DD2356 VT25 Methods in High Performance Computing
Final Project: 1D FDTD Simulation

Rishi Vijayvargiya[†] rishiv@kth.se
Paul Mayer[†] pmayer@kth.se
Lennart Herud [†] herud@kth.se

Prefix

The code for our project can be found at this location: `https://github.com/paulmyr/DD2356-MethodsHPC/tree/master/5_project`.

Contents

[†]Authors made equal contribution to the project

1 Serial Implementation and Correctness

The serial implementation provides our optimization baseline.

1.1 Serial Runtimes

We varied the grid size on Dardel on 1 node and examined the runtimes of the provided serial implementation. Figure 2 shows the runtimes obtained. The runtimes increase exponentially (linearly on a logarithmic scale) with an exponential increase in the input size. The *input size* throughout this report refers to the number of elements in the `E` and `H` lists. The Slurm script required to generate this plot along with the runtimes can be found in the `1_baseline` directory of the repository.

The results in Figure 2 are within our expectations, as we would expect the amount of work to be done (without any parallel intervention) to increase in proportion with the increase in the input size.

1.2 Correctness

We check the implementation correctness of our optimizations against the baseline implementation by visual inspection of the intermediate and final values of the e-field. To implement this, the e-field values are printed into separate files and matched against each other. Additionally the peak position at the final time step is validated against the peak position of the serial implementation.

For both verification methods our results comply with the given code, yielding identical results. Figure 12 shows the final e-field for all implementations. Due to the scaling these e-fields look like a single Dirac-impulse instead of two pulses. The numerical results in Table 1 show that there are indeed two independent and symmetrical peaks, which are identical with the estimated peaks of the serial implementation.

Additionally we checked for different `NX` sizes. Figure 13 shows that for large enough grid sizes ($NX > 400$) the simulation result of the given reference simulation is stable and yielding two symmetrical peaks, while for small grid sizes ($NX <= 400$) this tends to shift to point symmetrical peaks, which we assume is due to the absorbing boundary conditions interfering. To adjust for this, we chose sufficiently large grid sizes ($NX > 400$) to ensure correctness and large enough array sizes to generate computational load for our implementations.

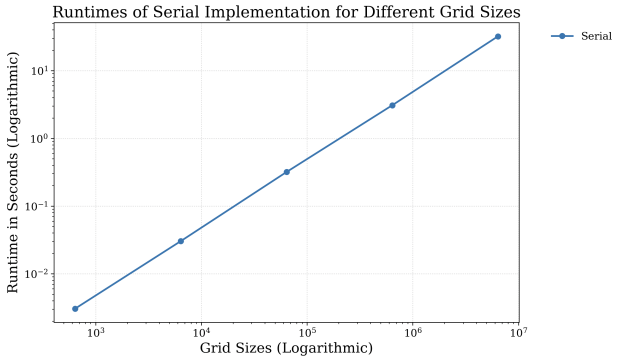


Figure 2: Serial Implementation Runtime

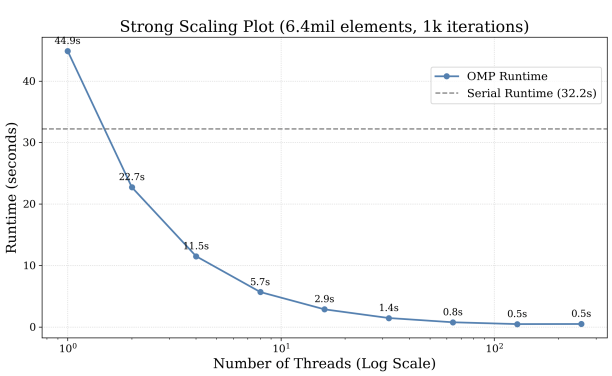
2 Parallelism through OpenMP

The code for this section is present in the `2_openmp` directory. The `dardel_runtimes` subdirectory contains the runtimes plotted in the graphs below, which were averages across 3 runs. Additionally, the `outputs/` directory contains the snapshots of `E` and `H` at different time-steps for the serial and parallel implementation(s), which were used to verify correctness using the `verify_outputs.sh` script.

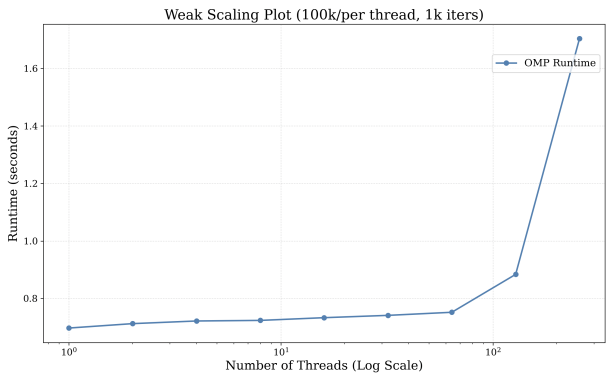
2.1 Implementation description

We started the optimisation process using shared memory parallelisation. The main loop of the computation exists of two update methods, each of them performs a simple `for-loop`. We noticed that all iterations of the loop are embarrassingly parallel. The easiest parallelisation strategy is to split up the iterations on multiple workers. Since we assume a shared memory architecture, communication is needed, in the sense that workers have to exchange information.

OpenMP provides a very simple but effective directive for exactly this situation: `#pragma omp parallel for`. When checking on Dardel, which scheduling is used as a default, we found that the default is `#pragma omp parallel for shedule(static, 0)`, meaning that the scheduling used is a static scheduler with unspecified chunk size. We tried different schedul-



(a) Strong Scaling (6.4 mil, 1k iters)



(b) Weak Scaling (100k/thread, 1k iters)

Figure 1: Strong and Weak Scaling (OpenMP)

ing approaches, different chunk sizes or initialising only a single parallel region once; however, we were not able to outperform this first, most simplest approach. In the OMP documentation[1] we find:

> When kind is static, iterations are divided into chunks of size chunk_size, and the chunks are assigned to the threads in the team in a round-robin fashion in the order of the thread number. Each chunk contains chunk_size iterations, except for the chunk that contains the sequentially last iteration, which may have fewer iterations. When no chunk_size is specified, the iteration space is divided into chunks that are approximately equal in size, and at most one chunk is distributed to each thread. The size of the chunks is unspecified in this case.

