Solving ARGESIM Benchmark CP2 'Parallel and Distributed Simulation' with Open MPI and Matlab PCT – Lattice Boltzmann Simulation

David Jammer[1,2]*, Peter Junglas[2], Sven Pawletta[1]

[1]Research Group Computational Engineering and Automation, University of Applied Sciences Wismar,
Philipp-Müller-Straße 14, 23966 Wismar, Germany; *david.jammer@cea-wismar.de

[2]PHWT-Institut, PHWT Vechta/Diepholz, Am Campus 2, 49356 Diepholz, Germany;

SNE 33(2), 2023, 93-100, DOI: 10.11128/sne.33.bncp2.10646
Received: 2023-06-01; Improved: 2023-06-05
Accepted: 2023-06-06
SNE - Simulation Notes Europe, ARGESIM Publisher Vienna
ISSN Print 2305-9974, Online 2306-0271, www.sne-journal.org

Abstract. The ARGESIM benchmark CP2 consists of three different tasks to study current technologies for the parallelization of simulation programs, two of which have been addressed in a previous publication. The third one is the study of the fluid flow in a special geometry using the Lattice Boltzmann method. The task is studied with two methods for up to 256 cores, again using the MPI message passing library Open MPI and Matlab from The MathWorks in combination with the Parallel Computing Toolbox. Solutions with different grid sizes are compared with each other in terms of runtime and speedup. The Open MPI version generally shows good speedups even for large core numbers, while the Matlab version has poor results for 32 cores or more. On the other hand, the scalar Matlab version is several times faster than the Open MPI version, leading to a smaller runtime for up to 32 cores.

Introduction

In this article the solution of one task of the ARGESIM benchmark CP2 [1] will be presented. The benchmark contains three tasks, two of which have already been solved and presented in [2]. The remaining task, the Lattice Boltzmann simulation, is solved in this article. The solution is again implemented with the two technologies Open Message Passing Interface (Open MPI) [3] and Matlab Parallel Computing Toolbox (Matlab PCT)[4].

Since the description of the Lattice Boltzmann task in [1] is very sketchy, the basic method is explained in more detail in the first section, followed by a description of the scalar algorithm. Then the strategy for parallelization of this algorithm is presented. Finally, implementations in MPI and PCT are examined and compared.

The investigations were performed on an HPC system of the PHWT Institute called Seneca. The specifications of the system are shown in Table 1. All programs and scripts necessary to reproduce the results presented here are available from [5].

1 Introduction to Lattice Boltzmann Simulation

The Lattice Boltzmann Method (LBM) is a computational fluid dynamics method that has gained popularity, since it is especially well suited for parallel computers. In the following its basic ideas will be presented and the CP2 benchmark task described in full detail. This should make the task accessible to non-experts. A thorough introduction into the method and its connection to the familiar Navier-Stokes description can be found in [6].

LBM is based on the kinetic theory, which describes a fluid on a mesoscopic scale between the macroscopic description of a continuous fluid and the microscopic tracking of individual particles.

Nodes	3
Cores	288
Processors	AMD Epyc 7552
Main memory	1536 GiByte
High-speed network	100 GBit/s InfiniBand
Management network	1 GBit/s Ethernet
Operating system	OpenSuse Leap 15.3
Middleware	OpenHPC
Cluster management	Warewulf
Job Scheduler	SLURM
Software	GCC 9.3.0 GSL 2.6 open MPI 4.1.1 ucx 1.13.0 libfabric 1.13.0 hwloc 2.1.0 Matlab R2021a

Table 1: Seneca hardware and software overview.

