High Performance Computing

Master degree in physical engineering

Project 2

Authors:
Chloé Halbach
Quentin Elias

Supervisors:
Christophe Geuzaine
Anthony Royer

December 23, 2018

Contents

1 Introduction

The goal of this project is to numerically solve differential equations in a three dimensional environment, using two different schemes: Euler explicit and Euler implicit. The former is easy to implement but is not always stable, while the latter is always stable but is more complicated. For both schemes, MPI and OpenMP have been used to run the program in parallel.

2 Implementation

The program is split into four files: *main.c*, *Explicit.c*, *Implicit.c*, and *Functions.c*. While the first file contains the main instructions, the second and third files define the functions specifically used in the explicit and implicit scheme respectively, and the last file contains the functions used in both numerical schemes.

First, the program reads a file of parameters and an integer corresponding to the scheme type (0 for explicit and 1 for implicit), both provided by the user. This is done before the parallel region because this operation does not require any communication between the processes. Moreover, all these values have to be directly available to each process. Then, the parallel region is initialized.

The 3D domain is split between all the processes in an optimized way so that if the dimension of the cube is not divisible by the number of processes, the difference between the most loaded and the less loaded process is minimal. The function *partition* splits the domain in several slices of width *z_chunk* perpendicular to the z-axis. Each process stores its own value of *z_chunk*. Once each process has created its own part of the domain, all the sub-domains are initialized at 0, then the initial condition(s) can be inserted at the right place. The output file at t=0 is written before starting the explicit or implicit algorithm.

2.1 Euler explicit

At the start of each time step n, each process has to send and receive the edges of its domain to the other adjacent domains, since they require these data to calculate the value of the edges of their own domain. To avoid any deadlock, the function *send_receive* splits all the sub-domains in two groups, based on their parity. When one group sends its edge values to the next one, the other group receives them. This operation is carried out for each edge and for each group of processes. The values of the slice coming from the previous process are stored in the first half of *z_slice*, while the values of the slice coming from the next process are stored in the second half. The values of the first (resp. second) half of *z_slice* on the first (resp. last) process are 0.

After that, each process calculates the value for each coordinate of its domain. At each time step, the function *boundary* checks if the ink has reached a threshold value at the borders of the domain. If this is the case, the simulation is stopped after writing the data of the last step. One the other hand, if the ink has not yet touched the border and if n is a multiple of S, each process simultaneously writes its values, in the correct order, in a common file.

2.1.1 Addition of openMP

In the explicit scheme, the implementation of openMP is quite straightforward: each *for* loop in the *boundary* function and the *for* loop of the y axis (j loop) in the main environment are parallelized. When several *for* loops are nested, only one loop is parallelized so as to not spread the work into too small chunks.

2.2 Euler implicit

For the implicit scheme, we first need to compute the A sparse matrix. Its shape is known in advance and therefore, one can deduce the total number of non-zero elements Nz, along with their values, from the dimension of the matrix and the Euler implicit formula. In our program, each process has full access to the whole matrix stored in the i, j, and v vectors of size Nz (respectively: *row, column* and *value*). The function

get_ijv fills those vectors with respectively the index of the row of a non-zero value, the index of the column of this non-zero value, and then its value. Next, each process computes the number of non-zero values in its domain in the *array_Nz* of size *np*. This array is shared among all processes and will later be used in the *matrixvector* function.

Before starting the algorithm, all the required arrays are declared with appropriate size, followed by the scalars. Then the time loop (n loop) can start. At each step, the arrays are initialized before the *while* loop. The communication is carried out at the start of each *while* iteration. The *send_receive* function is the same as the one in the explicit scheme. Next, all the calculus operations are carried out. The *vectorSum* operation can be done locally on each process. The *vectorVector* product can also be done locally, but the resulting scalar is obtained through *MPI_Allreduce* and thus communicated to all processes. The *matrixvector* function requires elements of the array *p_0* from outside each process domain and thus requires the slices of elements obtained through communication. Each process can find the element of *value* and its position thanks to a counter, whose initial value depends on the sum of the number of non-zero elements on all processes before that one. The *column* value of the considered non-zero value determines if the corresponding matrix element is in the domain or not. If not, the element is accessed through the *z_slice* array.

As in the explicit scheme, the boundaries are checked. If the value at a point of the boundary is greater than a given epsilon, the program exits the n loop and writes the data of the last step. If this is not the case, the data are saved if n is a multiple of S.

2.2.1 Addition of openMP

The *boundary* function is the same as in the explicit scheme and so the parallelization is done in the same way. In this scheme, there are no loops over the spatial coordinates *x, y* and *z*. Instead, a linear system has to be solved with the conjugate-gradient method, which implies the use of matrix and vector products, vector products and vector sums. While the vector sum can be parallelized without difficulty on each process, the vector product requires a sum reduction operation for the temporary sums of each thread, on each process. Unfortunately, our group did not manage to parallelize the *matrixvector* function with openMP and so the related scalings are expected to be less efficient, since the matrix vector is an important element of the implicit method.