This means, that the chunk size is approximately $NX/NThreads$.

2.2 Strong and Weak Scaling

We performed two types of runtime tests: strong and weak scaling. Figure 1 displays the runtimes we measured using the static scheduling.

The strong scaling was performed by allocating one full node on Dardel, meaning one node, one process per node and 256 CPUs per process.

The only parameter changed was the `OMP_NUM_THREADS` environment variable. Each configuration was run three times, we display the average runtime in the plot. The grid size for all measurements was performed with $NX = 6.4 * 10^6$, $NSTEPS = 1000$ and the `OMP_NUM_THREADS` $\in [2^0, 2^1, ..., 2^8]$. For more information on standard deviation, min and max runtimes, please see the appendix on page 9. The runtimes are depicted in Figure 1a. We achieve logarithmic speed-up in the string scaling case, which peaks for 128 threads. For 256 threads the runtime remains the same (a tiny bit worse in performance than 128 threads).

In each weak scaling computation, we fixed the grid size for each working thread to 10^5. The node allocation as well as all other parameters were the same as for the strong scaling experiments. The runtimes are depicted in Figure 1b. For thread counts $2^0, ..., 2^6$ see a near constant runtimes, which is close to optimal performance; however, when increasing to 128 and then 256 we see a sharp drop in performance.

2.3 Discussion

The first important question we have to ask, is: 'why does the default scheduler outperform the other schedulers?' In our assessment, it mainly comes down to the fact that each loop iteration has an approximately equal load. This leads us to believe that reducing any scheduling overhead by using a static scheduler is the main reason for the good performance. Different chunk sizes mostly do more harm than good. There is not much reason why smaller chunk sizes would outperform bigger ones, when computational intensity does not vary for different loop iterations. In the worst case, for very small chunk sizes, it mostly interferes with cache access patterns.

To correctly interpret the weak scheduling runtimes, it is important to understand the topology of a compute node on Dardel.

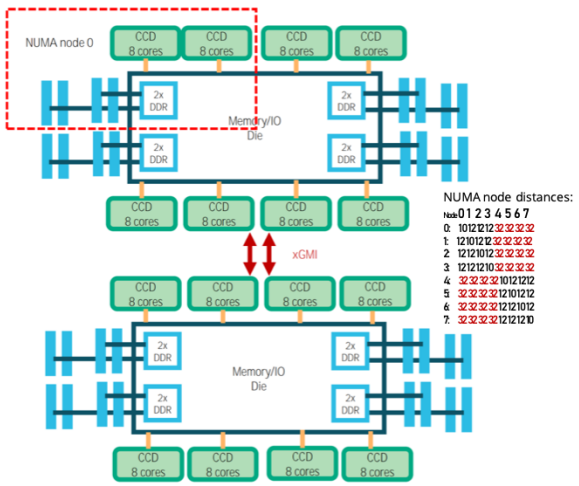


Figure 3: Dardel node topology. Image taken from 'Dardel sustainability at a glance'[BS23]

Each node consists of two sockets with two CPUs respectively. Each of those nodes has 4 NUMA nodes, each NUMA node has 16 physical cores (32 virtual cores) available. That makes a total of 128 physical cores and 256 virtual cores. Figure 3 depicts the overall architecture of the node. We can also see the NUMA node distances.

Due to the fact that we ran all simulations on a fully allocated node, we expect that the OMP-Scheduler makes use of the available resources as good as possible. This means that if we run the code with the `OMP_NUM_THREADS` environment variable set to 16, we expect OMP to schedule them on 16 distinct physical cores. Under this assumption, the weak scaling (see 1b) behaves as expected. For the first 64 threads, the runtime is mostly constant. We assume that all threads run on the same socket, meaning very small NUMA node distances when performing memory operations. However, when running the simulation with 128 threads, we can see a significant decrease in performance. We attribute this to the fact that OMP now has to schedule on the second socket as well (to make use of more physical cores), which increases the runtime due to memory synchronisation issues. When scaling to 256 threads, the performance nearly halves. This is also expected, because the physical core limit has been reached, and now threads are scheduled to virtual cores. Under the assumption that each of the previous 128 threads has a high CPU load, making use of virtual cores has a small effect on the performance.

3 OpenMP GPU Hand-off

The code for this section is present in the `2_openmp` directory. The `dardel_runtimes` subdirectory contains the runtimes plotted in the graphs below, which were averages across 3 runs. Additionally, the `outputs/` directory contains the snapshots of `E` and `H` at different time-steps

[1] `https://www.openmp.org/spec-html/5.0/openmpsu41.html#x64-1290002.9.2`

for the serial and parallel implementation(s), which were used to verify correctness using the `verify_outputs.sh` script.

3.1 Implementation Description

The approach is the same as described in section 2.1. However, this time we use the directive that tells OMP to hand off the computation to the GPU. The complete directive can be found in listing 11 in the appendix.
`#pragma omp target teams distribute` signals OMP to use the GPU, whereas
`map(to : E[0 : NX]) map(tofrom : H[0 : NX])` specifies which data needs to be copied to the GPU and which data has to be copied from the GPU back to RAM. In this for loop we update the `H` array, therefore it is sufficient to copy `E` to the GPU and not copy it back to memory — for `H` we copy to and from the GPU.

3.2 Runtimes

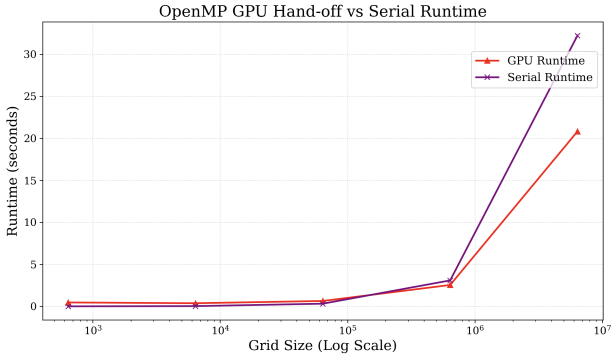


Figure 4: GPU runtime scaling over different grid sizes.

