Congresso de Métodos Numéricos em Engenharia 2015
Lisboa, 29 de Junho a 2 de Julho 2015

PERFORMANCE ANALYSIS OF A PARTICLE-IN-CELL PLASMA PHYSICS CODE ON HOMOGENEOUS AND HETEROGENEOUS HPC SYSTEMS

Xavier Sáez[1*], Alejandro Soba[2], Edilberto Sánchez[3], Mervi Mantsinen[1,4] and José M. Cela[1]

1: Barcelona Supercomputing Center (BSC-CNS)
C. Gran Capità 2-4, 08034 Barcelona, Spain
e-mail: xavier.saez, mervi.mantsinen, josem.cela@bsc.es, web: http://www.bsc.es

2: Centro de Simulación Computacional para Aplicaciones Tecnológicas. (CSC-CONICET)
Godoy Cruz 2390, CABA, Argentina
e-mail: soba@cnea.gov.ar

3: Laboratorio Nacional de Fusión (CIEMAT)
Avenida Complutense 40, 28040 Madrid, Spain
e-mail: edi.sanchez@ciemat.es, web: http://fusionsites.ciemat.es

4: Institució Catalana de Recerca i Estudis Avançats (ICREA)
Pg Lluís Companys 23, 08010 Barcelona, Spain
web: http://www.icrea.cat

Keywords: Particle-in-Cell Plasma, Heterogeneous HPC systems, GPU, ARM

Abstract. *PIC methods are one of the most used methods in plasma simulations. We present a comprehensible evaluation of the PIC code performance on four current parallel platforms: IBM PowerPC, Intel Nehalem (SMP), Intel Sandy Bridge (SMP) and ARM GPU. The behavior of computational algorithms and data structures are analyzed to deduce which code optimizations will make the best use of each platform.*

1 INTRODUCTION

During the last few decades, high-performance computing (HPC) has been dominated by the rapid scaling of the CPU clock frequency (c.f. Moore's Law) (figure 1). Currently, the performance of next-generation supercomputers is limited by the power efficiency and, as a result, several novel hardware designs have emerged.

Originally, many scientific codes were not developed for present designs of supercomputers based on multi-core and heterogeneous architectures. Basically, they were only leveraging task level parallelism through message passing models such as MPI. Unfortunately, a hand-tuning of these codes is often required to exploit the modern platforms capabilities.

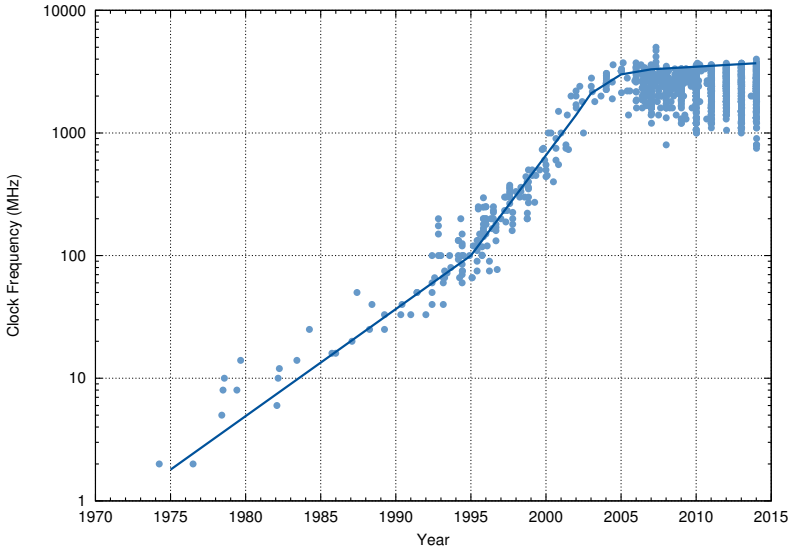

Figure 1: Evolution of the CPU clock frequency. The processor frequency grew up rapidly until around 2005.

We explore the particle-in-cell (PIC) method through a production plasma physic code which simulates fusion plasma instabilities in three-dimensional (3D) geometries. The PIC method is one of the most used methods in plasma simulations. Macroscopically, it describes the plasma dynamics by a system of partial differential equations (continuous model) while microscopically a set of discrete particles is used. The quality of the results achieved with this method relies on tracking a very large number of particles. Therefore, PIC codes require intensive computation and their adaptation to new computing platforms is of particular interest.

We will present a comprehensible evaluation of the PIC code performance on four current parallel platforms: IBM PowerPC, Intel Nehalem (SMP), Intel Sandy Bridge (SMP) and ARM GPU. The behavior of computational algorithms and data structures are analyzed to deduce which code optimizations will make the best use of each platform.

Moreover, in order to check the ported code for a realistic problem, we perform a number of tests in 3D geometries. Traditionally, 3D simulations have been very time-consuming even in the simplified linear limit but are becoming more accessible given the code optimization on modern computing platforms such as those analyzed here.

2 PARTICLE-IN-CELL METHODS

PIC methods are used to model physical systems whose behavior varies over different ranges of spatial scales. Macroscopically the dynamics is described by a system of partial differential equations (continuous model), while microscopically is modeled by a set of discrete particles (figure 2).

These methods attempt to deal with both levels, bypassing the gap between the mi-

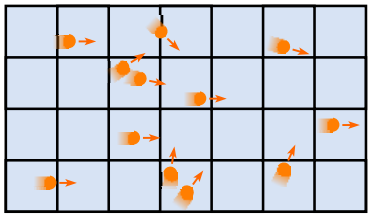

Figure 2: The PIC method. Particles are tracked in a continuous space, whereas moments of distribution are computed on stationary mesh points.

croscopic dynamics and the macroscopic behaviour of the system. Explicitly, a PIC method follows the individual macroparticles (or fluid elements) in a continuous phase space, whereas moments of the distribution (such as densities and currents) are computed concurrently on stationary mesh points.

Nowadays, PIC methods are used in several research areas such as astrophysics, plasma physics and semiconductor device physics [1]. In plasma physics, these methods are one of the most popular approaches to simulate the interaction of independent charged particles with each other and with electromagnetic fields [2].