3 Scalability analysis

In this section, the scalability analysis of both the explicit and the implicit schemes will be conducted, with and without openMP. A perfect parallel program would scale linearly: this means that the total execution time is reduced by the number of processes used. A 100% linear parallel program is impossible to obtain due to the communication required between the different processes which implies that many more operations need to be executed by the program. The goal is of course to get as close as possible to a linear scaling. In order to do so, the communication must be kept to a minimum and the workload must be well balanced. Two types of scaling will be considered: the weak scaling and the strong scaling. [1]

Weak scaling consists in increasing the workload, for example solving a bigger problem, while increasing the number of processes. In this case, the workload remains constant for each process: if we consider using two processes, we double the workload that there was on a single process, and spread it evenly. In the case of weak scaling, linear scaling is achieved if the run time stays constant while the workload is increased in direct proportion to the number of processors. The weak scaling efficiency is given, in percentage of linearity, by the following formula:

$$\frac{t_1}{t_N} \times 100\%,$$

[1] Definitions from `https://www.sharcnet.ca/help/index.php/Measuring_Parallel_Scaling_Performance` [Page consulted on 12-12-2018]

where t_1 is the time taken to complete a work unit with one processing element and t_N is the amount of time to complete N of the same work units with N processing elements. Weak scaling is interesting for someone who wants to solve bigger problems which use more memory than the one available on a single node.

Strong scaling consists in keeping a constant workload while increasing the number of processing units. If the amount of time to complete a work unit with one processing element is t_1, and the amount of time to complete the same unit of work with N processing elements is t_N, the strong scaling efficiency, in percentage of linearity, is given by the following formula:

$$\frac{t_1}{N \times t_N} \times 100\%.$$

In this case, a program is considered to scale linearly if the speedup is equal to the number of processing elements used.

For the purpose of the scaling analysis, the program will execute the algorithm for all the time steps even though the boundary has been reached. In practice, this means that an *epsilon* greater than one has been chosen for the function boundary. This way, the timings should be more accurate while the density of work for each process remains the same.

3.1 Explicit scheme

3.1.1 Stability

The explicit Euler scheme is only stable for certain combinations of parameters. Specifically, the stability depends on the time step m, the diffusion coefficient D and the spatial step h. As can be seen in Figure 1 (on page 5), when the combination of parameters is unstable, concentrations oscillate between positive and negative values at each time step. Furthermore, these values diverge very quickly instead of converging towards zero.

For a fixed spatial step $h = 0.1$ and a fixed diffusion coefficient $D = 0.5$ the scheme is stable for a time step $m \leq 0.0025$, whereas for a fixed spatial step $h = 0.1$ and a fixed time step $m = 0.0025$ the scheme is stable for a diffusion coefficient $D \leq 0.5$. In conclusion, for a fixed spatial step $h = 0.1$ the explicit program is stable for

$$mD \leq 0.00125. \tag{1}$$

Moreover, if the time step m and the diffusion coefficient D satisfy Equation 1, the explicit Euler scheme is only stable for a spatial step $h \geq 0.1$.

3.1.2 Weak scaling without openMP

For the weak scaling, the goal is to increase the workload in the same proportions as the number of processes. In this project, the number of elements to consider for each process is one part of the 3-dimension cubic space. Since we consider 3 dimensions, the number of elements scales with the power 3 of the dimension of the cube (i.e. $\frac{L}{h} + 1$). This means that in order to analyze the weak scalability of the program, one cannot increase the dimensions in an arbitrary way. Firstly, the fraction $\frac{L}{h}$ must remain an integer because of the discretization of the domain. Secondly, the total number of elements to be processed (i.e. $(\frac{L}{h} + 1)^3$) must be divisible by the number of threads used, since the number of elements per process must be an integer too. Those constraints restrain the number of processes that one is allowed to use if one wants to limit the error. They might also induce a small error in the timings used in the weak scaling efficiency analysis and this should be accounted for. Finally, to perform the weak scalability analysis, the number of elements on each process must remain constant. Therefore, a fixed value of 64,000 elements per process is chosen arbitrarily. This number should be suitable for a scalability analysis while maintaining a reasonable execution time.

Furthermore, it should be taken into account that the explicit scheme is not stable for any set of parameters. As stated above, for a fixed value h=0.1, the product mD must be inferior or equal to 0.00125. To comply

