Final Project

wenlewei

2022.06.09

1 Equation analysis

$$\begin{aligned} \rho c\frac{\partial u}{\partial t} - \kappa\frac{\partial^2 u}{\partial x^2} &= f \quad \text{on } \Omega \times (0,T) \\ u &= g \quad \text{on } \Gamma_g \times (0,T) \\ \kappa\frac{\partial u}{\partial x}n_x &= h \quad \text{on } \Gamma_h \times (0,T) \\ u|_{t=0} &= u_0 \quad \text{in } \Omega. \end{aligned}$$

In the 1D case, we can consider the following options.

$$f = \sin(l\pi x), \quad u_0 = e^x, \quad u(0,t) = u(1,t) = 0, \quad \kappa = 1.0.$$

So, we can get the fomulation

$$\rho c\frac{\partial u}{\partial t} - \kappa\frac{\partial^2 u}{\partial x^2} = sin(l\pi x)$$

Use explicit difference scheme.

$$\begin{aligned} \frac{\partial u}{\partial t} &= \frac{u_j^{n+1} - u_j^n}{\Delta t} \\ \frac{\partial^2 u}{\partial x^2} &= \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{\Delta x^2} \end{aligned}$$

A explicit numerical solution can be obtained.

$$u_j^{n+1} = \frac{\kappa\Delta t}{\rho c\Delta x^2}u_{j+1}^n + (1 - \frac{2\kappa\Delta t}{\rho c\Delta x^2})u_j^n + \frac{\kappa\Delta t}{\rho c\Delta x^2}u_{j-1}^n + \frac{\Delta t sin(l\pi x)}{\rho c}$$

Use implicit difference scheme.

$$\begin{aligned} \frac{\partial u}{\partial t} &= \frac{u_j^{n+1} - u_j^n}{\Delta t} \\ \frac{\partial^2 u}{\partial x^2} &= \frac{u_{j+1}^{n+1} - 2u_j^{n+1} + u_{j-1}^{n+1}}{\Delta x^2} \end{aligned}$$

A implicit numerical solution can be obtained.

$$CFL = \frac{\kappa\Delta t}{\rho c\Delta x^2}$$

$$-CFLu_{j-1}^{n+1} + (1 + 2CFL)u_j^{n+1} - CFLu_{j+1}^{n+1} = u_j^n + \frac{\kappa\Delta t}{\rho c}sin(l\pi x)$$

To solve the implicit equation, we need to solve the diagonal matrix first.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ -CFL & 1+2CFL & -CFL & 0 & \cdots & 0 \\ 0 & -CFL & 1+2CFL & -CFL & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & & \\ 0 & 0 & \cdots & -CFL & 1+2CFL & -CFL \\ 0 & 0 & 0 & 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} u_0^{n+1} \\ u_1^{n+1} \\ u_2^{n+1} \\ \cdots \\ u_{n-1}^{n+1} \\ u_n^{n+1} \end{bmatrix} = \begin{bmatrix} u_0^n + \frac{\kappa\Delta t}{\rho c} sin(l\pi x) \\ u_1^n + \frac{\kappa\Delta t}{\rho c} sin(l\pi x) \\ u_2^n + \frac{\kappa\Delta t}{\rho c} sin(l\pi x) \\ \cdots \\ u_{n-1}^n + \frac{\kappa\Delta t}{\rho c} sin(l\pi x) \\ u_n^n + \frac{\kappa\Delta t}{\rho c} sin(l\pi x) \end{bmatrix}$$

Then we calculate the analytical solution.the solution of the partial differential equation will converge to a steady state solution as time $t \to \infty$. In particular, the steady state is characterized by $\partial u/\partial t = 0$.

$$\begin{aligned} \kappa \frac{\partial^2 u}{\partial x^2} + sin(l\pi x) &= 0 \\ \frac{\partial u}{\partial x} - \frac{cos(l\pi x)}{l\pi} + c_1 &= 0 \\ u - \frac{sin(l\pi x)}{l^2\pi^2} + c_1 x + c_2 &= 0 \end{aligned}$$

form the initial condition we can get $c_1 = \frac{sin(l\pi)}{l^2\pi^2}, c_2 = 0$.

