University of Illinois at Urbana-Champaign
Department of Computer Science

Hybrid Parallel Implementation of LU Decompositon and the Conjugate Gradient Method

Parallel Programming for Scientists and Engineers

CS 420

Written by

robijns2 Michel Robijns
bilberg2 Christopher Prinds Bilberg

December 8, 2015

1 Introduction

This report will treat two common methods to solve systems of linear equations as they commonly arise in the discretization of partial differential equations (PDE's). These methods can generally be categorized as either direct solution methods or iterative methods. Direct solution methods use algorithms such as Gaussian elimination whereas iterative methods start off by guessing a solution and proceed by strategically modifying the guess until it converges to the solution.

This report will compare the conjugate gradient method, which is an iterative method, with LU decomposition, which is a direct solution method. Each of these methods and their respective advantages and disadvantages and their degree of suitability for parallelization will be discussed. The report will conclude with performance measurements and a recommendation on when to use which method and when not.

2 Sample Problem

Since the two methods discussed in this report solve systems of linear equations as they commonly arise in the discretization of partial differential equations (PDE's), it is instrumental to devise a representative test scenario for such problems. The requirements for such a test scenario are twofold:

1. There should exist an analytical solution to verify the numerical solution. An analytical solution does not exist for the far majority of PDE's, which is the reason why numerical solutions are used in the first place. It is however necessary to verify the correctness of the algorithm before using the solvers on PDE's that do not feature an analytical solution.
2. The resulting matrix should be symmetric and positive definite (SPD). Some of the advanced iterative solution methods such as the conjugate gradient method require the matrix to be SPD.

The finite difference discretization of the Helmholtz equation satisfies both requirements and will therefore be used as a representative test scenario.

2.1 The Helmholtz Equation

We consider the Helmholtz equation on a square two-dimensional domain, Ω, with Direchlet boundary conditions on its boundary, $\partial\Omega$:

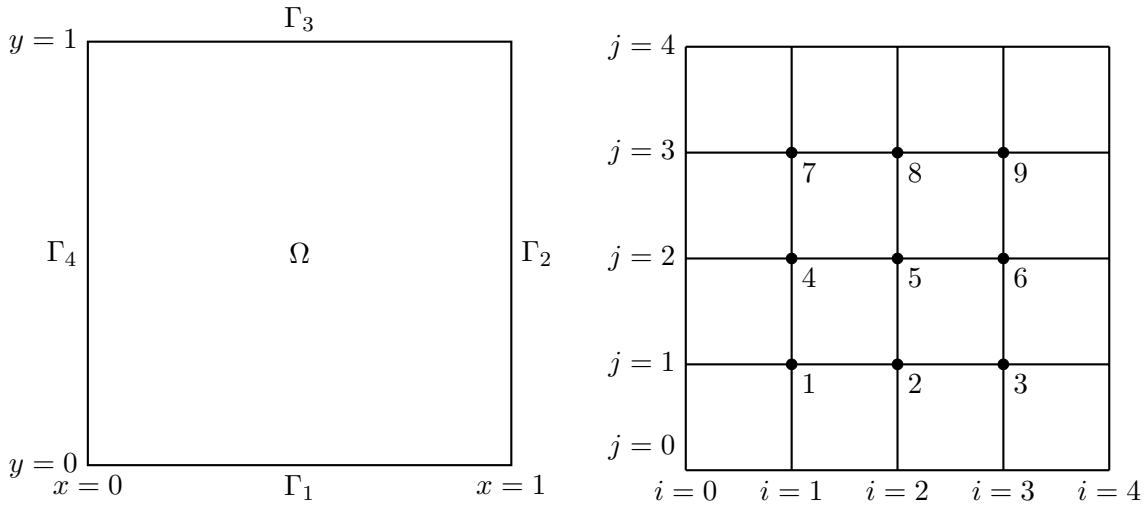
$$\begin{aligned} -\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) + 10u &= f(x,y) \quad \text{on} \quad \Omega := (0,1) \times (0,1) \\ u &= g(x,y) \quad \text{on} \quad \partial\Omega := \Gamma_1 \cup \Gamma_2 \cup \Gamma_3 \cup \Gamma_4 \end{aligned} \tag{1}$$

where

$$f(x,y) = (x^2 + y^2 + 10)y\sin(xy) - 2x\cos(xy) \tag{2}$$

and the boundary conditions are given by

$$g(x,y) = \begin{cases} 0 & \text{on } \Gamma_1 \\ y\sin(y) & \text{on } \Gamma_2 \\ \sin(x) & \text{on } \Gamma_3 \\ 0 & \text{on } \Gamma_4 \end{cases} \tag{3}$$



(a) Domain (b) Discretization

Figure 1: Two two-dimensional domain and the finite-difference discretization.

