# Solvers for Heat Transfer Problem

Wei Wang, weice@bu.edu
Yibo Zhao, yibozhao@bu.edu
Ziyan Chen, zychen02@bu.edu
Shining Yang, shiningy@bu.edu

## Introduction

Consider a heat transfer problem like this: in a L*L grid $(L = 32, 64, \ldots 512)$, apply a dot source in the right middle of it at t=0, and let the heat naturally flow with time. Let's assume the discrete heat transfer function is

$$T[x, y, t] - [T[x-1, y, t] + T[x+1, y, t] + T[x, y-1, t] + T[x, y+1, t]] + m^2/2 * T[x, y, t] = \sum source$$

where m^2 is a damping term and let it equals 0.0001. This article discusses 4 different algorithms to accelerate the calculation of this problem.

## Red-black Gauss-Seidal

### Background

One of the most basic and common solver algorithm for heat transfer problem is Gauss-Seidal solver. By updating the value with its neighbors at the current iteration instead of last iteration, Gauss-Seidel will converge to a fixed residual in half the number of residuals of Jacobi iterations. However, in the common practice, Jacobi's performance is much better than Gauss-Seidal because it can be fully parallelized, while Gauss-Seidal cannot, due to the data dependencies.

To decouple the data dependencies of Gauss-Seidal to make it parallelizable, people took a look at its updating formula:
T[x][y] = (1-r)*T[x][y]+r/4*(T[(x+1)%L][y] + T[(x-1+L)%L][y] + T[x][(y+1)%L] + T[x][(y-1+L)%L]);

and they found that the access pattern of points is a bipartite lattice, which means updating the even points (the points (x,y) that satisfied (x+y)%2=0) only depends on odd points, and updating the odd points only depends on the even points. And this finding provides an opportunity to parallelize the calculation by splitting the formula into even-odd updates: In each iteration, we first update all the even points and then update all the odd points. Since this only depends on the site itself and its neighbors, there are no dependencies issues and can be calculated out of order and parallelize.

Therefore, a naive updating iteration of red-black Gauss-Seidal with periodical boundaries is:

```
for (i = 0; i < N; i++) {
    for (j = (i % 2); j < N; j += 2) {
        T[i][j] = scale * (T[i][(j - 1 + N) % N] + T[i][(j + 1) % N] + T[(i - 1 + N) %
N][j] + T[(i + 1) % N][j]) + b[i][j];
    }
}
for (i = 0; i < N; i++) {
    for (j = ((i + 1) % 2); j < N; j += 2) {
       T[i][j] = scale * (T[i][(j - 1 + N) % N] + T[i][(j + 1) % N] + T[(i - 1 + N) %
N][j] + T[(i + 1) % N][j]) + b[i][j];
    }
}
```

In our testbed (1 V100 GPU, N=512, error threashold=10^-6), it took 193.37 second to converge (lower than the error threshold).

## RB with OpenACC and loop collapse

To parallelize this algorithm, let's first add the OpenACC derivatives:

```
#pragma acc data copy(T[0:N][0:N]), copy(b[0:N][0:N])
#pragma acc parallel loop
```

The first derivative copies T and b arrays in and out between CPU and GPU, and the second one tries to parallelize the loop (we tried with collapse(2), but failed with compiler error). It took 43.929 second to converge with 77.28% acceleration beyond the naive implementation.

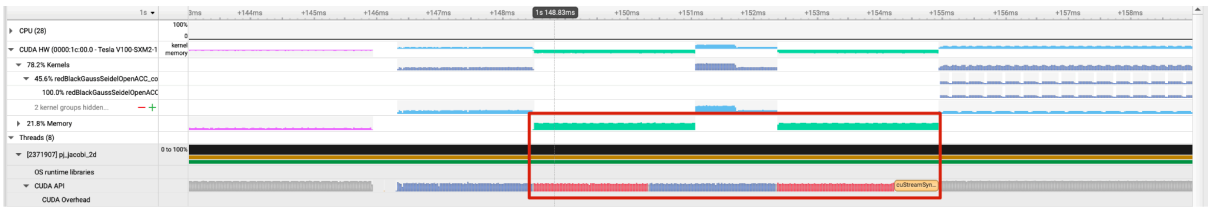
In the compile information, we found the compiler cannot fully parallelize the 2-layer nested loop because they think there are data dependencies between them. Hence, we tried to collapse the nested loop manually to ensure the loop is fully parallelized. We change the nested loop to:

```
for (index = 0; index < N * N; index++) {
    int i = index / N; int j = index % N;
    if ((i + j) % 2 == 0) {
```

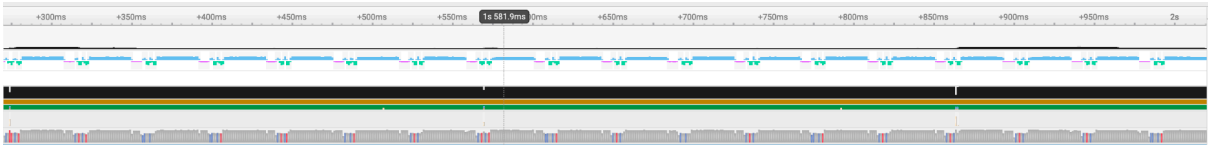
and the gain is significant: this implementation only took 6.095 second to converge, achieving 96.85% acceleration beyond the naive implementation and 86.13% beyond the OpenACC implementation.