$$u = \frac{sin(l\pi x)}{l^2\pi^2} - \frac{sin(l\pi)}{l^2\pi^2} x$$

2 Stability analysis

From the above steps, we get the explicit solution and the implicit solution.The stability of them was analyzed by von Neumann. $CFL = \frac{\kappa\Delta t}{\rho c \Delta x^2}$

For explicit difference scheme

$$\begin{aligned} \delta u_i^{n+1} &= CFL\delta u_{j+1}^n + (1-2CFL)\delta u_j^n + CFL\delta u_{j-1}^n \\ \delta u_j^n &\sim e^{\sigma n\Delta t} e^{i(kj\Delta x)} \\ e^{\sigma\Delta t} &= CFLe^{ik\Delta x} + (1-2CFL) + CFLe^{-ik\Delta x} \\ &= |1 - 2CFL + 2CFLcos(k\Delta x)| \leq 1 \end{aligned}$$

$$\begin{aligned} -1 \leq cos(k\Delta x) &\leq 1 \\ 0 \leq CFL &\leq \frac{1}{2} \end{aligned}$$

For implicit difference scheme

$$\begin{aligned} \delta u_j^n &= -CFL\delta u_{j-1}^{n+1} + (1+2CFL)\delta u_j^{n+1} - CFLu_{j+1}^{n+1} \\ \delta u_j^n &\sim e^{\sigma n\Delta t} e^{i(kj\Delta x)} \\ e^{\sigma\Delta t} &= \frac{1}{2CFL(1-cos(k\Delta x))+1} \\ &= |\frac{1}{2CFL(1-cos(k\Delta x))+1}| \leq 1 \end{aligned}$$

unconditionally stable.

3 Technical details of development

Restart facility using HDF5

There are dedicated HDF5 reads and writes on PETSC. Read and write the file using the function PetscViewerHDF5Open. Vectors from a file can be read using the function VecLoad, and vectors can be written to a file using the function VecView. The design idea is to check whether the restart status is determined by the field. It does not run properly on restart and writes data to an HDF5 file every 10 iterations. If the HDF5 file is restarted, read the data from the HDF5 file first and continue with the above operations. Note how the data is stored, the number of data grids to be stored, the time interval between iterations, and the result vector generated by the last breakpoint. Because I can't find a function in pets that stores scalars into HDF5. So I store the above three scalars into a vector and access them together.As shown in the following figure.

```
if(!(j%10)){
    /*add t to save_value */
    for(i=hstart; i<hend; i+=hlocal){
        ierr = VecSetValue(save_value,i,n,INSERT_VALUES);CHKERRQ(ierr);
        ierr = VecSetValue(save_value,i+1,CFL,INSERT_VALUES);CHKERRQ(ierr);
        ierr = VecSetValue(save_value,i+2,t,INSERT_VALUES);CHKERRQ(ierr);
    }
    ierr = VecAssemblyBegin(save_value);CHKERRQ(ierr);
    ierr = VecAssemblyEnd(save_value);CHKERRQ(ierr);
    ierr = VecView(save_value,viewer);CHKERRQ(ierr);
```

