Efficient Strict-Binning Particle-in-Cell Algorithm for Multi-Core SIMD Processors

— Artifacts: Overview Document —

Yann Barsamian[1,2], Arthur Charguéraud[2,1], Sever A. Hirstoaga[2,3] and Michel Mehrenberger[3,2].

- ybarsamian@unistra.fr
- arthur.chargueraud@inria.fr
- sever.hirstoaga@inria.fr
- mehrenbe@math.unistra.fr

This document explains how to handle the software `Pic-Vert` *included as "artifact" with our article [3]. Our chunk data structure supports a thread-safe atomic insertion operation. This allows to run simulations on 3d simulations efficiently with an arbitrary number of cores, and improves a previous work with chunks [1]. This implementation of the strict-binning approach copes with fast particles, which improves previous efforts on this approach [15].*

[1]Université de Strasbourg, CNRS, ICube UMR 7357 (Strasbourg, France)
[2]Inria (Nancy, France)
[3]Université de Strasbourg, CNRS, IRMA UMR 7501 (Strasbourg, France)

Contents

Chapter 1

Getting Started

1.1 Introduction

1.1.1 What is the purpose of our software?

Our software written in `C`, called `Pic-Vert`, is a Particle-in-Cell (PIC) code for plasma physics. Three important steps are required for a code:

- (computer science) it should be efficient. PIC codes are usually memory bound, as a recent paper [17] points out: "*metrics such as flop/s or percentage-of-peak are less relevant for the predominantly memory-bound gyrokinetic PIC methods, as modern architectures require 10 flops per byte moved from DRAM in order to be compute-limited.*". One should then show some memory bandwidth usage.

- (mathematics) it should be verified. Some test cases have almost-exact theoretical solutions. One should verify that the code outputs the desired solution. For other test cases, some other diagnostics may be performed to assess correctness of the results, like the conservation of total energy.

- (physics) it should be validated. The physical model used in a given code always have some limitations. One should validate the model chosen with real-life simulations.

For performance, we provide a PIC code that (a) achieves close-to-minimal number of memory transfers with the main memory, (b) exploits SIMD instructions for numerical computations and (c) exhibits a high degree of OpenMP-level parallelism. We also coded MPI parallelism through particle decomposition, even though for future work in 3d, domain decomposition is a better approach. This last layer of parallelism is thus not the core of our work.

For verification, we simulate a classical 3d Landau-damping test case [4, 11]. We use the same parameters as in [16]. It is possible to test the correctness of the simulation by comparing the decay slope with the theoretical value obtained from the dispersion analysis ($\gamma = -0.008466$).

For validation, we simulate a 2d3v electron hole test case [14]. We are able to reproduce the results of this paper, which are validated by satellite measurements of phase-space holes in various parts of the magnetosphere.

1.1.2 How to check our results?

Depending on the hardware you can use, you will be able to check some parts of our results or all of them.

First, you need to install dependencies, as explained in Section 1.2. Then you need to edit our compile scripts in order to compile and run our code, as explained in Section 2.1. Finally, you will then be able to compare the outputs of the simulation to our claims, as explained in Section 2.4.

For performance, `Pic-Vert` requires that you use the full machine: as many threads as you have cores, and as many particles as can fit in memory. You can test `Pic-Vert` on your laptop, even though for performance it is probably not the best choice. Be aware that:

- by the number of cores, we refer to the number of physical cores. On one of our laptops, “About this laptop” will tell us that we have “Intel® Core^{TM} i5-6300U CPU @ 2.40 GHz x 4”. But if we type `lstopo` inside a terminal, we see that there are in fact only 2 physical cores with two processing units inside each of them, leading to 4 processing units in total (hyper-threading). Because our PIC code is memory bound, we wrote scripts in order to use only physical cores, which avoids hyper-threading. Whenever we run `Pic-Vert` on this laptop, we thus use 2 threads. If you try to put more threads than the number of physical cores, our script will output an error.
- by the memory available, we mean the total memory available minus the memory already in use by the Operating System (and maybe other applications). On the same laptop, “About this laptop” will tell us that we have 15.6 GB available. But if we type `htop` inside a terminal, we see that there are typically already 2 GB used. So there is only 13.6 GB of memory left for our simulations (if we use more, there will be memory swapping, and this greatly decreases performance).

