The Journal of Supercomputing (2024) 80:12021–12042
https://doi.org/10.1007/s11227-024-05896-2

On the parallelization of multipacting simulation codes for the design of particle accelerator components

Javier Navaridas[1] · Jose A. Pascual[1] · Julen Galarza[1] · Txomin Romero[2] · Juan L. Muñoz[3] · Ibon Bustinduy[3]

Accepted: 4 January 2024 / Published online: 7 February 2024

Abstract

Particle trajectory and collision simulation is a critical step of the design and construction of novel particle accelerator components. However it requires a huge computational effort which can slow down the design process. We started from a sequential simulation program which is used to study an event called *Multipacting*. Our work explains the physical problem that is simulated and the implications it can have on the behavior of the components. Then we analyze the original program's operation to find the best options for parallelization. We first developed a parallel version of the Multipacting simulation and were able to accelerate the execution up to $\sim 35\times$ with 48 or 56 cores. In the best cases, parallelization efficiency was maintained up to 16 cores ($\sim 95\%$) and the speed-up plateaus at around 40–48 cores. When this first parallelization effort was tried for multi-power simulations, we found that parallelism was severely limited with a maximum of $20\times$ speed-up. For this reason, we introduced a new method to improve the parallelization efficiency for this second use case.

✉ Javier Navaridas
javier.navaridas@ehu.eus

✉ Jose A. Pascual
joseantonio.pascual@ehu.eus

Julen Galarza
jgalarza006@ikasle.ehu.eus

Txomin Romero
txomin.romero@dipc.org

Juan L. Muñoz
jlmunoz@essbilbao.org

Ibon Bustinduy
ibustinduy@essbilbao.org

1 Department of Computer Architecture and Technology, University of the Basque Country UPV/EHU, 20018 Donostia, Gipuzkoa, Spain

2 Donostia International Physics Center, 20018 Donostia, Gipuzkoa, Spain

3 ESS Bilbao, Edificio 207-B, 48160 Derio, Biscay, Spain

This method uses a shared processor pool for all simulations of electrons (OnePool). OnePool improved scalability by pushing the speed-up to over 32×.

Keywords Multicore systems · Multipactor effect · Parallel programming · Particle simulation

1 Introduction

This paper presents the improvement and optimization of a finite differences particle simulation program used by ESS Bilbao to design and manufacture some essential components of, among other particle accelerator technology projects, the European Spallation Source project [1] (hereafter ESS). ESS Bilbao is a strategic center of international reference in Neutronic Technologies that provides knowledge and added value through in-kind contributions to the European Spallation Source, a singular scientific infrastructure that uses nuclear spallation, a process in which neutrons are liberated from heavy elements by impacting them with high energy protons. This source of neutrons is critical for many domains of research spanning the realms of physics, chemistry, geology, biology and medicine. See Fig. 1 for a diagram of the installation. The ESS is being built in the Swedish city of Lund through an exceptional international effort and will support research on a wide range of topics. In addition to this, our partners also work on the construction of their own particle accelerator [2], which will facilitate regional research and development in high energy physics. In order to accelerate protons to near light speeds, these particle accelerators are made up of a wide range of components [3–6] to deal with progressively higher energy particles. In addition, the design includes many electronics for monitoring, control and diagnostics of the beams [7]. Figure 2 shows a diagram with the different sections of the accelerator hardware.

An important step when designing and building these components is to simulate the behavior of the particles inside them to ensure they behave as expected and that no pathological effects will occur in the accelerator component. In this work, we have worked with one of the programs used for these simulations, which is mainly used to analyze a crucial physical process called *Multipacting*.

The *Multipactor effect* [8] or, simply, Multipacting, is an avalanche discharge of electrons generated by the synchronization between the intense alternating electric field used to accelerate the particles and the emission of secondary electrons (secondary electron yield or, simply, SEY) on surfaces exposed to electrons accelerated by this field in vacuum conditions. It begins when an electron that is inside one of the mentioned components collides with one of the surfaces of the component and as, a consequence of the collision, it gets absorbed by the surface, but the energy of the impact rips out other electron(s) from the surface. These electrons, in turn, can collide and pull out even more electrons. If this event is repeated, a cloud of electrons can be created, forming what is called multipacting or multipactor effect. Analyzing the conditions under which this avalanche of electrons is most likely to form in each component can be of utter importance because during regular operation it is essential to prevent it from happening. Otherwise, it could influence the particles in the accelerator, which in

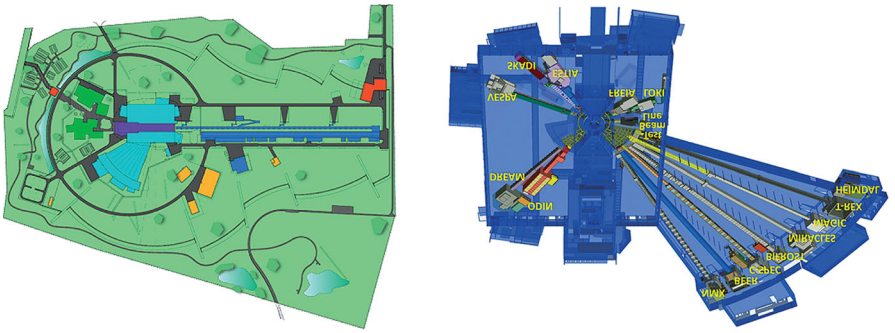

Fig. 1 Structure of the European spallation source. *Left* the ESS site layout, showing accelerator buildings (dark blue), target (violet), instrument halls (light blue), offices and labs (green) and auxiliary buildings (red and yellow). *Right* the positions of the first 15 neutron instruments: diffractometers for hard-matter structure determination (DREAM, HEIMDAL and MAGIC), macromolecular crystallography (NMX) and engineering studies (BEER); small-angle scattering instruments for the study of large-scale structures (LOKI and SKADI); reflectometers for the study of surfaces and interfaces (ESTIA and FREIA); spectrometers for the study of atomic and molecular dynamics (BIFROST, C-SPEC, MIRACLES, T-REX and VESPA); and a neutron imaging station (ODIN). Credit: European Spallation Source

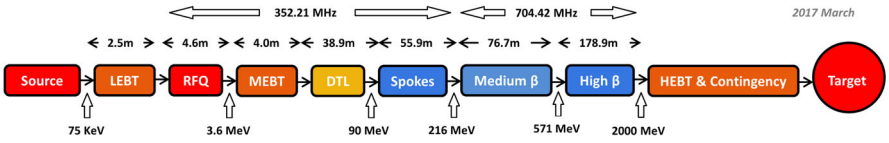

Fig. 2 Schematic of the different sections of the ESS accelerator. Regular components: Microwave Discharge Ion Source (MDIS), Low Energy Beam Transport (LEBT), Radio-Frequency Quadrupole (RFQ), Medium Energy Beam Transport (MEBT), Drift Tube Linac (DTL), High Energy Beam Transport (HEBT) and Rotating Target Wheel (Target). Superconducting components: Superconducting Spoke Resonator (Spokes), Elliptical Superconducting Radio-frequency Linear Accelerator (Medium-β **and High-**β)

turn affects the experiments that are being conducted, or even worse, it could damage equipment. However, on some occasions it might be desirable to produce the avalanche on purpose under controlled conditions, as it can be used for maintenance work such as, for instance, fine-polishing the surface of a component.

