Université de Liège

Faculté des sciences appliquées

3D FDTD algorithm: parallel implementation with OpenMPI

Amaury Baret & Finta Ionut

High performance scientific computing (INFO0939)

December 2021

1 Introduction

In this report, we describe our implementation of the FDTD algorithm introduced by Kane S. Yee in 1966 [1]. As the mathematical details are presented in the project statement, we will not go over them again. First, we detail our implementation and the various tweaks used to make it more efficient. Then, we validate the results obtained by comparing them to the theoretical results presented in the project statement, and we analyze the energy of the system as a function of time. Next, we perform a stability analysis and we present corrected stability conditions. Afterwards, we briefly describe our implementation of the source. Subsequently, we perform a strong and a weak scalability analysis of our algorithm. Finally, we study the limits of our algorithm by performing a rough analysis of its arithmetic intensity, and we discuss our results. In the appendice, we present the update equations used for the FDTD algorithm, as requested in the statement.

2 Comments of implementation

2.1 Sequential implementation

In this section, we report the the global structure of our algorithm, as well as the various tweaks used to make our sequential implementation of the code more efficient.

First of all, all the parameters of the algorithm are stored in a Parameters structure. The fields are stored in a Fields structure, which contains 6 arrays corresponding to the E_i and H_i ($i = x, y, z$) components of the electric and magnetic fields. As updating a field does not require to know the values of this field at the previous timestep, the values in those arrays correspond to a single timestep. The choice to use a 1D array allows the loops to be more efficient by using the locality of the information. Furthermore, in order to fully benefit from this locality, we paid close attention to avoid cache invalidation. This was performed by iterating over the arrays in a less intuitive manner: first over the z axis, then the y axis, and finally the x axis.

Given the structure of the Yee cell (Fig. 1), we note that the various Field arrays do not have the same dimensions along the various axis, as explained in the project statement. Therefore, care was taken in order to correctly access the various indices. To do this, we wrote 6 different index functions, one for each field array. Since these functions were called a lot, we have declared them `inline`. This allows the compiler to replace each function by its content, so there is less overhead in assembler: we avoid a call, a push stack, a pop stack, and a jump at each index call. Compared with the version without the `inline`, this lead to an average speedup of 1.38.

In order to deal with the boundary conditions, the loops used to iterate over the field values do not all iterate over the same indexes. For the E field components, we did not want to update the components located on the borders of the simulation box, so that they remain equal to 0. As for the H field components, the boundary conditions were implicitly dealt with due to the fact that they only depend on the E field values, which already take those conditions into account.

2.2 Parallel implementation

In order to parallelize our algorithm we OpenMPI, we decided to divide the simulation grid into chunks of XY planes. In other words, the system was divided along the Z axis, as illustrated in Fig. 2. The first challenge consisted in equally sharing N_z planes to N processes. A naive approach would be to assign $\frac{N_z}{N}$

planes to each process, and to give the remaining planes (if N_z is not a multiple of N) to the last process. However, this would create a bottleneck as all processes would have to wait for the last process (which has more tasks to accomplish). In order to ensure a fair distribution of the N_z planes to the N processes, we have decided to assign $\frac{N_z}{N}+1$ planes to the $N_z\%N$ first processes, and $\frac{N_z}{N}$ to the others. This is the most equal distribution that can be achieved, as the maximum difference between the number of tasks to achieve for each process is 1. For instance, if we have $N_z = 13$ and $N = 5$, the first 3 processes are given $2+1=3$ planes each, while the last 2 processes will deal with 2 planes each. We can verify that $3\times 3+2\times 2=13$.

The next problem consisted in dealing with the boundaries of each chunk. As displayed in Fig.1, the update of the magnetic / electric fields in one cell depends on the electric / magnetic fields in the adjacent cells, respectively.

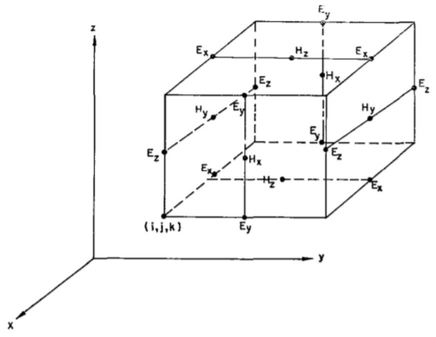


Figure 1: Yee cell, from [1]

Therefore, when updating the field values in cells located on the uppermost plane of the chunk, the algorithm needs to be able to access the field values of the cells of the lowest plane of the upper chunk. (along the Z axis). Similarly, the update of the cells on the lowest plane of a chunk requires that the fields on the uppermost plane of the chunk below are known. As a consequence, the cores need to exchange data. In practice, given the structure of the Yee cell and the update equations related, only the E_x and E_y need to be sent to the lower process, and only the H_x and H_y need to be sent to the higher process. The other fields values are indeed not required to update the field values on the boundaries. In order to perform the data exchanges, we assigned two more planes of data to each process' chunk. These two extra layers store the data sent by the upper / lower process and can be used to update the fields in the current chunk. This is represented in Fig. 2. We note that the values in those two extra layers must not be updated, as they are only used for the calculations.