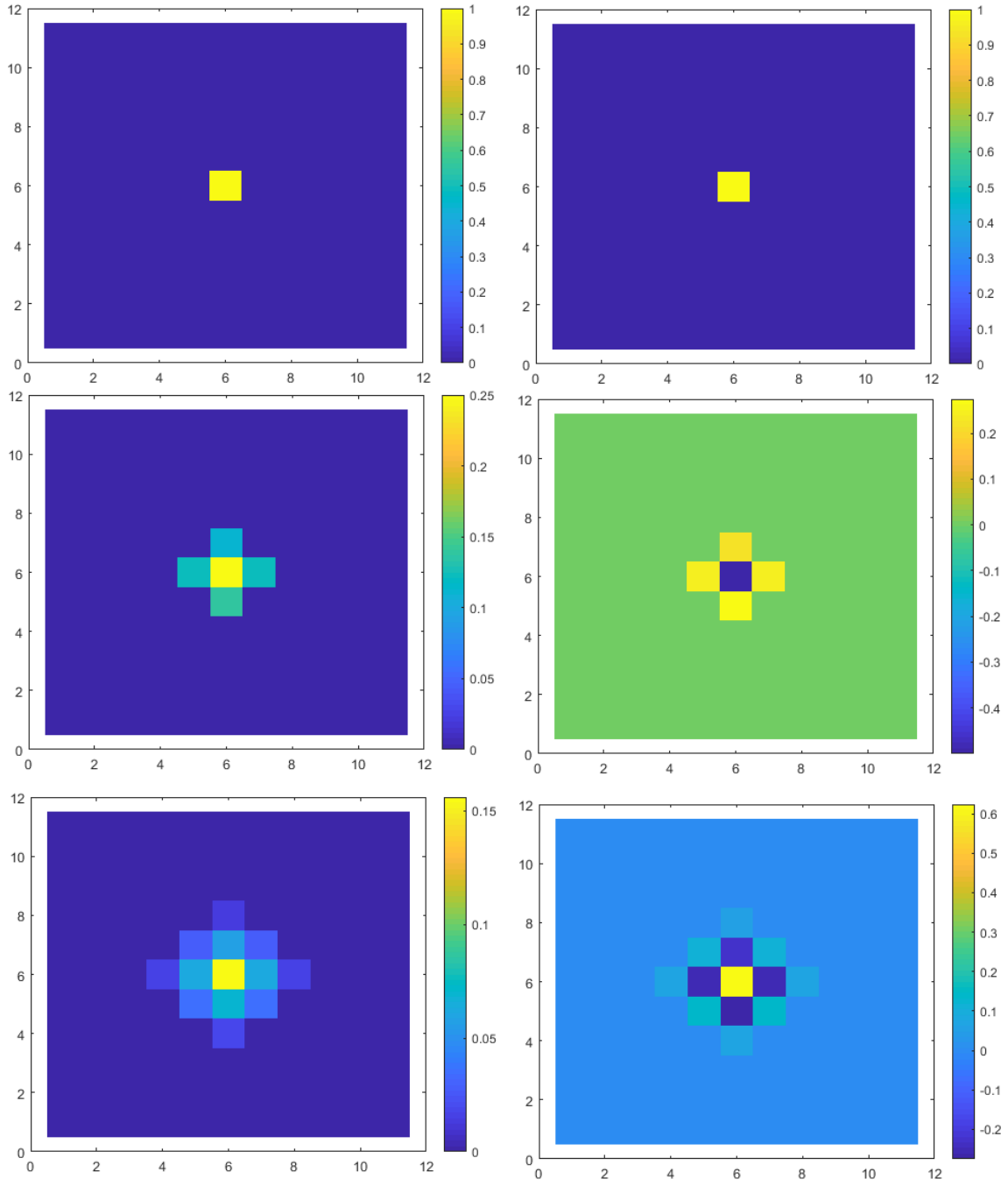

Figure 1: While the graphs on the left hand side correspond to stable parameters, the graphs on the right hand side result from unstable parameters. The first row is taken at the first time step, the second at the second time step and the third at the third time step.

with this constraint, both m, D and h will remain constant, and L will vary with the number of processes. In fact, all parameters but L will remain constant and are chosen as follows:

$$h = 0.1, v_x = 0, v_y = -1, v_z = 0, D = 0.05, T_{max} = 1, m = 0.001, S = 20.$$

L will vary along the number of processes in a way that the number of elements per process is as close to 64,000 as possible, while maintaining a number of elements that is an integer. The exact number of elements is taken into account to better approximate the efficiency. The values recorded for the timings are the average of 5 different runs. The pertinent data is reported in Table 1 while the efficiency and timings are plotted in Figure 2. The communication time is the total time taken by the function *send* *receive*, corresponding to the communication between all the processes.

np	**L**	**Elements per process**	**Communication time (s)**	**Main time (s)**	**Efficiency**
1	3.9	64000	0	6.004	100%
2	4.9	62500	0.02	6.134	95.5%
3	5.6	61731	0.048	6.4	90.37%
4	6.3	65536	0.05	6.751	91.03%
6	7.4	62208	0.061	6.62	87.48%
7	7.6	65219	0.07	7.094	86.21%
8	7.9	64000	0.075	7.033	85.3%
9	8.3	65856	0.08	7.716	80.05%
11	8.7	61952	0.08	7.45	77.92%
12	8.9	60750	0.08	7.605	74.89%

Table 1: Data of the weak scaling analysis for the explicit scheme

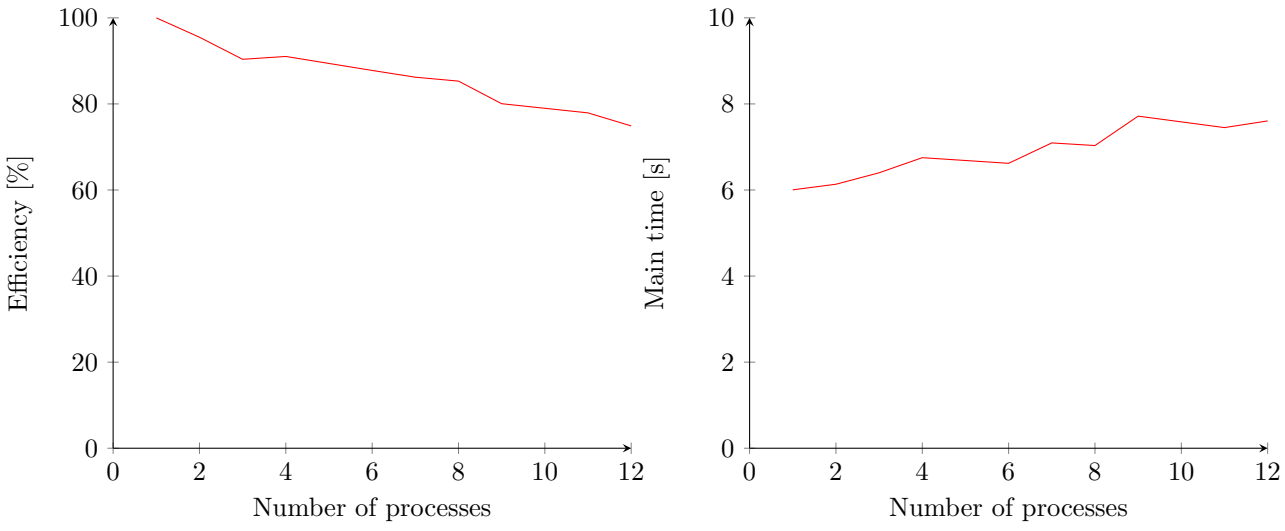

Figure 2: Weak scaling of the explicit scheme with an increasing number of processes

As shown in Figure 2, the efficiency decreases slowly as the number of processes increases, while maintaining an overall good weak scaling efficiency. As shown in Table 1, the communication time is small compared to the total execution time.