Nevertheless, if we want our experiments to be as reliable as possible, we need to simulate very large numbers of electrons and also perform numerous repetitions. For this reason, the execution time of a program of these characteristics that simulates the path of each one of the electrons in series can be very high. In this paper, we discuss the steps taken to parallelize the execution of the original program, so it can take advantage of the computing resources offered by the ATLAS supercomputer[1] at the Donostia International Physics Center. The ultimate objective is, of course, reducing the time-to-solution of this type of simulation to, in turn, streamline the design cycle of new components. Our aim is, therefore, to achieve the best possible speed-up to both reduce execution time and enable the execution of simulations with much larger amounts of electrons, more complex shapes and/or more refined meshes.

[1] More information available at: http://dipc.ehu.es/cc/computing_resources/systems/atlas-edr/.

To test our developments, we used the description of different components that are part of the design work of different components of the ESS. In total, we used three different meshes with varying complexity, from 20K to 900K polygons. In particular, this work discusses two approaches that were employed to carry out the parallelization. The original code employed by ESS was written as a Python wrapper for the FEniCSx library [9], a popular open-source computing platform for solving partial differential equations.[2] FEniCSx provides support for parallel execution through MPI. This parallelization capability is particularly optimized for solving the electromagnetism differential equations by the finite element method. However, in our simulator, the electric field distribution is computed once at the beginning of the simulation and, then, it is used as a background for tracking the electrons, which requires very fine-grain calculations from the library and would not benefit from parallelizing. Thus, using the MPI capabilities of the FEniCSx library can not be used in our advantage for speeding up simulations. Indeed, most of the heavy lifting of the computation is done outside the library and therefore a different approach is required. Employing parallelism at the electron level was found to be more adequate and, therefore, is the level of granularity we use in our parallel code. Since no communication between electrons is needed for the kind of simulations we are carrying out, a feature-rich *communication-oriented* programming model such as MPI was, indeed, considered unnecessary. For this reason, when we started developing the parallel versions of the multipacting simulation, we decided to rely on Python's Multiprocessing library [10], a lighter library with sufficient parallelization features for our needs.

Our first implementation, multiprocessing, simply runs all the electrons of each generation in parallel. This was found to provide more than adequate efficiency and speed-up for running single multipacting configurations; over $35\times$ speed-up for 40 cores in the best case, and averages of around $30\times$ for two of the three meshes. However, for longer simulations that explore a range of power values, the achieved speed-up was relatively limited; plateauing at around $15-20\times$. The reason for this is that the *naïve* parallelization approach was not able to keep all the available processors busy. In consequence, we decided to further develop the code and introduced another algorithm for these cases, where the processing resources are shared in a more balanced way. We finally introduced the OnePool algorithm, which uses the same pool of processors for running the electrons of all generations and all powers at the same time. OnePool demonstrated a much better scalability by pushing the speed-up to over $35\times$ and obtaining the highest speed-ups in most experiments.

2 Background

In this section, we first describe the Multipactor effect which the original code simulates. The description also includes a justification of why its occurrence can be problematic in an actual installation. Furthermore, we outline the operation of the simulator, providing the pseudocode for its three main operation modes.

[2] https://fenicsproject.org/.

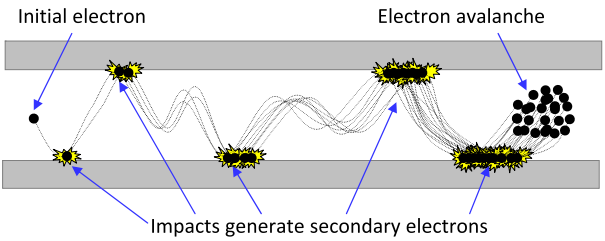

Fig. 3 Diagram of the multipacting effect. The gray boxes represent the surfaces of the RF component. The black circles represent electrons, with the dotted lines showing their trajectories. The yellow star shapes represent collisions that produce secondary electrons

2.1 Multipactor effect

The multipactor effect [11], or simply multipacting, is the phenomenon of resonant secondary emission multiplication in radio-frequency (RF) amplifier vacuum tubes and waveguides. It occurs when one or more electrons within the RF components of the particle accelerator generate an electron avalanche caused by the emission of so-called *secondary electrons*.

The collision of an electron with a surface can release one or several of these secondary electrons into the vacuum, depending on the energy and angle of the collision. These secondary electrons are then accelerated by the RF field and if they also collide with any surface they can release even more secondary electrons. Due to the multiplicative effect of each collision, the number of emitted secondary electrons can grow exponentially and may lead to operational problems of the RF system such as damaging the RF components or distorting the RF signal which, in turn, can skew or even completely invalidate the experimental results.

In the context of particle acceleration, where huge energy levels are employed, this effect can be catastrophic. Multipacting can manifest itself in the form of heat generated by the impacts of the electrons. In the most serious cases this heat can be sufficient to melt internal components, or to perforate vacuum walls, which will end up in disastrous consequences. Another way in which it can manifest itself is by failure of ceramic or glass windows at unexpectedly low power levels, due to the extreme charge or heat produced by the electron cloud.

Figure 3 shows an illustration of the effect. The diagram starts with a single electron which can be part of the experimentation or, more commonly, might be a rogue electron which has been torn off from the surface by the strong RF field. Upon collision with the lower surface, this initial electron strips off two secondary electrons. In turn, when these impact with the upper surface, four secondary electrons are emitted. In this example, each new collision doubles the total number of electrons, growing up to 8, 16 and 32, respectively, for the next collisions depicted. It is easy to see that the number of electrons grows exponentially and, thus, that it can run out of proportion very rapidly with a relatively small number of RF cycles. For simplicity, the diagram assumes that each collision produces *two* secondary electrons. In practice, this figure can be as high as *six* secondary electrons and, indeed, is not constant. In fact, the number of emitted electrons depends on many factors such as the energy of the incident electron, the

angle of impact, the frequency and power of the magnetic field, the temperature, the material the component is made of and the smoothness of the surface [12, 13]. Note also that, in the figure, all depicted electrons follow a similar trajectory, from the left of the image to the right. This is done for illustrative purposes and for the sake of legibility, but it is not necessarily the case in real RF components. Indeed, because the secondary electrons move in resonance with the oscillating electric field, they tend to stay within a confined space, generating a dense cloud of electrons.[3] This confinement exacerbates the effects the surge in the number of electrons can have on the results of the experiment and on the integrity of the components because the heat and charge anomalies discussed above are concentrated in small, localized points.

