Homework 1

Julian Sanders
4675045

1 Exercises

In this exercise, we will exploit **shared memory parallelism** to solve a standard partial differential equation, the 3D Poisson equation:

$$-\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}+\frac{\partial^2 u}{\partial z^2}\right)=f(x,y,z) \tag{1}$$

on a unit cube $\Omega=[0\ldots 1]\times[0\ldots 1]\times[0\ldots 1]$, subject to Dirichlet boundary conditions.

- u(x,y,z)=0 if x=0, x=1, y=0, y=1, or z=1,
- u(x,y,z)=g(x,y) if z=0.

The PDE is discretized using second order finite diferences, and solved via the Conjugate Gradient method.

1.1 Exercise 1

Assignment:

Implement the missing functions `init`, `axpby`, `dot`, and `apply_stencil3d` using standard sequential for-loops. The specification of these functions can be found in the file `operations.hpp`. Implement suitable unit tests to verify they work as expected. Remember that a good unit test does not re-implement the operation, but verifies the correct behavior by checking mathematical relations for simple, well-understood cases. For instance, the Laplace operator implemented in `matvec` computes the second derivative of a 1D function in each direction, and should be second order accurate. Some simple examples of a unit test is given in `test_operations.cpp`, where you can add your own as well. To compile and run the unit tests, use `make test`. This may be done on the login node when developing on DelftBlue.

Note: you can implement the matrix-vector product with a 7-point stencil in any way you like (matrix-free, by storing diagonals, etc.). Start with a simple implementation and create sufficient unit tests so that you can then refine it. The tricky part here are the boundary conditions.

Implementation:

I implemented `init`, `axpby`, `dot` using standard for loop code.

`apply_stencil3d` was implemented such that at the boundaries, whichever values of the stenci that fall outside the domain will not get added to the value of the resulting array. This is in accordance with how it should be for homogenous dirichlet boundary conditions, as well as nonhomogenous dirichlet boundary conditions. In the second case the boundary condition will not influence the matrix A, but only the vector $\mathbf{b}$ in the resulting equation $A\mathbf{x}=\mathbf{b}$

I did not come up with more useful tests for these methods that the ones already defined.

1.2 Exercise 2

Assignment:

Complete the Conjugate Gradient method in `cg_solver.cpp` by calling the previously implemented functions in the locations indicated by an ellipse (`[...]`). Compile and run the driver application by typing `make`. This executable creates a 3D Poisson problem and solves it using the CG method. Add tests to a new file `test_cg_solver.cpp` and modify the `makefile` so that it is compiled into the `run_tests.x` executable. `n` iterations).

Implementation:

I implemented the operations in `cg_solver.cpp` using the previously defined operations. I compiled tan ran the application. the result was a list with the residual values of each of the 502 iterations of the CG method.

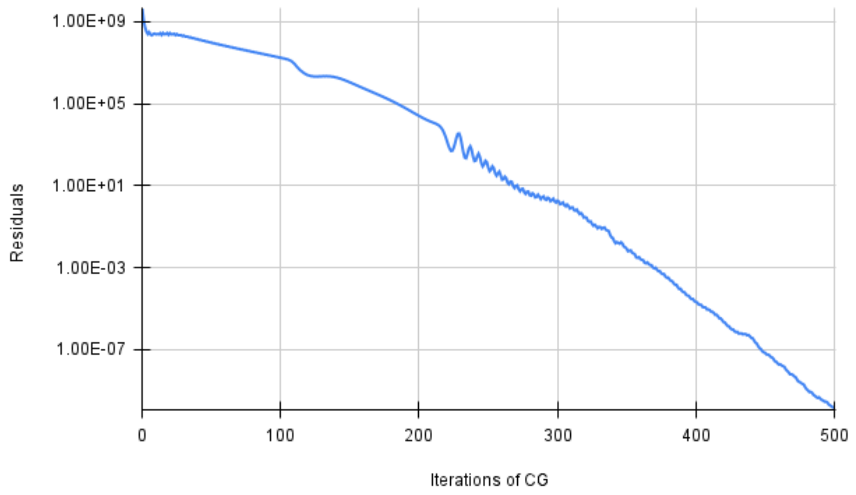

Figure 1: A logarithmic plot of the residuals of the CG method

The provided code solves Laplace's equation for:

$$f(x, y, x) = z\sin(2\pi x)\sin(\pi y) + 8z^3; \tag{2}$$

$$g(x, y) = x(1-x)y(1-y); \tag{3}$$

I added my own test, one simple one to see if on a 16*16*16 discretized grid, the solution for $f = 0$ and $g = 0$ is indeed 0. The starting vector was had -1 values for elements. That way convergence was not reached immediately at the first iteration.

Further tests that could be added are those for simple choices of f and g, for instance $f = 0$ and $g(x, y) = \sin(2\pi x)\sin(2\pi y)$, of which the exact solution can be obtained easily. I chose not to add these tests yet, considering the previous one do be a sufficient diagnostic tool.

1.3 Exercise 3

Assignment:

Parallelize your basic operations from step 1 using `OpenMP`. For the `apply_stencil3d` function, parallelize the outer loop only. Rerun your tests using make test to verify that everything still works.

Implementation:

I added the line: `#pragma omp parallel for` In front of the first for loop in the `apply_stencil3d` function. The tests still work with this addition.

Table 1: Run time of 500 iterations of the CG method for $n_x = n_y = n_z = 128$ without parallelization of the stencil

label	**calls**	**total time**	**mean time**
After cg_solver	1	0.00224	0.00224
Before cg_solver	1	161.8	161.8
Start of program	1	161.8	161.8

Table 2: Run time of 500 iterations of the CG method for $n_x = n_y = n_z = 128$ with parallelization of the stencil

label	**calls**	**total time**	**mean time**
After cg_solver	1	0.002386	0.002386
Before cg_solver	1	20.29	20.29
Start of program	1	20.31	20.31

When I compile and run the program again, I can tell it is a lot faster. The timer summary makes this more concrete; the code runs nearly 8 times faster with parallelization of just the stencil operation.

I proceeded to add the line `#pragma omp parallel for` before the other for loops called in `operations.cpp`. In `dot` the syntax is a bit different, as this for loop adds something to the variable `res` in each itteration. Therefore me must use `#pragma omp parallel for reduction(+:res)`. This made the program even more efficient, the result of which can be seen in the next exercise.

1.4 Exercise 4

Assignment:

Add a new main program `main_benchmarks.cpp` and update the `Makefile` to compile an executable `main_benchmarks.x`. You may find the Timer object useful, defined in `timer.hpp`. For the individual operations from task 1, perform weak and strong scaling experiments on a DelftBlue node (using up to 48 cores). Include plots in your report, and interpret them. To run these jobs, modify the commands in the jobscript `run_cg_poisson.slurm` to your needs, but leave the `SBATCH` lines unchanged. How many cores are optimal for the `main_cg_poisson.x` application on a 512^3 grid?

Implementation:

Due to the fact that I couldn't run large `.slurm` programs on DelftBlue due to a time limit for my account, I worked around this issue by modifying the functions in the code to have a `threadnum` variable as input. I made the following tables and plots by looping over this variable:

Table 3: A table of the time it takes to run the program when the operations in `operations.cpp` are split into threads that are executed on seperate cores. The time is given in seconds. The speedup of the program is also calculated with respect to the program run on one core. The timing was investigated for $n_x = n_y = n_z = 128$ and $n_x = n_y = n_z = 512$.