3.1.3 Strong scaling without openMP

Another goal of parallel programming is to execute a program faster. In science, many problems need to be simulated on computer and these can quickly take hours or days. Fortunately, parallel programming can drastically reduce the simulation time. When running a program on several processes, the goal is to get a strong scaling efficiency which is as close as possible to linear. In this case, the problem size remain constant as the number of processes is increased. Ideally, the total execution time should be divided by the number of processes used.

The strong scaling analysis will be conducted on the set of parameters *tiny, small, medium* and *big* and a number of processes increasing in power of two: 1, 2, 4, 8, 16 and 32. The measurements are taken as an average of 10 separate runs. Some tests were needed to be run several times because of extreme variations in the execution time, probably due to a heavy solicitation of the cluster. In that case, the most stable set of 10 runs was recorded while the others were discarded.

For the set of parameters *tiny*, the number of processes could only be increased up to 8. Since the problem is a cube of dimension 11 and our program divides the cubic domain in slices, the number of processes cannot be greater than the number of slices. For the same reasons, the number of processes for the set of parameters *small* cannot be higher than 21. This problem could be solved by dividing the domain in smaller entities, like cubes for example. However, this would imply a lot of unnecessary additional programming, since this problem only occurs for *tiny* and *small*, which are extremely quick to solve and are only there for testing purpose. For bigger simulations, for which parallel programming is actually useful, this problem does not appear.

Another issue with those sets of parameters is a great instability in the execution time, sometimes taking a short time, which is expected for those parameters, and sometimes shooting up to 45 seconds for unexplained reasons. Such instabilities were recorded for the following program arguments: *tiny* with np=4 and np=8; *small* with np= 4 and np=16. In these cases, the best results were kept in order to better represent the scaling. This liberty was taken because the *tiny* and *small* sets of parameters are less representative of the parallel program scaling, since they would never be parallelized in practice, in contrast with a bigger problem such as *medium* or *big*.

The data resulting from the analysis is reported in Table 2 (on page 8) and the speedup is plotted in Figure 3 .

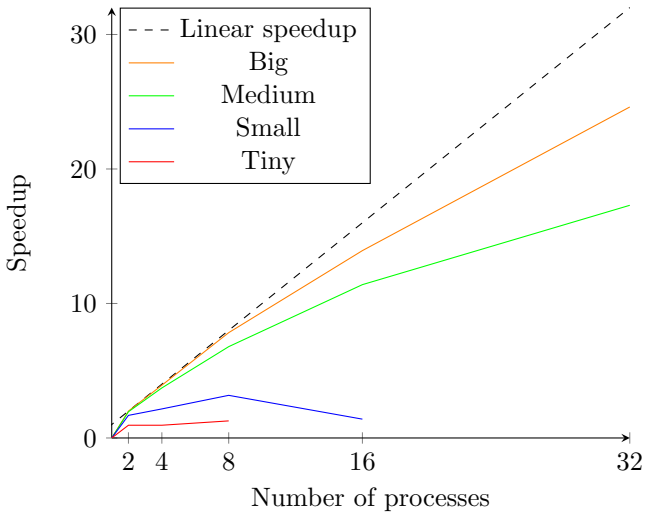

Figure 3: Speedup of the explicit program for the different set of parameters and increasing number of processes

.

Parameters	np	Communication time (s)	Main time (s)	Speedup	Efficiency
Tiny	1	0	0.18	0	100%
Tiny	2	0.0047	0.19	0.947	47.3%
Tiny	4	0.0199	0.19	0.947	23%
Tiny	8	0.006	0.141	1.27	15.9%
Small	1	0	1.069	0	100%
Small	2	0.015	0.635	1.68	84.1%
Small	4	0.026	0.494	2.16	54%
Small	8	0.019	0.337	3.17	39.6%
Small	16	0.08	0.76	1.4	8.7%
Medium	1	0	79.24	0	100%
Medium	2	0.16	40.56	1.95	97.8%
Medium	4	0.4	21.33	3.72	92.8%
Medium	8	0.29	11.67	6.79	79%
Medium	16	0.28	6.95	11.4	71%
Medium	32	0.29	4.58	17.3	54%
Big	1	0	196.68 min	0	100%
Big	2	8.12	98.18 min	2	100%
Big	4	15.81	50.58 min	3.92	97.2%
Big	8	14.31	25.1 min	7.83	97.9%
Big	16	27.37	14.12 min	13.93	87%
Big	32	14.03	7.99 min	24.61	76.9%

Table 2: Data of the strong scaling analysis of the explicit scheme for all pre-established sets of parameters.

It is worth noticing that the sets *tiny* and *small* scale very badly. The execution time even increases for the *tiny* problem. As stated above, those results were highly unstable and are not as pertinent for a parallel programming scaling analysis as *medium* and *big* are. As a matter of fact, the sets of parameters *medium* and *big* scale quite well. The linear efficiency for the *medium* problem only drops below 70% for more than 16 processes. The *big* problem scales even better: the linear efficiency does not drop below 75%, even for 32 processes. It has to be noted that the values obtained with 2 processes gave a perfect linear efficiency, which is quite surprising. Since the efficiency should not reach 100%, more test runs could be made to have a better accuracy. In conclusion, one can say that the goal of parallel programming to reduce the execution time was achieved.