2.2 Numerical method

For the numerical calculation of the multipacting effect several steps are taken. First, the electromagnetic field inside the calculations domain is computed by electromagnetic Finite Element Method (COMSOL Multiphysics commercial code or ELCANO open source FEniCSx library based) code. Field magnitude is scaled to a certain value of reference (for example, a total power of 1 W). The simulation 3D tetrahedral mesh and the corresponding values of electric vector field in the mesh nodes are exported to files that are given as input to the multipacting calculation software.

In the multipacting calculation itself, the movement of free electrons are tracked. As a first step the electric field magnitude is scaled to the corresponding input power. Then, the initial N electrons are randomly initiated in any external boundary element of the computation domain. These elements represent a random point in the metallic surface that enclosed the radio-frequency device under consideration. The electrons are assumed to be emitted from the surface with a certain kinetic energy and perpendicular to the surface.

The electrons then are tracked in the interpolated electric field inside the computational domain. The vector field value at any point is interpolated from the finite element mesh. The tracking is done using the Boris algorithm [14–16]. This is a de-facto algorithm for relativistic particles in electric and magnetic fields. The method is not implicit. For each time step Δt, half of the electric impulse is added to electron velocity to obtain a partial velocity increase; then the action of the magnetic field is considered as a rotation in velocity vector, and finally the remaining half of electric action is added. This method is very stable for long calculations, and it keeps accuracy for arbitrary large number of time steps.

When one of the tracked electrons hits the cavity walls (this is detected by checking if the electron is outside of the computational domain mesh), the nearest boundary entity is computed as hitting point. The energy and direction of the impact is collected and passed to the secondary electron routine, that will determine how many, and with which energies, the secondary electrons are extracted from the surface. The algorithm used for this is based on the one described in [17]. This algorithm has a physical component, where the cavity metallic material (usually Copper or Niobium) and its

[3] See, for instance, Fig. 6, where electrons (green paths) stay within a small sector of the coaxial component in a single plane, bouncing off the inner and outer surfaces.

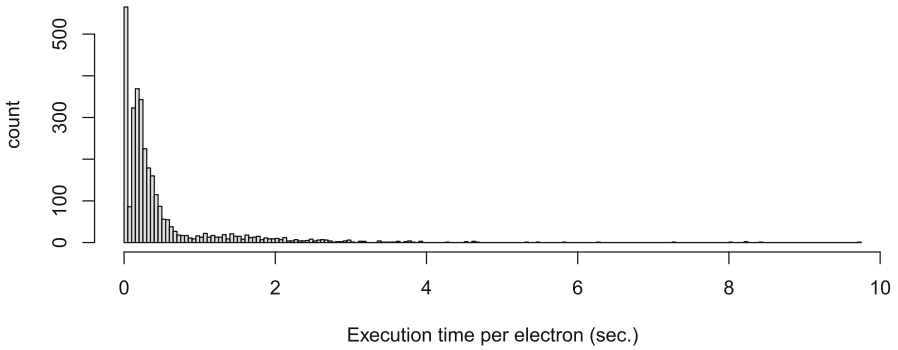

Fig. 4 Histogram of the execution time required to simulate the trajectory of a single electron in the simplest mesh (coaxial)

roughness are taken into consideration as parameters to compute a secondary electron yield figure of merit that is then fed into a probabilistic function that is based on the Poisson distribution that will determine the number of new electrons, their direction and initial energy. These electrons enter the simulation as new elements to be tracked. In the occurrence of multipacting effect situation, the ratio of new electrons is always increasing, resulting in an exponentially growing avalanche of electrons.

2.3 Execution modes

The multipacting simulator has three modes of operation. The first one simulates the trajectory of a single electron. The second one simulates whether multipacting will occur by placing an electron under a given configuration (power and frequency), simulating its trajectory, generating secondary electrons upon collision and then simulating these new electrons, iterating for a number of generations. The third one carries out the multipacting simulation of many configurations (a given frequency and a range of powers).

2.3.1 Single electron trajectory

This mode simulates the trajectory of a single electron using a finite-difference method. The process iterates over the position, velocity and acceleration of a single electron based on the force exerted by the electric field generated by the RF field. The process is interrupted when the electron collides with a surface or if it survives without collision for a predefined time limit, *Max. cycles*, measured in periods of the RF frequency, at which point it is assumed that a stationary trajectory has been reached. Listing 1 presents the pseudocode for this function.

This process can not be parallelized efficiently for two reasons. Firstly, the computation of a single time step is very lightweight, so it would not benefit from being executed in parallel. Secondly, there exist dependencies between consecutive time steps of the simulation, so there is no parallelism available in this plane either. There-

Listing 1 Pseudocode of single_electron_trajectory.

```
input: Electron, Power, Δt, Max.cycles, Mesh
output: Electron
begin
 while not collision(Electron) and  T < Max.cycles
  Force = calculate_force(Electron, Power, Mesh)
  Electron = update_acc_vel_pos(Electron, Force, Δt)
  T = T + Δt
 end
end
```

fore, the simulation of a single electron is considered the basic building block for parallelization in the methods we developed.

2.3.2 Single electron multipacting

This execution mode simulates the multipacting effect, starting with a single electron. The trajectory of the electron is simulated and upon collision, the electron is absorbed by the surface and, based on several physical characteristics,[4] a number of secondary electrons is generated following a probability distribution [12]. These new electrons are queued into the next generation data structure, from which their characteristics will be extracted later to be simulated, possibly producing electrons for the next generation. The simulation continues until there are no more electrons to simulate or until a predefined number of generations is finalized. The number of generations should be large enough so that the multipacting effect can be clearly distinguished. The pseudocode for this function is shown in Listing 2.

Listing 2 Pseudocode of single_electron_multipacting.

```
input: Electron, Generations, ...
output: Electrons
begin
 electron_list[0] = {Electron}
 for g in [0, Generations)
  while not empty(Electron_list[g])
   Electrons[g]++
   e = first_element (Electron_list[g])
   e' = single_electron_trajectory(e, ...)
   generate_sec_electrons(Electron_list[g+1], e')
  end
 end
end
```

It is clear that there exists dependencies from one generation to the next. However, all electrons within a given generation can be executed in parallel and the number of electrons per generation grows rapidly when the multipacting effect occurs, which simplifies extracting parallelism out of the execution. However, efficient parallelization

[4] Such as the power level, the energy of the electron and the secondary electron yield of the surface it has collided with.

is challenging because, as illustrated in Fig. 4, the time needed to simulate each electron varies significantly depending on the initial conditions, so load balancing issues place a definite constraint on the efficiency of parallelization.

2.3.3 Power range

This mode iterates the simulation of the multipacting process across a range of different starting conditions (power level applied to the component). Listing 3 shows the pseudocode for the Power Range simulation.

