Parallel Inversion of Polynomial Matrices

Alina Solovyova-Vincent
Computer Science
University of Nevada, Reno
Reno, NV 89557
alina@cs.unr.edu

Frederick C. Harris, Jr.
Computer Science
University of Nevada, Reno
Reno, NV 89557
fredh@cs.unr.edu

M. Sami Fadali
Electrical Engineering
University of Nevada, Reno
Reno, NV 89557
fadali@ee.unr.edu

Abstract

This paper presents an overview of different methods proposed in the last several decades for computing the inverse of a polynomial matrix, concentrating on Busłowicz's algorithm. A detailed description of Busłowicz's algorithm and its sequential implementation is followed by the presentation of a new parallel algorithm, based on Busłowicz's. The distributed and shared memory versions of this parallel algorithm are discussed, and the resulting computation times are analyzed and compared.
keywords: parallel algorithm, polynomial matrix inversion

1 Introduction

The problem of inverting polynomial matrices (or, more generally, rational matrices) has been under investigation for over half of a century. This research is well motivated because the computation of such inverses is needed in many fields. For instance, in multivariable control systems, a system is often described by a matrix of rational transfer functions. The problem of finding the inverse of a rational matrix arises in analysis and design using the inverse Nyquist array method [21, 27], in parameterization design of linear decoupling controllers [19, 25], in robust stability analysis [10], and in design using the QFT method [13, 21]. The inversion of polynomial matrices is also required in various fields of control system synthesis [15, 29]. Furthermore, the inversion of rational matrices is required in the analysis and synthesis of passive and active RLC networks for inversion of admittance or impedance matrices [16] and in the analysis of power systems using the method of diakoptics [1]. When a rational matrix is expressed as a ratio of a numerator polynomial matrix and a denominator scalar polynomial, the computation of the inverse essentially reduces to the computation of the inverse of a polynomial matrix [20]. Thus, in many cases, the problem of finding the inverse of a rational matrix can be solved by inverting the corresponding polynomial matrix.

The rest of this paper is outlined as follows: Section 2 introduces definitions and notations and provides the overview of other existing inversion methods along with their advantages and disadvantages. It also introduces Busłowicz's algorithm and outlines reasons for selecting this algorithm as the basis for a parallel implementation. Section 3 describes the details of the sequential implementation of the algorithm as well as the changes necessary to parallelize it. Discussions of the shared memory and distributed memory parallel implementations complete this section. Section 4 presents and analyzes the results of the sequential and parallel versions of the program. Conclusions and directions for future work are provided in Section 5.

2 Notation, Literature Review, and Busłowicz's Algorithm

2.1 Introduction of Notation

A polynomial matrix is a matrix which has polynomials in all of its entries. Consider a polynomial matrix $H(s)$ of degree n

$$H(s) = H_n s^n + H_{n-1} s^{n-1} + H_{n-2} s^{n-2} + ... + H_0,$$

where H_i are $r \times r$ constant square matrices, $i = 0, ..., n$. An example of such a matrix is

$$H(s) = \begin{bmatrix} s+2 & s^3+3s^2+s \\ s^3 & s^2+1 \end{bmatrix}.$$

In this case, the degree of the polynomial matrix is $n = 3$, and the size of the matrix H_i is $r = 2$. For this example,

$$H_0 = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}, H_1 = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix},$$

$$H_2 = \begin{bmatrix} 0 & 3 \\ 0 & 1 \end{bmatrix}, H_3 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

$H(s)$ is considered column proper if its highest degree coefficient matrix H_n is non-singular [29]. $H(s)$ is row proper if its transpose, $H^T(s)$, is column proper.

The notation used to denote the inverse of a matrix is $H^{-1}(s)$. Only unimodular matrices (*i.e.*, polynomial matrices with a non-zero determinant that is independent of s) have inverses that are themselves polynomial matrices [22].

Rational matrices are matrices whose entries are rational functions in s, which are non-singular at $s = 0$. A rational function can be expressed as Hd^{-1}, where H is a polynomial matrix and d is a scalar polynomial. Thus the problem of inverting a rational matrix can be reduced to inverting a polynomial matrix.