For any problem size, the communication time is small in comparison to the total execution time. However, for larger problems, the additional number of instructions to execute due to communication, such as accessing the slices of elements received from other processes, is small in comparison to the total number of instructions to realize on the local process memory. Therefore, it is understandable that the bigger the problem, the better the scaling. Finally, if the workload is spread evenly and the communication kept to a minimum, the scaling should be close to linear.

3.1.4 Weak scaling with openMP

In this section, the effect of openMP on the execution time will be analyzed. This analysis will be conducted on different number of processes: 1, 2, 4, 8 and 12. Each time, the number of threads will vary as follows: 2, 4, 8 and 16. The parameters used will be the same as the one used for the weak scaling analysis, with L varying in accordance with the number of processes. The timings reported in Table 1 (on page 6) will be used as a reference for a single thread. First, the different schedules for the openMP parallel loops will be tested to determine which one gives the best performances. The best one will be used in all further testings. The speedup curves for the different schedules are plotted in Figure 4 (Page 9).

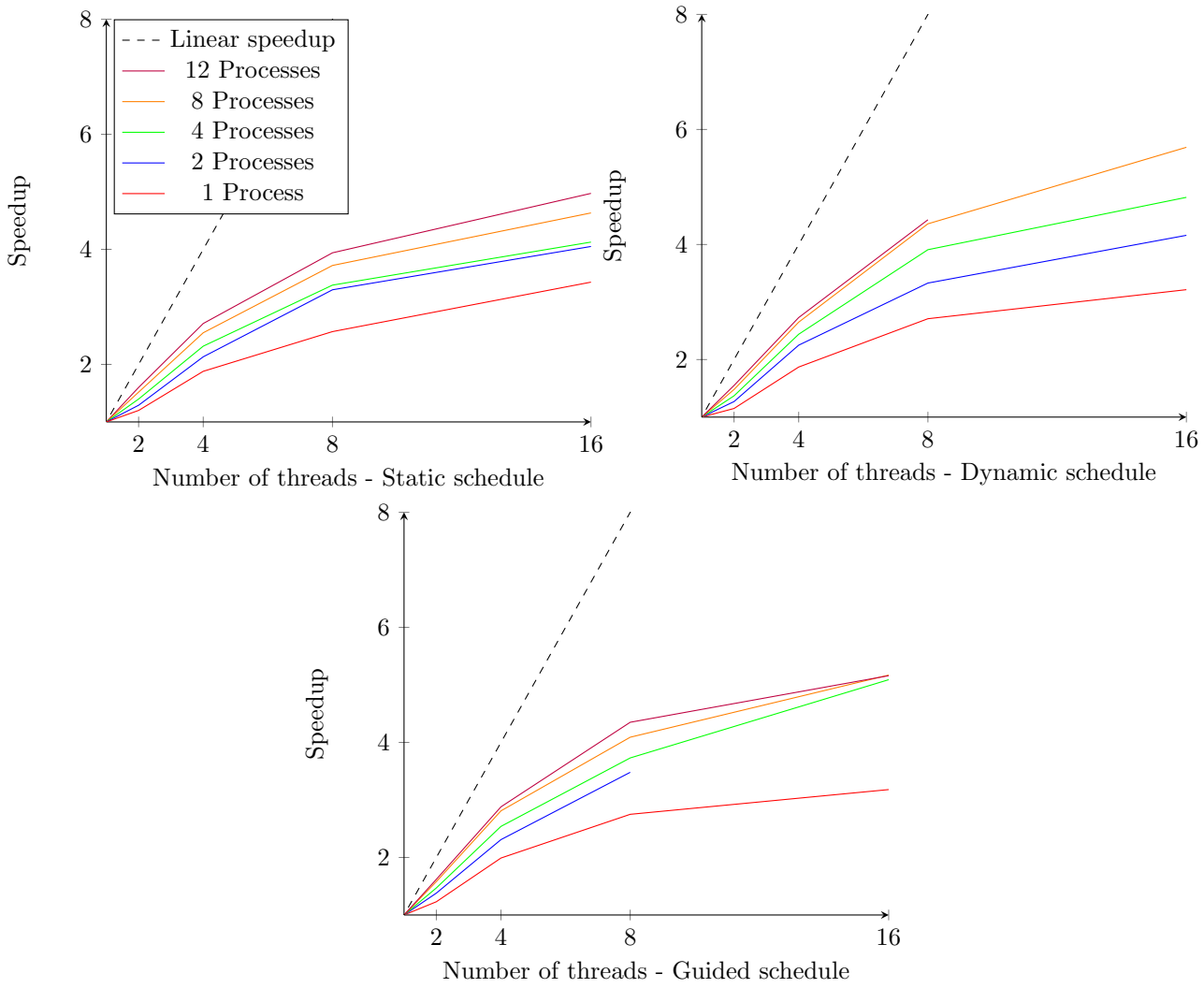

Figure 4: Speedup scaling of the explicit program for an increasing number of threads, for different schedules and an increasing number of processes

.

The figure shows that the speedup induced by openMP is a lot less effective than the one by MPI. The former is far from linear although the speedup is noticeable. This could be explained by the fact that only a few regions could be parallelized with openMP. Moreover, the parallelized *for* loops are called at every time step; consequently, the parallel regions are initialized and closed frequently, and in turn reduce the efficiency.