A *full kinetic description* using the PIC method is implemented by replacing the distribution function f_s by a number of *macroparticles*, which represents a cloud of particles. The charges and densities of macroparticles are accumulated by interpolation on the spatial mesh and then the field equations are solved on the mesh. Finally, the forces acting on macroparticles are obtained by the interpolation of the fields at the macroparticles positions [3].

After an initialization phase, it is possible to summarize a *PIC algorithm* with three steps (figure 3) repeated at each time step [4]:

- **pull**: the particle properties are interpolated to neighboring points in the computational mesh.
- **field solve**: the moment equations are solved on the mesh.
- **push**: the momentum of each particle is calculated by interpolation on the mesh. The particles are repositioned under the influence of the momentum and the particle properties are updated.

In general, many PIC codes designed to simulate various aspects of the plasma behavior [5, 6] were not originally developed for new supercomputers based on multi-core architectures and basically only exploited the *task level parallelism*.

The inclusion of new parallel programming techniques (as *hybridization*) allows to make the best of new multiprocessor supercomputers under development. There are several codes written in this way [7, 8], but in general they are simplified versions of

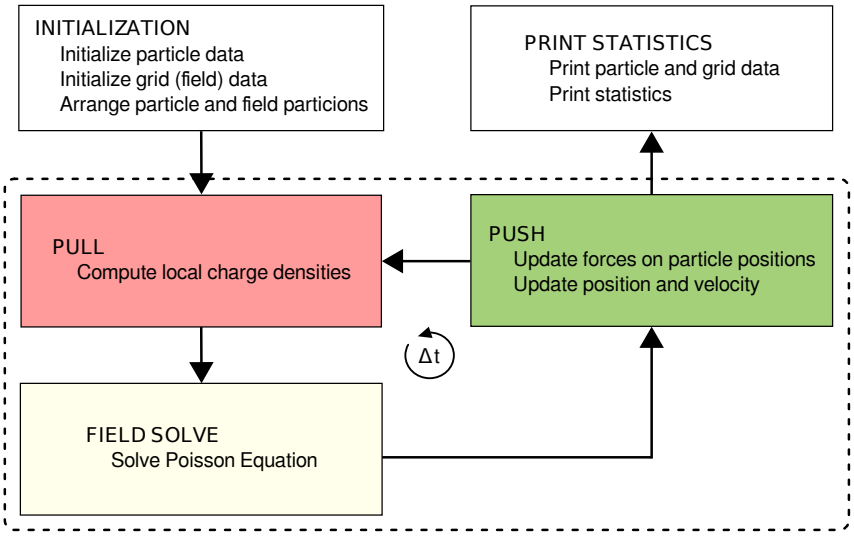

Figure 3: Phases of a general PIC code.

production codes used by scientific groups in the simulation of real plasma physics environments. Nevertheless, this trend is reinforcing to provide more power computing to these applications.

2.1 The particle-in-cell method in EUTERPE

EUTERPE is a parallel gyrokinetic PIC code for global simulations in three-dimensional geometries [9]. It was created at *Centre de Recherches en Physique des Plasmas* (CRPP) in Switzerland, and has subsequently been further developed at *Max-Planck-Institut für Plasmaphysik* (IPP) in Germany. The Fusion Theory unit from *Centro Investigaciones Energéticas, Medioambientales y Tecnológicas* (CIEMAT) and *Barcelona Supercomputing Center* (BSC) in Spain collaborate with IPP on the development and exploitation of this code.

In general, EUTERPE can simulate up to three kinetic species and self-consistently simulate the evolution of electric and magnetic fields, given by the Poisson equation and the Amperes law (in the electromagnetic version).

The evolution of the distribution function of each kinetic species is given by the gyrokinetic equation:

$$\frac{\partial f_s}{\partial t} + \dot{\mathbf{R}}\frac{\partial f_s}{\partial \mathbf{R}} + \dot{v_\parallel}\frac{\partial f_s}{\partial v_\parallel} = C(f_s), \tag{1}$$

where the evolution in time of $v_\parallel$ and $\mathbf{R}$ is given, in the electrostatic approximation used in this work, by the non linear equations

$$\frac{\mathrm{d}\vec{\mathrm{R}}}{\mathrm{dt}} = v_\parallel \vec{b} + \frac{\mu B + v_\parallel^2}{B\Omega_i}\vec{b}\times\nabla B + \frac{v_\parallel^2}{B\Omega_i}(\nabla\times B)_\perp - \frac{\nabla\langle\phi\rangle}{B}\times\vec{b} \tag{2}$$

$$\frac{\mathrm{dv}_\parallel}{\mathrm{dt}} = -\mu \left[\vec{b} + \frac{v_\parallel}{B\Omega_i}(\nabla \times B)_\perp\right] \nabla B - \frac{q_i}{m_i}\left(\vec{b} + \frac{v_\parallel}{B\Omega_i}[\vec{b} \times \nabla B + (\nabla \times B)_\perp]\right) \nabla\langle\phi\rangle \quad (3)$$

$$\frac{\mathrm{d}\mu}{\mathrm{dt}} = 0, \quad (4)$$

where: μ is the magnetic moment per unit mass (which is a constant of motion), q_i and m_i are the ion charge and mass respectively, $\Omega_i = \frac{q_i B}{m_i}$ is the ion cyclotron frequency, $\frac{\vec{b}\nabla B}{B}$ is the unit vector in the magnetic field B direction and $\langle\phi\rangle$ is the renormalized potential introduced in [10].

In this work collisions are not taken into account, so that $C(f_s) = 0$.

A control variates (or δf decomposition) is used in the code: the distribution function is separated into an equilibrium part (Maxwellian) and a time-dependent perturbation. Only the evolution of the perturbation is followed, which allows to reduce the noise and the required resources, in comparison with the alternative of simulating the evolution of the full distribution function.