We performed all runtime tests on the GPU partition of Dardel. We allocated a single node, single process per node, single CPU per process as well as one GPU. Figure 4 displays the runtime over different grid sizes, that we achieved the experiment setup described above. A trend is clearly visible, for small grid sizes the runtime differences are minimal, however for bigger grid sizes we start to see big improvement in runtime. It is important to note that the grid sizes for this problem are still relatively small. We restricted ourselves to not overly congest the system; however, we do expect the trend visible continue for larger, more mature grid sizes.

3.3 Discussion

We expect the GPU runtimes to be much more competitive to the other optimisation methods that we present in this report. However, for the grid sizes used in our experiments, the GPU hand-off performed not as efficiently. We contribute this to the fact that the problem sizes for our experiments where still to small, making the overhead of copying data from and to the GPU the main bottleneck of the computation. Also, increasing the number of loop iterations (increasing `NSTEPS`) of the computation would probably aid the GPU computation a lot compared to the serial computation, since the copy operation is only performed once before and after the loop, effectively reducing the relative overhead.

There is still a lot of optimisation potential using the GPU, however, we did not focus on that more in this report, since this is not the main goal of this course. Future work (maybe in the context of the applied GPU programming course) could further investigate this potential.

4 Parallelism through MPI

The code for this section is present in the `3_mpi` directory. The `dardel_runtimes` subdirectory contains the runtimes plotted in the graphs below, which were averages across 3 runs. Additionally, the `outputs/` directory contains the snapshots of `E` and `H` at different time-steps for the serial and parallel implementation(s), which were used to verify correctness using the `verify_outputs.sh` script.

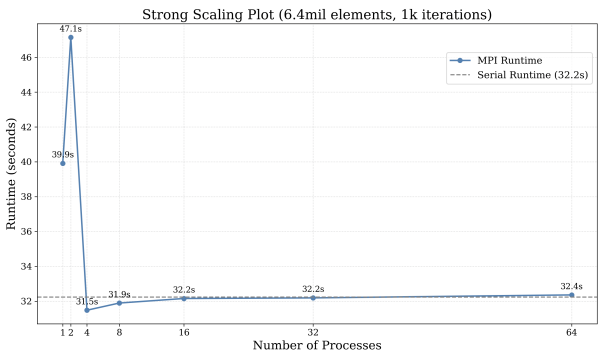
4.1 Implementation Description

To parallelize the serial code with the use of MPI, we first initialize the global `E` and `H` grids with the process with rank 0. This is followed by a scattering of this global grid to all processes in the method present here in the `fdtd_mpi.c` file. This scattering occurs over a 1D Cartesian communicator, which is first created with the help of `MPI_Cart_create`.

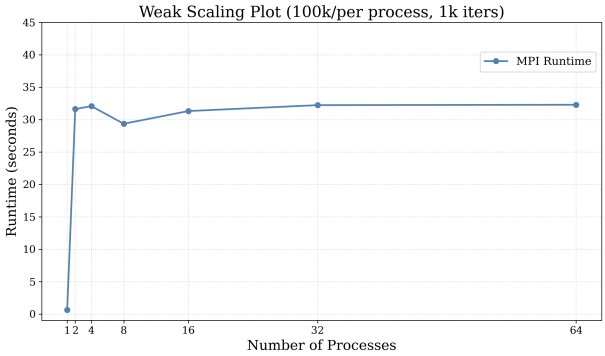
After this, the following three steps are performed at each step of the update-loop:

- Halo-Exchange for E: Each process sends the first value of the chunk assigned to it to its left neighbour and receives a value in return from its right neighbour. This is done through the blocking `Sendrecv` method to make the exchange easier to reason about.
- Compute H: The update for the `H` array are performed. These would have required the most up-to-date values for `E`, which is why the halo exchange for `E` occurred in the previous step. We take special care of the boundary condition for `H` when the process performing the update is responsible for the last chunk of `H`.
- Halo-Exchange for H: Each process sends the last value of its updated `H` chunk to its right neighbour and receives a value from its left neighbour. This is done in an analogous way to the halo-exchange for `E`.
- Compute E: We finally update the contents of `E`, using the updated values from the `H` ghost cells received in the previous step. This is done in an analogous manner to the updates to `H`, with care taken of the boundary condition in case the process is updating the last chunk of E.

Note that here, all halo-cell exchanges are performed through blocking, synchronous communication. Finally, once the computation loop is complete, we add a barrier to ensure that all processes agree to completing all steps of the computation before proceeding. After this barrier, we gather the results of the local `E` and `H` arrays from each process in to a global `E` and `H` grid (respectively) at the rank 0 process in this function. This can then be printed to file for diagnostic purposes if needed. The full-code for our implementation briefly described above can be found in the `fdtd_mpi.c` file here.



(a) Strong Scaling (6.4 Million, 1k Iters)



(b) Weak Scaling (100k/Process, 1k Iters)

Figure 5: Strong and Weak Scaling (MPI)

4.2 Strong and Weak Scaling + Communication Overhead

The strong and weak scaling plot for the implementation can be found in Figure 5. The configurations used to generate these runtimes can be found in the `run_strong_scaling.sh` and `run_weak_scaling.sh` files.