2.2 Literature Review

We begin with a review of the special case of inverting the resolvent matrix $[sI_r - H]$, where I is the unit matrix and H is an $r \times r$ matrix of constants. A process for finding $[sI_r - H]^{-1}$ is well documented and is known as Leverrier's algorithm [18]. Leverrier's algorithm as well as multiple extensions of this method (*i.e.*, Leverrier-Faddeev algorithm [11], Souriau-Frame-Faddeev algorithm [26], *etc.*) serve as a basis for several matrix inversion techniques that follow.

A number of different approaches for the inversion of polynomial matrices have been proposed over the past years. The assumptions made by different authors vary, and the results do not always have the same form. One of the first papers on this topic by Kosut [17] gives a direct algorithm based on a generalization of the Leverrier method. His method contains many polynomial operations and is not very general.

Munro and Zakian [22] used the approach suggested by Kosut for the inversion of rational polynomial matrices by the Souriau-Frame-Faddeev algorithm. They considered two distinct methods in their paper: one based on the Gaussian elimination algorithm and the other one based on the Faddeev algorithm. Both methods involve performing direct computation of the adjoint matrix obtained by polynomial operations. However, their methods have downfalls in that such operations are lengthy, require a "large degree of involved bookkeeping" [22], and are known to cause numerical problems. In addition, operations in the field of rational functions utilized in both methods are not suitable for computer programming.

Downs [4, 5] presented another approach, based on exact Gaussian elimination for matrices with integer coefficients. His method still contained many polynomial operations. Almost at the same time, Emre *et al.* [9] proposed a method of inversion of rational matrices based on Cramer's rule. The primary motivation for introducing this new method was to avoid polynomial arithmetic and to establish an algorithm systematically dealing with constant matrices. This approach required only simple arithmetic. Their method originally required restrictive assumptions that the polynomial matrix $H(s)$ is non-singular at $s = 0$ and that the determinant is known at the outset. The problem of polynomial cancellation was not taken into account by Emre *et al.* Downs was the one to point out the many restrictions and problems of their approach. In a series of papers that followed [6, 7, 8], most of these problems were resolved. Another point worth mentioning is that the inversion presented by Emre *et al.* was carried out by computing the determinants recursively. This inversion method requires that the determinants of $(n+1)r$ constant matrices be evaluated in order to compute the determinant of a polynomial matrix $H(s)$ of order r whose degree is n. Computation time for this method is large for large r and n.

Inouye [14] approached the problem of inverting polynomial matrices by generalizing Fadeev's recursive formula. His method is an extension of the Souriau-Frame-Faddeev algorithm. It does not require prerequisite determinants and requires operations with only constant matrices. It simultaneously determines the adjoint matrices and the coefficients of the determinants. The author showed that his algorithm is "faster than the existing ones." One of the disadvantages of his method is that it works only for row- or column-proper polynomial matrices. It also gives the inverse in the minimal degree form only if the polynomial matrix to be inverted is not a special form, but it cannot ensure that the denominator and inversion numerator matrix obtained will be irreducible for a general case.

Num [23], and much later Schuter and Hippe [28], proposed finding the inverse by generalizing known polynomial interpolation approaches. Both techniques require a careful choice of base points in order to avoid ill-conditioned equations. Both methods require complex computations. Another problem with interpolation methods is that only upper bounds for the degrees of the determinant and the adjoint are usually available. The interpolation thus involves redundant

equations and polynomials with unnecessarily high degrees.

In 1980 Busłowicz [24] published a paper with a method that is similar to the method proposed by Inouye [14] but more general in that it works for any non-singular polynomial matrix (as opposed to only row- or column-proper matrices). Busłowicz's recursive algorithm computes the inverse by Cramer's rule, explicitly calculating the adjoint matrix and the determinant starting from the coefficient matrices. It requires operations with only constant matrices. The drawback of Busłowicz's algorithm is that the irreducible form cannot be ensured in general.

Still another approach was developed independently by Zhang [30] and Chang *et al.* [3]. They both used a division algorithm for polynomial matrices to compute the inverse in irreducible form; however, their algorithms had increased computational complexity.

Fragulis *et al.* [12] proposed an algorithm that is a generalization of the Leverrier-type algorithm. The inverse is calculated using the recursive formula. Their approach does not seem to be significantly different from the one proposed by Busłowicz and does not provide any clear advantage over it.

