Proceedings of the 2023 International Conference on Machine Learning and Automation
DOI: 10.54254/2755-2721/40/20230654

Investigating the applications and analysis of physics engine technologies

Sheng Chen

Software College, Northeastern University, Shenyang, 110819, China

lrice72961@student.napavalley.edu

Abstract. This research project delves into the performance impact of implementing parallel programming techniques in physics engine applications. With the advent of multi-core processors in contemporary computing environments, optimizing physics simulations through parallel programming has become increasingly feasible. A conventional blob collision physics engine serves as the benchmark for evaluation, and its performance is juxtaposed against a parallel-programmed variant. Experimental findings indicate a significant reduction in computational time required for collision detection and response when parallel processing is employed. This efficiency gain is particularly pronounced in scenarios involving a large number of blobs, showcasing the scalability advantages of parallelization. Moreover, parallel programming facilitates optimal harnessing of multi-core processor capabilities, thereby enhancing the overall efficiency and performance of the physics engine in question. This study not only substantiates the technical merits of applying parallel programming but also illuminates the practical benefits, including resource-efficient operation and quicker simulation times. Consequently, the research provides valuable insights for developers and engineers aiming to fully exploit the capabilities of modern computing hardware in physics-based simulations and applications.

Keywords: Physics Engine, Collision Detection, Parallel Programming, Parallel Optimization, OpenMP, Speedup.

1. Introduction

The objective of this project is to engineer a robust physics engine capable of simulating the dynamics of hundreds of small spheres in a two-dimensional environment. Such a simulation demands precise processing of collision events, as well as meticulous calculation of each sphere's position and velocity post-collision.

To enable real-time or near real-time simulation, the engine must grapple efficiently with considerable data and computational load. Given the interactive nature of each sphere with others, algorithmic complexity grows exponentially with the quantity of spheres involved. Consequently, without any intervention, the simulation speed could be hampered significantly, especially when dealing with a high number of interacting spheres. To circumvent this challenge, the project incorporates parallel programming techniques. By assigning computational tasks across multiple processor cores, this approach facilitates simultaneous processing of numerous tasks. As such, the motions and interactions of multiple spheres can be processed in parallel, resulting in a substantial boost in the overall efficiency

DOI: 10.54254/2755-2721/40/20230654

of the simulation. In summary, the crux of this project lies in devising and implementing a physics engine tailored for sphere collision simulations. By leveraging parallel programming methods, the engine aims to efficiently handle voluminous data and complex computations, thereby achieving high-speed simulations under real-time or near real-time conditions.

2. Relevant theories

2.1. Physical Simulation

Physics engines primarily tackle two core issues: collision detection and collision resolution. Collision detection itself is commonly divided into two sequential phases—BroadPhase and NarrowPhase—to enhance computational efficiency. The BroadPhase serves as a rapid filtering mechanism, designed to eliminate object pairs that are unlikely to collide and thus pose no constraints. Pairs dismissed at this stage are not subjected to further scrutiny or resolution. Conversely, object pairs that survive the BroadPhase enter the NarrowPhase, where a more rigorous analysis is conducted to ascertain the likelihood of constraints. This phase yields precise data such as the collision point between objects, penetration depth, and separation normals, among other details. These findings are instrumental in the subsequent collision resolution phase, also known as the ResolvePhase. The ultimate aim of the ResolvePhase is to manipulate the position or velocity of the objects in question to satisfy a set of predefined conditions or constraints. This ensures that the simulated physics are as closely aligned with real-world behavior as possible, completing the dual process of detection and resolution that is central to the functionality of physics engines.

2.1.1. Broad Phase. In the Broad Phase of collision detection, the system conducts an initial screening of all potential object-pairs, eliminating those unlikely to collide and thus bypassing further calculations for them in subsequent phases. Axis-Aligned Bounding Boxes (AABB) serve as an effective tool for rapidly filtering out non-colliding object pairs. The technique involves encapsulating each object within a rectangle, followed by a check for overlap between each generated rectangle-pair. Should an overlap occur, the corresponding object-pair is considered likely to collide within the current frame. Conversely, if no overlap is found, the object-pair is excised from further refined calculations.