The difficulty in writing this part is the problem of vector access to numbers. Because many functions in PETSc are based on the current process. Therefore, it is very difficult to access vectors accurately, because you may find that even if you specify the subscript, the subscript is based on the process, so you will find that my data appears under other subscripts. The way to solve this problem is to set the process space of the vector to 3, and access the three scalars in each process, although The figure below briefly shows my approach.

```
/* set vector save_value to store n,CFL,t*/
ierr = VecCreate(PETSC_COMM_WORLD,&save_value);CHKERRQ(ierr);
ierr = PetscObjectSetName((PetscObject) save_value, "save_value");
ierr = VecSetSizes(save_value,3,PETSC_DECIDE);CHKERRQ(ierr); // local set 3
ierr = VecSetFromOptions(save_value);CHKERRQ(ierr);
/* git the start and end of save_value in each cpu */
ierr = VecGetOwnershipRange(save_value,&hstart,&hend);CHKERRQ(ierr);
ierr = VecGetLocalSize(save_value,&hlocal);CHKERRQ(ierr);
ierr = PetscPrintf(PETSC_COMM_SELF,"rank = [%d] nlocal = %d hstart = %d hend = %d\n", rank, hlocal,hstart, hend);CHKERRQ(ierr);
/* get value from hdf5*/
if(restart){
    ierr = PetscViewerHDF5Open(PETSC_COMM_WORLD,"explicit.h5",FILE_MODE_READ,&viewer);CHKERRQ(ierr);
    ierr = VecLoad(save_value, viewer);CHKERRQ(ierr);
    ierr = VecView(save_value,PETSC_VIEWER_STDOUT_WORLD);CHKERRQ(ierr);
    for(i=hstart; i<hend; i+=hlocal){
        col[0] = i;col[1] = i+1;col[2] = i+2;
    }
    ierr = VecGetValues(save_value, 3, col, value);
    n = (int)value[0]; CFL = value[1]; t = value[2];
}
```

