Parallelization of a Stellarator Simulation Code in Plasma Physics

Luis Diego Chavarría
School of Computing
Costa Rica Institute of Technology
luchavarria@ic-itcr.ac.cr

Esteban Zamora, Iván Vargas
School of Physics
Costa Rica Institute of Technology
estebanzp84@yahoo.es
ivargas@tec.ac.cr

Esteban Meneses
Advanced Computing Laboratory
Costa Rica National High Technology Center
School of Computing
Costa Rica Institute of Technology
esteban.meneses@acm.org

***Abstract*—There are multiple applications of plasma. To better understand the fundamental characteristics of plasma, physicists at Costa Rica Institute of Technology built a plasma confinement device, code named SCR-1. This device was paired with a simulation code. The computer program emulates the dynamics of plasma as it is confined within SCR-1. This program is computationally intensive. To alleviate the performance demand, it was necessary to apply high performance computing machinery. In this paper we present the work on improving the performance of the this application by implementing vectorization and parallelism through distributed and shared memory. We also include experimental results that display acceleration factors higher than 30X. Additionally, we show speedup results for weak and strong scaling experiments.**

***Keywords*—Plasma physics, stellarator, simulation, parallelism, vectorization.**

I. Introduction

A stellarator is a device to confine toroidal plasma that uses effects arising in the absence of toroidal symmetry to maintain the magnetic configuration without the need for current drive [1]. The concept is characterized by two contradictory features in relation to its counterpart concept *tokamak*: its relative easy operation and its complexity in design and construction. This motivated the Plasma Group of the Costa Rica Institute of Technology in the year 2009 [9] to start the development of the *Stellarator of Costa Rica 1* (SCR-1), with the objective of improving the engineering of the design of a small 2-field period modular stallarator. The SCR-1 has twelve modular cooper coils that produce magnetic fields that confine the plasma (see Fig. 1). These magnetic fields are important to define the best *electron cyclotron frequency heating* (ECH) system and they are also needed to evaluate the confinement of the device [10].

In the SCR-1 verification process, it was necessary to program a code named *Biot-Savart Solver for Computing and Tracing magnetic fields* (BS-SOLCTRA). Written initially in Matlab, the code simulates a 3D stationary magnetic field using a Biot-Savart law and a reduced model of the twelve modular coils. Currently, BS-SOLCTRA is an essential tool in the development of SCR-1 and its results are present in three international contributions (one scientific paper and two conferences) [8] [11] [7]. Because of performance demands, BS-SOLCTRA was migrated to C++, splitting the tool into two modules: the computationally heavy one in C++ and a light one responsible of drawing of the magnetic fields in Matlab. In this paper we present the work made to improve the performance of the C++ module through parallelization of the code by using MPI and OpenMP in the Kabré supercomputer at the Costa Rica National High Technology Center.

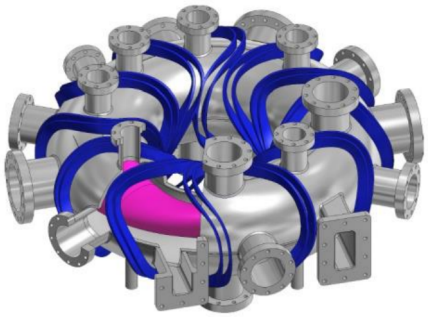

Fig. 1: CAD drawing of SCR-1 coils and vacuum vessel [9]

This paper is organized as follows. Section II provides a background of the initial state of the code, how it works, its composition and its implications in the SCR-1. Then the methodology on the parallelization of the code is shown in section III. Section IV presents the results of the execution on the Kabré supercomputer and the analysis of the different experiments that we run and the configuration to achieve optimal performance. And finally the conclusions are presented in section V.

II. Background

Initially, the code was completely developed in Matlab. After realizing how computational demanding it was, the simulation code was migrated to C++. This code implemented a fourth order Runge-Kutta's method based on [2] and [5] for tracing of paths in a magnetic field. At high level, before the magnetic fields calculations, the code loads the information of the twelve modular coils, one Cartesian point per grade. This information needs to be loaded only once and remains constant throughout the entire execution.

