that has been most popular lately is to consider squared polynomials. This approach is extremely attractive because there is a one-to-one correspondence between positive matrices and squared polynomials of a given degree.

The problem with this scenario is that while Moore's law says we should expect exponentially increasing speed, actual commercial CPU speeds have saturated. In fact, a consensus has developed that it will not be possible to increase the maximum single-core processor speed by more than an order of magnitude over the next 20-30 years [8]. Instead, manufacturers have embraced the idea of creating massively multi-core chips, wherein hundreds or thousands of processing centers are contained on a single chip. Multi-core processing is similar to what is commonly understood to be shared-memory parallel processing, only with reduced latency and faster message-passing. Intel refers to this paradigm as "cloud on a chip" computing [11] and has recently rolled out a prototype with 84 cores. It is likely that the number of per-chip cores will continue to increase at an exponential rate. The result is that while the newest chips have dramatically increased operations per second, they have actually decreased in per-core clock speed. In addition to recent changes in the CPU market, distributed computing and grid computing [7] are computing paradigms which have developed over the last 20 years or so. These models describe networks of computational resources connected by a relatively high-latency network such as the internet. Hundreds of these computational networks exist, with prominent examples being CONDOR [12] and the SETI@home project which had the peak resources of 5.2 million users. More recently, GPU computing has already made TFLOP processing available at desktop computing prices [13]. In this scenario graphics card companies such as NVidea have developed large arrays of relatively cheap processors. Finally we observe that the gold standard of computational tools have always been cluster computing and supercomputing. These architectures achieve high computational power by combining large networks of processors with relatively low latency; e.g. the IBM roadrunner supercomputer alone has 122,400 cores. To summarize, we observe that the world has vast reserves of computational power. The real problem with computing is not the availability of resources, but rather the lack of algorithms capable of efficiently utilizing those resources.

If we are to address the optimization of polynomials in a massively parallel world, the first thing to consider is the problems of linear programming and semidefinite programming. Because of their ubiquitous presence in algorithms for controller design and analysis, an efficient parallel implementation of a linear programming or semidefinite programming solver would solve most of the problems in control. Unfortunately, however, it has been shown that both linear programming and semidefinite programming are inherently sequential problems [9]. These means that there is an irreducibly sequential com-

[1]M. M. Peet is an Assistant Professor of Aerospace Engineering at the Illinois Institute of Technology, Chicago, IL. mpeet@iit.edu.

[2]Y. V. Peet is a Computational Scientist at Argonne National Laboratory, Argonne, IL peet@mcs.anl.gov.

ponent to these problems which will dominate computation as the number of processors increases. Naturally, there do exist parallel versions of semidefinite programming solvers. See [1, 4, 23] for several excellent implementations. These solvers have been shown to work exceptionally well for reasonably large numbers of processors. As computation becomes increasingly parallel, however, we expect that Amdahl's Law of diminishing returns will dominate these algorithms.

2 Notation and Background

Let $\mathbb{R}^n$ denote the n-dimensional real vectors and $\mathbb{R}^{n\times m}$ the n by m real matrices. We denote by $\mathbb{R}[x]$ the ring of real-valued polynomials in variables x. $\mathbb{S}^n \subset \mathbb{R}^{n\times n}$ is the subspace of symmetric matrices. $\mathbb{N}$ is the natural numbers. For $M \in \mathbb{S}$, $M \geq 0$ ($M > 0$) denotes that M is a positive semidefinite (definite) matrix. For $M \in \mathbb{R}^{n\times m}$, we denote by $\lambda_{\max}(M)$ and $\lambda_{\min}(M)$, the maximum and minimum eigenvalues of M, and $\texttt{col}(M)$ is the vector of columns of M. A monomial, $m(x)$ is a polynomial in $x \in \mathbb{R}^n$ with a single term and is represented as x^α where $\alpha \in \mathbb{N}^n$ and $x^\alpha = \Pi_{i=1}^n x_i^{\alpha_i}$.

2.1 Polya's Theorem and Variations

The results covered in this section have appeared in multiple sources. In particular, we refer the reader to the survey [19].