One notices that the speedup is slightly better for a larger number of processes. This increase is hard to explain since the workload is the same for any number of processes. However, the difference of speedup is quite small, and this is consistent with what is expected for weak scaling analysis: the number of elements to compute for each process being the same, the speedup in function of the increasing number of threads should be the same too.

The best speedups are given by the guided schedule. Indeed, the number of elements per process is not always divisible by the number of threads, and thus a static scheme is not always appropriate. Also, the guided schedule seems to always give better results than the dynamic one. As a consequence, the guided schedule will be used for all further testings including openMP. It should be noted that some results were unstable (dynamic: np=12 and 16 threads; guided: np=2 and 16 threads) and were not reported in the graphs.

3.1.5 Strong scaling with openMP

This time, only the *medium* and *big* problems will be considered since they are the most relevant. Both the number of processes and the number of threads will increase as a power of 2 up to 16. The data of the analysis is reported in Table 3 and 4 and the speedups are plotted in Figure 5. One notices that this time the speedup is better for fewer processes. However, for the *medium* parameters, the speedups are not very stable and even decrease for 4 and 8 processes. Further observation shows that, as expected, the efficiency is a lot better for the *big* problem, since the parallelization works better when the scale of the simulation is larger. Again, some results were unstable and were not plotted in the graphs (*medium*: np=16 and 16 threads; *big*: np=8 and 16 threads).

np\threads	**1**	**2**	**4**	**8**	**16**
1	79.24	44.33	25.3	16.33	13.54
2	40.56	23.03	13.2	8.57	7.16
4	21.33	12.57	7.47	5.02	4.47
8	11.67	7.211	4.401	3.12	3.03
16	6.95	4.02	2.64	2.09	unstable

Table 3: Total execution time (in seconds) for *medium* parameters and an increasing number of threads and processes

np\threads	**1**	**2**	**4**	**8**	**16**
1	196.68	108.5	61.7	30.68	21.9
2	98.18	54.35	29.5	17.11	11.3
4	50.58	27.9	15.3	8.86	6.25
8	25.1	14.1	7.83	4.58	unstable
16	14.12	7.83	4.45	2.69	2.12

Table 4: Total execution time (in minutes) for *big* parameters and an increasing number of threads and processes

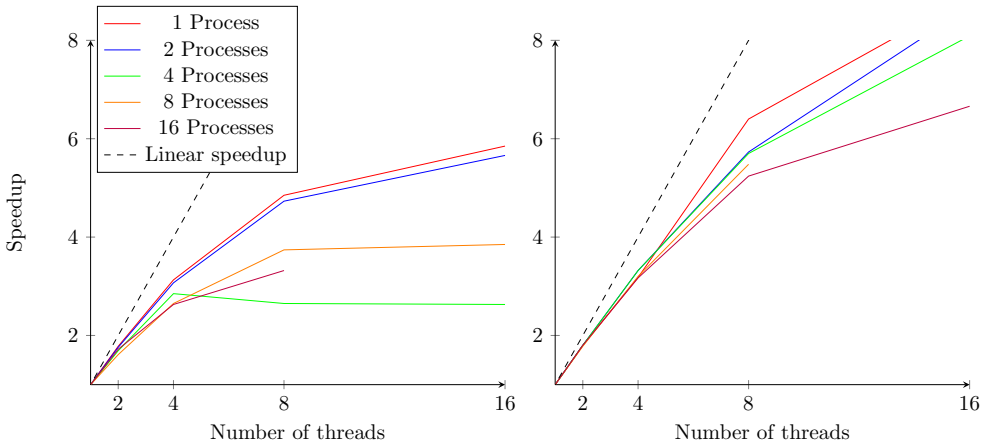

Figure 5: Speedup of the strong scaling analysis for an increasing number of threads and increasing number of processes, for the *medium* (left) and *big* (right) parameters

.

3.2 Implicit scheme

3.2.1 Weak scaling without openMP

The weak scaling analysis for the implicit scheme will use the same parameters as the one used for the explicit scheme, even though the former is stable for any combination of parameters. The total execution time is expected to be higher than that of the explicit scheme due to the conjugate gradient algorithm, needing to solve a linear system (with a threshold set at $R_{thr} = 0.001$). The pertinent data, presented as an average of 10 different runs, is reported in Table 5, while the linear efficiency and the timings are plotted in Figure 6.

np	L	Elements per process	Communication time (s)	Main time (s)	Efficiency
1	3.9	64000	0	45.489	100%
2	4.9	62500	0.07	45.776	97%
3	5.6	61731	0.11	44.502	98.6%
4	6.3	65536	0.21	48.095	96.8%
6	7.4	62208	0.19	46.036	96%
7	7.6	65219	0.23	48.78	95%
8	7.9	64000	0.24	47.424	95.9%
9	8.3	65856	0.24	54.037	86.6%
11	8.7	61952	0.24	49.602	88.7%
12	8.9	60750	0.45	52.516	82.2%

