Revista Colombiana de Computación
Vol. 25, No. 1. January – June 2024, pp. 39-47
e-ISSN: 2539-2115, https://doi.org/10.29375/25392115.5053
*Selected paper previously presented at the Latin America High Performance Computing Conference (CARLA 2023), an event held in Cartagena de Indias, Colombia, September 18-22, 2023.*

REVISTA COLOMBIANA DE COMPUTACIÓN

# An Implementation of a Plasma Physics Application for Distributed-memory Supercomputers using a Directive-based Programming Framework

Christian Asch[1], Emilio Francesquini[2], Esteban Meneses[1,3]

[1] Advanced Computing Laboratory, National High Technology Center, Costa Rica
[2] Federal University of ABC, Brazil
[3] School of Computing, Costa Rica Institute of Technology, Costa Rica
casch@cenat.ac.cr, e.francesquini@ufabc.edu.br, casch@cenat.ac.cr

(Received: 6 February 2024; accepted: 18 June 2024, Published online: 30 June 2024)

**Abstract.** To extract performance from supercomputers, programmers in the High Performance Computing (HPC) community are often required to use a combination of frameworks to take advantage of the multiple levels of parallelism. However, over the years, efforts have been made to simplify this situation by creating frameworks that can take advantage of multiple levels. This often means that the programmer has to learn a new library. On the other hand, there are frameworks that were created by extending the capabilities of established paradigms. In this paper, we explore one of this libraries, OpenMP Cluster. As its name implies, it extends the OpenMP API, which allows seasoned programmers to take advantage of their experience to use just one API to program in sharedmemory and distributed-memory parallelism. In this paper, we took an existing plasma physics code that was programmed with MPI+OpenMP and ported it over to OpenMP Cluster. We also show that under certain conditions, the performance of OpenMP Cluster is similar to that of the MPI+OpenMP code.

**Keywords:** Parallel Programming, Directive-based Programming, Plasma Physics

## 1 Introduction

Plasma physics studies the properties and applications of plasma, a state of matter obtained when the electrons in a superheated gas are separated from their corresponding atoms and produce an ionized substance. Plasma, also called the fourth state of matter, represents more than 99% of the visible universe. Plasma physics has many applications: plasma propulsion (to build highly efficient propulsion engines for space ships), plasma medicine (used for sterilization and other treatments), astrophysics (to understand phenomena, such as solar winds and star properties), fusion energy (to develop clean energy reactors based on particle fusion), and others. Therefore, there has been a prominent interest in the scientific community in understanding plasmas. Such an understanding requires computer simulation and high performance computing (HPC). There are several reasons behind this. To fully grasp plasma behavior, it is necessary to deal with complex physics, solving sets of coupled nonlinear equations with sophisticated numerical methods. Also, some important phenomena in plasma physics implies the analysis of large-scale systems (fusion reactors or space effects) that may span vast regions and include a massive number of particles and parameters. Finally, evolution in time is also crucial, including a long series of timesteps. More recently, the scientific community has focused on developing multiphysics computer simulation codes for plasma research in HPC systems (Allmann-Rahn, Lautenbach, Deisenhofer, & Grauer, 2024; Choi, et al., 2018).

Programming plasma physics simulations (or any computer simulation for that matter) on HPC machines most likely requires the combination of programming paradigms: distributed-memory for communication across nodes, sharedmemory to exploit the computing power of multi-core processors, and singleinstruction-multiple-data (SIMD) to leverage accelerators. Consequently, a programmer is forced to learn multiple paradigms and use several libraries to build an application. A huge portion of simulation codes follow the MPI+X style, where the message passing interface (MPI) standard is combined with other standard interfaces that are meant to extract the intra-node performance. Would it be possible for a programmer to use a single parallel programming interface and run an application on a modern supercomputer? That is the claim of OpenMP Cluster (OMPC), a recent tool that leverages the well-known OpenMP standard to program applications on an arbitrarily large machine.

