Parallel Mixing Chamber Application using OpenMP and OpenACC

Glener Lanes Pizzolato, Natiele Lucca, Mariana Toledo Costa, Claudio Schepke

Federal University of Pampa – Alegrete – RS – Brasil
{glenerpizzolato,natielelucca,marianatoledo}.aluno@unipampa.edu.br
claudioschepke@unipampa.edu.br

Abstract. An application was previously developed to simulate mixing chamber problems. So it is possible to propose new catalysis substances, develop new propulsive engines, or evaluate fuel combustion computationally. However, the discrete representation of a simulation has an expressive processing time. In this article, parallelization techniques are proposed and evaluated for the mixing chamber application to run the simulations in multicore and GPU architectures. The results obtained show that the application was 500% faster in multicore and 50% in a Quadro M5000 GPU architecture.

1 Introduction

Compilation directives make it possible to parallelize an application by adding comments (pragmas) to the source code, generating minimal impacts on the original version [12], or even worsening the performance of the original code [5]. Using such a parallelism approach requires knowing the application and identifying in advance where parallelism can be applied. Directives can also generate different types of concurrent execution (thread, loops, vector instructions, GPU kernel, ...).

In this article we use a mixing chamber simulation application as previously discussed in [13]. This application has an expressive simulation time. The **objective** of the work is to reduce the simulation time and to evaluate ways to inject parallel compilation directives and their impacts on performance and code change.

The **contributions** of this paper is to accelerate a mixing chamber application using OpenMP and OpenACC.

The remainder of the paper does organize as follows. Section 2 presents the related work. Section 3 presents the functionalities of the application and the structure of the code. The development of the parallelization does detail in Section 4. The methodology of the experiments is described in Section 5. Section 6 shows performance results for the different parallel approaches. At last, Section 7 presents the conclusion and future work of this paper.

2 Related Work

This chapter covers related works that use parallel APIs for concurrent application execution. Table 1 shows a comparison of the parallel programming interfaces and objectives of each of the eight selected works.

Table 1. Related Works

Article Name	**API Used**	**Kind of Work**
[**1**]-Practical Parallelization of Scientific Applications with OpenMP, OpenACC and MPI	OpenMP, OpenACC and MPI	Realizes a parallelization of four sequential applications
[**9**]-Performance analysis of CUDA, OpenACC and OpenMP programming models on TESLA V100 GPU	CUDA, OpenMP and OpenACC	Comparison of the APIs
[**14**]-Experiences in porting mini-applications to OpenACC and OpenMP on heterogeneous systems	OpenACC and OpenMP	Methodology for migrating small applications to OpenMP and OpenACC
[**7**]-Parallel programming languages on heterogeneous architectures using OpenMPC, OmpSs, OpenACC and OpenMP	OpenMPC, OpmSs, OpenACC and OpenMP	Review three programming frameworks that solve Jacobi's iterative method
[**15**]-Parallel Computation of a Dam-Break Flow Model Using OpenACC and OpenMP	OpenACC and OpenMP	Parallelize a flow simulation model *dam-break*
[**4**]-Concurrent Parallel Processing on Graphics and Multicore Processors with OpenACC and OpenMP	OpenMP, OpenACC and MPI	Explore a hybrid shared and distributed memory system
[**10**]-Power and energy footprint of OpenMP programs using OpenMP runtime API	OpenMP	Study of coarse and fine level characteristics of OpenMP programs for energy usage
[**8**]-Exploring loop scheduling enhancements in OpenMP	OpenMP	Presents a detailed performance study of the loop scheduling methods.
[**6**]-Parallel computation of aerial target reflection of background infrared radiation: Performance comparison of OpenMP, OpenACC, and CUDA implementations	OpenMP, OpenACC and CUDA	An application evaluated in distinct parallel implementations