Listing 3 Pseudocode of power_range.

```
input: Power_range, ...
output: Electrons
begin
 for p in Power_range
  Electrons[p] = single_electron_multipacting(p, ...)
 end
end
```

Since the simulation of each power level can be performed independently, this execution method is very well suited for parallelization but, again, load balancing problems can appear. The unbalance in this execution mode is not only because of the differences in the time needed to simulate each electron (as above) but also because different power levels can show very different behavior with respect to the appearance of multipacting and its magnitude. For this reason, as we will see, parallelizing only within each power level is not very efficient in the general case.

3 Parallelization methods

As discussed above, the main objective of this work is to achieve a substantial acceleration of the particle simulation code in order to expedite the pre-fabrication testing of the components being designed. In addition, this acceleration will enable for larger simulations to be carried out without exceeding the runtime limits imposed by the supercomputer's scheduling policies. This section describes the parallel methods that we developed and that will be evaluated later on in this paper.

3.1 Multiprocessing

We started by developing a first parallel version for the multipacting simulation. The parallelization is based on the Python Concurrent.Futures library [10] which is used to create different execution threads or processes in the program.

This library provides a high-level interface to be able to execute programs asynchronously and simultaneously. In it, there are two different modules: *ThreadPoolExecutor* and *ProcessPoolExecutor*. The difference between these two modules

is that in the first of them, *ThreadPoolExecutor*, the execution can be divided into different threads of execution, while in the second, *ProcessPoolExecutor*, the execution is divided into different processes.

The definition of the terms *threads* and *processes* may vary depending on the context, but in the case of the python interpreter the main difference is that threads run within the same python process and, thus, cannot run in parallel at the same time due to the GIL [5] (Global Interpreter Lock) of python, which does not allow different threads of a same process to be executed in parallel to avoid atomicity and coherence problems. In contrast, processes are created independently and do not share memory space. Thus each process has its own GIL and many process can be run in parallel without consistency issues.

When using either module, the options are the same. There is an abstract class called "Executor" that provides a series of methods to execute calls asynchronously. These methods are as follows:

- **map:** This method is used to schedule the execution of a function in parallel with different input parameters and returning at the end a Future type object that represents the results of said function. For example, the different parameters with which you want to execute the function are included in a list and the list is passed as a parameter, so that said function is executed with all the parameters in the list simultaneously. The maximum number of different processes/threads that can be created at the same time can be modified with the *max_workers* variable.
- **submit:** This method has the same function as *map*, but is used to schedule functions manually; typically one at a time. That is, instead of using the *map* method once with a list of parameters, you must use the *submit* method once for each of the parameter sets.
- **shutdown** Tells the executor to kill all executables when the ones currently running finish.

Finally, the library also includes the aforementioned *Future* type object, which is a class that is used to obtain information about the executions of the function. You can know whether any of the executions have failed and whether the function returns something, in which case, the results are stored in lists arranged in the same order as the parameter sets so to unequivocally know which results corresponds to each input parameter set.

In particular, our code relies on the *Executor* class and the *ProcessPoolExecutor* module. These allow creating independent processes which we leverage to simulate all the electrons of each generation so that they can be run in parallel. The library itself is in charge of dealing with the scheduling of the generated processes. Listing 4 shows the pseudocode for the first parallel implementation, which we codenamed Multiprocessing. This implementation can be used to run both the single electron multipacting mode and the Power Range mode. In the latter, it simply runs sequentially the parallel execution of each power in the predefined range.

[5] https://realpython.com/python-gil/.

Listing 4 Pseudocode of Multiprocessing implementation.

```
input: Generations, Power_range, Cpus
output: Electron_list
begin
 for p in Power_range
  Electron_list[p][0] = generate_initial_electrons(p)
  for g in [0, Generations)
   electron_list[p][g+1] = ProcessPoolExecutor(
       single_electron_trajectory(Electron_list[p][g]), workers=
       Cpus)
  end
 end
end
```

3.2 OnePool

This is an improved parallel implementation that we introduced in order to deal with the unbalancing inherent to the large differences among power levels in the Power Range simulations. OnePool creates a single ProcessPoolExecutor capable of simulating, within a generation, electrons from all the different power levels in parallel. Listing 5 shows the pseudocode for OnePool. OnePool tries to exploit the most parallelism by keeping the execution lanes as busy as possible with processes to run by allowing to simulate in parallel electrons from all sources, independently of the pace at which each electron progresses.

Listing 5 Pseudocode of OnePool implementation.

```
input: Generations, Power_range, Cpus
output: Electron_list
begin
 for p in Power_range
  Electron_list = generate_initial_electrons(p)
 end
 for g in [0, Generations)
  new_electrons = ProcessPoolExecutor(single_electron_trajectory
      (Electron_list),workers=Cpus)
  Electron_list = new_electrons
 end
end
```

3.3 Discussion

To illustrate how the two methods distribute the execution of electrons across the available processors, we show Fig. 5. In the example, there are 3 different power levels to simulate (represented in Blue, Green and Red) and each of them is simulated for 4 generations, represented by increasingly lighter colors. Time evolves from the top down, and electrons from consecutive generations cannot be executed in parallel. Time slots of different lengths are depicted because, as explained above, the time to simulate each electron varies greatly so, the utilization of resources is uneven and the scheduling

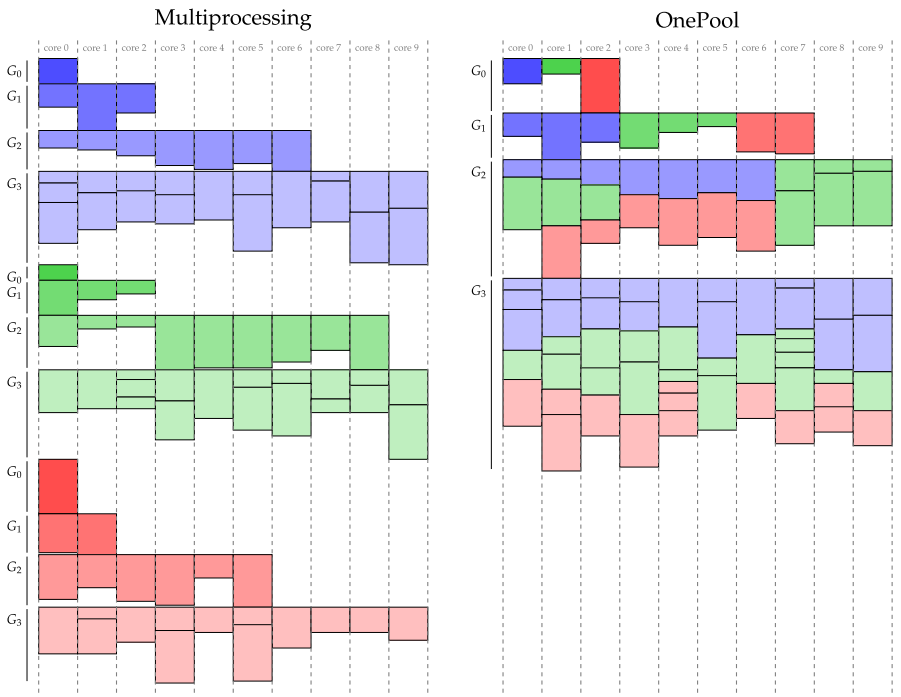

Fig. 5 Example of electron scheduling in the Power Range execution with 10 processing threads. Different colors represent different power levels, and different shades of the color represent different generations. Time flows downwards

of each electron is done *on demand* as resources get available. The disparity in electron times produces significant internal fragmentation when cores need to wait until the slowest of a generation is consumed. In the example, the simulation of electrons is distributed between 10 cores, whose execution lanes are delimited by the gray dotted lines.