This paper describes our experience on migrating a pre-existing MPI+OpenMP plasma physics application to OMPC. We provide the design principles that regulated such migration, along with implementation details. In addition, we offer an experimental evaluation of both versions (traditional MPI+OpenMP and OMPC).

## 2 Background

### 2.1 BS-SOLCTRA

The Biot-Savart Solver for Computing and Tracing Magnetic Field Lines (BS-SOLCTRA) (Jiménez, et al., 2020) is a computer simulator created for the design and analysis of the Stellarator of Costa Rica 1 (SCR-1) (Coto-Vílchez, et al., 2020), a modular magnetic plasma confinement device, built at the Costa Rica Institute of Technology. BS-SOLCTRA is mainly used to determine if a particular modular coil-configuration would confine plasma. However, several other uses are possible. For instance, during the device development process, simulation results are gathered to produce scientific visualizations that convey meaningful information about physical phenomena inside the device chamber.

The original BS-SOLCTRA is a C++ code that was later parallelized with three standard technologies: vectorization, shared-memory programming through OpenMP pragmas, and distributed-memory programming with MPI. This standard implementation allows the code to scale to tens of thousands of computational cores (Jiménez, Meneses, & Vargas, 2021). The first versions of BS-SOLCTRA were optimized for Intel's Xeon Phi Knights Landing (KNL) architecture, where the AVX-512 vector operations are crucial to extract performance. The simulator itself implements a field-line tracing algorithm based on the Biot-Savart's Law to model a three-dimensional vacuum magnetic space within a modular stellarator. Derived plasma phenomena can be studied from the simulation results that are stored in particle trajectory output files. The coil structure is provided as an input to the simulator, which traces the trajectories as a set of particles moves through the magnetic fields created by the coils. Field lines create tracks over which particles will follow their trajectory.

Figure 1 provides a view of the execution flow of the original BS-SOLCTRA version. There are two main input sets for the simulator. First, a geometry for the coil collection. Such input parameter gives users the ability to understand how different coil configurations alter the physical variables of the confinement device.

Second, a set of initial positions of the particles. Each particle will describe a trajectory for which coordinates at several moments will be recorded. The set of particles is divided first among MPI ranks, having a single MPI rank per compute node. Later, each MPI rank will spawn OpenMP threads to be each scheduled on compute cores within a node. Each thread will be in charge of computing the trajectories for a subset of particles as they go through the magnetic fields generated by the set of coils. BS-SOLCTRA uses a fourth-order Runge-Kutta method (RK4) to compute the position of each particle at every moment in the simulation.

### 2.2 OpenMP Cluster

OpenMP Cluster (OMPC) is a directive and task-based programming model that extends OpenMP to support cluster programming (Yviquel, et al., 2023). Using OpenMP's own offloading standard, OMPC is capable of automatically distributing computational tasks to nodes on a distributed computing system. To achieve that, under the hood OMPC employs MPI and uses OpenMP's task dependency directives to determine data-dependencies between tasks, perform data transfers, dispatch tasks, gather execution results, and deal with failures (Di Francia Rosso & Francesquini, 2022). These features allow developers to focus on their application and avoid developing (and including) code specific to the distribution of their parallel applications, in their code base.

Listing 1 exemplifies the use of OMPC. It is a short yet informative application that performs a simple sequence of calculations. As any OpenMP annotated program, the code would still be valid and would produce the same output if we were to remove the lines beginning with #pragma omp. Line 5 tells OMPC

(via OpenMP pragmas) that the next block of code (lines 7-14) is to be treated as a parallel region. Line 5 then indicates that the loop indices (line 7) should not be broken into chunks and distributed for execution among multiple local execution threads, but rather be executed by a single thread (locally). Inside the loop body, we create two tasks. The first task (lines 8-10) performs the calculation of N, whereas the second (lines 11-13) the calculation of M. The omp target pragma creates a task and defines its data dependencies as well as its inputs and outputs. For instance, the first task has a data dependency to the value of N but also modifies its value as well as the value of the i variable (the second task, has similar behavior regarding to M and j). Note that, in total, 6 tasks will be created (2 per iteration, in a total of 3 iterations of the loop on line 7).