All related works evaluate the OpenMP and parallel programming interfaces. Some also consider other interfaces. Article [1] evaluates the parallelization of 4 codes, but each uses a specific interface or architecture. Article [9] provides a specific GPU architecture assessment through two *benchmarks*. The authors of [10] also use benchmark to evaluate energy consumption impact. Article [14] evaluates the performance of 4 routines/methods in 5 computing environments, exploring heterogeneous parallelism (CPU/GPU) of the latest OpenMP and

OpenACC specifications. The parallelism evaluation of the interfaces that provide task parallelism uses the Jacoby method in [7]. However, the codes presented show loop parallelism in the OpenMP and CUDA versions. The results presented are limited to OpenMP 4, OpenACC, and OpenMPC. Article [15] parallelizes a 2D shallow water control application, using finite volume discretization with OpenMP and OpenACC. The speedup was 17.64 and 8.6 for each interface. Article [4] also uses the OpenMP and OpenACC interfaces for an application called MBFLO3 that provides a 3D general-purpose multidisciplinary solution using structured meshes discretized by finite volumes. The article [8] addresses aspects of OpenMP scheduling. Articles [15], [4] and [6] are similar to our proposal, essentially changing the application itself, while the article [7] is just the parallelization of a numerical method.

3 Mixing Chamber Application

The flowchart of the application is presented in Figure 1. The algorithm is composed essentially of a big loop where for each time step the physical properties are calculated. The algorithm executes for a number `N` of iterations predefined by the user, wherein each iteration step of the Runge-Kutta method of sixth-order executes following a time interval `dt` also predefined. In addition to the iterative step, the algorithm is composed of an initialization step responsible for reading input data and allocating and initializing variables. After the iterative step, it performs memory deallocations and application termination. Data structures, including those that store the discrete values of each physical property, as well as used constants, parameters, and limits, are maintained globally.

The operations performed within the main loop that iterates over the `dt` time step are: `acoustic font`: simulates a disturbance (turbulence), acting as a pulse inside the chamber; `rhs_euler`: performs all derivative calculations for all physical properties of interest; `bufferxy`: handles non-reflective boundary conditions; `rk steep`: concatenates the steps of the sixth-order Runge-Kutta method; `filtering`: treats numerical noise for each discrete point, considering seven neighboring points on each side; `ccpml`: it handles the boundary conditions of the region of interest.

4 Implementation

We develop two parallel versions of the application. The OpenMP version explores loop parallelism. For the OpenACC implementation, we provide kernels to the main functions of the code.

4.1 Parallelism with OpenMP

OpenMP is a programming model that has a simple and flexible interface for parallel application development [2]. OpenMP is standardized by directives. These