In the electrostatic approximation the system of equations (2-4) is complemented with the Poisson equation from [10], which can be simplied by neglecting high order terms and by making the long-wavelength approximation, so that a quasi-neutrality equation is obtained:

$$\langle n_i\rangle - n_0 = \frac{e n_0(\phi - \bar{\phi})}{T_e} - \frac{m_i}{q_i}\nabla\left(\frac{n_0}{B^2}\nabla_\perp\phi\right) \quad (5)$$

with n_0 the equilibrium density, e the elementary charge, T_e the electron temperature, $\langle n_i\rangle$ the gyroaveraged ion density and $\bar{\phi}$ the flux surface averaged potential.

The electrostatic potential is represented on a spatial mesh and the distribution function is discretized using markers (macroparticles) that carry the electric charge.

Two coordinate systems are used in the code: a system of magnetic coordinates (PEST) (s, θ, ϕ) for the potential and a system of cylindrical coordinates (r, z, ϕ) for pushing the particles, where $s = \Psi/\Psi_0$ is the normalized toroidal flux.

The equations for the fields are discretized using finite elements (B-splines). The resulting matrix equations are solved using the PETSc library (Portable, Extensible Toolkit for Scientic Computation) [11].

Finally, an in-depth explanation about the first version of the code can be found in [12], and details about improvements in newer versions of the code can be found in [13, 14, 15, 16, 17]

3 FRAMEWORK

3.1 Platforms

The study of the code was performed in the following platforms:

- **Marenostrum II** (BSC) is a cluster of IBM JS21 server blades. Each node has 2 PowerPC 970MP processors (2-Core) at 2.3 GHz with 8 GB of RAM. The nodes are connected using a Myrinet network. The peak performance is 94.21 TFlops.
- **Minotauro** (BSC) is a NVIDIA GPU cluster with 128 Bull B505 blades. Each node contains 2 Intel Xeon E5649 (6-Core) at 2.53 GHz and 2 M2090 NVIDIA GPU Cards with 24 GB of RAM. The peak performance is 185.78 TFlops.
- **Arndale prototype** (BSC) is a cluster of ARM-based machines which has 3 nodes and each of them contains an ARM Cortex-A15 (2-Core) at 1.7 GHz and an ARM Mali T604 GPU with 2 GB of RAM. The nodes are interconnected using a 100 Mb Ethernet network.
- **Curie** (CEA) is a supercomputer composed of three different architectures. In our case, the code run in the *Curie fat* which consists of 360 nodes. Each node has 4 Intel Nehalem-EX processors (8-core) at 2.27 GHz with 128 GB.

3.2 Tools

The following tools developed at BSC have been used in the work reported in this paper:

- **Extrae** is a performance extraction tool to trace programs. Extrae gathers data like hardware performance counters and source code references and generates trace files that can be later visualized with Paraver.
- **Paraver** (PARAllel Visualization and Events Representation) is a tool to visualize and analyze a parallel events trace file. Paraver is based on a simple interface to explore the collected data [18].

4 PROCESS LEVEL PARALLELISM

The key step in parallelizing a program is called *decomposition* and consists in breaking the problem into discrete parts that can be solved simultaneously. In this way, each part of the problem can be executed concurrently on different processors.

A program consists of a set of data and an algorithm, so there are two main ways of approaching the problem: partition the data and then work out how to associate the computation with divided data (*data decomposition*) or partition the computation into disjoint tasks and then dealing with the data requirements of these tasks (*functional decomposition*).

4.1 Data Decomposition

The parallelization of EUTERPE is based on the data decomposition, concretely the particle data and the mesh (with the field data) are distributed among the tasks. There are three parallelization strategies used in EUTERPE depending upon which of these data drives the division (figure 4): *domain decomposition*, *particle decomposition* and *domain cloning*.

The programming model used to implement these strategies is the Message Parallel Interface (MPI).

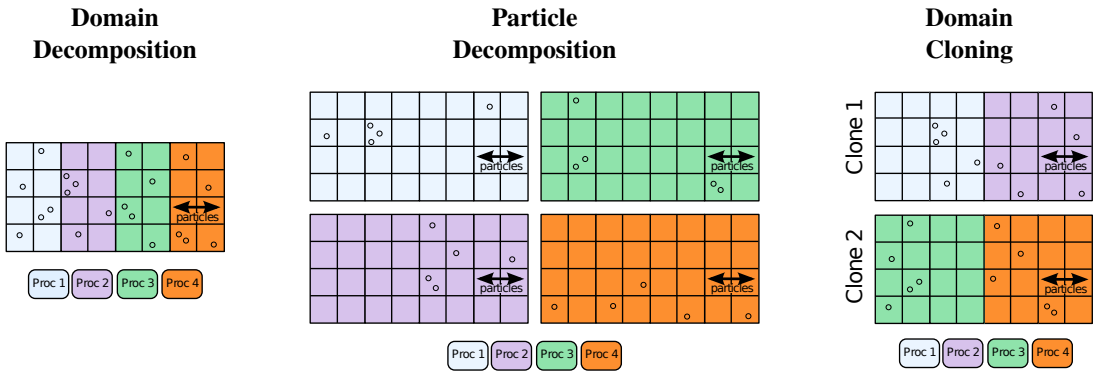

Figure 4: Data decomposition strategies.

4.1.1 Domain decomposition

The *domain decomposition* is the most usual technique to parallelize a PIC code. In EUTERPE, the mesh (domain) is divided into portions in the toroidal (ϕ) direction and each portion (subdomain) is assigned to a processor. Then the particles are sent to the processor responsible for the mesh portion where they are located to split efficiently the computation among processors.

The main advantages of this technique are the increase of data locality and the intrinsic scalability of the physical-space resolution, although the parallelization is limited by the mesh divisions.

On the other hand, this strategy implies a particle migration, since at the end of each time step the particles which have moved to an adjacent domain portion placed in other processor have to be transferred to their new processor.

However, although each subdomain has approximately the same number of particles, the particle migration between subdomains can cause a load imbalance because the strategy ignores the particle distribution. And this could lead to the borderline case where all particles will finish in the same subdomain.

4.1.2 Particle decomposition

The *particle decomposition* technique divides the particle population into subsets that are distributed among the processors. However, the mesh is not distributed and the whole spatial grid is assigned to each processor.