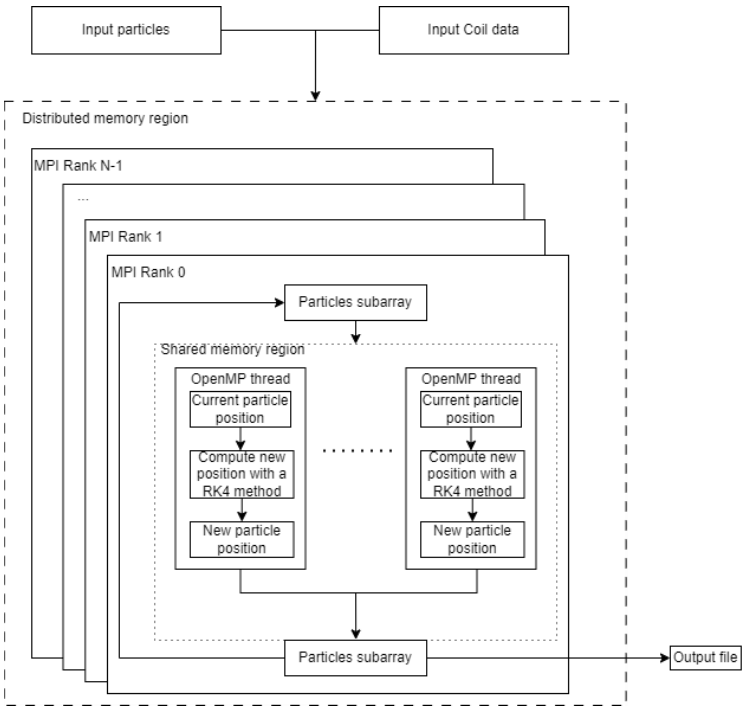


**Figure 1.** One Code structure of original MPI+OpenMP BS-SOLCTRA version

```
int main(){
  float N = 5.0;
  int i, j = 1;
  #pragma omp parallel
  #pragma omp single
  for (int k = 0; k < 3; k++){
    #pragma omp target depend(inout:N)\
    map(tofrom:N, i) nowait
    N *= i++;
    #pragma omp target depend(inout:M)\
    map(tofrom:M,j) nowait
    M /= j++; }
    Year := 0;
  printf("N value = %f||M value = %f\n", N, M);
  return 0; }
```

**Listing 2.** OMPC usage example

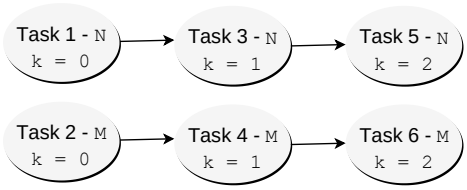


**Figure 2.** OMPC tasks created by the code in Listing1. Arrows indicate data dependencies between tasks (pointing from the provider to the consumer).

Automatically taking into account the data dependencies between these tasks (see Figure 2), OMPC will transparently distribute and schedule their execution among the available cluster nodes using MPI and a HEFT-based (Topcuoglu, Hariri, & Wu, 2002) scheduler. Tasks that are not linked by any dependency between them, are executed in parallel as soon a computing node is available. At the end of the execution of each task, the necessary data transfers are automatically performed between the computing nodes and the program execution continues.

### 2.3 Related Work

The BS-SOLCTRA code has already been used to port the original MPI+X version to other parallel programming tools. Jiménez, Meneses, and Vargas (2021) used the Charm++ parallel-object framework to provide BS-SOLCTRA with a dynamic mechanism to adapt to load imbalance scenarios. The Charm++ version of BS-SOLCTRA was able to automatically balance the load in an uneven execution, increasing CPU utilization from 45.2% to 80.2%, and getting a 1.64X speedup over the base MPI+X implementation. The Charm++ version scaled up to 1,024 nodes on Theta Supercomputer at Argonne National Laboratory. That version also uses the migratability of objects to offer a fault tolerance checkpointing capacity. In other work, Jiménez et al. (2022) used the offload features of the OpenMP standard to port the original BS-SOLCTRA version and use a GPUenabled cluster. Their paper shows how portability across different accelerator architectures is possible with OpenMP directives, including both prescriptive and descriptive modes. Their results include a 6X speedup over a CPU version of the code. Also, they argue that a complete performance-portability of the code is not fully automatic, as final optimizations are architecture-dependent.