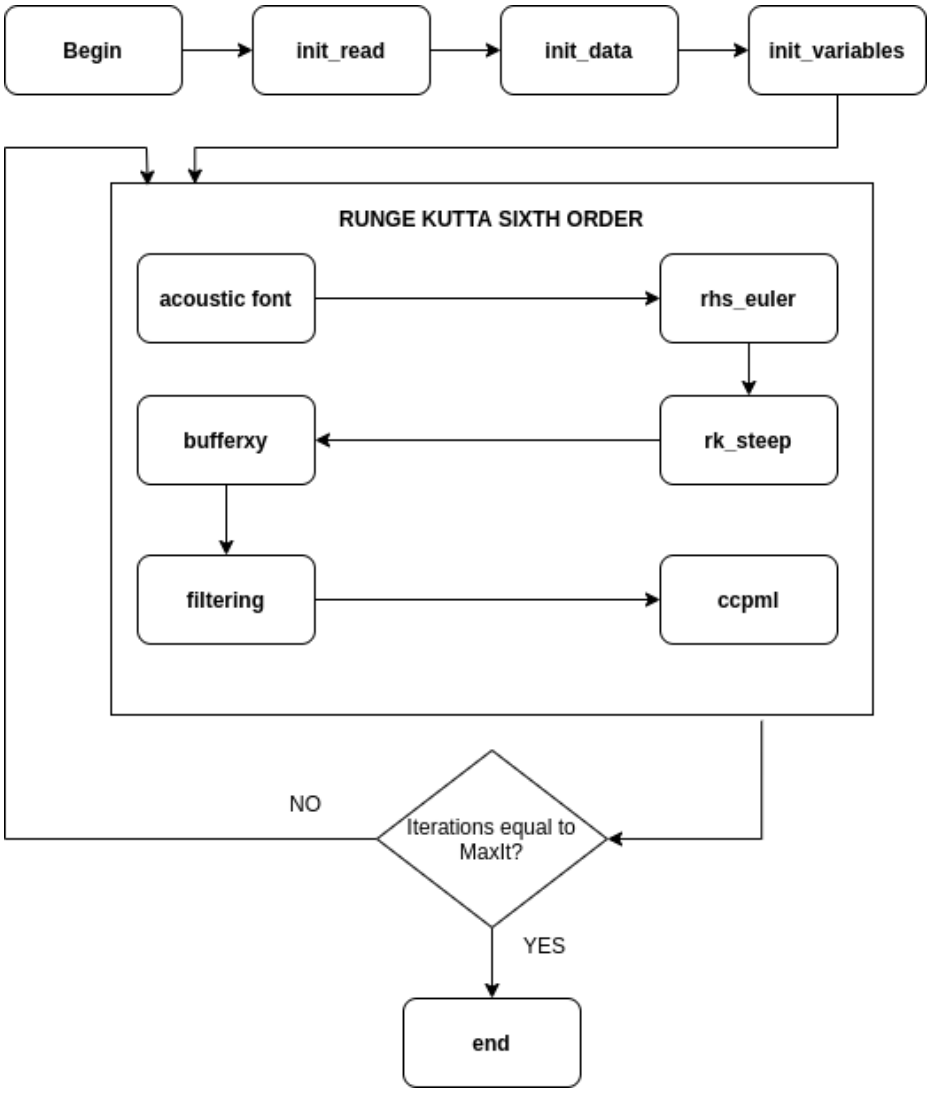


Fig. 1. Steps of the algorithm

directives define what parallelism will be applied at runtime. OpenMP has directives to launch and synchronize threads in loops, independent sections, and parallel regions, for example. It is also possible to define the number of threads if the variables are public or private and reductions, for example.

In our work, we add the `!$omp parallel for` directive to distribute the loop operations among the defined number of threads. The code has a large number of single and nested loops. Some of them have data dependencies, and can not be nested. A code snippet is present in Algorithm 1. The parallel threads operates over the x and y dimensions of the data. In this code, nk iterates each of the 5 physical properties. While this is not the case for this example, in other cases there are interactions among the data of the physical properties in a specific loop.

Similar parallel loops are localized and injected in all 6 routines of the iterative step. In the tests, we run individual experiments for each evaluated number of threads.

Algorithm 1 Code region used OpenMP

```
1   do k=1,nk
2   !$omp parallel do private(i,j)
3     do i=ii,fi
4       do j=j1,j2
5            Du=0.d0
6            do s=-1,5
7               Du=Du+dfb(s)*u(k,i,j+s)
8            end do
9            u(k,i,j) = u(k,i,j)-sigmaf*Du
10      end do
11    end do
12    !$omp end parallel do
13    end do
```

4.2 Parallelism with OpenACC

OpenACC is an API based on directives for developing parallel applications on heterogeneous architectures, available for C/C++ and FORTRAN [3]. Those directives specify loops and code blocks that can be offloaded from the CPU to an attached accelerator [11].

OpenACC allows the execution of an instruction on multiple data reducing the execution time considerably. However, OpenACC has the onus of the memory copies from the host to the device and from the device to the host. These memory copies can make them unfeasible and significantly reduce performance gain.