This strategy presents the advantages of *intrinsic load balancing* since the particle population per processor is keeping constant over the simulation.

However, to update the electromagnetic field data, the partial contributions of each subset of particles at the grid points have to be communicated among processors, which means there is a costly all-reduce communication over all processors for accumulating the field data stored on each processor.

In addition, the strategy requires that the whole spatial grid fits in the memory of each processor and if we increase the number of processors then it will only allow to increase the number of particles and not the spatial resolution. Therefore, one may conclude that the particle decomposition is not suitable for highly parallel architectures.

4.1.3 Domain cloning

The *domain cloning* [19] is a combination between the domain decomposition and the particle decomposition.

The processors are distributed into groups (called *clones*). Then the mesh (*domain*) is divided into as many portions as clones and each clone is responsible for a *subdomain*. Finally the particles are divided in two steps: firstly among the clones, and secondly among the processors inside a clone.

This strategy is less restrictive than the domain decomposition because the particle decomposition inside clones increases the application scalability and the particle migration is reduced as only happens inside the clones.

4.2 Analysis of MPI implementation

The first analysis is the scalability of the initial version (pure MPI) of EUTERPE on Curie supercomputer. The data set used correspond to a typical scenario of ITGs in a cylindrical geometry, divided in the ϕ direction into 256 portions, so the domain cloning has been applied to allow the problem be distributed on more than 256 processors.

The figure 5 shows a good scalability until 2000 processors. This indicates a good load-balance, therefore the decomposition of the problem has good quality since all the processors make the same work.

The figure 6 shows a detailed view of the temporal sequence of the MPI tasks. This time the execution is on Marenostrum using 32 tasks without clones. The resolution of the spatial grid is $n_s \times n_\theta \times n_\phi = 32 \times 64 \times 64$ and the number of markers is 1 million.

EUTERPE has two pre-processing steps: the generation of the electrostatic fixed equilibrium as initial condition done by an external application and the computation of the

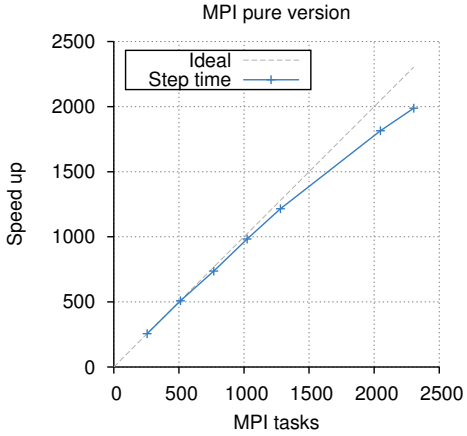

Figure 5: Scalability of the time step using $8 \cdot 10^9$ markers on Curie. Domain cloning has been applied and each clone contains 256 processors.

finite element matrix contained in the quasi-neutrality equation. Their computational loads are not significant compared to the main part because they are only executed once and therefore they are omitted in the figure.

This figure confirms the good behaviour of the MPI version and the well balanced work between the tasks, as all the tasks describe a very similar behaviour. Additionally, the peak performance of push routine, 650 MFlops, gives us signals that we are in front of a memory bound issue, because of the cost of memory operations is linear with the cost of computation.

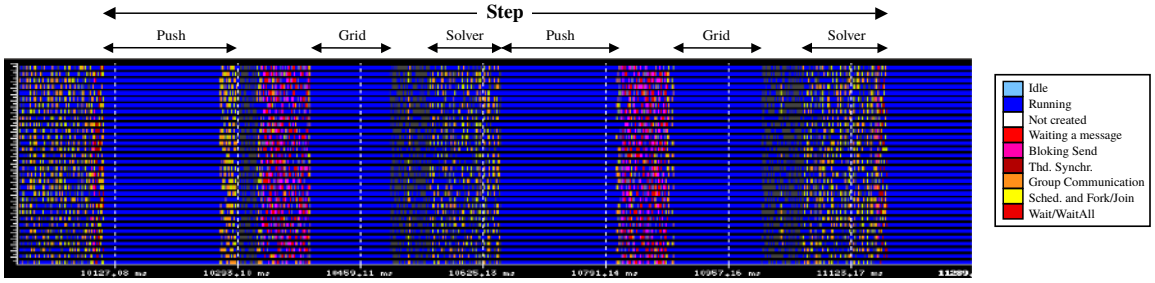

Figure 6: Timeline showing the state of MPI tasks during one time step. The name of the routines is specified on the top of the graphic. On the right, there is the legend to interpret the state of the tasks.

5 THREAD LEVEL PARALLELISM

EUTERPE was originally parallelized following the above strategies at the task level using MPI. For that reason, we developed a hybrid version of the code using OpenMP to take advantage of all the levels of parallelism that a multi-core architecture can offer [20].

5.1 OpenMP implementation

OpenMP [21] is a shared memory parallel programming model that allows to implement multithreading applications by using a set of compiler directives.

As Amdahl's law [22] teach us, only the most time-consuming phases of an application are worth parallelizing. Accordingly, the algorithm has been profiled to find the most time-consuming routines, considering four datasets with different number of markers (figure 7).

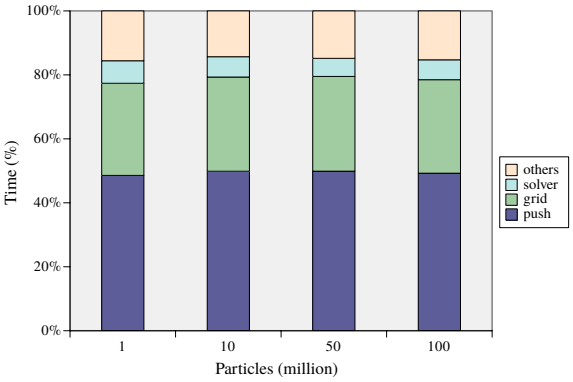

Figure 7: Time distribution across routines in EUTERPE. The tests were run on Marenostrum. The simulations moved 1, 10, 50 and 100 millions of markers into a cylinder divided in a grid of $64 \times 1024 \times 1024$ pieces.