Table 5: Data of the weak scaling analysis for the implicit scheme

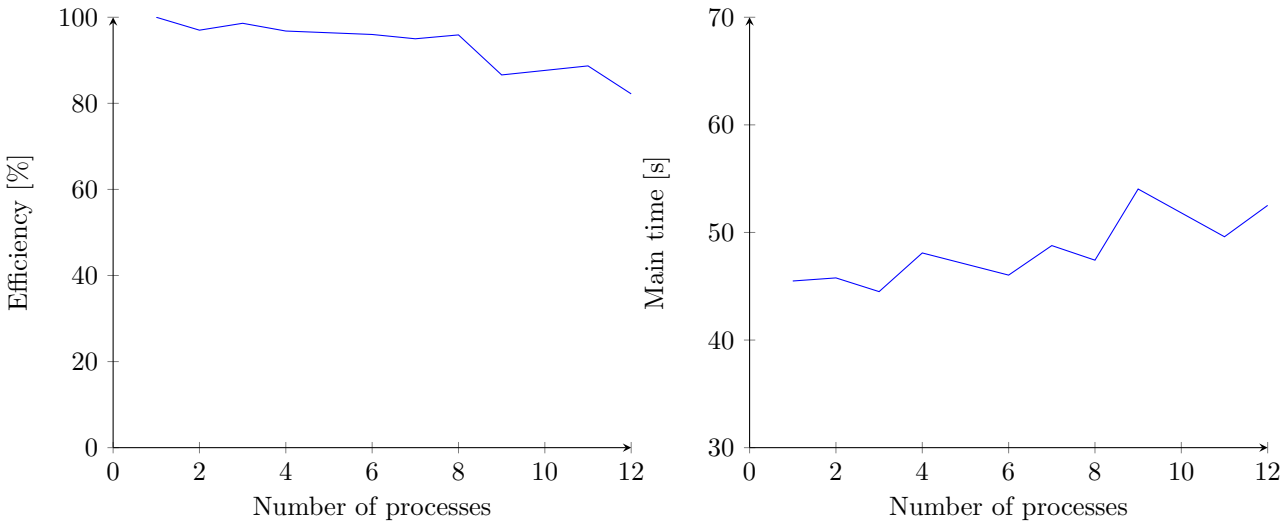

Figure 6: Weak scaling of the implicit scheme with an increasing number of processes

Like for the explicit scheme, the weak scaling is great, as the linear efficiency decreases slowly and does not drop below 82% for 12 processes. The scaling here is even better, mostly due to the fact that for the same parameters, the total execution time is greater for the implicit scheme and so the effect of parallelization is more important. The communication time plays a bigger role than for the explicit scheme but is still small in comparison to the main time.

3.2.2 Strong scaling without openMP

As for the explicit scheme with openMP, only the *medium* and *big* parameters will be considered. The data is reported in Table 6 (on page 12).

Parameters	np	Communication time (s)	Main time	Speedup	Efficiency
Medium	1	0	278.87 s	1	100%
Medium	2	0.3	150.48 s	1.85	92.6%
Medium	4	0.3	73.62 s	3.78	94.6%
Medium	8	0.3	40.77 s	6.84	85.5%
Medium	16	0.28	21.01 s	13.18	82.3%
Medium	32	0.3	14.73 s	18.9	59%
Big	1	0	675 min	1	100%
Big	2	10.36	392 min	1.72	86%
Big	4	18.9	174 min	3.84	96%
Big	8	31.5	88.41 min	7.63	95%
Big	16	18.9	46.86 min	14.04	90%
Big	32	18.6	26.68 min	25.29	79%

Table 6: Data of the strong scaling analysis for the implicit scheme, for the *medium* and *big* set of parameters

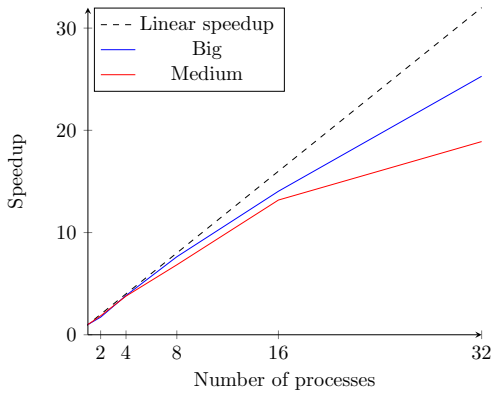

Figure 7: Speedup of the implicit scheme for the *medium* and *big* set of parameters

.

This analysis clearly shows that the bigger the problem, the better the scaling. The *big* implicit problem is the biggest considered this far and it is the one that provides the best speedup. As shown in Figure 7, the speedup is almost linear for the *big* parameters. The parallelization of the program allows an 11-hour long program to be reduced to less than half an hour. This can be further reduced with the help of openMP.

3.2.3 Weak scaling with openMP

The same parameters as usual will be used in this section, with a number of processes ranging from 1 to 12 and a number of threads varying from 1 to 16. The timings reported in Table 7 (on page 13) are the average timings of five different runs. The associated speedups are plotted in Figure 8 (on page 13). One observes that the speedup is really bad, especially above 8 threads, due to the fact that we did not manage to parallelize the main function of the algorithm. The overall openMP parallelization should be improved and will be covered in more details in section 4. Again, the scaling is better for an increasing number of processes but the speedup shape is similar. To conclude, when looking at the two weak scaling analyses, one can see that, for the same parameters, the implicit version takes between 7 to 9 times longer to complete.

np\threads	1	2	4	8	16
1	45.489	37.62	32.43	30.75	33.68
2	45.776	36.6	31.2	29	30.8
4	48.095	60.5	52.09	19.23	58.27
8	47.42	37.3	31.4	28.6	unstable
12	52.51	38.2	32.1	29.1	unstable

Table 7: Timings (in seconds) of the openMP weak scaling analysis for the implicit scheme

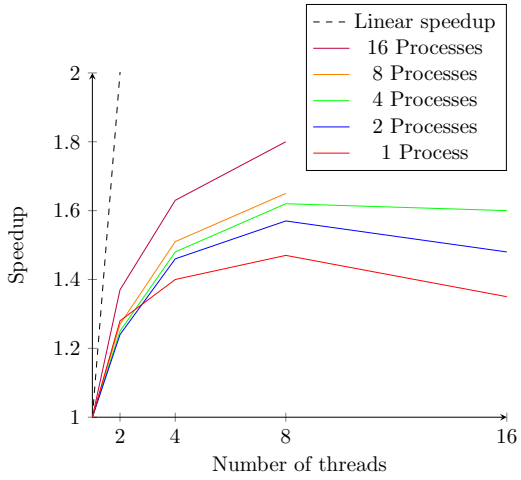

Figure 8: Speedup of the implicit program for an increasing number of threads and with the weak scaling parameters