We can see that, with multiprocessing, the few first generations do not generate many electrons, so most of the processing resources are left idle. However, as multipacting produces an exponential growth in the number of electrons, in a few more generations, the execution resources would be saturated most of the time. This effect is not shown in full here for the sake of simplicity, but the fourth generation of all power levels is already saturating processing lanes. OnePool is capable of keeping most of the cores simulating electrons most of the time, since it can schedule electrons from any available power, so it can keep processors busy for a larger proportion of time and from much earlier; from the third generation in the example. Note that in the example, electrons from the different power levels are consumed in order for the sake of clarity, but in real simulations, electrons will be added to the execution queues as soon as they are generated. After that, the library will decide *in runtime* in which order they are sent to execution and, therefore, they will be naturally interleaved.

Table 1 Details of the meshes used for evaluation

Component	Mesh complexity	
Coaxial	3885 nodes	19,425 faces
Pikachu 500k	100,553 nodes	502,765 faces
Pikachu 900k	177,897 nodes	889,485 faces

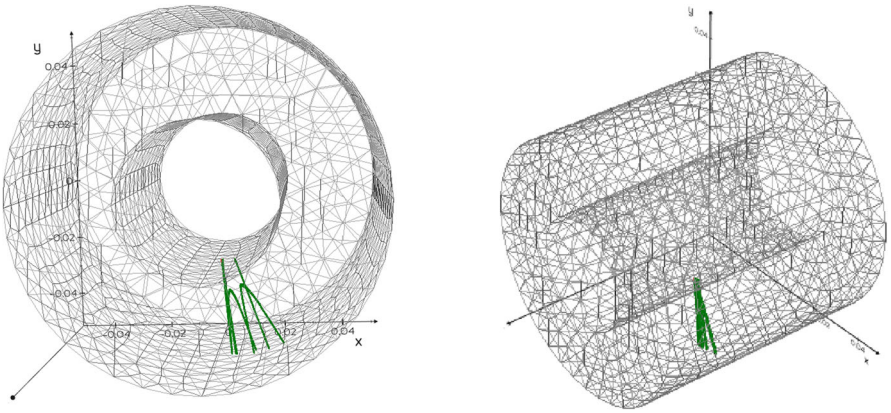

Fig. 6 Depiction of the Coaxial mesh, showing an instance of multipacting from two different angles. Grey lines represent the wire-frame of the component. The thicker green lines represent electron trajectories

4 Performance evaluation

In this section, we describe the experimental set-up used to carry out the evaluation. We start describing the experimental platform and the details of the three meshes used for the evaluation, following with the description of the simulation parameters employed to perform the two types of experiments: Single Electron Multipacting and Power Range.

4.1 Experimental set-up

Our experiments were run on 3 different platforms: a development laptop with 4 cores, a production server with 20 cores and the ATLAS supercomputer which has hundreds of servers featuring dual-socket Intel Xeon Platinum 8280 processors with 56 cores and 192 GB of RAM. Results were consistent across all platforms so, for the sake of brevity, only results from the supercomputer are included in the paper as they allow for the largest scale of simulations and because it is the production system where the simulations will be run. ATLAS uses a CentOS Linux 7 operating system and the Slurm scheduler to distribute the work among the compute nodes.

We considered three different meshes in our evaluation, whose characteristics are gathered in Table 1. The first mesh, Coaxial, is a relatively simple model and was used as the baseline to check functionality and scalability. Figure 6 shows the coaxial mesh from two angles. The other two, Pikachu500k and Pikachu900k, are high detail

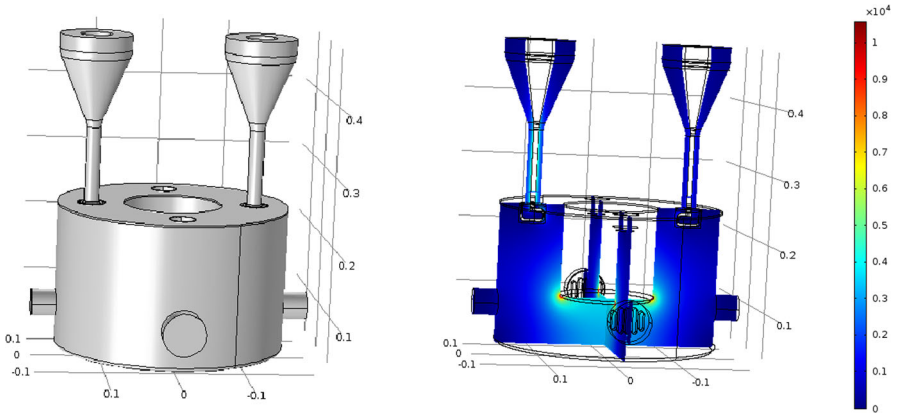

Fig. 7 The Pikachu component: solid render of the model (left), and electric field generated at a given RF frequency (right)

models of a component that was being designed for the ESS accelerator and whose simulations are much more compute intensive and have, in fact, motivated the work carried out in this paper. Figure 7 shows renders of the Pikachu model.

In particular, we carried out two sets of experiments: First, we executed the multipacting experiments for each of the meshes explained before to ascertain the level of scalability we can obtain with this mode. Afterward, we executed the Power Range experiments for each of the meshes and, since the original parallel method did not obtain the expected performance results, we explored why this was the case and, indeed, proposed a new, improved version which we also evaluated. In all cases, we repeated each experiment 8 times (limited by the long runtime of the sequential experiments) and report the average speed-up and the standard deviation.

The code was instrumented so that in all execution modes, each random seed produces deterministic sets of electrons regardless of the number of cores used for the execution. This ensures a fair comparison of speed-up results. Table 2 and Table 3 show the parameters of all the simulations performed. All these parameters have typical values, except for the number of generations, which was reduced to accommodate ATLAS wall-time clock limit when running the sequential code.

For the first set of experiments, we selected only random seeds that generated multipacting, because in the cases where multipacting does not happen, the execution is nearly instantaneous and there is no need for parallelization. Moreover, such results would skew the obtained results. For the second set, where each power level may or may not generate multipacting, we did not do any filtering and simply executed using consecutive random seeds. This is done because in this case, there were no trivial executions. Furthermore, we were looking for the typical range of behaviors so that we can obtain average values from varied simulations. In other words, the objective was to evaluate the Power Range simulations under realistic conditions. However, some (sequential) simulations were exceedingly long and were aborted by the scheduler because they reached the run-time limit of the supercomputer.