After loading the coil information, the tool calculates $\hat{e}_n$, the unit vector along each segment of each coil [4]. This value is calculated as the difference between the next point and the current point divided by the norm of the current point:

$$\hat{e}_n = \frac{P_{n+1} - P_n}{norm(P_n)}$$

Once the coil information is loaded and the $\hat{e}_n$ for every point of the coils, a loop is executed to get the Runge-Kutta calculation. This loop keeps its flow following the algorithm in Fig. 2. Note that the calculation of $\hat{e}_n$ is made only once for all the execution. This is because the coils are fixed and, consequently, the $\hat{e}_n$ are same for all the Runge-Kutta calculations.

$P_0 \leftarrow StartPoint$
loop
  $K_1 \leftarrow$ MAGNETIC FIELD(P_0)
  $P_1 \leftarrow \frac{K_1}{2} + P_0$
  $K_2 \leftarrow$ MAGNETIC FIELD(P_1)
  $P_2 \leftarrow \frac{K_2}{2} + P_0$
  $K_3 \leftarrow$ MAGNETIC FIELD(P_2)
  $P_3 \leftarrow K_3 + P_0$
  $K_4 \leftarrow$ MAGNETIC FIELD(P_3)
  $P_0 \leftarrow P_0 + \frac{K_1 + 2*K_2 + 2*K_3 + K_4}{6}$
end loop

Fig. 2: Algorithm for the fourth order Runge-Kutta

The starting point of the loop in Fig. 2 is taken as the start point of the trajectory of the particle for which we want to calculate the magnetic fields.

The composition for the magnetic field calculation is given by two nested loops where the code executes a set of operations based on the given point, the information of the coils initially loaded, and the $\hat{e}_n$ is calculated at the beginning of the simulation. The result of each iteration is taken as a point of the magnetic field of the current trajectory. This result is saved to a file to load and visualize all of them in Matlab.

III. Methodology

The C++ sequential implementation of the algorithm in Fig 2 was used as the starting point. This sequential version was defined as a reference code for later developments. Therefore, results between the sequential version and any parallel version had to be equivalent. Since this original version of the algorithm implemented in C++ was a migration from Matlab, it was aimed to match every operation in the Matlab code in C++. Because of this, a cleanup and refactorization of the code was also necessary to provide a more flexible ground to attempt parallelization of the functions. Besides, additional functionality was added to facilitate the usage of the simulation.

A. Computational Model

The main goal of this work is to run a simulation on a parallel computing system. We understand such a system as a collection of n nodes connected through a high-speed network. Although the topology of the network usually has an impact on performance, we ignore such property and assume that every node can communicate with every other node.

Each node is conceived as a multicore processor. Since the cores could be hyper-threaded or not, we say that each multicore process will have c computational workers. Those workers within the same processor share a common memory space. In total, the system contains $p = n \times c$ *processing elements*. Hence, p describes the size of the system in terms of its computational capacity. Each core has an internal level of parallelism exposed to the programmer: vectorization units. These hardware components allow the same operation to be applied to multiple operands and represent a broadly available feature in current computer architectures.

B. Parallel Programming

1) Vectorization: The first step of the parallelization of the code was to reorganize the data structures with the aim of using vectorization. This is a single-instruction-multiple-data (SIMD) approach, where the compiler takes operations in loops to use the VPU (*Vector Processing Unit*) instead of the regular ALU. How this is performed by the compiler depends on the ISA available in the processor. On a code basis, the compiler might detect where to vectorize, but also, the programmer can explicitly indicate where and how to add vectorization to the application.

2) Shared Memory Programming (OpenMP): We used *shared-memory programming* to provide parallelism at the core-level. This strategy assumes that all cores within the same node can access a common memory address space. The usual implementation of this strategy is to create a computational thread on each core.