## 3 Design and Implementation

### 3.1 Design principles

As there are no dependencies between the particles in this simulation, we defined each task as running the simulation on a subset of all particles. Data transfers are only necessary when launching the tasks and when collecting the final result of the simulation; even so, we wanted to minimize data transfers between nodes, and we also wanted to minimize synchronous execution of the code, so we stated the data dependencies between the tasks and relied on the OMPC runtime to schedule the transfers and the execution.

### 3.2 Implementation Details

To run the simulation, it is necessary first to map the data to the different processes that are going to be working on the problem. In MPI+OpenMP we would use point-to-point communication to send data from Rank 0 to the rank where the specific data is needed and we would use collective communication to broadcast data needed by all ranks to perform their computations, as we can show in Listing 2. Meanwhile, in OMPC we can send the data in each omp target construct, which would be equivalent to sending data to a specific rank, we can also map data with the omp target enter data construct, which allows any created

task to access the data, similar to a broadcast in MPI. The depend clause creates dependencies between the tasks based on the data.

In Listing 3 we show a simplified version of our communication code with OMPC, first we create our parallel region, and within it, we enter a region that will be executed by a single process, the parent process in Figure 3, this parent process is tasked with mapping all the data with omp target enter data, adding the dependencies, and assigning the other processes with running the simulation (runParticles) which the necessary data. In this program, most of the data is constant and needed by all the processes; however, the particles can be divided, as each task only deals with a portion of the data. In this case, we decided to map all the particles to all processes because the number of particles made this more efficient than mapping each subsection of the array to each task. This decision meant that there was now a loss in parallelism because there was a dependency between the tasks on the particle array, making the simulation sequential. We fixed this by explicitly stating which particles are needed by each task both when it was mapped (as an out dependency) and when launching the tasks (as an in dependency). After the simulation, we map the particle array from the other processes to the local array, and as we can see on line 9, particle_ptr points to the data of the particles vector, so it ends up with the final data.

Within each task, at each time step, we take advantage of data parallelism by using a omp parallel for construct and apply the simulation to each particle in the particle array.

```
// Rank 0 broadcasts global data among all ranks
  MPI_Bcast(&coil_data, 1, MPI_TYPE, 0, MPI_COMM_WORLD);
  …
  // Every rank then calculates their displacement and share of the
particle vector and reports them to the Rank 0.
  auto myShare=…; auto myDisplacement = …; MPI_Gather(…);
  // Finally Rank 0 scatters the particle vector among the ranks.
  MPI_Scatterv(…); MPI_Barrier(MPI_COMM_WORLD); runParticles(…);
  // Finally, we gather the results
  MPI_Gatherv(…);
```

**Listing 2.** Extract of the MPI+OpenMP code

```
#pragma omp parallel
#pragma omp single
{
  auto * data_pointer = data_vector.data()
  #pragma          omp          target          enter          data
map(to:data_pointer[:data_vector.size()])
depend (out:data_pointer_ptr) nowait
  …
  Cartesian * particle_ptr = particles.data();
  #pragma          omp          target          enter          data
map(to:particle_ptr[:particles_size()])
depend(iterator(j=0:particle_num),out:particles_ptr[j]) nowait
  for(int i = 0; i < nworkers; ++i) {
    #pragma                        omp                        target
depend(iterator(j=start:start+share),in:particles_ptr[j]) nowait
    runParticles(…)}
#pragma omp target exit data map(from: particles_ptr[:particle_num])
depend(iterator(j=0:particle_num), out: particles_ptr[j]) nowait
}
```

