Available online at www.prace-ri.eu

**Partnership for Advanced Computing in Europe**

# Optimization of the LPPic2D code for scaling magnetized cold plasma simulations

U. K. Seth[a*], G. Hautreux [a*], C. Jourdain [a], V. Cameo [a]

[a] *Centre Informatique National de l'Enseignement Supérieur (CINES), France*

**Abstract**

This report summarizes the optimization work carried out on LPPic2D, an in-house code of the propulsion group based at 'Laboratoire de Physique des Plasmas (LPP)'. The work was carried out by the HLST team of CINES in collaboration with the developers of LPPic2D. LPPic2D simulates the dynamics of plasma for several applications. The main application target of this research group is to simulate magnetized cold plasma physics in low-pressure Hall Effect thrusters, used for electric propulsion of satellites in space. LPPic2D is a Fortran code parallelized with message passing interface (MPI). The simulation algorithms used in the code are based on Particle-in-Cell (PIC) and Monte-Carlo collisions methods.
Using profiling reports on LPPic2D, several steps were taken to optimize the code. The work started with the memory access optimizations, followed by other general and more specific optimizations during the course of this project. A comparison of speedup and parallel efficiency of the original code against the optimized code is provided in this report. As a final step of the project, the scalability results of optimized code on the Irene AMD-Rome partition of the Joliot-Curie supercomputer are also presented here.

DOI: 10.5281/zenodo.7061340

## 1 Introduction

The propulsion group at the Laboratoire de Physique des Plasma (LPP), Sorbonne University, Paris, has been developing the code LPPic2D for a few years now [1]. It is a purely particle based code which uses 'Particle-in-cell' technique to model the movements and interactions of charged particles in a 2-dimensional plane. The interactions among the particles are modelled with classic Monte-Carlo collision algorithms. LPPic2D is a Fortran code parallelized with message passing interface (MPI). It uses the Hypre library [2] for solving Poisson equations and HDF5 for parallel I/O operations. The main application area of this research group with the LPPic2D code, is to simulate magnetized cold plasma physics in low-pressure Hall Effect thrusters, used for electric propulsion of satellites in space.
The research group has been using the HPC cluster 'Occigen' of CINES for quite a few years now, and in recent years the group was among the top consumers of CPU-hours on Occigen. Our initial assessment suggested that the group required HPC support to optimize the code so that they can optimize their resource consumption on Occigen. The initial steps involved preparing the validation cases and profiling the original version of the code. The whole optimization project was managed on GitLab at CINES [3]. We performed the profiling tests with Intel Vtune [4] and Score-P [5] to catch possible hotspots in the code. The profiling was carried out on both serial and parallel cases. Several hotspots were identified which were considered for optimizations at different stages of the project. Some snapshots of some profiles are available in appendix A. We also generated vectorization reports

* Corresponding author email address: seth@cines.fr, hautreux@cines.fr

with Intel Advisor [6] and analyzed them. Vectorization reports on serial test runs showed that only ~6% of the elapsed time, on the base case, was spent in vectorized loops and rest of the code was running in serial mode.
An Arrays of Structure (AoS) memory strategy has been used in the original code, where derived data types for managing different physical parameters for different plasma species were implemented. These AoS are not suitable to exploit the ever-expanding vector features of modern processors as they prevent an effective use of the memory hierarchy of the processors as well. The overall parallel performance of the code was also analyzed with Intel's Application Performance Snapshot (APS) tool [7]. The APS analysis indicated significant memory and cache stalls with a test run on 2 nodes (48 cores). More than 50% of the elapsed time was spent in MPI library calls alone, having a fraction of around 20% of the total elapsed time in MPI_WAIT calls.
Based on our profiling results, a test case was prepared with a Structure of Arrays (SoA) approach, instead of AoS, in just one subroutine. Similar profiling tests were carried out on this test case, which showed a speedup higher than 1.4 in elapsed time inside the updated subroutine alone, and a speedup of 1.14 in the overall elapsed time of this small run. Also, the time spent in vectorized loops in this run increased from ~6% to ~13%, indicating that several loops were now vectorized with the SoA approach. These test results and overall analysis of the code suggested that the code was not able to fully realize the capacity of the cluster as whole, and significant performance gains could be attained by converting the AoS strategy to hardware friendly SoA approach. Considering the outcomes of this test case run several steps were identified to optimize the code in a step-by-step manner which are discussed in the next section of this report.

# 2 Optimization of LPPic2D

Optimization work was carried out on LPPic2D by the HLST team at CINES in collaboration with the developers/users of the code at LPP. Considering the profiling reports and initial tests, several steps were considered to optimize the code which will be discussed in this section of the report. Firstly, the code was compiled and tested with a more recent and stable version of compiler and libraries' environment than the one which the research group was using at the time. It was observed that compiling with Intel compilers and Intel MPI version 2018 took around 10% less execution time in comparison with the *Intel/17* and *Open MPI/2.0* environment, for the 2h30min test case. Thus, the research team was advised to use this more recent compiler environment and benefit from the recent compiler updates.

## 2.1 Memory access optimizations

Optimization work started with an attempt to improve memory access by the code by modifying the data structure scheme used in the code. In the original code, different types of charged particles (electrons, species etc.) were defined by using Fortran's derived data type (structure). Listing 2.1 shows the derived data type named '`particle`', which has particle properties as its components. Every type of particle was then defined as a dynamic array of this particle structure (AoS) as follows:

```
!definition of a data structure for particles
type particle
    integer             :: index
    integer             :: charge
    real(8),dimension(3):: V
    real(8)             :: X
    real(8)             :: Y
    real(8)             :: Z
 end type particle

!declaration of a particle type as AOS for electrons
type(particle), dimension(:), allocatable :: elecs
!allocation of 'elecs' AOS
allocate(elecs(5000))
```

Listing 2.1 Example data structure 'particle' and corresponding array of structure (AOS) declaration for electrons