The method of finding the inverse of a polynomial matrix based on state space realizations is used by Lin and Hsieh [20]. They compute neither the determinant nor the adjoint matrix. This method does not yield the exact solution, but experiments show that the algorithm gives accurate results for rational matrices that arise in the analysis and design of linear multivariable control systems.

2.3 Busłowicz's Algorithm

2.3.1 Advantages of Busłowicz's Algorithm

As mentioned in Section 2.2, Busłowicz based his approach for finding the inverse of a polynomial matrix on a generalization of Fadeev's recursive formula. A similar method was proposed by Inouye [14] in 1979, and it was the fastest and most general method at that time. Busłowicz's algorithm is even more general, does not require knowledge of the determinant at the outset, and works for any non-singular polynomial matrix. The only operations required are those on constant matrices.

There were several reasons for choosing this method for implementation. First, methods proposed before Busłowicz published his paper were obviously less general. Second, we wanted to implement an exact method, thus eliminating available algorithms that use approximations or interpolations such as [23], [28] and [20]. Two other newer methods [3, 30], provide only slight improvement of the results in that they yield the inverse in the already irreducible form. However, both authors agree that their algorithms require additional "complex computations". Finally, Busłowicz claimed that his algorithm was suitable for computer programming [24]. We agreed with this assessment and also saw a potential for great speedup in the parallel implementation.

2.3.2 The Algorithm

One of the general ways to compute the inverse of a matrix $H(s)$ is to evaluate the expression given by

$$H^{-1}(s) = \frac{\operatorname{adj} H(s)}{\det H(s)}, \tag{1}$$

where $\operatorname{adj} H(s)$ denotes the adjacent matrix $H(s)$, which is found as

$$\operatorname{adj} H(s) = \sum_{k=0}^{n(r-1)} Q_k s^k, \ Q_k \in R^{r\times r} \tag{2}$$

and

$$\det H(s) = \sum_{k=0}^{nr} a_k s^k, \ a \in R. \tag{3}$$

The problem of finding the inverse of a polynomial matrix comes down to finding an efficient method for calculating matrices Q_k, $k = 0, 1, ..., n(r-1)$, and the coefficients a_k, $k = 0, 1, ..., rn$, from the given matrices H_i, $i = 0, 1, ..., n$.

Busłowicz showed in his paper that the matrices Q_k of $\operatorname{adj} H(s)$ can be computed as

$$Q_k = (-1)^{r+1} R_{r-1,k}, \ k = 0, 1, ..., n(r-1), \tag{4}$$

and the coefficients a_k of $\det H(s)$ can be found using the formula

$$a_k = \frac{(-1)^{r+1}}{r} \mathrm{tr} G_{r,k}, \ k = 0, 1, ..., nr \tag{5}$$

where tr denotes the trace of a matrix.

The matrices $R_{r-1,k}$ and $G_{r,k}$ appearing in the above expressions are computed from the following iterative formulae:

$$G_{i,k} = H_0 R_{i-1,k} + H_1 R_{i-1,k-1} + ... + H_n R_{i-1,k-n} \tag{6}$$

$$a_{i,k} = -\frac{1}{i} \mathrm{tr} G_{i,k}, \ i = 1, 2, ..., r, \text{ and} \tag{7}$$

$$R_{i,k} = G_{i,k} + I_r a_{i,k}, \ i = 1, 2, ..., r-1 \text{ and } k = 0, 1, ..., in, \tag{8}$$

where

$$R_{0,k} = \begin{cases} I_r & \text{for } k = 0 \\ R_{0,k} = 0 & \text{for } k \neq 0 \end{cases} \quad (9)$$

and

$$R_{i,k} = 0 \text{ for } j < 0 \text{ or } k < 0 \text{ or } k > jn. \quad (10)$$

In addition,

$$R_{r,k} = G_{r,k} + I_r a_{r,k} = 0, \ k = 0, 1, ..., rn. \quad (11)$$

The algorithm proposed by Busłowicz for inversion of the polynomial matrices then consists of the following steps:

1. Using formulae (6)-(10), calculate $G_{i,k}, a_{i,k}$ and $R_{i,k}$ for $i = 1, 2, ..., r-1$ and $k = 0, 1, ..., in$, and calculate from formula (4) the matrices $Q_k \ for \ k = 0, 1, ..., n(r-1)$.