From the results obtained, the three functions that covered jointly more than 80 per cent of the execution time were chosen for introducing OpenMP as explained below:

- **push**: It moves particles inside a subdomain. It is an appropriate place to introduce OpenMP because the computation of the movement of any particle is independent from the rest of particles, so several threads can read simultaneously the electric field on the nearest grid points to a given particle without conflicts (figure 8.a).
- **grid**: It computes the charge density on the grid points. This time there is an issue because several threads can update the same grid point at the same time (figure 8.b). Depending on the number of cores and the available amount of memory this conflict is solved by atomic arithmetic operations or by defining a private copy of the mesh per thread.
- **pets_solve**: It solves the field equation. An alternative hybrid solver was developed following a Jacobi Preconditioned Conjugate Gradient algorithm (figure 8.c).

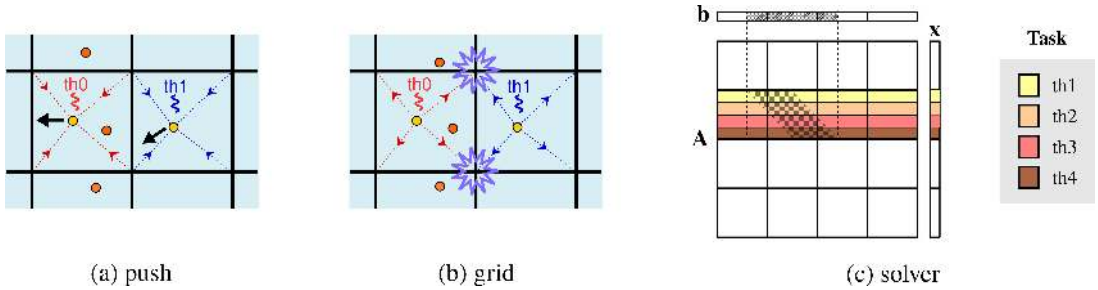

Figure 8: Distribution of work among threads (OpenMP) inside a task. In the push and grid subroutines, the particles are distributed among the threads, while in the solver subroutine the matrix rows are distributed among the threads.

5.2 Analysis of OpenMP implementation

Figure 9 shows graphically the behavior of the hybrid version (MPI+OpenMP) on Minotauro. To facilitate the reading of the figure a small test is shown but similar behavior is expected with bigger cases.

According to the figure, we deduce that the work is well balanced and the threads work most of the time. Only a few sections of the code (indicated in the figure) do not yet have OpenMP, hence they are executed by a single thread in each task while the rest of threads are shown as idle.

5.3 OpenCL implementation

OpenCL (Open Computing Language) is an open royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors, giving software developers portable and efficient access to the power of these heterogeneous processing platforms [23].

OpenCL divides the computing system into a *host* (CPU) and a set of *compute devices* (CPU, GPU or another accelerator). The application is divided in two parts: host code and kernels. The *host code* is responsible for setting up the environment and sending computing portions of the application (*kernels*) to the compute devices, which execute many instances of these kernels (*work-items*). The work-items are grouped by *work-groups* and the work-items inside a group share resources such as memory.

The OpenCL development started from the work carried out in the OpenMP version. This time the two most compute-intensive subroutines were chosen to build a kernel:

- **push kernel**: moves the particles. The work distribution is simple: one work-item is assigned to each particle. Since each work-item works on a different particle, all work-items write in different memory locations (as figure 8a shows but replacing threads with work-items). Therefore, the adaptation of the OpenMP routine was reduced to a direct translation.

- **pull kernel**: determines the particle charge on the grid. This kernel is more chal-

