Solving the Steady State Heat Equation Accelerated with OpenACC

Heat Equation

-

- The Heat equation is a partial differential equation which governs the movement of heat via conduction within a material.
- It is given by

$$\frac{\partial u(x,y,z)}{\partial t} = \alpha \nabla^2 u(x,y,z) = \alpha \left[\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right]$$

- Where u is the temperature, α the thermal diffusivity coefficient (which related to the thermal conductivity (k), density (ρ) and specific heat capacity (c) as $\alpha = \frac{k}{c\rho}$), t is time and x,y, and z are the usual cartesian coordinates

Steady State Heat Equation

-

- A particularly interesting case is the steady state problem where

$$\frac{\partial u(x,y,z)}{\partial t} = 0$$

- Thus

$$0 = \alpha \nabla^2 u(x,y,z) = \alpha \left[\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right]$$

Method of Lines/Finite Difference

-

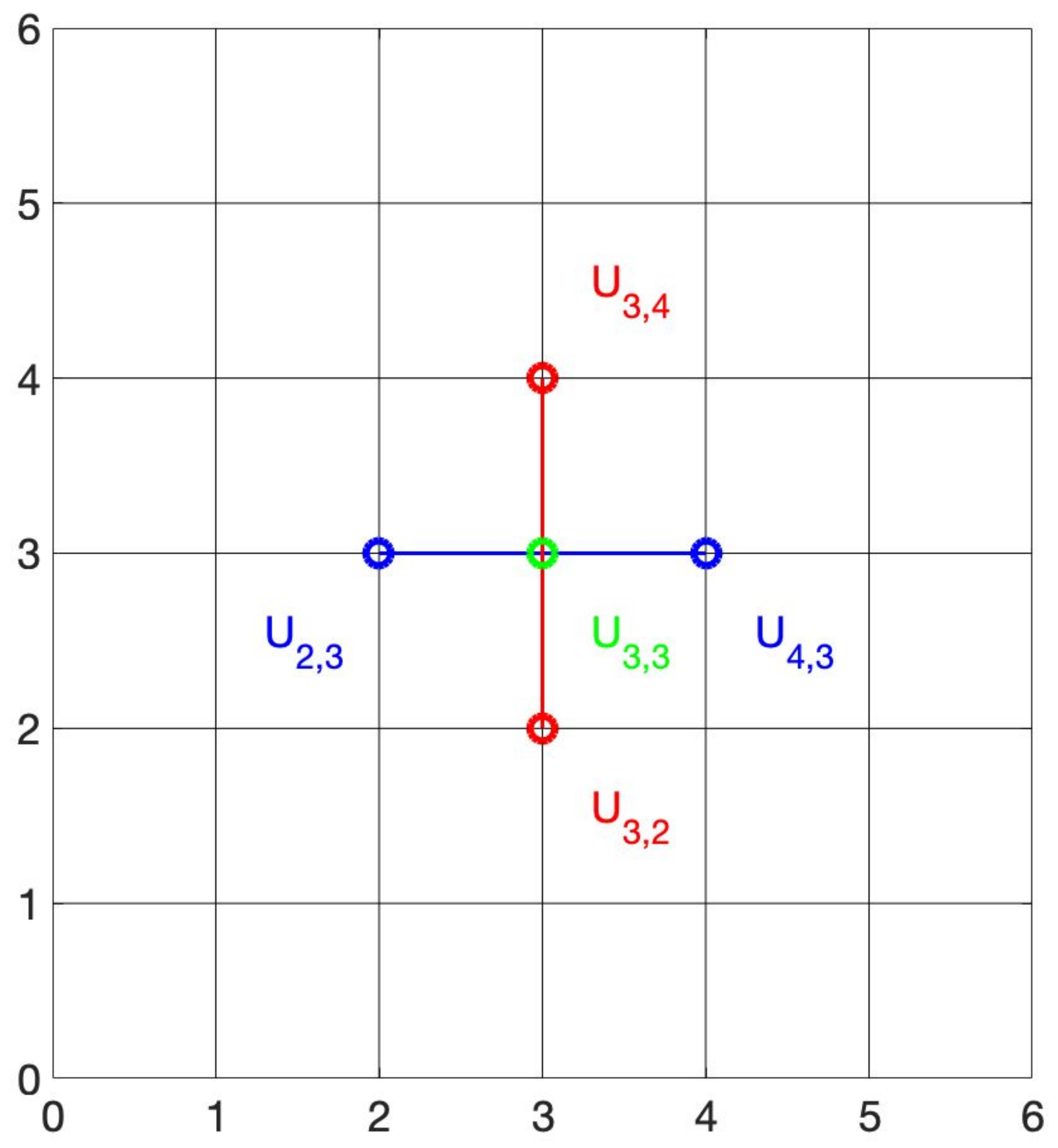
- The derivatives on the right hand side can be approximated as finite differences, for instance