2. Using formulae (4) and (2), calculate the matrix $adjH(s)$.

3. Calculate the matrices $G_{r,k}$ for $k = 0, 1, ..., rn$ from the formulae (6)-(8) and the coefficients $a_{i,k}$ of the polynomial $\det H(s)$ from formula (5).

4. From formula (3) calculate the polynomial $\det H(s)$.

5. Calculate the matrix $H^{-1}(s)$ from formula (1).

6. The computations could be checked using the following equation:

$$G_{r,k} + (-1)^r a_k I_r = 0, \ k = 0, 1, ..., rn.$$

Note: In the case where the polynomial matrix has no inverse, the coefficients a_k, $k = 0, 1, ..., rn$, calculated by formula (5) will be equal to zero.

3 Sequential and Parallel Algorithms

3.1 Sequential Algorithm

Before the sequential code is discussed in detail, we introduce the variables that appear in the program.

- Each H_i is an $r \times r$ matrix. $n+1$ of them form the matrix $H(s)$, the inverse of which is to be computed (see the first formula in 2.1). Each H_i is represented by a 3-dimensional array $H[i][x][y]$, where $i = 0, 1, ..., n$, $x = 0, 1, ..., r-1$ and $y = 0, 1, ..., r-1$.
- Each $G_{i,k}$ is an $r \times r$ matrix (see formula (6) in 2.3.2), that is represented by a 4-dimensional array $G[i][k][x][y]$, where $i = 0, 2, ..., r$, $k = 0, 1, ..., rn$, $x = 0, 1, ..., r-1$ and $y = 0, 1, ..., r-1$.
- Each $R_{i,k}$ is an $r \times r$ matrix (see formulae (8)-(11)), that is represented by a 4-dimensional array $R[i][k][x][y]$, where $i = 0, 2, ..., r$, $k = 0, 1, ..., rn$, $x = 0, 1, ..., r-1$ and $y = 0, 1, ..., r-1$.
- Each $a_{i,k}$ is a coefficient (see formula (7)), that is represented by a 2-dimensional array $a[i][k]$, where $i = 0, 1, ..., r$ and $k = 0, 1, ..., rn$.
- Each a_k is a coefficient of $\det H(s)$ (see formula (3)), that is represented by a 1-dimensional array $alpha[k]$, where $k = 0, 1, ..., rn$.
- Each Q_k is one of $r \times r$ matrices that compose $\text{adj}H(s)$ (see formula (2)). They are represented by a 3-dimensional array $Q[k][x][y]$, where $i = 0, 1, ..., n$, $x = 0, 1, ..., r-1$ and $y = 0, 1, ..., r-1$.
- $Ident[x][y]$ is an $r \times r$ unit matrix.

Several changes had to be made to the algorithm outlined in Busłowicz's paper. First of all, step 1 of the algorithm (see Section 2.3.2) specifies the calculation of $G_{i,k}$, $a_{i,k}$ and $R_{i,k}$ for $i = 1, 2, ..., r-1$ and $k = 0, 1, ..., in$. However, our algorithm separates this step into two steps because different approaches are required for calculating the variables for $i = 1$ and $i > 1$. Thus, the algorithm first calculates $G_{i,k}$, $a_{i,k}$ and $R_{i,k}$ for $i = 1$ and $k = 0, 1, ...in$ and then continues with the rest of the calculations for $i > 1$. Second, as mentioned above, steps 1 and 3 of the algorithm are combined. Computations of Q_k are delayed until everything else in steps 1 and 3 is calculated. Hence, step 2 is also performed later in the program.

3.2 Parallel Algorithm

Armed with a working sequential version of Busłowicz's algorithm, we began analyzing program dependencies in order to decide on parallelization techniques. This section presents the details of the parallel implementation of Busłowicz's algorithm and outlines the changes made and challenges encountered in the process of parallelizing the sequential version of the program.