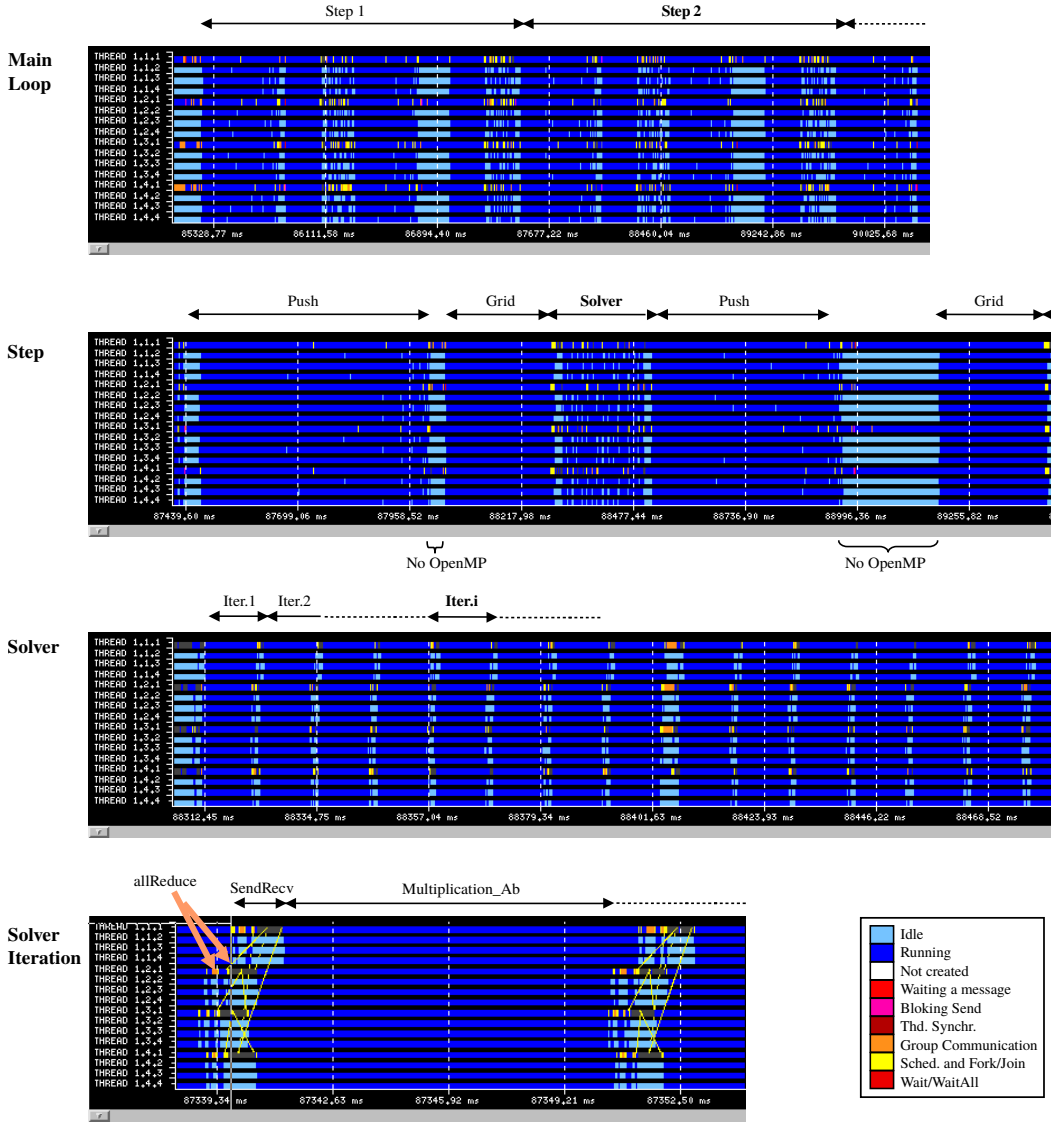

Figure 9: Timeline showing the state of OpenMP threads during one time step. The dark blue colour means that the thread is running and the light blue colour means that the thread is idle.

lenging to implement, since different particles can contribute to the charge density on the same mesh point. As we maintain the same work distribution than in push kernel, several threads can update the same memory location (as figure 8b shows but replacing threads with work-items). The strategy to solve this issue consisted of creating a mesh copy per work-group, so only the work-items which belong to the same work-group can accumulate contributions on the same copy using atomic operations. And finally, a global reduction (*kernel global reduction*) is performed by a new kernel to reduce all these copies to the final mesh.

The next step is to take advantage of all resources, so a simple algorithm on the host decides how to distribute the particles between CPU and GPU. The particles on the CPU are computed using the OpenMP version, whereas the particles on the GPU are computed using the OpenCL version.

5.4 Analysis of OpenCL implementation

Extrae allows to create and analyze traces of OpenCL programs, but it is not ready for a hybrid version with OpenMP and thus is not possible to get a valid trace. So as it is not possible to get any graphical representation of the hybrid version working, the figures 10 and 11 only contain information about a pure OpenCL code.

This time the simulation was done on Arndale prototype and consisted of a grid with cylindrical geometry and a resolution of $n_s \times n_\theta \times n_\phi = 32 \times 32 \times 16$. The simulation moved 1 million of markers and the data set was smaller because of the lower amount of memory available in the machine.

The figures 10 and 11 provide timelines to show OpenCL calls on the host and compute device and also to show the calls to the push, pull and global reduction kernels. These figures make evident the significant additional work needed to configure the OpenCL environment (left of the figures), although luckily it only has to be paid once at the beginning. Also, it is really important to be careful with the data movement between host and compute devices if we do not want to ruin the improvement achieved with the GPU.

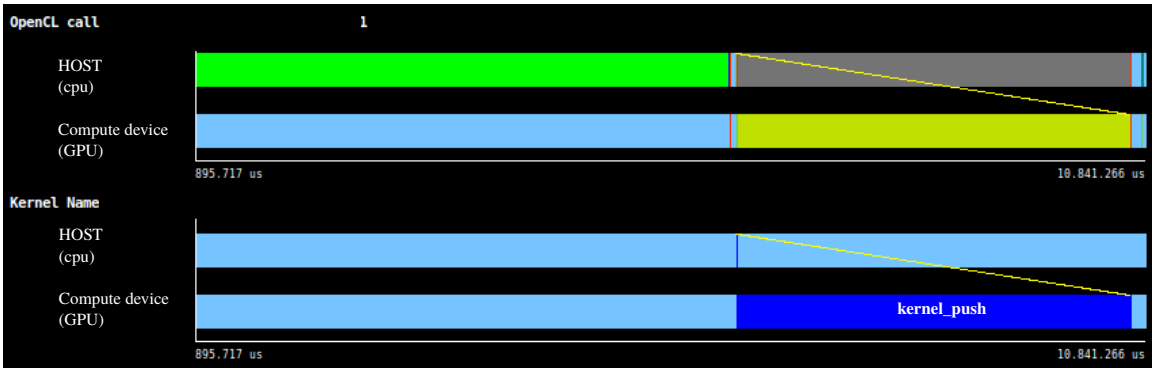

Figure 10: Timeline showing OpenCL calls and the kernel push running. Each work-group contains 32 work-items (it means that is in charge of 32 particles).

5.5 OmpSs implementation

Unfortunately, the main drawback of OpenCL is the low productivity because is a low-level programming language. Therefore, porting an application to a certain platform can be very time consuming if we want to reach the maximum performance of it.

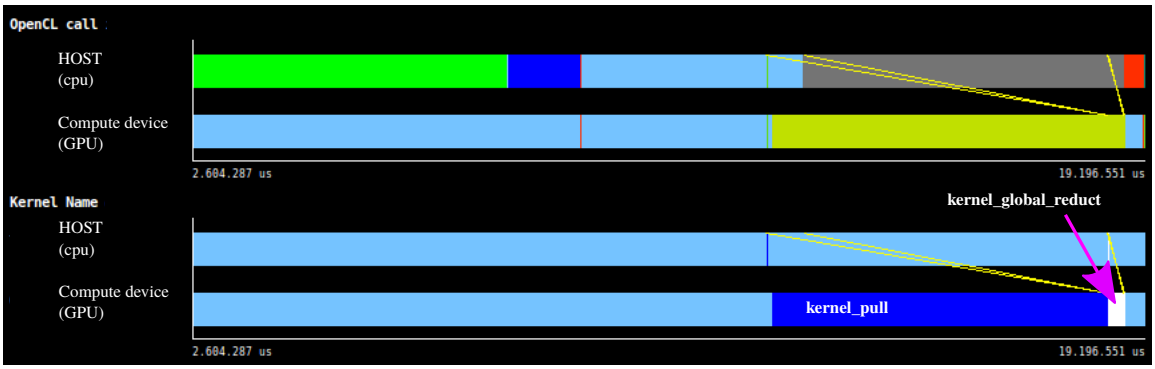

Figure 11: Timeline showing OpenCL calls and the pull and global_reduction kernels running. Each work-group contains 32 work-items (it means that is in charge of 32 particles).