Threads	**Time** $n_x = 128$	**Speedup** $n_x = 128$	**Time** $n_x = 512$	**Speedup** $n_x = 512$
1 threads	186	1.000	14400	1.000
2 threads	93.16	1.997		
3 threads	63.61	2.924		
4 threads	47.96	3.878	4548	3.166
5 threads	37.19	5.001		
6 threads	33.54	5.546	3489	4.127
7 threads	26.84	6.930		
8 threads	27.38	6.793	2594	5.551
9 threads	21.54	8.635		
10 threads	19.82	9.384	2102	6.851
11 threads	17.63	10.550		
12 threads	17.07	10.896	1881	7.656
13 threads	15.58	11.938		
14 threads	14.92	12.466	1539	9.357
15 threads	14.6	12.740		
16 threads	12.56	14.809	1365	10.549
17 threads	12.02	15.474		
18 threads	12.3	15.122	1304	11.043
19 threads	11.92	15.604		
20 threads	11.48	16.202	1102	13.067
21 threads	11.61	16.021		
22 threads	10.57	17.597	986.2	14.602
23 threads	10.92	17.033		
24 threads	10.23	18.182	927.2	15.531
25 threads	10.55	17.630		
26 threads	9.375	19.840	862.5	16.696
27 threads	9.718	19.140		
28 threads	9.272	20.060	858.4	16.775
29 threads	9.771	19.036		
30 threads	9.057	20.537	799.6	18.009
31 threads	9.917	18.756		
32 threads	8.678	21.434	784.5	18.356
33 threads	12.44	14.952		
34 threads	12.73	14.611	857.8	16.787
35 threads	12.73	14.611		
36 threads	12.65	14.704	867.2	16.605
37 threads	12.62	14.739		
38 threads	12.61	14.750	869.6	16.559
39 threads	12.51	14.868		
40 threads	12.31	15.110	860.9	16.727
41 threads	12.29	15.134		
42 threads	12.1	15.372	847.4	16.993
43 threads	11.52	16.146		
44 threads	11.59	16.048	833.6	17.274
45 threads	11.6	16.034		
46 threads	11.71	15.884	820.4	17.552
47 threads	11.52	16.146		
48 threads	11.52	16.146	808.1	17.820

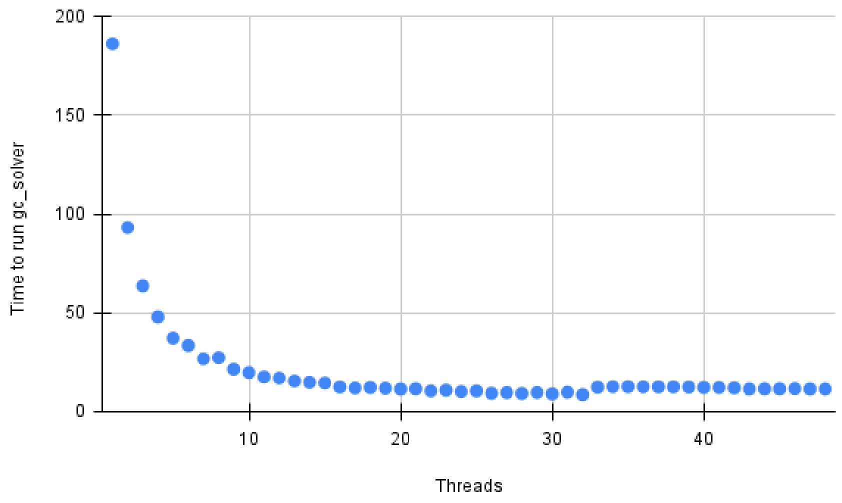

Figure 2: The run time in seconds of 500 iterations of `cg_solver` for $n_x = n_y = n_z = 128$ with the stencil operation parallelized into different numbers of threads

