KONWIHR Report: Optimizing the Parallel Granular Gas Solver to study the crater formation

Ali Shakeri[1], Katrin Nusser[2], Dominik Ernst[2], Georg Hager[2], Gerhard Wellein[2], and Thorsten Pöschel[1]

[1]*Institute for Multiscale Simulations, Friedrich-Alexander Universität, Erlangen, Germany*
[2]*Regionales Rechenzentrum Erlangen, Universität Erlangen, Erlangen, Germany*

June 21, 2019

Abstract

In this report, we describe the optimization procedure of a MPI parallelized code that developed based on C++ to solve the granular flow hydrodynamics. The optimizations includes single-core and multi-core MPI optimizations. In this regard, we used mainely the LIKWID and the ITAC profiling tools to track each step of the optimizations. The improvements include the inlining of the functions, memory management and eliminating expensive operations which we will discuss them with more details. At the end we provide efficiency measurements and suggest further optimizations for future studies.

1 Introduction

The hydrodynamic equations of the granular flow, describes the system by average field variables like particle number density $n(\vec{r}, t)$, flow velocity $\vec{u}(\vec{r}, t)$ and temperature $T(\vec{r}, t)$. The time evolution of these quantities are governed by hydrodynamic equations [1]

$$\begin{aligned} \frac{\partial n}{\partial t} + \nabla \cdot (n\vec{u}) &= 0, \\ n\left(\frac{\partial \vec{u}}{\partial t} + (\vec{u}\cdot\nabla)\vec{u}\right) &= -\frac{1}{m}\nabla\cdot\mathbb{P} + n\vec{g}, \\ n\left(\frac{\partial T}{\partial t} + (\vec{u}\cdot\nabla)T\right) &= -\nabla\cdot\vec{q} - \mathbb{P} : \nabla\vec{u} - \zeta n T. \end{aligned} \tag{1}$$

Along with equations (1) the constitutive equations should be also provided. In above equations $\mathbb{P}_{ij}$ is the stress tensor, $\vec{q}$ is the heat flux, $\vec{F}$ is external body force, m is the mass of a single grain and ζ describes the rate of energy loss due to dissipative nature of granular gas. To Navier-Stokes order, the stress tensor and heat flux are given by $\mathbb{P}_{ij} = p\delta_{ij} - \eta(\partial_i u_j + \partial_j u_i - \frac{2}{3}\delta_{ij}\nabla\cdot\vec{u}) - \gamma\delta_{ij}\nabla\cdot\vec{u}$ and $\vec{q} = -\kappa\nabla T - \mu\nabla n$, where p is hydrostatic pressure, η is shear viscosity, γ is bulk viscosity, κ is thermal conductivity and μ is a coefficient that relates heat transfer to density gradient.

The hydrodynamic equations (1) can be writen in terms of conservtive variables: density, n, momentum density, $n\vec{u}$ and energy density, $nE = n(\frac{1}{2}m\vec{u}^2 + \frac{3}{2}T)$. These new variables fulfill conservation of mass, momentum and energy respectively:

$$\begin{aligned}
&\frac{\partial n}{\partial t} + \nabla \cdot (n\vec{u}) = 0, \\
&\frac{\partial (n\vec{u})}{\partial t} + \nabla \cdot \left[\vec{u} n \vec{u}^T + \frac{p}{m}\mathbb{I}\right] = \frac{1}{m}\nabla \cdot \tau + n\vec{g}, \\
&\frac{\partial (nE)}{\partial t} + \nabla \cdot [(nE + p)\vec{u}] = \nabla \cdot [\tau \cdot \vec{u} - \vec{q}] + mn\vec{v} \cdot \vec{g} - \frac{3}{2}n\zeta T.
\end{aligned} \tag{2}$$

where $\mathbb{I}$ is the identity matrix. It can be shown that formulation (1) and (2) lead to different solutions in the presence of shock waves [8]. In addition, Hou and Le Floch [3] shown that non-conservative schemes do not converge to the correct solution in the presence of shock waves. Therefore, it is inevitable to work with conservative formulation (2), since shocks appear frequently in granular systems.

