David Degirmenci
u1624423

<u>Original code run time vs final code (mean over 10 runs on stan1):</u>

Original: 47.14s Final: 1.65s

<u>Arithmetic and repeated calculation:</u>

"/pow(dx,2.0)" is expensive as it involves division and calling pow, set a variable dx2r = 1/(dx*dx)

Similarly, "/pow(dy,2.0)" -> dy2r = 1/(dy*dy), "/dx/2.0" -> dxr = 0.5/dx, "/dy/2.0" -> dyr = 0.5/dx

"dt*nu" -> dtnu = dt*nu, this is a repeated multiplication, saves little but nonzero amount of time

"Lapl" and "grad" don't need to be set to 0.0 every time they are used, in the ix pass, they can just be set equal to their calculation rather than added.

These constants are defined outside of the istep loop to make sure they aren't repeatedly redefined.

<u>Factorising:</u>

Factorising reduces the amount of multiplication and other arithmetic.

The current general form of Lapl, grad and u_new are (i,j depend on the condition, d2r and dr are dx2r/dy2r and dxr/dyr):

Lapl: -2.0*u[i1][j1]*d2r + u[i2][j2]*d2r + u[i3][j3] *d2r

grad: (u[i4][j4] - u[i5][j5])*dr

u_new: u[ix][iy] - dt*u[ix][iy]*grad + dtnu*Lapl

They can be factorised to:

Lapl: (-2.0*u[i1][j1] + u[i2][j2] + u[i3][j3]) *d2r

grad: (u[i4][j4] - u[i5][j5])*dr (same)

u_new : u[0][i]*(1-dt*grad) + dtnu*Lapl

This is quite significant for u_new (especially since it runs many times), factorising reduces the number of times the u array is accessed.

<u>Branches and passing through data:</u>

There are many if, else if statements within a nested loop – very expensive.

The only condition that needs to be within the nested loop is when both ix,iy!=0 or Nx-1,Ny-1.

All other conditions can be taken out of the inner loop, and some can be taken out of the outer loop.

Outside the outer loop are the following conditions:

- ix,iy==0 (bottom left)
- ix==0,iy==Ny-1
- iy==0,ix==Nx-1
- ix==Nx-1,iy==Ny-1

These 4 conditions are exact.

Within the outer loop remain the following conditions:

- ix==0 boundary (away from iy==Ny-1,0 boundaries)
- iy==0 boundary (away from ix==Nx-1,0 boundaries)
- ix==Nx-1 boundary (away from iy==Ny-1,0 boundaries)
- iy==Ny-1 boundary (away from ix==Nx-1,0 boundaries)

This is a large time save.

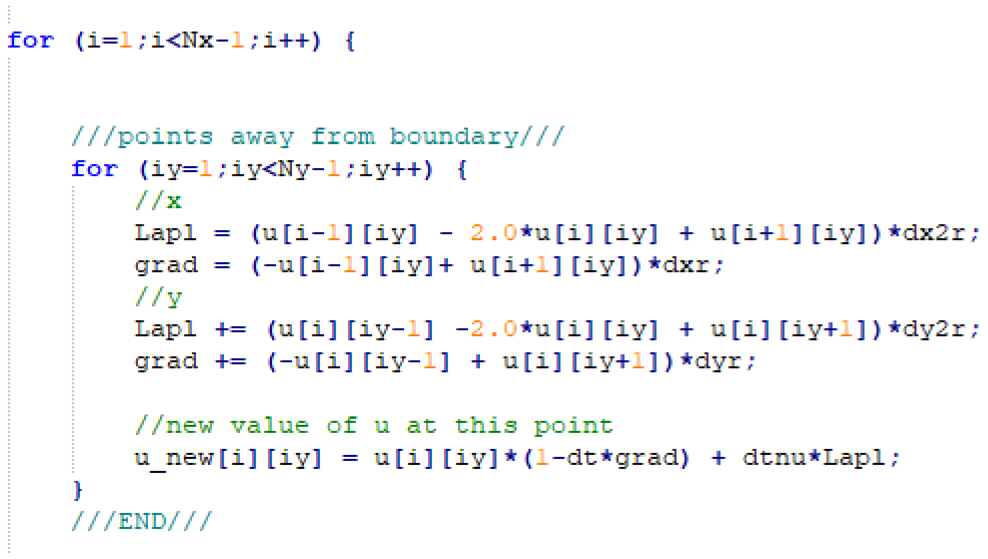
Optimising memory access and patterns for efficient use of cache:

In C, 2d arrays are stored in memory with row major ordering, so ordering accesses of arrays should be first in order of the first index then the second index.