2.1.1 Sum-of-Squares Polynomials

The question of determining positivity of a polynomial has a long history, only parts of which will be recounted here. By far the most commonly used method for proving positivity of a polynomial is to construct a representation of the polynomial using sums of squares. Since it is known that any squared, real-valued function is positive, a function which is composed of squares is naturally positive. While it is known that there exist positive polynomials that are not sum-of-squares, it is also known that for an arbitrary positive homogeneous polynomial $p\,(x)$, there exists a $d > 0$ such that

$$\left(\sum_i x_i^2\right)^d p\,(x)$$

is a sum-of-squares polynomial. For polynomials which are positive on a subset of the real numbers there are alternative representations. For example, it is obvious that a polynomial $p\,(x)$ which is positive for all x such that $g\,(x) \geq 0$ will have a representation as

$$p\,(x) = s_0\,(x) + g\,(x) s_1\,(x)$$

for positive functions s_0 and s_1. Under certain conditions on g, we know that s_0 and s_1 can be themselves sum-of-squares polynomials. Results of this type are called Positivstellensatz results and some important examples include [22, 21, 18].

Although the method of sum-of-squares is well-developed, it has the disadvantage that the structure of the problem does not admit an easily distributed computational solution. For this reason, we examine a different approach to the question of polynomial positivity.

2.1.2 Polya's Theorem

Polya's Theorem is an alternative to existing sum-of-squares algorithms for polynomial optimization. Unchanged since 1928, the original version of Polya's theorem states

Theorem 1. *The homogeneous polynomial $f(x) > 0$ for all $x_i \geq 0$, $x \neq 0$ if and only if for all p sufficiently large,*

$$g(x) := \left(\sum_{i=1}^n x_i\right)^p f(x)$$

has all positive coefficients.

Modern versions consider the unit simplex $\Delta := \{x : x \geq 0,\ \sum_i x_i = 1\}$ and use matrix-valued polynomials [20]. It is important to note that for matrix-valued polynomials, positivity requires that all coefficients of g must be *positive definite* (in the symmetric matrix sense). Polya himself noted the algorithmic nature of the result in [10], wherein it is stated:

> "The theorem gives a systematic process for deciding whether a given form F is strictly positive for positive x. We multiply repeatedly by $\sum x$, and, if the form is positive, we shall sooner or later obtain a form with positive coefficients."

This comment describes a sequential relaxation algorithm, wherein we have a sequence of sufficient tractable conditions which are shown to be necessary as the sequence goes to infinity. Furthermore, recent research has shown that we often have an upper bound for the number of iterations. Such a bound on the exponent p is referred to as the Polya exponent. This bound has been developed in a number of sources including, recently, in [6, 5]. A matrix version of this bound is used in [20].

Theorem 2. *If $p > \frac{U(f)}{L(f)}$, then $(\sum x_i)^p F(x)$ has all positive definite coefficients, where*

$$U(F) = \max_{x\in\Delta} \lambda_{\max}(F(x)) \ \ \textit{and} \ \ L(F) = \min_{x\in\Delta} \lambda_{\min}(F(x))$$

Note that computing these bounds is itself intractable.

In the paper [14], a practical centralized implementation of Polya's algorithm for polynomial optimization is given and a number of numerical tests are conducted to demonstrate the efficacy of the approach.

2.2 Complexity in Parallel Machines

Complexity theory for parallel computation has been studied in some depth. Although the notions of complexity are not as uniform as for sequential computing, there

is a well-established notion of P-completeness, which states that a problem is in NC if there exist integers c and d such that a problem can be solved in $O(log(n)^c)$ time on $O(n^d)$ processors [9]. In rough terms, the per-core complexity does not depend strongly on the problem size and the number of processors grows in a polynomial manner. The class of P-complete problems is the class of problems which are all equivalent up to an NC reduction. A problem which is at least as hard as any P-complete problem is P-hard. It is generally believed that $NC \neq P$. That is, it is thought that NC is the set of parallelizable problems and P-complete is the class of inherently sequential problems. The most important conclusion for the purposes of this paper is that linear programming is P-complete [9], which implies that semidefinite programming is P-hard; i.e. general-purpose semidefinite programming can never be efficiently parallelized. Another notion of complexity is PAR, defined to be the class of problems which can be solved in polynomial time using an exponentially-bounded number of processors. Clearly the complexity class PAR contains P. Further, it has been shown that the class PAR contains NP [3]. Additionally, polynomial positivity is in PAR [3]. Thus we have a reasonable expectation that massively parallel algorithms are capable of solving polynomial computing in polynomial time.

