CSE 6220 Programming Assignment 3

Jacobi's Method

Qi Zheng (qzheng61); Wenqing Shen (wshen35)

Overview

The **Jacobi method** (or **Jacobi iterative method**) is an algorithm for solving a system of linear equations $Ax = b$, with A having no zeros along its main diagonal. Each diagonal element is solved for, and an approximate value is plugged in. The process is then iterated until it converges. It can be shown that for diagonally dominant matrices, Jacobi's method is guaranteed to converge. The goal of this assignment is to develop a parallel algorithm for Jacobi's method, and compare the performance with sequential algorithm.

Algorithm Design

Sequential Implementation

Given a full rank $n \times n$ matrix $A \in R^{n \times n}$, a right-hand size vector $b \in R^n$, a termination accuracy *l2_termination* $\in R$, a maximum iteration count iter_max. Jacobi's method iteratively approximates $x \in R^n$ for $Ax = b$ following the steps:

```
Initialize x = [0, 0, ...0]
D = diag(A)
InvD = D^{-1}
R = A – D
iter_count = 0
residual = ||Ax − b||    // ||•|| is the L2 norm

while (residual > l2_termination && iter_count < iter_max) do {
        x =invD * (b − Rx)    // update x
        residual = ||Ax − b|| // update residual
}
```

Parallel Implementation

In this project, we use a 2-D mesh (grid) as the communication network. We assume that the number of processors is a perfect square: $p = q \times q$, q is an integer. p processors will be arranged into a grid of size $q \times q$. The inputs are distributed on the grid in a pattern we define as block distribute method. The method distributes n components onto q processors following a pattern of:

$$\begin{cases} \lceil \frac{n}{q} \rceil & \text{if } i < (n \mod q) \\ \lfloor \frac{n}{q} \rfloor & \text{if } i \geq (n \mod q) \end{cases}$$

Initially, the matrix A and vector b both are both stored in processor (0, 0) in the grid. The parallel algorithm is described as follows:

1. **Vector Distribution**

   Block distributes the input vector of size *n* from process (0, 0) to processes (i, 0), i.e., the first column of the 2D grid communicator. Elements of the vector are equally distributed, such that ceil(n/q) elements go to the (*n mod q*) first processes and floor(n/q) elements go to the remaining processes in a pattern described in previous texts. Vector b is distributed in this pattern.

2. **Matrix Distribution**

   Equally block distribute the matrix A stored on processor (0,0) onto the whole grid (i, j). First equally distribute the row blocks of A onto the first column of processors using MPI Scatterv through the column sub-communicator created by MPI_Cart_sub. Then for each row in the grid, equally scatter the elements onto each processor in the same row using row sub-communicator.

3. **Transpose a vector**

   Define a function that 'transpose' the input vector *x* distributed among the first column (i,0) of processors to the whole grid, such that it is block decomposed by row of processors. To accomplish this, first, each processor (i, 0) alone the first column sends its local vector to the diagonal processor (i, i). Then the diagonal processor (i, i) broadcasts the message among its column using a column sub-communicator.

4. **Solves A*x* = b for *x* using Jacobi's method,**

   Implement parallel Jacobi's algorithm as follows:

```
D = diag(A)               // block distribute to first column (i,0)
invD = 1/ D
R = A - D                 // make a copy of A except setting diagonal to zero
x = [0, ..., 0]           // initialize x to zero, block distributed on first column

for (iter in 1:max_iter):
        w = R*x           // The matrix multiplication is solved by first transposing the input
                          // vector x as defined earlier, then locally multiplying the row
                          // decomposed vector by the local matrix. Then, the resulting local
                          // vectors reduce along rows onto the first column.
        x = invD * (b - P)        //  local on first column
        w = A*x                   // using distributed matrix vector multiplication as described
                                  // in w = R*x
        l2 = ||b - w||            // calculate L2-norm in a distributed fashion
        if (l2 <= l2_termination):
                return            // exit if termination criteria is met, make sure
                                  // all processor know that they should exit (-> MPI_Allreduce)
```

Results and Discussions

To analyze the performance of parallel computing and compare that with sequential program, we ran our program with p ranging from 1 to 64 including both odd and even numbers of processors, and the problem size varying from 10 to 30,000.

Runtime Analysis

Figure 1 shows the plot of runtime as a function of processor number on log scale.

- Generally, when problem size n is small, the runtime is dominated by communication time, hence it increases with the increase in p.
- When problem size n is large, computing takes much longer time, hence when increasing p from 1, the computing power of the cluster increases, and thus reduces the runtime. However, as p increases, communication starts playing a more and more important role.

When p exceeds certain threshold, communication takes the lead, and thus offsets the runtime reduce from the increase of computing capability. As a result, the plots flatten out, and even increase for some cases.

In Figure 2, the sequential runtime is plotted against parallel runtime as a function of problem size n with p ranging from 1 to 64. In this problem, there are quite a few communication processes going on during runtime. Hence, when the problem is not computational intensive (n is not large enough), parallel computing does not exhibit much advantage. Only when n is large enough (in this case, when n approaches 10^3, some parallel cases start run faster than sequential, and when n exceeds 10^4, most parallel cases outperforms sequential), parallel computing excels sequential in performance. Also, the dominant power of communication also exhibits itself with flat plots on the graph when a small n is varying.

Speedup and Efficiency Analysis

The speedups of the parallel and sequential algorithm as a function of processor numbers for different problem sizes are plotted on Figure 3. Speedups as a function of problem size with different numbers of processors are plotted on Figure 4. As we can from the graphs, with fixed processor number, generally speedup is larger for problems of larger size. Also, for a problem with fixed size, when the size is small, using more processors actually reduces speedup. They conform with our previous observations. Only when the problem size is large enough, using parallel computing with decent number of processors is valuable. This conclusion can be further confirmed by the two plots of efficiencies as a function of processor numbers with varying problem size and as a function of problem size with varying processor numbers.

One thing we observed from the graph was that the efficiency for parallel computing exceeds 1 for a problem size of 30,000. We think this is a result of limited memory of each core in the cluster. Within our cluster, each core has a memory of 4G. However, the matrix A with n= 30,000 results in a total storage space of 7.2G, hence, it has to be broken down and stored in two cores, resulting in communication overhead.

In the end, plotting and analyzing the runtime, speedup and efficiency plots will enable us to determine the optimal number of processors to use for problems of certain sizes with a goal of improving the efficiency, reducing runtime, and using smaller number of processors if possible.

Optimizations

1. In the original algorithm described in the assignment, D is stored as matrix. For a large size A, D is very sparse with non-zero values only on diagonal. To save the memory, for the diagonal matrix D, we used an array of size n to hold the data, and block decompose it onto the first column in the same pattern as vector b.
2. Since matrix (vector) D is only used for the calculations of R and D^{-1}. During the iterative Jacobi's method, only D^{-1} is used, hence we actually don't need D. D^{-1} could be obtained directly from taking the inverse of the diagonal values of A, while R could be obtained by copying A and setting diagonal values to 0. This could further save storage memory, especially for solving large size problems with limited number of processors. (When n=30000, the memory required for storing A has already reached 7.2GB!)