**Listing 3.** Communication code of the OMPC version

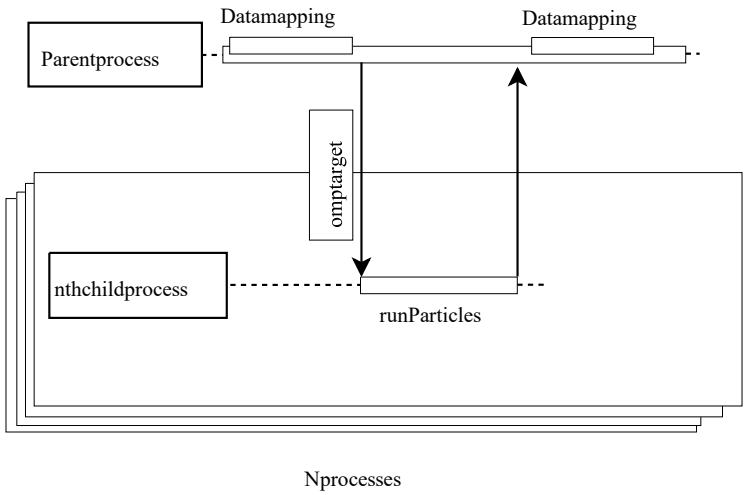


Figure 3. Simplified lifetime of the processes

# 4 Experimental Results

## 4.1 Experimental Setup

We conducted our experiments on the Kabré supercomputer at the National High Technology Center (CeNAT) of Costa Rica. This supercomputer has more than 50 nodes for applications in scientific computing, bioinformatics, data science, and artificial intelligence. The largest partition, called nu (for dog in a native Costa Rican language), has 32 Intel Xeon Phi KNL nodes, each with 64 physical cores, 96 GiB of main memory. The software environment of the experiments can be seen in Table 1.

Table 1. Software configuration

| Software | Version |
|---|---|
| Operating System | CentOS 7 |
| Kernel | 3.10.0-1160.83.1.el7.x8664 |
| Singularity | 3.8.1 |
| OMPC clang | 15.0.1 |
| OMPC Docker image | ompcluster/runtime-dev:ubuntu20.04-cuda11.2-mpich |
| MPI Library | MPICH 4.0.1 |

Table 2 shows the variables used in the experiments. We ran all the experiments using the same singularity container. We compiled both programs using the modified clang++ compiler provided by the OMPC team. We used the -ffast-math, -march=knl, -mavx512f, -mavx512pf, -mavx512er, and -mavx512cd compiler flags to take advantage of the characteristics of the KNL architecture. During the tests, we found that the -O3 flag introduced optimizations that did not allow a proper measurement of the effect of the OMPC framework. Therefore, we made measurements using the -O0 flag. We performed 10 repetitions for each treatment with 50 steps and 64 threads per node, reporting the average execution time and standard deviation. Only one MPI rank is run per node.

**Table 2.** Variables used in the experiment

| Variable | Description |
|---|---|
| Number of ranks | Ranks used in each run |
| Number of steps | Simulation steps |
| Number of particles | Particles being simulated each step |
| OMP_NUM_THREADS | The number of threads to be used by OpenMP |

## 4.2 Strong Scaling Experiment

To estimate the slowdown of OMPC over MPI+OpenMP we performed a strong scaling experiment. We ran the simulation with 2, 4, 8, 12, 16 and 20 computing nodes and kept the number of particles constant at 40,960. As the OMPC version needs a node to be the head node, we allocated one extra node on each run to have the same number of computing nodes working on both versions. We can see the results of this comparison in Figure 4. We compute the slowdown as $\frac{T_{OMPC}}{T_{MPI+OpenMP}}$ and show the results in Figure 5. This shows that the slowdown was below 20% in all our tests.