For the sake of brevity, the conservative formulation can be written in a compact form

$$\frac{\partial U}{\partial t} + \nabla \cdot \vec{F}^c(U) = \nabla \cdot \vec{F}^\nu(U, \nabla U) + S(U), \tag{3}$$

by introducing the vector of conservative variables $U = (U_1, U_2, U_3, U_4, U_5)^T$, the convective fluxes $\vec{F}^c = (F^c_x, F^c_y, F^c_z)^T$, and the diffusive fluxes $\vec{F}^\nu = (F^\nu_x, F^\nu_y, F^\nu_z)^T$. Eq. (3) is a subset of convection-diffusion equations. While diffusive processes affects the field variables along its gradient in all direction, convection propagate the fields only in flow direction [9]. Therefore, it is difficult to design a single robust scheme that can handle all possible balances of the convection and diffusion effects [2]. Splitting methods allow us to overcome this difficulty, since they make it possible to utilize different efficient schemes that are developed specifically for each convection and diffusion problems.

Let $\mathcal{S}^c_{\Delta t}$ denote the exact solution operator of the corresponding convective problem, $\partial_t U + \nabla \cdot F_c(U) = 0$, which propagates the solution $U(t)$ by one timestep, $U(t+\Delta t) = \mathcal{S}^c_{\Delta t} U(t)$. Likewise, let $\mathcal{S}^\nu_{\Delta t}$ denote the exact solution operator, that propagates the solution $U(t)$ of the following diffusive problem with sources, $\partial_t U + \nabla \cdot F_\nu(U, \nabla U) = S(U)$, to a later time $t + \Delta t$. Then, the Strang splitting scheme [6] gives us the combined solution of the convection-diffusion problem (Eq. (3)):

$$U(t + \Delta t) = \left[\mathcal{S}^c_{\Delta t/2} \mathcal{S}^\nu_{\Delta t} \mathcal{S}^c_{\Delta t/2}\right] U(t). \tag{4}$$

The splitting scheme (4) is used in our current code.

2 Single-core optimizations

The problem that we chose as the test case is usually called *Bouncing bed* in literature. To simulate this problem, we discretized our simulation domain into a cartesian mesh of $5 \times 40 \times 5$ size. We are interested in 100 timesteps of this simulation. The choice of this small domain is to save computation time for the single-core optimizations. We will study larger systems when we move to multi-core optimizations.

We need a measure to check the validity of the code after each optimization. For this reason, we compute the **L1-norm** which returns the difference in solution compared to the reference result. If

X_i denotes the reference solution over the whole mesh and x_i denotes the same solution but for the modified code, then for a cartesian mesh the **L1-norm** is defined as

$$\text{L1-norm} = \frac{1}{N}\sum_{i=1}^{N}|x_i - X_i| \tag{5}$$

where N is the mesh size. We expect this quantity to be zero since the optimizations should not change the results.

As the first candidate for improvement, we tried to inline the functions that are called most frequently. We found that the Intel compiler flag `-ipo` does the inlining perfectly. After this change the functions like `VecOperations::operator*(double, Vec5 const&)` disappear from the `gprof` profile and the performance improves by about 34 percent.

We used the Library of Iterative Solvers for Linear Systems (LIS) to solve the linear equations resulting from the diffusion equation. In the profile of the code, there was a serious bottleneck in `lis_realloc` function. By looking at the code we found out that an unnecessary allocation and deallocatin is happening every time step. To fix this issue, all allocations are moved to `DiffusiveFlux()` constructor, which will be called after initializing the `Simulation` class. In addition, all deallocations are moved to the `~DiffusiveFlux()` destructor which will be called at the end of simulation. After this change the code runs 7 times faster.

The next candidates for optimizations were the WENO function `Weno::compute_omega()` which computes the interpolation weights, the equation of state for the pressure `EquationsOfState::pressure()` and the MUSTA flux terms. Since, the divisions are more expensive than multiplications, the idea is to avoid unnecessary divisions in this expensive functions. These changes improved the performance by about 35 percent.