The four boundaries are labelled $\Gamma_1 \dots \Gamma_4$. See Figure 1a for a graphical depiction. The exact solution to this particular problem is given by

$$u(x, y) = y \sin(xy) \tag{4}$$

2.2 Discretization by Finite-Differences

The Helmholtz equation will be discretized by means of a central-difference scheme to attain a discretization of second order accuracy. The central difference approximation to the second derivative of a generic function $y(x)$ is given by

$$\frac{d^2y}{dx^2}(x_i) = \frac{y_{i-1} - 2y_i + y_{i+1}}{h^2} + O(h^2) \tag{5}$$

The approximation is second order accurate, as indicated by the $O(h^2)$ term. The partial derivatives in the Helmholtz equation can therefore be discretized as

$$\frac{d^2u}{dx^2}(x_i) = \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h^2} + O(h^2) \tag{6}$$

$$\frac{d^2u}{dy^2}(x_i) = \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h^2} + O(h^2) \tag{7}$$

Substitution of Equation (6) and Equation (7) into Equation (1) gives

$$-\frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h^2} - \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h^2} + 10u_{i,j} = f_{i,j} + O(h^2) \tag{8}$$

Rearranging Equation (8) and omitting the $O(h^2)$ term yields

$$\frac{-u_{i,j-1} - u_{i-1,j} + 4u_{i,j} - u_{i+1,j} - u_{i,j+1}}{h^2} + 10u_{i,j} = f_{i,j} \tag{9}$$

2.3 Determining the Coefficient Matrix and the Right-Hand Side Vector

Consider the grid shown in Figure 1b. The grid is split into four parts in each direction, so $N = 4$. The result is a grid with 25 nodes. Only 9 out of the 25 nodes are internal nodes and the remaining 16 nodes are boundary nodes. The values of u at the internal nodes are unknowns, so there are precisely 9 unknowns and the coefficient matrix A will therefore be of size 9×9. After all, 9 equations are required to solve for 9 unknowns in a determined system.

The internal nodes were numbered with a horizontal numbering scheme, with the first node located in the bottom left-hand corner. The matrix A, as given in Equation (10), contains the coefficients corresponding to the 9 unknowns. For reasons of clarity, the zeros in A have been omitted and h has been left as a variable, even though it is known that $h = \frac{1}{N} = \frac{1}{4} = 0.25$ in this particular case.

$$A = \frac{1}{h^2}\begin{bmatrix} 4 & -1 & & -1 & & & & & \\ -1 & 4 & -1 & & -1 & & & & \\ & -1 & 4 & & & -1 & & & \\ -1 & & & 4 & -1 & & -1 & & \\ & -1 & & -1 & 4 & -1 & & -1 & \\ & & -1 & & -1 & 4 & & & -1 \\ & & & -1 & & & 4 & -1 & \\ & & & & -1 & & -1 & 4 & -1 \\ & & & & & -1 & & -1 & 4 \end{bmatrix} + 10I \tag{10}$$

All relevant known grid values, such as the forcing term and the boundary condition, are moved to the right-hand side vector $\mathbf{f}$, given in Equation (11).