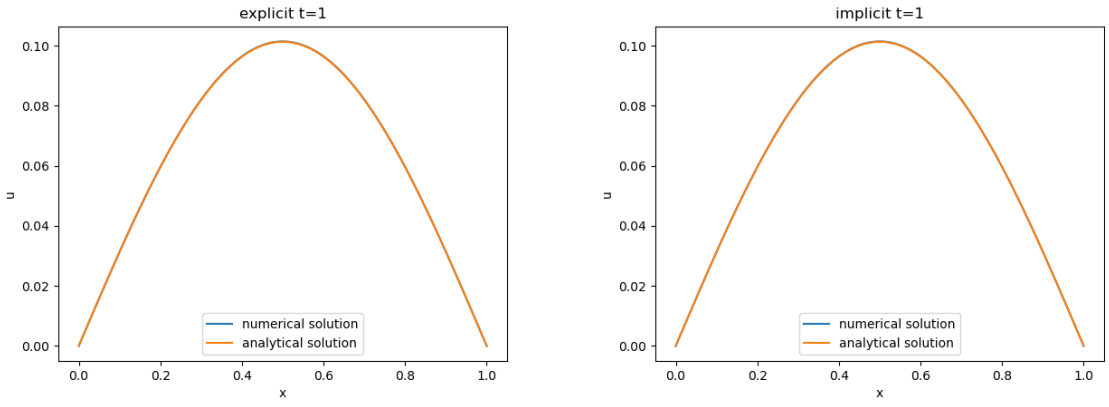
Valgrind

The report said that 10 bytes of data must be leaked. After discussion with classmates, it was found that the library file was leaked. Open –leak-check=full to detect and get warnings about some dynamic libraries in the library file, so it cannot be solved. The explicit and implicit reports are similar, so a detection screenshot of the explicit results is placed below.

```
==191795== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==191795== Using Valgrind-3.14.0 and LibVEX; rerun with -h for copyright info
==191795== Command: /share/intel/2018u4/compilers_and_libraries_2018.5.274/linux/mpi/intel64/bin/mpirun ./explicit.
==191795==
==191797==
==191797== HEAP SUMMARY:
==191797==     in use at exit: 57,780 bytes in 1,208 blocks
==191797==   total heap usage: 1,764 allocs, 556 frees, 95,855 bytes allocated
==191797==
==191797== LEAK SUMMARY:
==191797==    definitely lost: 0 bytes in 0 blocks
==191797==    indirectly lost: 0 bytes in 0 blocks
==191797==      possibly lost: 0 bytes in 0 blocks
==191797==    still reachable: 57,780 bytes in 1,208 blocks
==191797==         suppressed: 0 bytes in 0 blocks
==191797== Rerun with --leak-check=full to see details of leaked memory
==191797==
==191797== For counts of detected and suppressed errors, rerun with: -v
==191797== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

==191857==
==191857== HEAP SUMMARY:
==191857==     in use at exit: 60,119 bytes in 1,297 blocks
==191857==   total heap usage: 3,524 allocs, 2,227 frees, 150,006 bytes allocated
==191857==
==191857== LEAK SUMMARY:
==191857==    definitely lost: 10 bytes in 1 blocks
==191857==    indirectly lost: 0 bytes in 0 blocks
==191857==      possibly lost: 0 bytes in 0 blocks
==191857==    still reachable: 60,109 bytes in 1,296 blocks
==191857==         suppressed: 0 bytes in 0 blocks
==191857== Rerun with --leak-check=full to see details of leaked memory
==191857==
==191857== For counts of detected and suppressed errors, rerun with: -v
==191857== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==191795==
==191795== HEAP SUMMARY:
==191795==     in use at exit: 63,551 bytes in 1,281 blocks
==191795==   total heap usage: 3,796 allocs, 2,515 frees, 156,007 bytes allocated
==191795==
==191795== LEAK SUMMARY:
==191795==    definitely lost: 0 bytes in 0 blocks
==191795==    indirectly lost: 0 bytes in 0 blocks
==191795==      possibly lost: 0 bytes in 0 blocks
==191795==    still reachable: 63,551 bytes in 1,281 blocks
==191795==         suppressed: 0 bytes in 0 blocks
==191795== Rerun with --leak-check=full to see details of leaked memory
```

Visualization of results



(a) explicit (b) implicit

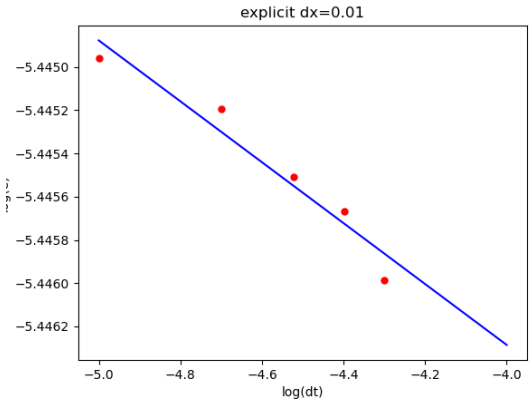
Use Python to visualize the data. When t=1, the results obtained are compared with the analytical solution. It can be found that the curve is consistent, and the coding of the two methods can be verified to be correct.

Note: all visualizations in this project are in Python

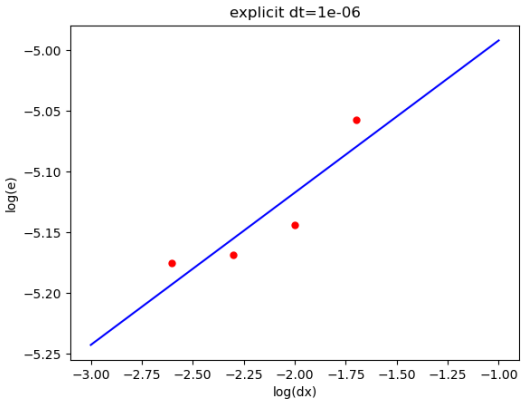
4 Manufactured solution method

The error is defined as $e := \max_{1 \le i \le n} |u_{\text{exact},i} - u_{\text{num},i}|$, where $u_{\text{exact},i}$ and $u_{\text{num},i}$ are the i-th component of the solution vectors with the vector length being n. The error is related to the mesh resolution as $e \approx C_1 \Delta x^{\alpha} + C_2 \Delta t^{\beta}$. To determine the value of α and β, you need to progressively refine your mesh and document the error value with the corresponding mesh size.