While this screening method is effective, it is computationally expensive due to its time complexity of $O(N^2)$.To ameliorate this issue, the Sweep and Prune algorithm is employed for further optimization. The underlying principle of this algorithm posits that if two AABBs overlap, their projections onto the x and y axes must likewise overlap [1]. An AABB projection onto any axis simplifies to a straight line. Hence, if the projections do not overlap on at least one axis, the AABBs cannot intersect, streamlining the detection process. In the context of the Sweep and Prune approach, the relative positioning of objects can shift as they move. However, due to the continuous nature of object movement, drastic changes within a single frame are unlikely. As such, the utilization of insertion sorting significantly accelerates the sorting process, thereby enhancing the overall algorithmic efficiency.

2.1.2. Narrow Phase. In the Narrow Phase of collision detection, objects identified as potential collision candidates during the Broad Phase are subjected to more rigorous scrutiny to ascertain actual collision events. Specifically, between two spheres, the determining factor for a collision is whether the distance between the centers of the circles exceeds the sum of their respective radii. By calculating this distance and comparing it to the sum of the radii, a direct conclusion about the occurrence of a collision can be drawn.

2.1.3. Resolve Phase. When two objects collide or are subject to an artificially defined constraint relationship, it becomes essential to resolve these constraints. Through this resolution process, attributes like the position, orientation, and velocity of the involved objects are adjusted to adhere to the corresponding constraint equation or inequality.

DOI: 10.54254/2755-2721/40/20230654

First, when two objects collide, the law of conservation of momentum is met. Let's say the momentum of the two objects before the collision are m_1v_1 and m_2v_2, and then according to the momentum conservation theorem, the sum of the momentum of the two balls after the collision m_1v_3 plus m_2v_4 still remains unchanged. And the formula is as below.

$$m_1v_1 + m_2v_2 = m_1v_3 + m_2v_4 m_1v_1 + m_2v_2 = m_1v_3 + m_2v_4 \quad (1)$$

Second, two objects meet the law of conservation of mechanical energy before and after collision. That is, if no external force does work and only conservative forces do work in the system, the mechanical energy of the system remains unchanged. Let's say the mechanical energy of the two objects before the collision are $\frac{1}{2}m_1{v_1}^2$ and$\frac{1}{2}m_2{v_2}^2$, and then according to the mechanical energy conservation theorem, the sum of the mechanical energy of the two balls after the collision $\frac{1}{2}m_1{v_3}^2$plus $\frac{1}{2}m_2{v_4}^2$still remains unchanged. And the formula is as below.

$$\frac{1}{2}m_1{v_1}^2 + \frac{1}{2}m_2{v_2}^2 = \frac{1}{2}m_1{v_3}^2 + \frac{1}{2}m_2{v_4}^2 \quad (2)$$

Therefore, by combining the law of conservation of momentum, and the law of conservation of mechanical energy, the velocities after the collision, v_3 and v_4 can be calculated.

2.2. Parallel Optimization

Details of parallelization are explored using OpenMP to enhance the efficiency of collision detection, which involves extensive loops and floating-point calculations. Certain OpenMP pragmas, theoretically suited for this task, are selected for implementation. The efficacy of these pragmas is subsequently verified by measuring the time required for frame updates.

2.2.1. Profiling. Before embarking on any code modifications, it is crucial to identify the key performance indicators for the project. Utilizing a benchmarking tool can help establish a baseline performance metric. Once this baseline is established, profiling tools can identify the functions within the code that are most time-consuming [2]. Subsequent steps involve optimizing the algorithms for these bottleneck functions and implementing parallelization techniques to further improve performance. After these changes are implemented, performance metrics should be reassessed and compared to the initial baseline. This comparison will indicate whether the modifications have led to performance improvements. A decision can then be made on whether the updated performance is satisfactory or if further optimization is required. Given that the project is developed using Visual Studio on a Windows OS, the built-in diagnostic tools of Visual Studio serve as the profiling tool of choice [3]. In this particular program, the primary focus is on the time consumption of each individual function. Accordingly, optimization efforts are primarily targeted at the most time-consuming functions.

2.2.2. OpenMP. In the realm of thread-level parallelism, several options exist including pthreads, OpenMP, and MPI. Given that the focus is not on cluster computing, MPI is set aside in favor of OpenMP for its stability and ease of implementation. The OpenMP API offers an assortment of compiler directives, library functions, and environment variables. This provides a parallel programming model compatible with architectures from a variety of manufacturers, with several companies offering dedicated OpenMP API compilers [4].