$$f = \begin{bmatrix} f_1 \\ f_2 \\ f_3 + \frac{1}{h^2} y_1 \sin(y_1) \\ f_4 \\ f_5 \\ f_6 + \frac{1}{h^2} y_2 \sin(y_2) \\ f_7 + \frac{1}{h^2} \sin(x_1) \\ f_8 + \frac{1}{h^2} \sin(x_2) \\ f_9 + \frac{1}{h^2} y_3 \sin(y_3) + \frac{1}{h^2} \sin(x_3) \end{bmatrix} \tag{11}$$

where f_i is the value of $f(x, y)$ at node i. The system of equations can now be formulated:

$$A\mathbf{u} = \mathbf{f} \tag{12}$$

where $u = [u_1, u_2, \ldots, u_9]^T$. The goal of the algorithms discussed in the following sections is to solve the system $A\mathbf{u} = \mathbf{f}$ for the solution vector $\mathbf{u}$. It is trivial to scale the matrix A and the right-hand side vector $\mathbf{f}$ to much larger problem sizes and a simple program has been written to automatically generate A and $\mathbf{f}$ for any desirable value of N.

3 Direct Solution Method: LU Decomposition

LU decomposition is rooted in the Gaussian elimination algorithm. The coefficient matrix A is written as the product of a lower triangular matrix L and an upper triangular matrix U. Substitution of $A = LU$ into

$$A\mathbf{u} = \mathbf{f} \tag{13}$$

gives

$$LU\mathbf{u} = \mathbf{f} \tag{14}$$

By defining $U\mathbf{u} \equiv \mathbf{y}$, we can solve the above system for $\mathbf{y}$:

$$L\mathbf{y} = \mathbf{f} \tag{15}$$

This process is called the forward substitution. The final result, $\mathbf{u}$, can now be obtained by performing the backward substitution step, given by

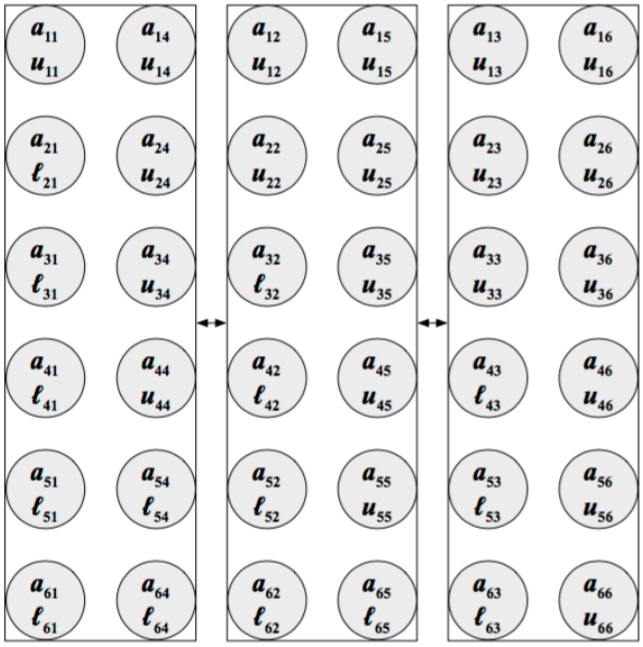
$$U\mathbf{u} = \mathbf{y} \tag{16}$$

The two substitution steps yield a very low computational effort, which is the strength of LU decomposition. Performing Gaussian elimination to obtain the L and U matrices is a computationally expensive step. Gaussian elimination is applied such that the lower triangle of the coefficient matrix A contains only zero elements and thus becomes the upper triangular matrix, U. The matrix manipulations required to do are stored in a second matrix, L, the lower triangular matrix. In practice, A can be overwritten such that it contains both L and U, which saves memory.

LU decomposition can be parallelized by splitting the coefficient matrix A up into columns, as shown in Figure 2b. Each process will own a set of columns. The pseudocode in Figure 2a shows an outer `for` loop that goes over the columns from left to right. The first inner `for` computes the values of L. After these values have been computed, they need to be communicated to all the processes to the right. After that, each process will start subtracting rows from each other corresponding to the values of L. Both inner `for` loops will be parallelized by means of OpenMP.

```
for k = 1 to n − 1
    if k ∈ mycols then
        for i = k + 1 to n
            ℓ_ik = a_ik / a_kk
        end
    end
    broadcast {ℓ_ik : k < i ≤ n}
    for j ∈ mycols, j > k
        for i = k + 1 to n
            a_ij = a_ij − ℓ_ik a_kj
        end
    end
end
```

(a) Pseudocode.



(b) The 1-D column parallel algorithm.