Before running test cases to match our results, you may want to run a quick test on your laptop, just to know if everything runs fine. When you edit the scripts, you can then reduce the number of particles (about 10 millions) and reduce the number of iterations (about 20). This should run quickly (less than a minute) and already prove that everything runs without problem. Then, you can increase those numbers (increase the number of particles to match our requirement that there are a lot of particles per grid cell; increase the number of iterations so that the average time per iteration computed makes good statistical sense).

If you can use a dedicated server with more cores (maybe around 20), then of course the performance should be more in accordance to our results. We think that this is probably the best option, as you probably have one of such computers available within your research team.

In the two last cases, it will not be possible to test MPI scaling of our code, like we did on 128 MPI processes. This is not a crucial claim in our paper. You can test the MPI scaling of our code if you can use a dedicated supercomputer, but the scripts to run `Pic-Vert` are not included within this artifact. We can help you build those scripts if needed, just contact us.

As a first side note, we would like to say that the 3d test case is the most important one. Then comes our claim that our code is still efficient with a substantial number of fast particles. Finally comes our 2d3v test case, which takes more time to check (HDF5 outputs to be opened with VisIt in addition to the performance check).

As a second side note, we encourage you to run simulations with some slight changes in the parameters, in order to be convinced that our code is really a plasma simulation, and does not just mimic it before outputting pre-determined diagnostics. We believe that by providing our source code, you will not fear that we are “cheating” on the outputs of our software.

1.1.3 What is in the artifact?

This artifact is composed of several folders:

- `include` and `src` contain all the modular code for `Pic-Vert` (as .h and .c files). By opening them, it is possible to check that we implemented the data structures we describe in the paper:
    - `compiler_test.h`: compiler check to know which features we can use.
    - `diagnostics.c/h`: electric field diagnostics.

  - `fields.c/h`: data structure for the electric fields.
  - `hdf5_io.c/h`: handling of HDF5 outputs.
  - `initial_distributions.c/h`: possible initial distributions of particles.
  - `math_functions.h`: useful mathematical functions not already in standard libraries.
  - `matrix_functions.c/h`: dynamic allocation of arrays.
  - `meshes.c/h`: data structure for the grid.
  - `output.c/h`: handling of outputs other than HDF5.
  - `papi_handlers.c/h`: handling of PAPI [18] performance counters.[1]
  - `parameter_reader.c/h`: reading parameter files for simulations.
  - `parameters.h`: useful parameters (architecture, mathematics, simulation. . .)
  - `particle_type_concurrent_chunkbags_of_soa_XXX.c/h`: data structure for particles.[2][3]
  - `poisson_solvers.c/h`: data structure for the FFT Poisson solver.
  - `random.c/h`: pseudo-random number generators available.
  - `rho.c/h`: data structure for the charge density.
  - `space_filling_curves.c/h`: use of space-filling curves in the code.[4]
  - `variadic.h`: allows to write function with default arguments in `C`.[5]

- `simulations` contains the PIC simulations that we used for this article, and that uses the files in the previous point.[6]
- `scripts` contains the compile and run scripts for our simulations.
- `memory` contains spread sheets to compute the number of particles you should put in the simulation, given your architecture (prior to compilation). This step is explained later.
- `Stream-test` contains the Stream benchmark [13], in order to detect the practical memory bandwidth peak of your architecture. Please be aware that the theoretical peak of your architecture may not be the one given by the vendor, as maybe not all the memory channels are installed. To know how many channels are installed, please look at the memory banks after you type:

```
sudo lshw -class memory
```

- `post-process` contains scripts to "automate data extraction and the production of plots".
- `sample-outputs` contains "files that represent expected outputs".

[1]They can be activated in our code with `-DPAPI_LIB_INSTALLED`, provided that you installed the PAPI library first.

[2]The thread-safe atomic insertion operation can be found on line 422 of the `C` file, inside the function `bag_push_concurrent`.

