Student Name: Lieke van der Linden
Student ID: 4702697

WI4450 Special Topics in Computational Science and Engineering
Homework 1

---

1 Implementation of standard operations

The standard operations `dot`, `init`, `axpby` and `apply_stencil3d` are implemented as in `operations.cpp` (ignore the parallelization for now). For `init` and `dot`, suitable tests were written. For `axpby`, the following test is added.

- Initialize `x[i]=i+1, y[i]=n-(i+1) i=1,..,n`.
- Calculate $\mathbf{y} = a\mathbf{x} + a\mathbf{y}$ with `axpby`.
- Check whether $y_i = an \quad \forall i$.

The output of the function `apply_stencil3d` applied on the canonical basis, is checked on whether it is symmetric. Next to that, a test is added to check whether the output of the function on the canonical basis is diagonal dominant (for a diagonal dominant stencil) in the test `stencil3d_diagdom`.

2 Conjugate Gradient method

The Conjugate Gradient Method is implemented in `cg_solver.cpp`. Tests for this Conjugate Gradient method are based on the fact that the Conjugate Gradient method computes the solution to the system $A\mathbf{x} = \mathbf{b}$, when A is symmetric. Therefore, one test of this Conjugate Gradient method is as follows.

- Initialize the stencil `S` based on a discretization of the Poisson problem.
- Initialize the solution `u_known[i] = i%3` (for example, can be any function of i).
- Compute the right-hand side vector by applying the stencil to `u_known` with `apply_stencil3d`.
- Use as a starting vector `u[i] = 1.0`.
- Solve $S\mathbf{u} = \mathbf{rhs}$ with `cg_solver(&S, n, u, rhs, ..)`
- Check whether `u` and `u_known` are close enough.

Furthermore, we know that the Conjugate Gradient method converges to the exact solution in one iteration when the stencil is the identity. To check if our Conjugate Gradient method corresponds with this, the stencil in the first step in the previous test is replaced with the identity operator and the maximum number of iterations is set to 1 in the test `identity`.

When executing `main_cg_poisson` with a problem size of $n = 128$, the residual norm converges to zero as shown in Figure 1.

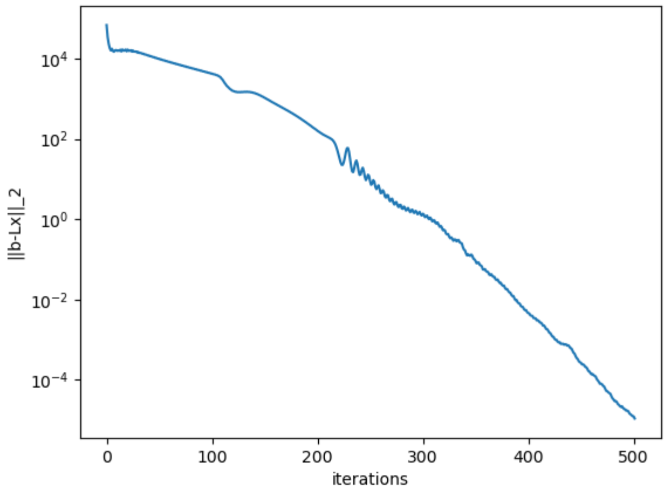

Figure 1: The convergence behaviour of the Conjugate Gradient method on the Poisson problem.

3 Parallelize the operations

To parallelize the basic operations `#pragma omp parallel for` is used. This parallelizes the for loop after the statement. In the function `dot`, each individual loop adds something to the result `dot_product`, so in the end the total sum has to be derived. This is done with adding `reduction(+: dot_product)` in the pragma. The other functions do not need this addition, because all the loops calculate a different number. The final implementation can be found in `operations.cpp`.

4 Scalability

4.1 Strong scalability

To investigate strong scalability, we ask ourselves the question "What happens if I try to exploit more parallelism to solve this problem?". We fix the problem size to $n = 128$ and run the Poisson problem with a varying number of threads. The result is shown in Figure 2. We can see that in the beginning, the speedup increases rapidly as the number of threads increases. However, for a large number of threads, the speedup can decrease, probably because of large overhead. This is in agreement with the observation that the function `dot` asks for more communication, so has relatively more overhead and will become inefficient for less number of threads.
The result are also in agreement with Ahmdahl's law, which states that there is an upper limit for the speedup $1/s$. The value s is the amount of work that cannot be parallelized. Since we do not know this exactly, we cannot derive the upper limit exactly. However we can see in Figure 2 that the speedup will not exceed one determined number for all different operations.

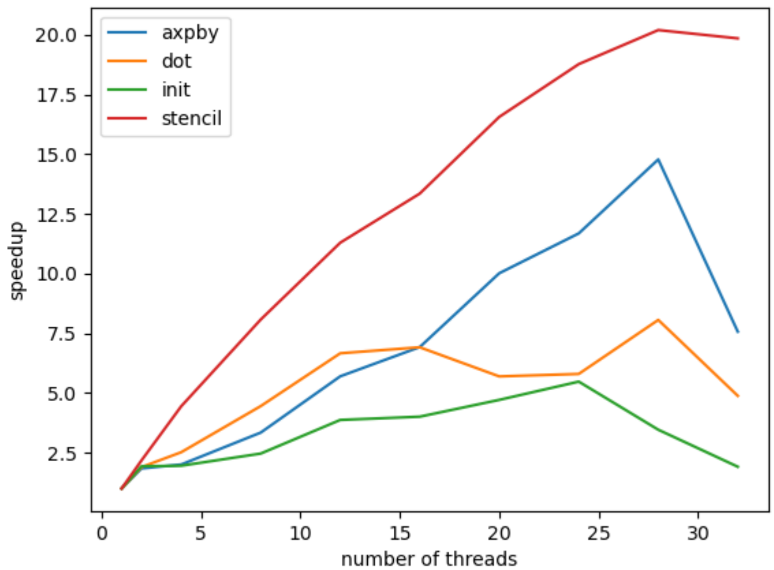

Figure 2: Speedup with strong scalability of the basic operations.