There are several libraries to write multithreaded code. We chose OpenMP (Open Multi Processing) to program nodes. It is composed by a set of compiler directives: pragmas that hints to the compiler that certain pieces of codes can be parallelized. There is a wide range of directives in OpenMP. However, we concentrated on a few directives that bring substantial performance benefits if appropriately used in the right type of programs. In particular, the pragma `parallel` establishes a parallel region that is executed by multiple threads. That is where all parallelism starts. Pragma `for` precedes a loop and distributes independent iterations of the loop among the running threads. Finally, pragma `simd` signals a loop that has to be vectorized.

3) Distributed Memory Programming (MPI): We used *distributed memory programming* to provide parallelism across the system. This is the node-level parallelism. At this level, the usual implementation is to create a computational *process* in each node. We chose the *Message Passing Interface* (MPI) standard to implement our parallel code. MPI is a library to use message-passing [6] between the different processes. Since this approach assumes distributed memory, not all processes have to run on the same computer. Instead you might a cluster with the processes being executed in different nodes.

The MPI library has a set of more than 500 functions to exchange messages between the processes that are running [3]. That includes functions for operations such as *send*, *receive*, and *broadcast*, and more advanced operations like *gather*, *scatter*, and *reduction*. The MPI library also provides operations for process synchronization, such as barriers, and provides options for synchronous or asynchronous message passing. Another requirement of MPI is that a program cannot being executed on its own: it needs of a set of wrappers.

C. Performance Measures

To understand how the parallel implementation performs, we resort to the traditional indicators *speedup* and *efficiency*. Let us define W as the total amount of work a simulation must perform to achieve its goal. In our case this could mean the total number of operations necessary to compute the magnetic fields of all the trajectories in the input. Let T_1 represent the total execution time of the sequential version of the simulation while performing work W. Let T_p represent the total execution time when using p processing elements. The ratio T_1/T_p is known as *speedup*. Ideally, if p processing elements are used, we expect a speedup of p. Reality, however, presents a much different speedup curve, usually deviating from the identity function as p increases. The overhead of parallel computing (thread/process creating and destruction, syncronization) and the sequential part of the code usually explain why linear speedup is hard to achieve. The other performance descriptor is the *efficiency*, defined as the ration of *speedup* and *p*, and ranges between 0 and 1. Intuitively, efficiency represents the utilization of the system (is the fraction of the resources that are being effectively used in a parallel system). The ideal value for efficiency is 1, but that value can be hardly achieved. When computing speedup and efficiency, the size of the problem is usually kept the same while the number of processing units is increased. So, W remains constant as p increases. That is called *strong scaling*. On the other hand, *weak scaling* we when W increases linearly with p. In a weak-scale experiment, the total amount of work is usually Wp.

D. High Performance Computing System

We collected the results using the *Kabré* supercomputer, hosted at the Costa Rica National High Technology Center (CENAT). We used specifically the queue compound by nodes made up of Intel Xeon Phi (codenamed *Knights Landing*). Each of these nodes have 64 cores, each with 4 hyper-threads. Kabré has the Intel's version of both MPI and OpenMP libraries installed, to take maximum advantage of the hardware. The parallel implementation of the simulation was submitted through Kabrés' job submission system. Every data point reported is the average of 10 executions.

E. Parallel Application

Both libraries, MPI and OpenMP, were used on the parallelization of the different executions of the Runge-Kutta method. Since we have a defined amount of processes being executed, when the program is iterating on the initial set of trajectories given by the user, we check if the number of iteration corresponds to the "my" MPI *rank* (MPI unique identifier number of the process). To do this, we calculate the module of the current particle count with the MPI *rank*. If it equals zero, the given process executes the Runge-Kutta algorithm in Figure 2; if not, the process skips it until it gets one that correspond to its rank.

IV. Results and Analysis

To test and benchmark the parallelization of the code, we have separated the results in two sections: vectorization improvements only (section IV-A), and OpenMP and MPI (section IV-B). On both sets of experiments, we chose one point and iterated n times using it depending on the experiment. We use the same $StartPoint$ in Fig. 2 in all iterations, ensuring that all the iterations have the same work load in all the experiments.