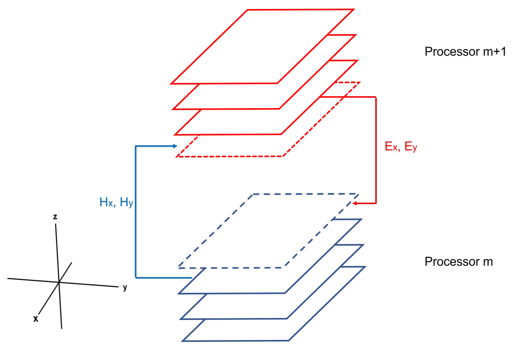


Figure 2: Exchange process between two chunks, inspired from [3]. The lower chunk sends its uppermost H_x and H_y data to the bottom storing layer (dotted) of the upper chunk. The upperchunk, however, sends the E_x and E_y values from its lowest plane to the upper storing layer (dotted) of the lower chunk. Note that those exchanges do not happen at the same moment (cf. Fig. 3).

The whole process of field exchanges is summarized in Fig. 3. We note that the source implementation is not represented. More information on the source implementation can be found in section 5.

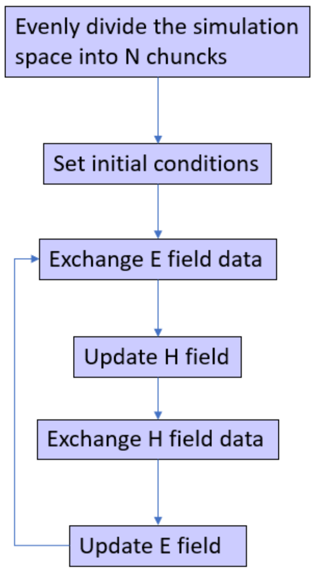


Figure 3: Propagation process of the FDTD algorithm with MPI. The loop is repeated until the total simulation time is reached.

The exchange of data itself is performed through the use of the `MPI_ISend` and `MPI_Recv` commands. The non-blocking Send makes the exchange process slightly more efficient as all the Sends can be performed in parallel, and must not wait until the previous Send-Rcv operation is completed. However, as a consequence of the non-blocking Sends, we needed to set a `MPI_Waitall` command at the end of the exchange function to make sure all the data has been successfully exchanged before starting the update of the fields.

Another issue was to be able to rejoin all of the chunks into a single Silo file. The best option would be to use a `MPI_Gather` in order to send all of the data to the main process (of rank 0). However, due to our even division of the payload, all processes do not have the same amount of data. Therefore, we have opted for an easier solution which consisted in using an `MPI_Send` to transfer each processor's data to the main process. Once the data has been recieved, the main process was in charge of creating the output Silo file. We also note that since we need to know the amount of data that will be recieved from each process, the main process allocates an array in which the sizes of all the other processes are stored. The values in this array are obtained thanks to a `MPI_Gather` that sends each process' k starting index to the main process at the moment the chunks are created.

3 Validation of the simulation

In order to validate our code, we imposed the following initial conditions, which correspond to the TE_{101} resonant mode:

$$E_y = \sin\left(\frac{m\pi x}{a}\right)\sin\left(\frac{l\pi z}{d}\right) \tag{1}$$

All the other field components are set to zero. $E_y(t=0)$ is displayed in Fig. 4

The analytical results that should be obtained for the E_y, H_x, H_z field components are well known and reported in the project statement. In order to verify that those results correspond to the ones produced by our

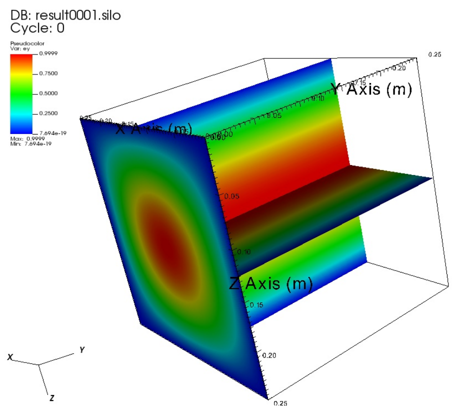


Figure 4: TE_{101} initial conditions for E_y on a $0.25 \times 0.25 \times 0.25 m^3$ box, $\Delta x = 0.001m$. The E_y field values are constant along the y axis, as expected.

simulation, we calculated the relative error ε_r associated to the whole grid at each time step. For instance, the relative error associated to the F_i component is computed as follows:

$$\varepsilon_r = \sqrt{\frac{\sum_\Omega \left|\mathbf{F_{i,c}} - \mathbf{F_{i,a}}\right|^2}{\sum_\Omega \left|\mathbf{F_{i,a}}\right|^2}} \tag{2}$$

Where F_i is a field component ($F = E, H$), $F_{i,c}$ is the value calculated with the algorithm, and $F_{i,a}$ is the analytical one.

At the final timestep for a simulation run on a grid of size $0.25 \times 0.25 \times 0.25 m^3$ with a $\Delta t = 10^{-11} s$ and for a total time of 100ns, we obtained $\varepsilon_r = 0.73\%$ for E_y, $\varepsilon_r = 0.046\%$ for H_x, and $\varepsilon_r = 0.027\%$ for H_z.