For explicit difference scheme



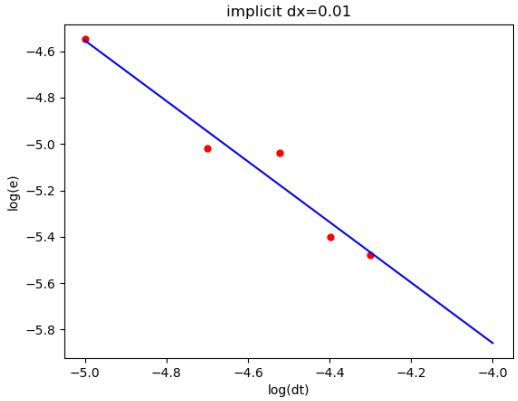
(c) dx=0.01



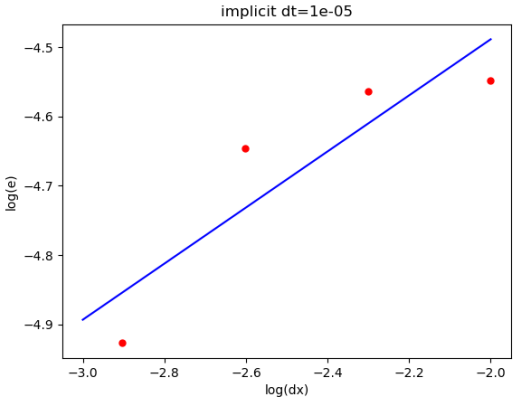
(d) dt=1e-05

Use a straight line to approximate the values of $log(e)$ against $log(\Delta x)$.We can find the slop in figure (a) is -0.001409. The slop in figure (b) is 0.12542. So $\alpha = 0.12542 \quad \beta = -0.001409$.

For implicit difference scheme



(e) dx=0.01



(f) dt=1e-05

Use a straight line to approximate the values of $log(e)$ against $log(\Delta x)$.We can find the slop in figure (a) is -1.305024. The slop in figure (b) is 0.404907. So $\alpha = 0.404907 \quad \beta = -1.305024$.

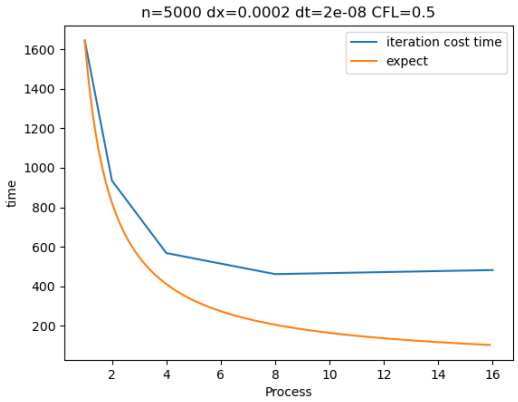
It can be summarized from the above results.The denser the grid, the smaller the error of numerical solution. So the accuracy will be higher.And it can be seen from the slope. Implicit solutions are more affected by spatial-temporal partition density than explicit solutions.

5 Parallelism

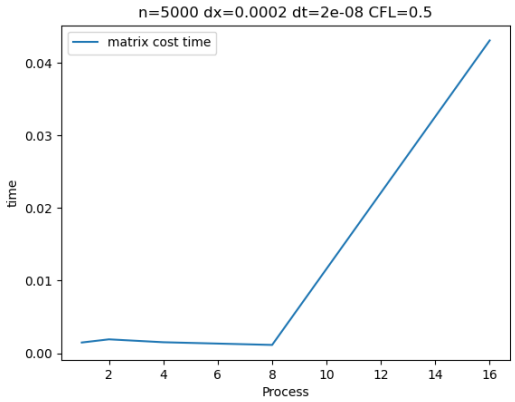
Based on different time and solutions, scalability can be divided into strong scalability and weak scalability. The characteristic of strong scalability is that when more processors are added, the scale of the problem itself will not increase. The characteristic of weak scalability is that as more processors are added, the scale of problems to be handled in each process remains the same.