Table 2 Simulation parameters employed for the single electron multipacting execution mode experiments

Parameter	Value
Generations	10
Power level	4.0 MW
RF frequency	704.4 MHz
Max. cycles	500
Random seeds	*Coaxial:* 24, 28, 44, 45, 128, 150, 152, 167
	Pikachu 500k: 6, 13, 15, 37, 70, 152, 161, 197
	Pikachu 900k: 4, 8, 15, 37, 53, 77, 119, 137

Table 3 Simulation parameters employed for the Power Range execution mode experiments

Parameter	Value
Generations	8
Power levels	*All:* 0.2, 0.4, 0.6, 0.8, 1.0 MW
RF frequency	704.4 MHz
Max. cycles	500
Random seeds	*All:* 1, 2, 3, 5, 6, 7, 10, 13

4.2 Multipacting experiments

We start by analyzing the performance improvement achieved for the multipacting operation mode. This first set of results, which is depicted in Fig. 8, represents the speed-up obtained as the number of cores is increased. In the figure, the average speed-up achieved using the Coaxial mesh is shown. From the results, it is clear that our implementation scales well from 2 to 32 cores, obtaining in all cases an almost perfect efficiency. However, as the number of cores goes beyond that, the speed-up grows slower, reaching a maximum value of 34× when using 56 cores. Similar results are obtained for the Pikachu 500k mesh with very good efficiency up to 32 cores and achieving a maximum speed-up of 32× using 40 cores. However, in this case, as we increase the number of cores to 48 and 56 cores, we observe diminishing performance as the speed-ups decrease with the number of cores.

Regarding the much more complex Pikachu 900k mesh, the speed-up that we are able to achieve is lower than with the other meshes. In this case, good efficiency is maintained until 24 cores. Afterwards, the performance keeps increasing until 40 cores, where over a 30× speed-up is achieved, but efficiency decreases subsequently, staying below 30×. Adding more cores does not seem to result in any substantial difference in terms of speed-up.

For all three meshes, we found that the variability of the results is relatively small. We can also observe that the more complex the mesh, the higher the variability, which is reasonable since the unbalance in terms of execution time for each electron is expected to grow with mesh complexity. At any rate, achieving speed-ups of around 30× with 32–40 cores with all the meshes is more than adequate for our purposes, as

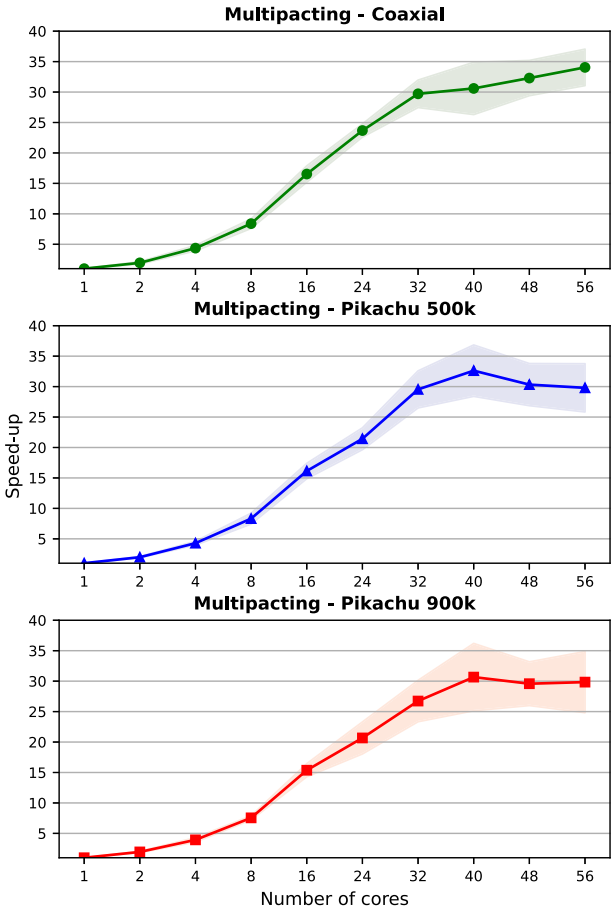

Fig. 8 Speed-up of the Multiprocessing method when simulating Multipacting using the three meshes under consideration. The solid lines represent the average speed-up. The shaded area around them represents the standard deviation

it allows experiments that previously required hours to be run in a matter of minutes. Moreover, it enabled executing, in a few hours, very long experiments that would have taken many days to be executed and, hence, exceeded the wall-time limit imposed by the supercomputing center's policies.

4.3 Power range experiments

We move now to analyze the results obtained for the Power Range experiments. These experiments have been carried-out using the Multiprocessing and OnePool parallelization methods for each of the three meshes.

Let us start analyzing the results obtained with the Multiprocessing method, which are depicted in Fig. 9. In this case, this method clearly achieves much worse results than with Multipacting, with maximum speed-ups of around 20× for the three meshes

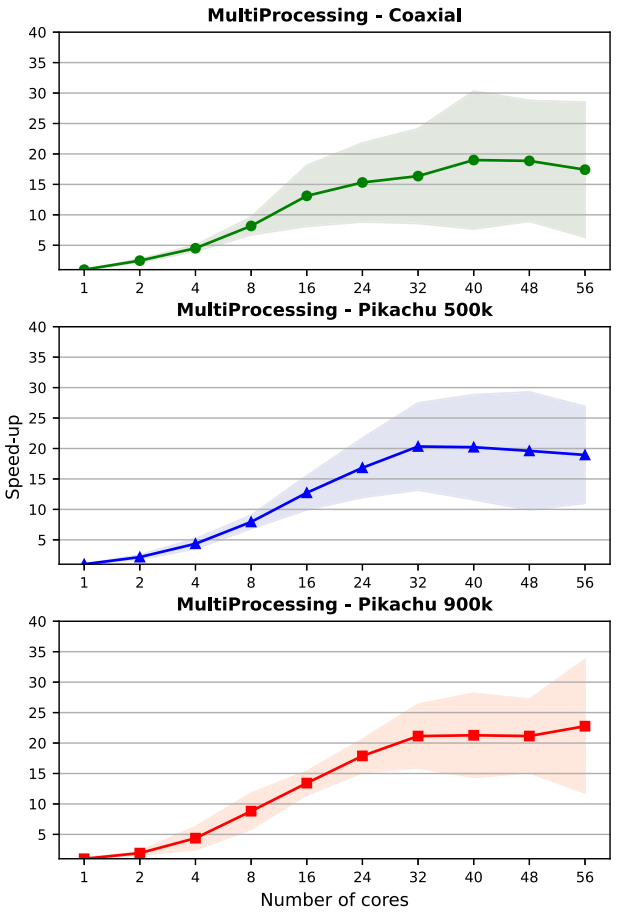

Fig. 9 Speed-up of the Multiprocessing method when executing power range simulations with the three meshes under consideration. The solid lines represent the average speed-up. The shaded area around them represents the standard deviation

and adequate scalability maintained only up to 8 cores. With the Coaxial mesh, the performance increases very slowly until it reaches 40 cores. With Pikachu 500k there is no performance gain above 32 cores and, indeed, performance diminishes slowly after that. With Pikachu 900k, the performance with 32, 40 and 48 cores is the same, with a slight improve for 56 cores. Although speed-ups of around 20× could be enough for our purposes, it is clear that this method does not scale well with the number of cores for this execution mode. In this case, our analysis found load balancing issues which reduce the efficiency of the parallelization.