[3]Our claim that "*the space used by our algorithm, in addition to the minimal amount of memory needed to represent the particles, grows in proportion with:* $4K \cdot \mathsf{nbCells} \cdot \mathsf{sizeof(particle)}$" can be checked in the function `init_all_chunks`. This function is much more general than what we describe in our paper. In our paper, `nb_bags_per_cell` is equal to 2, which implies that `number_of_spare_chunks_per_parity` is equal to $2 \cdot \mathsf{nbCells}$, and because a chunk weights in memory $K \cdot \mathsf{sizeof(particle)}$, our claim can be verified on line 817 of the `C` file.

[4]They are particularly useful with other data structures for particles [2].

[5]This somehow mimics the default argument feature, e.g. in `C++` or `Fortran`.

[6]It is possible to check that we implemented the algorithm shown in Figure 2. The "Foreach time step" of line 5 of this algorithm starts at line 592 in `sim3d_aocosoa.c`, with the `for` loop on the variable `i_time`.

1.1.4 Can we customize the artifact?

Apart from the runs we detail in this document, other test cases are available. They are documented in the files `initial_distributions.c/h`. You may add some initial distributions if needed. All the parameters can be modified in the parameter files (see `parameters_2d3v.txt` for a 2d3v simulation or `parameters_3d.txt` for a 3d simulation). We can provide simulation files in 1d or 2d if needed, and we plan to have the full code of our software (including other data structures, other simulations, information on how to tune the data structures and loop optimizations depending on your architecture. . .) available as Open Source.

1.2 Dependencies

1.2.1 What are the dependencies?

`Pic-Vert` has several "additional software to install":

- a `C` compiler that supports OpenMP [5]. The compiler must at least support OpenMP 3.0, but it is better if it supports OpenMP 4.0 (`#pragma omp simd`; without this support, those pragmas are desactivated in our code, but the vectorization would then be done with the automatic vectorization from the compiler which usually gives less satisfactory results). For example, `icc` supports OpenMP 4.0 since version 15.0 and `gcc` since version 4.9.1.

- an MPI [6] wrapper. All the scripts are given for a standard installation of OpenMPI [9], but you should be able to modify them to use another wrapper if needed.

- the FFTW3 library [8].

- the HDF5 library [10]. This dependency is only needed if you want to output an HDF5 file for the data needed for Figure 4. Note that for this image we needed to use a lot of memory to fit 64 billion particles (hence a lot of compute nodes). If you have only access to a standard machine and have to use less particles (e.g. 2 billion), you will have a image with more noise. However, most of our claims are on the 3d code, so if you cannot install the HDF5 library, you can test our 3d code, which does not require the HDF5 library. If really needed, we can re-write our 2d3v code to get rid of this dependency, and output binary and/or ascii files instead, but this would decrease performances.

1.2.2 Which precise versions were used for our paper?

In our paper, the following versions of dependencies have been used:

- icc 17.0.4, icc 18.0.0, gcc 5.4.0 and gcc 6.4.0. Performance results are presented in our paper with an Intel-compiled code. If using another compiler, variations in performance may happen. In our tests with the GNU compiler, there are not much differences for our `Pic-Vert` code, but there is sometimes up to 30% memory bandwidth variation for the Stream test, a variation also reported by others[7].

- OpenMPI versions 1.10.2 and 1.10.3.

- FFTW version 3.3.4.

- HDF5 versions 1.8.16 and 1.8.17.

[7] `https://software.intel.com/en-us/forums/intel-c-compiler/topic/599289`

1.2.3 How to install these dependencies?

We guess that you already have one favorite `C` compiler installed. For the other dependencies, here is how to install them under Ubuntu:

For openmpi and hdf5 type:

```
sudo apt-get install libopenmpi-dev openmpi-bin libhdf5-openmpi-dev
```

For the FFTW library, type:

```
sudo apt-get install libfftw3-dev
```

1.3 What are the post-processing tools needed?

To reproduce the figures in our paper, the following tools are needed:

- a tool like gnuplot [19] to convert output files to graphs for Figure 3 and Figure 5.
- a tool like VisIt [12] to convert the HDF5 files to images for Figure 4.
- a standard spread sheet editor like Libre Office Calc [7].