3 An Algorithm for Positivity

Based on Polya's theorem, we introduce the following idealized algorithm. For brevity, we only include the case of a homogeneous polynomial with symmetric matrix-valued coefficients. However, the algorithm also applies to scalar-valued homogeneous polynomials and can be easily modified to accept arbitrary polynomials.

A critical aspect of this algorithm is the choice of d. This is both the degree of the refutation and the number of steps in the algorithm. Although there are upper bounds for this degree (e.g. Theorem 2) computing these bounds is itself intractable. However, given a desired level of accuracy, ϵ, we can use the following lemma.

Lemma 3. *Let $d > U(f)/\epsilon$ where $U(f)$ is as in Theorem 2. If $(\sum_i x_i)^d f(x)$ does not have all positive coefficients, then there exists some $x \in \Delta$ such that $f(x) \leq \epsilon$. Conversely, if $(\sum_i x_i)^d f(x)$ has all positive coefficients, then $f(x) \geq 0$ for all $x \in \Delta$.*

Complexity Analysis In a perfectly distributed implementation, the number of processors required at each iteration is given by the number of monomials in $Z(x)$. For a homogeneous polynomial, this is

$$N_d = f(n,d) = \begin{cases} 0 & \text{for } n = 0 \\ \frac{(d+n-1)!}{d!(n-1)!} & \text{for } n > 0, \end{cases} \qquad (1)$$

where n is the number of variables and d is the degree of the polynomial. Thus the number of processors grows in a non-polynomial manner in n and d jointly (although more slowly in n or d individually.). However, the ***per-core*** complexity is simply $O(dm^3)$ where the coefficients are in $\mathbb{R}^{m\times m}$. If m is relatively large, the positivity test can be parallelized [2] to yield a per-core complexity of $O(d \log^2 m)$ with the gain in the number of processors required being $O(m^{3.5})$. The per-core communication complexity is also simply $2 \times n \times d$. Thus both per-core communication and computational complexity is low, while the number of processors required is high. This makes the algorithm ideally suited for distributed computation.

4 An Implementation using MPI

The simple algorithm proposed in the previous section is difficult to implement in practice for several reasons. In the first place, in a decentralized environment, the communication pathways change at every iteration depending on the distribution of monomials to processors. Thus, in order to perform the required calculations, there must exist a method for determining where to send data without consulting or searching a centralized list. In the second place, an unbounded number of processors needed for implementation of Algorithm 1 might not be readily available for users utilizing finite resources.

In this section we describe a practical implementation of Polya's algorithm and detail the solution to several technical difficulties which were encountered. In particular, we describe an efficient method for computation in the presence of a fixed number of processors with no centralized memory storage. In addition, we present a method for indexing calculations and determining communication pathways in a distributed computing environment.

4.1 Overview

In most parallel protocols, even those running on the largest massively-parallel computers, one needs to specify the number of required processors during the initial job submission, and this number, although it can be very large, is nonetheless fixed. Protocols which can dynamically delete or add processors to the resources allocated to a specific job are not readily available. In order to fully realize the potential of a finite number of processors, we describe a parallel algorithm designed specifically for a fixed (relatively small) number of processors.