Its fundamental variable is the distribution function $f(\vec{x},\vec{\xi},t)$, which represents the mass density of particles at position $\vec{x}$, velocity $\vec{\xi}$ and time t. It is connected to the macroscopic mass density $\rho(\vec{x},t)$ and the macroscopic velocity $u(\vec{x},t)$ by

$$\rho(\vec{x},t) = \int f(\vec{x},\vec{\xi},t)\,d^3\xi$$
$$\rho(\vec{x},t)\,\vec{u}(\vec{x},t) = \int \vec{\xi} f(\vec{x},\vec{\xi},t)\,d^3\xi.$$

Using standard assumptions one can show that the distribution function will reach an equilibrium distribution given by

$$f^{\mathrm{eq}}(\vec{x},\vec{v},t) = \rho\left(\frac{1}{2\pi R_i T}\right)^{\frac{3}{2}} e^{-v^2/(2R_iT)},$$

where the relative velocity $\vec{v} := \vec{\xi} - \vec{u}$ is the deviation of the particle velocity from the macroscopic velocity. f satisfies the Boltzmann equation

$$\frac{\partial f}{\partial t} + \xi_i \frac{\partial f}{\partial x_i} + \frac{F_i}{\rho}\frac{\partial f}{\partial \xi_i} = \Omega(f).$$

Here, $\vec{F}$ describes external forces, while Ω, the so called *collision operator*, is given by the forces between the particles during collisions. In LBM it is usually replaced by the *BGK collision operator* (named after its inventors Bhatnagar, Gross and Krook)

$$\Omega(f) = -\frac{1}{\tau}(f - f^{\mathrm{eq}}).$$

which simply describes the evolution of f towards f^{eq}. The corresponding relaxation time τ is related to the macroscopic viscosity.

For the computation of f with LBM one discretizes space, time and velocity, choosing one of several schemes according to the dimension of the problem. Applying standard lattice units, the CP2 task uses a fixed time step $\Delta t = 1$ for t, a two-dimensional square grid with $\Delta x = \Delta y = 1$ for $\vec{x}$ and a set $\vec{c}_i$, $i = 0,\ldots,8$ of velocities $\vec{\xi}$ (cf. Fig. 1), e. g. $\vec{c}_0 = (0,0)'$, $\vec{c}_1 = (1,0)'$, $\vec{c}_5 = (1,1)'$ etc. .

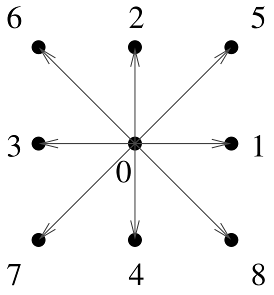

Figure 1: Velocities used in discretization scheme.

For simplicity one replaces the argument $\vec{\xi}$ with an index corresponding to one of the nine possible velocities, writing

$$f_i(\vec{x},t) := f(\vec{x},\vec{c}_i,t), \quad i = 0,\ldots,8$$

The macroscopic quantities are then given simply by

$$\rho(\vec{x},t) = \sum_i f_i(\vec{x},t)$$
$$\rho(\vec{x},t)\,\vec{u}(\vec{x},t) = \sum_i \vec{c}_i f_i(\vec{x},t).$$

The equilibrium distribution can now be simplified to

$$f_i^{\mathrm{eq}}(\vec{x},t) = w_i\rho\left(1 + \frac{\vec{u}\vec{c}_i}{c_s^2} + \frac{(\vec{u}\vec{c}_i)^2}{2c_s^4} - \frac{\vec{u}\vec{u}}{2c_s^2}\right),$$

with the speed of sound $c_s = 1/\sqrt{3}$ and the weights

$$w_i = \left(\frac{4}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{36}, \frac{1}{36}, \frac{1}{36}, \frac{1}{36}\right)'.$$

The corresponding Lattice Boltzmann Equation (LBE) in the force-free case $\vec{F} = \vec{0}$ is

$$f_i(\vec{x} + \vec{c}_i, t+1) = f_i(\vec{x}, t) + \Omega_i(\vec{x}, t),$$

again using the BGK collision operator

$$\Omega_i(f) = -\frac{1}{\tau}(f_i - f_i^{\mathrm{eq}}),$$

where the relaxation time is given by the viscosity ν:

$$\tau = \frac{\nu}{c_s^2} + \frac{1}{2}.$$

Finally one needs initial and boundary values and a few parameters. The CP2 task requests a quadratic lattice of size $n_x \times n_x$ with $n_x = 257$ and boundary values describing a cavity flow that has been studied in [7]: The upper grid line has a constant velocity $\vec{u}_0 = (0.1, 0)'$, while all other boundary points are fixed walls (cf. Fig. 2).

