Program Documentation

Enzhi Li

March 20, 2017

1 Introduction

My program is designed to solve heat diffusion equation using generalized Euler method. The program consists of four parts. Part I of the program contains file `GhostVector.py` in which the class `GhostVector` is defined. The generalized Euler method relies heavily on the matrix-vector multiplication which we wish to parallelize using MPI, and in Part II of the program, parallel matrix-vector product is implmented in file `spmvParallel.py`. Conjugate gradient algorithm is widely used to solve simultaneous linear equations, and is employed in my program to solve the discretized heat diffusion equation. In Part III of the program, conjugate gradient algorithm is implemented in file `spmvcgParallel.py`, and MPI is used to accelerate the program. Part IV of the program contains the solution of heat diffusion equation using generalized Euler method, and comparison is made between analytical result and numerical result.

2 **`GhostVector`** class

Part I of the program contains file `GhostVector.py`, and in this file class `GhostVector` is defined. The methods contained in this class are: `createGhostVector`, `getGlobalSize`, `getLocalSize`, `getGhostSize`, `exchangeGhostValues`, `dotProduct`, `gather`, and `gatherAll`. Method `createGhostVector` takes `globalSize` which is the total length of the non-ghost vector as argument, and uses the MPI communicator to calculate the length of local vectors and local ghost vectors for each processor, and for each processor creates ghost vectors which are all initialized to 0. Methods `getGlobalSize`, `getLocalSize`, and `getGhostSize` are getter functions for `globalSize`, `localSize`, and `ghostSize`, respectively. In `exchangeGhostValues`, ghost points are exchanged between neighboring processors. `dotProduct` is used to calculate the inner product of the non-ghost vector. To get the inner product of the total vector, inner products of local vectors are calculated and summed together. The method `gather` is used to gather all the local ghost vectors to processor 0, and in processor 0 a class field `totalGhostVector` is created which concatenates together all the local ghost vectors. In method `gatherAll`, the field `totalGhostVector` which was initially created only in processor 0 is now broadcast to all the other processors.

3 Parallel program for sparse matrix vector product

Through discretization of heat diffusion equation, we can get a tridiagonal matrix, and thus in this program, we only need to consider the matrix-vector product for tridiagonal matrix. This tridiagonal matrix has this form:

$$\begin{pmatrix} 2a+1 & -a & 0 & 0 \\ -a & 2a+1 & -a & 0 \\ 0 & -a & 2a+1 & -a \\ 0 & 0 & -a & 2a+1 \end{pmatrix}. \tag{1}$$

We choose to represent the tridiagonal matrix using `coo_matrix` defined in scipy.sparse. It clear that this matrix can be fully specified as long as we know the matrix dimension and the parameter `a`. Thus, in file `spmvParallel.py`, a function `sparseMatrix` is defined, which takes as argument the matrix dimension and the parameter `a`, and returns the matrix dimension, the row, column, and data information for the sparse matrix.

In order to visualize the matrix, two auxiliary functions, `printSparseMatrix`, and `createSparseMatrix`, are defined. In function `createSparseMatrix`, a sparse matrix is created from the dimension, row, col, and data information that is returned by `sparseMatrix`, and in function `printSparseMatrix`, this sparse matrix is printed in matrix format. However, these two functions are not essential to the program, since the matrix-vector product will be realized using other method than numpy.dot, and that method will be described below in detail.

The core function in the file `spmvParallel.py` is `productParallel`. Since we have chosen to reprsent our matrix using coo_matrix format, we need to specify our matrix using matrix `dimension`, `row`, `col`, and `data`. From `dimension, row, col, data`, we can, but we will not, create a sparse matrix. However, for sake of clarity, we will still refer to the row number and column number of the sparse matrix, and we call this non-existent matrix `virtual matrix`. To parallelize this matrix-vector multiplication, we will slice this virtual matrix into some rectangular matrices, calculate the matrix-vector product for these rectangular matrices and concatenate together the resultant vector into a single vector. The row number of each processor starts from `startIndex`, and ends at `endIndex`. In order to extract the data that is required for matrix-vector product on each processor, we need to calculate `imin` and `imax`, which are indices indicating the starting point and end point for list `data`. Once we have extracted the data that will be used for matrix-vector product on each processor, we can calculate the resultant vector segments `y` for each processor and generate `resultVector` which concatenates together all of the vector segments.

We run the program using different number of processors to see if the parallelization can really acclerate the program. In order to make the time difference obvious enough, we have chosen to make the matrix dimension really large. In our test, we set matrix dimension equal to 200,000, and get the benchmark result, as shown in figure 3.1.