With all these improvements the code operates about 14.6 times faster than the initial version. The `gprof` profile of the optimized code is as follows:

```
Each sample counts as 0.01 seconds.
  %   cumulative         self
time   seconds    calls    ms/call  name
35.27     3.31    25452000 0.00  Weno::compute_omega(double, double, double, Stencil)
 9.02     4.15    96601248 0.00  EquationsOfState::pressure(double, double)
 8.43     4.94                  lis_matvech_csr
 8.16     5.71    606      1.26  Weno::find_gauss_legendre_points(Cell*, int)
 7.15     6.38                  lis_matvec_csr
 5.76     6.92    3577824  0.00  Musta::musta_Gc(Vec5, Vec5, double)
 5.28     7.41    3577824  0.00  Musta::musta_Hc(Vec5, Vec5, double)
 3.79     7.77    3577824  0.00  Musta::musta_Fc(Vec5, Vec5, double)
 3.31     8.08    606      0.51  Weno::find_intermediate_points(Cell*, int)
 2.45     8.31    606      0.38  Weno::find_center_of_surfaces(Cell*, int)
```

Listing 1: The top expensive functions in the profile of the optimized code. The Weno function that computes the interpolation weights is responsible for about 35 percent of the total run time.

In the procedure of optimizations, we used LIKWID software and the `likwid-perfctr` tool to get information about the memory bandwidth and performance of the current optimized version. The first option `-g DIVIDE` gives us information about division operations:

```
+-----------------------+---------+--------------+
|         Event         | Counter |    Core 0    |
+-----------------------+---------+--------------+
|   INSTR_RETIRED_ANY   |  FIXC0  | 169790500660 |
| CPU_CLK_UNHALTED_CORE |  FIXC1  | 103210050216 |
|  CPU_CLK_UNHALTED_REF |  FIXC2  |  76205325416 |
|      ARITH_NUM_DIV    |   PMC0  |    702778841 |
|  ARITH_FPU_DIV_ACTIVE |   PMC1  |  38406839257 |
+-----------------------+---------+--------------+

+----------------------------------+-----------+
|              Metric              |   Core 0  |
+----------------------------------+-----------+
|       Runtime (RDTSC) [s]        |   36.0019 |
|       Runtime unhalted [s]       |   46.9131 |
|           Clock [MHz]            | 2979.6445 |
|               CPI                |    0.6079 |
|       Number of divide ops       | 702778841 |
| Avg. divide unit usage duration  |   54.6500 |
+----------------------------------+-----------+
```

Listing 2: The likwid output of the optimized code. It provides information about the number of divisions.

The second option `-g MEM` provides information about memory bandwidth. For clarity, we just report part of the information here

```
+------------------------------------+-----------+
|               Metric               |   Core 0  |
+------------------------------------+-----------+
|        Runtime (RDTSC) [s]         |   34.4098 |
|        Runtime unhalted [s]        |   46.4950 |
|            Clock [MHz]             | 2991.7720 |
|                CPI                 |    0.6078 |
|  Memory read bandwidth [MBytes/s]  |   36.0318 |
|  Memory read data volume [GBytes]  |    1.2398 |
| Memory write bandwidth [MBytes/s]  |   34.9019 |
| Memory write data volume [GBytes]  |    1.2010 |
|    Memory bandwidth [MBytes/s]     |   70.9337 |
|    Memory data volume [GBytes]     |    2.4408 |
+------------------------------------+-----------+
```

Listing 3: The likwid output with informations about memory bandwidth. This output shows that the code is not memory bounded.