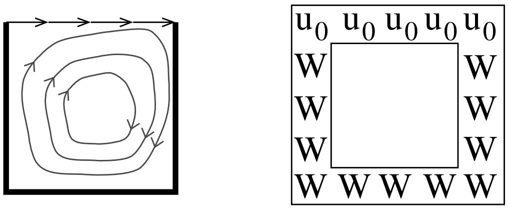

Figure 2: Definition of boundary values.

The initial values of the macroscopic properties are defined as

$$\begin{aligned}\rho(\vec{x}, 0) &= 1\\ \vec{u}(\vec{x}, 0) &= \vec{0}.\end{aligned}$$

The end time of the simulation is given as $t_{end} = 350,000$, when a steady state should have been reached. To adapt to the much larger processing power and memory size since the publication of [1], larger grids up to $n_x = 2048$ and end time $t_{end} = 500,000$ will be studied in the following.

At last, a Reynolds number of $\mathrm{Re} = 1000$ is given, which leads to a viscosity of

$$\nu = \frac{(n_x - 1)u_{0,x}}{\mathrm{Re}} = 0.0256.$$

2 Basic Scalar Implementation

The scalar implementation follows the lines of the basic algorithm described in [6]. The essential idea is to split the LBE in two parts, the collision step

$$f_i^*(\vec{x}, t) = f_i(\vec{x}, t) - \frac{1}{\tau}(f_i(\vec{x}, t) - f_i^{\mathrm{eq}}(\vec{x}, t)),$$

which is a local operation at a point, and the propagation step

$$f_i(\vec{x} + \vec{c}_i, t+1) = f_i^*(\vec{x}, t),$$

which transports the distribution to neighbouring points. Initially the distribution function is set to the equilibrium values given by $\rho(\vec{x}, 0)$ and $\vec{u}(\vec{x}, 0)$. Then the following computations are performed in a loop over time steps:

1. computation of ρ and $\vec{u}$ from f_i,
2. computation of f_i^{eq} from ρ and $\vec{u}$,
3. collision step to get f_i^*,
4. propagation step to get $f_i(\vec{x}, t+1)$.

During the collision step the boundary behaviour is implemented: At the top line f is set to the equilibrium value defined by the given $\vec{u}_0$; at the wall points the full-way bounce-back method is used, which reverses the velocity of an incoming distribution, i.e. it sets

$$f^*(\vec{x}, \vec{c}_i, t) = f(\vec{x}, -\vec{c}_i, t).$$

After the time loop the relative magnitude $|\vec{u}|/|\vec{u}_0|$ of the macroscopic velocity is computed and plotted, as is requested in the benchmark. Fig. 3 shows the result for the standard benchmark parameters.

3 Parallel Lattice Boltzmann Simulation

In this chapter, a parallelization strategy for the Lattice Bolzmann simulation is introduced and investigated on the HPC system Seneca using two technologies.

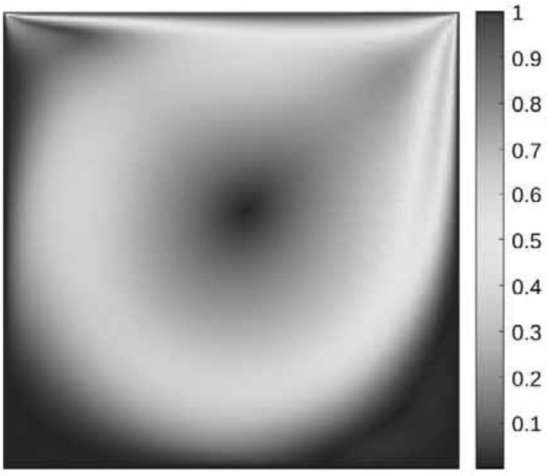

Figure 3: Relative macroscopic velocity magnitude for $n_x = 257$ after 350,000 iterations.

