A parallel-in-time solver for the heat equation

Noah Brenowitz and Irena Vankova

May 18, 2015

1 Introduction

The aim of this project is to test the “Parallel-in-time” concept on the heat equation

\[ u_t = \Delta u \]

with periodic boundary conditions. The parallel-in-time strategy is distinct from any sort of spatial domain decomposition approaches, and we do not implement any spatial parallelism in this example. We choose a parallel in time strategy based on (Falgout et al., 2014). We solve the equation on two temporal grids, coarse and fine, where the solutions on the fine grid provide initial data for the the next coarse time step. Details are described in Section 3.

2 Heat equation solver

The serial implementation of the heat equation solver is relatively straightforward. The typical 2 dimensional discrete laplacian is used with a implicit trapezoid rule time stepping procedure (e.g. Crank-Nicolson). The advantage of the Crank-Nicolson scheme is that it is unconditionally stable like the backward Euler scheme, but also has second order accuracy.

The solver requires a uniform grid in both spatial dimensions. In detail, if the discrete laplacian operator is given by

\[(Lu)_{i,j} = u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j-1} - 4u_{i,j},\]

the time step is \( \tau \) and the spatial grid size is \( h \). Then the implicit trapezoid rule gives

\[ u^{n+1} = \frac{\lambda}{2} (Lu^n + Lu^{n+1}), \]

where the CFL number \( \lambda = \tau / h^2 \).

In practice, this iteration can be written as a matrix multiplication followed by a linear solve in the following way

\[ u^{n+1} = \left( I - \frac{\lambda}{2}L \right)^{-1} \left( I + \frac{\lambda}{2}L \right) u^n. \]

3 Code design

The code for this final project is available at [https://github.com/nbren12/hpc15final](https://github.com/nbren12/hpc15final). It consists of two parts:

- The serial heat equation solver provides a library in ./src/ called liblaplace.a with a simple function that evolves the heat equation forward in time with a given time step. The signature of this function is
  
  \[
  \text{void evolve_heat_equation_2d}(\text{double } *x, \text{ int } n, \text{ double } dx, \text{ int } nt, \text{ double } dt, \text{ int } \text{output_interval});
  \]

1
This header provides C-linkage, but the library is written in C++ using a MATLAB like linear algebra package called Armadillo++ for the sparse linear solve and matrix multiplications. Armadillo++ is essentially a high-level wrapper for libraries like BLAS, SuperLU, and LAPACK. The advantage over using MATLAB or numpy is that there is no temporary arrays formed with expressions like \( a = b + c \).

- To test the parallel in time idea, we fix special grid and the final time \( T \) and choose total number of time steps \( nt \). The fine grid time step is \( dt = T/nt \). We compute solution on this fine grid by calling evolve_heat_equation_2d and this fulfills the purpose of exact solution \( u_{\text{exact}} \) for the comparison of time parallel tests. Next we choose a coarse time step \( dt_c \), and number of coarse time steps \( nt_c \) will equal the number of processors used, \( np \). We evolve the equation on the coarse time grid, and we take the coarse grid values as initial values to be evolved independently on a fine grid of length \( nt/nt_c \) on each of the \( np \) processors. This process is repeated maximum of \( np \) times and in each iteration one less processor is used. After each iteration we compute the relative error \( E = \sqrt{\sum (u_{\text{exact}} - u)^2 / \sum (u_{\text{exact}})^2} \) and we set a tolerance of 10%, to stop iterating when \( E < 0.1 \). There are time savings when the number of iterations is smaller than number of iterations required converge exactly to the fine grid solution \( (np) \).

### 4 Scalability

We ran tests with 640,1280, 2560, 5120 and 10240 total time steps \( (nt) \) and for 5, 10, 20, 40 and 80 time steps per processor. This together gave us 25 cases to compare. The measure of savings in time we use is the ratio of required iterations to get within the tolerance to \( np \), (percentage of time saved). The results are summarized in Figure[1]. On the top graph each color represents fixed amount of time steps per processor (coarse block). As we refine the time step \( (= \) increase the number of coarse blocks), the utility of parallelization in time improves. Due to the simplicity of heat equation, at some point the refined solution is as good as the coarse solution (that is what 100% time saved means). On the lower graph each color represents fixed \( nt \). Given that, parallelizing with a higher number of coarse blocks improves the performance in time. Figure[2] shows that the convergence to the solution on the corresponding fine grid is exponential, for some sample runs.

### 5 Conclusion

Once an algorithm is perfectly spatially parallelized, it may be possible, based on the system of equations, to speed up the computation by parallelizing in time. We tested this idea on heat equation in two dimensions, and obtained good results, which we attribute mainly due to the simplicity of the equation. Although an interesting idea, we found this approach rather wasteful, due to the amount of processors needed for repetitive iterative calculations, in order to achieve significant time savings. This conclusion however maybe a result of personal philosophy rather than anything else.

### References

Figure 1
Figure 2