In this work, we use directives for copying memory and creating kernels in nested loops and independent loops. In the parallel implementation using Ope-

nACC, we explore kernels or loop parallelism for GPU, like the code snippet presented in Algorithm 2. Similar parallelism approach was adopt in other functions called by the main iterative step.

Algorithm 2 Code region used OpenACC

```
do k=1,nk
  !$acc kernels present(u,dfb)
  do i=ii,fi
    do j=j1,j2
        Du=0.d0
        do s=-1,5
           Du=Du+dfb(s)*u(k,i,j+s)
        end do
        u(k,i,j) = u(k,i,j)-sigmaf*Du
    end do
  end do
!$acc end kernels
end do
```

5 Methodological Approach

We compare the numerical results of the parallel versions to the results obtained by the sequential implementation to guarantee numerical compatibility. That is, to have identical numerical results, which indicates that the code is free of the insertion of programming errors resulting from parallelization. We consider identical results when the difference between in the results are more than 10^{-12}.

5.1 Parameters of Input

We select 2 domains for the simulation of of mixing in the chamber. The Case A is a mesh of 461×381, simulating 100 discrete time steps; and the Case B is a mesh of 921×761, running 200 discrete time steps. An expressive number of discrete-time steps are necessary to simulate the same time interval when more mesh elements represent a domain.

Table 2 describes the two case studies. *Mesh I* represents a default mesh. *Mesh II* defines a second great mesh. This one has approximately $4\times$ the number of discrete elements, being more accurate (number of iterations and time advance) and, consequently, more expensive to process.

The results are the average of 30 repetitions for each test performed in this work. We define the number of threads as 2, 4, 8, 16, and 32 for OpenMP tests These values help to exploit the available hardware resources.

Table 2. *Mesh I* and *Mesh II* parameters

Description	Variable	*Mesh I* Value	*Mesh II* Value
`x` Dimension of the chamber	imax	461	921
`y` Dimension of the chamber	jmax	381	761
Time advance	dt	0.01	0.005
Number of iterations	maxit	100	200

5.2 Validation Environment

Table 3 shows the description of the execution environment. We compile the code with pgf90. We use the flag `-O3` for sequential execution. The tag `-fopenmp` is added for the code compilation with OpenMP in addition to the flag `-O3`. The tags `-O3`, `-fast`, `-acc`, and `Minfo=all` are added for the OpenACC version in the compilation process.

Table 3. Architecture Specification

specification	Xeon E5-2650 (×2)	Quadro M5000
Frequency	2.00 GHz	1.04 GHz
Cores	8 (×2)	2048
Threads	16 (×2)	
Cache L1	32 KB	64 KB
Cache L2	256 KB	2 MB
Cache L3	20 MB	
RAM/Global memory	128 GB	8 GB

6 Experimental Results

In this section, we show the results generated by the experiments. Figure 2 and Figure 3 present the execution time for *Mesh I* and *Mesh II*. The graphics present the sequential, OpenMP, and OpenACC measured time. The standard deviation was less than 1% in relation to the medium value.

The parallel OpenMP implementation provides execution time reduction. For all experiments, using more OpenMP threads continuously reduces the execution time, except when hyperthreading is active (32 OpenMP threads). The *Mesh II* presents more performance gain than the *Mesh I* experiment for the same number of threads. In the best case, the speedup was around 5 using 16 threads in the *Mesh II* experiment.

The OpenACC implementation does not provide acceleration for the *Mesh I*. It occurs due to the small mesh size in this test. In this case, the allocations, `copyin` and `copyout` of memory operations in each iteration demand a similar