This has been done for almost every array access. This is limited by multiplication and factorisation in general, but it is more expensive to have extra multiplication.

Further Optimisations:

Currently, each condition's code looks something like this:

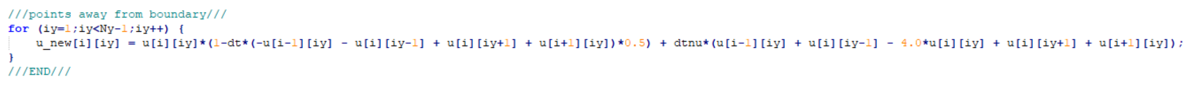


```c
for (i=1;i<Nx-1;i++) {

    ///points away from boundary///
    for (iy=1;iy<Ny-1;iy++) {
        //x
        Lapl = (u[i-1][iy] - 2.0*u[i][iy] + u[i+1][iy])*dx2r;
        grad = (-u[i-1][iy]+ u[i+1][iy])*dxr;
        //y
        Lapl += (u[i][iy-1] -2.0*u[i][iy] + u[i][iy+1])*dy2r;
        grad += (-u[i][iy-1] + u[i][iy+1])*dyr;

        //new value of u at this point
        u_new[i][iy] = u[i][iy]*(1-dt*grad) + dtnu*Lapl;
    }
    ///END///
```

(this condition is within a nested loop, the others are not within the inner loop, and some are not even within the outer loop, although all are within the istep loop)

Both dx,dy=1.0, so dx2r=dy2r=1.0, and dxr=dyr=0.5. This means that these variables don't need to be used in the arithmetic. In fact

By setting Lapl, grad equal to the sum rather than adding on afterwards, we lose readability but we can reorder the array accesses for more efficient cache use. We can then place these into grad and Lapl in u_new, so Lapl and grad don't have to be used, shown here:



```c
///points away from boundary///
for (iy=1;iy<Ny-1;iy++) {
    u_new[i][iy] = u[i][iy]*(1-dt*(-u[i-1][iy] - u[i][iy-1] + u[i][iy+1] + u[i+1][iy])*0.5) + dtnu*(u[i-1][iy] + u[i][iy-1] - 4.0*u[i][iy] + u[i][iy+1] + u[i+1][iy]);
}
///END///
```

This entire optimisation is of negligible difference for each iteration, but the code above runs 10000*254*254=645,160,000 times, so it adds up to up to a few seconds without any optimisation flags. I have done this similarly with the rest of the code, which saves a tiny amount of time.

Additionally, the way that u_new was being copied into u was very inefficient, with a nested loop (running a copy 650+million times). Ideally a simple memcpy(u, u_new, Nx*Ny*sizeof(double)) could be used, but the way memory is allocated to these arrays means that is not possible. Instead slices of the array are iterated through and memcopied from u_new to u.

The snapshots of grid to file involves a branch, which runs 10000 times, but makes a negligible difference to the time, so this can be removed or kept (obviously kept if user wants to see and for there to be no compiler warnings).

Setting the optimisation compiler flags -O1, -O2, -O3 reduces the time of the final code from on average 9.93 seconds to (average running on stan1):

- -O1: 3.11s
- -O2: 2.42s
- -O3: 1.65s

These all produce the same output as without any optimisation flags.

On the original code it reduces the time from 47.14s to (average running on stan1):

- -O1: 6.98s
- -O2: 5.47s
- -O3: 5.42s

(On my personal machine, with the -O3 flag, I have managed to get the time down to less than 1 second)