Chapter 2

Step-by-Step Instructions

2.1 Compiling and Running

All the compile (`a_compile_XXX.sh`) and run (`b_run_XXX.sh`) scripts needed for `Pic-Vert` are inside `Pic-Vert/scripts`. We have 4 different set of scripts, in order to prove the different claims of our article:

- `3d_performance` is here to assess our claim that in 3d, our algorithm achieves "30.4 million particles per second per core" and "54% of the practical peak" (memory bandwidth). It also allows to verify Figure 3.
- `Heat_crossing` is here to assess the 4 first lines of Table 1 (7 test cases that achieve the given percentages for particle crossing and atomic pushes).
- `Heat_performance` is here to assess the last line of Table 1 (the slowdown due to fast particles in the 6 last cases, with respect to the first test case).
- `2d3v_performance` is here to assess our claim that in 2d3v, our algorithm achieves "40.7 million particles per second per core". It also allows to verify Figure 4 (although you should have an image with more noise if you have to use less particles) and Figure 5.

An additional folder, `Pic-Vert/Stream-test`, allows to test the practical memory bandwidth peak, with the Stream benchmark [13]. It also contains compile and run scripts.

Before starting, you have some modifications to make. Modify the system parameters in `your_configuration.sh`:

Choose your compiler. Here, scripts are only given for `icc` and `gcc`. If you want to use another compiler, it should be possible, please modify the scripts accordingly. Managing the thread affinity is crucial for performance: feel free to use your preferred compiler as long as you can manage the thread affinity. Change the following line if needed:

```
compiler=icc
```

If you use `icc`, please make sure to enter the correct path to the dynamic Intel OpenMP library `libiomp5.so`, by modifying the following line:

```
INTEL_OPENMP_DYNAMIC_LIBRARY_PATH=/opt/intel/.../intel64_lin/
```

Choose your architecture parameters. The number of threads and sockets has to be chosen. As an example, in our configuration, we launch simulations on a processor which is made of two 24-cores sockets. So the number of sockets would be 2 and the number of threads 24 (leading to a total number of 48 threads). Please put as many sockets as you have NUMA nodes to avoid NUMA bandwidth problems, and put as many threads as there are cores per NUMA node. Change the following lines if needed:

```
nb_threads=10
nb_sockets=2
```

Update library paths if needed. If the compiler complains, you have to add the FFTW library path in the compile script, with `-I/your/path/to/header -L/your/path/to/library`. We did not have to do it on the different hardwares we tested, but you will maybe have to do it. For the 2d3v simulation only, however, it is mandatory to check for the path of the external library HDF5 (and possibly zlib if your compiler complains, exactly as for FFTW). Instructions to do it quickly are given in `your_configuration.sh`:

```
#Include path for hdf5.h (find your right path by typing "locate hdf5.h"
#                         in a terminal, maybe preceded by "updatedb")
HDF5_HEADER_PATH=/usr/include/hdf5/openmpi

#Library path for libhdf5.a (find your right path by typing "locate libhdf5.a"
#                            in a terminal, maybe preceded by "updatedb")
HDF5_LIBRARY_PATH=/usr/lib/x86_64-linux-gnu/hdf5/openmpi
```

Modify the simulation parameters in `scripts/the-run-you-want/parameters_XXX.txt` or `a_compile_XXX.sh`:

Update the number of particles. You have to check the memory you have on one socket (one NUMA node) of your architecture. In `memory/memory-XXX.ods` we provide a simple way to compute the maximum number of particles that can fit, provided that the memory limit of your architecture is correctly filled. Our hardware had 96 GB of memory, this is the cell you have to change. If you have a machine with really low memory (e.g., 2 GB), you may also lower the number of grid cells to be able to fit more particles. Do not forget to update the parameter files in this case. Please set the number of particles as close as possible to this maximum, as our software makes the assumption - explicitly written in our paper - that there is a relatively high number of particles per grid cell. To set the number of particles per NUMA node in the simulation, please edit the following line in the parameter file:

```
nb_particles=XXX
```

(this line has to be modified in the compile script instead of the parameter file for the "heat" simulations because it was more convenient to change the value in just one file rather than in 7 different parameter files, as there are 7 different "heat" test cases).