The parallel algorithm is given in Figure 1. Two new variables appear in this parallel code segment: *NUMPROC* is the number of processors used for calculations, and p is the distinct number associated with each processor p=0,...,*NUMPROC*-1. Because a SPMD (single program multiple data) programming structure was used, each processor executed the code shown on its portion of data. This algorithm was im-

```
1  for (k=p;k<n+1;k+=NUMPROC)
2  {
3      for(x=0;x<r;x++)
4          for(y=0;y<r;y++)
5              G[1][k][x][y]=H[k][x][y];
6      tr=0;
7      for(l=0; l<r;l++)
8          tr+=G[1][k][l][l];
9      a[1][k] =-tr;
10     for(x=0;x<r;x++)
11         for(y=0;y<r;y++)
12             R[1][k][x][y]=G[1][k][x][y]
                   + a[1][k]*Ident[x][y];
13 }
14 barrier(barrier1, NUMPROC);
15 for(i=2;i<r+1; i++)
16 {
17     for(k=p;k<n*i+1;k+=NUMPROC)
18     {
19         min=k;
20         if(k>n)
21             min=n;
22         for (ll=0;ll<min+1;ll++)
23         {
24             kr=k-ll;
25             if(kr<=(i-1)*n)
26                 for(x=0;x<r;x++)
27                     for(y=0;y<r;y++)
28                     {
29                         G[i][k][x][y]=0;
30                         for(l=0;l<r;l++)
31                             G[i][k][x][y]+=H[ll][x][l]
                                   *R[i-1][kr][l][y];
32                     }
33         }
34         tr=0;
35         for(l=0; l<r;l++)
36             tr+=G[i][k][l][l]);
37         a[i][k]=-tr/i;
38         for(x=0;x<r;x++)
39             for(y=0;y<r;y++)
40                 R[i][k][x][y]=G[i][k][x][y]
                       +a[i][k]*Ident[x][y];
41     }
42     SYNCHRONIZATION
43 }
```

Figure 1: The parallel algorithm.

plemented for both distributed memory and shared memory machines.

Implementing the program in a shared memory environment allowed the creation of variables that could be accessed directly by every process. In the shared memory environment, the shared memory segments are created using the *shmget()* system calls. Because there is a limit on the number of shared memory segments that can be created, 2-, 3- and 4-dimensional matrices are represented as 1-dimensional arrays. Shared memory segments are attached to the data segments of the calling process before performing calculations using *shmat()* and then are detached after the computations are completed. In a distributed memory environment, variables computed by one process that are required by another have to be passed explicitly by the program. MPI was chosen to provide the functionality required for programming in a distributed memory environment.

Because most parallelism occurs in the loops, a first attempt to parallelize any code typically requires looking for independent loops that can be split across multiple processors. Independent loops can be executed in any order without affecting the semantics of the program. There are several *for* loops in the sequential program, but, unfortunately, not all are independent. Clearly, the large outer i-loop (line 15 in Figure 1) is not independent. Calculations in the i^{th} iteration depend on the results of the previous $(i-1)^{st}$ iteration because the i^{th} iteration involves operations on $R_{i-1,k}$ (line 31). However, the k-loops are independent and can be parallelized (lines 1 and 17 in Figure 1). This parallelization was accomplished by performing striped partitioning of the matrices across the processors.

The presence of the dependent loops in the program created another challenge: synchronization of the processes and data. Looking at lines 15 and 17, one can notice that there are two nested loops, with an independent loop inside the dependent one. To make matters worse, the number of inner iterations (ll-loop on line 22 varies. The dependence on k can be seen in lines 19-22 Thus there are so-called partially parallel loops, *i.e.*, loops whose parallelization requires synchronization to ensure that iterations are executed in the correct order and produce the correct output. Specifically, no process can go on with execution of the i^{th} iteration until every other process had completed its $(i-1)^{st}$ iteration. In a shared memory environment, this synchronization is accomplished by placing a barrier before starting the next iteration of the i-loop (line 42 in Figure 1). Another barrier is placed on line 14

(Figure 1) to synchronize the processes, making sure that $R_{1,k}$ is calculated for all values of k before continuing with calculations for $i > 1$. In a distributed memory environment, synchronization is as important, but the data calculated by the processes must also be explicitly exchanged so it can be used in the next iteration by other processors. This explicit exchange using MPI communication calls implicitly accomplishes the process synchronization required. The rest of the program is left unchanged from the sequential version.

