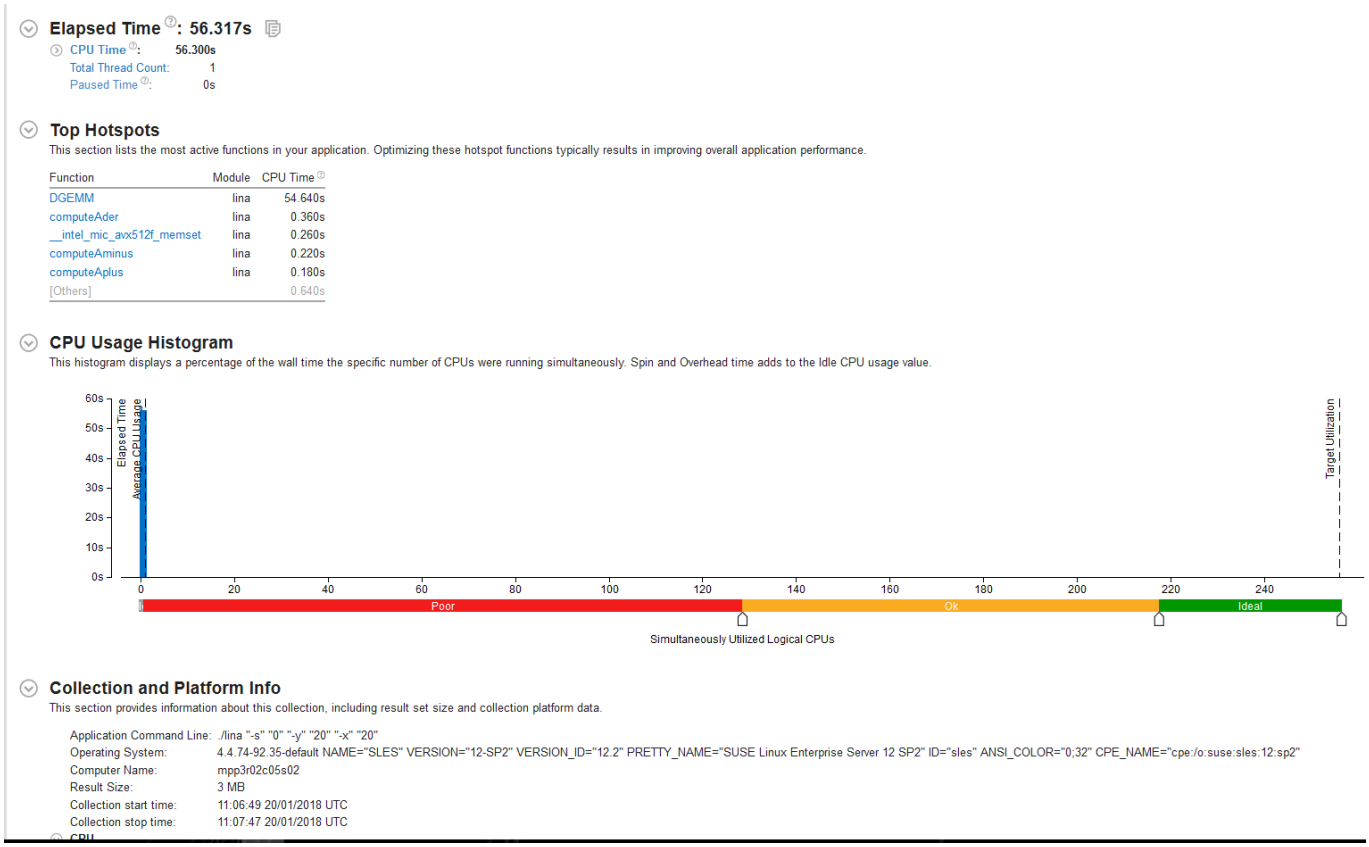
Optimization Approach

Original Approach

At first we looked at some profiling results of gprof, LIKWID and the Vtune Amplifier. The results didn't seem very useful because the original code of course is just single threaded. It became clear that almost all the computation time is spend in different calls of GEMM. GEMM is also called from different parts of the code. From the Kernel.cpp called by Simulator.cpp, the WaveFieldWriter.cpp and the Model.cpp.