However, there is an alternative developed at BSC to face this issue. It is called *OmpSs* [24] and is a task-based programming model that provides an abstraction to the user which reduces programmer effort and unifies the SMP, heterogeneous and cluster programming in one model.

The programmer annotates sections of the code to parallelize (*tasks*) with special directives. Over these tasks, the user can define *data dependencies* among them and specify the device where these tasks can be executed. Then, the OmpSs runtime arranges tasks that have their dependencies satisfied and performs the required data transfers.

5.6 Analysis of OmpSs implementation

For this analysis, we repeat the same test of the previous section 5.4 on Arndale prototype.

The figures 12 and 13 show the results obtained when executing the push and pull kernels, respectively. Concretely, these figures indicate the distribution of the OpenMP and OpenCL tasks within the CPU and GPU, the evolution of the tasks in the execution and how the OmpSs runtime distributes the tasks.

In both figures, the reader can see that tasks are distributed relatively balanced between the hardware resources (CPU and GPU) automatically. It is probable that the application does not achieve the maximum performance but compensates the improvement on productivity thanks to the reduction in programming complexity.

Finally, the greater length of the tasks in the figure 13 (kernel pull) is due to the lock contention by memory conflicts when different particles contribute to the charge density on the same mesh points. On the other hand, the short length of the tasks in the figure 12 (kernel push) is due to the lack of conflicts because the computation of one particle is independent from the rest of them.

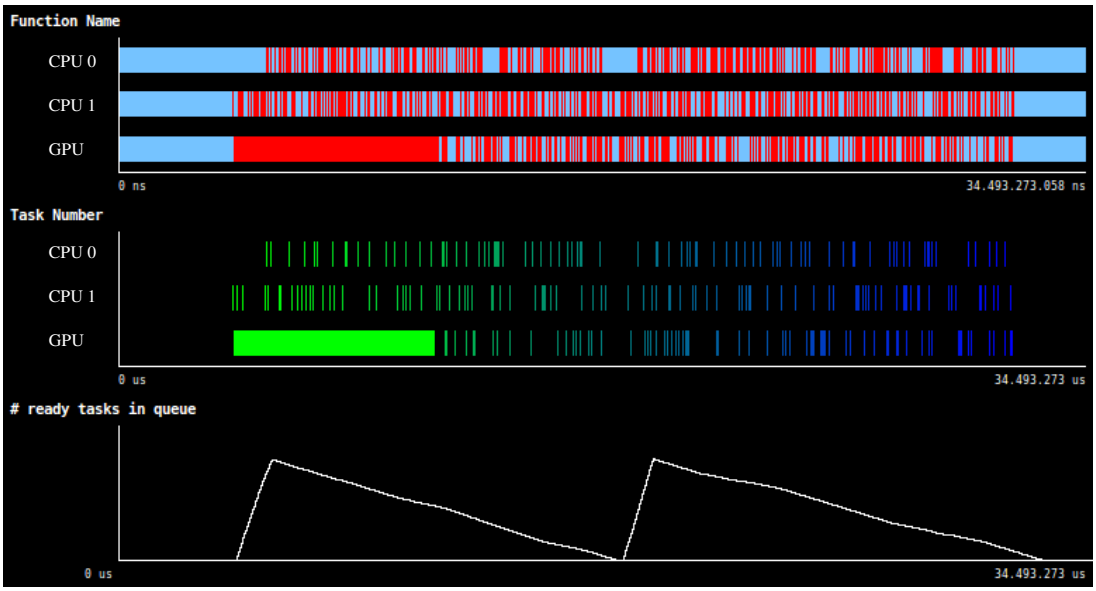

Figure 12: Kernel Push. Timeline showing the state and evolution of the tasks as well as the management of the task queue done by the runtime. The OmpSs runtime distributes the kernel computation in 977 tasks. In the above plot the red colour means running tasks. In the middle plot the green colour shows the first created tasks and the blue colour shows the last ones.

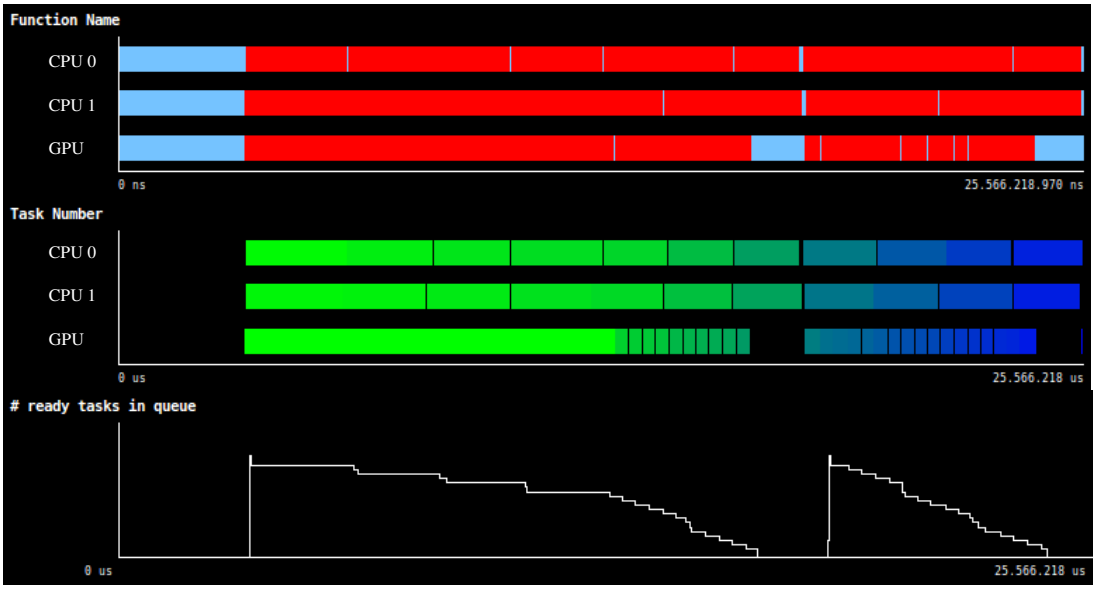

Figure 13: Kernel Pull. Timeline showing the state and evolution of the tasks as well as the management of the task queue done by the runtime. This time the OmpSs runtime only distributes the kernel computation in 25 tasks to reduce the number of mesh copies. In the above plot the red colour means a running task. In the middle plot the green colour shows the first created tasks and the blue colour shows the last ones.