4.2 Weak scalability

For weak scalability, we ask ourselves the question "What happens if I want to solve a larger problem with more computing resources?". In this case, we will use varying problem sizes and a varying number of threads. When the problem size increases with a factor c, the number of threads has to increase by a factor of c^3, because in `main_benchmarks` we use that $n = nx \cdot ny \cdot nz = nx^3$ where nx is the value used as input.

In Figure 3, the weak scalability is shown. The number of threads and problem size used, are listed in Table 1.

problem size	256	323	406	512
# threads	6	12	24	48

Table 1: Values used for weak scalability with increasing factor of $2^{1/3}$.

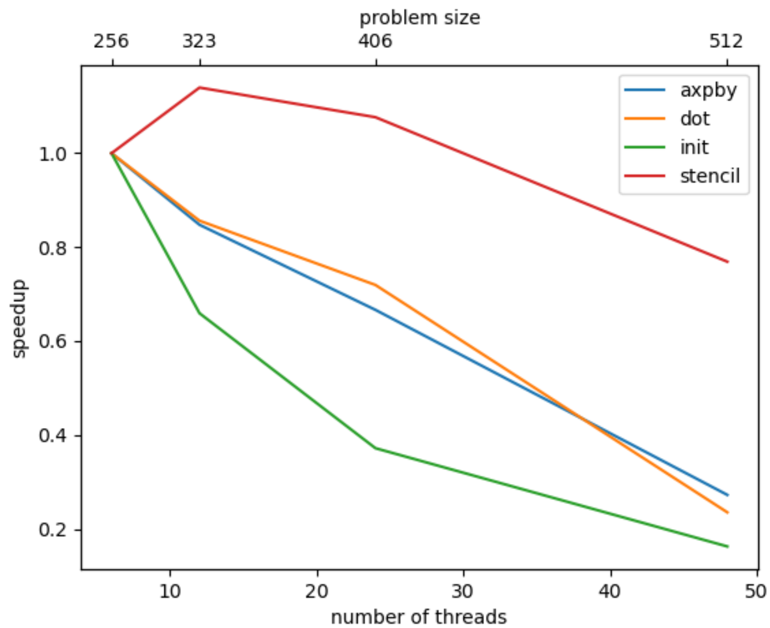

Figure 3: Speedup with weak scalability of the basic operations.

The running time of the Conjugate Gradient method will probably be dominated by the `apply_stencil3d` function. Therefore to predict what would be the best number of threads for a 512^3 grid, we look at the speedup of the stencil operation in Figure 2. This is best for 28 threads. The speedup for a higher problem size when using a varying number of threads will not differ much as shown in Figure 3. Therefore we expect that 28 threads are optimal for a 512^3 grid.

5 Interchanging nested loops

We want to investigate whether interchanging the loops in `apply_stencil3d` gives better performance. For this, we interchange the loops and run `main_cg_poisson.x` with the problem size $nx = ny = nz = 64$. The results are shown in Table 2. When looking at the running times, the loop with order kji seems the best.

The reason for this is based on the way the bytes are accessed. The array **v** is structured in such a way that when you increase index k with 1, the index of **v** increases with $ny \cdot nx$, since `index_c(i, j, k) = (k*ny +j)*nx + i`. In the same way, if you increase index j with 1, the index of **v** increases with nx. Therefore, you want the index k to be fixed as much as possible while looping over all values and therefore place it outside the other loops. Inside the k loop, you want the index j to be fixed as much as possible, so this loop is placed around the loop over all i's.

order	running time (s)
ijk	0.305
ikj	0.207
jik	0.096
jki	0.198
kij	0.100
kji	0.091

Table 2: The running time of the Conjugate Gradient method with problem size $128 \times 128 \times 128$. The shown times are the average of 50 runs.

6 Performance difference for different grids

For two grids $G_1 = 128 \times 128 \times 1024$ and $G_2 = 1024 \times 128 \times 128$, we executed the program `main_cg_poisson` with 8 cores. In Table 3, we can see that both have the same running time, but G_2 has a somewhat smaller residual norm. This is probably due to rounding errors in the function `apply_stencil3d` in the loop. In the first grid partitioning, the outer loop (which is parallelized) is the biggest and therefore there can be made more communication errors. In the second grid partitioning, the outer loop is smaller, so there is less communication between the different parallel tasks.

	running time (s)	$\|\|residual\|\|_2^2$
Grid 1	54.36	6.499e+08
Grid 2	54.47	5.504e+08

Table 3: Running time for different grid layouts.

When adding `collapse(3)` to the function `apply_stencil3d`, we expect that both grids have the same performance, since with this command, the 3 for loops are distributed over the threads and not only the first loop. However, since probably the numbers 1024 and 128 can be divided by 8, the outer loops are first distributed over the tasks and after that the inner loops. Therefore we get the same performance as without `collapse(3)`, as shown in Table 4.

	running time (s)	$\|\|residual\|\|_2^2$
Grid 1	53.98	6.499e+08
Grid 2	54.52	5.504e+08

Table 4: Running time for different grid layouts with `collapse(3)`.

7 Bulk-synchronous performance model

This is skipped due to time. A try for implementing the pipelined Conjugate Gradient method as proposed [1], can be found in `pcg_solver.cpp`.

References

[1] Jeffrey Cornelis, Siegfried Cools, and Wim Vanroose. "The Communication-Hiding Conjugate Gradient Method with Deep Pipelines". In: (Jan. 2018).