4 Results

The distributed memory code using MPI was tested on three different platforms. The first was a network of SGI O2 workstations. These machines have 180MHz MIPS R5000 processors with 320MB ram. The second platform was a network of Pentium IV workstations, each with a 1.8 GHz processor and 256MB ram. Both of these first two platforms have a standard 100 megabit network. The third platform is a cluster of Pentium IV Xeon processors, each at 2.2 GHz with 2GB of ram. The communication network is Myrinet 2000.

The shared memory code was tested on two different platforms. The first platform was an SGI Power Challenge 10000. This machine is a shared memory multiprocessor, consisting of 8 MIPS R10000 and 1 GB of ram. The second platform was an SGI Origin 2000. This machine is a shared memory multiprocessor, consisting of 16 MIPS R12000 300MHz processors and 2 GB of ram.

The complexity of the implemented sequential algorithm is $\mathcal{O}(n^2r^5)$. Thus the run times increase rapidly as the problem size increases. The problem size can be increased either by scaling the degree of the polynomial matrix n, the size of the matrix r, or both. We considered only real-life cases in the field of control theory, where neither the size of the matrix nor the degree of the polynomial typically exceeds 25. The figures in this section illustrate the computation times of a sequential program under various conditions as well as computation times obtained on the distributed and shared memory platforms with various numbers of processors. For comparison of the platforms, the sequential run times for the largest problem size are provided in Table 1.

4.1 Distributed Memory Implementation

The results obtained on the distributed memory platforms were not as expected. On the first two platforms, the algorithm provided some speedup on two processors for all problem sizes. However, when more processors were added, speedup was obtained only on the larger problem sizes, and the efficiency decreased drastically. On the third platform we obtained speedup across all processors, but the efficiency was poor for more than two processors. The efficiency of the parallel algorithm on the third distributed memory platform is shown in Table 2.

Platform	Sequential Time (sec)
SGI O2 NOW	2645.30
P IV NOW	29.99
P IV Cluster	18.75
SGI Power Challenge	913.99
SGI Origin 2000	552.95

Table 1: Sequential run times for different platforms ($n = 25$, $r = 25$).

Processors	2	4	8
Efficiency	89.6%	68.8%	49.7%

Table 2: Efficiency, P IV cluster ($n = 25$, $r = 25$).

4.2 Shared Memory Implementation

The results obtained on both shared memory platforms were outstanding. Figure 2 represents the average computation times (in seconds) on the first shared memory platform for the case when the degree of the polynomial matrix was fixed ($n = 10$) and the matrix size was varied from $r = 2$ to $r = 25$. As we saw earlier, the algorithm is $\mathcal{O}(n^2r^5)$. Figure 3 provides a computation-time surface showing how the changes of problem size (n and r) on 8 processors cause the computation time to increase drastically.

Table 3 presents the average efficiency of our algorithm for $n = 25$ and $r = 25$ on both shared memory platforms.

5 Conclusions and Future Work

The results obtained reflect the major difference between shared and distributed memory environments. Excellent performance in the shared memory environment shows that an efficient parallel algorithm can be

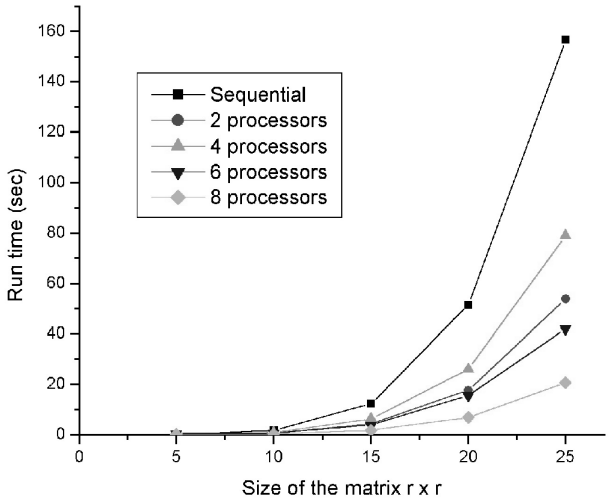

Figure 2: Run times, SGI Power Challenge ($n = 10$, r is varied).