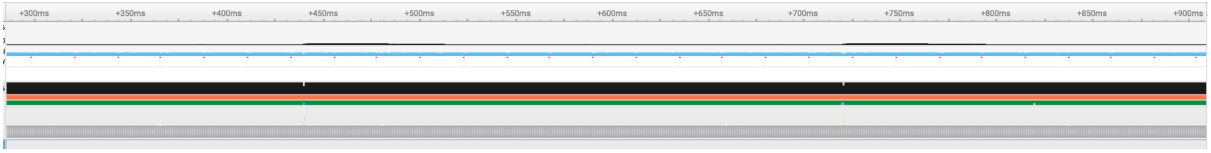
## RB with OpenACC and data transferring optimization

To further accelerate the implementation, I ran a profiling of the loop collapse code with Nsight System. The result shows there is a great portion of time running data transfer, and not masked by computation.



In the profiler output, we found the data transferring is really frequent, and this is because the OpenACC part is executed within the RedBlack function, resulting in a data transfer every time the function is called. So the optimization is to move the update iteration to main function, and move the residue calculation to GPU as well, ensuring that only one data transfer is necessary. Moreover, we don't need to copy array b back to CPU, since b remains unchanged during the update process.



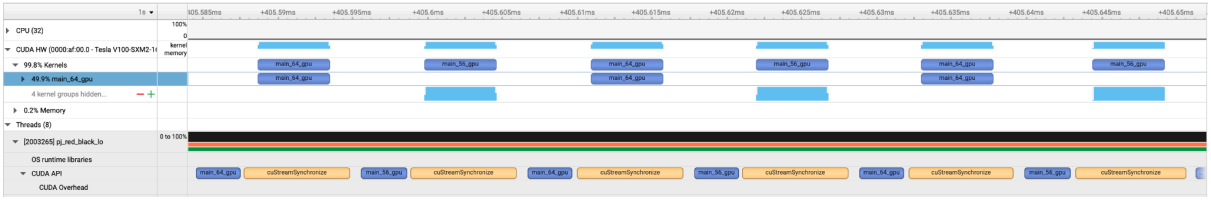


From the profiler output before and after theses optimization, we can find that most data transferring process (the lower bars with red and blue) is obviate, and it took 3.569 second to converge, 41.44% faster than the loop collapse version.

## Extra Optimization Exploration

One common way to accelerate the GPU computation is using low-precision calculation, because it can save GPU memory usage (especially shared memory), and have higher arithmetic intensity. In our experiment, when using FP32 instead of double, it brings 26.70% acceleration. However, A low-precision implementation's effectiveness greatly depends on the specifics of the problem. If the precision is insufficient, it may fail to converge, necessitating a conversion to higher precision in such scenarios.

Another observation is also from the profiler output: after each single iteration, it needs to do a global synchronization to ensure the T is new in the next iteration. Cuda do it by reloading the kernel, and the synchronization and kernel loading overhead makes the program much slower.

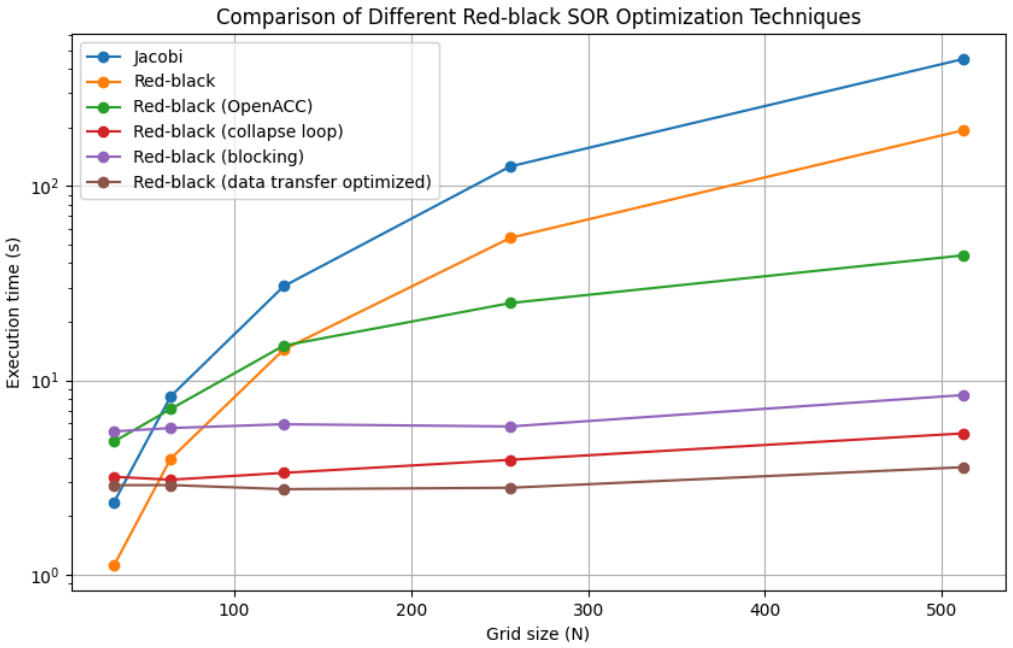


One solution is using blocking blocking to obviate the global synchronization. However, a simple blocking using OpenACC derivative does not work, it is even 36% lower in our experiment. There are some fancier blocking techniques proposed by stencil computation

researcher, for example, Shan et al. compares different blocking strategies in stencil computation, like 3.5D blocking and 2.5D blocking.