From Figure 5a, we see worse runtimes for 1 or 2 processes, which could be attributed to the greater MPI overhead compared to the benefits of the limited parallelism through 1 or 2 processes. As the number of processes increase, the runtimes are marginally better – achieving their lowest at 4 processes and then increasing till they get slightly worse at 64 processes. This trend indicates that the benefits of parallelism start to diminish as the number of processes increase, likely because of greater MPI overhead (halo-exchange, barrier at the end of the compute-loop, etc.) and an over-burdening of resources on 1 Dardel node once we have more than 16 processes involved in a communication. Thus, a balance is struck at 4 processes between MPI overhead and parallelism which is optimal. However, there isn't a big variation in the runtimes at ≥ 4 processes, especially from 4 to 8 processes. Thus, it could be the case that a different run could give better runtimes at 8 processes instead of 4. However, the sharp decline in runtime from 2 to 4 processes shows the promise of MPI parallelism if we use 4-8 processes.

Figure 5b shows that after a sharp increase in runtime going from 1 to 2 processes, the runtime seems to be in the stable 29-33 second range for all process counts. The ideal scenario for a weak-scaling test would be to have runtimes remain stable as the problem size increases – as this would indicate that larger problems can be solved with the help of larger resources. Except for the jump on going from 1 to 2 processes (because of the MPI overhead with regards to blocking halo-exchanges, etc), we see this ideal trend we hoped for in the weak-scaling test.

```
rishiv@login1:~/Public/project/4_opt> scorep-score profiling_mpi_64/profile.cubex

Estimated aggregate size of event trace:                   18MB
Estimated requirements for largest trace buffer (max_buf): 280kB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):       4097kB
(hint: When tracing set SCOREP_TOTAL_MEMORY=4097kB to avoid intermediate flushes
 or reduce requirements using USR regions filters.)

flt     type max_buf[B]  visits time[s] time[%] time/visit[us]  region
         ALL    286,200 385,216 2209.62   100.0         5736.05  ALL
         MPI    190,082 128,960 2175.11    98.4        16866.56  MPI
         COM     48,072 128,192    0.98     0.0            7.63  COM
         USR     48,000 128,000   33.49     1.5          261.61  USR
     SCOREP          46      64    0.04     0.0          641.50  SCOREP
```

Figure 6: ScoreP on MPI (6.4 Million Elements, 1k Iters, 64 Processes, 1 Node)

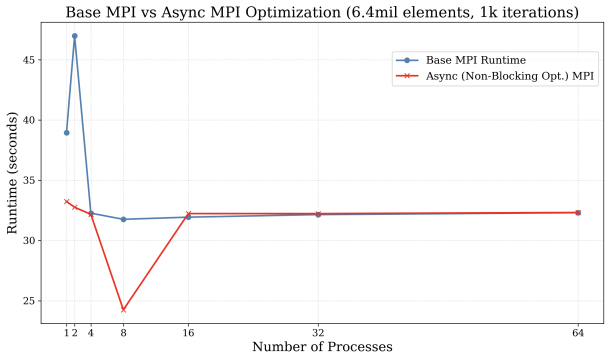
In addition to these tests, we profiled this MPI implementation with ScoreP to get an idea of communication overhead involved with halo-exchanges, using 6.4 million elements and 1k iterations with 64 processes on 1 Node. The results of this can be seen in Figure 6.

We notice a considerably large amount of time being spent in the `MPI` portion of code at a staggering 98.4% of the total time and an incredibly high `time/visit`. The MPI calls done in the compute-loop we are interested in are the blocking `Sendrecv` communication calls for halo-exchanges. Thus, this indicates that because of its blocking nature, each `Sendrecv` call is costing a significant amount of time, bumping up the communication overhead and thus the runtime (total time and time per visit) for the MPI section. In comparison, the `USR` section has a similar number of visits but a drastically lower time spent per visit. This might also explain why we did not see a significant improvement in runtime compared to the serial version as the number of processes increase – any benefit gained from parallelism was likely countered by the overhead of blocking communication between more processes. With an increase in the number of processes, the sub-problem size and the chunk of the compute loop gets smaller. Hence, more time is spent waiting for halo-exchanges to complete than in the local computation by each process. Thus, performing halo exchanges in a non-blocking manner could lead to some improvements, which is what guided our optimization strategy in the next section.

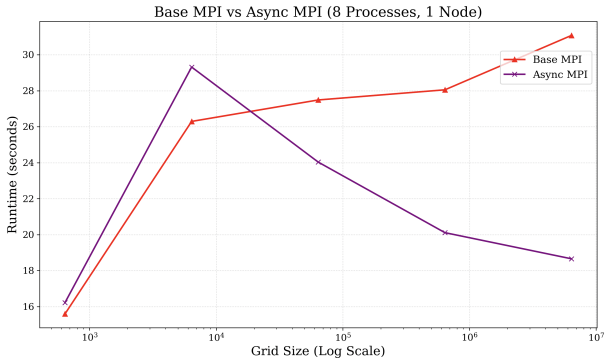
5 Optimizing MPI Parallelism

For optimization of the parallel implementations, we focus on the parallel MPI implementation in Section 4. The OpenMP implementations were incredibly fast compared to the serial counterparts already, and we felt we could improve MPI a lot more. The files referred to in this section can be found under the `4_opt` directory of the repository. The `dardel_runtimes/` and the `outputs/` directory serve the same purpose as in Section 4.

5.1 Attempt 1: Communication-Computation Overlap w/ Non-Blocking Halo Exchange



(a) Varying Processes (Same Input)



(b) Varying Input (8 Processes)

Figure 7: Sync (Base) MPI vs Async MPI

From the ScoreP analysis of the MPI code in Section 4.2, we concluded that the blocking `Sendrecv` halo-exchanges are likely contributing to the lacklustre performance. To rectify this issue, we decided to implement halo-exchanges through non-blocking mechanisms: with the help of `Irecv` and `Isend`. This allowed us to overlap communication (through halo-exchanges) with computation: since the *interior* portion of each chunk can be computed without the need of data from halo-exchanges.

Thus, we initiate `Irecv` and `Isend` requests for a halo-exchange, then compute the updated values for the interior of each chunk. After this computation, we *wait* on these 2 halo-exchange requests to complete, after which we compute the boundary cell(s). We repeat this method for updating both the `E` and the `H` grids. The code for this can be found in the `fdtd_async_opt.c`. We will refer to this as the Async/Non-Blocking Optimization (Opt 1) of the MPI code from Section 4, while the code from Section 4 will be referred to as the Base MPI implementation.