To manage irregular yet relatively independent tasks, the directive #pragma omp task is employed [5]. This is often combined with #pragma omp parallel and #pragma omp single. Under the #pragma omp parallel directive, a team of OpenMP threads execute the code region. Within this parallel context, #pragma omp single ensures that only one thread in the team executes the associated structured block [6]. However, special settings are required when utilizing task directives in a Visual Studio environment [7]. Figure 1 in the documentation illustrates the application of omp parallel, omp single, and omp task directives in the broad phase stage of collision detection. This enables x-axis and y-axis detection in the Sweep and Prune algorithm to proceed concurrently [8]. Instead of employing #pragma omp taskwait

DOI: 10.54254/2755-2721/40/20230654

to create an explicit barrier, the implicit barrier provided by #pragma omp single is used to minimize overhead.

```
1.  void State::broadPhase()
2.  {
3.   this->possible_on_x.clear();
4.   this->possible_on_y.clear();
5.   this->possible_collision_pairs.clear();
6.  #pragma omp parallel
7.   {
8.  #pragma omp single
9.    {
10.    detectAxises();
11.   }
12.  }
13.  intersectAxises();
14. }
15.
16. void State::detectAxises() {
17. #pragma omp task
18. detectAxisX();
19. #pragma omp task
20.  detectAxisY();
21. }
```

Figure 1. The using of omp parallel, omp single and omp task directives in broad phase of the stage of collision detection (Photo/Picture credit: Original).

To achieve loop parallelization, the #pragma omp parallel directive is employed. This directive signifies that the subsequent 'for' loop will be executed in a multi-threaded fashion, under the condition that there is no interdependence between each loop iteration. Upon completing their respective tasks, the threads in the team await at an implicit barrier at the conclusion of the single construct, unless a 'nowait' clause is specified [9].

Data-sharing attribute clauses, commonly known as reduction clauses, offer a mechanism for executing specific types of recursive calculations concurrently. Reduction clauses are generally categorized into two types: participating clauses and scoping clauses. The former describes the participants involved in the reduction, while the latter outlines the area where the reduction is computed. Each reduction clause specifies a reduction-identifier and one or more list elements [10]. In contrast to utilizing lock routines within the 'for' loop, the employment of reduction clauses significantly minimizes thread waiting times when attempting to access shared variables concurrently. However, due to version limitations, OpenMP within the utilized version of Visual Studio does not support the reduction-identifier. As an alternative, a unique private vector is designated for each thread, followed by a serial merging of these vectors outside the parallel region [11]. The use of #pragma omp critical is integrated to ensure that threads populate the 'possible_collision_pairs' vector in a serialized manner, thereby maintaining data consistency. Figure 2 showcases the application of 'omp parallel for,' 'omp critical' directives, and private vectors in the broad phase of the collision detection stage, enabling the parallelization of intersection computations between x-axis and y-axis collision detection results.

DOI: 10.54254/2755-2721/40/20230654

```
void State::intersectAxises() {
 int x{ (int)this->possible_on_x.size() };
 int y{ (int)this->possible_on_y.size() };
#pragma omp parallel
 {
  std::vector<std::pair<Object*, Object*>> vec_private;
#pragma omp for nowait
  for (int i{ 0 }; i < x; i++)
  {
   for (int j{ 0 }; j < y; j++)
   {
    if (this->possible_on_x[i].first == this->possible_on_y[j].first &&
     this->possible_on_x[i].second == this->possible_on_y[j].second)
    {
     vec_private.push_back(this->possible_on_x[i]);
     break;
    }
   }
  }

#pragma omp critical
  this->possible_collision_pairs.insert(this->possible_collision_pairs.end(),
  vec_private.begin(), vec_private.end());
 }
}
```

Figure 2. The using of omp parallel for, omp critical directives and private vector in broad phase of the stage of collision detection, so as to we can parallelize the intersection computation between the results of collision detection in x-axis and y-axis (Photo/Picture credit: Original).