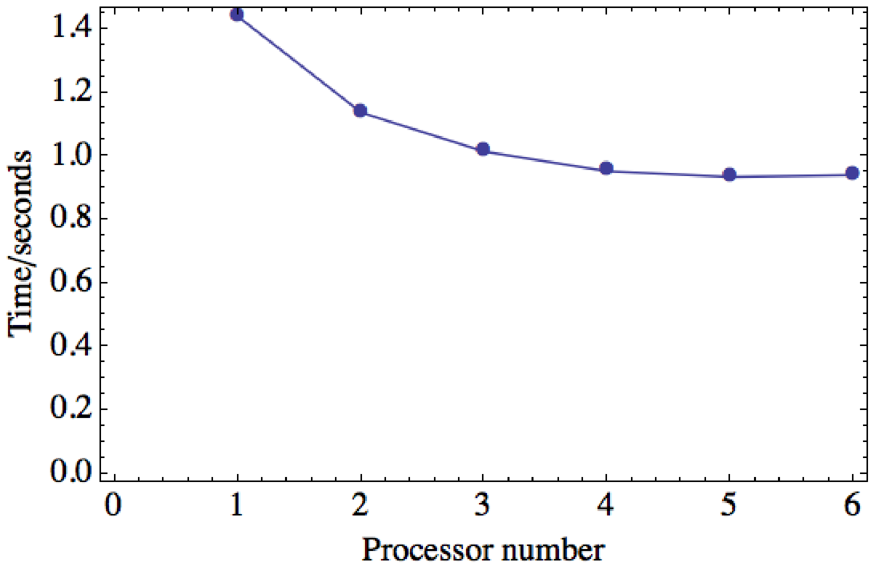

Figure 3.1: Bench mark spmvParallel.py, Time versus Number of processors.

We can see from the figure that running time does decrease with increasing number of processors. When the number of processors reaches 6, the time saturates to a constant value, indicating that MPI communication overhead has overshadowed the time we saved from parallelization.

4 Parallel program for conjugate gradient algorithm

Part III of the program contains file `spmvcgParllel.py`. In this file, function `conjugateGradient` is defined. This function calculates simultanesou linear equations of this form:

$$Ax = b, \tag{2}$$

where matrix A is a sparse matrix of the form in Equation (1). The matrix A can be uniquely specified by giving the matrix dimension and the parameter a. To parallelize this program, we need to parallelize the sparse matrix vector product that is frequently used in this algorithm, and this parallelization has already been done in Part II of the program.

Test of this program is shown below.

```
Matrix A:
[[ 1.4  -0.2  0.   0.   0.   0. ]
 [-0.2  1.4  -0.2  0.   0.   0. ]
 [ 0.  -0.2  1.4  -0.2  0.   0. ]
 [ 0.   0.  -0.2  1.4  -0.2  0. ]
 [ 0.   0.   0.  -0.2  1.4  -0.2]
 [ 0.   0.   0.   0.  -0.2  1.4]]
Vector b:
[ 1.  2.  3.  4.  5.  6.]
Solution of Ax = b:
[ 0.99993392  1.99953746  2.9968283   3.97826066  4.85099635  4.97871376]
Ax =
[ 1.  2.  3.  4.  5.  6.]
```

It can be seen that this program yields correct result for linear equations. Next, we test the parallel program to see if the program has been accelerated by using more processors. The figure below shows running time versus number of processors for matrix dimension = 50000, and a = 0.1. As can be seen from this figure, running time decreases as a function of number of processors. This shows that the program accelerates when we use more processors, just as expected.

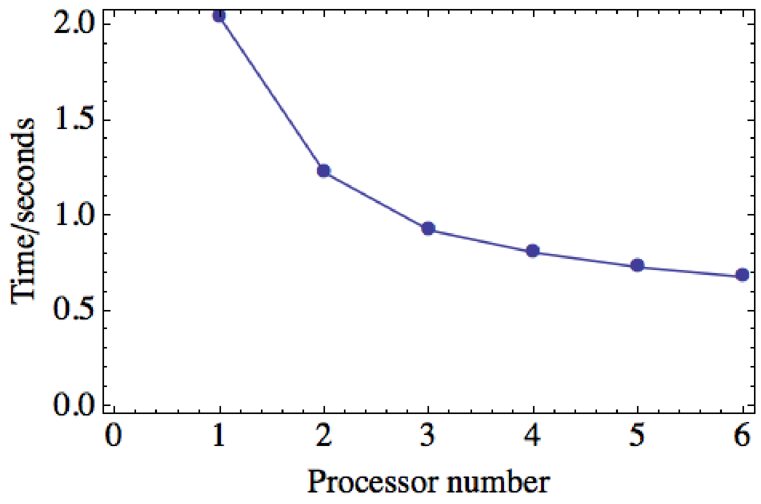

Figure 4.1: Time versus number of processors for parallel conjugate gradient algorithm