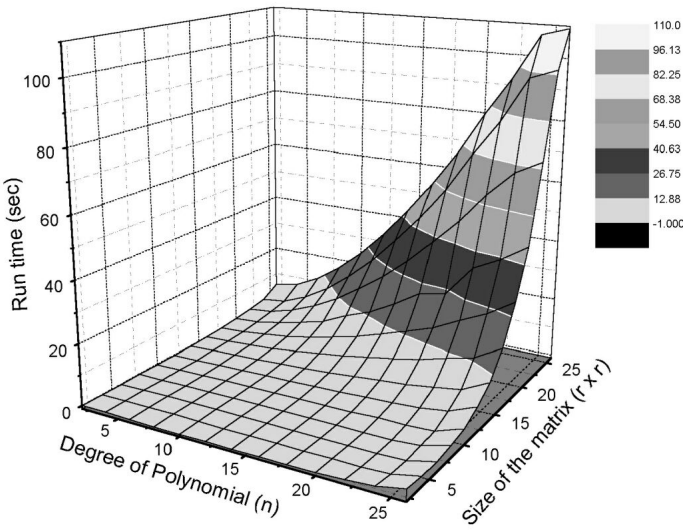

Figure 3: Run times, SGI Power Challenge, 8 processors, varied n and r.

Processors	2	3	4
SGI Power Challenge	99.65%	98.4%	98.2
SGI Origin 2000	99.9%	101.0%	98.7%

5	6	7	8	16
98.2%	97.9%	97.9%	95.8%	n/a
100.5%	99.0%	98.7%	98.2%	93.8%

Table 3: Efficiency, shared memory platforms ($n = 25$, $r = 25$).

designed for the highly data intensive problem of polynomial matrix inversion. These excellent results are due to the fact that communication and exchange of data were handled in the hardware in the fast shared memory implementation on the SGIs. In the case of a distributed memory environment, however, the dependencies of the original algorithm required transfer of large amounts of data between processors after each iteration, thus emphasizing the weaknesses of that environment and leading to a minimal speedup.

In the distributed memory environment, three platforms were used. Even with faster CPUs and faster networks, the efficiency did not improve much. These poor results were due to the fact that the communication costs far outweighed the performance gain of multiple processors. Because the problem sizes were limited to real-life applications, data sets of much larger sizes were not considered.

In the shared memory environment, near linear speedup was achieved on both platforms as can be seen in Table 3. This speedup means that the algorithm can take full advantage of the distributed computing power in the shared memory environment as the size of the problem increases. This great speedup can be attributed to the high degree of parallelism we were able to extract from the original algorithm as well as to the elimination of the need for the program to perform the communication explicitly. The efficiency over 100% on the Origin 2000 can be attributed to the architecture design where the processors are 2 CPUs to a card and the cache is 8 MB per CPU. This design allows the cache for each CPU to be used by either processor on the card, and allows one CPU to use the cache of both when the other CPU is idle. The NUMA memory architecture, which is tightly coupled to the CPU cards, also contributes to the behavior.

We have presented a parallel algorithm for computing the inverses of polynomial matrices. We have performed an exhaustive search of all available algorithms for polynomial matrix inversion and based our parallel algorithm on the method proposed in [24]. We have implemented the sequential version as well as two parallel versions. Based on the shared memory implementation results, we conclude that this new parallel algorithm is very efficient but should not be used on a distributed memory environment for small problem

sizes.

We see this work continuing in a variety of different ways. First, there is an algorithm for inverting multivariable polynomial matrices [2] that has never been parallelized. Second, we anticipate evaluating the distributed memory implementation in order to minimize message passing, thus improving performance. Third, larger problem sizes may also be considered in order to determine when the computation time overtakes the communication overhead.

References

[1] A. Brameller, M. N. John, and N. R. Scott. *Practical Diakoptics for Electrical Networks.* Chapman and Hall, London, England, 1969.

[2] Bulent Özgüler and Özay Hüseyin. On the inversion of multidimensional polynomial matrices. *IEEE Trans. on Circuits and Systems*, CAS-27(2):224–226, March 1980.

[3] F. R. Chang, L. S. Shieh, and B. C. Mc Innis. Inverse of polynomial matrices in the irreducible form. *IEEE Trans. Automat. Control*, **AC-32**(6):507–509, June 1987.