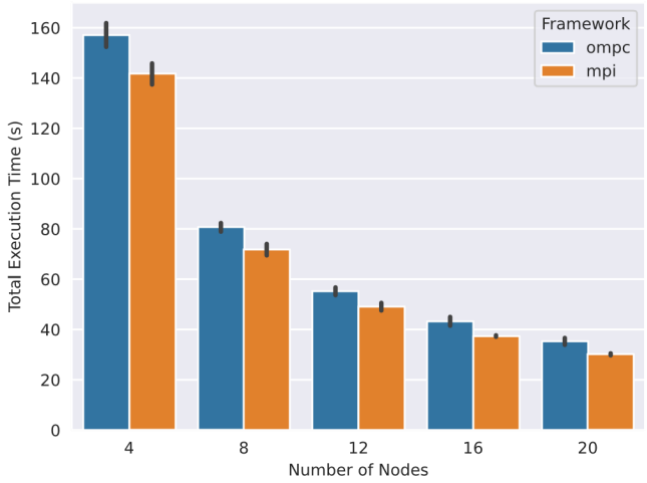


**Figure 4.** Strong scaling experiment with 40,960 particles

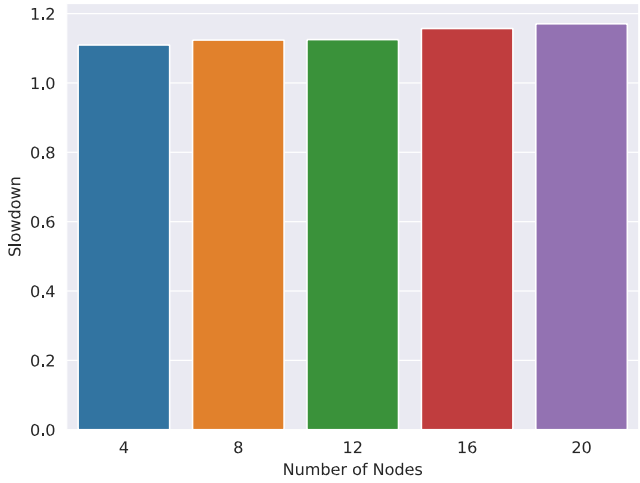


Figure 5. Slowdown of OMPC compared to MPI+OpenMP

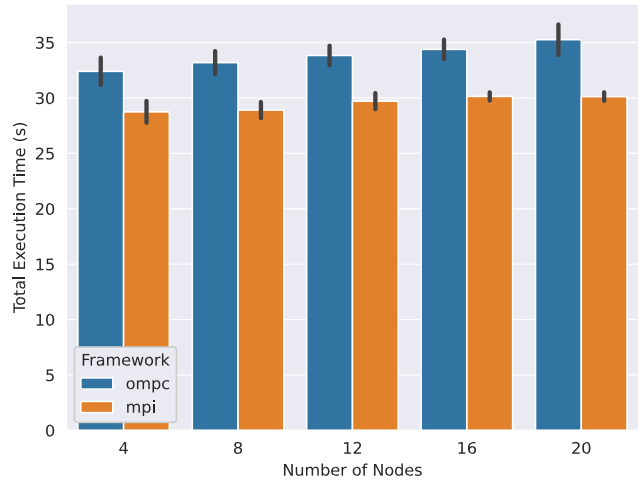


Figure 6. Weak scaling experiment with 2048 particles for each rank

### 4.1 Experimental Setup

We did a weak scaling experiment to check how OMPC scales in comparison to MPI+OpenMP. We ran 2,048 particles for each node, going from 8,192 particles with four nodes to 40,960 particles for 20 nodes. The result of this experiment can be seen in Figure 6 where we can see that both frameworks scale in a similar way, with a mean growth rate of 0.34 seconds in the MPI + OpenMP version and 0.71 seconds in the OMPC version.

## 5 Discussion and Final Remarks

In this paper we ported an existing simulation that uses MPI+OpenMP onto the OMPC framework. This involved rethinking the way that data was being mapped and distributed to the different computing units and thinking in terms of tasks rather than ranks. On the other hand, we were able to use the same computing kernel with very little modification. Our results show that the slowdown of the OMPC framework compared

to the MPI+OpenMP code is low for our simulation without compiler optimizations. OpenMP Cluster shows promise by allowing programmers to use a single API for distributed memory under a tasking interface.