3.1 Parallelization Strategy

To process the Lattice Boltzmann simulation in parallel, the set of cores is virtually arranged in a two-dimensional cartesian coordinate system as shown in Figure 4. The data grid f is distributed equally along this grid, so that each core contains a section of the entire grid according to its coordinates. Except for the propagation step, all calculations on such a section can be done indepedently by each core.

(0,0) 0	(0,1) 1	(0,2) 2	(0,3) 3
(1,0) 4	(1,1) 5	(1,2) 6	(1,3) 7
(2,0) 8	(2,1) 9	(2,2) 10	(2,3) 11
(3,0) 12	(3,1) 13	(3,2) 14	(3,3) 15

Figure 4: Organization of cores on a two-dimensional cartesian coordinate system.

To simplify the decomposition, the data grid size n_x is restricted to a power of two in the implementation, e.g. 32x32, 64x64, etc. The number of cores n_C is subject to the same condition so that the entire grid can be divided equally.

Therefore, n_C was varied from $1, 2, 4, \ldots, 256$ in the following measurements. Of course, an uneven distribution could be implemented easily. But a fine-grained study of the effects of the total core number would reveal more about the memory architecture and the interconnect structure of the hardware than about the properties of the high-level tools we are interested in.

In the propagation step, displacements are performed in the north, east, south and west directions, as well as the four diagonal directions. In these shifts, data must be communicated with neighboring cores. For horizontal or vertical shifts, there is at most one neighbor to send and one to receive and complete edges of the local matrix are sent or received. Data points at the global edges, which do not receive values through a shift, retain their current values. The diagonal shifts are a bit more complicated because there are more communication partners here (cf. Fig. 5). At first sight, these operation can be simplified by combining two of the simple shift operations. For example, a shift to the north-east can be realized by shifting first to the east, then to the north. But actually, this leads to wrong values at the outer edge points, as can be seen in Fig. 6. Therefore, the following implementations are realized using explicit diagonal shifts.

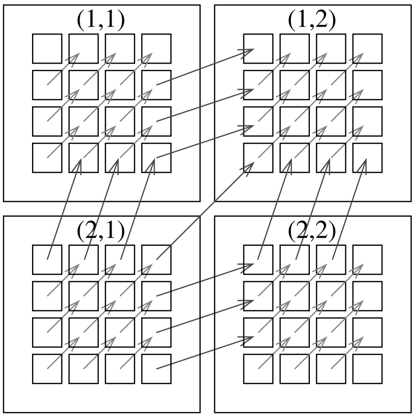

Figure 5: Matrix shift to the northeast with communication neighbors.

A different approach is used by the ScaLAPACK library [8]: There the entire grid is divided into smaller matrices of fixed blocksize, which are then distributed cyclically across the core grid. Since this seems to lead to a much more involved communication pattern in our application, it has not been followed here.