This value remained on the same order of magnitude for simulations ran with various parameters.

For this grid, the f_{mnl} and Z_{TE} obtained were:

$$\begin{aligned} f_{mnl} &= f_{101} = 847,941120 MHz \\ Z_{TE} &= 532.7884\Omega \end{aligned} \tag{3}$$

In order to analyze the accuracy and the stability of our algorithm, we verify that the electromagnetic energy in the simulation box is roughly equal to the theoretical energy associated to the resonant mode TE_{101} in a box with the aforementioned dimensions. The theoretical energy is given in Eq.4.

$$W = \varepsilon_0 abd/8 \tag{4}$$

where W is the total energy, a, b, d the dimensions of the box, and ε the absolute permittivity of free space. The electric and magnetic energies in a simulation domain Ω are given by

$$\begin{aligned} W_e &= \frac{\varepsilon_0}{2} \int_\Omega |\mathbf{E}|^2 \mathrm{dv} \\ W_m &= \frac{\mu}{2} \int_\Omega |\mathbf{H}|^2 \mathrm{dv} \end{aligned} \tag{5}$$

Where W_e and W_m are the electric and magnetic energies, respectively. In order to calculate this integral on the domain, we used the mean value of the field components in each cell. For the electric field components, there are 4 values around each cell used for the mean calculation, while only two are used for the magnetic field components. As an example, here is how the calculation of $\bar{E}_y$ is performed on a cell (i,j,k):

$$\bar{E}_y[i,j,k] = \frac{E_y[i,j,k]+E_y[i+1,j,k]+E_y[i,j,k+1]+E_y[i+1,j,k+1]}{4}\mathrm{dv} \quad (6)$$

Where $dv = \Delta x \Delta y \Delta z$. This calculation is depicted in Fig.5 for the E_y field component.

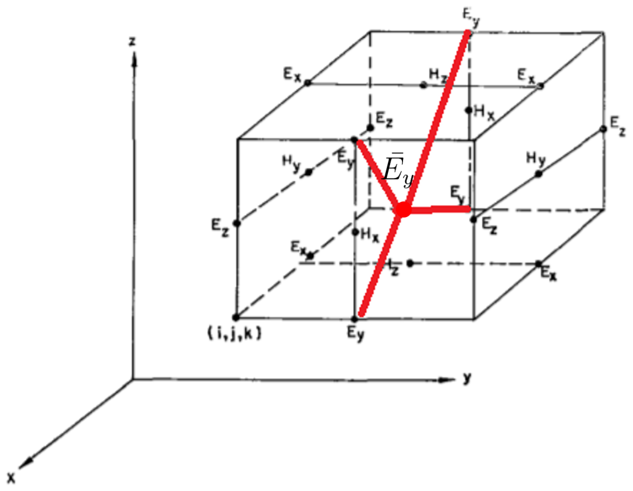


Figure 5: Example of mean field component calculation with E_y, adapted from [1].

The same was done for all other field components. We can then obtain $|\mathbf{F_{i,j,k}}|^2 = |F_{x,i,j,k}|^2 + |F_{y,i,j,k}|^2 + |F_{z,i,j,k}|^2$, where F is $\bar{E}$ or $\bar{H}$. The squared mean field values are then summed on the whole domain (on all i,j,k cells), and multiplied by $dv = dx \times dy \times dz$. The obtained results are shown in Fig. 6.

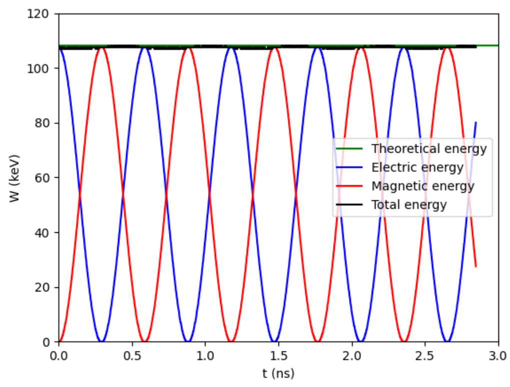


Figure 6: Evolution of the energy in the system as a function of time. The simulation parameters were a grid of $a = b = d = 0.25m$, $\Delta t = 10^{-2}ns, t_{max} = 3ns$, and $\Delta x = 0.001m$. We see that the electric and magnetic energies oscillate with a phase shift, giving a constant total energy (which corresponds to their sum) close to the theoretical one.

Since the system is isolated and no energy is given to or taken from the system, the total energy in the system must remain constant. As expected, we observe an exchange of energy between the electric and magnetic fields, but their sum remains relatively constant. We have calculated the error associated to the theoretical energy: the maximum relative error ($\frac{Calculated-theoretical}{theoretical}$) is equal to 0.2%. We believe this kind of error is expected, as our algorithm is not infinitely precise.

Finally, we observe that when $a = b = d$, the ratio E_{H_x}/E_{H_z} is constant and equal to 1 at all time steps.