$$\frac{\partial^2 u(x,y)}{\partial x^2} \approx \frac{1}{\Delta x}\left[\frac{u_{4,3} - u_{3,3}}{\Delta x} - \frac{u_{3,3} - u_{2,3}}{\Delta x}\right]$$

$$\frac{\partial^2 u(x,y)}{\partial x^2} \approx \left[\frac{u_{4,3} + u_{2,3} - 2u_{3,3}}{(\Delta x)^2}\right]$$

- Similarly

$$\frac{\partial^2 u(x,y)}{\partial y^2} \approx \left[\frac{u_{3,4} + u_{3,2} - 2u_{3,3}}{(\Delta y)^2}\right]$$



Linear Equations

-
- Using the finite difference approximation we can turn the partial differential equation into a linear one. Thus the steady state heat equation in 2D becomes

$$\left[\frac{u_{i+1,j} + u_{i-1,j} - 2u_{i,j}}{(\Delta x)^2}\right] + \left[\frac{u_{i,j+1} + u_{i,j-1} - 2u_{i,j}}{(\Delta y)^2}\right] = 0$$

- If the mesh is square (Δx=Δy)

$$u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j} = 0$$

- This equation holds at every point (i, j) except at points we define to be a particular temperature, these are known as the boundary points, the linear equation defining these points looks like

$$u_{i,j} = c_{i,j}$$

Where $c_{i,j}$ is the defined temperature at the point (i, j).

Solving Linear Equations

- There are multiple ways to solve this set of linear equations. Typically in linear algebra classes we talk about gaussian elimination or LU factorization but in this case that would be inefficient.

- Let us suppose we have a 2D, 100 x 100 mesh, that is 10,000 linear equations. As each equation has 10,000 coefficients, but only at most 5 non-zero coefficients, that means only 0.05% of the matrix is non-zero. A traditional dense method (like Gaussian Elimination or LU factorization) would result in a huge number of multiplies and adds involving 0 – this is inefficient.

- We will be using an iterative method adapted for use with sparse matrices (matrices with lots of 0's), it will be significantly faster as this method will not involve the large number of multiplications by 0 or additions of 0 to another number. We will be using the Jacobi method

$$x_i^{(k+1)} = \frac{1}{a_{i,i}}\left(b_i - \sum_{j \neq i} a_{i,j} x_j^{(k)}\right)$$

- Where $a_{i,j}$ are the individual elements of the coefficient matrix, b_i are the elements of the right hand side of the linear equations (either a 0 or a specific temperature), $x_j^{(k)}$ are the elements of the kth trial solution -- we seed the method with a guess solution $\{x_j^{(0)}\}$ and repeat until $\{x_j^{(k+1)}\} \approx \{x_j^{(k)}\}$.

Populating the Matrix

- We'll be using a 750x750 (numbered 0 to 749) mesh where the edges are being held at 0 degrees, and the central point (375,375) is set to 100 degrees.
- The diagonal and the off diagonal elements are not used in the same way by the Jacobi iterative method, therefore we will store them separately. $A = D + C$, where D is the diagonal, and C stores the off diagonals.
- D is a dense vector, as such will be stored traditionally diag[i]=$D_{i,i}$. There are multiple method of storing the sparse off diagonals. We'll be storing only the non-zero elements along with their locations in the matrix in the arrays Arow, Acol, and Avalue in the compressed sparse row format.

```
index=-1; // Where in the off diagonal array we are placing the value
  for (int i=0;i<NMesh*NMesh;i++) {
       Arow[i]=index+1;
       x[i]=0;
       if (i<NMesh) {
            diag[i]=1;
            b[i]=0;}
       else if ((NMesh*NMesh-i)<NMesh) {
            diag[i]=1;
            b[i]=0;}
       else if ((i%NMesh)==0) {
            diag[i]=1;
            b[i]=0;}
       else if ((i%NMesh)==(NMesh-1)) {
            diag[i]=1;
            b[i]=0;}
       else if (i==(NMesh*NMesh/2+NMesh/2)) {
            diag[i]=1;
            b[i]=100;
            x[i]=100;}
       else {
            diag[i]=-4;
            index=index+1;
            b[i]=0;
            Acol[index]=i-NMesh;
            Avalue[index]=1;
            index=index+1;
            Acol[index]=i-1;
            Avalue[index]=1;
            index=index+1;
            Acol[index]=i+1;
            Avalue[index]=1;
            index=index+1;
            Acol[index]=i+NMesh;
            Avalue[index]=1;} }
  Nnz=index+1;
```