Figure 7a shows the performance of this optimization compared to the blocking/base MPI implementation from Section 4. We vary the number of processes keeping the problem size constant (6.4 million elements, 1k iterations). The script for obtaining these runtimes can be found in the `run_async_opt.sh` file here. From Figure 7a, we can see that till 8 processes, the async MPI implementation performs visibly better than the synchronous version. We believe that the reason for this is the communication-computation overlap to update the interior of the grids makes more efficient use of the idle time that was otherwise wasted in the synchronous halo-exchange.

However, the performance from 16 processes onwards is quite similar, where any differences between the two runs could likely be attributed to being within a margin of error from each other. Thus, this shows that with an increase in the number of processes, the cost of synchronizing the MPI communication between different/adjacent workers adds up – negating the potential benefits of the asynchronous exchange. Additionally, as the number of processes increase but the problem size remains the same, we have more processes responsible for smaller chunks of the global arrays. This means that the *computation* part of the computation-communication overlap is smaller, and thus the processes are more likely to end up waiting for a longer time at the `MPI_Waitall` call after the interior-computation is finished. Finally, going from 8 to 16 processes on a single Dardel node (which has 8 NUMA nodes) could over-burden the resources on a single Dardel node and could thus be a contributing factor to the communication overhead. All these reasons would give us similar behaviour to the sync MPI implementation after a certain threshold, which is what we observed.

Based on the results from Figure 7a, we concluded that 8 processes seems to offer the best balance between async MPI communication and parallelism, and wanted to explore this further. This was done in Figure 7b, where we analysed the runtimes across different grid sizes for the 2 implementations, both using 8 processes. The SLURM script for this can be found in the `run_mpi_async_opt_compare.sh` file here. For smaller grid sizes, the two implementations seem to perform similarly, with the Base MPI even having a slight edge likely because of the simpler structure of the code and fewer calls to the MPI library making the async communication an unnecessary excess. Another reason might be the smaller *computation* portion because of smaller grid-sizes, as explained earlier.

However, as the grid sizes increase, we start seeing the noticeable benefit of using async over sync communication for halo-exchanges. As the grid sizes increase, so does the "computation" part of the overlap, meaning that more time is spent efficiently performing interior-updates be-

```
rishiv@login1:~/Public/project/4_opt> scorep-score profiling_mpi_8/profile.cubex

Estimated aggregate size of event trace:                   2225kB
Estimated requirements for largest trace buffer (max_buf): 279kB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):       4097kB
(hint: When tracing set SCOREP_TOTAL_MEMORY=4097kB to avoid intermediate flushes
 or reduce requirements using USR regions filters.)

flt     type max_buf[B] visits time[s] time[%] time/visit[us]  region
         ALL    284,856 48,152  264.37   100.0        5490.23  ALL
         MPI    188,738 16,120  196.75    74.4       12205.38  MPI
         COM     48,072 16,024    0.29     0.1          17.83  COM
         USR     48,000 16,000   67.33    25.5        4207.98  USR
     SCOREP         46      8    0.00     0.0         194.36  SCOREP
```

(a) Synchronous (Base) MPI ScoreP

```
rishiv@login1:~/Public/project/4_opt> scorep-score profiling_async_8/profile.cubex

Estimated aggregate size of event trace:                   3850kB
Estimated requirements for largest trace buffer (max_buf): 482kB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):       4097kB
(hint: When tracing set SCOREP_TOTAL_MEMORY=4097kB to avoid intermediate flushes
 or reduce requirements using USR regions filters.)

flt     type max_buf[B] visits time[s] time[%] time/visit[us]  region
         ALL    492,856 80,152  235.66   100.0        2940.10  ALL
         MPI    396,738 48,120  188.56    80.0        3918.50  MPI
         USR     96,000 32,000   46.79    19.9        1462.06  USR
         COM         72     24    0.31     0.1       12895.13  COM
     SCOREP         46      8    0.00     0.0         210.82  SCOREP
```

(b) Async (Optimized) MPI ScoreP

Figure 8: Sync (Base) MPI vs Async MPI ScoreP

fore hitting the eventual blocking `MPI_Waitall` call. This call then finishes quickly and brings forth the anticipated advantage. *Note: While the runtimes plotted in Figure 7b are averages across 3 runs, we did observe considerable differences between these runs for both the Base MPI and the Async MPI implementations. We believed this might be because of unexpected allocation of processes to CPUs on the Numa nodes by SLURM (making for variable communication time), and thus requested more processes on the same node to get better allocation. Despite this, we did observe some non-trivial variation between runs on the same invocation. However, the trend we observed seemed to largely follow the one presented here.*

We also profiled using ScoreP the Base MPI and the Async MPI implementation using 8 processes (6.4 million elements, 1k iterations). Figure 8a and Figure 8b show the result of the analysis for the Sync and Async implementations with this configuration, respectively.

Despite a greater percentage of time being spent in the MPI section for the Async implementation and a higher number of visits to MPI, we notice that the time per call for MPI is significantly lower compared to that for the Sync Implementation. We believe that this is because of the fewer amount of waiting time required near blocking calls (such as `MPI_Waitall`), since the relatively larger computation portion gives enough time for the halo-exchanges to have finished by the time we reach the blocking portion of the code. This is also evident in the total time (for `ALL`) spent in the simulation – where we see a nearly 30 second decrease in the Async version compared to the Sync version. While the actual running times are likely much higher for both implementations here because of the ScoreP profiling intervention, a 30 second difference is considerably large, which would likely translate to a noticeable difference *without* ScoreP as well – as was observed in the plots in Figure 7.