This can be easily understood from the analytical shape of the H_x and H_z solutions. If $a = d$, we have:

$$\frac{E_{H_x}}{E_{H_z}} = \frac{\int_\Omega |\mathbf{H_x}|^2 \mathrm{dv}}{\int_\Omega |\mathbf{H_z}|^2 \mathrm{dv}} = \frac{(\omega \mu a)^2}{Z_{TE}^2} = 1 \tag{7}$$

This holds true at all times because the dependence in time of H_x and H_z is the same $(2\pi \sin(f_{mnl} t))$.

3.1 Stability analysis

The stability conditions first presented by Kane S. Yee in 1966 [1] were the following:

$$c\Delta t \leq \sqrt{(\Delta x)^2 + (\Delta y)^2 + (\Delta z)^2} \tag{8}$$

Where c is the speed of light in vacumm. However, A. Taflove has shown in a later paper (1975) that the stability conditions obtained by K. S. Yee were incorrect [2]. In reality, the stability criterion of the system is given by

$$c\Delta t \leq \left(\frac{1}{(\Delta x)^2} + \frac{1}{(\Delta y)^2} + \frac{1}{(\Delta z)^2} \right)^{-1/2} \tag{9}$$

In this project, we considered $\Delta x = \Delta y = \Delta z \equiv \Delta s$. As a consequence, we can easily represent the ensembles of $\{\Delta s, \Delta t\}$ values that satisfy the stability conditions expressed in Eq.8 and 9. They are shown in Fig.7. In addition to the stability regions, Fig. 7 also displays the stability results (stable or unstable simulation) obtained with our algorithm. As expected, our results are in good accordance with the corrected stability criterion from A. Taflove, altough some runs outside the stability domain appeared to remain stable. We believe a longer simulation time might have shown their divergence.

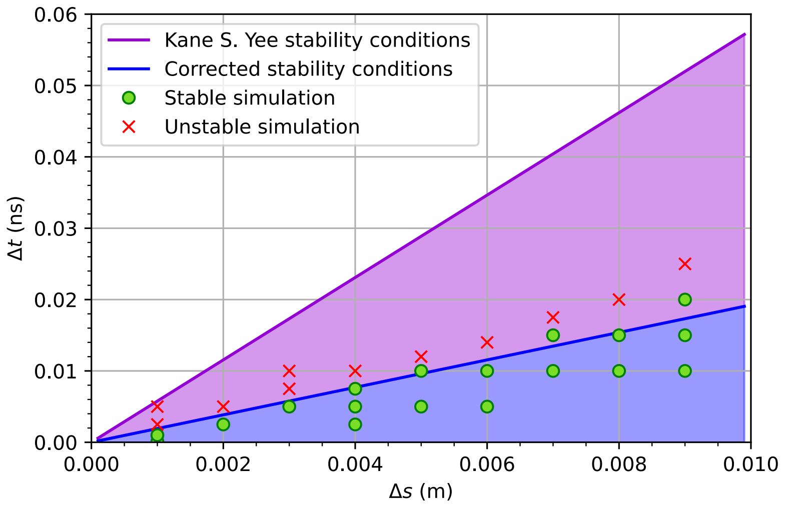


Figure 7: Stability domains from Kane S. Yee (purple) and A. Taflove (blue). We note that the purple domain extends to the x axis, but is shadow by the blue domain. The stability results obtained with our algorithm are displayed in green dots and red crosses for stable and unstable simulations, respectively. The simulations were run on a box of size $0.5 \times 0.5 \times 0.5 m^3$, with 16 cores and for a maximum simulation time of 100ns (the process was stopped if the energy diverged to infinity).

4 Source implementation

In this section, we describe our implementation of the source as a waveguide port. In order to make the source initialisation function easier to write, we decided to place it on the bottom $z = 0$ plane: since our space was divided along the z axis, this allowed us not not to have to worry about placing the source at the correct location on the various processes. The source was set to the TE_{10} mode, in accordance with the project statement. As a consequence, the source field only had E_y and H_x components ($E_x = H_y = 0$):

$$\begin{aligned} H'_x &= -Z_{TE}^{-1} \sin(2\pi f t) \sin\left(\frac{x'\pi}{a'}\right) \\ E'_y &= \sin(2\pi f t) \sin\left(\frac{x'\pi}{a'}\right) \end{aligned} \tag{10}$$

With $f = 2.45GHz$. We display the results obtained for a source of size $a' = 0.1m$ and $b' = 0.05m$ in Fig.8

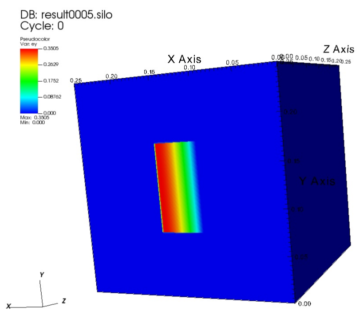


Figure 8: Source of size $0.1 \times 0.05m^2$ on a grid of $0.25 \times 0.25 \times 0.25m^3$. It is located on the $z = 0$ plane.

When the simulation mode is on, the source is artificially set to its value (given by Eq. 10) after each E or H field update. We also note that we made it so that the source is centered on the $z = 0$ place, no matter what its dimensions are (given that they are smaller than the xy plane, of course).