TL;DR: edit `your_configuration.sh`; modify the variables `nb_runs`, `compiler`, `nb_threads` and `nb_sockets` (if not sure, begin with 1 socket and 1 thread for a quick test). Modify the variable `INTEL_OPENMP_DYNAMIC_LIBRARY_PATH` if you use `icc`.

Edit `scripts/the-run-you-want/parameters_XXX.txt`; modify the variables `nb_particles` and `nb_iterations` if needed (10 millions particles with 20 iterations allows a quick test). Do not try the 2d3v test case first, as it takes more time.

Warning: OpenMPI outputs a warning because we use the deprecated feature `--cpus-per-proc`. This should not be a problem.

2.2 Our hardware configuration and performance results

Our hardware is the one available on the A3 partition of the EUROfusion Marconi supercomputer[1]. Each node is made of two sockets, where a socket is an Intel Xeon Platinum 8160 @ 2.1 GHz (Skylake), with 96 GB of RAM, 6 memory channels, and 24 cores [2]. The theoretical

[1]https://www.cineca.it/en/content/marconi

[2]https://ark.intel.com/products/120501

memory bandwidth peak of our hardware is 127.99 GB/s [3]. The practical peak, as measured by the Stream benchmark [13], is 98.2 GB/s. Our `C` code was compiled using Intel C Compiler 17.0.4, and the FFTW3 library for the Poisson solver. To avoid as much performance issues as possible, the performances are recorded by running simulations on a full node, one MPI process per socket, and as many threads as there are cores per socket. Here, it means 2 MPI processes and 24 OpenMP threads per MPI process.

- Stream benchmark: the scripts together with the results for the Marconi supercomputer can be found in `sample-outputs/Stream-test`. For each Stream run, we extracted the "triad" results (which provided the highest memory bandwidth) in `skylake-bandwidth.dat`. The raw output from the Stream test from 1 to 24 cores can be found in `std_output.txt`.

- 3d simulation: the scripts together with the results for the Marconi supercomputer can be found in `sample-outputs/3d_performance`.
  Results in the paper are given for a grid size of 64 x 64 x 64 on 2 x 24 cores. They can be found in `3d_performance/std_output_run1.txt`. The total execution time was 1370.09 s (22 minutes). The number of particles processed by second was 1.45975 billion, or 30.4 million particles per second per core.

- Percentages of fast-moving particles: the scripts together with the results for a local machine can be found in `sample-outputs/Heat_crossing`. This time we used a local computer instead of the supercomputer Marconi, because we needed a lot of time to tune the parameters to get the percentages shown in the paper. Waiting for the supercomputer to run our jobs was not an option, as we had to test, adjust, and modify a lot of times our parameters.
  Results in the paper are given for a grid size of 64 x 64 x 64. They can be found in each of the sub-folders `Heat_crossing/sub-folder/std_output.txt`. Those percentages are summarized in `Heat_crossing/heat_crossing.ods`. The total execution times vary between 100 s and 120 s (we recall that there is additional instrumentation code to count the different particles that cross cells, so timings obtained with the simulation `instrum_sim3d_aocosoa.c` should not be taken to assess our code's performance).

- Impact of fast-moving particles on performance: the scripts together with the results for the Marconi supercomputer can be found in `sample-outputs/Heat_performance`.
  Results in the paper are given for a grid size of 64 x 64 x 64 on 24 cores. They can be found in `Heat_performance/std_output.txt` and `Heat_performance/heat_performance.ods`. The total execution times vary between 159 s and 170 s.

- 2d3v simulation: the scripts together with the results for the Marconi supercomputer can be found in `sample-outputs/2d3v_performance_XXX`.
  Results in the paper are given for a grid size of 256 x 256 on 24 cores. They can be found in `sample-outputs/2d3v_performance_256x256/std_output_runXXX.txt`. The total execution time was around 4116.05 s (1 hour and a quarter). The number of particles processed by second was 971.806 million, or 40.5 million particles per second per core.
  Additional results for a grid size of 512 x 512 are here given on 2 x 24 cores. They can be found in `sample-outputs/2d3v_performance_512x512/std_output_run1.txt`. The total execution time was 5496.61 s (1 hour and a half). The number of particles processed by second was 1.8193 billion, or 37.9 million particles per second per core.
  Finally, we provide all the diagnostics needed to reproduce our Figure 5. They also contain additional diagnostics to check the total energy; these additional diagnostics require an additional loop over the particles and therefore reduce the performance. They can be found in the directory `sample-outputs/Muschietti_y_energy`.