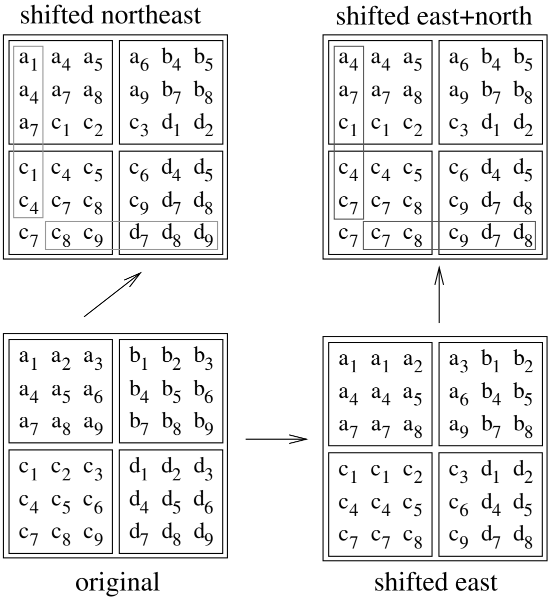
shifted northeast
$a_1\ a_4\ a_5$ $a_6\ b_4\ b_5$
$a_4\ a_7\ a_8$ $a_9\ b_7\ b_8$
$a_7\ c_1\ c_2$ $c_3\ d_1\ d_2$
$c_1\ c_4\ c_5$ $c_6\ d_4\ d_5$
$c_4\ c_7\ c_8$ $c_9\ d_7\ d_8$
$c_7\ c_8\ c_9$ $d_7\ d_8\ d_9$
shifted east+north
$a_4\ a_4\ a_5$ $a_6\ b_4\ b_5$
$a_7\ a_7\ a_8$ $a_9\ b_7\ b_8$
$c_1\ c_1\ c_2$ $c_3\ d_1\ d_2$
$c_4\ c_4\ c_5$ $c_6\ d_4\ d_5$
$c_7\ c_7\ c_8$ $c_9\ d_7\ d_8$
$c_7\ c_7\ c_8$ $c_9\ d_7\ d_8$
$a_1\ a_2\ a_3$ $b_1\ b_2\ b_3$
$a_4\ a_5\ a_6$ $b_4\ b_5\ b_6$
$a_7\ a_8\ a_9$ $b_7\ b_8\ b_9$
$c_1\ c_2\ c_3$ $d_1\ d_2\ d_3$
$c_4\ c_5\ c_6$ $d_4\ d_5\ d_6$
$c_7\ c_8\ c_9$ $d_7\ d_8\ d_9$
original
$a_1\ a_1\ a_2$ $a_3\ b_1\ b_2$
$a_4\ a_4\ a_5$ $a_6\ b_4\ b_5$
$a_7\ a_7\ a_8$ $a_9\ b_7\ b_8$
$c_1\ c_1\ c_2$ $c_3\ d_1\ d_2$
$c_4\ c_4\ c_5$ $c_6\ d_4\ d_5$
$c_7\ c_7\ c_8$ $c_9\ d_7\ d_8$
shifted east

Figure 6: Comparison of direct and composite diagonal shifts.

3.2 Open MPI

For the first parallel implementation, the C programming language and the Open Message Passing Interface (MPI) [3] were used. In order to realize the organization of the cores as a cartesian grid, Open MPI provides two functions: `MPI_Dims_create` factors the total number of cores into a balanced product. For the core numbers used here, this results either in a quadratic core grid or the size in one direction is twice the size of the other one. The function `MPI_Cart_create` then uses these sizes to create a communicator that maps the existing cores to a two-dimensional coordinate system. Due to the restrictions in grid size and total core number, the local data grid sizes of a core can then simply be calculated by dividing the total length n_x by the sizes of the core grid.

The calculation of the propagation step was divided into eight functions, one function for each direction. To identify the communication partners in north, east, south and west direction the function `MPI_Cart_shift` was used. This function returns the communication partners for sending and receiving the boundary values of the local data grids, but is restricted to shifts along the cartesian axes. It simply returns `MPI_PROC_NULL`, when a shift would leave the core grid. This value can be used safely in a send or receive function resulting in a null operation.

For the diagonal shifts, the coordinates of a core in the core grid and the corresponding target and source coordinates were determined with `MPI_Cart_coords`. They were then used to determine the corresponding ranks for the send and receive operations with `MPI_Cart_rank`. It is important to use this function only with valid coordinates, otherwise the program terminates with an error. To prevent this, one can again use the `MPI_Cart_get` function.

To collect the data at the end of the simulation the function `MPI_Cart_coords` was used to define the core with coordinates $(0,0)$ as the master and for sorting of the data. This allowed the grid segments to be associated with the correct points in the global grid.

3.3 Matlab PCT

The second parallel implementation was created using Matlab and the Parallel Computing Toolbox (PCT) [4]. A very short introduction to Matlab/PCT and the different parallelization strategies it supplies can be found in [2]. The underlying scalar version was taken from [9], which already compares different earlier parallel toolboxes from Mathworks and other authors.