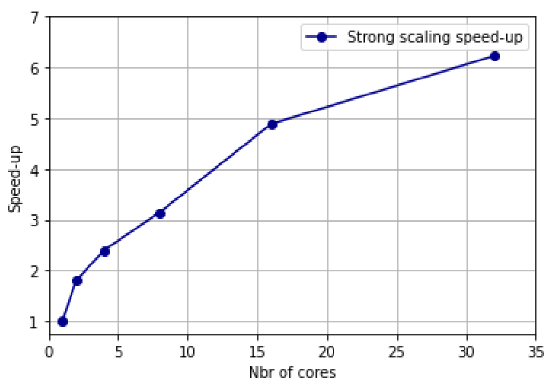
5 Scalability analysis

In this section, we analyze both the strong and weak scaling of our implementation. As a quick reminder, we note that the strong scaling is a measure of how the computational time of an algorithm scales with the number of cores allocated. We can measure it by simply calculating the speed-up S:

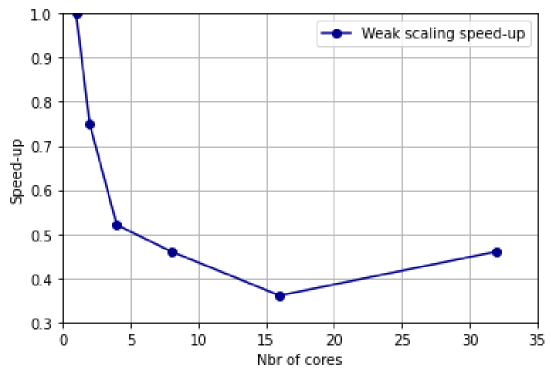
$$S = \frac{T_1}{T_n} \tag{11}$$

Where T_1 and T_n are the times required to perform the same simulation with 1 and n core(s), respectively. The speed-ups of our algorithm are represented in Fig. 9a. We can see that we are quite far from the theoretical perfect linear evolution. This is explained by the time required to perform the various communications between the processes, and the fact that a process must wait for other processes to be done before continuing its calculations. Furthermore, the process of rank 0 has more calculations to perform, as it is responsible for joining all the chunks into a single field array that will be used for the output. This induces the loss of parallelism, which makes the strong scaling less efficient than for embarrassingly parallel problems. We note that the benchmark did not take into account the output of the data to Silo files. Since this output procedure is sequential and requires much more time than the calculations themselves, it would make the speed-up almost equal to 1 for any number of cores. The obvious improvement would be to make the output procedure parallel too.

On the other hand, the weak scaling is a measure of the scalability of an algorithm with the size of the data to process. In this work, to estimate the weak scaling of the algorithm, we calculated the speed-ups associated to an amount of tasks to process that scales exactly with the number of CPUs used. In other words, if we obtain T_1 for the simulation time with 1 CPU on a grid of $N_x \times N_y \times N_z$, we calculated T_n on a grid of $n \times N_x \times N_y \times N_z$. An efficient weak scaling would give us a speed-up of 1 for all n. Our results are displayed in Fig. 9b. We can see that we quickly move away from a speed-up of one. We believe this is due to the additional overhead required to exchange larger chunks of data between the processes. This indicates that our algorithm is effectively limited by the speed of the data exchanges, rather than by the CPU speed.



(a) Strong scaling speed-ups. The workload is constant.



(b) Weak scaling speed-ups. The workload increases with the number of CPUs.

Figure 9: Strong and weak scaling speed-ups obtained with our algorithm. The benchmark was performed on the NIC5 supercomputer of the CECI. Each point is the mean result of 5 runs with the same amount of CPUs. The mean $T1$ was $211s$. The parameters used for the benchmark were a grid of $a = b = d = 0.25m$, $\Delta t = 10^-11s, t_{max} = 10^3\Delta t$, and $\Delta x = 0.001m$. The mode was the validation mode. We note that this benchmark was performed without taking into account the output of the data to Silo files.

6 Limits of the FDTD algorithm

We profiled our script and analysed the cache utilisation using Valgrind with a cubic simulation space of $0.05 \times 0.05 \times 0.05m^3$ and with $\Delta x = 0.001m, \Delta t = 0.001$, and $t_{max} = 100\Delta t$. The profiling was performed on a single core on the NIC5 clusters and the results can be observed in figures 13a and 13b in appendix C.

In order to calculate the CPU efficiency of the algorithm, we analyzed the `update_E_field` method. It took 34 seconds in total to perform the 101 updates. The `update_E_field` method contains 15 double precision operations that are performed $N_x * N_y * N_z = 1.25 * 10^5$ times. Therefore, in our case, $1.25 * 10^5 * 15 * 101 = 189.375 * 10^6$ operations are performed in total in 34s. Therefore, we can conclude that the CPU roughly performed $189.375 * 10^6/34 = 5,5$MFLOPS.

The provided benchmark of the EPYC 7542 CPU states that the processor is capable of 132,298 MFLOPS with its 64 threads which gives us an approximate capability of 2067 MFLOPS per thread [4]. This would imply that our algorithm is memory bound, as the CPU is capable of performing many more operations per second than our algorithm consumes.