4 Iterative Solution Method: The Conjugate-Gradient Method

Often when dealing with solving linear systems the coefficient matrix will be sparse, such that most of the elements will be zero. In this case the direct methods can be impractical and hence

it can be more useful to use iterative solvers. The conjugate gradient method is an example of such an iterative algorithm.

The difference between the direct method and iterative method is that in case of the iterative method an initial guess is made and for every iteration the guess is improved until the error is smaller than a certain tolerance. In order to solve a linear system by the conjugate gradient method the matrix must be a symmetric square matrix and be positive definite. The pseudocode for the sequential algorithm is given by:

$k = 0; x_0 = 0, r_0 = b$
`while`($||r_k||^2$ >tolerance) `and` ($k <$ `max_iter`)
  $k++$
  `if` $k = 1$
    $p_1 = r_0$
  `else`
    $\beta = \frac{r_{k-1} \cdot r_{k-1}}{r_{k-2} \cdot r_{k-2}}$
    $p_k = r_{k-1} + \beta_k \cdot p_{k-1}$
  `endif`
  $s_k = Ap_k$
  $\alpha_k = \frac{r_{k-1} \cdot r_{k-1}}{p_k \cdot s_k}$
  $x_k = x_{k-1} + \alpha_k \cdot p_k$
  $r_k = r_{k-1} - \alpha_k \cdot s_k$
`endwhile`
$x = x\_k$

4.1 Implementation

In each iteration results from previous iteration is needed and hence it becomes a bit difficult to parallelize the outer while loop. However for a distributed memory implementation the data in the coefficient matrix and the different vectors can be distributed among the processors such that each processor can work on its own local data. It can be seen from the pseudocode that for each iteration the following linear algebra operations has to be performed:

- Computation of dot product
- Computation of matrix vector product
- Computation of vector addition $a + k \cdot b$

If the coefficient matrix is of size $n \times n$ and the vectors of size n each process can be given a part of each vector of size $\frac{n}{p}$. Hence each process will be able to compute it's own local dot product and vector addition without communication with the other processors. The final dot product of a given vector can then be obtained by calling `MPI_Allreduce()` across the processors.

The coefficient matrix can be distributed among the processors row wise such that each process holds $\frac{n}{p}$ rows and a total of $\frac{n}{p}n$ elements. In order for each process to compute its own local matrix vector product it needs to have all the values in the vector to be multiplied. Hence it will be necessary to call `MPI_Allgather()` with respect to the given vector before the matrix product can be computed. However since each process holds its own part of the resulting vector it will not be necessary to broadcast the result among the processors afterwards.

The overall implementation steps with respect to MPI and distributed memory can be summarized as:

1) The coefficient matrix and resulting vector is distributed among the processors using `MPI_Scatter()` such that each processor holds $\frac{n}{p}$ elements of the vector and $\frac{n}{p}$ rows of the matrix.

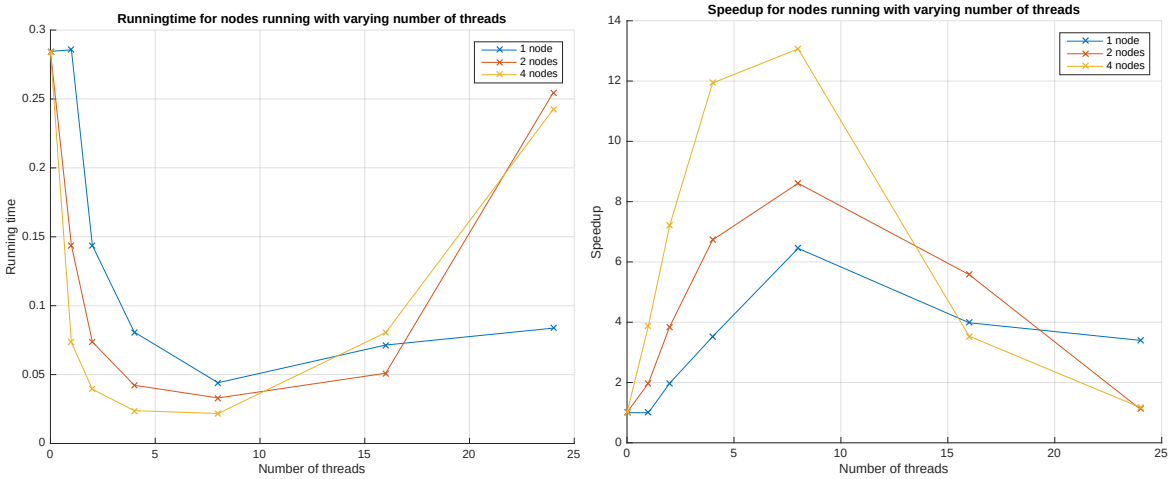
2) Computation of dot products and reduction for the r vector using `MPI_Allreduce()`