The elements of an array are stored contiguously in memory. So with AoS strategy, every element of an AoS (eg. `elecs`) is a particle (eg. `elecs(1)`) which has all its physical properties (the components of derived data type) placed contiguously in memory. This way, while accessing this AoS, the code will access particle by particle in memory and each particle will be loaded with all its components together. For example, if we have a simple structure which contains only the coordinates (X,Y,Z) of a particle in space, as shown in Listing 2.2; then an AoS

can be declared to store the positions of all particles, '`position_elecs`', as shown in Listing 2.2. For every particle, which is an element of this AoS, the three coordinates are stored contiguously in memory as shown in Figure 2.1. In Figure 2.1 every colour represents an element of this AoS. The blue colour represents the contiguous memory locations which store the coordinates of first particle in AoS, and similarly the other colours represent second, third and Nth particles of this AoS.

```
type position
    real(8)             :: X
    real(8)             :: Y
    real(8)             :: Z
 end type particle

!declaration of a particle type as AOS
type(position), dimension(:), allocatable :: position_elecs
!allocation of AOS
allocate(position_elecs(5000))
```

Listing 2.2 Example data structure 'position' and corresponding array of structure (AoS) for representing electrons' coordinates

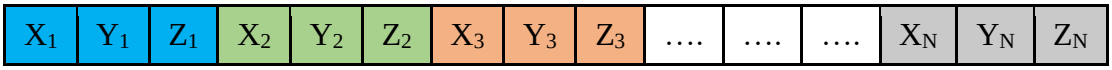


Figure 2.1 Contiguously stored particle coordinates (X,Y,Z) for the AoS shown in listing 2.2. 'N' represents the last particle index which is 5000 as shown in listing 2.2

It was observed with our tests that using dynamic arrays as the components of a derived data type (Listing 2.3) is more beneficial for LPPic2D, as in most of the compute intensive loops only some of the components (eg. only velocities) of the derived data type are used. In such cases, having arrays for individual components leads to vectorized loops with less memory stalls and thus efficient use of SIMD capabilities of the processors. This format of derived data type is termed as structure of arrays (SoA), where each type of particle is defined as a distinct object of type '`particles_SOA`'. Each component of this SoA is a dynamic array, which represents different particle properties as shown in Listing 2.3.

```
!definition of a particle type
type particles_SOA
    integer,dimension(:),allocatable    :: index
    integer,dimension(:),allocatable    :: charge
    real(8),dimension(:,:),allocatable  :: V
    real(8),dimension(:),allocatable    :: X
    real(8),dimension(:),allocatable    :: Y
    real(8),dimension(:),allocatable    :: Z
end type particle

!declaration of a particle type as SOA
type(particle_SOA) :: elecs
!allocation of some components of SOA
allocate(elecs%index(5000))
allocate(elecs%V(5000,3))
allocate(elecs%X(5000))
```

Listing 2.3 Example data structure 'particles_SOA' defined as a structure of arrays (SoA), and corresponding declaration and allocation of a particle SoA 'elecs'

The memory layout of an SoA is shown in Figure 2.2 with our previous example of structure which contains only the coordinates of a particle. The corresponding SoA is shown in Listing 2.4. This SoA has the three coordinates (X,Y,Z) as its components, which are individual dynamic arrays. The size of each array is equal to the number of particles represented with this SoA (eg. 5000 in Listing 2.4). With this SoA approach, each component of the structure is accessed as an array (contiguous memory access) for all the particles of the respective particle type. So, as shown in Figure 2.2 the individual coordinates X, Y or Z will be accessed together for all particles.

```
type position_SOA
    real(8),dimension(:),allocatable    :: X
    real(8),dimension(:),allocatable    :: Y
    real(8),dimension(:),allocatable    :: Z
end type particle

!declaration of a particle type as SOA
type(position_SOA) :: position_elecs
!allocation of SOA
allocate(position_elecs%X(5000))
allocate(position_elecs%Y(5000))
allocate(position_elecs%Z(5000))
```

Listing 2.4 Example data structure 'position_SOA' and corresponding SoA object 'position_elecs' for representing electrons' coordinates

In LPPic2D, several kinds of physical variables were defined with different array of structures including electrons, positively charged species, electric and magnetic fields, grid parameters etc. We started changing the AoS data to SoA format from electrons and progressed with other variables in the code with a step-by-step approach. In the original code, we had 40.65% of total memory pipeline slots that stalled in our test case as measured with Intel APS. Table 2.1 shows that as we progressed with the SoA in the code, the percentage stalling of memory pipelines decreased significantly. The impact of the SoA conversion in memory pipeline stalls for three main data objects: electrons, species and tabgrid is shown in Table 2.1. Some grid and domain properties are defined in the 'tabgrid' SoA.

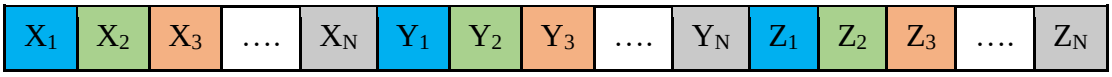


Figure 2.2 Accessing individual arrays from a structure of arrays (SoA) data type

Table 2.1 Percentage fall in memory stalls with increasing use of SoA

| Stage of SoA conversion | % of pipeline slots which stall |
|---|---|
| Original code (AoS) | 40.65 % |
| SoA (electrons) | 23.9 % |
| SoA (Species) | 19% |
| SoA (Tabgrid) | 17.68 |

It should also be noted that the number of memory operations (allocations, deallocation, number of copy calls etc.) have increased with the SoA version. In the AoS version, these memory operations were performed for the whole array of structure just once, and in the SoA version individual operations have to be performed for each attribute of the particle structure, which are defined as dynamic arrays. This aspect is clearly visible with allocation operations shown in all the code listings presented in this section above. Similarly, with the SoA version the number of MPI calls for exchanging particle data has also increased, however, the amount of data exchanged via MPI calls remains the same and we have the flexibility to send only the required arrays of an SoA. It is difficult to keep track of the performance impact of every modification in code, however, these aspects need to be further analysed. The overall quantitative results on performance of optimized LPPic2D are discussed in Section 3 of this article.