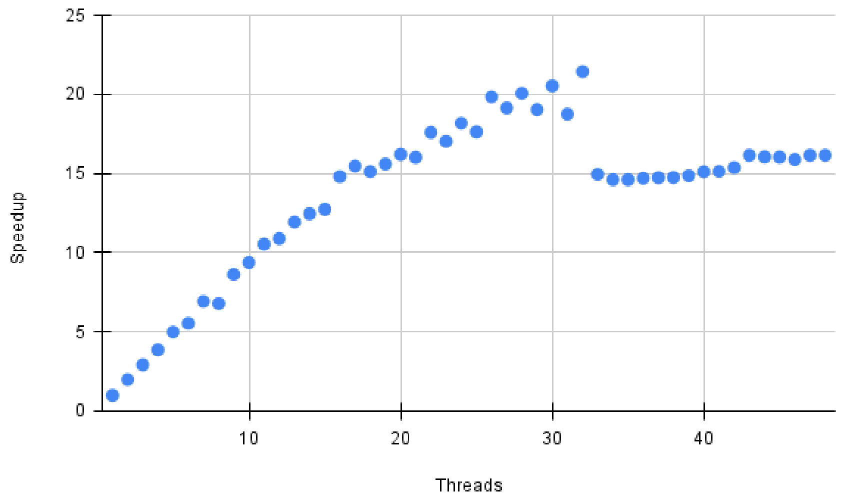

Figure 3: The speedup of `cg_solver` for various thread numbers. Here $n_x = n_y = n_z = 128$.

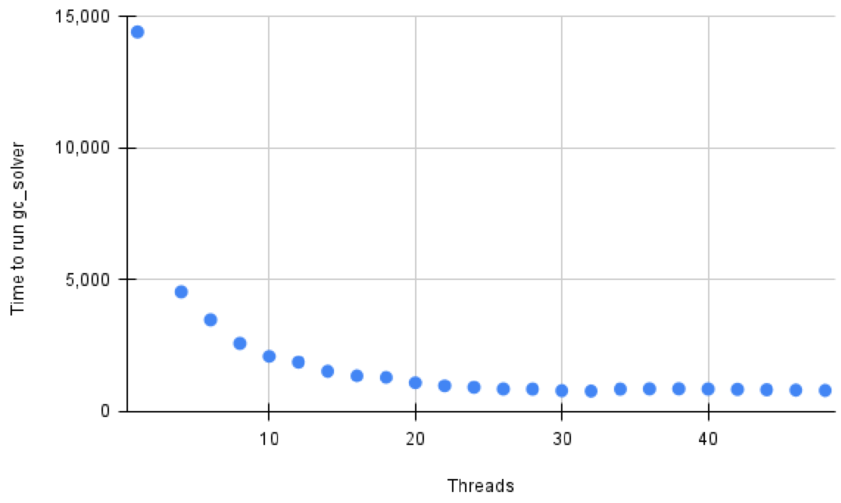

Figure 4: The run time in seconds of 500 iterations of `cg_solver` $n_x = n_y = n_z = 512$ with the stencil operation parallelized into different numbers of threads

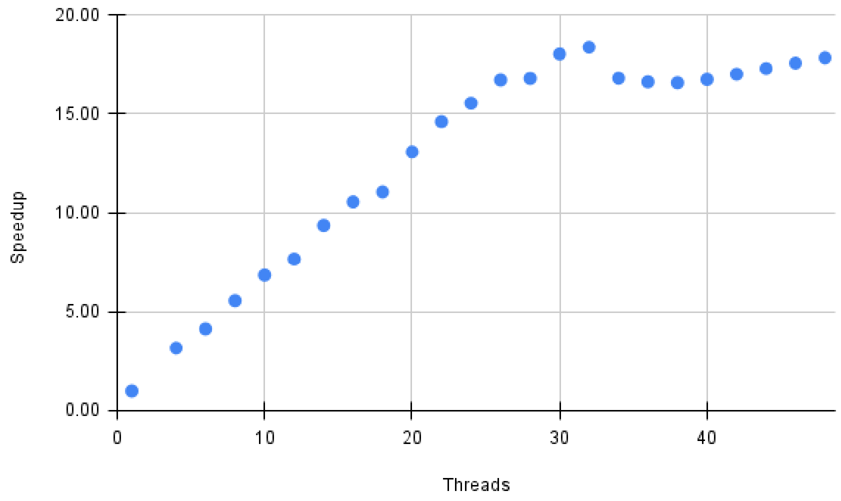

Figure 5: The speedup of `cg_solver` for various thread numbers. Here $n_x = n_y = n_z = 128$.