3) Computation of matrix vector product by gathering partial vector by `MPI_Allgather()`

5 Results

5.1 Conjugate Gradient Method

After running the conjugate gradient hybrid method on the Golub Campus Cluster the following runningtime and speedup times were plotted as shown in the figure below.



(a) Runningtime for conjugate gradient method (b) Speedup for conjugate gradient method

Figure 3: Results for the conjugate gradient method for a matrix of size 1024×1024.

It can be seen from the plots that in general more nodes make the program execute faster which is expected since the matrix and vectors are distributed among more processors. For the different number of nodes it can be seen that the running time is decreasing with the number of threads until a certain point, which is 8 threads in this case. After this sweet spot the execution time increases again. This is most likely due to the overhead of initializing all the threads compared to the size each thread is working on, as well as the overhead that might occur for the threads in terms of synchronization.

In order to compare the result of running the program with many ranks compared to many threads the program was run with a fixed number of 32 processing elements, which can be summarized in the table below:

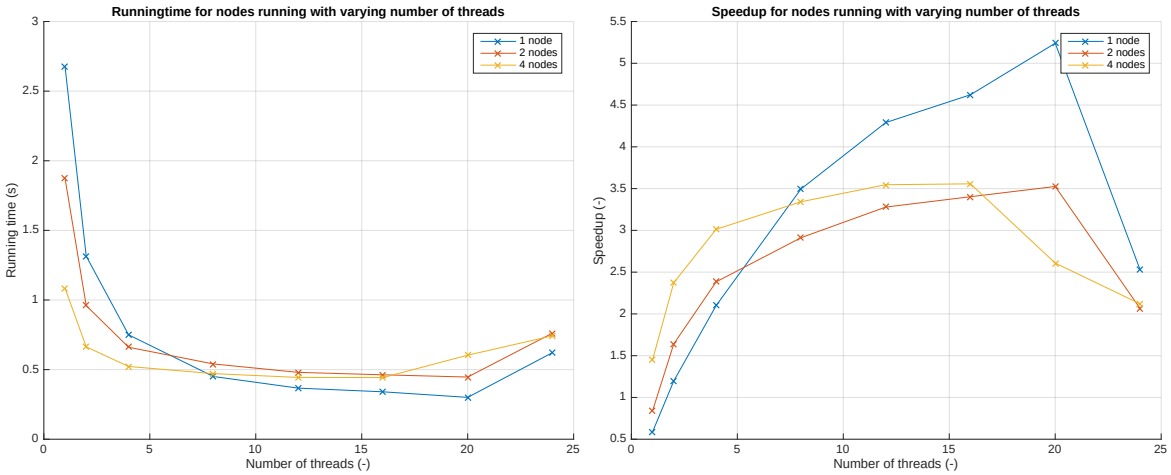
Ranks	Threads	Running Time
4	8	0.03212595
8	4	0.01710820
16	2	0.02140903
32	1	0.14737511

Table 1: Runningtime for 32 processors

As it can seen from the table the fastest execution time is achieved when there is slightly more mpi processors than openmp threads. However if the number of MPI processors becomes too big as the case for 32 processors the running time becomes much slower, which is probably due to the communication overhead between the nodes. It can be seen that the optimal ratio between MPI processors and threads in this case is $\frac{8}{4} = 2$. However the ratio $\frac{16}{2} = 8$ is pretty close. Thus it can be seen that there is a sweet spot between processors and threads to get the fastest execution time.

