

## Goals of the Project

- Solve a physical problem modelled by partial differential equations using a finite difference scheme coded in C.
- Experiment with the stability of explicit time integration schemes.
- Parallelize the code on distributed memory systems using MPI, and on shared memory systems using OpenMP.
- Combine MPI and OpenMP to make the most of a supercomputing cluster.
- Explore the acceleration potential of modern GPUs using OpenMP.
- Learn to profile the code and run weak and strong scalability analyses.

## Project statement

The propagation of sound waves is governed by the following system of first order partial differential equations:

$$\frac{\partial P}{\partial t} = -\rho c^2 \operatorname{div} \mathbf{v}, \quad (1)$$

$$\frac{\partial \mathbf{v}}{\partial t} = -\frac{1}{\rho} \operatorname{grad} P, \quad (2)$$

where, in three dimensions and assuming a Cartesian coordinate system and MKS units,  $P = P(x, y, z, t)$  is the scalar pressure field (in Pa) and  $\mathbf{v} = \mathbf{v}(x, y, z, t)$  is the vector velocity (in m/s). Both the pressure and the velocity are function of the position  $(x, y, z)$  and of time  $t$ . The material parameters are the speed of sound  $c = c(x, y, z)$  (in m/s) and the density  $\rho = \rho(x, y, z)$  (in kg/m<sup>3</sup>), which vary in space but are independent of time.

Expanded in terms of the three components  $v_x, v_y, v_z$  of the velocity  $\mathbf{v}$ , system (1)–(2) can be rewritten as

$$\frac{\partial P}{\partial t} = -\rho c^2 \left( \frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} + \frac{\partial v_z}{\partial z} \right), \quad (3)$$

$$\frac{\partial v_x}{\partial t} = -\frac{1}{\rho} \frac{\partial P}{\partial x}, \quad (4)$$

$$\frac{\partial v_y}{\partial t} = -\frac{1}{\rho} \frac{\partial P}{\partial y}, \quad (5)$$

$$\frac{\partial v_z}{\partial t} = -\frac{1}{\rho} \frac{\partial P}{\partial z}. \quad (6)$$

In this project you will compute an approximate solution of this system of partial differential equations using the Finite Difference Time Domain (FDTD) method (see e.g. [1]). The FDTD method is based

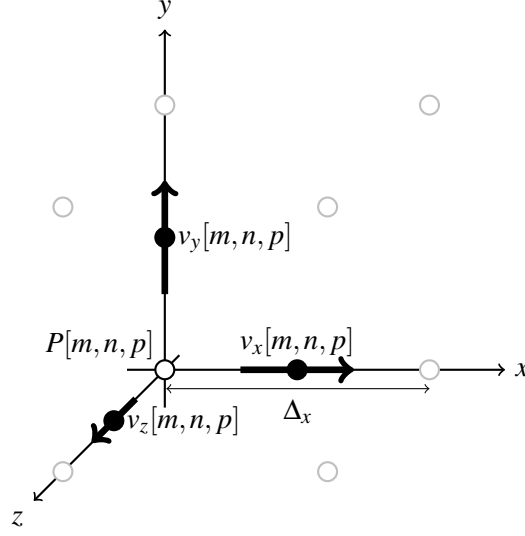


Figure 1: Spatial discretization in three dimensions: arrangement of velocity nodes relative to the pressure node with the same spatial indices  $[m, n, p]$  (the time index  $q$  is omitted for clarity).

on a discretization of the pressure and the components of the velocity in both space and time, on regularly spaced grid nodes. A pressure node is surrounded by velocity components such that the components are oriented along the line joining the component and the pressure node. The distance (offset) between pressure nodes along the  $x$ ,  $y$  and  $z$  directions is supposed to be constant, and equal respectively to  $\Delta_x$ ,  $\Delta_y$  and  $\Delta_z$ . In addition to these spatial offsets, the pressure nodes are assumed to be offset by half of a temporal step  $\Delta_t$  from the velocity nodes. Given four integer indices  $m$ ,  $n$ ,  $p$  and  $q$ , we introduce the following notations for the representation of the discretized pressure and velocity components (see Figure 1):