For the task studied here, the PCT concept of codistributed arrays seems to be ideal: The data array is distributed to the workers automatically, and the local workers can still use the global index to address all array points. If an element is not available locally, it is sent immediately to the accessing worker. Unfortunately, the corresponding functions are badly documented and – more importantly – show an extremely bad performance [2]. Therefore, the `spmd` construct was used in the implementation, which creates a parallel section that is executed by each worker. This leads to a programming style that is smiliar to the MPI implementation. Since PCT does not provide functions for creating a two-dimensional core grid and to map cores to a grid, this functionality was implemented in the application using the functions `numlabs` and `labindex`. For conveniance, an array was supplied that contains the positions of all labindex values in the core grid.

For the propagation step, separate functions were created for each of the eight directions. To determine the communication partners a function `cart2D_shift` was written, which determines the rank of the target and the source from a direction vector and a displacement. This makes it easy to determine the communication partners. The sending and receiving of the data was realized with `labSend` and `labReceive`.

After the `spmd` section the local arrays are directly available in one cell array. Therefore the merging of the results could be realized simply by rearranging all velocity data in a global array.

3.4 Comparison of solutions

The two implementations were tested on Seneca and the runtime was analyzed depending on the grid size and the number of cores. The grid size was selected from 256x256 to 2048x2048, the number of cores from 1 to 256. Since Seneca consists of three nodes with 96 cores each, runs with up to 64 cores were computed on one node using local memory only, while for 128 and 256 cores two and three nodes were utilised, and data had to be sent across the high-speed network. For small numbers of cores the number of iterations has been reduced to make the computing times feasable. The timing results have been scaled afterwards to the total number of iterations for easier comparison. The scalar overhead of the computations was negligible in these cases.

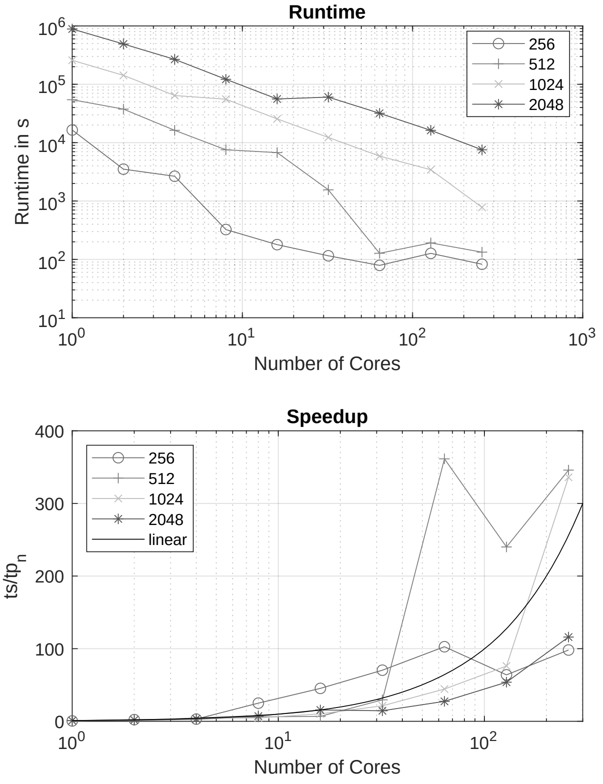

Figure 7: Runtime and speedup of the Open MPI implementation for different grid sizes.

Figure 7 shows the runtime and speedup results of the Open MPI implementation. The most basic model would assume runtimes that fall with $1/n_C$ and rise with n_x^2, which in a log-log representation would lead to parallel decreasing straight lines with constant separation. The runtime plots very roughly show this behaviour, especially for larger grid size. The deviations can be better analyzed in the speedup plot, in particular when comparing the results with the ideal case of linear speedup. Here the occuring superlinear speedup can be seen clearly, which is usually due to the memory architecture of the system: With increasing core number, the local memory decreases, until it fits in a fast cache memory. This interpretation is supported by the observation that the point, where superlinear speedup begins, rises with grid size and core number in such a way that the local memory size stays constant. Another expected behaviour is the drop of speedup for large core numbers, when more than one node is used. This effect gets smaller with rising grid size, because the amount of computation dominates the rising communication times. The results suggest that for large grid sizes an even higher speedup can be reached with more cores. In Figure 8 the results of the Matlab implementation are presented.