Elapsed Time: 56.317s

CPU Time: 56.300s
Total Thread Count: 1
Paused Time: 0s

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time
DGEMM	lina	54.640s
computeAder	lina	0.360s
__intel_mic_avx512f_memset	lina	0.260s
computeAminus	lina	0.220s
computeAplus	lina	0.180s
[Others]		0.640s

CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU usage value.

Collection and Platform Info

This section provides information about this collection, including result set size and collection platform data.

Application Command Line: ./lina "-s" "0" "-y" "20" "-x" "20"
Operating System: 4.4.74-92.35-default NAME="SLES" VERSION="12-SP2" VERSION_ID="12.2" PRETTY_NAME="SUSE Linux Enterprise Server 12 SP2" ID="sles" ANSI_COLOR="0;32" CPE_NAME="cpe:/o:suse:sles:12:sp2"
Computer Name: mpp3r02c05s02
Result Size: 3 MB
Collection start time: 11:06:49 20/01/2018 UTC
Collection stop time: 11:07:47 20/01/2018 UTC

GEMM also appeared to be one of the few functions that actually do calculations instead of calling other functions. We decided that the GEMM.cpp is a very important file. Therefore we decided that we would start with optimizing the single core execution time of GEMM similarly to what we did in exercise 1. That means using 512 bit AVX512 intrinsic functions.
We also considered eliminating divisions in function calls by saving often used values as constants.

For example:

```
DGEMM(  NUMBER_OF_BASIS_FUNCTIONS, NUMBER_OF_QUANTITIES, NUMBER_OF_QUANTITIES,
        -1.0 / globals.hx, tmp, NUMBER_OF_BASIS_FUNCTIONS,
        A, NUMBER_OF_QUANTITIES,
        1.0, derivatives[der], NUMBER_OF_BASIS_FUNCTIONS );
```

The `-1.0 / globals.hx` does not during execution, so we saved it as a constant. We later aborted this idea because we didn't see any improvements in execution time. The code for it can still be

found in the `single_core_opti` branch.

We decided that we would do MPI next. We were planning on doing domain decomposition with a ghost layer as suggested in the presentation in order to distribute the grid among multiple MPI tasks. The original plan was to divide the grid in rows but during the implementing process we switched to 'rectangles' for the sub-grids.

We decided to add OpenMPI last, because no big code changes are need to make it work and because we wanted to make sure everything was working correctly before we add it.

For the parallel wavefield output we considered doing it using HDF5.

Single-core optimization

The original approach of kind of reusing the implementation from assignment1 does not really work, as it requires the matrix sizes to be a multiple of 8 in each direction. This is due to the AVX512 isntructions loading and writing 8 values at once.
Because the matrices are quite small and N being always fixed to 3 alot of padding would be necessary in order to pack A, B and C into larger matrices.
This requires the creation of new buffers, filling than with zeros and copying the values from A, B and C every time the DGEMM function is executed.
This ended up being slower than our final approach.

In assignment2 we used LIBXSMM to generate optimized DGEMM microkernels. These kernels also run faster than our original implementation from assignment1.
Because the size of the matrices only depends on the order we were able to generate optimized kernels for almost every call of DGEMM. The only DGEMM kernels we weren't able to generate were 3 calls in `Kernels.cpp` using an alpha value different from 1 or -1.

```
  DGEMM(  NUMBER_OF_BASIS_FUNCTIONS, NUMBER_OF_QUANTITIES, NUMBER_OF_QUANTITIES,
          1.0 / globals.hy, tmp, NUMBER_OF_BASIS_FUNCTIONS,
          B, NUMBER_OF_QUANTITIES,
          1.0, degreesOfFreedom, NUMBER_OF_BASIS_FUNCTIONS );
```

For these calls we used our own implentation with padding. The generated DGEMM kernels are located in `GeneratedGemm.h`.