5 Parallel program for solution of heat diffusion equation

Part IV of the program contains file `thetaEulerParallel.py`. In this file, the function `thetaEuler` is defined, which takes as argument the number of x points `nx`, the maximum iteration number `ntmax`, the parameter `theta`, and MPI communicator `comm`. This function aims to solve this linear equation:

$$\begin{pmatrix} 1+2\lambda\theta & -\lambda\theta & 0 & 0 \\ -\lambda\theta & 1+2\lambda\theta & -\lambda\theta & 0 \\ 0 & -\lambda\theta & 1+2\lambda\theta & -\lambda\theta \\ 0 & 0 & -\lambda\theta & 1+2\lambda\theta \end{pmatrix} .u^{n+1} \tag{3}$$

$$= \begin{pmatrix} 1-2\lambda(1-\theta) & \lambda(1-\theta) & 0 & 0 \\ \lambda(1-\theta) & 1+2\lambda\theta & \lambda(1-\theta) & 0 \\ 0 & \lambda(1-\theta) & 1+2\lambda\theta & \lambda(1-\theta) \\ 0 & 0 & \lambda(1-\theta) & 1+2\lambda\theta \end{pmatrix} .u^n + \mathrm{dt} f$$

where $\lambda = \frac{\delta t}{\delta x^2}$, and θ is the parameter that determines whether we are using forward or backward Euler method.

5.1 $\theta = 0$, forward Euler method

When we use forward Euler method, δt and δx should satisfy the condition that $\frac{\delta t}{\delta x^2} <= 0.5$. To test this program, we set `nx` = 100, and run the program for different number of processors. The figure below plots the running time versus number of processors.

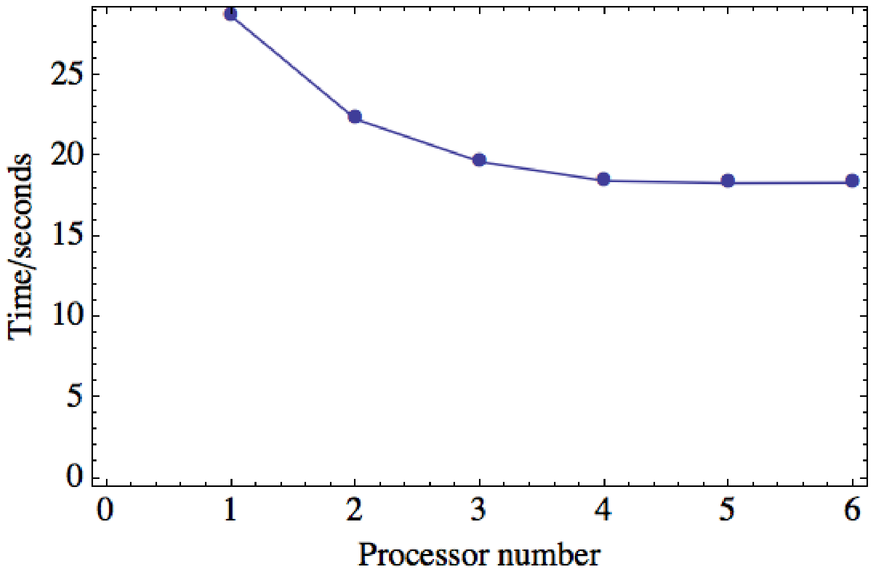

Figure 5.1: Running time vs. processor number when $\theta = 0$

We can see that the total running time decreases as the number of processors increases and saturates to a constant value when the number of processors reaches 6.

5.2 $\theta = 0.5$

When $\theta = 0.5$, we use half the new value and half the old value to update the vector, and the restriction that $\delta t/\delta x^2 <= 0.5$ still applies. We run the program for `nx` = 100, and plot the running time versus number of processors in figure 5.2.

It is clear that the running time decreases as the number of processors increase. However, the total running time for $\theta = 0.5$ is longer than the time for forward Euler method.