For explicit difference scheme

Firstly, the strong scalability of the explicit solution is analyzed.



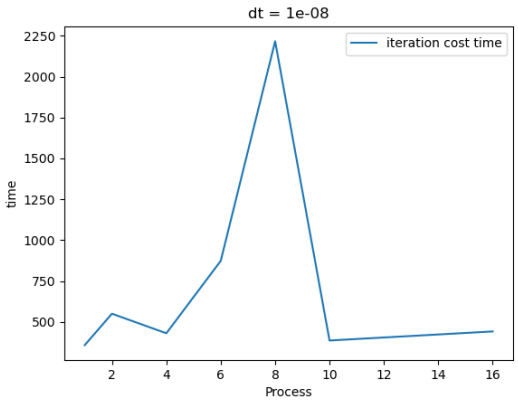
(g) explicit iteration



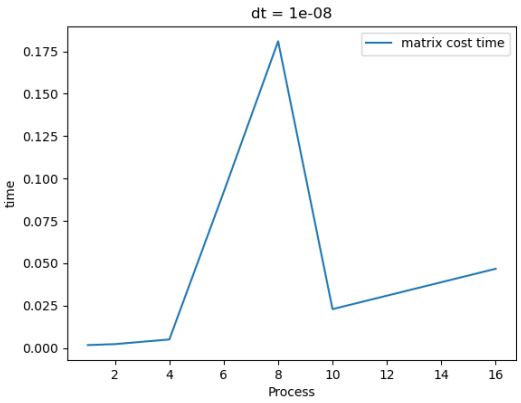
(h) explicit matrix

Figure (e) is a strong scalability analysis of iteration time. It can be found that as the number of processes increases, less time is spent. In an ideal state, the time spent on p cores should be equal to $\frac{1}{p}$ times the time spent on a single core. It can be seen from the figure that as the number of cores is greater than 8, the decline curve begins to pick up. Through analysis, this situation should be affected by the high proportion of processes communication. This situation can also be found from the time of matrix assembly in Figure (f).

Then we are going to analyze weak scalability.



(i) explicit iteration

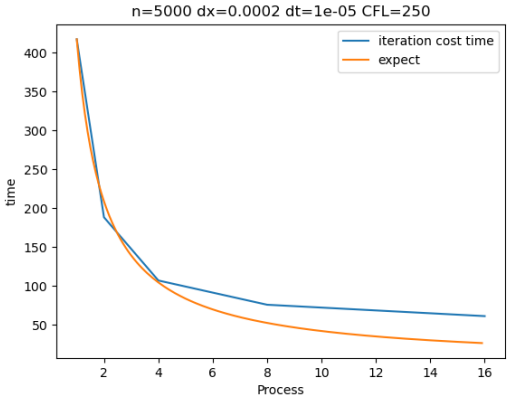


(j) explicit matrix

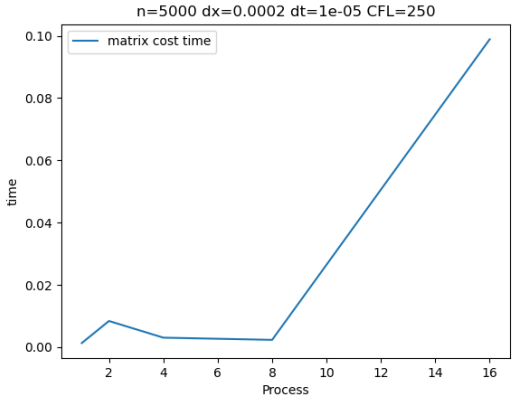
The weak scalability analysis is that the task size of each core is the same. As you can see from the figure, it takes a lot of time when processors is 8. It can be concluded that the weak scalability is poor under the current variable.

For implicit difference scheme

Firstly, the strong scalability of the implicit solution is analyzed.