$$P^q[m, n, p] := P(m\Delta_x, n\Delta_y, p\Delta_z, q\Delta_t), \quad (7)$$

$$v_x^{q+1/2}[m, n, p] := v_x((m+1/2)\Delta_x, n\Delta_y, p\Delta_z, (q+1/2)\Delta_t), \quad (8)$$

$$v_y^{q+1/2}[m, n, p] := v_y(m\Delta_x, (n+1/2)\Delta_y, p\Delta_z, (q+1/2)\Delta_t), \quad (9)$$

$$v_z^{q+1/2}[m, n, p] := v_z(m\Delta_x, n\Delta_y, (p+1/2)\Delta_z, (q+1/2)\Delta_t). \quad (10)$$

For this project it will be assumed that the spatial grid offsets are equal in all directions, i.e.  $\Delta_x = \Delta_y = \Delta_z = \delta$ . Replacing the derivatives in (3) with finite differences and using the discretization (7)–(10) yields the following update equation for the pressure:

$$P^q[m, n, p] = P^{q-1}[m, n, p] - \rho c^2 \frac{\Delta_t}{\delta} \left( v_x^{q-1/2}[m, n, p] - v_x^{q-1/2}[m-1, n, p] + v_y^{q-1/2}[m, n, p] - v_y^{q-1/2}[m, n-1, p] + v_z^{q-1/2}[m, n, p] - v_z^{q-1/2}[m, n, p-1] \right). \quad (11)$$

The update equation for the  $x$  component of the velocity is obtained from the discretized version of

(4), which yields

$$v_x^{q+1/2}[m, n, p] = v_x^{q-1/2}[m, n, p] - \frac{1}{\rho} \frac{\Delta_t}{\delta} (P^q[m+1, n, p] - P^q[m, n, p]). \quad (12)$$

The update equations for the other two components are derived in a similar way:

$$v_y^{q+1/2}[m, n, p] = v_y^{q-1/2}[m, n, p] - \frac{1}{\rho} \frac{\Delta_t}{\delta} (P^q[m, n+1, p] - P^q[m, n, p]) \quad (13)$$

$$v_z^{q+1/2}[m, n, p] = v_z^{q-1/2}[m, n, p] - \frac{1}{\rho} \frac{\Delta_t}{\delta} (P^q[m, n, p+1] - P^q[m, n, p]). \quad (14)$$

In all your simulations, you can assume a homogeneous initial condition for both the pressure and the velocity, i.e.

$$P^0[m, n, p] = v_x^{1/2}[m, n, p] = v_y^{1/2}[m, n, p] = v_z^{1/2}[m, n, p] = 0, \quad \forall m, n, p, \quad (15)$$

and impose a Dirichlet condition on selected pressure nodes to impose a sound source:

$$P^q[m, n, p] = P_{\text{source}}^q[m, n, p], \quad m, n, p \in S, \quad (16)$$

where  $S$  denotes the subset of nodes where the sound source  $P_{\text{source}}^q[m, n, p]$  is imposed. For example, if there are  $N_x$ ,  $N_y$  and  $N_z$  nodes respectively along  $x$ ,  $y$  and  $z$ , a point-wise sinusoidal sound source at frequency  $f$  can be imposed in the middle of the domain by choosing

$$P_{\text{source}}^q[\lfloor N_x/2 \rfloor, \lfloor N_y/2 \rfloor, \lfloor N_z/2 \rfloor] = \sin(2\pi f q \Delta_t). \quad (17)$$

On the boundary of the domain, one assumes homogeneous Neumann boundary conditions for the pressure (where any node “outside” the domain by one offset is given the same value as the one associated to the closest node on the boundary, e.g.  $P^q[-1, n, p] := P^q[0, n, p]$  or  $P^q[m, N_y, p] := P^q[m, N_y - 1, p]$ ) and zero normal velocity.

## Instructions




A serial C code that solves the discretized equations in three dimensions over a rectangular cuboid is available on the NIC5 cluster: see <https://people.montefiore.uliege.be/geuzaine/INFO0939/project.html#start>.