## 2.2 Hypre Library optimizations

LPPic2D uses the Hypre library [2] for solving the Poisson equations encountered while simulating the plasma dynamics. The research team used an older version of Hypre library available on the Occigen machine. The supercomputer 'Occigen' consists of about 86000 cores of Intel Haswell and Broadwell processors; and its peak theoretical floating-point performance is 3.5 PFlop/s [8]. A more recent version (first `V2.18` and finally `V2.20`) of Hypre was installed on Occigen to get some performance improvement by using versions that are recent. For using the Hypre library, some general steps are recommended by the developers [2]. From code's development point of view, mainly three major stages can be identified as 1) setting up the solver, 2) calling the solver and 3) deallocating the solver.

The first stage is mainly the initialization phase in which we choose and prepare the conceptual interface between the code and the external Hypre library. This stage comprises of setting up the required grid structure, defining the

discretization stencil, setting up the matrix of equation coefficients (A), setting up of vectors (B and x), defining the solver with all its numerical and simulation related parameters etc. In the second stage, the defined solver is called with required input information to solve the linear system of equations. Finally, at the end of simulations, the deallocation functions of Hypre are called to destroy all allocated objects (grid, matrix, solver etc.).
In LPPic2D, the subroutine calls which setup these three Hypre stages for the code were not called at correct locations. Ideally, all the function/subroutine calls for setup should be called only at once in the beginning of simulations. In LPPic2D, these setup calls were made every time the Poisson subroutine was called which was not required. In addition, the setting up of solver was also performed at each iteration in the solver stage (2nd stage) which should have also been done only once. We optimized the subroutine calls for these 3 stages by placing them at only the necessary stages in the code. In the optimized code, we perform the setup stage only at the beginning of the simulation and similarly the deallocation is done at the end of the simulation. The function calls related with setting up of solver parameters were also removed from the solver part and we put them in the overall setup stage of the Hypre library. Important gain in execution time was observed with these optimizations in the setup of Hypre library calls.
The Hypre library has several options for solvers and preconditioners for solving different linear systems of equations. Depending on the structure of the equation matrix (sparse, diagonal, triangular etc.), different combinations of solvers and preconditioners give varied performances. In several cases, the use of suitable preconditioners transforms the equation matrix in a more suitable format to improve the overall solver efficiency. The research group had already conducted a study on some solvers of Hypre without extending to use preconditioners. During this HLST project, we extended this study to include several other possible combinations of solvers and preconditioners with code LPPic2D.
Profiling LPPic2D shows that the Poisson solver part in the code is one of the hotspots that takes a significant fraction of the overall execution time. Some initial tests performed with Hypre library's solvers suggested that PFMG was the most time efficient solver for solving LPPic2D's Poisson equation [9]. We started our work with PFMG, and our profiling results with Intel APS and Score-P suggested that more than 50% of execution time (He BM case) was spent in MPI library calls, and most of these MPI calls were made inside the Hypre library's functions. It was also noted that MPI_WAIT type calls inside Hypre library were taking significant time (~ 15%) out of overall MPI library time. This observation prompted us to think that parallel communications in Hypre's PFMG functions were not as efficient as desired.
It was also noted that tests on solvers' performances reported by [9] did not include any cases with pre-conditioners. Preconditioning is a manipulation of the original system of equations to improve its spectral properties with a preconditioning matrix. The rate of convergence of iterative methods depends on the spectral properties of the iteration matrix, which can be improved with preconditioning. Their study included tests with two Krylov solvers: GMRES and BiCGStab. A recent publication reported a comparison study on preconditioned Krylov subspace methods for large-scale non-symmetric linear systems [10]. Based on this article, we mainly tested GMRES and BiCGStab solvers with different preconditioners available with Hypre library.

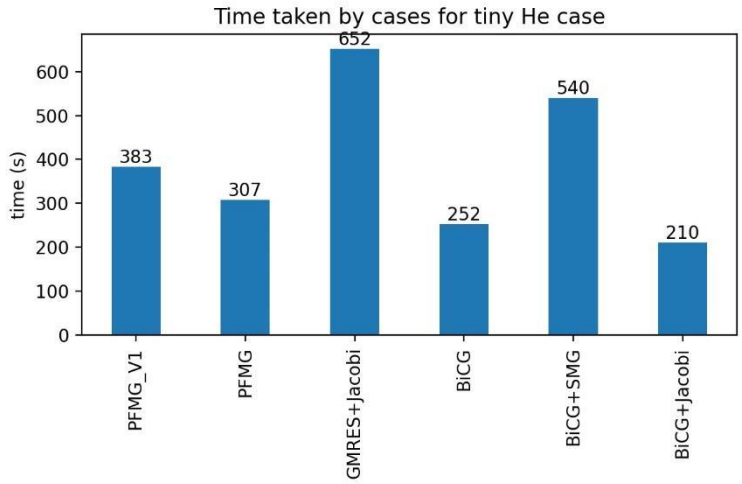


Figure 2.3 Time taken by various Hypre solvers with 'Tiny He' benchmark case

Initial tests of validation were performed with 'Tiny He' benchmark case, which was our smallest benchmark case. The combination of solvers and preconditioners that showed some gain in execution time were then tested with the Helium (He) benchmark (medium sized benchmark case) and also z-theta (larger benchmark) cases. All the test runs were duly validated against the reference physical results.

Figure 2.3 shows the results obtained with the 'Tiny He' benchmark case. The first bar (PFMG_V1) represents the original version of LPPic2D before any optimizations in the source code. The rest of the bars represent cases with different Hypre solvers on optimized source code. It was observed that by using BiCGStab as solver and Jacobi as a preconditioner the solution time has significantly reduced (right most bar). Similar behavior was observed with the Helium benchmark case also. Some solver and preconditioner combinations also took significantly more time than the PFMG solver. Especially with GMRES and its combinations with preconditioners, we did not notice any gain. The z-theta case was also run with the combination of BiCGstab and Jacobi preconditioner. However, none of the combinations used in our test cases showed any gain with the z-theta case. Plots of execution time with these combinations for z-theta and He benchmark cases are provided in Appendix B of this article. Further analysis with various profiling tests is required to understand the observed behavior with z-theta and He benchmark cases.