Looking at the speedups, Matlab PCT reaches good to reasonable values for up to 16 cores, but very poor results for more cores. The drop in performance at 128 cores can be seen again, except for a very peculiar rise at the grid size $n_x = 2048$. But when comparing the runtimes, the image changes completely: The table 2, which displays the runtimes for the scalar case $n_C = 1$, shows the impressive performance of Matlab compared to the direct C implementation. This is probably due to the efficient matrix operations Matlab is based on.

grid size	runtime [min] Open MPI	runtime [min] Matlab PCT	ratio
256	274.4	36.1	7.59
512	906.3	175.6	5.16
1024	4297.3	1173.1	3.66
2048	14757.2	4347.6	3.39

Table 2: Comparison of runtimes for $n_C = 1$.

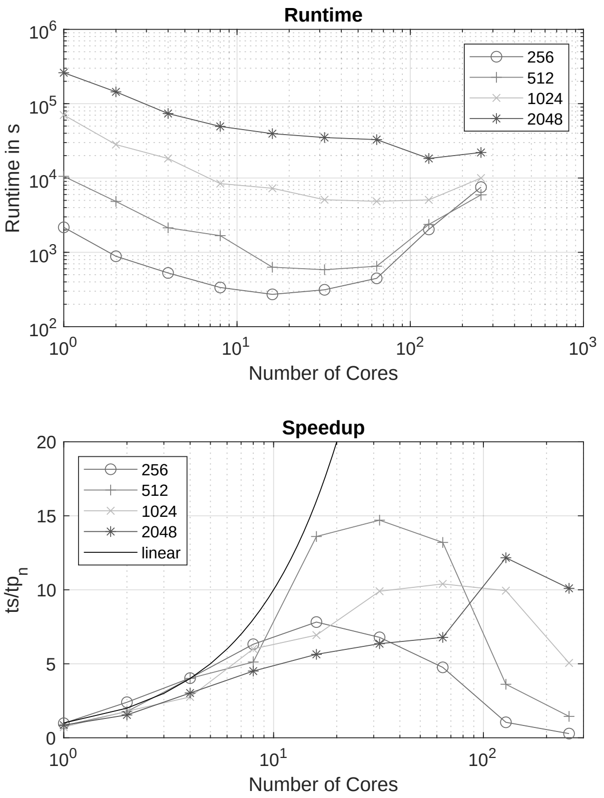

Figure 8: Runtime and speedup of the Matlab PCT implementation for different grid sizes.

Figure 9 shows the runtime ratios of the parallel Open MPI and Matlab PCT versions, compared for the same number of cores. Apart from the case of small grid size, Matlab is faster for up to 32 cores, and for the larger grids even up to 64 cores. It takes a lot of cores, until the better speedup of Open MPI catches up with the better scalar performance of Matlab PCT.

4 Conclusion

The goal of the CP2 benchmark cannot be to compare latest parallelization strategies for current hardware architectures – this would require much more complex algorithms such as those studied in the HPCG benchmark [10]. Instead, the underlying question of the benchmark is: How can non-experts use tools and computing environments to take advantage of modern multi-core CPUs? Correspondingly, the MPI implementations of the three CP2 tasks that have been presented here and in [2] mainly serve as points of reference and use well known techniques, whereas the parallelization methods and outcomes provided by a scientific computing environment such as Matlab are the main point of interest.

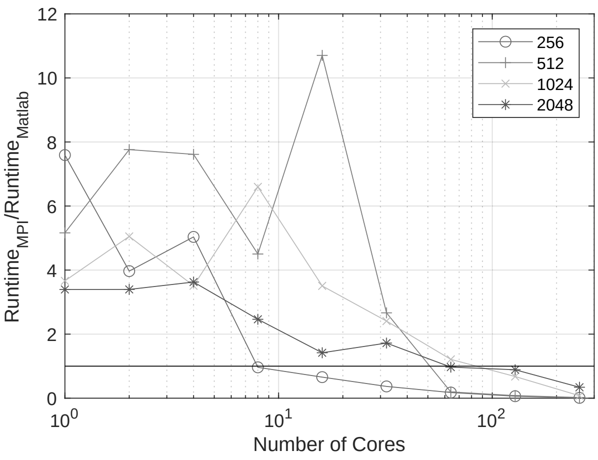

Figure 9: Ratio of Runtimes for MPI and Matlab implementations.