For this reason, we proceed to investigate these results in more detail. First, we can observe that the variability for all 3 meshes is very high, which makes interpreting the obtained results more difficult. Delving deeper into this issue, we realized the reason for this variability is that, with this method, the results depend greatly on the input parameters. With some random seeds, lots of electrons are created in most of the power

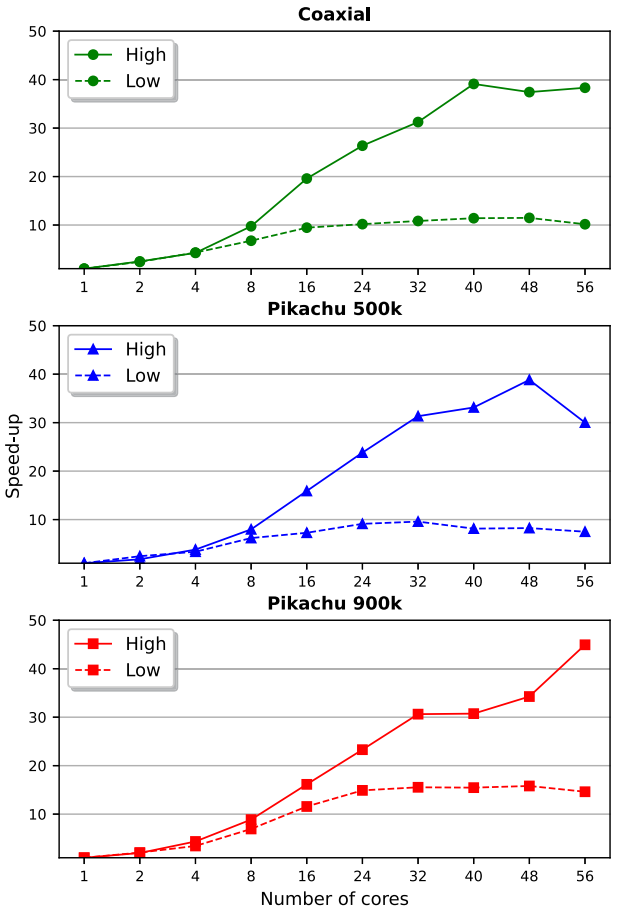

Fig. 10 Variation of speed-up using the multiprocessing method with the three meshes under consideration when high and low numbers of electrons are simulated

levels, so the parallelization is rather efficient, akin to those of the previous subsection. In other cases, the number of electrons in several of the power levels is more limited and scalability is relatively poor. Figure 10 illustrates this by showing the scalability results for each of the three meshes when the number of electrons in all power levels is high (solid) or low (dashed). We can see the great difference in terms of scalability between these two types of runs. The best cases produce numerous electrons in all power levels, so the achieved scalability is in line with the results we observed for Multipacting. In contrast, none of the power levels of the worst cases produce many electrons, so the execution pool was not saturated as often as required to allow for high parallelism. This justified the need for a parallel method that provides more reliable and consistent acceleration, which motivated the introduction of the OnePool method

Let us focus now on the results of that method. Figure 11 shows the results obtained using each of the three meshes. We can observe that the behavior of this parallelization method is much more beneficial than with the Multiprocessing method. Now, the

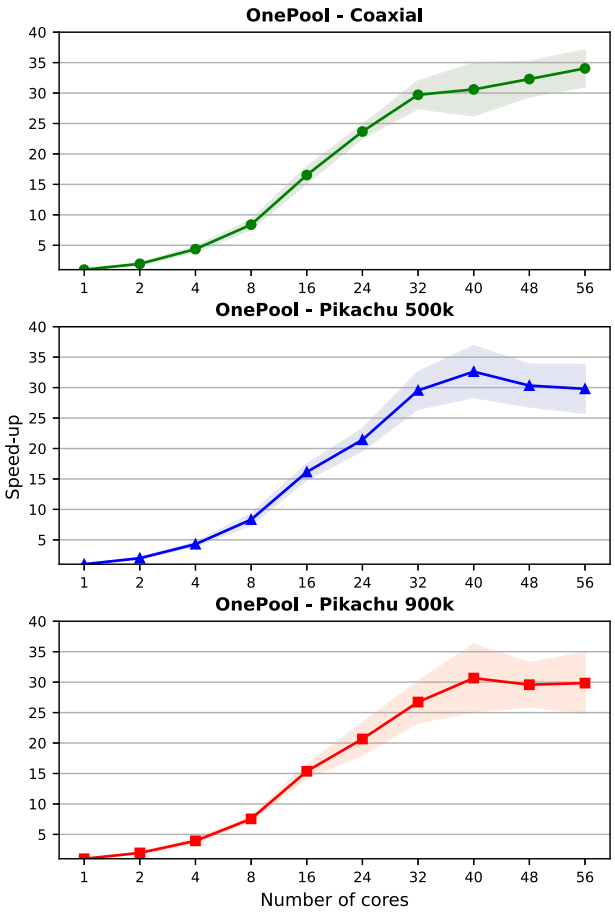

Fig. 11 Speed-up of the OnePool method when executing power range simulations with the three meshes under consideration. The solid lines represent the average speed-up. The shaded area around them represents the standard deviation

speed-up grows steadily until 24 cores where speed-ups of over 20× are achieved, regardless of the mesh. After that number of cores, the speed-up keeps growing up to 56 cores, where all meshes reach a maximum of nearly 30× These results represent an increase in computing throughput of between ∼ 20% and ∼ 50% with respect to the original Multiprocessing method. In addition, it is also worth noticing that the variability of this method is smaller than with Multiprocessing, but still larger than with the multipacting experiments.

5 Conclusions and future work

This paper has discussed the parallelization of a particle simulation program used by ESS Bilbao to design and construct some essential components of the European Spal-

lation Source. The program is mainly utilized to analyze an effect called multipacting which happens when an electron that is inside one of the components collides with the surface of the component itself and, as a consequence, an avalanche of electrons occurs. The simulation of such effect can be very time-consuming and, for that reason, the parallelization of that code is of critical importance. We were able to not only reduce the execution time, but also to enable simulations with much larger amounts of electrons. The parallelization of the code has been performed at the electron level and follows two different approaches. Our first approach, Multiprocessing, consists of simulating the electrons within each generation on different processes. OnePool extends this approach for the Power Range execution mode and tries to achieve the maximum parallelization, by dynamically distributing electrons from different power levels and generations.