5.2 Attempt 2: OpenMP Parallelism w/ Non-Blocking Halo Exchange

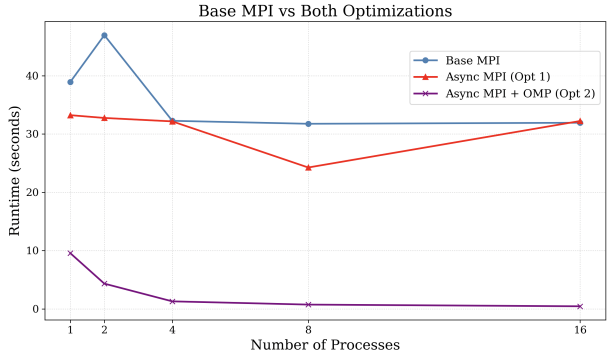


Figure 9: Async MPI + OMP Performance

We investigated if we could obtain noticeable runtime improvements over what was seen in Section 5.1, and were able to do so by incorporating OpenMP into the `E` and `H` updates. The code for this can be found in the `fdtd_async_omp_opt.c` here. The main changes here involved the use OpenMP parallelism in the compute-intensive loop-based updates into the *interior* of the chunk that each process is assigned. This snippet illustrates this for the `H`-interior updates.

Given this code, we investigated the impact of spreading the computation on up-to 4 Dardel Nodes, where we spawn up-to 4 processes (meaning we tested on up to 16 processes in total) on each Node and have 16 threads per process. The Slurm file used to obtain the runtimes with this configuration can be found here.

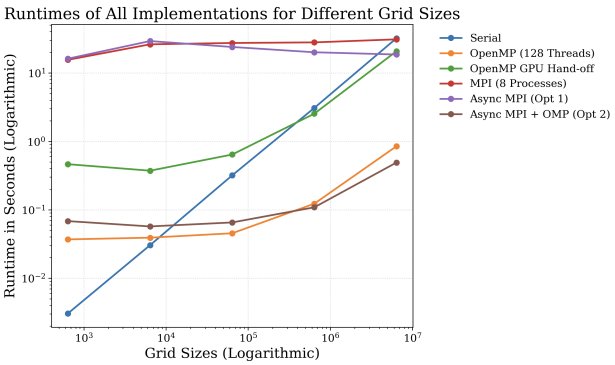
Figure 9 shows the comparison between the Async MPI + OpenMP implementation with the other two MPI implementations discussed earlier. The problem size is kept constant here (6.4 million elements, 1k iterations). We can see that unlike the Async-only MPI implementation (Optimization 1), the performance of the Async MPI + OpenMP implementation (Optimization 2) continues to improve as we go from 8 processes to 16 processes. One explanation for this could be that the use of OpenMP threads in the compute portion of the communication-compute overlap drastically accelerates the overall runtime of *each step* in the overall simulation. Thus, despite the compute part becoming faster (because of the use of OpenMP) and consequently there being less of an *overlap* between the compute and the communication, Optimization 2 makes each step of the compute more efficient, thus consistently reducing the runtime. In addition, we spawn a maximum of 4 nodes per process, which could be preventing an over-burdening of resources on each of the Dardel nodes and lead to more efficient (and quicker) MPI communication.

6 Conclusion

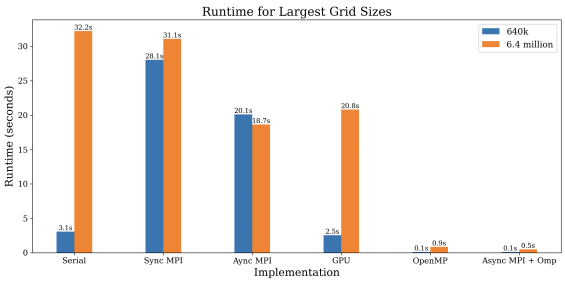
In the previous sections, we discussed the runtimes of the different implementations and optimizations of the 1D-FDTD simulation. Here, we present a final graph with the runtimes of each of the implementations discussed plotted against the serial runtime as we increase the grid size. For implementations that can have different configurations (such as: different number of threads in OpenMP), we have plotted the runtimes of the best configuration observed while examining the section. Figure 10a shows a line-plot for runtimes of different implementations on varying grid-sizes, and Figure 10b displays this runtime in a bar-graph format to truly highlight the performance gains.

One exception to this is the configuration chosen for the Base MPI implementation from Section 4, where we chose 8 processes instead of 4 for this plot. As discussed earlier, we believe that the placement of the processes on a Dardel node by SLURM could play a non-trivial impact in determining the runtime of the blocking MPI implementation (especially since there is no compute-overlap to *hide* the cost of the communication). While 4 processes gave the lowest runtime in the strong-scaling test in Section 4, it failed to give better runtimes than 8 processes when ran across different grid sizes in a separate run. As argued earlier, the runtimes for different 4 and 8 process counts in the Strong-Scaling test in 4 seemed to be within a margin of error from each other. Thus, we decided to go with 8 processes here, which gave lower runtimes on one of the runs (likely because of more favourable process-placements by Dardel on this run).

From Figure 10, the serial implementation outperforms all parallel strategies for the smallest grid sizes, demonstrating that the overhead required to setup the different



(a) Runtime for Different Grid Sizes



(b) Bar Plot for Largest Grid Sizes

Figure 10: Concluding Results

parallelization techniques overwhelms the amount of compute required to perform such small simulations. However, the serial implementation begins to be outshone by parallelization techniques very quickly.

The OpenMP implementations seem to perform the best out of all implementations, likely because of the sheer compute-acceleration offered by parallelizing the relatively simple and independent `E` and `H` updates in each simulation-iteration. Between these 2 approaches, the OpenMP only approach outperforms the MPI + OpenMP approach till a certain grid size. However, for the last 2 grid sizes, the Async MPI + OpenMP approach outperforms the raw OpenMP approach. This is likely because for bigger inputs, we begin to see the advantages of domain decomposition and better distribution of the input problem across different nodes and processes, reducing the load on a single node/process.

After these two implementations, we see that the GPU hand-off technique seems to perform reasonably well. The initial cost of moving the data to and from the GPU seems to overburden the runtime for smaller grid sizes, but as the grid size increases, we begin to see the advantage of GPU parallelization over the serial implementation. A way to improve the GPU parallelization even more in the future might be to explore involving multiple GPUs in the computation, which would require decomposing and distributing the problem manually across these GPUs. This might worsen runtimes for smaller grids, but might lead to better runtimes for much larger grids.