They are called using if-clauses depending on the DGEMM parameters:

```
if (M_ == 3 && N_ == 3 && K_ == 3 && alpha == 1 && beta == 0 && lda == 3 && ldb ==
3 && ldc == 3) {
        DGEMMm3n3k3a1b0(A,B,C);
    }
    else if (M_ == 6 && N_ == 3 && K_ == 6 && alpha == 1 && beta == 0 && lda == 6
&& ldb == 6 && ldc == 6) {
        DGEMMm6n3k6a1b0(A,B,C);
    }
```

```
    else if (M_ == 10 && N_ == 3 && K_ == 10 && alpha == 1 && beta == 0 && lda ==
10 && ldb == 10 && ldc == 10) {
        DGEMMm10n3k10a1b0(A,B,C);
    }
    .
    .
    .
```

This means that all the DGEMM calls are executing their calculations using some kind of AVX512 instructions.

MPI

The MPI optimization consists in splitting the grid data computation equally among the MPI processes. Initially, all processes form their own copy of `materialGrid` and `degreesOfFreedomGrid`.

The user is able to configure how many columns and rows from the grids each process has to compute by passing the `-a` and `-b` command line parameters. The sub-grids have to divide the whole grid perfectly, therefore `-x` and `-y` should be divided by `-a` and `-b` respectively.

Example run command:

```
mpirun -np 2 build/lina -s 0 -x 10 -y 10 -a 10 -b 5
```

According to its rank, each process will compute its processing limits as follows:

```
m_Xfrom = mpiRank * m_pX % m_X;
m_Xto = m_Xfrom + m_pX;
m_Yfrom = mpiRank * m_pX / m_X * m_pY % m_Y;
m_Yto = m_Yfrom + m_pY;
```

where `m_pX` and `m_pY` are the values specified by `-a` and `-b` parameters.

During the first part of the main loop in `Simulator.cpp` all oprations are local to an element. However, the second part interacts with neighbouring elements. Thus, to compute the next iteration borders of the sub-grids, each process needs to get the rows and column of the neighbouring ranks encompassing its own sub-grid before the second processing part.

This acquisition is realized in 2 steps. First, each even ranked process sends its last row to the underneath neighbouring process. At the same time, the odd ranked processes hand their first row to the process above. Then, the same procedure happens in reverse order, odd processes handing their last row and even ones their first row.

```
for (int i = m_Yfrom / m_pY; i < m_Yfrom / m_pY + 2; i++)
  if (i % 2 == 0) {
    int under = coordsToRank(periodicIndex(m_Yto, m_Y), m_Xfrom);
```

```
    MPI_Sendrecv(&m_data[(m_Yto - 1) * m_X + m_Xfrom], m_pX * sizeof(T), MPI_BYTE,
under, 0,
                 &m_data[periodicIndex(m_Yto, m_Y) * m_X + m_Xfrom], m_pX *
sizeof(T), MPI_BYTE, under, 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  } else {
    int above = coordsToRank(periodicIndex(m_Yfrom - 1, m_Y), m_Xfrom);
    MPI_Sendrecv(&m_data[m_Yfrom * m_X + m_Xfrom], m_pX * sizeof(T), MPI_BYTE,
above, 0,
                 &m_data[periodicIndex(m_Yfrom - 1, m_Y) * m_X + m_Xfrom], m_pX *
sizeof(T), MPI_BYTE, above, 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  }
```

The same approach is applied for bordering left and right columns. However, as the columns are not contiguous in memory, we have created an MPI type to specify how many elements are in the column and the memory gap between them.

```
MPI_Type_vector(m_pY, sizeof(T), m_X * sizeof(T), MPI_BYTE, &MPI_YGHOST);
MPI_Type_commit(&MPI_YGHOST);
```