You are asked to:

1. Improve the evaluation of the local speed of sound and density. In the serial code the evaluation is done using a nearest-neighbor search (cf. the `interpolate_inputmaps` function). You should improve this by using trilinear interpolation.
2. Experimentally study the stability of the scheme depending on the ratio  $c\Delta_t/\delta$ , for a homogeneous medium with a constant speed of sound.
3. Perform a rough analysis of the arithmetic intensity of the FDTD and determine the main limiting factor for the speed of your algorithm: is the code memory bound or CPU bound? What do you expect if you run the code on a machine capable of 200 GB/s of memory bandwidth and 2.8 TFLOPS/s of processing power?

4. Evaluate potential bottlenecks in the serial code, based on the lecture on CPU cache hierarchies.
5. Parallelize the serial code using MPI, by subdividing the domain into  $P_x \times P_y \times P_z$  partitions, where the positive integers  $P_x$ ,  $P_y$  and  $P_z$  designate the number of partitions along  $x$ ,  $y$  and  $z$ , respectively, and are chosen to minimize the amount of MPI communications.
6. Parallelize the serial code using OpenMP. Could you advantageously make use of the collapse clause?
7. Integrate the OpenMP parallelization strategy into the MPI code to produce a hybrid MPI+OpenMP code that can efficiently target the cluster architecture of the NIC5 supercomputer.
8. Perform a scalability analysis of the MPI, OpenMP and MPI+OpenMP codes, by evaluating both the strong and the weak scaling on one or more well-chosen example(s). Using the tools presented during the lectures (Score-P, Scalasca, Cube, likwid), explain the results of the scalability study.
9. Accelerate the serial code using OpenMP on a single GPU of the Lucia supercomputer. Compare and analyze the performance of the OpenMP CPU and GPU codes.
10. Imagine a few combinations of input parameters, speed and density maps and sources to study the propagation of sound in interesting configurations. It is possible to propagate audio sources and evaluate the sound at different locations (e.g. across a room, or behind a wall): see <https://people.montefiore.uliege.be/geuzaine/INF00939/project.html#audio>.

Here are some optional enhancements that you can work on if you would like to go further:

-  Perform a theoretical study of the stability of the FDTD scheme.
-  Implement more sophisticated boundary conditions on the boundary of the domain to simulate a transparent boundary, allowing waves to exit the simulation domain without (too many) reflections.
-  Combine MPI and OpenMP for multi-GPU parallelization.

### ASCII parameter file format

The input parameter file is an ASCII text file that contains the simulation parameters, structured as follows:

```
delta
delta_t
max_t
sampling_rate
input_speed_filename
input_density_filename
source_specification
```

```

output_specification_1
output_specification_2
...

```

with

- `delta` (double): spatial grid offset  $\delta$ ;
- `delta_t` (double): time step  $\Delta_t$ ;
- `max_t` (double): maximum simulation time; the maximum number of time steps is  $\lfloor \text{max\_t} / \Delta_t \rfloor$ ;
- `sampling_rate` (int): sampling rate at which output files should be created (0: never save, 1: save all steps, 2: save 1 step out of 2, etc.);
- `input_speed_filename` (string): name of the input file containing the sound speed profile (the file format is described in “Binary data file format” below);
- `input_density_filename` (string): same as `input_speed_filename` but for the density profile;
- `source_specification`: specification of the source. The source can be a sine wave or an audio file. A sine wave specification format is `sine freq posx posy posz`. For example, for a sine source of 3400 Hz placed at  $x, y, z = 0.5$ , the source specification will be `sine 3400 0.5 0.5 0.5`. For an audio source, the specification format is `audio filename.dat posx posy posz`. For example, if the name of the file for the audio source is `in_audio.dat` and that the source is located at  $x = 0.1, y = 0.2$  and  $z = 0.3$  the source specification will be `audio in_audio.dat 0.1 0.2 0.3`.
- `output_specification`: one or more output specifications. The general format is `type output_source filename coords`. Where
  - `type`: a keyword specifying the type of output to produce. `type` keyword can be `all` to write all the data or `cut_x` to write data extracted from cut along the  $x$  axis or `cut_y/cut_z` for a cut along the  $y$  or  $z$  axes. If you want to extract a single point, you can use the `point` keyword.
  - `output_source`: a keyword specifying the data to use to produce the output. `output_source` can be `pressure`, `velocity_x`, `velocity_y` or `velocity_z`.
  - `filename`: the name of the file to which the output should be written
  - `coords`: one or three space separated doubles that specify from where the cut or the point extraction should be performed. For a point three values are needed while a cut only requires one value. If `type` is `all`, no coordinates are needed.