Fig. 2. Execution Time - Mesh I

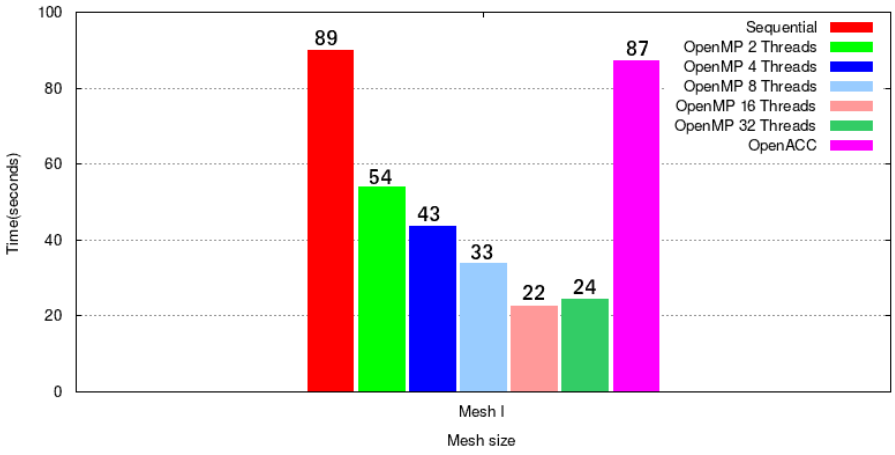
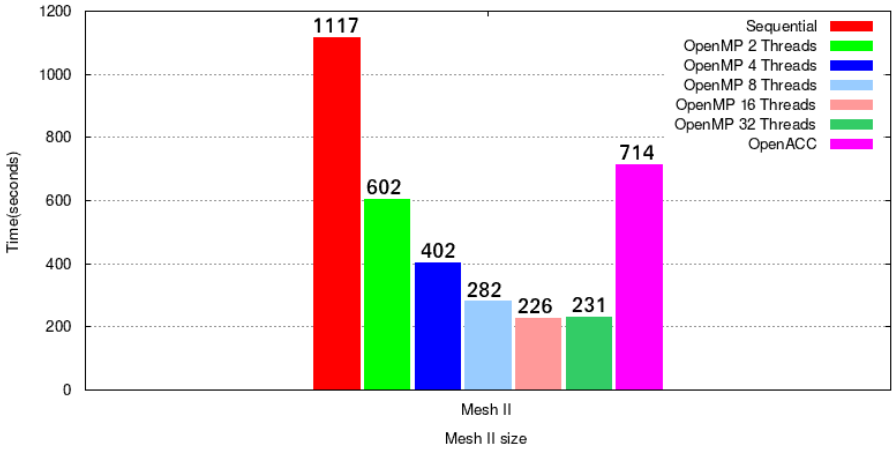


Fig. 3. Execution Time - Mesh II



time in relation to the gain of the parallel GPU execution. In the *Mesh II*, the OpenMP implementation provides some performance gain once the number of elements to compute is large, and more operations are processed in GPU.

7 Conclusion and Future Works

Applications that enable the simulation of phenomena and physical environments correspond to a range of efficient strategies for solving problems in several areas, including complex environments, such as the simulation of a mixing chamber.

The main objective of this work was to evaluate CPU and GPU parallel approaches in a mixing chamber simulation application. The performance gain was around 500% for the best case in tests performed on a multicore architecture, using 16 *threads*. The OpenACC implementations provide limited performance gain due to the size of the mesh evaluated. But, for the second case study, we obtain around 50% of performance gain. The performance on GPUs is below what is found in other related works. In this sense, it is necessary to modify the code and maintain the presence of data on the GPU throughout the iterative stage, in order to minimize the impacts of data transfer.

In future works, we intend to optimize the code using other parallel approaches provided by OpenMP and OpenACC APIs (SIMD, section, task, target, and others) in order to increase the speed up of the application. We also will explore CUDA API, using GPU to increase the computational power and obtain better performance gains.

8 Acknowledgements

This study was partially funded by the Fundação de Amparo à Pesquisa do Estado do Rio Grande do Sul (FAPERGS): 07/2021 PqG project Nº 21/2551-0002055-5, PROBIC and PROBITI programs, Coordenação de Aperfeiçoamento de Pessoal de Nivel Superior - Brasil (CAPES) - Finance Code 001, and the Federal University of Pampa.