The final column sharing is as follows:

```
for (int i = m_Xfrom / m_pX; i < m_Xfrom / m_pX + 2; i++)
  if (i % 2 == 0) {
    int right = coordsToRank(m_Yfrom, periodicIndex(m_Xto, m_X));
    MPI_Sendrecv(&m_data[m_Yfrom * m_X + m_Xto - 1], 1, MPI_YGHOST, right, 0,
                 &m_data[m_Yfrom * m_X + periodicIndex(m_Xto, m_X)], 1, MPI_YGHOST,
right, 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  } else {
    int left = coordsToRank(m_Yfrom, periodicIndex(m_Xfrom - 1, m_X));
    MPI_Sendrecv(&m_data[m_Yfrom * m_X + m_Xfrom], 1, MPI_YGHOST, left, 0,
                 &m_data[m_Yfrom * m_X + periodicIndex(m_Xfrom - 1, m_X)], 1,
MPI_YGHOST, left, 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  }
```

If the wavefield output is not done in parallel, then it shall remain the rank 0 process' job to output it. To do this, it needs to gather the `degreesOfFreedomGrid` from all other processes at the beginning of each time step. The `Grid.h` now provides and implementation for that with the `gather` method.

```c++ void Grid::gather(int root) { if (mpiRank == root) { for (int i = 0; i < mpiSize; i++) if (i != root) { std::pair<int, int> coords = rankToCoords(i); MPI_Recv(&m_data[coords.first * m_X + coords.second], 1, MPI_SUBGRID, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE); } } else { MPI_Send(&m_data[m_Yfrom * m_X + m_Xfrom], 1, MPI_SUBGRID, root, 0, MPI_COMM_WORLD); } }

```

```
As with the columns, the sub-grids are not stored completely in a
contiguous memory area and thus we have create the `MPI_SUBGRID` type, defining
the length of each subgrid row and the memory gap between each consecutive rows.

```c
MPI_Type_vector(m_pY, m_pX * sizeof(T), m_X * sizeof(T), MPI_BYTE, &MPI_SUBGRID);
MPI_Type_commit(&MPI_SUBGRID);
```

The same gather procedure is applied at the end of the `simulate` function so that the error output in the main function may have the final computed `degreesOfFreedomGrid`.

Testing revealed a speedup proportional to the number of MPI processes.

OpenMP

As processing each element in the grid is independent from the rest, we can parallelize the two for loops iteration over the grid in the `Simulation.cpp`.

```
#pragma omp parallel for collapse(2)
for (int y = ylimits.first; y < ylimits.second; ++y) {
  for (int x = xlimits.first; x < xlimits.second; ++x) {
```

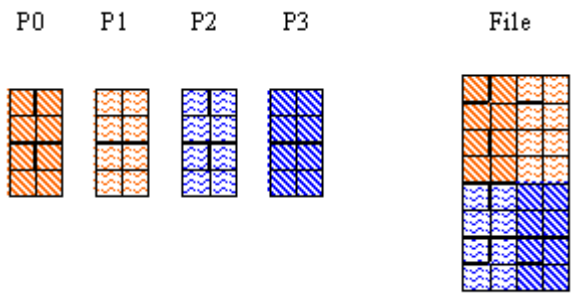
As the for loops remain unchanged, this approach can be applied on both the original and the MPI version.

Parallel Wavefield Output

When we enable output only the master process can write to the file. This means that all information needs to be gathered at the master process leading to overhead. In order to strive for a better time we tried to allow for parallel writes to a file. For this purpose we use the hdf5 file format which allows for writing multiple scientific datasets to the same file. The format functions as follows:

- Each dataset is written to a specific path in the file. For example: Pressure goes to filename:/pressure, uvvel goes to filename:/u and so on.
- Each process writes to a hyperslab governed by the starting and ending indexes (of the Grid) that it is working on.

This process is illustrated in the following figure:
```



The XDMF format that paraview uses has readers for HDF5 files built in. Hence, we only need to point to the correct path for the dataset as illustrated above and change the format from 'Binary' to 'HDF'.

After these changes no gather step needs to be done at the master and each process is responsible for writing its own wavefield output.