## Result

We compare the time consuming of different implementation for N=32 to N=512, and here is the reuslt.




The algorithmswith loop collapse exhibit good parallelism, as the execution time remaining relatively consistent across different N. The slight increases observed may be attributed to the increase in data transfer sizes.

# Multigrid Solver

## Background

Multigrid is an efficient solver algorithm renowned for its ability to swiftly alleviate errors across various frequencies, hence accelerating the solving process. In Multigrid, each iteration has several layers with grids of varying granularity. Every layer undergoes relaxation, accompanied by downscaling and patching up to establish connectivity with adjacent layers. The overall residual root is computed, marking the transition to the subsequent iteration.

Within Multigrid, operations such as relaxation, downscaling, patching up and getResRoot are computationally intensive. These tasks are amenable to parallel optimization using frameworks like OpenMP, OpenACC, and OpenMPI. In this project, we implemented the

Jacobi algorithm within the Multigrid and leveraged the parallel computing capabilities of both OpenMP and OpenMPI to enhance computational efficiency.

### Parameter Setting for Multigrid

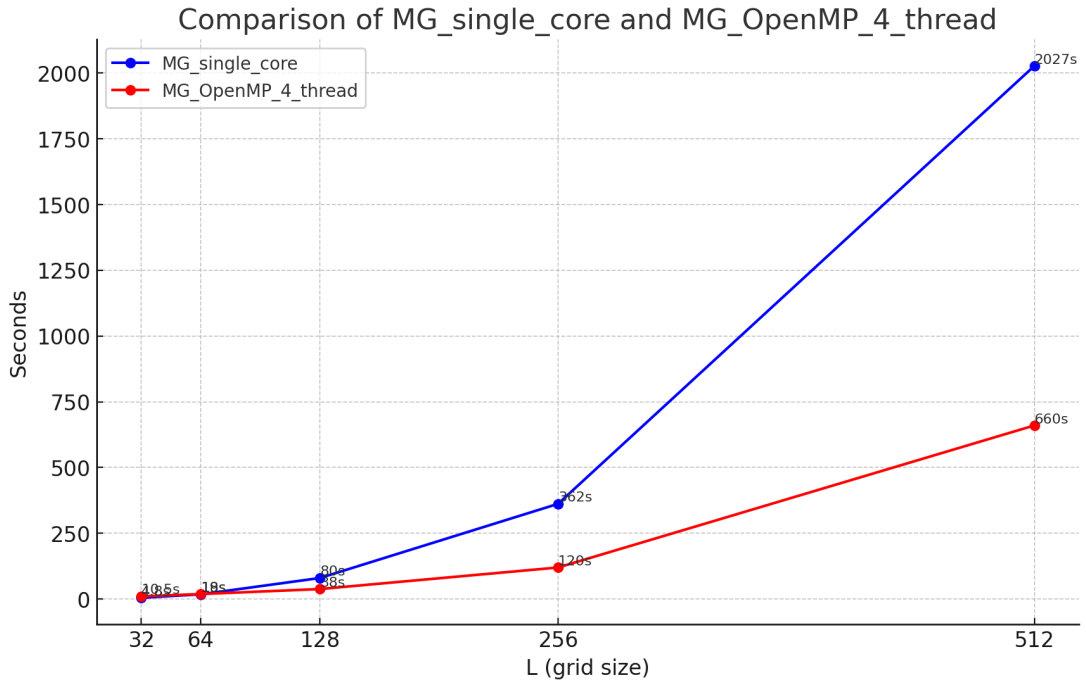
| | |
|---|---|
| 2D Plane Size | L*L (32~512) |
| Internal Cycle in Relaxation | 10 |
| Number of Layers | $\log_2(L) - 1$ |

## Parallel with OpenMP

The Jacobi method updates the values in a matrix based on the previous iteration's values. This process inherently involves using an old array of data to update a new array. Given the nature of this operation—where the update of each element is independent of others within the same iteration—it lends itself well to parallel execution.

To implement parallelism, as the relaxation, downscaling and patching up have no data dependency, we can simply annotate the primary loop that iterates over the data array with OpenMP's `#pragma omp parallel for`. For getResRoot, all the cores need to update the same memory, as it is a sum option, we can use omp reduction to solve this.

### Results Comparison and Analysis




We compared a single-threaded multigrid and a 4-thread OpenMP-accelerated multigrid on the same machine, as shown in the graph above. The blue line represents the

single-threaded Multigrid, while the red line represents the Multigrid using OpenMP. We can see that at a smaller size (L=32), OpenMP is slower than single-threaded Multigrid. The reason is that the smaller size does not demand high computational power, and the introduction of OpenMP leads to resource management among threads, making it slower than the single thread. However, as the size increases, the speed difference between OpenMP and the single thread widens. At L=512, the speed of OpenMP is three times that of the single thread.