[3] https://en.wikichip.org/wiki/intel/xeon_platinum/8160

2.3 The different output files of our simulations

Each time a simulation is run, different outputs are created.

First, the most important one: the parameters and performance outputs, on the standard output (also stored in a file). For an example 3d simulation from our paper, it reads as follows:

```
Creation time (2000000000 particles) : 723.24 sec
#CHUNK_SIZE = 256
#VEC_ALIGN = 64
#OMP_TILE_SIZE = 2
#OMP_TILE_BORDERS = 1
#mpi_world_size = 2
#num_threads = 24
#alpha = 0.050000000000000003
#x_min = 0
#x_max = 22
#y_min = 0
#y_max = 22
#z_min = 0
#z_max = 22
#delta_t = 0.050000000000000003
#thermal_speed = 1
#initial_function_case = LANDAU_3D_PROD_OF_ONE_PLUS_COS
#ncx = 64
#ncy = 64
#ncz = 64
#NB_PARTICLE = 2000000000
#num_iteration = 500
---------- Vlasov-Poisson 3d - Array of [Optim] Chunkbags of SoA (1 private + 1 shared / cell) -
---------- always sort ----------
Execution time (total)    : 1370.09 s
Array of particles update : 1362.32 s (99.4327%)
- Including particle loop : 1357.22 s (99.0606%)
- Including append        : 5.09836 s (0.372118%)
Reduction of rho          : 4.21777 s (0.307845%)
Poisson solver            : 1.27372 s (0.0929657%)
Nb. particles / s : 1.45975e+09
Time / particle / iteration : 0.681161 ns
```

You have first the output of the parameters chosen, to control that the simulation was run with the correct parameters. Then comes timings of different parts of the main time loop. Here, the particle loop (lines 6-23 of our algorithm in Figure 2), then the append phase (lines 24-28), then the reduction of rho (lines 29-33) and finally the Poisson solver (line 34). The sum of all those timings gives the total execution time, and the total number of particles divided by the total execution time gives the number of particles processed by second. As stated in the paper, this takes into account all the operations performed, not just the particle loop. If taking into account only the particle loop, this number increases slightly. We also give the time taken for one particle per iteration, which is sometimes given by papers instead of the number of particles processed by second (it is of course equivalent to present one or the other). Those performance results are also given in files `print_time_sorting_rankXXX.dat` because sometimes we need to check that every MPI process has roughly the same execution times that the others (on heterogeneous clusters, it may happen that one will wait for the others to finish at each iteration - if you have to launch on heterogeneous clusters, then you have to load balance the particles differently, this is out of the scope of this paper; on our cluster everything should be homogeneous and if the timings are not all similar, it probably means that there is a hardware problem).

Then you have a file named `diag_lee_XXXcorners.txt` (4 corners in 2d3v, 8 corners in 3d). It gives the electric energy, useful to produce Figure 3. In 2d3v, an additional file `diag_energy.txt` additionally gives the energy of the different Fourier modes (and also the kinetic energy if the code is compiled with `-DPIC_VERT_KINETIC_ENERGY`; we can then deduce the total energy from electric and kinetic energies), useful to produce Figure 5.

You also have a file named `diag_speed_XXXcorners.txt` (4 corners in 2d3v, 8 corners in 3d). This files outputs performance results iteration by iteration. You can check that the timings are mostly unchanged during the simulation, from which we can conclude that the performance results are fairly presented (it would not be fair to present results on the best iterations from a full simulation).

Finally, for 2d3v simulations, you have hdf5 files (.h5 and .xmf files).