6 CONCLUSION

In this paper we have summarized the work done in the analysis of a PIC application, EUTERPE, on different current computing platforms. We have presented analysis that complements the classic view of the execution time as the main performance measurement. We find that gathering the information that OmpSs offers can provide us valuable knowledge for the improvement of the applications.

As a future work, we plan to extend this analysis to new platforms as MIC. Intel MIC or Intel Many Integrated Core Architecture is a new coprocessor computer architecture developed by Intel. Also, we plan to study new tools (as clustering) to find the structure of parallel applications by characterizing its computation regions.

ACKNOWLEDGEMENT

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under the Mont-Blanc Project (http://www.montblanc-project.eu), grant agreement nº 288777 and 610402.

REFERENCES

[1] R. W. Hockney and J. W. Eastwood. *Computer Simulation Using Particles*. Taylor & Francis, Inc., Bristol, PA, USA, 1988.

[2] J.M. Dawson. Particle simulation of plasmas. *Rev. Mod. Phys.*, 55:403–447, Apr 1983.

[3] P. Pritchett. Particle-in-cell simulation of plasmas a tutorial. In *Space Plasma Simu.*, volume 615 of *Lecture Notes in Phys.*, pages 1–24. Springer Berlin Heidelberg, 2003.

[4] E. Akarsu et al. Particle-in-cell simulation codes in high performance fortran. In *Proceedings of the 1996 ACM/IEEE Conf. on Supercomputing*. IEEE Computer Society.

[5] S. Jolliet, A. Bottino, P. Angelino, R. Hatzky, et al. A global collisionless {PIC} code in magnetic coordinates. *Computer Physics Communications*, 177(5):409 – 425, 2007.

[6] J.A. Heikkinen, S.J. Janhunen, T.P. Kiviniemi, and F. Ogando. Full f gyrokinetic method for particle simulation of tokamak transport. *Journal of Computational Physics*, 227(11):5582, 2008.

[7] H. Burau, R. Widera, et al. Picongpu: A fully relativistic particle-in-cell code for a gpu cluster. *Plasma Science, IEEE Transactions on*, 38(10):2831–2839, Oct 2010.

[8] M. Ragan-Kelley. Cs267 report particle simulation on a gpu with pycuda. Technical report, 2010.

[9] G. Jost, Tran T.M., L. Villard, W.A. Cooper, and K. Appert. First global linear gyrokinetic simulations in 3D magnetic configurations. In *26th EPS Conference on Controlled Fusion and Plasma Physics*, 1999. 26th EPS Conference on Controlled Fusion and Plasma Physics, Maastricht, The Netherlands, June 1999.

[10] T. S. Hahm. Nonlinear gyrokinetic equations for tokamak microturbulence. *Physics of Fluids (1958-1988)*, 31(9):2670–2673, 1988.

[11] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.5, Argonne National Laboratory, 2014.

[12] G. Jost, T. M. Tran, W. A. Cooper, L. Villard, and K. Appert. Global linear gyrokinetic simulations in quasi-symmetric configurations. *Physics of Plasmas (1994-present)*, 8(7):3321–3333, 2001.

[13] V. Kornilov, R. Kleiber, and R. Hatzky. Gyrokinetic global electrostatic ion-temperature-gradient modes in finite equilibria of wendelstein 7-x. *Nuclear Fusion*, 45(4):238, 2005.

[14] K. Kauffmann, R. Kleiber, R. Hatzky, and M. Borchardt. Global linear gyrokinetic simulations for lhd including collisions. *Journal of Physics: Conference Series*, 260(1):012014, 2010.

[15] R. Kleiber, R. Hatzky, A. Knies, K. Kauffmann, and P. Helander. An improved control-variate scheme for particle-in-cell simulations with collisions. *Computer Physics Communications*, 182(4):1005 – 1012, 2011.

[16] R. Kleiber and R. Hatzky. A partly matrix-free solver for the gyrokinetic field equation in three-dimensional geometry. *Computer Physics Communications*, 183(2):305 – 308, 2012.

[17] M. Borchardt, R. Kleiber, and W. Hackbusch. A fast solver for the gyrokinetic field equation with adiabatic electrons. *Journal of Computational Physics*, 231(18):6207 – 6212, 2012.

[18] V. Pillet, J. Labarta, T. Cortes, and S. Girona. PARAVER: A Tool to Visualize and Analyze Parallel Code. In Patrick Nixon, editor, *Proceedings of WoTUG-18: Transputer and occam Developments*, pages 17–31, mar 1995.

[19] R. Hatzky. Domain cloning for a particle-in-cell (pic) code on a cluster of symmetric-multiprocessor (smp) computers. *Parallel Computing*, 32(4):325 – 330, 2006.

[20] X. Sáez, A. Soba, E. Sánchez, J. M. Cela, and F. Castejón. Particle-in-cell algorithms for plasma simulations on heterogeneous architectures. In *Proceedings of the 19th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 385–389. IEEE Computer Society, 2011.

[21] OpenMP Architecture Review Board. OpenMP application program interface v3.0, May 2008.

[22] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[23] OpenCL: Opencl 1.1 reference pages. https://www.khronos.org/opencl.

[24] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, et al. Ompss: A proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.