Finally, the option `-g FLOPS_DP` reveals the actual performance of the code:

```
+-----------------------+-----------+
|        Metric         |   Core 0  |
+-----------------------+-----------+
|  Runtime (RDTSC) [s]  |   34.0013 |
| Runtime unhalted [s]  |   46.0993 |
|      Clock [MHz]      | 2998.5086 |
|          CPI          |    0.6075 |
|      DP MFLOP/s       | 1541.9145 |
|    AVX DP MFLOP/s     |         0 |
|    Packed MUOPS/s     |   34.3259 |
|    Scalar MUOPS/s     | 1473.2627 |
|  Vectorization ratio  |    2.2769 |
+-----------------------+-----------+
```

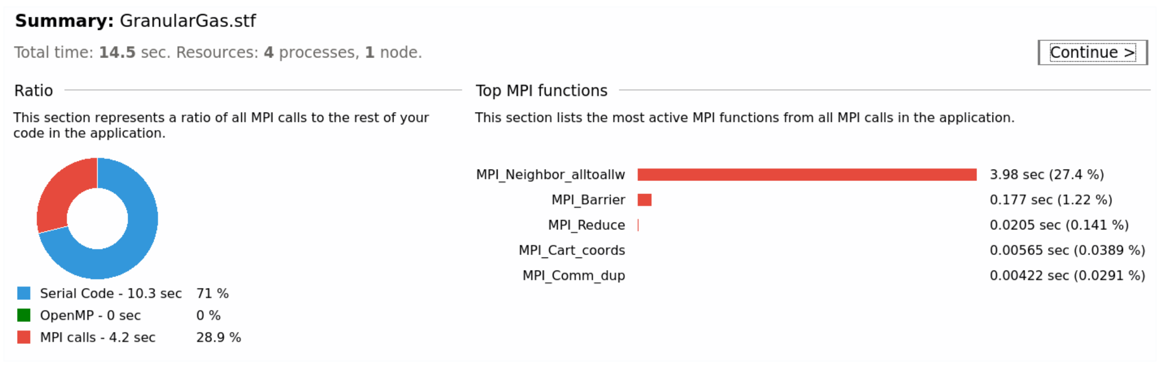
Listing 4: The likwid software gives information about floating point operations per second and also vectorization ratio.

3 Multi-core optimizations

To benchmark the MPI code, we studied the explosion test [8]. Here we used a cartesian grid of size $48 \times 48 \times 48$, with periodic boundary condition in all directions. The simulation is done after 100 time steps.

In this study we used Intel Trace Analyzer and Collector (ITAC) software, which gives us valuable information about the MPI code. There is no need to modify the code or rebuild the application. We just need to add `-trace` flag to the `mpirun` command. It is also necessary to run `psxevars.sh` script which sets up the environment variables for compilers, Intel MPI Library, and ITAC.

After running the jobfile we see the `*.stf` files appear in the specified directory. The next step is to open the main `.stf` file with the `traceanalyzer` tool. Then the Trace Analyzer tool appears. The summary view itself gives us many useful informations about the MPI code (see Fig. 1) By clicking on continue we can see more details such as load balance (see Fig. 2)



Summary: GranularGas.stf

Total time: **14.5** sec. Resources: **4** processes, **1** node.

Ratio

This section represents a ratio of all MPI calls to the rest of your code in the application.

🟦	Serial Code - 10.3 sec	71 %
🟩	OpenMP - 0 sec	0 %
🟥	MPI calls - 4.2 sec	28.9 %

Top MPI functions

This section lists the most active MPI functions from all MPI calls in the application.

MPI_Neighbor_alltoallw	3.98 sec (27.4 %)
MPI_Barrier	0.177 sec (1.22 %)
MPI_Reduce	0.0205 sec (0.141 %)
MPI_Cart_coords	0.00565 sec (0.0389 %)
MPI_Comm_dup	0.00422 sec (0.0291 %)