## 2.3 General optimizations for improving vectorization

The code was updated at several locations to improve the vectorization of bottleneck loops. Intel tools like Vtune, Advisor, etc. were used to profile the code at different stages to find these bottleneck parts of the code. Firstly, the relevant compiler optimization flags *(-O3 -xCORE-AVX2 -simd -align array64byte -fpp -real-size 32 -integer-size 32 -fp-model fast=2 -fast-transcendentals)* were added in the `cmake` file. The research group was using only -O2 as the optimization level, but with aggressive compiler optimizations also the physical results were validated.

There are expressions that consume more CPU cycles, time, and memory. These expressions should be replaced with cheaper expressions without compromising the output of the expression, which is called strength reduction. Strength reduction was performed for several computationally expensive expressions in the code by replacing these expressions with computationally cheaper expressions. Some of these strength reduction operations include replacing division by relevant multiplication, computing constant expressions only once out of the loops, execution of the expression only by relevant MPI processes and not by all etc. In several parts of the code, the ordering of operations was modified to vectorize as many loops as possible. The order of the several loops was modified to have contiguous memory access for vectorization. Similarly, for some multidimensional arrays the indexing of elements was optimized to access elements contiguously in memory as required by the loops directions *(i,j,k)*. OpenMP pragmas for SIMD vectorization of loops were also added in some bottleneck parts of the code to improve the vectorization.

It was observed that the code was writing a lot of debug info in output files even in production runs, which was found unnecessary after discussing with the research team. With the help of research team, modifications were made at several locations in the code to add pre-processor directives to avoid compiling the debug related parts in the production runs. Thus reducing the size of the compiled code for production runs. For the overall optimization of the code LPPic2D, several other improvements were made following the good Fortran programming practices like explicit use of intent attributes for defining dummy variables, loop labelling as required, correct use of global and local variables, using correct formats of double precision constants (1.d0 and not 1.0) to avoid unnecessary data type conversions etc. Table 2.2 lists the general optimization performed on LPPic2D.

Table 2.2 Some general optimizations performed on LPPic2D

| | |
|---|---|
| 1 | Optimization with compiler flags |
| 2 | Strength reduction |
| 3 | SIMD vectorization |
| 4 | Minimizing data type conversion operations |
| 5 | Loop index order modification for optimized memory access |
| 6 | Array index modification for optimized memory access |
| 7 | Use of optimized Fortran features |
| 8 | New pre-processor directives |

## 2.4 Optimization of MPI communications

After the general optimizations, parallel message exchanges were targeted. It was noticed with the profiling reports that above 50% of execution time was spent in MPI library calls in the original version of LPPic2D. Most of it was spent in MPI_WAITALL and MPI_SENDRECV calls. Significant portion of MPI imbalance was also noticed in Hypre library's internal MPI communications, as suggested by the profiling reports generated with Score-P. For

most of the MPI exchanges, the message tags were hardcoded in the original code, which is not conducive for safe MPI exchanges and may produce errors in certain cases while scaling the application on thousands of cores.
During our optimization work, all the hardcoded MPI message tags were replaced with explicit tags for every MPI communication (send-receive). These explicit tags were generated with simple functions having both sender's and receiver's MPI ranks, along with their multiplication with small prime numbers (3 & 7) to ensure explicit integer numbers for each MPI message and keeping the tag same for both sender and receiver for that message. With these explicit tags, the MPI message envelope for a message follows the MPI standard's requirement of matching tags for both sender and receiver; and there is never a same tag for two different messages. With LPPic2D, we observed numerical errors with hardcoded tags while running on more than 5 AMD-Rome nodes (640 cores), and with explicit tags simulations were tested up to 60 nodes of AMD-Rome (7680 cores).
The use of non-blocking MPI message calls (MPI_ISEND and MPI_IRECV) was also optimized in the code. In the original code, the buffers used with non-blocking calls were updated immediately, with MPI_WAITALL call, just after the non-blocking send and receive calls were made. Thus using the non-blocking calls almost like blocking calls. In this way, no other operations were performed by MPI processes until the receive buffer was updated with the sent message. We realized that there was a good scope for overlapping some computations with communications. We updated the relevant parts of the code by shifting the MPI_WAITALL to lines just before the buffers were reused, and thus providing the scope of performing other operations to processes between non-blocking send and finally updating the receive buffers. For this, the subroutine, which was used to make the non-blocking send and receive calls along with MPI_WAITALL, was divided in two parts by shifting the MPI_WAITALL call to a new subroutine.
These new strategies to optimize the MPI communications in the code were successfully tested and validated for more than 7000 cores on both AMD-Rome and Intel Broadwell nodes. The impact on overall scalability is visible with the scalability results, which are discussed in the next section of this report.

## 2.5 Parallel I/O optimizations

LPPic2D utilizes the HDF5 library [11] for parallel I/O operations. HDF5 works on the underlying MPI-I/O for managing the parallel I/O of the application. Along with the features of the underlying MPI-I/O, HDF5 also manages the metadata of applications, such as the datatypes themselves, dimensionality of an array, names for specific objects, etc. The two popular ways to perform parallel I/O are collective and independent MPI-I/O. Depending on the requirements of the applications, both strategies, collective and independent, can be used for getting performance with the overall I/O operations. In LPPic2D, only collective MPI-I/O was available with HDF5.
During our work, we also implemented independent MPI-I/O in LPPic2D. This new feature was made available for the users as an option along with collective I/O (Listing 2.5). In cases where only a modest number (several hundreds) of MPI processes are used for the simulations and thus for the parallel I/O, the independent I/O with HDF5 can give a performance boost. With independent I/O, some costly collective operations are avoided and individual processes can write or read their data in the same file. This feature was not extensively explored for the typical production use cases of LPPic2D, however, the users are advised to explore this feature for benefitting from this I/O strategy when it is suitable.

```
#ifdef HDF5_COLLECTIVE_IO
    CALL h5pset_dxpl_mpio_f(plist_id, HDFD_MPIO_COLLECTIVE_F, error)
#else
    CALL h5pset_dxpl_mpio_f(plist_id, HDF5_MPIO_INDEPENDENT_F, error)
#endif
```