In this section, we assume that we are given a bound, $d_{\max}$ on maximum degree of the refutation. This corresponds to $d_{\max} - d$ iterations, where d is the degree of the given polynomial. The input is given by a homogeneous polynomial $h(x)$ of degree d in n variables and is specified in the form $h(x) = c^T Z(x)$, where $Z(x)$ is the vector of all possible monomials of degree d in variables x, in lexicographical order, and $c = (c_1, c_2, \ldots c_n)$ is the array of the corresponding coefficients. For a given degree, d, and number of variables, n, we denote by N_d the number of all possible monomials of degree d in n variables,

Input: A homogeneous matrix-valued polynomial in the form $h(x) = c^T Z(x)$, where $Z(x)$ is the vector of all monomials with appropriate degree, in lexicographical order; The number of desired iterations, d.
foreach *Processor* i **do**
  Associate the processor i with monomial $Z_i(x)$ and initialize $C(i) = c_i$.
  Let I_i be the indices of the monomials obtained by multiplying $Z_i(x)$ by $\sum_j x_j$.
  Let J_i be the indices of the monomials obtained by dividing $Z_i(x)$ by any x_j.
end
for $k = 1$ *to* d **do**
  foreach *Processor* i **do**
    Processor i sends $C(i)$ to each element of I_i.
    Processor i receives $C(j)$ from each element $j \in J_i$ and updates $C(i) = \sum_{j \in J_i} C(j)$.
  end
end
if $C(i) \geq 0$ *for all* i **then**
  h is positive
end

Algorithm 1: A parallel algorithm for determining polynomial positivity.

given by Eq. (1). We denote the number of processors by N_p. In the algorithm, we represent $Z(x)$ using the matrix $Z^{N_d \times n}$. Note that the matrix Z is entirely different from the vector-valued function $Z(x)$. Here the element $Z_{i,j}$ is the degree in variable j of the i^{th} monomial of degree d in lexicographical ordering. Thus the i^{th} row of Z is $\alpha(i) \in \mathbb{N}^n$ where $x^{\alpha(i)}$ is the i^{th} monomial of degree d in lexicographical ordering.

4.2 Processor Assignment and Load Balancing

In order to implement the algorithm on N_p processors, one needs to distribute the coefficient vector c and monomial matrix Z among those processors. At each iteration step a new polynomial $\tilde{h}(x) = h(x) \cdot \sum_{j=1}^{n} x_j$ is formed, with the new set of coefficients $\tilde{c}$ containing more entries $N_{d+1} = (d+n)/(d+1) \cdot N_d$. Thus, the new coefficients and the corresponding monomials have to be redistributed among the processors at every iteration step.

One of the basic technical requirements for a working parallel implementation is an efficient, decentralized way to determine at each iteration step the set of processors $\mathcal{P}(I) \subset \{0, \ldots, N_p - 1\}$, which, for each $I \in \{1, \ldots, N_d\}$, will receive the coefficient c_I. In the present algorithm, we establish a simple correspondence between the lexicographical index I of the coefficient c_I (as given by its associated monomial) and the index $p\,(I) \in \{0, \ldots, N_p - 1\}$ of the processor containing it, $p\,(I) = \texttt{floor}(I/N_c)$, where $N_c = \texttt{ceil}(N_d/N_p)$

In order to determine the set of processors $\mathcal{P}(I)$, one needs to compute the corresponding set $J(I)$ of lexicographical indices of the new monomials which are created by multiplication of $x^{\alpha(I)}$ with the term $\sum_{j=1}^{n} x_j$. At each iteration step, the set $J(I)$ will contain n members, corresponding to n new monomials which can be constructed. $\mathcal{P}(I)$ is then defined as $\mathcal{P}(I) = \{\texttt{floor}(j/N_c) : j \in J(I)\}$.

4.3 Monomial Ordering

A critical element of the parallel algorithm is the ability to determine the lexicographic index, I, given the monomial, $x^{\alpha(I)}$. The following closed-form formula allows us to do this in a distributed manner without excess computation.

$$I = \sum_{j=1}^{n-1} \sum_{i=1}^{\alpha_j(I)} f\left(n - j,\, d + 1 - \sum_{k=1}^{j-1} \alpha_k(I) - i\right) + 1, \quad (2)$$