Solving the Matrix

- As previously discussed, we'll be using Jacobi's iterative method to solve the system
- As we've stored the diagonal separate, we will not need to test for the diagonal when calculating the sum on the previous slide.
- The process starts with a guess, which is then improved upon by the Jacobi method.
- The differences between this and the previous iteration are summed in the last loop.
- The process is repeated until the summed difference (conv) is beneath a certain threshold.

```
conv=100;
   // this is the maximum temperature, thus by definition not
   // converged
iter=0;
int k;
int j;
while ((conv>convcriteria)&&(iter<MaxIter)) {
    iter=iter+1;
    for (int i=0;i<NMesh*NMesh;i++) {
      interim[i]=0.0;}
      for (int i=0;i<NMesh*NMesh;i++) {
         for (j=Arow[i];j<Arow[i+1];j++){
           k=Acol[j];
           interim[i]=interim[i]+Avalue[j]*x[k];}}
      conv=0.0;
      for (int i=0;i<NMesh*NMesh;i++) {
           temp=(b[i]-interim[i])/diag[i];
           conv=conv+fabs(x[i]-temp);
           x[i]=temp;}}
```

How to Port to GPU using OpenACC?

Solving the Matrix

- Over 99% of the run time happens in this section of code.
- The data dependency and the use of Acol to determine the index of x[] in the 3rd loop (the nested loop) proved too complex for the PGI (version 19.10) compiler if simple pragmas were used, returning messages such as, “PGCC-S-0155-Compiler failed to translate accelerator region (see -Minfo messages): Could not find allocated-variable index for symbol – Acol”
- The acc data region, and the marking of the variables using present() gives the key information the compiler needs.

```
conv=100;
    // this is the maximum temperature, thus by definition not
    // converged
iter=0;
int k;
int j;
#pragma acc data copyin(Arow[NMesh*NMesh+1],
      Acol[NMesh*NMesh*4],Avalue[NMesh*NMesh*4])
      copy(x[NMesh*NMesh]) create(interim[NMesh*NMesh])
{
while ((conv>convcriteria)&&(iter<MaxIter)) {
     iter=iter+1;
#pragma acc parallel loop
     for (int i=0;i<NMesh*NMesh;i++) {
        interim[i]=0.0;}
#pragma acc parallel loop private(k,j)
     present(Arow[NMesh*NMesh+1],Acol[NMesh*NMesh*4],
     Avalue[NMesh*NMesh*4])
     for (int i=0;i<NMesh*NMesh;i++) {
          for (j=Arow[i];j<Arow[i+1];j++){
               k=Acol[j];
               interim[i]=interim[i]+Avalue[j]*x[k];}}
      conv=0.0;
#pragma acc parallel loop private(temp) reduction(+:conv)
      for (int i=0;i<NMesh*NMesh;i++) {
           temp=(b[i]-interim[i])/diag[i];
           conv=conv+fabs(x[i]-temp);
           x[i]=temp;}}
}
```

Timing

Measurement	Serial Runtime (ms)	OpenACC Runtime (ms)
Time to Solve	1.788×10^8	4.134×10^7
Total Run time	1.788×10^8	4.135×10^7

- The speed up is not as impressive as for other codes. It is however significant that the speedup takes place with just a few lines.

Conclusions

- While this problem seems simpler than many, it actually proved very difficult for the compiler to do on its own with simple acc kernels or acc parallel loop statements.
- With a more detailed specification the compiler was able to accelerate the code.
- While the speed up (~ a factor of 4) was not as impressive as can be achieved with some other codes, it is non trival.

Additional Suggested Exercises

- Investigate how the problem scales with size:
  - if the number of bodies increases does the speedup increase or decrease? How does the code scale for a problem 2 times larger?
- If you know OpenMP, parallelize the code with OpenMP and get timing data – compare the performance as a function of wattage (you'll need to look this up for the hardware in your machine.