Furthermore, the Valgrind analysis shows that the program heavily uses the cache. In fact, the `update_E_field` method performs approximately $6.750 * 10^6$ memory R/W operations in total. Even if the data is in cache, these operations require a lot of CPU cycles to be performed, which explains the difference in the computation intensity observed between this function and the benchmark of the CPU (5,5 MFLOPS vs 2067 MFLOPS, respectively).[1]

As previously stated, this algorithm is memory-bound. More specifically, its speed mostly depends on the latency of the memory/cache accesses, rather than on the memory bandwidth. In our case, the entire data could be held in the cache, so we did not really experience the DDR4 latency.

6.1 Expectations from a theoretical machine

A machine capable of 250 GB/s of memory bandwidth and 1.3 TFLOPS could generate similar results to ours. Indeed, we believe it is the latency of the memory access that effectively limits our algorithm efficiency, rather than the memory bandwidth itself. We can also calculate that a processor capable of performing 1.3TFLOPS would require $650 * 10^9$ double precision numbers being fed from memory per second to saturate its processing power[2]. This represents a required memory bandwidth of 1,04 TB/s (given that a double is 8 bytes) assuming that the R/W operations are instantaneous, which is not realistic. This far exceeds the capacities of the memory of hypothetical machine presented in the statement.

7 Conclusion

In this work, we described our 3D FDTD algorithm with OpenMPI. In the first section, we detailed both the sequential and parallel implementations. We then validated the algorithm by comparing our results with the analytical solutions, and verified that the absolute error was smaller than 0.1% We then computed the energies in the domain, and showed that the total calculated energy was equal to the theoretical one for the TE_{101} mode. Afterwards, we performed a stability analysis by comparing our results with the theoretical stability criteria to those obtained by A. Taflove in 1975. In section 4, we briefly described the implementation of the source as a waveguide port for the TE_{10} mode. Next, we carried out a scalibility analysis. Both the strong and weak scalings were limited by the bottleneck created by the field exchanges and by the larger workload associated to the process of rank 0. Finally, we roughly studied the limits of the algorithm by evaluating its CPU intensity, and concluded that the it was mostly bound by memory latency.

References

[1] Kane S. Yee, "Numerical Solution of Initial Boundary Value Problems Involving Maxwell's Equations in Isotropic Media ", IEEE Transactions on Antennas and Propagation (1966).

[2] A. Taflove and M. E. Brodwin, "Numerical Solution of Steady-State Electromagnetic Scattering Problems Using the Time-Dependent Maxwell's Equations," IEEE Transactions on Microwave Theory and Techniques (1975)

[3] D. Lee, T. Kim, et Q.-H. Park, "Performance analysis of parallelized PSTD-FDTD method for large-scale electromagnetic simulation", Computer Physics Communications, (2021).

[4] `https://www.cpubenchmark.net/cpu.php?cpu=AMD+EPYC+7542&id=3604`, visited on 21/12/2021.

[1]Even considering the fact that a double precision numbers can be worth 2 single precision numbers in terms of operation costs, 11 MFLOPS is still far from 2GFLOPS.

[2]This is indeed the case with our FDTD algorithm: all the operands come from memory

8 Appendice A: update equations

In this section, we present the 6 update equations for the **E** and **H** fields components.

$$\begin{aligned}
E_x^{n+1}(i+1/2,j,k) = E_x^n(i+1/2,j,k) \\
+ \frac{\Delta t}{\varepsilon \Delta y}\Big((H_z^{n+1/2}(i+1/2,j+1/2,k) - H_z^{n+1/2}(i+1/2,j-1/2,k)\Big) \\
- \frac{\Delta t}{\varepsilon \Delta z}\Big((H_y^{n+1/2}(i+1/2,j,k+1/2) - H_y^{n+1/2}(i+1/2,j,k-1/2)\Big)
\end{aligned} \tag{12}$$

$$\begin{aligned}
E_y^{n+1}(i,j+1/2,k) = E_y^n(i,j+1/2,k) \\
+ \frac{\Delta t}{\varepsilon \Delta z}\Big((H_x^{n+1/2}(i,j+1/2,k+1/2) - H_x^{n+1/2}(i,j+1/2,k-1/2)\Big) \\
- \frac{\Delta t}{\varepsilon \Delta x}\Big((H_z^{n+1/2}(i+1/2,j+1/2,k) - H_z^{n+1/2}(i-1/2,j+1/2,k)\Big)
\end{aligned} \tag{13}$$