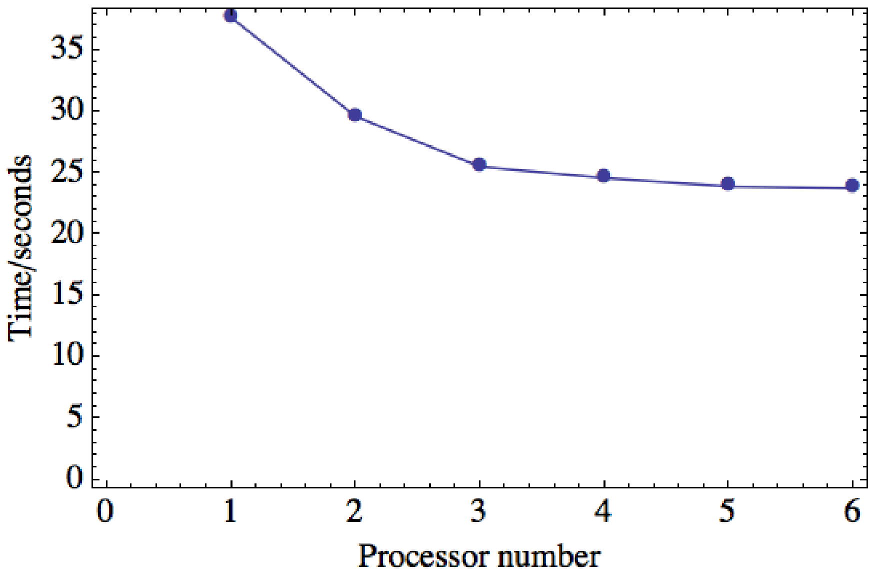

Figure 5.2: Total running time vs. processor number for $\theta = 0.5$

5.3 $\theta = 1$, backward Euler method

For backward Euler method, there is no restriction for the values of δx and δt, and thus we can choose the value of δt to our own discretion. Here we still set the value of `nx` $= 100$, and plot the total running time versus number of processors used.

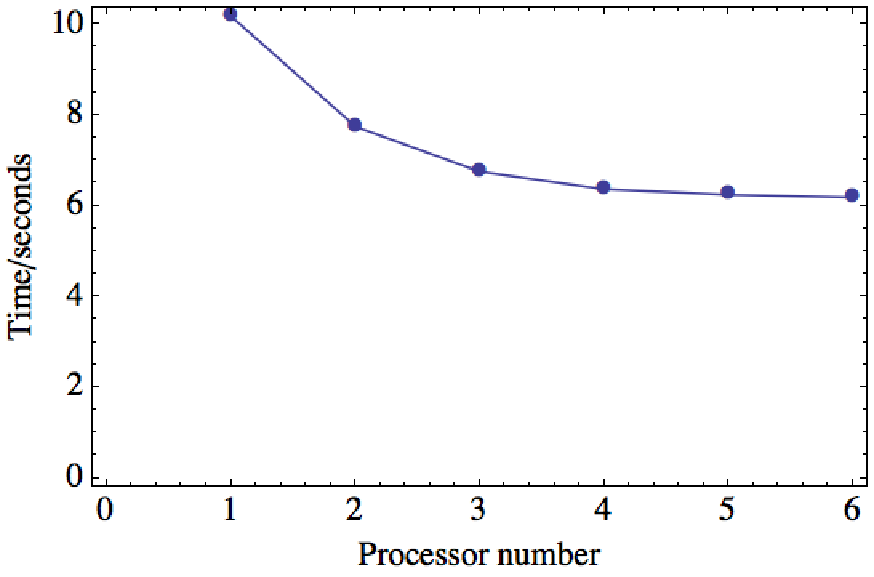

Figure 5.3: Total running time vs. processor number for backward Euler method

One thing that is noteworthy is that the running time for backward Euler method is much shorter than that of forward Euler method, since there is no restriction on the value of δt for backward Euler method. From this figure we can see that the running time also decreases when the number of processors increases, and saturates to a constant value when the number of processors reaches 6.

5.4 Test of the correctness of the program

We are supposed to solve this equation:

$$\begin{aligned} \frac{\partial u}{\partial t} &= \frac{\partial^2 u}{\partial x^2} + f(x) \\ u(x=-1) = u(x=1) &= 0 \\ u(x, t=0) &= u_0(x) \end{aligned} \tag{4}$$

In our program, we have set $f(x) = 1, u_0(x) = 0$, and our program yields the solution of this equation when temperature distribution reaches equilibrium state. The equilibrium solution to Equation (4) is:

$$u(x, t) = -\frac{1}{2}(x+1)(x-1) \tag{5}$$

Comparison is made between the analytical result and the numerical result, and they are both plotted in figure 5.4.