Listing 2.5 Code lines for updates in parallel I/O with HDF5

In some production cases, huge files of several 10s of GBs are generated with LPPic2D. We did some performance tests with Lustre striping for optimizing further the use of parallel file system on Occigen's `/scratch` space (Lustre FS). In our test cases, using a striping count of 30 to 40 showed an important fall in execution time with respect to default striping count of 1. The relevant information regarding the use of striping on big files was provided, and explained, to the users to benefit from this technique. An example of striping command used during our tests:

```
$ lfs setstripe -c 30 simulations/rundebug/data/
```

# 3 Scalability results

Different physical cases of plasma dynamics were validated with the optimized code; and scalability and speedup plots were produced. Figure 3.1 shows the increasing speedup at various intermediate optimization stages with respect to the original version of the code. This first plot was generated with the Helium benchmark case on 2 nodes. The leftmost bar represents the base value of speedup, which corresponds to the original code on 2 Haswell nodes (HSW). It can be inferred from the plot that the speedup was progressively increased, following various optimizations explained in previous section and using different Hypre solvers, to 2.5 times with respect to the original code for this smaller physical case. The last bar plot in Figure 3.1 represents the run on 2 Broadwell (BDW) nodes with SMG solver of Hypre library. In Figure 3.1, the caption codes PFMG, BiCG, SMG represent different linear system solvers used from Hypre library and code 'H2.xx' represent the Hypre library versions used for the corresponding cases.

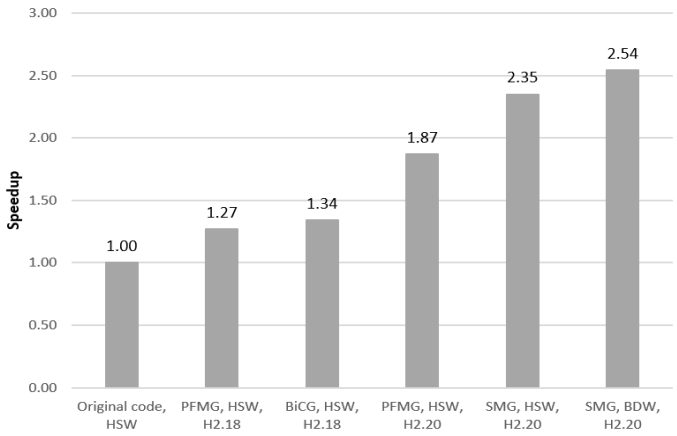


Figure 3.1 Speedup gain with respect to original code on He benchmark case with progressive optimizations and different Hypre library versions

Scalability tests with a large case were undertaken after the encouraging initial results. A z-theta case with grid parameters `ymax=500, xmax=2047` and 300 particles per cell (initial value) was simulated with a different number of nodes as seen in Figure 3.2. Time taken on 5 nodes was considered as the baseline case for this study (leftmost bar). Execution time was noted at two stages of the simulation 1) at 0.125 % completion and 2) at 0.25 % completion of the total number of loops asked. We used these in-built metrics (execution time etc.) of LPPic2D, which are used by the developers themselves for tracking the progress of the simulations. It was noted during the tests that the execution time of these initial stages was representative of the overall execution time of a production level simulation for this type of comparative scalability study. Our base case of comparison, which was a run on 5 nodes, took about 4 hours that is sufficiently long run to capture the behavior of a full simulation. It can be seen from Figure 3.2 that by increasing the number of nodes from 5 to 80, a speedup of around 11 was achievable for this case.

Simulations of Figure 3.2 were carried out on Intel Broadwell nodes (28 cores per node) of CINES. The same case was run with the original code to check the impact of optimization work on this case. A comparative speedup plot was generated considering the runs with original code, at different number of nodes, as the baseline cases (speedup 1). Figure 3.3 shows the speedup of optimized code with respect to the original code until 80 nodes. It is observed that the optimized code appears to deliver an average speedup of around 1.8 with respect to the original code, showing a tendency to increase this speedup with higher number of nodes and reaching 2.34 for 80 nodes.

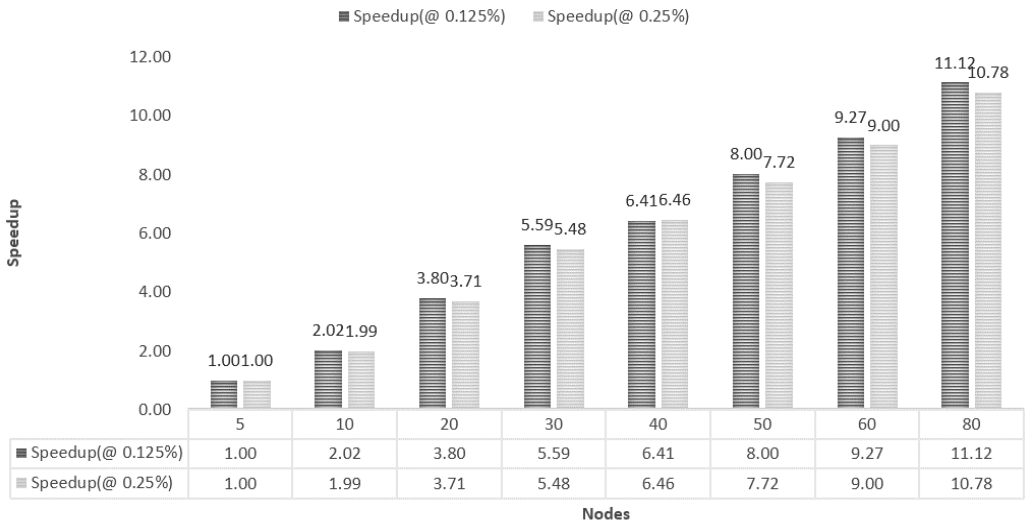


Figure 3.2 Speedup plot with z-theta case up to 80 Intel Broadwell nodes

The relative gain in speedup is shown in Figure 3.4 for this study. At this point, it should be noted that the visible limitation with scalability for higher than 80 nodes corresponds to the limitation of the scalability of the Poisson solver (Hypre library) with such grid sizes. Scalability was observed to be limited to 30 nodes for a grid size of 255*500 in some earlier cases. It can be stated with confidence that the scalability will improve with higher grid sizes, and the research group could attempt the cases with 2000*2000 grid sizes with higher number of nodes in near future.

