Parallel Gaussian Elimination

Contents

[Background 1](#_Toc290583477)

[Data and Task Partitioning 1](#_Toc290583478)

[Algorithm and Determinant Results 2](#_Toc290583479)

[Run-Time, Speedup, and Efficiency Results 3](#_Toc290583480)

[Results Analysis 4](#_Toc290583481)

# Background

Gaussian elimination to reduce a matrix to lower or upper triangular form is a classic method for solving matrices. In this project we implement Gaussian elimination to reduce a matrix to upper triangular form and then compute the determinant of that matrix to obtain a relative determinant to the original. The implementation uses the C++ MPI libraries.

# Data and Task Partitioning

The data and task partitioning was of uniform distribution. The columns of the initial matrix were distributed in a banded manner:

If the above equation held true for any arbitrary column, i, and the number of processes, P, and the rank of each process, Rank, then that column was given to process number Rank. The dimension of the matrix isn’t always divisible evenly by P. Thus any column that received less than the number they should have received to equalize the column distribution had an “identity column” added. That is, a column of zeroes with a one on the diagonal of the matrix. This created even data distribution across all processes.

Task partitioning was also uniform except for calculation of the determinant, which was reduced to the master process 0. Every process calculated its column’s multiplication factors and broadcasted them, then used those factors to reduce the ith column to zero below the diagonal, then used the other processes’ multiplication factors to perform row operations.

# Algorithm and Determinant Results

The algorithm runs as follows with :

* N=number of columns in the expanded matrix, after the identity columns have been added.
* myID=process rank (0..P)
* values=2D vector of columns for each process
* local\_det=aggregation of values on the diagonal for each process, reduced later

for all i = 0..N

if myID == (i mod P)

local\_max\_idx <- index >= i of value with greatest magnitude

local\_det = local\_det \* values[local\_max\_idx]

Broadcast local\_max\_idx from process (i mod P)

swap( values[i], values[local\_max\_idx] ) in all columns in all processes

if myID == (i mod P)

for j = i+1..N

mult\_factors.push\_back( values[j] / values[i] )

Broadcast mult\_factors from process (i mod P)

for j = i+1..N

for all columns in all processes:

values[j] -= mult\_factors[0..N-j] \* values[i]

end for

Reduce to the master process local\_det with MPI\_MULT

The idea is to relocate the maximum useful value in each column to the row intersecting the diagonal in that column to avoid divide by zero error and to avoid error propagation from using extremely small values. Since the data is distributed in a banded manner across all processes, all processes must know the index of the rows they are swapping, the ith and the local\_max\_idx’th rows. Then the process that owns the ith column finds all the values needed to zero out its column below the diagonal, stores, and broadcasts them. All processes then use this ratio to perform a scaling of each row owned. The determinant values computed for the initial matrix for each of the supplied input matrices is:

|  |  |
| --- | --- |
| Calculated Determinants for various N | |
| 800x800 | -0.0205119 |
| 1600x1600 | 2.62238e+36 |
| 2400x2400 | 9.98539e+30 |
| 3200x3200 | 6.43494e-12 |

# Run-Time, Speedup, and Efficiency Results

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Run-Times | | | | | | |
|  |  | P | | | | |
|  |  | **1** | **2** | **4** | **8** | **16** |
| N | **6.40E+05** | 49.02923 | 24.67303 | 12.614 | 6.504307 | 3.47633 |
| **2.56E+06** | 390.92 | 196.998 | 99.56653 | 50.6844 | 26.02213 |
| **5.76E+06** | 1318.053 | 663.673 | 334.1203 | 169.044 | 86.2484 |
| **1.02E+07** | 3123.467 | 1571.643 | 789.335 | 398.145 | 201.935 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Speedup | | | | | | |
|  |  | P | | | | |
|  |  | **1** | **2** | **4** | **8** | **16** |
| N | **6.40E+05** | 1 | 1.987159 | 3.88689 | 7.537965 | 14.10373 |
| **2.56E+06** | 1 | 1.984386 | 3.926219 | 7.712827 | 15.0226 |
| **5.76E+06** | 1 | 1.985998 | 3.944846 | 7.797102 | 15.28206 |
| **1.02E+07** | 1 | 1.987389 | 3.957086 | 7.845048 | 15.46768 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Efficiency | | | | | | |
|  |  | P | | | | |
|  |  | **1** | **2** | **4** | **8** | **16** |
| N | **6.40E+05** | 1 | 0.993579 | 0.971723 | 0.942246 | 0.881483 |
| **2.56E+06** | 1 | 0.992193 | 0.981555 | 0.964103 | 0.938912 |
| **5.76E+06** | 1 | 0.992999 | 0.986212 | 0.974638 | 0.955129 |
| **1.02E+07** | 1 | 0.993694 | 0.989272 | 0.980631 | 0.96673 |

# Results Analysis

It can be seen that the run times for each matrix are approximately halved with each successive doubling of P, leading to a speedup figure of O(P) for all matrices tested. This is achieved mostly through the banded distribution of data and tasks as explained in the Data and Task Partitioning section. Each process is kept eliminating numbers and performing row operations as much as possible. Additionally, the relatively decreasing amount of communication overhead necessary for each matrix relative to the dimension of the matrix increases speedup. The number of data elements transmitted for each run is:

Since the number of transmitted elements follows the above relation, it can be seen (and is shown in the efficiency table) that as P is held constant and N increases, efficiency increases because more processors are working simultaneously rather than waiting on data transmission.