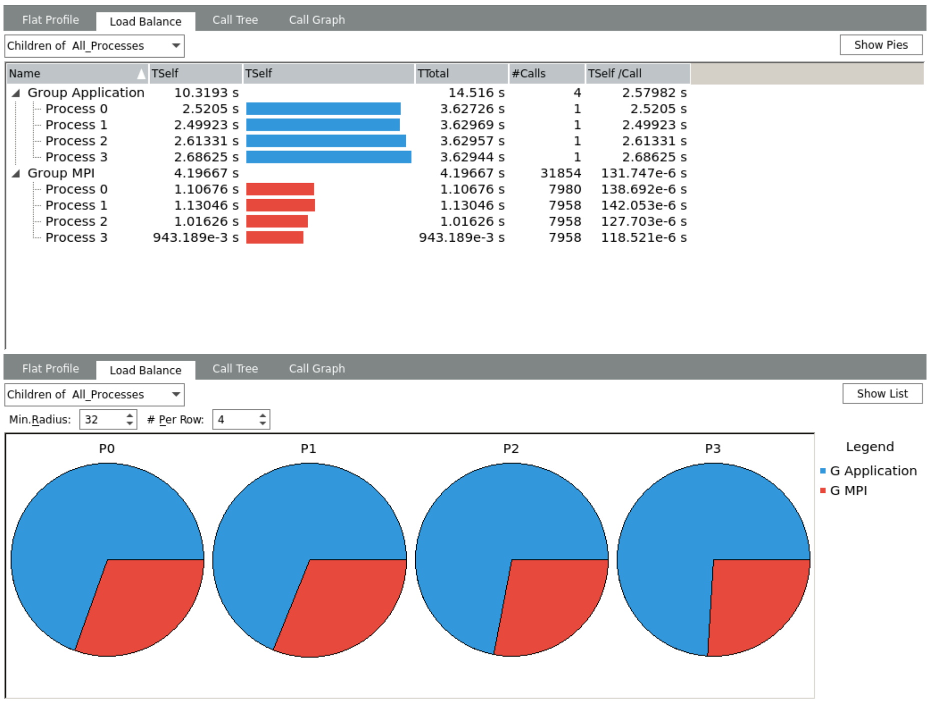
Figure 1: The summary view shows the total run time and the percentage of serial code and MPI calls. The bar chart on right tells us that the `MPI_Neighbor_alltoallw` is the main MPI call that needs improvement.

As we can see in Fig. 1 and Fig. 2 the MPI calls take about 29 percent of total time. If we use more cores the percentage of MPI calls increases due to higher amount of communications.

One of the problems in the initial version of the code is that there is only one function for communication of the grid cell data which sends all the data during any communication. However, in different stages of the code only certain part of cell data is necessary to be sent. In other words, we are sending unnecessary data which can be improved.

A solution to this is to write specialized communication functions that only send necessary information. We need to implement the following communication functions:

- To send only the conservative variables we use the function `communicate_U (Cell*, MPI_Comm)`, which is used most frequently (see Fig. 3).
- To send only the quadrature points of the center of faces we use `communicate_qpts_center (Cell*, MPI_Comm)` which will be used in after the first step of WENO interpolation (see Fig. 3).
- To communicate the intermediate quadrature points we use the function `communicate_qpts_intermediate (Cell*, MPI_Comm)` which will be used after second step of WENO method (see Fig. 3).



Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
Group Application	10.3193 s		14.516 s	4	2.57982 s
Process 0	2.5205 s		3.62726 s	1	2.5205 s
Process 1	2.49923 s		3.62969 s	1	2.49923 s
Process 2	2.61331 s		3.62957 s	1	2.61331 s
Process 3	2.68625 s		3.62944 s	1	2.68625 s
Group MPI	4.19667 s		4.19667 s	31854	131.747e-6 s
Process 0	1.10676 s		1.10676 s	7980	138.692e-6 s
Process 1	1.13046 s		1.13046 s	7958	142.053e-6 s
Process 2	1.01626 s		1.01626 s	7958	127.703e-6 s
Process 3	943.189e-3 s		943.189e-3 s	7958	118.521e-6 s

Figure 2: Top: load balance as a bar chart. Bottom: load balance as a pi chart.

- To communicate the gaussian quadrature points which are essential in space integration, we use `communicate_qpts_gauss_legendre (Cell*, MPI_Comm)` which will be called after WENO third step (see Fig. 3).
- To communicate all the data contained in the `class Cell` we use `communicate_all (Cell*, MPI_Comm)` which is only used in the old version of the code and during testing the MPI communications.