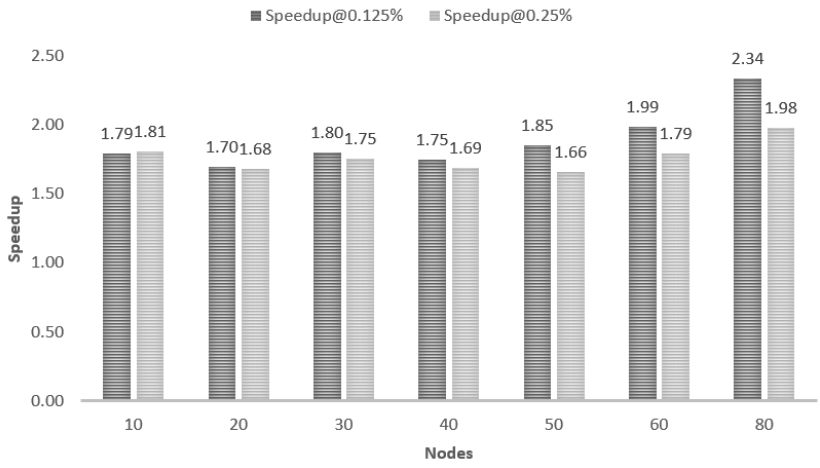


Figure 3.3 Speedup of optimized code with respect to the original code (speedup 1) at two stages (0.125 % and 0.25% completion of loops).

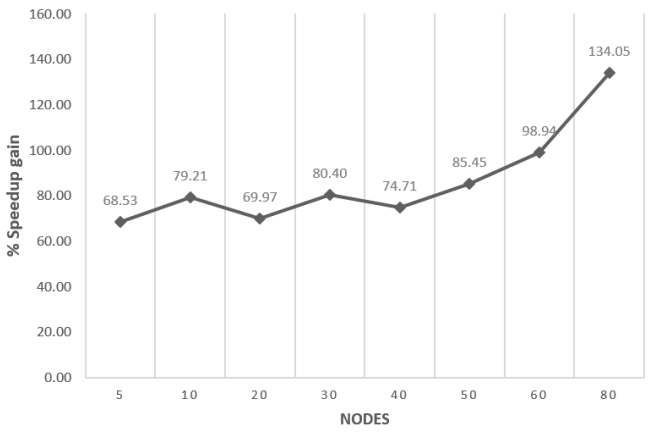


Figure 3.4 The relative speedup with respect to the original code for 0.125 % completion stage

A parallel efficiency plot with this study is shown in Figure 3.5. It is observed that the optimized code has an efficiency of 75 % until 60 nodes for this case. As mentioned above, the gap in efficiency is increasing between the original and optimized code as we go for higher number of nodes. Figure 3.6 compares the speedup with the ideal case. The speedup with original code seems to have reached a threshold (speedup = 8) for the case at around 50 nodes, however, the optimized code still shows an upward linear trend up to 80 nodes (speedup = 11) as observed in Figure 3.6. Even with the optimized code, the test case is showing limitations with strong scalability above 60 nodes; and considering that larger test cases have been suggested to users of LPPic2D for running simulations on higher number of nodes (>60).

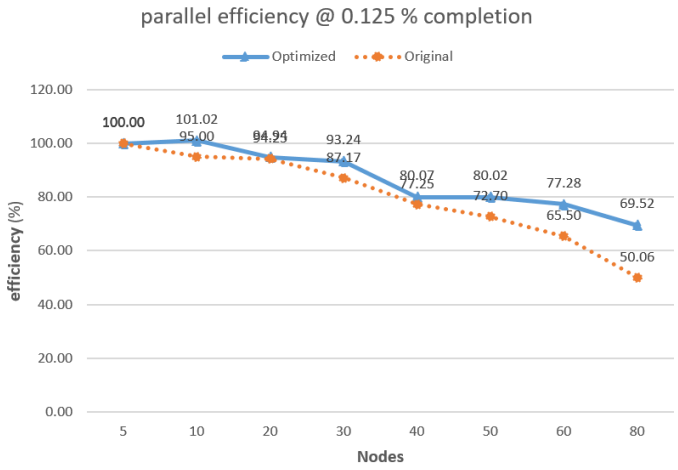


Figure 3.5 Parallel efficiency for optimized and original code

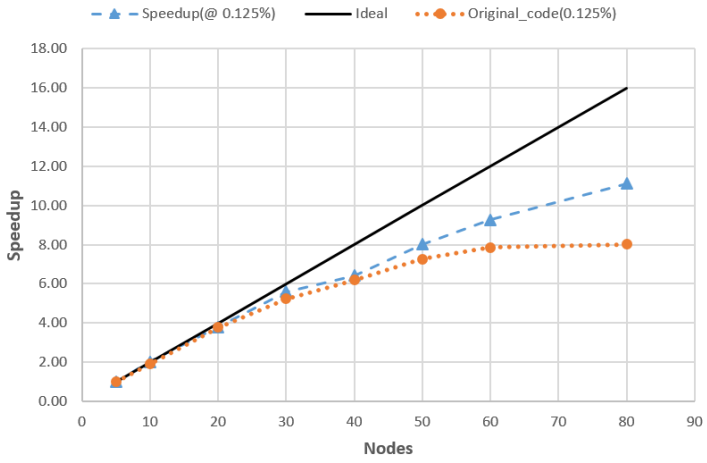


Figure 3.6 Parallel speedup comparison of optimized and original code with an ideal case.

It can be inferred as a general conclusion, from all the above results, that for such large grid cases, running on higher number of nodes (> 50), the optimized version of LPPic2D would require at-least 40% less execution time than would be required by the original version of the code.

# 4 Scalability results on AMD-Rome

With the aim of running LPPic2D on tier-0 supercomputers of Europe, a scalability study was carried out on AMD-Rome nodes of the Irene system at TGCC. Each AMD-Rome compute node on Irene has 128 physical cores [12]. At the initial stage, there were some troubles while running cases on more than 5 AMD-Rome nodes when compiled with Intel compilers and relevant libraries available on Irene. Finally, a combination of *gcc/8.3* compiler with *openmpi/4.0.2* library was used for the scalability tests. Figure 4.1 shows that LPPic2D scaled as desired until 60 AMD-Rome nodes for the 0.125 % completion of simulation.