References

1. Aldinucci, M., Cesare, V., Colonnelli, I., Martinelli, A., Mittone, G., Cantalupo, B., Cavazzoni, C., Drocco, M.: Practical Parallelization of Scientific Applications with OpenMP, OpenACC and MPI. Journal of Parallel and Distributed Computing 157 (06 2021)
2. Ayguade, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P., Zhang, G.: The Design of OpenMP Tasks. IEEE Transactions on Parallel and Distributed Systems 20(3), 404–418 (2009)
3. Chandrasekaran, S., Juckeland, G.: OpenACC for Programmers: Concepts and Strategies. Addison-Wesley Professional, 1st edn. (2017)

4. Christopher P. Stone, Roger L. Davis, D.Y.L.: Concurrent Parallel Processing on Graphics and Multicore Processors with OpenACC and OpenMP. Accelerator Programming Using Directives (2018), https://www.springerprofessional.de/concurrent-parallel-processing-on-graphics-and-multicore-process/15456018
5. Gonçalves, R., Amaris, M., Okada, T., Bruel, P., Goldman, A.: OpenMP is Not as Easy as It Appears. In: 2016 49th Hawaii International Conference on System Sciences (HICSS). pp. 5742–5751 (2016)
6. Guo, X., Wu, J., Wu, Z., Huang, B.: Parallel Computation of Aerial Target Reflection of Background Infrared Radiation: Performance Comparison of OpenMP, OpenACC, and CUDA Implementations. IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing 9(4), 1653–1662 (2016)
7. Hernandez, E., Palacios, G., Marín, C.: Parallel Programming Languages On Heterogeneous Architectures Using Openmpc, Ompss, Openacc, and Openmp. Tecnura 18 (08 2015)
8. Kasielke, F., Tschüter, R., Iwainsky, C., Velten, M., Ciorba, F.M., Banicescu, I.: Exploring Loop Scheduling Enhancements in OpenMP: An LLVM Case Study. In: 2019 18th International Symposium on Parallel and Distributed Computing (ISPDC). pp. 131–138 (2019)
9. Khalilov, M., Timoveev, A.: Performance analysis of CUDA, OpenACC and OpenMP programming models on TESLA V100 GPU. Journal of Physics: Conference Series 1740, 012056 (jan 2021), https://doi.org/10.1088/1742-6596/1740/1/012056
10. Nandamuri, A., Malik, A.M., Qawasmeh, A., Chapman, B.M.: Power and Energy Footprint of OpenMP Programs Using OpenMP Runtime API. In: 2014 Energy Efficient Supercomputing Workshop. pp. 79–88 (2014)
11. OpenACC: What is OpenACC? (2021), https://www.openacc.org/, [Online; acessed july, 20 2021]
12. Parikh, D.N., Huang, J., Myers, M.E., van de Geijn, R.A.: Learning from Optimizing Matrix-Matrix Multiplication. In: 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 332–339 (2018)
13. Pizzolato, G., Schepke, C., Lucca, N.: Aceleração de uma Aplicação de Simulação de Câmara de Combustão em Multi-Core. In: Anais do XXII Simpósio em Sistemas Computacionais de Alto Desempenho. pp. 36–47. SBC, SBC, Porto Alegre, RS, Brasil (2021), https://sol.sbc.org.br/index.php/wscad/article/view/18510
14. Vergara Larrea, V.G., Budiardja, R.D., Gayatri, R., Daley, C., Hernandez, O., Joubert, W.: Experiences in porting mini-applications to OpenACC and OpenMP on heterogeneous systems. Concurrency and Computation. Practice and Experience 32(20) (4 2020), https://www.osti.gov/biblio/1649533
15. Zhang, S., Yuan, R., Wu, Y., Yi, Y.J.: Parallel Computation of a Dam-Break Flow Model Using OpenACC Applications. Journal of Hydraulic Engineering 143, 04016070 (08 2016)