# Parallel with Open MPI

For OpenMPI, in our scenario, we distribute a two-dimensional plane evenly across different ranks by rows. Each rank stores all the data, but only the data it is responsible for is up-to-date.

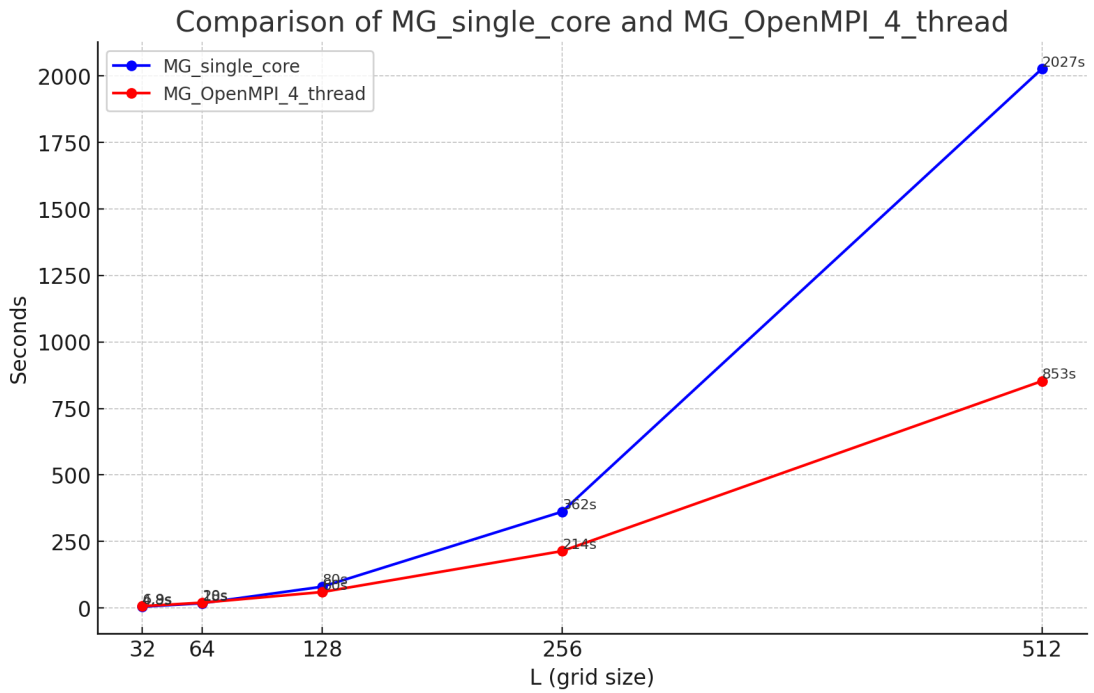
**Overlap computation and communication**: To achieve overlap between computation and communication in the relaxation process, we first compute the inner data, then move on to the edge data. This allows us to send and receive the edge data necessary for computation from other ranks while the inner data is being processed.

**Trade off between optimal speed and code complexity**: For operations such as downscaling, patching up, and getResRoot, each rank should theoretically determine which rank to fetch data from based on the operation's requirements. For example, in downscaling, each row needs to compute its double line number and communicate with the corresponding rank. Since we distribute data by rows, ranks with lower row numbers have to fetch more data from other ranks, resulting in heavier communication loads on these ranks. This can make these smaller ranks a communication bottleneck.

To simplify code design, we use an allgather operation after the relaxation process, allowing all ranks to update their data. Each rank can then perform local downscaling and patching up operations independently. This approach simplifies the programming design and, although it might not offer optimal speed theoretically, it should not be significantly slower compared to a design where communication bottlenecks at smaller ranks slow down the entire process.