In this respect Matlab PCT is a very interesting approach, but the results presented here show that it doesn't quite live up to its prospects. On the one hand the `spmd` construct and standard message-passing methods make it possible to reach reasonable speedups for small core numbers, but require knowledge of standard parallelization techniques. Furthermore, the `parfor` command provides a convenient tool for very simple cases like Monte Carlo studies. On the other hand, the promising concept of codistributed arrays provides a very elegant way to cope with the distribution and parallel processing of large matrices, but it has shown very poor performance even in the simple PDE task.

For very compute-intense applications, which benefit most from larger core numbers, one typically would resort to standard languages like C or C++, together with optimizing compilers. But as we have seen in the PDE and LBM tasks, Matlab can be compatible or even better in the scalar case, due to its highly optimized mathematical libraries. Therefore it would be very desirable, if Matlab PCT could sustain good speedups to larger numbers of cores.

In spite of its interesting topic, the ARGESIM CP2 benchmark has not been addressed before. To some extent, this might be due to the rather short description of the Lattice Boltzmann task, which needs background reading to understand the algorithm at all. Hopefully, the short introduction provided here can help as a starting point for further studies.

Another problem of CP2 is the small problem size. To make the tasks challenging on modern architectures, one has to scale up the problem sizes considerably. Maybe, one could modernize this benchmark by using scaling methods instead of fixed sizes, and reissue it as a new entry in the standard ARGESIM benchmark suite, giving up on the idea of a special “parallel benchmarks” series.

References

[1] Breitenecker F, Höfinger G, Pawletta T, Pawletta S, Hinz R. ARGESIM Benchmark on Parallel and Distributed Simulation. *SNE Simulation News Europe*. 2007;17(1):53–56. ISSN 0929-2268.

[2] Jammer D, Junglas P, Pawletta S. Solving ARGESIM Benchmark CP2 ’Parallel and Distributed Simulation’ with Open MPI/GSL and Matlab PCT - Monte Carlo and PDE Case Studies. *SNE Simulation Notes Europe*. 2022;32(4):211–220. DOI: 10.11128/sne.32.bncp2.10625.

[3] Graham RL, Woodall TS, Squyres JM. Open MPI: A flexible high performance MPI. *Lecture notes in computer science*. 2006;3911:228.

[4] The MathWorks. *MATLAB Parallel Computing Toolbox*. URL `https://de.mathworks.com/help/pdf_doc/parallel-computing/parallel-computing.pdf`

[5] CEA Wismar. *ARGESIM benchmark CP2 on GitHub*. `https://github.com/cea-wismar/ArgesimCP2`.

[6] Krüger T, Kusumaatmaja H, Kuzmin A, Shardt O, Silva G, Viggen EM. *The Lattice Boltzmann Method: Principles and Practice*. Cham, Switzerland: Springer. 2017.

[7] Hou S, Zou Q, Chen S, Doolen GD, Cogley AC. Simulation of cavity flow by the lattice Boltzmann method. *Journal of computational physics*. 1995; 118(2):329–347.

[8] Blackford LS, Choi J, Cleary A, D’Azevedo E, Demmel J, Dhillon I, Dongarra J, Hammarling S, Henry G, Petitet A, Stanley K, Walker D, Whaley RC. *ScaLAPACK Users’ Guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics. 1997.

[9] Fink R. Untersuchungen zur Parallelverarbeitung mit wissenschaftlich-technischen Berechnungsumgebungen. Ph.D. thesis, Universität Rostock, Rostock. 2007.

[10] Dongarra J, Heroux MA, Luszczek P. A new metric for ranking high-performance computing systems. *The International Journal of High Performance Computing Applications*. 2016;30(1):3–10.