A. Vectorization Only

We isolated the vectorization by turning the OpenMP and MPI off and comparing the execution time of the application with and without (`-no-vec` and `-qopenmp-stubs` compiler options) vectorization.

The results obtained from this experiment were that without vectorization the code lasts in average 829.28s, and with vectorization, 97.93s. These results show that only with the vectorization provided on the KNL in the Kabré supercomputer, we can get an acceleration of 8.47 with the vectorized version of the application.

B. OpenMP and MPI

For the OpenMP and the MPI experiments, we show the results of the speedup for weak scaling and for strong scaling (section III-C). These experiments are contained into a single KNL socket, consequently, we have a maximum number of simultaneous workers of 256 (4 threads on 64 cores). Also, for both experiments we changed the number of MPI ranks and OpenMP threads, spreading the threads across all cores.

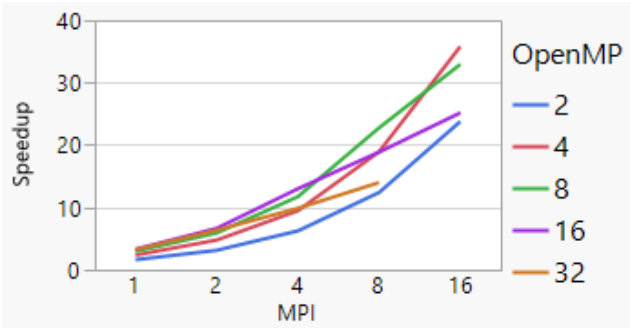

Fig. 3: Speedup results of BS-SOLCTRA on the KNL for the weak scaling scenario

The weak scaling results are shown on Fig. 3. We clarify that the line for 32 OpenMP threads does not have 16 MPI ranks because that would exceeded the total threads available in one KNL, which would generate an overbook of resources. In this graph we can see that the lines follow an increment curve, some with a slope higher than others.

In Fig. 3 we see that the speedup does not get increased as the number of threads is increased. We can see that for OpenMP=2 (2 OpenMP thread per MPI rank) is the lower one with incremental breaks on each MPI rank increment. The same happens for the case of OpenMP=4. On the contrary, for OpenMP=8, the incremental break stops when the MPI=16, and even this point has a lower speedup than OpenMP=4. This effect becomes worst for OpenMP=16. For this case we have an incremental break until MPI=4, where the slope changes to be lineal. For the OpenMP=32 the slope is always lineal.

From the results in Fig. 3, we see the common factor for these breaks as the moment when the number of threads per

core is increased from 1. So we can conclude that speedup rises gradually while the the number of threads per core is kept to 1, then it passes to be lineal.

We finally got that the higher speedup for weak scaling when MPI=16 and OpenMP=4.

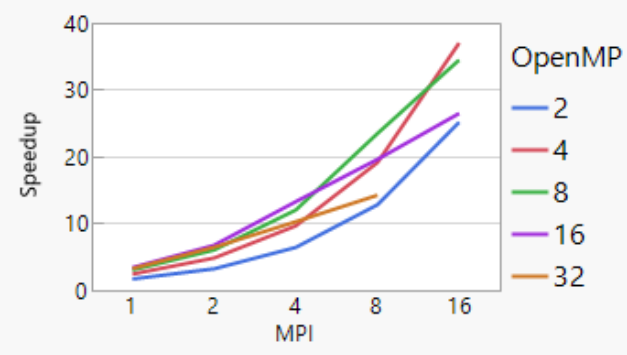

Fig. 4: Speedup results of BS-SOLCTRA on the KNL for the strong scaling scenario

Once having the results for weak scale, we ran the experiments for strong scale, which results are in Fig. 4. We can note that this graph follows the same trend that the graph in figure 3, where we observe a slope rises gradually in the speedup until the threads per core is increased from 1, where the increment changes to lineal.