Our experimental results suggest a successful parallelization of the code with speed-ups in the range 30× - 35× when using 32–56 cores. The baseline parallel implementation works very well for accelerating single Multipacting simulation, and we were able to accelerate the execution of dense multipacting scenarios up to $\sim 35\times$ with 48 or 56 cores. However, for the Power Range simulations, where the incidence of multipacting and its electron density is more limited, its performance was significantly lower, up to around 20×. For this reason, we developed the improved OnePool method which was able to increase substantially the parallelization efficiency for this second use case, pushing the speed-up to around 30×. In the best cases, parallelization efficiency was maintained up to 32 cores ($\sim 95\%$) and the speed-up plateaus at around 40 to 48 cores with speed-ups of above 30×. While scalability did not seem to extend beyond that number of cores, the obtained acceleration is adequate since it translates hours into minutes and days into hours, hugely accelerating the design process. Moreover, the parallelization has enabled the simulation of components that before were hitting the CPU time quotas of our supercomputing provider.

As future work, we will try to accelerate the simulation of other processes of interest for the design of particle accelerator components, including the behavior of the plasma [18, 19] that may be generated within the components or to support the interaction among electrons or other particles [20]. These processes are more complex than Multipacting and may require the use of more flexible parallel libraries such as MPI for Python [21] or Parallel Python.[6]

Another route to improve the performance that we will look into is porting the code to a more efficient programming language, such as C or C++. These languages are compiled and run at hardware speed, whereas Python is interpreted and, *a priori*, it is believed to be substantially slower. However, given that this in itself is a large software engineering project, going for a parallel version of the available Python code was decided to be a more appropriate plan because it was expected to obtain considerable acceleration much earlier. Another alternative that we may explore in the future is porting the code to Julia [22], an HPC, high-level programming language closer in nature to python than to C.

Acknowledgements This work is supported by the Basque Government (through projects KK-2023/00012, KK-2023/00090 and Consolidated Groups grant IT1504-22). Dr. Javier Navaridas is supported by a Ramón

[6] https://www.parallelpython.com/.

y Cajal fellowship (Grant RYC2018-024829-I) funded by MCIN/AEI/ 10.13039/501100011033 and, as appropriate, by "ESF Investing in your future" or by "European Union NextGenerationEU/PRTR". Julen Galarza had an apprenticeship funded by Donostia International Physics Center (DIPC) when this work was carried out.

Author Contributions All authors contributed to the study conception and design. The original multipacting code was developed by JLM and IB. The parallel version of the simulator was developed by JG with input from JLM and supervised by JN and JAP. Design and analysis of the experiments was done by JN, JAP and JG. JG performed the experiments and was in charge of data curation. TR provided the computing resources to carry out the experiments. JN and JAP wrote the first draft of the manuscript and all authors commented and approved the final version.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

Declarations

Conflict of interest The authors declare no competing interests.

References

1. Garoby R et al (2017) The European spallation source design. Phys Scr 93(1):014001. https://doi.org/10.1088/1402-4896/aa9bff
2. Pérez M et al (2020) ARGITU compact accelerator neutron source: a unique infrastructure fostering R&D ecosystem in Euskadi. Neutron News 31(2–4):19–25. https://doi.org/10.1080/10448632.2020.1819140
3. Celona L et al (2018) High intensity proton source and LEBT for the European spallation source. AIP Conf Proc 2011(1):020019. https://doi.org/10.1063/1.5053261
4. Fernández-Cañoto D et al (2021) Magnetic analysis and cross-talk fields for the ESS MEBT quadrupole magnet. Nucl Instrum Methods Phys Res Sect A 1014:165723. https://doi.org/10.1016/j.nima.2021.165723
5. Mereu P et al (2019) Design details of the European spallation source drift tube Linac. In: Linear Accelerator Conference (LINAC'18), Beijing, China, 16–21 September 2018, pp. 190–192
6. Li H et al (2019) Characterization of a $\beta = 0.5$ double spoke cavity with a fixed power coupler. Nuclear Instrum Methods Phys Res Sect A Accel Spectrom Detect Assoc Equip 927:63–69. https://doi.org/10.1016/j.nima.2019.02.003
7. Shea TJ et al (2018) Overview and status of diagnostics for the ESS project. In: Proceedings of the 6th International Beam Instrumentation Conference, IBIC 2017, pp. 8–15. https://doi.org/10.18429/JACoW-IBIC2017-MO2AB2
8. Vaughan JRM (1988) Multipactor. IEEE Trans Electron Devices 35(7):1172–1180. https://doi.org/10.1109/16.3387
9. Alnæs M et al (2015) The Fenics project version 1.5. Archive of Numerical Software 3(100)
10. Palach J (2014) Parallel programming with python. Packt Pub. Ltd, Birmingham
11. Vaughan JRM (1988) Multipactor. IEEE Trans Electron Devices 35(7):1172–1180
12. Lin Y, Joy DC (2005) A new examination of secondary electron yield data. Surf Interface Anal 37(11):895–900

13. Balcon N et al (2012) Secondary electron emission on space materials: evaluation of the total secondary electron yield from surface potential measurements. IEEE Trans Plasma Sci 40(2):282–290. https://doi.org/10.1109/TPS.2011.2172636
14. Boris JP et al (1970) Relativistic plasma simulation-optimization of a hybrid code. In: Proceedings of the Fourth Conference on Numerical Simulation of Plasmas, pp. 3–67
15. Qin H, Zhang S, Xiao J, Liu J, Sun Y, Tang WM (2013) Why is Boris algorithm so good? Phys Plasmas **20**(8)
16. Zenitani S, Umeda T (2018) On the Boris solver in particle-in-cell simulation. Phys Plasmas **25**(11)
17. Furman M, Pivi M (2002) Probabilistic model for the simulation of secondary electron emission. Phys Rev Spec Top Accel Beams 5(12):124404
18. Yu K et al (2017) Simulation of beam-induced plasma in gas-filled rf cavities. Phys Rev Accel Beams 20:032002. https://doi.org/10.1103/PhysRevAccelBeams.20.032002
19. Litos M et al (2014) High-efficiency acceleration of an electron beam in a plasma wakefield accelerator. Nature 515(7525):92–95
20. Wiedemann H (2015) Particle accelerator physics. Springer, New York
21. Dalcín L et al (2005) MPI for python. J Parallel Distrib Comput 65(9):1108–1115
22. Bezanson J et al (2017) Julia: a fresh approach to numerical computing. SIAM Rev 59(1):65–98. https://doi.org/10.1137/141000671