From the table 3 and the figures 2-5 we see that for both $n_x = n_y = n_z = 128$ and $n_x = n_y = n_z = 512$ the scaling is optimal for 32 threads. After this point, the time to execute the program actually increases by splitting the for loops into more threads.

I hypothesise that this is due to the fact that 128 and 512 are divisible by 32, and not by 33, 34, 35 etc. So a loop over 128 or 512 iterations (which is the case for the first of the three nested for loops in `apply_stencil3d`) can be evenly split into 32 tasks of the same amount of work (4 iterations for $n_x = 128$, 16 iterations for $n_x = 512$, but dividing it into 33 tasks will not add much more efficiency. Most of the threads of `apply_stencil3d` will still run 4 times the iteration inside the for loop (or 16 times for $n_x = 512$). However, now more time goes into splitting the work into threads and combining the results.

A possible solution to this is the `collapse(3` addition to the `pragma` line mentioned in exercise 6. Then the parallelization of the stencil will split all 128^3 iterations of the nested for loop, instead of just the 128 iterations of the outer for loop.

This hypothesis can also explain why the speedup loss in the 512 case after 32 threads is not as much as in the $n_x = 128$ case: for $n_x = 512$ each thread will consist of more iterations, so the time cost of splitting into 33 threads is smaller compared to time it takes to execute all the iterations in one thread.

Regardlessly, the program seems to run optimal when parallelized over 32 cores. Thus, I chose to investigate strong scaling for 32 cores and 16 cores. Here the maximum number of iterations was chosen to be 10000, so that for all n the programs could converge until the set tolerance for the residual is met.

Table 4: The scaling relationship between the run time and $n_x = n_y = n_z$ for 32 theads and 16 threads.

n_x	**Time for 32 threads**	**Time for 16 threads**
32	0.05515	0.07046
64	0.6591	0.9797
128	11.43	15.8
256	250.6	372
512	4003	7917

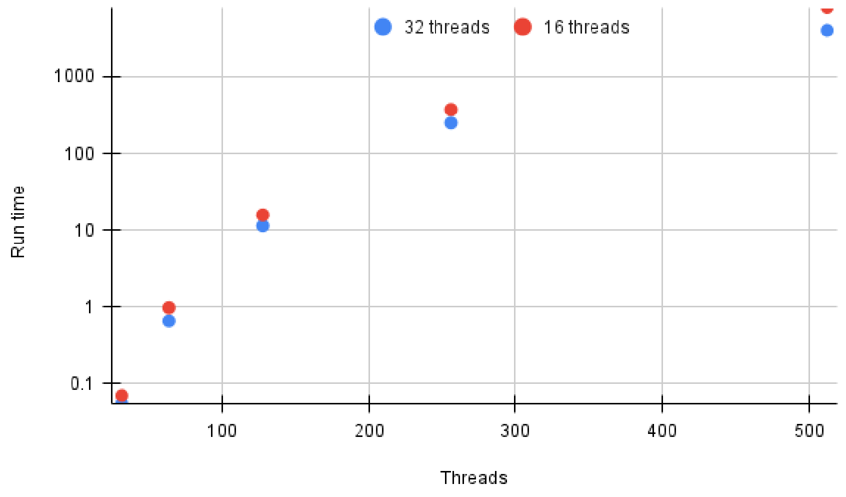

Figure 6: The run time of the cg solver for different values for $n_x = n_y = n_z$ ranging from 32 to 512. The run time axis is given in seconds, and scaled logarithmically.