As can be seen from this figure, the numerical result and the analytical result completely overlap with each other, and thus the correctness of our program as an equation solver is beyond any doubt.

5.5 Scaling behavior for different Euler methods

To see how the total running time changes as a function of `nx`, which is the number of x points used in the program, we will set $\theta = 0, 0.5, 1$, run the program with two processors, and plot the total running time versus `nx` for different θ values.

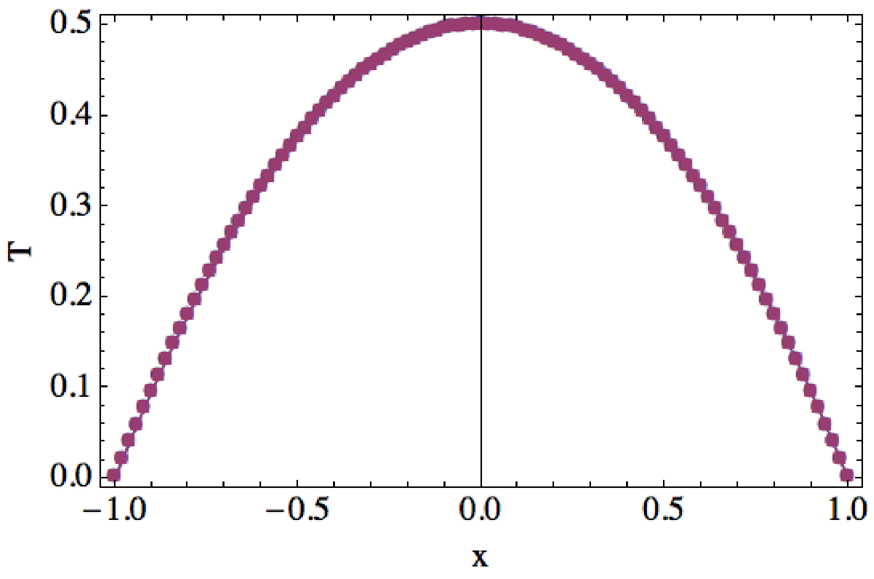

Figure 5.4: Comparison of numerical result and analytical result

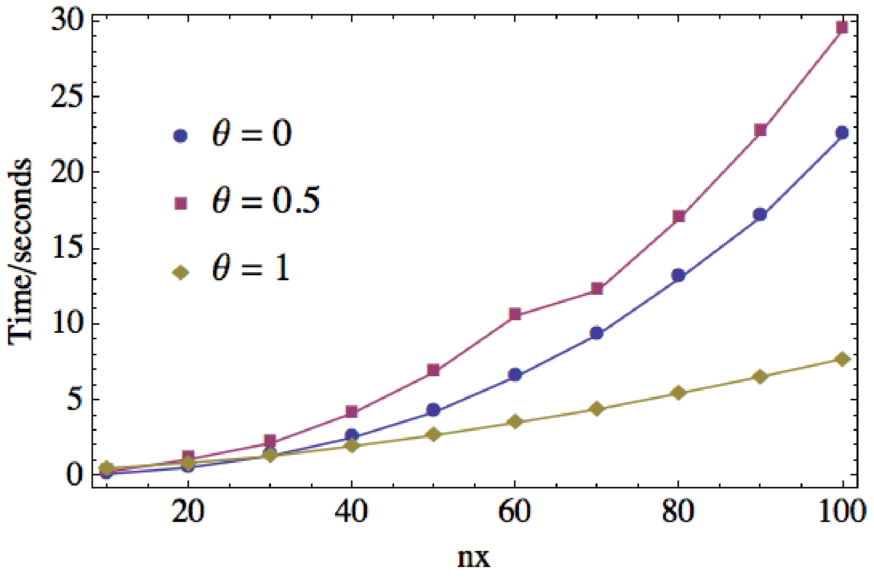

Figure 5.5: Sacling behavior for different values of θ. Running time vs. `nx`.

From figure 5.5, we can see that the scaling behavior of the program depends on the value of θ. When $\theta = 0, 0.5$, the restriction that $\delta t/\delta x^2 <= 0.5$ applies, and the total running time increases rapidly as `nx` increases. However, when $\theta = 1$, the restriction between δt and δx is lifted, and the total running time increases in a more sluggish manner than that for $\theta = 0, 0.5$.