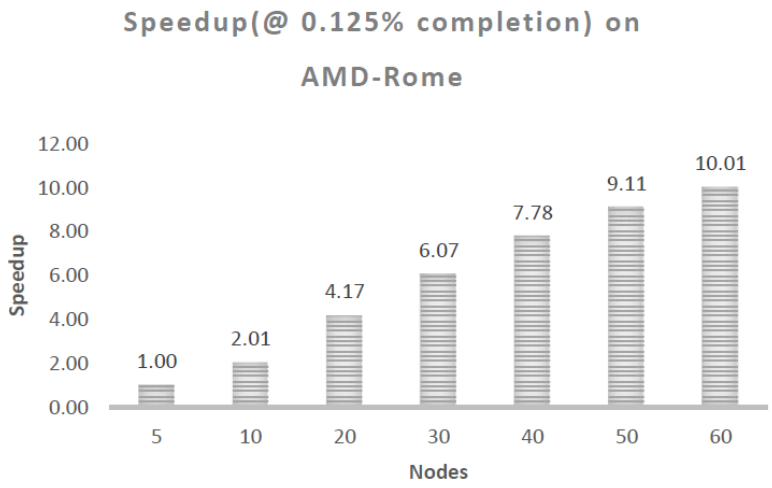


Figure 4.1 Speedup attained for 0.125 % of simulation completion on AMD-Rome nodes

This first study on a different hardware architecture, other than Intel, also verified the portability of LPPic2D on AMD processors. We wanted to carry out this study on a matching Intel software environment to have a better comparison on both hardware architectures, but finding a matching software environment on two different system of different generations is difficult. On CINES's system 'Occigen' we had Intel compiler and Intel MPI library which was an optimized environment for Intel hardware, however, on system Irene we had to contend with the GNU compiler and the Open MPI parallel communication library environment because within the project time we did not manage to scale LPPic2D on more than 5 AMD-Rome with Intel software environment. However, for a first portability study it was still interesting to compare the performance on these two different architectures. Figure 4. 2 shows a speedup on Rome nodes when the same Broadwell runs were considered as the base case from 5 to 60 nodes. It can be stated that on Rome nodes LPPic2D runs roughly 3 times faster than Broadwell nodes when node-to-node performance is compared for the long z-theta case.

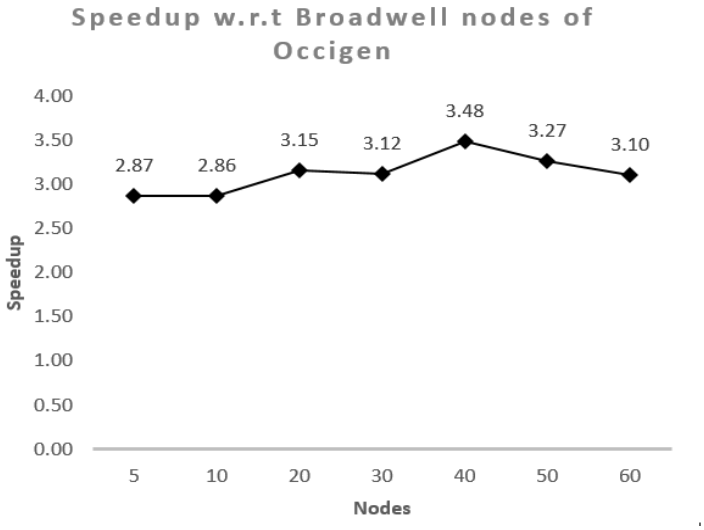


Figure 4. 2 Speedup gain on AMD-Rome nodes with respect to Intel-Broadwell nodes

It should be noted here that in terms of number of cores, we have about 4.5 times more number of cores on an AMD-Rome node (128 cores) of Irene than on Intel Broadwell node (28 cores) of Occigen. Considering only the increased number of cores, we should have obtained a speedup of more than 4 times on AMD-Rome, however, as mentioned above the software environment also was completely different on both systems and we believe that played an important role in the overall performance achieved with our LPPic2D cases. Another important point to note here is that the AMD-Rome processors are more energy efficient than Intel Broadwell processors. Irene has an energy efficiency of 4.87 GFlops/watt in comparison with Occigen's 1,74 GFlops/watt as reported on the GREEN500 list, which officially tracks the energy efficiency and other metrics of the most powerful supercomputers of the world [13]. We believe that with our work on LPPic2D we have also taken first steps for advancing this code towards a more energy efficient system. As the very first study for LPPic2D, on AMD-Rome processors, it is still encouraging to observe these results. This study should definitely serve as a base case for requesting computational resources on AMD processors. Further exploration would continue both with production runs and other relevant studies on the performance side on AMD-Rome nodes.

# 5 Conclusion

Considering the significant consumption of computational resources in recent years and the scope available for optimization, code LPPic2D was selected for optimization by the HLST team at CINES. The scalability plots provided with the previous DARI applications by the group suggested that the code's scalability was limited to around 20 Haswell nodes for the case considered.
The initial tests by the HLST team showed that the extensive use of several Array of Structure (AoS) data types was one of the key characteristics of the code which led to inefficient memory access patterns; and thus limiting the overall scalability of the code.
The steps taken for the overall optimization of the code include

- improving memory access with Structure of Arrays (SoA),
- SIMD vectorization,
- non-blocking MPI communications,
- efficient implementation of Hypre library,
- HDF5 and I/O optimizations
- Other general optimization mentioned in previous sections .

As explained in the report, for larger grid cases, running on higher number of nodes (> 50), the optimized version of LPPic2D would require at-least 40% less execution time than required by the original version of the code. This would mean that for a project, which currently needs ~ 12 million CPU-hours on Occigen (T1 at CINES), would now need less than 8 million CPU-hours.
Significant gain with scalability shows the optimized code scales with 1.6x more number of nodes than the original code; and this gain is more pronounced with larger grid cases running on higher number of nodes. Test simulations were carried out on AMD-Rome processors also and the scalability results until 60 nodes (7680 cores) showed about 84% speedup of the ideal case. The research team will continue further exploration on AMD processors, with both production runs and other relevant studies on the performance side.
The overall gain on this project is significant. The simulation time has been greatly reduced and the scalability of the application has been greatly improved. On top of that, the code has been ported and tuned on Joliot-Curie T0 systems on its AMD Rome partition, which is a much more energy efficient system than Occigen. As far as we know, the research team will now be able to run simulations that will be 10 to 100 times bigger than their current models and make the most of the PRACE T0 system available at TGCC.