The challenging task to implement the specialized communications, is to consider the compiler padding. This means that the address of the location in memory that is returned by `MPI_Get_address ()` function is not necessarily the correct address and the function `MPI_Type_create_resized()` should always be called to take care of padding. Fig. 3 shows the code structure after implementing the specialized MPI communications.

After this changes we measured the parallel efficiency to see how much this change affects the performance. As it can be seen from Fig. 4 the difference between the normal code with uniform communication and the optimized code with specialized communications becomes more clear after using more than 64 cores. The optimized code is about 17 percent more efficient andd about 80 percent faster than the normal code in runtime at 216 cores.

4 Further optimizations

Using schemes that lead to less communications and therefore less overhead, can improve the performance and saves a lot of computational power. As it is also suggested in the proposal of this

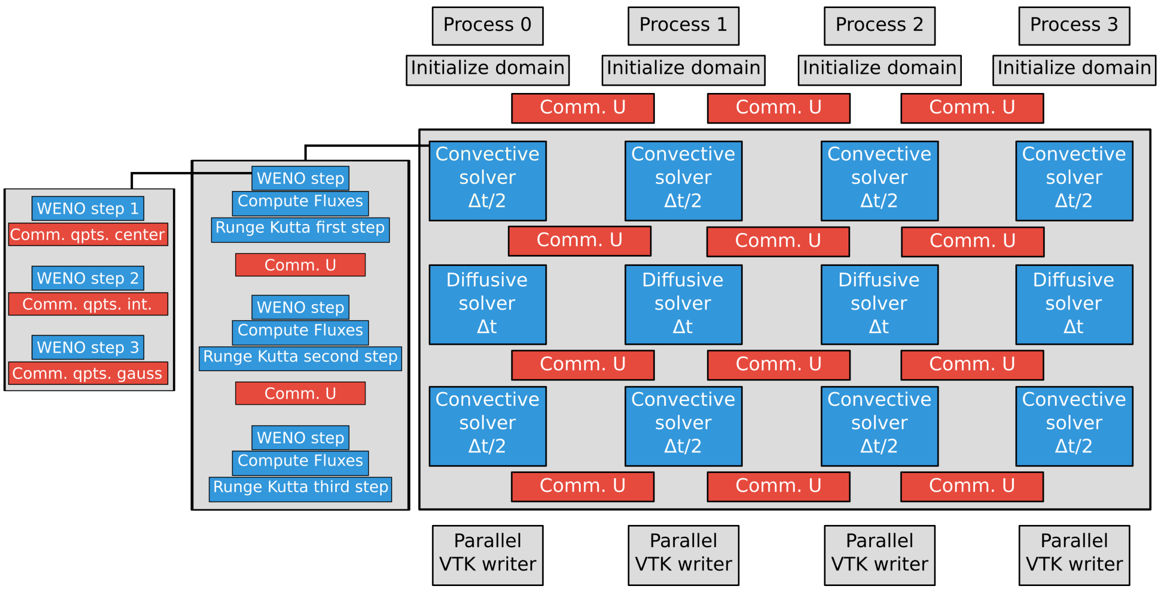


Figure 3: The schematic view of the code structure. The blue color shows the serial part and the red color shows the MPI communications. The three step WENO method that is used for interpolation needs three inter-communications.