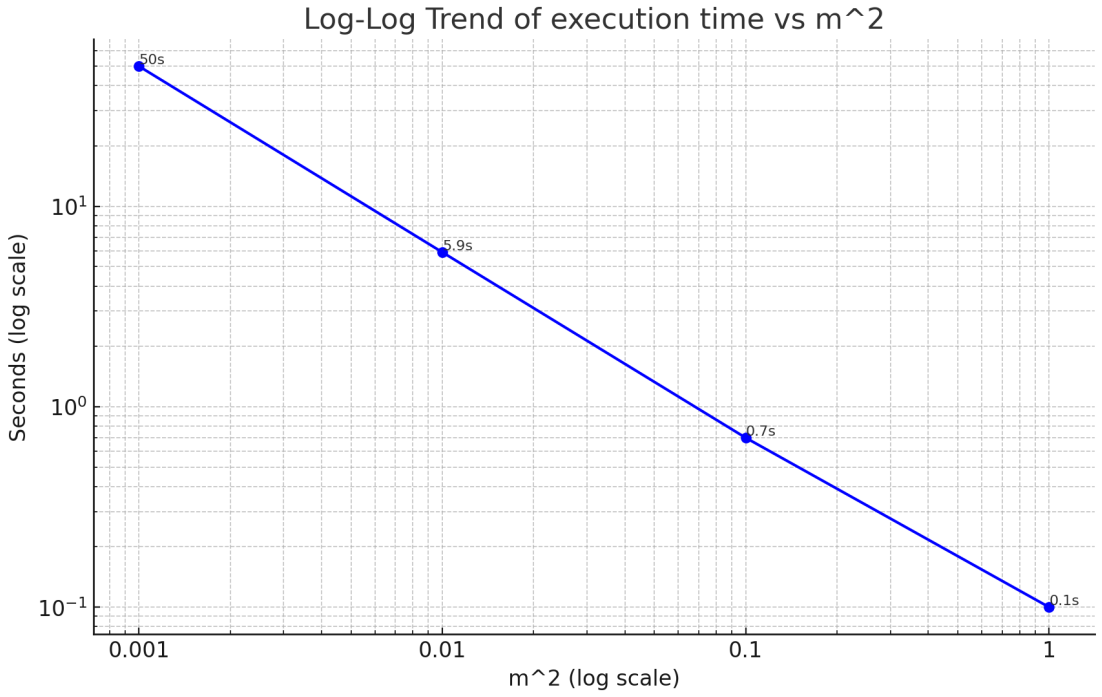
## Results Comparison and Analysis

We compared single thread multigrid and 4 threads using MPI on the same machine, and the results are shown in the figure above. The blue line is single-threaded Multigrid, and the red line is using OpenMPI. Also when the size is small (L=32), MPI is slower than Multigrid because MPI introduces additional communication and does not completely overlap the calculation, resulting in it being slower than a single thread. As L increases, the speed of MPI gradually becomes faster than that of a single thread. When L=512, the speed of MPI is 2.5 times that of a single thread. Not achieved 3x like OpenMP, which is expected due to the trade-off design.




## Compare Speed with Different M^2

In order to explore why multigrid has a significant time difference compared to other algorithms at $L$=512, I have tried modifying $m$^2. As can be seen from the figure below, as the order of magnitude of m^2 changes, the speed of multigrid also changes by orders of magnitude. However, multigrid is still slower than conjugate gradient (CG) for all the data sizes we have tested till now.



# Fast Fourier Transformation (FFT)

## Background

Fast Fourier Transformation ( FFT )is a widely used method for discrete signal analysis. FFT changes the data from space domain to frequency domain, making many calculations easier, faster and parallable.

For this problem we use Cooley-Tukey FFT Algorithm to

## Cooley-Tukey Algorithm and Butterfly Diagram

Named after J.W.Cooley and John Tucky, C-T algorithm is the most common method for FFT. It recursively breaks the problem into two halves, and apply FFT on them. Then we manipulate the data in the frequency domain and return it to the spatial domain.

Let's rewrite the FT procedure by a twiddle factor $(\omega_N^n)^k$

$$y_k \equiv FT_N[c_n] = \sum_0^{N-1} e^{i2\pi nk/N} a_n = \sum_0^{N-1} (\omega_N^n)^k a_n$$

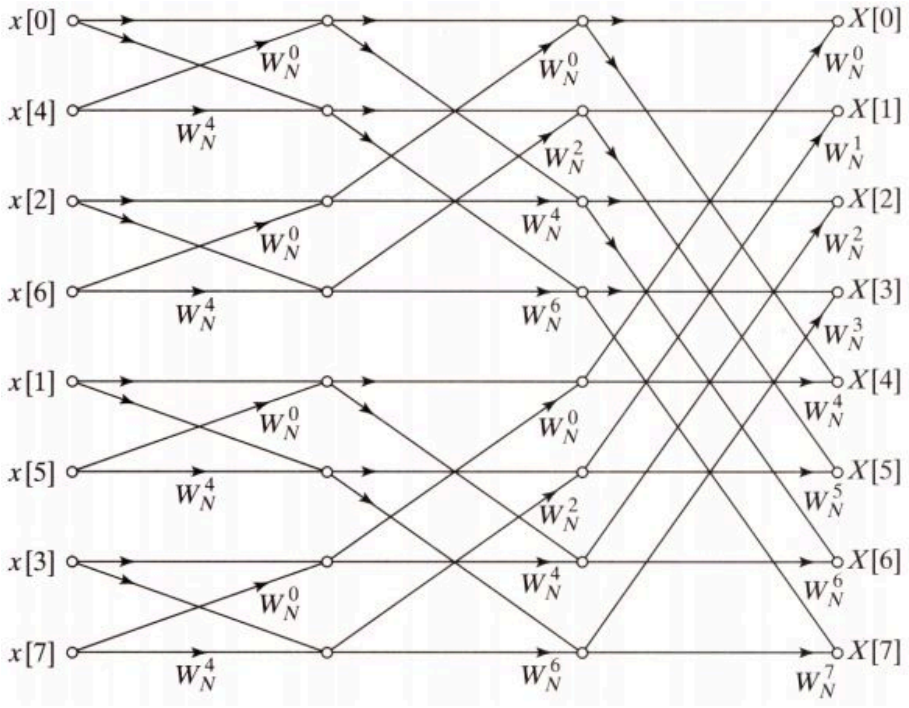
Separate n by even and odd, we have

$$y_k = \sum_0^{N/2-1} \exp[i\,2\pi nk/(N/2)] a_{2n} + \omega_N^k \sum_n^{N/2-1} \exp[i\,2\pi nk/(N/2)] a_{2n+1} = G[k + N/2] + \omega_N^k H[k]$$

Notice that G and H have period of N/2, and $\omega_N^{k+N/2} = -\omega_N^k$ , the whole procedure is divided into 2 smaller FFT with size of N/2. Then we have when $k = 2\bar{k}$ *or* $2\bar{k} + 1$,

$$y_{2\bar{k},\, or\ 2\bar{k}+1} = \sum_0^{N-1} e^{i2\pi n\bar{k}/(N/2)} [a_n \pm a_{n+N/2}]$$

This is the butterfly diagram example [4] for a FFT of N=8. For inverse FFT, simply do the graph below from right to left. For those N<2^k (k=2,...), add zeros in the end to make it length 2^k.