## References


[1] https://www.lpp.polytechnique.fr/-Plasmas-pour-la-propulsion-spatiale-?lang=fr
[2] https://hypre.readthedocs.io/en/latest/ch-intro.html
[3] https://dci-gitlab.cines.fr/veille/lppic (on internal server at CINES)
[4] https://www.intel.com/content/www/us/en/developer/articles/release-notes/intel-vtune-amplifier-2018-release-notes.html
[5] https://www.vi-hps.org/projects/score-p/
[6] https://www.intel.com/content/www/us/en/developer/articles/release-notes/intel-advisor-2018-release-notes.html
[7] https://www.intel.com/content/www/us/en/develop/documentation/application-snapshot-user-guide/top/introducing-application-performance-snapshot.html
[8] https://www.cines.fr/calcul/materiels/occigen/configuration/

[9] Modélisation bidimensionnelle de la décharge plasma dans un propulseur de Hall, Vivien Croes, Ph.D thesis, 2017
[10] A comparison of Preconditioned Krylov Subspace Methods for Large-Scale non-symmetric Linear Systerms, Aditi Ghai, Cao Lu, Xiangmin Jiao, 2018
[11] https://portal.hdfgroup.org/display/HDF5/HDF5
[12] http://www-hpc.cea.fr/en/complexe/tgcc-JoliotCurie.html
[13] https://www.top500.org/lists/top500/2022/06/

## Acknowledgements

This work was financially supported by the Members of PRACE aisbl and by the PRACE project funded in part by the EU's Horizon 2020 Research and Innovation programme (2014-2020) under grant agreement 823767. The authors also acknowledge the support and participation of LPPic2D developers at every stage of the project.

# Appendix A

Intel APS report for He benchmark case with AoS versions of LPPic2D (original code) which shows significant memory pipline stalls (~40%) and % time passed in MPI library calls. These two types of bottlenecks were analysed and code was accordingly optimized to reduce the impact of these factors.

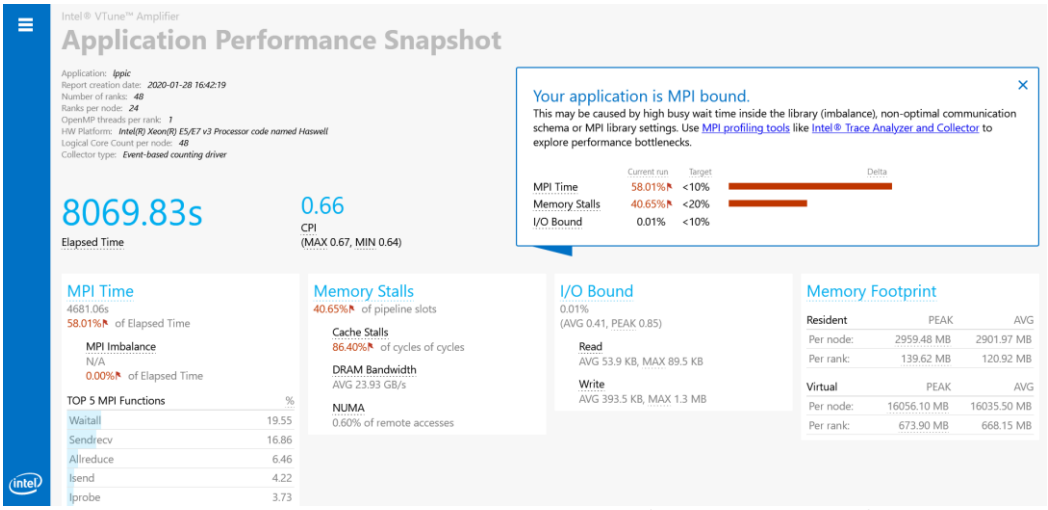


*Fig. A.1 Intel APS report with original code having Array of Structure (AoS) as data format*

Tests with various Hypre library solvers on our He benchmark (medium sized case) and z-theta (larger test case).

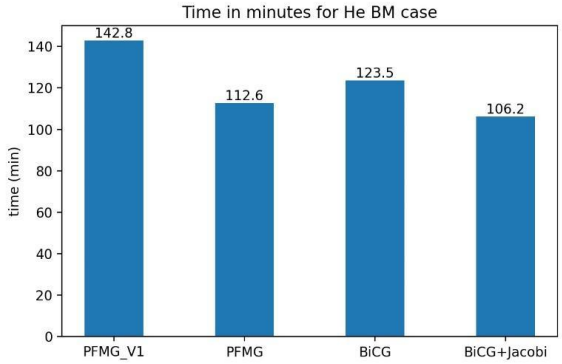


*Fig. A.2 Time taken by various solvers with He benchmark case*

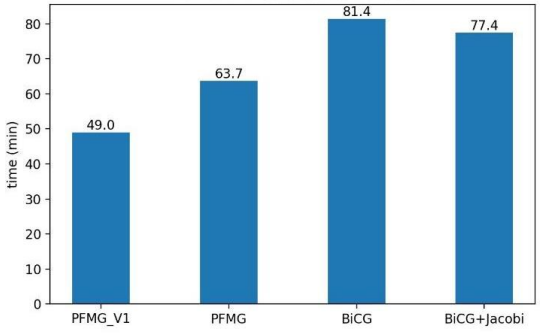


Fig. A.3 Time taken by various solvers with z-theta case, for 5 loops

The article [10] concluded that with Krylov solvers like GMRES and BiCGStab the algebraic solver BoomerAMG performs very efficiently, however, this preconditioner is not available for structured grid solvers in Hypre's version which we were using. Some tests were also carried out with Hybrid solver of Hypre that did not work in this attempt; further tests would be required with this solver.