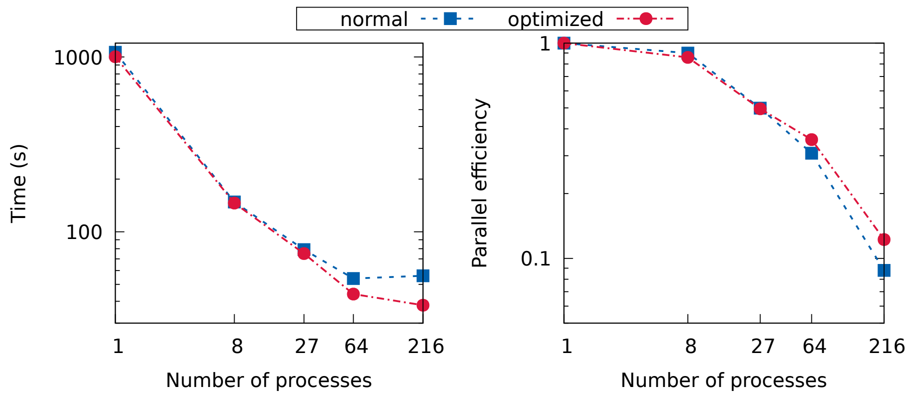


Figure 4: Left: the code run-time for normal and optimized versions. Right: parallel efficiency measured for normal and optimized versions. The optimized version has a better performance at higher number of cores although it is not very satisfactory.

project, we suggest two solutions to improve the software performance:

- **Implement ADER as time integration scheme**: ADER is faster, more accurate, higher order and need less computer memory, compared to the state of art finite volume WENO schemes [7]. ADER also improves the communication pattern due to the fact that it allows the design of a single step method that do not use numerical quadrature in time as opposed to multistep Runge-Kutta time integration method. On the other hand, ADER leads to less

WENO calls which again improves the performance. Therefore we propose to implement ADER scheme as our first improvement.

- **Develope a one-step WENO interpolation method**: A one step construction in which the interpolant is being reconstructed as a convex combination of biquadratic polynomials, $p(x, y)$, is suggested for two-dimensional systems [4]. To our knowledge, this method is not generalized to three-dimensional hyperbolic systems. The advantage of this one step WENO reconstruction [4], compared to frequently used three step WENO [5] is that it needs only one reconstruction sweep to obtain the quadrature values in all dimensions. Hence, less communications is required and the parallel overhead decreases by using the one step method.

5 Summary

In this work, we optimized a granular hydrodynamics flow code. The single-core optimizations lead to a code that is 14.6 times faster. The MPI optimization improved the efficiency by 17 percent at 216 cores. We believe that further optimizations is possible only by using more efficient algorithms. We suggest to use ADER scheme as time integration scheme and replacing the three step interpolation method by a single step method which only needs one communication per interpolation.

6 Acknowledgments

We gratefully acknowledge the support of the Competence Network for Technical and Scientific High Performance Computing in Bavaria (KONWIHR) in this research.

References

[1] Nikolai V Brilliantov and Thorsten Pöschel. *Kinetic theory of granular gases*. Oxford University Press, 2010.

[2] Helge Holden, Kenneth H Karlsen, and Knut-Andreas Lie. *Splitting methods for partial differential equations with rough solutions: Analysis and MATLAB programs*, volume 11. European Mathematical Society, 2010.

[3] Thomas Y Hou and Philippe G LeFloch. Why nonconservative schemes converge to wrong solutions: error analysis. *Mathematics of computation*, 62(206):497–530, 1994.

[4] Doron Levy, Gabriella Puppo, and Giovanni Russo. A fourth-order central weno scheme for multidimensional hyperbolic systems of conservation laws. *SIAM Journal on scientific computing*, 24(2):480–506, 2002.

[5] Jing Shi, Changqing Hu, and Chi-Wang Shu. A technique of treating negative weights in weno schemes. *Journal of Computational Physics*, 175(1):108–127, 2002.

[6] Gilbert Strang. On the construction and comparison of difference schemes. *SIAM Journal on Numerical Analysis*, 5(3):506–517, 1968.

[7] Vladimir A Titarev and Eleuterio F Toro. Ader schemes for three-dimensional non-linear hyperbolic systems. *Journal of Computational Physics*, 204(2):715–736, 2005.

[8] Eleuterio F Toro. *Riemann solvers and numerical methods for fluid dynamics: a practical introduction.* Springer Science & Business Media, 2013.

[9] Henk Kaarle Versteeg and Weeratunge Malalasekera. *An introduction to computational fluid dynamics: the finite volume method.* Pearson Education, 2007.