Finally, the MPI-only implementations seem to perform very poorly for small-moderate grid sizes, likely because the overhead of the halo-exchange involved in the split-input domain does not offer any significant advantage over simpler implementations. This trend remains true for the Synchronous MPI implementation, which isn't able to take advantage of the communication-compute overlap. The Async-only MPI implementation, on the other hand, is able to overlap compute with communication for halo exchange and sees runtimes go down as the grid sizes increase (compared to serial and Sync MPI) as the compute is able to *hide* the communication cost more effectively.

Thus, from this experiment, we believe that the best approach would be to use the serial implementation for small input sizes, then transition to the OpenMP only approach for moderate input sizes, and finally to switch the to the Async MPI + OpenMP approach for large input sizes.

References

[BS23] Michaela Barth and Gert Svensson. *Dardel sustainability at a glance*. [Online; accessed Jun 2, 2025]. Aug. 2023. URL: `https://www.pdc.kth.se/polopoly_fs/1.1271258.1692656955!/Dardel_Sustainability.pdf` (cit. on p. 3).

Appendix

Std Dev, Min, Max Runtimes

For brevity, we did not include the std dev, min, max of the runtimes obtained in the main report itself. The most important ones, however, be found in text-format in the repository, under the `misc/stats_for_nerds` directory here. Below we include some of the relevant files with these stats that might be of interest.

- `serial_grid_vary.txt`: Serial runtimes for different grid sizes. Plotted in Figure 2.
- `omp_grid_vary.txt`: OpenMP Runtimes for different grid sizes (with 128 threads). Plotted in Figure 10.
- `omp_strong_scaling.txt`: OpenMP Runtimes for Strong Scaling test.
- `omp_weak_scaling.txt`: OpenMP Runtimes for Weak Scaling test.
- `gpu_grid_vary.txt`: GPU Runtimes for Different grid sizes.
- `mpi_strong_scaling.txt`: Runtimes for Strong-Scaling test of the base (sync) MPI implementation. Plotted in Figure 5a
- `mpi_weak_scaling.txt`: Runtimes for Weak-Scaling test of the base (sync) MPI implementation. Plotted in Figure 5b
- `sync_async_mpi_grid_vary.txt`: Runtimes for Base MPI (Sync) and Async (Opt 1) MPI implementation for different grid sizes using 8 processes. Plotted in Figure 7b and Figure 10.
- `async_mpi_process.txt`: Runtimes for Async MPI (Opt 1) implementation for same input size (6.4 million elements, 1k iteration), along with the Sync MPI runtimes in the same run. Plotted in Figure 7a and Figure 9.
- `async_omp_process.txt`: Runtimes for Async MPI + OpenMP (Opt 2) implementation for same input size. Plotted in Figure 9
- `async_omp_grid_vary.txt`: Runtimes for different grid sizes for Async MPI + OpenMP (Opt) implementation. Plotted in Figure 10

GPU hand-off code listing

```
// Function to update the magnetic field H
void update_H(double *E, double *H, int NX) {
  // Update H from 0 to NX-2 (using forward differences)
#pragma omp target teams distribute parallel for map(to : E[0 : NX])            \
    map(tofrom : H[0 : NX])
  for (int i = 0; i < NX - 1; i++) {
    H[i] = H[i] + (DT / DX) * (E[i + 1] - E[i]);
  }
  // Simple absorbing boundary condition:
  H[NX - 1] = H[NX - 2];
}
```

Figure 11: OMP pragma used to hand off computation to GPU.

Verification results

The final e-field of all implementations is displayed in the following plot.