**Acknowledgments.** This research was partially supported by a machine allocation on Kabré supercomputer at the Costa Rica National High Technology Center.

## ORCID iD

Christian Asch https://orcid.org/0000-0002-3111-4858
Emilio Francesquini https://orcid.org/0000-0002-5374-2521
Esteban Meneses https://orcid.org/0000-0002-4307-6000

## References

Allmann-Rahn, F., Lautenbach, S., Deisenhofer, M., & Grauer, R. (2024, March). The muphyII Code: Multiphysics Plasma Simulation on Large HPC Systems. *Computer Physics Communications, 296*, 109064. doi:https://doi.org/10.1016/j.cpc.2023.109064

Choi, J. Y., Chang, C.-S., Dominski, J., Klasky, S., Merlo, G., Suchyta, E., . . . Wood, C. (2018). Coupling Exascale Multiphysics Applications: Methods and Lessons Learned. *2018 IEEE International Conference on e-Science and Grid Computing* (pp. 442-452). Amsterdam, Netherlands: IEEE. doi:10.1109/eScience.2018.00133

Coto-Vílchez, F., Vargas, V. I., Solano-Piedra, R., Rojas-Quesada, M. A., Araya-Solano, L. A., Ramírez, A. A., . . . Arias, S. (2020, July 8). Progress on the small modular stellarator SCR-1: new diagnostics and heating scenarios. *Journal of Plasma Physics, 86*(4), 815860401. doi:10.1017/S0022377820000677

Di Francia Rosso, P. H., & Francesquini, E. (2022). OCFTL: An MPI Implementation-Independent Fault Tolerance Library for Task-Based Applications. In I. Gitler, C. J. Barrios Hernández, & M. Esteban (Ed.), *High Performance Computing. 8th Latin American Conference, CARLA 2021, Guadalajara, Mexico, October 6–8, 2021, Revised Selected Papers. 1540*, pp. 131-147. Springer, Cham. doi:10.1007/978-3-031-04209-6_10

Jiménez, D., Campos-Duarte, L., Solano-Piedra, R., Araya-Solano, L. A., Meneses, E., & Vargas, I. (2020). BS-SOLCTRA: Towards a Parallel Magnetic Plasma Confinement Simulation Framework for Modular Stellarator Devices. In J. L. Crespo-Mariño, & E. Meneses-Rojas (Ed.), *High Performance Computing. 6th Latin American Conference, CARLA 2019, Turrialba, Costa Rica, September 25–27, 2019, Revised Selected Papers. 1087*, pp. 33-48. Springer, Cham. doi:10.1007/978-3-030-41005-6_3

Jiménez, D., Herrera-Mora, J., Rampp, M., Laure, E., & Meneses, E. (2022). Implementing a GPU-Portable Field Line Tracing Application with OpenMP Offload. In P. Navaux, C. J. Barrios H, C. Osthoff, & G. Guerrero (Ed.), *High Performance Computing. 9th Latin American Conference, CARLA 2022, Porto Alegre, Brazil, September 26–30, 2022, Revised Selected Papers* (pp. 31-46). Springer International Publishing. doi:10.1007/978-3-031-23821-5_3

Jiménez, D., Meneses, E., & Vargas, V. I. (2021, July 17). Adaptive Plasma Physics Simulations: Dealing with Load Imbalance using Charm++. *PEARC '21: Practice and Experience in Advanced Research Computing. Article No. 3*, pp. 1-8. New York, NY, USA: Association for Computing Machinery. doi:10.1145/3437359.3465566

Topcuoglu, H., Hariri, S., & Wu, M.-Y. (2002, March). Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems, 13*(3), 260-274. doi:10.1109/71.993206

Yviquel, H., Pereira, M., Francesquini, E., Valarini, G., Leite, G., Rosso, P., . . . Araujo, G. (2023, January). The OpenMP Cluster Programming Model. *ICPP Workshops '22: Workshop Proceedings of the 51st International Conference on Parallel Processing. Article No. 17*, pp. 1-11. Bordeaux, France: Association for Computing Machinery. doi:10.1145/3547276.3548444