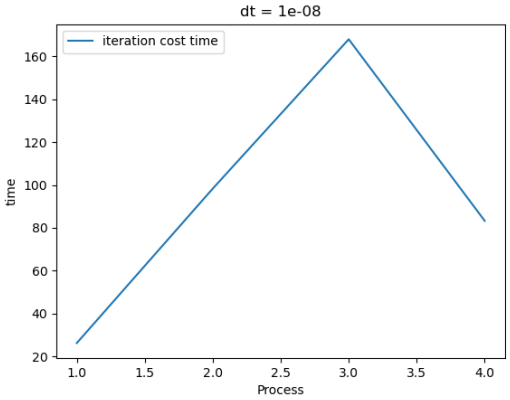
(k) implicit iteration



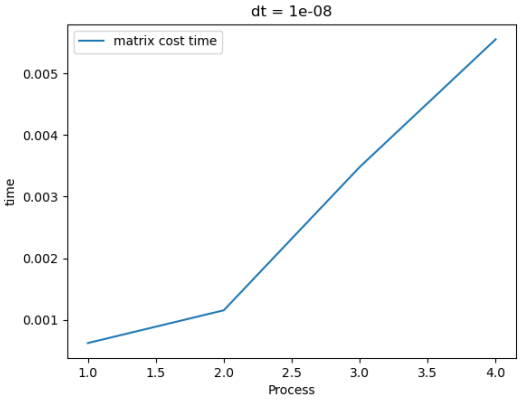
(l) implicit matrix

From the strong scalability analysis of the implicit solution, we can get the same trend as the explicit solution. It is more similar to the expected value than the explicit solution.

Then, the expansibility of the implicit solution is analyzed. Unlike the previous question on checking weak scalability, I increased the size of the work on each processor. When processors is 4, the number of grids is 4000.



(m) implicit iteration

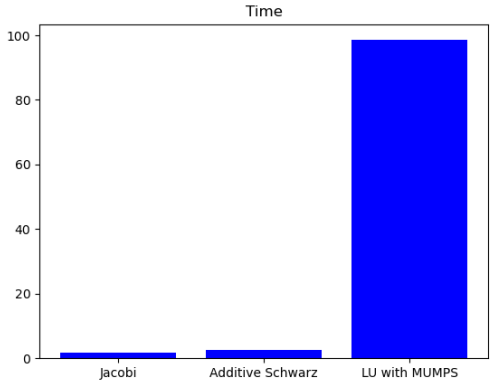


(n) implicit matrix

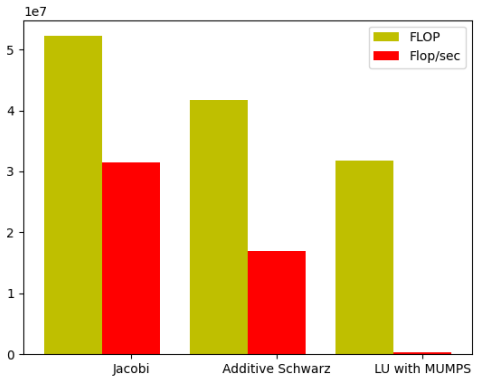
The expected result should be similar to the time spent, because the task quantity of each core is the same. However, from the difference in the figure, we can see that weak scalability is not good.

Investigate the following PC options from PETSc for your parallel test: (a) Jacobi; (b) Additive Schwarz; (c) LU with MUMPS.

From the figure below, we can see that using Jacobi takes the least time, while LU decomposition takes much more time than the other two methods. Looking at their floating-point performance, we can find that although the use of LU decomposition can reduce the total number of floating-point operations. However, it requires a large amount of data access, which reduces the amount of floating-point operations per unit time, so it takes the most time.


Time
100
80
60
40
20
0
Jacobi
Additive Schwarz
LU with MUMPS

(o) time


1e7
5
4
3
2
1
0
FLOP
Flop/sec
Jacobi
Additive Schwarz
LU with MUMPS

(p) flop