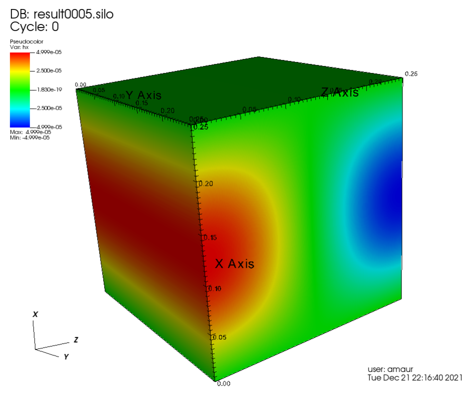
$$\begin{aligned}
E_z^{n+1}(i,j,k+1/2) = E_z^n(i,j,k+1/2) \\
+ \frac{\Delta t}{\varepsilon \Delta x}\Big((H_y^{n+1/2}(i+1/2,j,k+1/2) - H_y^{n+1/2}(i-1/2,j,k+1/2)\Big) \\
- \frac{\Delta t}{\varepsilon \Delta y}\Big((H_x^{n+1/2}(i,j+1/2,k+1/2) - H_x^{n+1/2}(i,j-1/2,k+1/2)\Big)
\end{aligned} \tag{14}$$

$$\begin{aligned}
H_x^{n+1/2}(i,j+1/2,k+1/2) = H_x^{n-1/2}(i,j+1/2,k+1/2) \\
+ \frac{\Delta t}{\mu \Delta z}\Big((E_y^n(i+1/2,j,k+1) - E_y^n(i,j+1/2,k)\Big) \\
- \frac{\Delta t}{\mu \Delta y}\Big((E_z^n(i,j+1,k+1/2) - E_z^n(i,j,k+1/2)\Big)
\end{aligned} \tag{15}$$

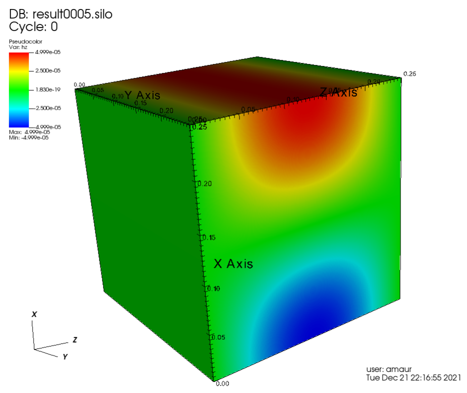
$$\begin{aligned}
H_y^{n+1/2}(i+1/2,j,k+1/2) = H_y^{n-1/2}(i+1/2,j,k+1/2) \\
+ \frac{\Delta t}{\mu \Delta x}\Big((E_z^n(i+1,j,k+1/2) - E_z^n(i,j,k+1/2)\Big) \\
- \frac{\Delta t}{\mu \Delta z}\Big((E_x^n(i+1/2,j,k+1) - E_x^n(i+1/2,j,k)\Big)
\end{aligned} \tag{16}$$

$$\begin{aligned}
H_z^{n+1/2}(i+1/2,j+1/2,k) = H_z^{n-1/2}(i+1/2,j+1/2,k) \\
+ \frac{\Delta t}{\mu \Delta y}\Big((E_x^n(i+1/2,j+1,k) - E_x^n(i+1/2,j,k)\Big) \\
- \frac{\Delta t}{\mu \Delta x}\Big((E_y^n(i+1,j+1/2,k) - E_y^n(i,j+1/2,k)\Big)
\end{aligned} \tag{17}$$

9 Appendice B: some images

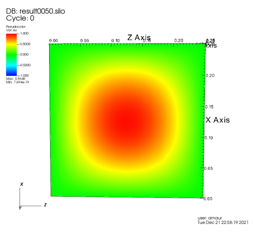


(a) H_x

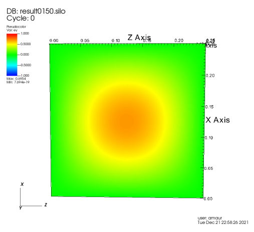


(b) H_z

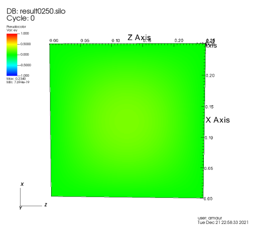
Figure 10: Pictures of the H_x and H_z fields with the simulation mode on ($t = \Delta t$) on a grid of $0.25 \times 0.25 \times 0.25 m^3, \Delta x = 0.001 m$. Their shape corresponds to the analytical form.



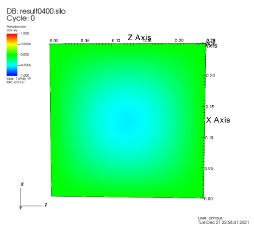
(a) $E_y(t = 0.5ns)$



(b) $E_y(t = 1.5ns)$

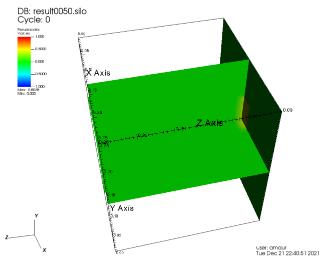


(c) $E_y(t = 2.5ns)$

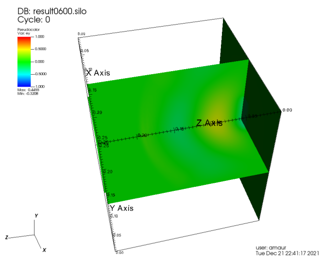


(d) $E_y(t = 4ns)$

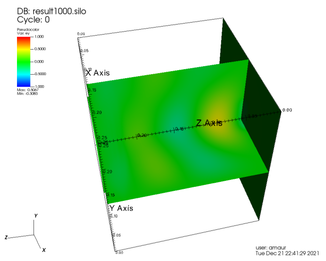
Figure 11: Oscillation in time of the E_y component on a grid of $0.25 \times 0.25 \times 0.25 m^3, \Delta x = 0.001 m$. The validation mode is on.