All those do the same for a 2D grid. For a grid G with size L*L, first do zero-padding to make it the size to the nearest 2^k*2^k, let's call it G'. Then, considering FFT only does well on periodic data, extend it as $G'' = \begin{pmatrix} G' & -G' \\ -G' & G' \end{pmatrix}$ to make it periodic in both dimensions. Regard G'' as a list of numbers with length of (4*2^k*2^k), and we can do the FFT and inverse FFT as before.

# Parallel with OpenAcc

We parallel the problem as below:

```
              // perform FFT on initial grid
            FFTrecursion(px, px, omega, L, Nfft);
            // iterate using OpenACC
#pragma acc data copy(px[0 : extended_size * extended_size]), create(px_new[0 :
extended_size * extended_size])
            for (iter = 0; iter < MAX_ITER; iter++)
            {
#pragma acc kernels loop independent
                for (i = 0; i < L; i++)
                {
                    for (j = 0; j < L; j++)
                    {
                        int idx = i * L + j;
                        px_new[idx] = (1 - R) * px[idx] + R / (4 + m_squared) * (px[((i
+ 1) % L) * L + j] + px[((i - 1 + L) % L) * L + j] + px[i * L + ((j + 1) % L)] + px[i *
L + ((j - 1 + L) % L)]);
                    }
                }

                // swap pointers
                Complex *tmp = px;
                px = px_new;
                px_new = tmp;
            }
```

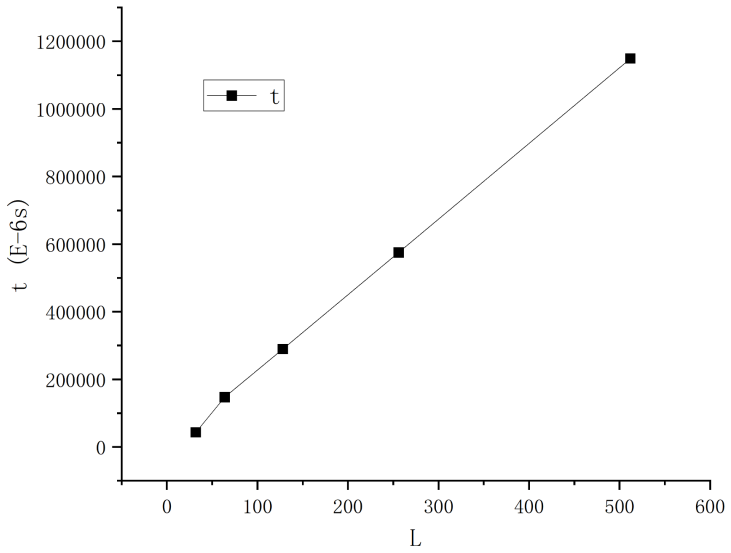
```
            // perform inverse FFT
            FFTrecursion_inv(px, px, omega, L, Nfft);
```

The whole procedure is simple, just:

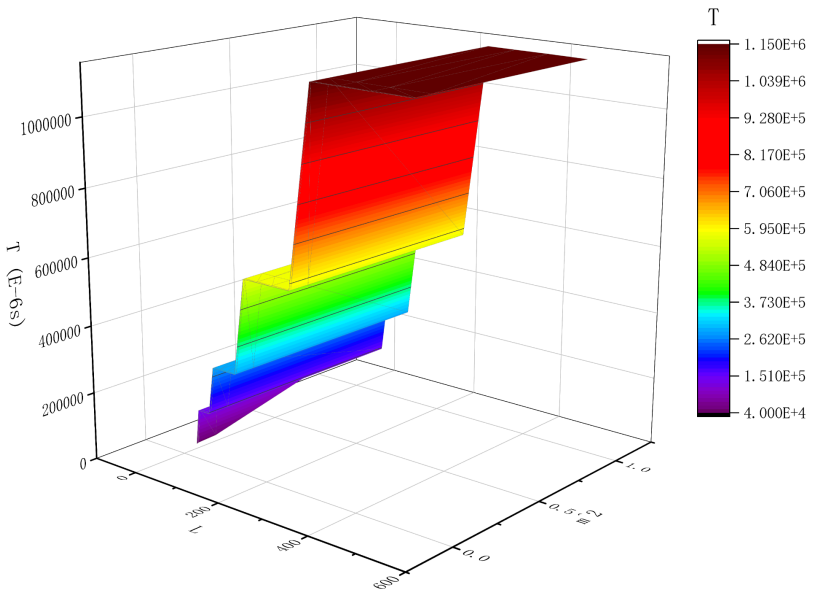
1. Do FFT;
2. Iterate in frequency domain. OpenAcc is applied here to accelerate the iterations;
3. Do inverse FFT and we have the final solution.