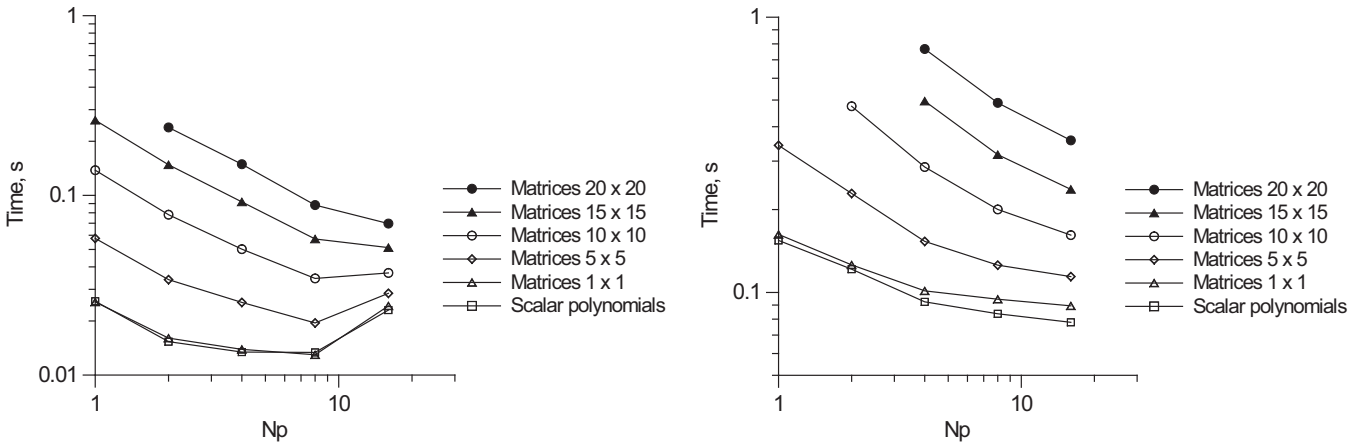
where the function $f(n, b)$ is the total number of monomials of degree b in n variables, given by Eq. (1), and $d = \sum_{j=1}^{n} \alpha_j$ is the degree of the monomial. The usefulness of this formula can be further extended by calculating the change in the monomial index ΔI when $\alpha_j(I)$ is incremented by one, corresponding to the multiplication of monomial $x^{\alpha(I)}$ by x_j Expanding Eq. (2), we can write

$$\begin{aligned} \Delta I|_{\alpha_j \to \alpha_j + 1} = \sum_{m=1}^{j-1} [f(n - m,\, d + 1 - \sum_{k=1}^{m-1} \alpha_k(I)) \\ - f(n - m,\, d + 1 - \sum_{k=1}^{m} \alpha_k(I))] \\ + f(n - j,\, d + 1 - \sum_{k=1}^{j-1} \alpha_k(I)). \end{aligned} \quad (3)$$

Eq. (3) is ideally suited for calculating the index of each new monomial $\tilde{I} = I + \Delta I(j)$ in a parallel environment. $\Delta I(j)$ can be calculated recursively, thus eliminating the need for recomputing the sums $\sum_{k=1}^{j-1} \alpha_k(I)$. This procedure requires only $O(n)$ operations to find all new monomials created from $x^{\alpha(I)}$ at each iteration step.

4.4 Efficient Parallel Implementation

One still needs to find the best way for evaluating $f(n - j,\, d + 1 - \sum_{k=1}^{j-1} \alpha_k)$ for each monomial index I and



(a) $n = 32$ variables

(b) $n = 64$ variables

Figure 1: Computational time versus the number of processors. Maximum number of monomials is 5×10^4.

each $j \in \{1, \ldots, n\}$ at each iteration. It is clearly not efficient to use Eq. (1) which requires a factorial number of operations to evaluate function $f(j, k)$ for each j and k. Alternatively, one could store all the functions $f(j, k)$ for $k \leq d$, and use the recursive relation $f(j,\, d+1) = (d+j)/(d+1) \cdot f(j,\, d)$, which only requires one multiplication per variable j. For a parallel algorithm, performance can be further improved by distributing $f(j,\, k)$ among the processors. In the current algorithm, each processor p contains all the functions $f(j,\, k)$, $k \leq d$ for a certain range of variables $j \in N_v(p)$. The number of variables per processor is given by $N_v = \texttt{ceil}(n/N_p)$.