5.2 LU Decomposition

LU Decomposition is notoriously hard to parallelize on distributed memory systems because updated columns of the lower triangular matrix L must be communicated to all other processes before computation can proceed, thereby acting as an implicit barrier. This results in frequent interruptions of the algorithm while the process waits for data. Running multiple MPI nodes only turned out to be effective when using one or two CPU cores per node, as shown in Figure 4a. Execution on a single node beats execution on multiple nodes when more than 6 CPU cores can be utilized. Figure 4b shows that a speedup of a factor 5 could be achieved compared to the sequential program. The execution time increases when more than 20 CPU cores are utilized because of overhead associated with thread initialization.



(a) Runningtime for conjugate gradient method (b) Speedup time for conjugate gradient method

Figure 4: Results for LU decomposition for a matrix of size 1024×1024.

5.3 Comparison

A comparison of execution time between the two algorithms is shown in Figure 5. The comparison was performed by using the sequential version of either algorithm in order to compare

the nature of the algorithms. LU decomposition outperforms the conjugate gradient method at small problem sizes by a small margin, whereas the conjugate gradient method shows its strength at larger problems. This is the reason why, in practicality, LU decomposition is only used for very small problems and the conjugate gradient method is the go-to method for large numerical problems.

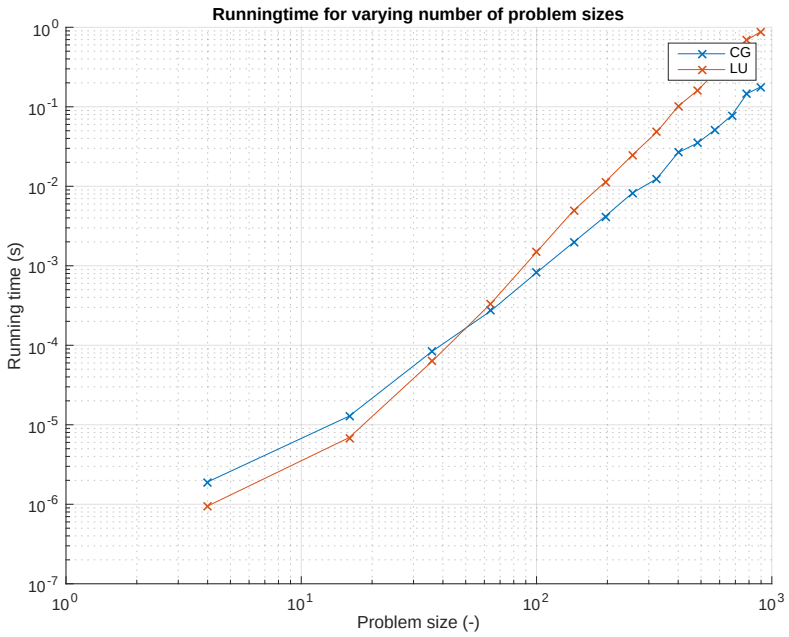


Figure 5: Comparison of LU decomposition with the conjugate gradient method.

6 Conclusion

This report provided a comparison between LU decomposition and the conjugate gradient method to solve systems of linear equations, typically written as $A\mathbf{u} = \mathbf{f}$. A realistic and representative sample problem was provided: a finite-difference discretization of a partial differential equation, the Helmholtz equation. The sample case has an exact solution that enables a trivial verification of the correctness of the algorithms.

LU decomposition is hard to parallelize on distributed memory systems. Using multiple MPI processes only yielded an improvement when the individual nodes where limited to one or two CPU cores per node. Shared memory parallelization in OpenMP beats MPI when more than 6 are available. This is caused by the large number of messages and the associated interruptions of the algorithm caused by the necessity to wait for new data.

The conjugate gradient method is easier to parallelize and is suitable for distributed memory systems as well as shared memory systems and hybrid systems. The computationally expensive parts of the conjugate gradient methods are a matrix-vector multiplication and several dot products, both of which are well-studied and trivial to parallelize.

LU decomposition performs marginally better than the conjugate gradient method when the problem size is very small, on the order of a few hundred elements at most. The conjugate gradient method is the go-to method for any larger problem. However, the conjugate gradient method requires the coefficient matrix to be symmetric and positive definite, while LU decomposition does not impose such requirements.