# Results

1. For those L=2^k and m^2==0.001, we have the following graph:




   We can see that time goes linearly with L (actually t=2271.22L-9845.29, with R^2=0.9992) as expected.
2. Adding more L and m^2 into consideration, say more L and $m^2 \in [1, 0.001]$, we have the following result:



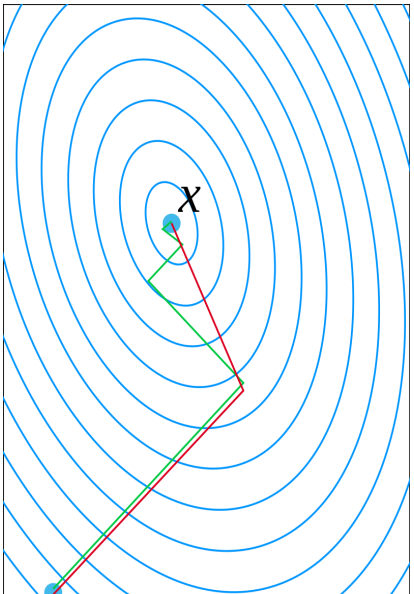

a) There are many platforms between 2^k and 2^(k+1). The reason is that, to best utilize the speed of FFT, those grid G with size L: $2^k < L < 2^{k+1}$ will always do zero-padding until L=2^(k+1). And FFT doesn't care whether there's a zero or not.
b) The data has almost no change in the direction of m^2. From the formula we know that m^2 term acts only as a coefficient, which will almost don't affect the time —- except m^2=1 and L=32, since the time for convergence at this time takes more time due to a large m^2.

# Conjugate Gradient

## Background[2]

The Conjugate Gradient (CG) method is an iterative algorithm commonly used to solve large sparse linear systems of equations. It's particularly efficient for solving symmetric positive definite (SPD) systems, which are common in many scientific and engineering applications.

Origin: The Conjugate Gradient method was initially developed by Cornelius Lanczos in 1952, and later refined by Magnus Hestenes and Eduard Stiefel in 1952. The method was originally proposed for solving systems of linear equations arising from finite difference methods applied to partial differential equations.




Iterative Solver: Unlike direct methods such as Gaussian elimination, which aim to find the exact solution to a linear system in a finite number of steps, the Conjugate Gradient

method is iterative. It generates a sequence of approximate solutions that converge to the true solution as the number of iterations increases.

Conjugate Directions: The key idea behind the method is to generate a sequence of search directions that are conjugate to each other with respect to the matrix of the linear system. Conjugate directions ensure that each step moves closer to the solution in a direction orthogonal to the previous steps, which accelerates convergence.

Algorithm: At each iteration, the method computes a search direction by combining the residual (the difference between the current approximation and the actual solution) with the previous search direction. It then performs a line search along this direction to find the optimal step size that minimizes the residual in that direction. The Conjugate Gradient method has favorable convergence properties. In exact arithmetic, it converges to the solution in at most N steps, where N is the size of the system. However, in practice, the number of iterations required for convergence depends on factors such as the condition number of the matrix and the convergence criterion used.

## Algorithm Implementation

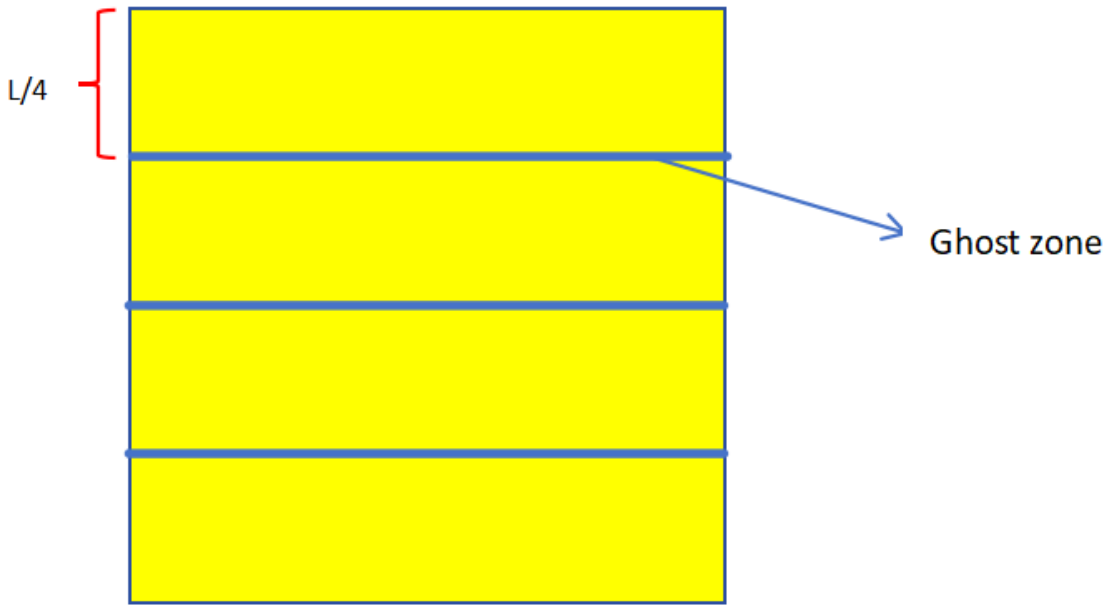
We have an iterative version of the CG algorithm:

To realize this in our heat flow problem, we want to first transform the Laplacian function into a linearized version

T[x][y] - 0.25*(T[x-1][y] + T[x][y-1] + T[x+1][y] + T[x][y+1]) + m^2 * T[x][y] = a****Source***

$\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$
$\mathbf{p}_0 := \mathbf{r}_0$
$k := 0$
**repeat**
$\alpha_k := \frac{\mathbf{r}_k^{\mathrm{T}}\mathbf{r}_k}{\mathbf{p}_k^{\mathrm{T}}\mathbf{A}\mathbf{p}_k}$
$\mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k\mathbf{p}_k$
$\mathbf{r}_{k+1} := \mathbf{r}_k - \alpha_k\mathbf{A}\mathbf{p}_k$
**if** $\mathbf{r}_{k+1}$ is sufficiently small **then** exit loop **end if**
$\beta_k := \frac{\mathbf{r}_{k+1}^{\mathrm{T}}\mathbf{r}_{k+1}}{\mathbf{r}_k^{\mathrm{T}}\mathbf{r}_k}$
$\mathbf{p}_{k+1} := \mathbf{r}_{k+1} + \beta_k\mathbf{p}_k$
$k := k + 1$
**end repeat**
The result is $\mathbf{x}_{k+1}$

Here we choose a=1 and m=0.01, which means m^2 = 0.0001. Different m would also affect the performance of our application.




To realize the mpi parallelization, we Split the data block into four parts along the x-axis.

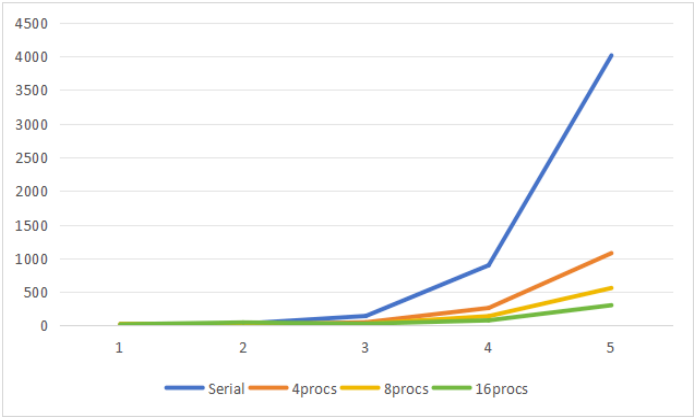
We use the Ghostzone to inform other processors of the boundary data. In this way we can avoid the complicated situation when every processor has to do vertical and horizontal boundary exchange at the same time. Thus simplifying the code complexity.

# Test Result

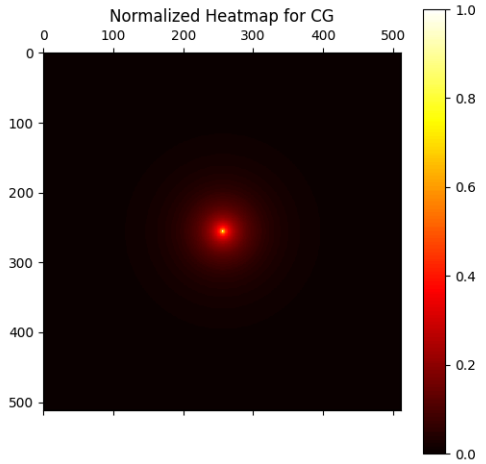
To explore the performance of CG on heat flow problem, we set a list of test groups:

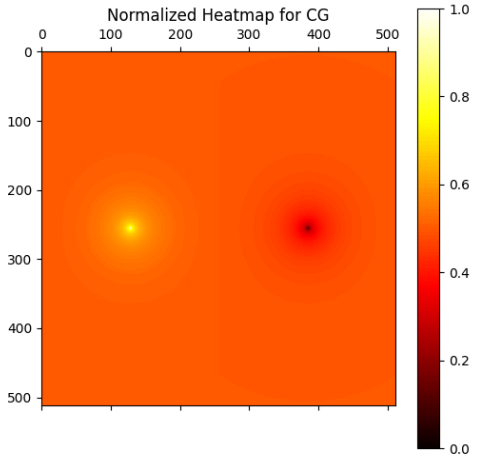
(m = 0.01 and time in unit of ms)

| | 32*32 | 64*64 | 128*128 | 256*256 | 512*512 |
|---|---|---|---|---|---|
| 1 proc | 7 | 28 | 142 | 896 | 4015 |
| 4 proc | 19 | 12 | 48 | 261 | 1076 |
| 8 proc | 23 | 17 | 27 | 139 | 558 |
| 16 proc | 18 | 46 | 28 | 77 | 302 |




We also have tested for single source and double sources condition:



# Conclusion

In this article, we discuss 4 different solvers to solve the heat transfer problem, each method has its own way to optimize the speed and accelerate the process. Among all 4 methods, CG is the fastest, followed by Gauss-Seidel and FFT. MG is the slowest (660s when L=512! ), it may be caused by parameter settings of the inter cycle of relaxation.

# Reference

[1] Conjugate gradient method, Wikipedia
[2] Cooley–Tukey FFT algorithm, Wikipedia