To evaluate the increment $\Delta I(j)$, each processor requires values of f for all variables $j \in \{1, \ldots, n\}$ which are not necessarily contained in the same processor. We have addressed this problem by using the cyclic technique: once sums in the right-hand side of Eq. (3) are evaluated for all $f(j, k)$, $j \in N_v(p)$ contained by the current processor p, the functions $f(j, k)$ are sent to the left and new functions $f(j, k)$, $j \in N_v(p{+}1)$ are received from the right etc.

4.5 Matrix-Valued Coefficients

In the preceding paragraphs, we did not specify whether the polynomial coefficients are scalars or matrices. The general structure of the algorithm will not change whether scalar-valued or matrix-valued coefficients are considered. The practical difference will be that in a scalar case the entries of a coefficient array $c = (c_1, c_2, \ldots c_n)$ will be scalars, and in the matrix case they will be vectors. Since we are concerned only with symmetric matrices, each entry of the coefficient array c_i is a vector of a size $m(m+1)/2$ containing upper triangular entries of a symmetric matrix $m \times m$. Therefore, in the scalar case, the positivity test consists of simply evaluating the sign of a scalar c_i, whereas in the matrix case it consists of determining whether the given symmetric matrix c_i is positive-definite, i.e. evaluating the sign of its m eigenvalues. In the present algorithm, the serial version of the linear algebra package LAPACK was used for computing eigenvalues of a symmetric $m \times m$ matrix using QR factorization.

4.6 Computational Complexity

In the present algorithm, each processor has N_d/N_p number of coefficients at each iteration, with N_d given by Eq. (1). For scalar-valued coefficients, the positivity test requires evaluating the sign of each coefficient and takes N_d/N_p operations per core. Determining the indices of new monomials requires $\sim n\, N_d/N_p$ operations per core, since each monomial produces n new monomials. Communication complexity is determined by $n\, N_d/N_p$ operations per core with the length of each transmitted message $n + 2$. For matrix-valued coefficients of size $m \times m$, the computational complexity of the positivity test with QR-factorization is $\sim 4/3\, m^3$ per coefficient, making it $\sim m^3\, N_d/N_p$ total. The process of constructing new coefficients still requires $\sim n\, N_d/N_p$ operations. The possibility of parallelizing the positivity test itself in m can be explored. The number of communication operations does not change with the addition of matrix-valued coefficients, but the size of the transmitted messages increases to $n + 1 + m$. Both computation and communication complexities in scalar and matrix versions of the algorithm are proportional to $1/N_p$, thus indicating that the proposed algorithm should achieve perfect scalability in an ideal parallel environment (where the time per communication operation does not depend on the number of processors, and the idle time is zero).

4.7 Scalability

Scalability tests were performed on a parallel IBM Linux cluster Cosmea at ANL. Computational time versus the number of processors is plotted as a log-log plot in Fig. 1 for 32 and 64 variables, for the maximum number of monomials equal to 5×10^4. Matrix-valued polynomi-

als with the matrix size up to 20×20 are evaluated as well as scalar-valued polynomials. It is seen that the performance of the matrix algorithm with 1×1 matrices is very similar to the scalar algorithm, as expected. Slight differences are due to the different implementation of the positivity test: simple check of the coefficient signs in a scalar case, and a separate call to an eigenvalues solve (QR algorithm) in a matrix case. Since the parallel environment is not ideal, the overhead of parallelization exceeds than the savings when the number of variables handled by each processor is too small. In addition, load balancing becomes more difficult. Thus, the efficiency of parallel algorithms reduces greatly when the number of variables per processor becomes too small (as indicated by the increase in running time as CPU grows in Fig. 1(a) for cases with small matrix sizes). It is seen that the scalability improves with the matrix size and with the number of variables, since the number of operations per processor increases. Scalability is observed at approximately 80% of optimal in the 20×20 case. It is anticipated that for very large problem sizes the perfect scalability ($time \sim 1/N_p$) could be approximately achieved.

5 Conclusion

In this paper, we have proposed a parallel implementation of a well-known polynomial positivity test. We have also addressed several technical issues related to a distributed memory environment. In addition, we have implemented the algorithm using MPI in a distributed memory environment. Future work in this area will lean toward the development of parallel algorithms for optimization of positive polynomials. In particular, as noted in previous literature, Polya's Lemma gives rise to a semidefinite programming test for stability of robust and nonlinear systems. An open question is whether this semidefinite programming approach can be extended to a highly scalable parallel implementation.