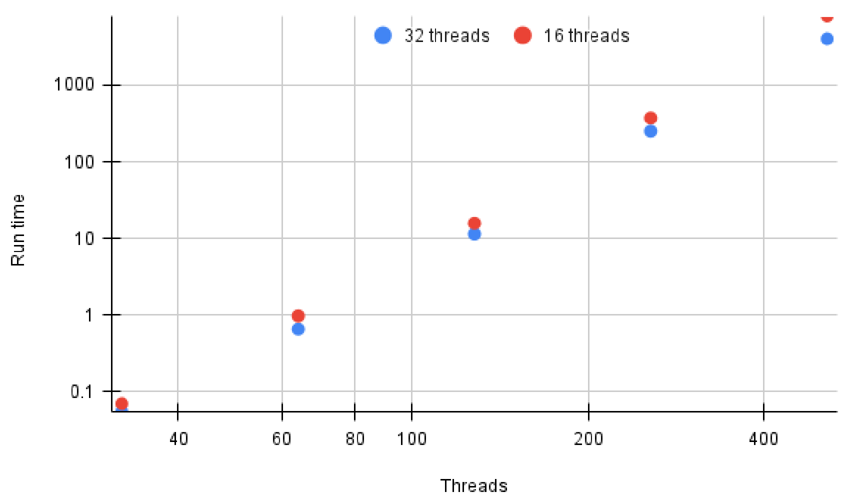

Figure 7: The run time of the cg solver for different values for $n_x = n_y = n_z$ ranging from 32 to 512. The run time axis is given in seconds, and scaled logarithmically.

Note that from the linearity of the points in this log log plot 7 it can be inferred that the scaling factor p for $T = \mathcal{O}(n^p)$ is 4.04 for 32 threads and 4.19 for 16 threads. This makes sense, as judging by the grid size the work per iteration increases $O(n^3)$. Howerver, for larger n the CG method will need more iterations to converge, leading to the fact that the total time will scale with a larger exponent of n.

1.5 Exercise 5

Assignment

For the `apply_stencil3d` operation, try interchanging the three nested loops, and run the code sequentially (set `OMP_NUM_THREADS=1`). Which loop order is best, and why? In case you stored the matrix entries in task 1, change the order in which they are accessed in `apply_stencil3d`.

Implementation

In order to investigate which loop order was best, I adapted the code so that the `apply_stencil3d` operation is run with only one thread.

Table 5: The run time of the cg solver for 500 iterations for different nested loop orders in `apply_stencil3d`

Nested loop order	Run time [s]
x-y-z	199.1
y-z-x	82.78
z-x-y	82.61

From the table, we see that he z-x-y order seems to be best. This refers to the order where the y direction loop is the inner for loop, the x direction loop the middle loop, and the z direction the outer for loop. Its performance is not much different from the y-z-x order. The only order which takes significantly longer is the x-y-z order. I presume this is in some way related to the lexographic ordering of the vectors `x` and `q` to which the stencil is applied. It makes intuative sense that in those orders for which the inner for loop loops over memory adresses that are stored closer together is more efficient that when the inner for loop calls memory adresses that are further apart.

To investigate this effect further, I also timed the order z-y-x. This way, the for loops call the elements of the vector that the stencil is being applied on and the vector in which the result is stored lexographically. The program with this ordering took only 77.75 seconds, which seems to validate our hypothesis about the efficiency of nested for loops that follow the lexographic ordering of the vectors.

1.6 Exercise 6

Assignment

Run the `main_cg_poisson.x` driver. Do you observe a performance difference when solving a $1024 \times 128 \times 128$ or a $128 \times 128 \times 1024$ problem on 8 cores? Add the `collapse(3)` clause to the `OpenMP pragma` line in `apply_stencil3d` and run again. Explain your observations.

Implementation

The results can be seen in the table. Both for the stencil code with a simple `#pragma omp parallel for` line as the stencil code with a `#pragma omp parallel for collapse(3)` line.