V. Conclusion and Future Work

In this paper we presented a work made to improve the performance of the BS-SOLCTRA application using parallelism. We showed how the performance can be improved by taking advantage of hardware for vectorization. Finally, we showed the results of the speedup on experiments for weak scaling and strong scaling and, based on those results, we conclude that the BS-SOLCTRA application is scalable.

Physicists at Costa Rica Institute of Technology are planning to build another Stellarator with 24 coils instead of 12. We propose as future work the adaptation of the BS-SOLCTRA to this new model and to explore and analyze the results with that configuration. Also, we will compare those results with this work to explore the behavior of the parallel implementation of the algorithm at higher workloads.

References

[1] A. H. Boozer, "What is a stellarator?" *Physics of Plasmas*, vol. 5, p. 1647–1655, May 1998.

[2] A. S. Chai, "Error estimate of a fourth-order runge-kutta method with only one initial derivative evaluation," in *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, ser. AFIPS '68 (Spring). New York, NY, USA: ACM, 1968, pp. 467–471. [Online]. Available: http://doi.acm.org/10.1145/1468075.1468144

[3] G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*, 1st ed. Boca Raton, FL, USA: CRC Press, Inc., 2010.

[4] S. P. Hanson, James D.; Hirshman, "Compact expressions for the biotsavart fields of a filamentary segment," *Physics of Plasmas*, July 2002.

[5] H. A. W. L. F. Shampine, "Comparing error estimators for runge-kutta methods," *Mathematics of Computation*, vol. 25, p. 445–455, July 1971.

[6] P. Pacheco, *An Introduction to Parallel Programming*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[7] R. Solano-Piedra, A. K¨hn, V.I. Vargas, F. Coto-Vílchez, M.A. Rojas-Quesada, D. López-Rodríguez, J. Sánchez-Castro, J. Asenjo and J. Mora., "First MHD equilibrium characterization and electromagnetic waves interaction on hydrogen plasma in the SCR-1 Stellarator," in *4th European Physical Society Conference on Plasma Physics (EPS)*, Shanghai, China, 26 - 30th June 2017.

[8] R. Solano-Piedra, V. I. Vargas, A. Köhn, F. Coto-Vílchez, J. Sanchez-Castro, D. Lopez-Rodríguez, M. A. Rojas-Quesada, J. Mora, and J. Asenjo, "Overview of the SCR-1 Stellarator," in *23rd IAEA Technical Meeting on the Research Using Small Fusion Devices (23rd TM RUSFD)*, Santiago, Chile, March 2017.

[9] V. I. Vargas and J. Mora and J. Asenjo and E. Zamora and C. Otarola and L. Barillas and J. Carvajal-Godínez and J. González-Gómez and C. Soto-Soto and C. Piedras, "Implementation of Stellarator of Costa Rica 1 SCR-1," 2016.

[10] V. I. Vargas, J. Mora, J. Asenjo, E. Zamora, C. Otárola, L. Barillas, J. Carvajal-Godínez, J. González-Gómez, C. Soto-Soto, and C. Piedras, "Constructing a small modular stellarator in latin america," *Journal of Physics: Conference Series*, vol. 591, no. 1, p. 012016, 2015. [Online]. Available: http://stacks.iop.org/1742-6596/591/i=1/a=012016

[11] V.I. Vargas, J. Mora, C. Otarola, R. Solano-Piedra, J. Asenjo, F. Coto-Vílchez, J. Sanchez-Castro, L.A. Araya-Solano, A.M. Rojas-Loaiza, J.M. Arias-Brenes, J. F. Rojas, J.I. Monge, N. Piedra-Quesada, D. López- Rodríguez, M.A. Rojas-Quesada and L. Barillas, "Engineering overview of the Fusion Research in Costa Rica: SCR-1 Stellarator and Spherical Tokamak MEDUSA-CR," in *27th IEEE Symposium on Fusion Engineering (SOFE)*, Shanghai, China, June 2017.