References

[1] S. J. Benson and Y. Ye, "DSDP5: Software for semidefinite programming," Argonne National Laboratory, Tech. Rep. ANL/MCS-P1289-0905, 2005.

[2] S. J. Berkowitz, "On computing the determinant in small parallel time using a small number of processors," *Inform. Process. Lett.*, vol. 18, no. 3, pp. 147–150, 1984.

[3] L. Blum, F. Cucker, M. Shub, and S. Smale, *Complexity and Real Computation*. Springer, 1998.

[4] B. Borchers and J. G. Young, "Implementation of a primaldual method for SDP on a shared memory parallel architecture," *Computational Optimization and Applications*, vol. 37, no. 3, pp. 355–369, 2007.

[5] M. Castle, V. Powers, and B. Reznick, "A quantitative pólya's theorem with zeros," *Effective Methods in Algebraic Geometry*, vol. 44, no. 9, pp. 1285–1290, 2009.

[6] J. A. de Loera and F. Santos, "An effective version of polya's theorem on positive definite forms," *J. Pure and Appl. Algebra*, vol. 108, no. 3, 1996.

[7] I. Foster, C. Kesselman, and S. Tuecke, "The anatomy of the grid: Enabling scalable virtual organizations," *The International Journal of High Performance Computing Applications*, vol. 15, no. 3, 2001.

[8] S. Furber, "The future of computer technology and its implications for the computer industry," *The Computer Journal*, vol. 51, no. 6, 2008.

[9] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo, *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, 1995.

[10] G. H. Hardy, J. E. Littlewood, and G. Polya, *Inequalities*. Cambridge University Press, 1952.

[11] J. Held, J. Bautista, and S. Koehl, "From a few cores to many: a tera-scale computing research overview," Intel Research, White Paper, 2006.

[12] M. Litzkow, M. Livny, and M. W. Mutka, "Condor - a hunter of idle workstations," in *Proceedings of the 8th International Conference on Distributed Computing Systems*, 1988.

[13] M. Macedonia, "The GPU enters computing's mainstream," *Computer*, vol. 36, no. 10, 2003.

[14] R. C. L. F. Oliveira and P. D. Peres, "Parameter-dependent LMIs in robust analysis: Charachterization of homogeneous polynomially parameter-dependent solutions via lmi relaxations," *IEEE T. Automat. Control*, vol. 52, no. 7, 2007.

[15] A. Papachristodoulou and M. M. Peet, "On the analysis of systems described by classes of partial differential equations," in *Proceedings IEEE Conference on Decision and Control*, 2006, pp. 747–752.

[16] P. Parrilo and S. Lall, "Semidefinite programming relaxations and algebraic optimization in control," *Eur. J. Control*, vol. 9, no. 2-3, pp. 307–321, 2003.

[17] M. M. Peet, A. Papachristodoulou, and S. Lall, "Positive forms and stability of linear time-delay systems," *SIAM J. Control Optim.*, vol. 47, no. 6, 2008.

[18] M. Putinar, "Positive polynomials on compact semi-algebraic sets," *Indiana Univ. Math. J.*, vol. 42, no. 3, 1993.

[19] B. Reznick, "Some concrete aspects of Hilbert's 17th problem," *Contemp. Math.*, vol. 253, pp. 251–272, 2000.

[20] C. W. Scherer and C. W. J. Hol, "Matrix sum-of-square relaxations for robust semi-definite programs," *Math. Program. Ser. B*, vol. 107, pp. 189–211, 2006.

[21] C. Schmüdgen, "The K-moment problem for compact semi-algebraic sets," *Math. Ann.*, vol. 289, no. 2, 1991.

[22] G. Stengle, "A nullstellensatz and a positivstellensatz in semialgebraic geometry," *Math. Ann.*, vol. 207, 1973.

[23] M. Yamashita, K. Fujisawa, and M. Kojima, "SDPARA : SemiDefinite Programming Algorithm paRAllel version," *Parallel Comput.*, vol. 29, pp. 1053–1067, 2003.