Table 6: The run times in seconds of the cg solver for a different grid dimensions for a stencil function with `#pragma omp parallel for` and one with `#pragma omp parallel for collapse(3)`. All runs were done using 8 threads.

Grid dimensions	**Iterations**	**Time**	**Time/it**	**Time** `collapse(3)`	**Time/it** `collapse(3)`
$1024 \times 128 \times 128$	3382	884.9	0.2616	357.8	0.1058
$128 \times 1024 \times 128$	3205	851.8	0.2658	331.1	0.1033
$128 \times 128 \times 1024$	3361	897.1	0.2669	345.8	0.1029

From these results, we can see that the performance difference for the different grids is not significantly large. The amount of iterations until convergence differ slightly, but this is expected due to the fact that the boundary conditions and the function f are not symmetric in x y and z. Solving the problem with a 1024 resolution in the x direction is not the same problem as one with a 1024 resolution in the z direction, so the iterations needed until the CG method converges will be different.

By dividing the total run time by the number of iterations, we see that the time per iteration does not differ significantly between the different grids.

Adding the collapse(3) clause to the `pragma` line improves the run time by more than 2 times for all grids. Because the nested for loops are now recognized as one large loop the OpemMP, the work can be distributed among threads more efficiently. This leads to the observed efficiency increase.

1.7 Exercise 7

Assignment

The bulk-synchronous performance model (BSP) views a program like `poisson_cg` as a sequence of parallel operations interleaved by communication phases. It predicts the overall runtime to be the sum of the cost of computation and communication phases. Can you spot opportunities in the CG algorithm to reduce the number of stages/loops? Implement your own version of the algorithm with fewer loops, and measure if this makes the method faster. Explain your observations.

Implementation

The number of stages (loops) can be reduced by combining some operations done in the while loop of the `CG_solver` into the same operation with a single for loop.

For example, the following operations could be combined into a single operation, which will only have one for loop as opposed to two:

The two operations:

```
    // x = x + alpha * p
    axpby(n, alpha, p, 1.0, x);

    // r = r - alpha * q
    axpby(n, -alpha, q, 1.0, r);
```

can be replaced by the single operation:

```
    // x = x + alpha * p
    // r = r - alpha * q
    twice_axpby(n, alpha, p, 1.0, x, -alpha, q, 1.0, r);
```

These functions have the following definitions:

```
void axpby(int n, double a, double const* x, double b, double* y)
{
  // A for loop that computes a*x+b*y elementwise and stores it in n dimensional
    array y
  #pragma omp parallel for
  for(int i = 0; i < n; i++)
  {
    y[i] = a*x[i] + b*y[i];
  }
  return;
}

void twice_axpby(int n, double a, double const* x, double b, double* y, double c,
     double const* z, double d, double* s)
{
  // A for loop that computes a*x+b*y elementwise and stores it in n dimensional
    array y
  #pragma omp parallel for
  for(int i = 0; i < n; i++)
  {
    y[i] = a*x[i] + b*y[i];
    s[i] = c*z[i] + d*s[i];
  }
  return;
}
```

The results of changing the script to this `twice_axpby` function can be seen in the table below:

Table 7: The average and standard deviation results of running the cg solver 5 times, both for with and without the `axpby` improvement.

	mean run time [s]	**standard deviation [s]**
before improvement	7.7298	0.06526
after improvement	7.7906	0.12718

Note that the run time after improvement seems to be larger, although not significantly, judging by the standard deviations. Either way, the improvement from combining two for loops into one for the two `axpby` calls does not seem to impove the efficiency of the program significantly.

Another possible location for such an improvement are the operations:

```
    // q = op * p
    apply_stencil3d(op, p, q);

    // beta = <p,q>
    beta = dot(n, p, q);
```

This is more complicated however, as one is the for loop with a summing variable and the other is not. Because the previous improvement already fulfills the scope of this exercise, I decided not to make this further improvement at this time.