(a) $E_y(t = 0.5ns)$



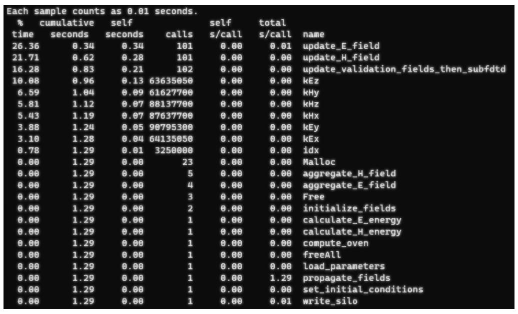
(b) $E_y(t = 6ns)$



(c) $E_y(t = 10ns)$

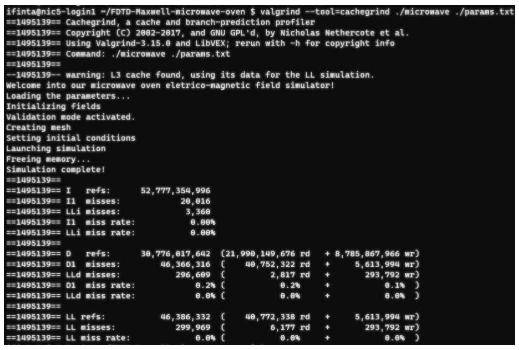
Figure 12: Propagation of the E_y field with a source on the $z = 0$ plane. We can see the wavefronts propagate away from the source.

10 Appendice C: Profiling and cache usage statistics



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
 26.36      0.34     0.34      101     0.00     0.01  update_E_field
 21.71      0.62     0.28      101     0.00     0.00  update_H_field
 16.28      0.83     0.21      102     0.00     0.00  update_validation_fields_then_subfdtd
 10.08      0.96     0.13 63635050     0.00     0.00  kEz
  6.59      1.04     0.09 61627700     0.00     0.00  kHy
  5.81      1.12     0.07 88137700     0.00     0.00  kHz
  5.43      1.19     0.07 87637700     0.00     0.00  kHx
  3.88      1.24     0.05 90795300     0.00     0.00  kEy
  3.10      1.28     0.04 64135050     0.00     0.00  kEx
  0.78      1.29     0.01  3250000     0.00     0.00  idx
  0.00      1.29     0.00       23     0.00     0.00  Malloc
  0.00      1.29     0.00        5     0.00     0.00  aggregate_H_field
  0.00      1.29     0.00        4     0.00     0.00  aggregate_E_field
  0.00      1.29     0.00        3     0.00     0.00  Free
  0.00      1.29     0.00        2     0.00     0.00  initialize_fields
  0.00      1.29     0.00        1     0.00     0.00  calculate_E_energy
  0.00      1.29     0.00        1     0.00     0.00  calculate_H_energy
  0.00      1.29     0.00        1     0.00     0.00  compute_oven
  0.00      1.29     0.00        1     0.00     0.00  freeAll
  0.00      1.29     0.00        1     0.00     0.00  load_parameters
  0.00      1.29     0.00        1     0.00     1.29  propagate_fields
  0.00      1.29     0.00        1     0.00     0.00  set_initial_conditions
  0.00      1.29     0.00        1     0.00     0.01  write_silo
```

(a) Profiling made on our program on a space of size described in question 7 on the NIC5.



```
ifinta@nic5-login1 ~/FDTD-Maxwell-microwave-oven $ valgrind --tool=cachegrind ./microwave ./params.txt
==1495139== Cachegrind, a cache and branch-prediction profiler
==1495139== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==1495139== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==1495139== Command: ./microwave ./params.txt
==1495139==
--1495139-- warning: L3 cache found, using its data for the LL simulation.
Welcome into our microwave oven eletrico-magnetic field simulator!
Loading the parameters...
Initializing fields
Validation mode activated.
Creating mesh
Setting initial conditions
Launching simulation
Freeing memory...
Simulation complete!
==1495139==
==1495139== I   refs:      52,777,354,996
==1495139== I1  misses:            20,016
==1495139== LLi misses:             3,360
==1495139== I1  miss rate:           0.00%
==1495139== LLi miss rate:           0.00%
==1495139==
==1495139== D   refs:      30,776,017,642  (21,990,149,676 rd   + 8,785,867,966 wr)
==1495139== D1  misses:        46,366,316  (    40,752,322 rd   +     5,613,994 wr)
==1495139== LLd misses:           296,609  (         2,817 rd   +       293,792 wr)
==1495139== D1  miss rate:            0.2% (           0.2%     +           0.1%  )
==1495139== LLd miss rate:            0.0% (           0.0%     +           0.0%  )
==1495139==
==1495139== LL refs:           46,386,332  (    40,772,338 rd   +     5,613,994 wr)
==1495139== LL misses:            299,969  (         6,177 rd   +       293,792 wr)
==1495139== LL miss rate:             0.0% (           0.0%     +           0.0%  )
```

(b) Valgrind cache usage of our algorithm statistics.