3.2.4 Strong scaling with openMP

This time, only the *medium* parameters will be considered, since the *big* problem takes a large amount of time to execute. The timings used are the average timings of five different runs. The data is reported in Table 8 and the speedup is plotted in Figure 9 (on page 14). The shape of the speedups is similar to that of the weak scaling and even worse with an increasing number of processes. This could be explained by the size of the medium problem: the size of each sub-domain might not be big enough when the number of processes increases. The parallel regions are repeatedly called and destroyed too often, hence the parallelization is not effective.

np\threads	1	2	4	8	16
1	278.8	220.62	189.16	177.85	201.44
2	150.48	114.99	98.15	92.96	105.39
4	73.62	60.5	52.09	19.23	58.27
8	40.7	33.7	28.89	27.57	31.9
16	21.01	18.06	15.68	15.11	19.54

Table 8: Timings (in seconds) of the openMP strong scaling analysis for *medium* parameters

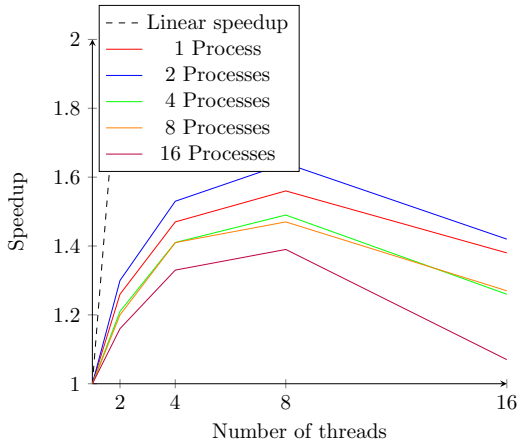

Figure 9: Speedup of the openMP analysis for *medium* parameters

3.3 Explicit vs. Implicit with varying diffusion and advection

In this section, the two schemes will be compared with a problem of equal dimensions, but with a varying diffusion D and advection v_y. The testings will be done on 4 processes with the same parameters as for the weak scaling:

$$L = 6.9, h = 0.1, v_x = 0, v_z = 0, T_{max} = 1, m = 0.001, S = 20, R_{thr} = 0.001.$$

First, the advection will be removed (v_x, v_y, v_z=0) whereas the diffusion will be almost maximum (D=1.2, which is close to the stable limit of the explicit scheme for these parameters: $mD \leq 0.00125$). Then the diffusion will be removed. Finally, several tests will be run for different relative importance of diffusion and advection. Each value is taken as an average of 5 different runs.

As can be seen in Table 9 (on page 15), the overall tendency is that the explicit timing is extremely stable while the implicit one varies with changing parameters. One observes an increase of the implicit time when the diffusion is removed, while the explicit time remains constant. This is due to the fact that the explicit scheme simply consists in sums inside a for loop; hence this scheme is mostly independent of the value of the parameters which intervene in the sums. On the contrary, for the implicit scheme, the parameters directly affect the values of the A matrix, and therefore the timings of the conjugate gradient method. Also, while the explicit scheme becomes unstable for some parameters, the implicit one remains stable but can take a lot of time to execute. As an example, the implicit scheme took more than 30 min to execute for v_x, v_y and v_z= 10.

In conclusion, if the parameters are small enough and are such that the explicit scheme is stable, one should use it since it is a lot faster than other combinations. If the parameters are such that the explicit scheme is unstable, one is forced to use the implicit one. However, even though this scheme remains stable, care should be taken when choosing the parameters as the execution time can increase out of control.

$\mathbf{D/v}_y$	**Explicit time (s)**	**Implicit time (s)**
1.2/0	7.9	42.9
0/-1	8.07	49.17
0.05/-1	6.7	48.05
0.05/-10	8.11	68.24
10/-1	unstable	44.25
1.2/-1	8.1	45.2
1.2/-10	8.12	68.24
0.05/-5	8.12	51.68

Table 9: Execution times of the explicit and implicit schemes for different sets of D and v_y

4 Things to improve

For the explicit scheme, the parallelization with openMP could be implemented better with larger parallel regions declared before the *for* loop in order to avoid creating and destroying threads at each iteration, hence reducing the efficiency.

For the implicit scheme, performances could be increased by avoiding the *get_ijv* function entirely. There is no need to store those arrays: since the number of non-zero values are known in advance from the dimension of the matrix and their values can be deduced from their row and column (see function *compute_Nz*). This way, the *matrixvector* function could be implemented with a series of *if* clauses where the calculation depends on the elements of the domain we consider. This would also lead to a straightforward openMP parallelization. We did not manage to parallelize the *matrixvector* function because we implemented it with a counter which depends on the value of Nz. This Nz depends on the process rank and on how the domain has been divided. Further parallelizing the *for* loop of the function with openMP implies a different counter for each thread, which is difficult to compute, along with increasing the amount of code.

As for the explicit scheme, the parallelization with openMP could be improved with larger parallel regions. In the implicit scheme, those were declared at the beginning of the functions *vectorsum* and *vectorvector*. Ideally, the region should be declared before the n loop, but there was an error related to the reduction variable inside the *vectorvector* function, and we never managed to work it around.