Example output specifications:

- `all pressure out_p.dat`: write the full pressure data to a file with name `out_p.dat`.
- `cut_z velocity_x out_vx_cutz_0_20.dat 0.20`: write a cut of  $v_x$  at  $z = 0.20$  to a file with name `out_vx_cutz_0_20.dat`.

- `cut_x velocity_y out_vy_cutx_0_40.dat 0.40`: write a cut of  $v_y$  at  $x = 0.40$  to a file with name `out_vy_cutx_0_40.dat`.
- `cut_y velocity_z out_vz_cuty_0_60.dat 0.60`: write a cut of  $v_z$  at  $y = 0.60$  to a file with name `out_vz_cuty_0_60.dat`.
- `point pressure out_p_point_0_40.dat 0.40 0.40 0.40`: extract a point at  $x, y, z = 0.40$  out of the pressure data and write it to a file with name `out_p_point_0_40.dat`.

Sample parameter files will be provided in class.

### Binary data file format

The input and output binary files are structured as follows:

`nx ny nz xmin xmax ymin ymax zmin zmax timesteps...`

with

- `nx (int)`: number of nodes along  $x$ ;
- `ny (int)`: number of nodes along  $y$ ;
- `nz (int)`: number of nodes along  $z$ ;
- `xmin (double)`: minimum  $x$  coordinate of the domain;
- `xmax (double)`: maximum  $x$  coordinate of the domain;
- `ymin (double)`: minimum  $y$  coordinate of the domain;
- `ymax (double)`: maximum  $y$  coordinate of the domain;
- `zmin (double)`: minimum  $z$  coordinate of the domain;
- `zmax (double)`: maximum  $z$  coordinate of the domain;
- one or multiple time step data. One time step is written as one int (step index), followed by one double (step time), followed by  $nx \times ny \times nz$  doubles. The values are assumed to be given “by row”, i.e.

```
for(int p = 0; p < nz; p++)
  for(int n = 0; n < ny; n++)
    for(int m = 0; m < nx; m++)
      printf("value[%d][%d][%d] = %g\n", m, n, p,
            value[ny * nx * p + nx * n + m]);
    }
  }
}
```

## General remarks on C coding style

Your coding style will be evaluated. As such, some quality in the submitted code is expected, and you should strive to have a clean and neat coding style. In general, think about who will read your code and ask yourself if it is clear and understandable. Please beware of some common mistakes that will lead to penalties on your final grade:

- Do all your computations in `double`. There is no reason to use single precision in this project.
- If you `malloc()` something, remember to `free()` it.
- Check return values for failure, especially from `malloc()`, `fopen()` or similar calls. Don't assume that those calls will always succeed.
- Don't use Variable Length Arrays, i.e.

```
int function(int N) {  
    int array[N];  
}
```

The value of `N` must be known at compile time. If it is not the case, use `malloc()`.

- Don't abuse comments. If something is obvious, don't comment it. If you feel the need to comment some code, ask yourself if perhaps you can write the code in a clearer way instead of putting that comment.
- Try to use relevant and appropriate variable and function names. Be coherent in naming things, in order to make it easy to track down where data goes.
- Avoid global variables. There are very few valid use cases for global variables and they are one of the factors that make your code not thread safe.
- Finally, a function *should* fit on your screen. If it doesn't, perhaps you should split it in smaller pieces. Also, if you have a function that takes more than 10 parameters, perhaps you need to think if you really need all of them, or it might be time to create a `struct`?

## References

[1] John B. Schneider, *Understanding the Finite-Difference Time-Domain Method*, 2010. Available online at: <http://www.eecs.wsu.edu/~schneidj/ufdtd/>.