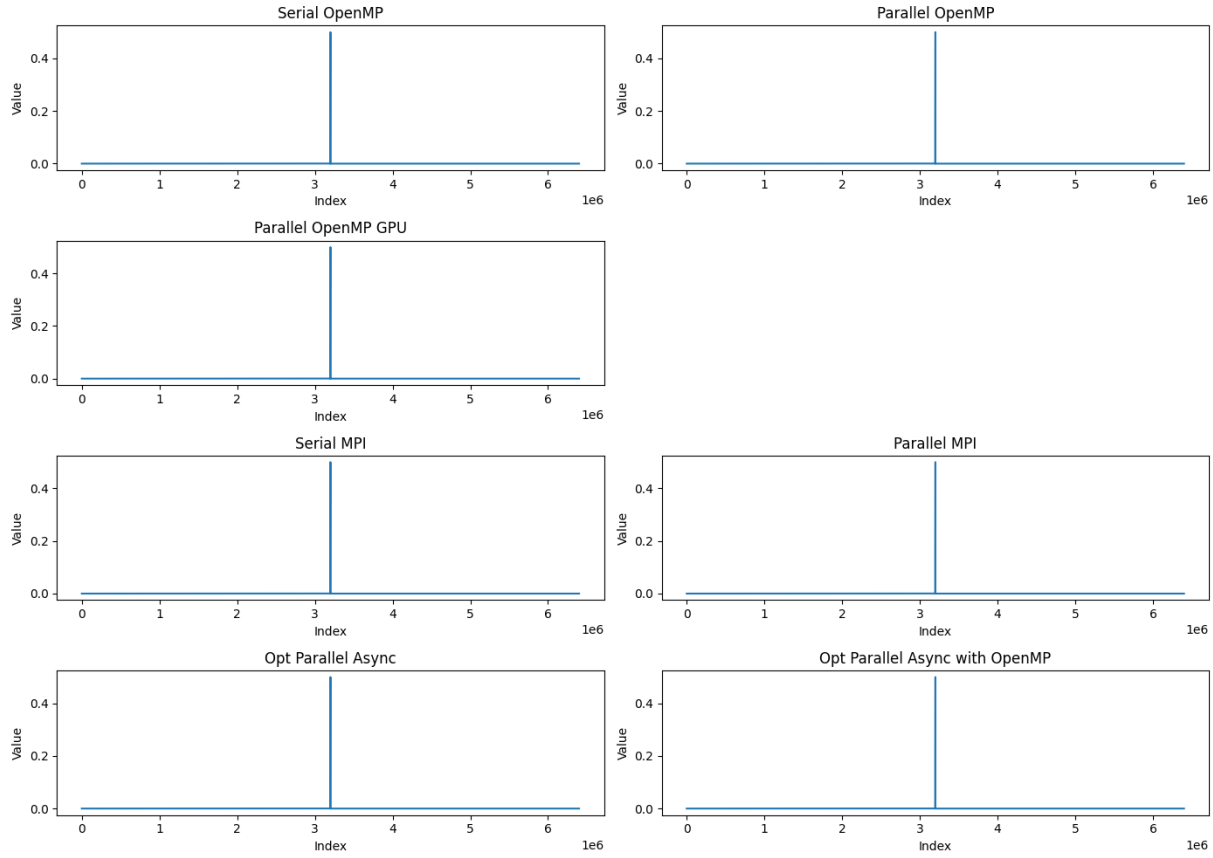


Figure 12: Serial Implementation Runtime

Numerical results for the peaks of the final e-fields are shown in the table below and yielding the identical values and indices (x-Axis) as the estimated peak of the serial implementation. `NX` is set to 6.4e6.

Table 1: Runtime values and indices for different implementations

OpenMP Implementation

Implementation	Value	Indices
Serial	0.499674	[3199500, 3200500]
GPU	0.499674	[3199500, 3200500]
OMP	0.499674	[3199500, 3200500]

MPI Implementation

Implementation	Value	Indices
Serial	0.499674	[3199500, 3200500]
Parallel	0.499674	[3199500, 3200500]

Optimization Implementation

Implementation	Value	Indices
Serial	0.499674	[3199500, 3200500]
Parallel Async	0.499674	[3199500, 3200500]
Parallel Async w. OMP	0.499674	[3199500, 3200500]

Final e-field results of serial reference for different grid sizes (`NX`).

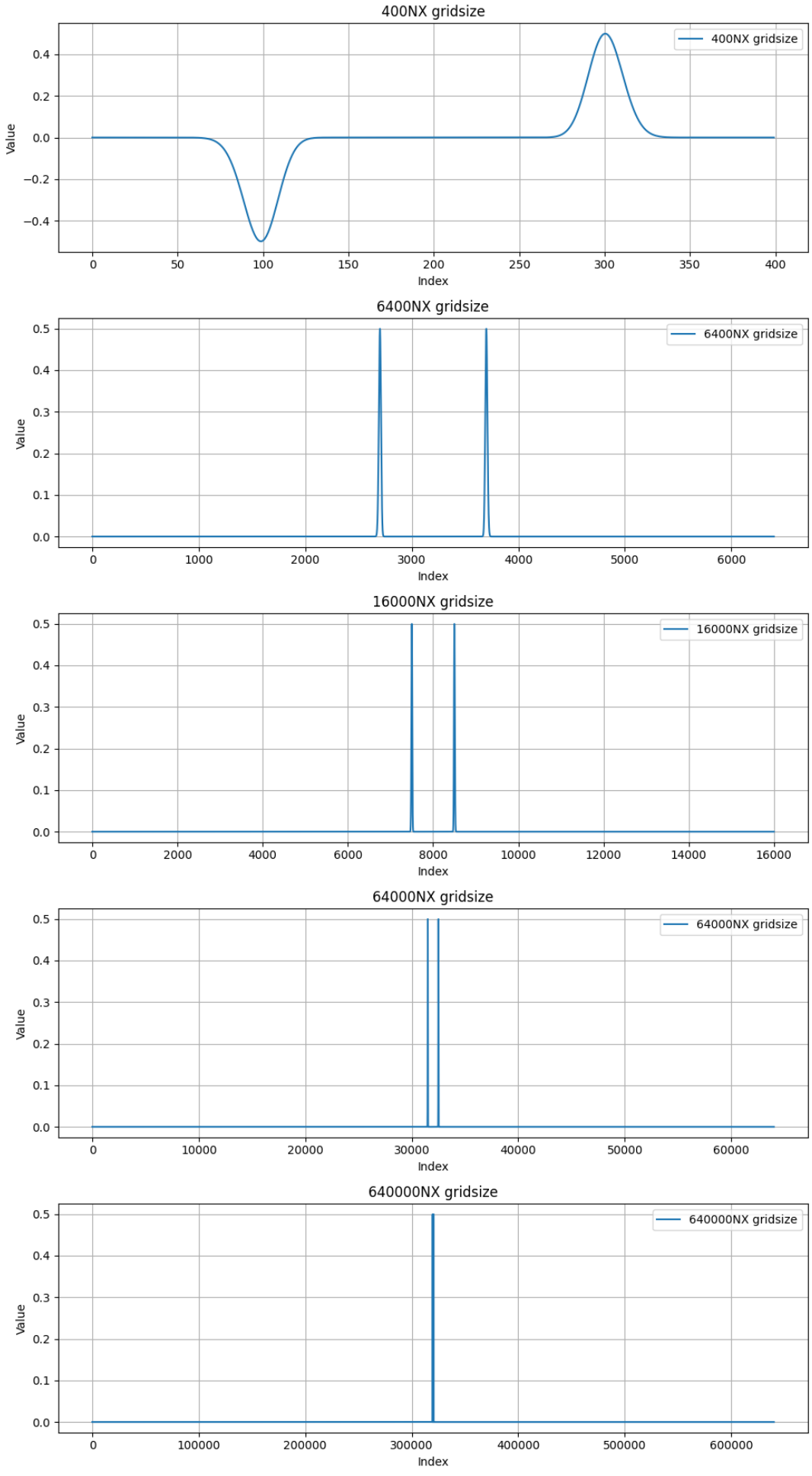


Figure 13: Serial simulation for different grid sizes (NX)