[4] T. Downs. On the inversion of a matrix of rational functions. *Linear algebra and its applications*, **4**:1–10, November 1971.

[5] T. Downs. On the reduction and inversion of the nodal admittance matrix in rational form. *IEEE Trans. Circuits Systems*, **21**:592–597, November 1974.

[6] T. Downs, E. Emre, Ö. Hüseyin, and K. Abdullah. Comments on "On the inversion of rational matrices". *IEEE Trans. Circuits Systems.*, **22**:375–376, April 1975.

[7] E. Emre, K. Abdullah, and Ö. Hüseyin. A new algorithm for the inversion of rational matrices. *Archiv für Elektronik und Übertragungstechnik*, **28**:461–464, 1975.

[8] E. Emre and Ö. Hüseyin. Generalization of Leverrier's algorithm to polynomial matrices of arbitrary degree. *IEEE Trans. Automat. Control*, **20**:136, 1975.

[9] E. Emre, Ö. Hüseyin, and K. Abdullah. On the inversion of rational matrices. *IEEE Trans. Circuts Systems*, **21**:8–10, 1974.

[10] M.S. Fadali. Stability testing for systems with polynomial uncertainty. In *Proc. 2002-ACC*, May 2002. *To appear.*

[11] D. K. Faddeev and I. S. Sominskii. *Sbornik zadach po vysshei algebre.* Moscow: Gostekhizdat, 2^{nd} edition, 1949.

[12] G. Fragulis, B. G. Mertzios, and A. I. G. Vardulakis. Computation of the inverse of a polynomial matrix and evaluation of its Laurent expansion. *Int. J. Control*, **53**(2):431–443, 1991.

[13] I.M. Horowitz. *Quantitative Feedback Theory (QFT).* QFT Publications, Boulder, CO, 1992.

[14] Y. Inouye. An algorithm for inverting polynomial matrices. *Int. J. Control*, **30**(6):989–999, 1979.

[15] T. Kailath. *Linear Systems.* Prentice-Hall, Inc, 1980.

[16] E. H. Keonig, Y. Tokad, and H. K. Kesavan. *Analysis of Discrete Physical Systems.* McGraw-Hill, New York, 1967.

[17] R. L. Kosut. The determination of the system transfer function matrix for flight control systems. *IEEE Trans. Automat. Control*, **AC-13**:214, April 1968.

[18] U. J. J. Leverrier. Sur les variationes séculaire des élementes des orbites pour les sept planètes principales. *Journal de Mathématique*, **5**(série 1):230 ff, 1840.

[19] C. A. Lin and T. F. Hsieh. Decoupling controller design for linear multivariable plants. *IEEE Trans.*, **36**:485–489, 1991.

[20] C. A. Lin, C. W. Yang, and T. F. Hsieh. Inversion of polynomial matrices. *Systems & Control Letters*, **27**:47–54, 1996.

[21] J. M. Maciejowski. *Multivariable Feedback Design.* Addison-Wesley, Reading, MA, 1989.

[22] N. Munro and V. Zakia. Inversion of rational polynomial matrices. *Electronic Lett.*, **6**(19):629–630, 17th Sept. 1970.

[23] O. T. Num, Y. Ohta, and T. Matsumoto. Inversion of rational matrices by using FFT algorithm. *Trans. IECE Japan*, **E-61**:732–733, 1978.

[24] M. Busłowicz. Inversion of polynomial matrices. *Int. J. Control*, **33**(5):977–984, 1980.

[25] R.V. Patel and N. Munro. *Multivariable system theory and design.* Pergamon Press, Oxford, 1982.

[26] H. H. Rosenbrock. *State-space and Multivariable Theory.* Nelson, London, 2^{nd} edition, 1970.

[27] H.H. Rosenbrock. *Computer Aided Control System Design.* Academic Press, London, 1974.

[28] A. Shuster and P. Hippe. Inversion of polynomial matrices by interpolation. *IEEE Trans. Automat. Control*, **37**:363–365, 1992.

[29] W.A. Wolowich. *Linear multivariable systems.* Springer-Verlag, 1974.

[30] S. Y. Zhang. Inversion of polynomial matrices by interpolation. *Int. J. control.*, **46**:33–37, 1987.