2.4 Comparing outputs to our claims

- `Stream-test`: once the test is finished, you will have an output that will look like the following. In the "Best Rate MB/s" column, take the best of the 4 numbers as the reference peak memory bandwidth for your hardware. Please go inside `post-process/bandwidth3d.ods`, and fill the green cell corresponding to the memory bandwidth of the Stream test (please convert the result given in MB/s to GB/s).

```
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:           94698.1     0.407255     0.405499     0.410791
Scale:          93897.0     0.409859     0.408959     0.412704
Add:            98194.6     0.588433     0.586590     0.599418
Triad:          98208.5     0.590043     0.586507     0.616153
```

- `3d_performance`: once the simulation is finished, you can go in `post-process/` and type:

```
gnuplot landau_electric_energy.plot
```

  This should output the same graph as Figure 3.
  You should then go in `3d_runs/run1/` (and possibly more if you asked for more than 1 run to avoid statistical noise). You will find a file named `std_output_run1.txt`. At the end of this file, there is a line that begins with `Nb. particles / s :`. Just divide the number after by the total number of cores of your processors. This should not necessarily be 30.4 million particles per second per core, like us, because it depends on your memory bandwidth. A better check is to look that the simulation indeed roughly gave 50% of the peak memory bandwidth. To check for the bandwidth, go inside `post-process/bandwidth3d.ods`, and fill the green cells with the parameter you used for the simulation (number of particles) and the execution time you got from the simulation (the line that begins with `Execution time (total)      : `). You will then have the percentage of peak computed - provided that you also filled the green cell from the Stream benchmark. If the Stream benchmark does not work, you can always compare the memory bandwidth to the theoretical one, as given by your vendor. Please be aware that it is possible that not all the memory channels are installed on your machine, and that this can modify the theoretical maximum memory bandwidth.

- `Heat_crossing`: once the simulation is finished, you can go in `post-process/` and type:

```
./heat_percentages.sh
```

  This will output a file named `heat_percentages.txt`. You can now check, for each of the 7 test cases, that the percentages given are in accordance with the 4 first lines of Table 1 (7 test cases that achieve the given percentages for particle crossing and atomic pushes).

- `Heat_performance`: once the simulation is finished, you can go in `post-process/` and type:

  ```
  ./heat_performance.sh
  ```

  This will output a file named `heat_performance.txt`. You can now copy-paste the timings inside `post-process/heat.ods` (in the green column) and check that they roughly give percentages (in the red column) corresponding to the last line of Table 1 (the slowdown due to fast particles in the 6 last cases, with respect to the first test case).

- `2d3v_performance`: once the simulation is finished, you can go in `post-process/` and type:

  ```
  gnuplot muschietti_electric_energy.plot
  ```

  This should output a graph with a curve similar to one of the curves in Figure 5, depending on the number of particles and grid size you could put.
  You should then go in `2d3v_runs/run1/` (and possibly more if you asked for more than 1 run to avoid statistical noise). You will find a file named `std_output_run1.txt`. At the end of this file, there is a line that begins with `Nb. particles / s :`. Just divide the number after by the total number of cores of your processors. This number depends on your memory bandwidth, and it is not really useful to test the performance both in 2d3v and in 3d; testing the accordance of your results to ours in 3d is already enough but the same methodology applies here of course.

  You should then open VisIt to reproduce our Figure 4. Click on the “Open” button. Go in the folder `2d3v_runs/run1`, filter by `*.xmf` and open `rho*.xmf database`. Then click on the “Add” button, choose “pseudo color” and “values”. Then click on “Draw”. You can now manage the different ρ snapshots taken during the simulation with the “Play / Stop / Next / Previous” buttons.

Chapter 3

References

Articles

[1] Y. Barsamian, A. Charguéraud, and A. Ketterlin. “A Space and Bandwidth Efficient Multicore Algorithm for the Particle-in-Cell Method”. In: *Parallel Processing and Applied Mathematics: 12th International Conference (PPAM)*. Vol. 10777. Lecture Notes in Computer Science. (Slides: http://www.barsamian.am/Slides/slides_2017-09.pdf). Springer, Cham, 2018, pp. 133–144 (cit. on p. 1).
DOI: 10.1007/978-3-319-78024-5_13.

[2] Y. Barsamian, S. A. Hirstoaga, and É. Violard. “Efficient Data Structures for a Hybrid Parallel and Vectorized Particle-in-Cell Code”. In: *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. (Slides: http://www.barsamian.am/Slides/slides_2017-06.pdf). IEEE Computer Society, 2017, pp. 1168–1177 (cit. on p. 5).
DOI: 10.1109/IPDPSW.2017.74.

