Final Project

3D Poisson equation with non-stationary iterative method

HBM514E - Parallel Numerical Algorithms and Tools

## 3D Poisson equation

The Poisson equation in 3D Cartesian space can be defined

|  |  |
| --- | --- |
|  | (1) |

where φ and f are functions. The particular case for is also known as the Laplace’s equation. We can expend this formula to the following

|  |  |
| --- | --- |
|  | (2) |

Then, assuming an equal step for all three dimensions, that is , we can write

|  |  |
| --- | --- |
|  | (3) |
|  | (4) |

Since we typically know the function f, while searching for φ, we can use the equation 4 as an equation on the form Ax = b, where x would be the value of φ, A is a symmetric sparse matrix with a diagonal of 1, and b a vector built using our f function. In addition, A is also a positive-definite matrix thanks to its borderline elements.

For this project, we built the b vector as , or in other words .

## 3D topology

Because we are working on the 3D cartesian space, the values we are looking for have three components. The easiest way to deal with them is therefore to decompose our research in a 3D topology as well. We assign each process with a range of coordinate for each component, that is X, Y and Z.

Due to the second order differential discretization of the Poisson’s equation, we need for each point the values of the six closest neighboring points. Because we are dividing all three dimensions among our processes, that also means we will need to communicate with the six neighboring processes to retrieve these values. Hopefully, we only need one layer of values, so we can limit the number of values to share.

## Conjugate Gradient method

In order to solve this 3D Poisson’s equation, we are using a non-stationary iterative method, the Conjugate Gradient method. This method iteratively gives a result closer to the real solution than the solution at the current step. The main stages of this method consist first to determine the gradient, that is the difference between the current solution (Ax) and the expected solution (b), then we determine the direction for the new solution, as well as its distance, that is the coefficient to applied to the direction, and lastly, we update the solution. The iterations are stopped whenever the gradient is small enough, or a set number of iterations have already been performed.

The operations of the conjugate gradients include three different manipulations of vectors: inner product, vector addition and matrix-vector multiplication. The inner product is the sum of the product of all members of same index, which means we need to perform a reduction on the whole vector. The vector addition generates a new vector whose values are the sum of each vectors’ values of same index, so it can be calculated locally with two vectors of same size. The matrix-vector multiplication also generates a new vector, where each of its values can be associated to an inner product. In other words, both the inner product and matrix-vector multiplication required synchronization between all processes.

Synchronizing the inner product is easily done by summing every local inner product. However, the matrix-vector multiplication theoretically requires the whole vector on each process to produce a local result. Hopefully, our matrix is sparse, which means we can reduce the number of elements shared. Moreover, we only require seven values for each of the vector values, which are also the direct neighbors of said values. For that reason, we can reduce the exchanged data to a singular layer, as mentioned earlier. This means we can reduce our communications on each step to eight, since we have two unique inner product and one matrix-vector multiplication on three dimensions.

## Results

We tried our algorithm on a basic case of 500x500x500 points, that is 125’000’000 points, with from 1 up to 128 processes. We set our algorithm to stop when reaching a gradient whose inner product is lower than 10-7, which is achieve in less than 30 iterations in this case. The time for the calculation alone, without the topology initialization and final values gathering, is as follow.

When comparing the speed-up and efficiency of the current algorithm relatively to the number of processes, we obtain the following results.

As we can see, the speed-up is linear relatively to the number of cores. This means that increasing the number of processes, up to 128, will decrease the time required proportionally to this number. As for the efficiency, we can surprisingly see that the current algorithm is more efficient with 2 or 4 processes than when run on a single process. We can also see that the efficiency does not much decrease, even with a high number of processes.

Considering the over-efficiency, it is most likely due to caches misses. For the 500x500x500 example case, a single process will handle 125’000’000 floats (500Mb), while two processes will handle half of it, 62’500’000 floats (250Mb). Considering I have a L3 cache of 32Mb, this means a single process will have L3 cache misses, or main memory access, while two processes will only have 8. Most of our vector operations manipulate two vectors at once, of the previously mentioned size, resulting to twice the amount of cache misses. This means we will have around 16 more access to the main memory, which takes around 50ns each, for a single iteration on a single vector operation, when running with one process, compared to two.

Even with a higher number of cores, said 128, we have an 8x4x4 topology, which means handling floats (4Mb). Because the L2 cache is limited to 256Kb, we will still have several L2 cache miss. Because of that, it is very likely that the time lost due to the communication is partially compensate by the time gained by avoiding the cache misses.

When trying with less data, that is 1’000’000 floats (4Mb), the efficiency between 16 and 32 cores falls from 0.95 to 0.35. This is most likely due to the L2 cache miss, as mentioned earlier. With 32 cores, we will handle around 128Kb of data for a single vector, which means we can fit two of them on the L2 cache. As mentioned earlier, there is no operation that manipulate more than two vectors at once. This means it is likely that there is no L2 cache miss with 32 cores, contrarily to 16 cores.

All of that said, I believe our program is quite effective and adaptable for a large number of inputs, as it is still able to handle 500Mb of data in about 5 minutes with a single core. Handling more data will likely pose more problem on memory rather than speed.