[3] Y. Barsamian, A. Charguéraud, S. A. Hirstoaga, and M. Mehrenberger. “Efficient Strict-Binning Particle-in-Cell Algorithm for Multi-Core SIMD Processors”. In: *24th International Conference on Parallel and Distributed Computing (Euro-Par)*. 2018 (cit. on p. 1).

[4] C. K. Birdsall and A. B. Langdon. *Plasma Physics via Computer Simulation*. McGraw-Hill, New York, 1985 (cit. on p. 3).

[8] M. Frigo and S. G. Johnson. “The Design and Implementation of FFTW3”. In: *Proceedings of the IEEE* 93.2 (2005). http://www.fftw.org, pp. 216–231 (cit. on p. 6).
DOI: 10.1109/JProceedings2004.840301.

[9] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. “Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation”. In: *Proceedings of the 11th European Parallel Virtual Machine / Message Passing Interface Users’ Group Meeting (EuroPVM/MPI)*. Vol. 3241. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 2004, pp. 97–104 (cit. on p. 6).
DOI: 10.1007/978-3-540-30218-6_19.

[11] R. W. Hockney and J. W. Eastwood. *Computer Simulation Using Particles*. Taylor & Francis, Inc., 1988 (cit. on p. 3).
DOI: 10.1201/9781439822050.

[13] J. D. McCalpin. “Memory Bandwidth and Machine Balance in Current High Performance Computers”. In: *IEEE Computer Society Technical Committee on Computer Architecture Newsletter (TCCA)* (1995), pp. 19–25 (cit. on pp. 5, 8, 10).
URL: https://www.cs.virginia.edu/stream/.

[14] L. Muschietti, I. Roth, C. W. Carlson, and R. E. Ergun. “Transverse Instability of Magnetized Electron Holes”. In: *Physical Review Letters* 85.1 (2000), pp. 94–97 (cit. on p. 3).
DOI: 10.1103/PhysRevLett.85.94.

[15] H. Nakashima, Y. Summura, K. Kikura, and Y. Miyake. “Large Scale Manycore-Aware PIC Simulation with Efficient Particle Binning”. In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, 2017, pp. 202–212 (cit. on p. 1).
DOI: 10.1109/IPDPS.2017.65.

[16] L. F. Ricketson and A. J. Cerfon. “Sparse grid techniques for particle-in-cell schemes”. In: *Plasma Physics and Controlled Fusion* 59.2 (2017), p. 024002 (cit. on p. 3).
DOI: 10.1088/1361-6587/59/2/024002.

[17] W. Tang, B. Wang, S. Ethier, G. Kwasniewski, T. Hoefler, K. Z. Ibrahim, K. Madduri, S. Williams, L. Oliker, C. Rosales-Fernandez, and T. Williams. “Extreme Scale Plasma Turbulence Simulations on Top Supercomputers Worldwide”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Press, 2016, pp. 502–513 (cit. on p. 3).
DOI: 10.1109/SC.2016.42.

Tools

[5] OpenMP ARB (Architecture Review Boards). *Open Multi-Processing*. 1997 (cit. on p. 6).
URL: http://www.openmp.org/.

[6] MPI Forum. *Message Passing Interface*. 1993 (cit. on p. 6).
URL: https://www.mpi-forum.org/.

[7] The Document Foundation. *Libre Office*. 2010 (cit. on p. 7).
URL: https://www.libreoffice.org/.

[10] The HDF Group. *Hierarchical Data Format 5*. 2007 (cit. on p. 6).
URL: https://www.hdfgroup.org/.

[12] Lawrence Livermore National Laboratory. *Visualize It*. 2002 (cit. on p. 7).
URL: https://wci.llnl.gov/simulation/computer-codes/visit/.

[18] Innovative Computing Laboratory at the University of Tennessee. *Performance Application Programming Interface*. 2000 (cit. on p. 5).
URL: http://icl.cs.utk.edu/papi.

[19] T. Williams and C. Kelley. *Gnuplot: An Interactive Plotting Program*. 1986 (cit. on p. 7).
URL: http://www.gnuplot.info/.