2.2.3. Evaluation of efficiency. Speedup is the ratio of the execution time for the entire task without the enhancement to the execution time for the entire task with the enhancement. And the formula is as below

$$Speedup = \frac{P_e}{P_w} = \frac{E_w}{E_e} \tag{3}$$

where

P_e is the performance of the entire task using improvement;

P_w is the performance of the entire task without the use of improvement;

E_w is the time -consuming of the entire task not using the improvement;

E_e is the time -consuming of the entire task using the possible improvement.

Amdahl's law states the theoretical latency speedup of a task with a given data set in a system whose code are optimized. To put it simply, it is a formula that identifies the maximum improvement possible by simply enhancing a specific component of a system. And the formula is as below [12].

$$S_{latency}(s) = \frac{1}{(1-p)+\frac{p}{s}} \tag{4}$$

where

$S_{latency}$ is the theoretical latency speedup of the run of the entire task;

s is the actually speedup of the run of the optimized task;

p is the ratio of time-consuming that the originally enhanced resource portion occupied.

In addition,

DOI: 10.54254/2755-2721/40/20230654

$$\begin{cases} S_{latency}(s) \leq \frac{1}{1-p} \\ \lim_{s\to\infty} S_{latency}(s) = \frac{1}{1-p} \end{cases} \tag{5}$$

Gustafson's law is a measurement of how the time-consuming of the system changes as there are more processors in the system with a fixed data set scale per processor; i.e., where the data set scale increases along with the increasing of the amount of processors. And the formula is as below.

$$S_{latency} = (1-p) + Np \tag{6}$$

where

$S_{latency}$ is the theoretical latency speedup of the run of the entire task;

N is the amount of processors the hardware has;

p is the ratio of time-consuming that the originally enhanced resource portion occupied.

The function is linear, and the slope is p. As $N \to \infty$, the intercept becomes less important, i.e., $S_{latency} = Np$ [13].

3. System analysis and application research

3.1. Serial

Initial performance analysis of the ball collision program, along with frame rate evaluation, was conducted using Visual Studio's CPU profile performance tool [14]. The examination revealed that the "Update" function is a significant CPU time consumer, accounting for approximately 60% of the overall program's CPU usage. Additionally, the program suffers from a low frame rate. Observations on CPU utilization indicate substantial time expenditure per frame. Specifically, the collision elapsed time per frame for a simulation of 200 small balls registers at 800 microseconds. Without any optimization, this duration demonstrates considerable potential for improvement. Code scrutiny shows that the program employs a 'for' loop for ball collision detection, resulting in an algorithmic complexity of O(n^2) [15]. Hence, there are two primary avenues for performance optimization: one is the improvement of algorithmic complexity, and the other is the parallelization of the 'for' loop..

3.2. Parallel Optimization

In this section, the focus initially centers on profiling the time consumption of various functions within the program. Following this, the efficacy of parallel enhancements is assessed through a comparative analysis between parallelized and serial versions of collision detection, employing speedup as the evaluation metric. Lastly, performance gains resulting from parallel optimizations are scrutinized across different scalability scenarios..

3.2.1. System Setup. We use Intel i5-7200U CPU as our testbed. It has 2 2.50GHz cores. Each core has a 3 MB Intel Smart Cache. And it has 8 GB of memory. The Operating System is Windows 10 64-bit. The IDE is Visual Studio Community 2022 (64-bit) - 17.6.5 and the version of OpenMP the IDE use is C/C++ Version 2.0.

3.2.2. Profiling. According to the Amdahl's law in Section 3.2.3, A function that accounts for the highest proportion of running time is identified for optimization. Table 1, generated through profiling in Visual Studio, indicates that 'broadPhase' emerges as the most time-consuming function developed. Accounting for 9.41% of the entire program's running time, as displayed in the third column, this function is earmarked for significant optimization using OpenMP.

DOI: 10.54254/2755-2721/40/20230654

Table 1. The result of profiling using Visual Studio.

Function	CPU total (%)	CPU self (%)	Module
std:vector<std::pair<Object *,Object *>,std::allocator<std::pair<Object *,Object *> > >::operator[]	7552(29.04%)	5031 (19.35%)	multhrphy
gdi32full.dll!0x00007ffce0939e2e	2881 (11.08%)	2881 (11.08%)	gdi32full
__CheckForDebuggerJustMyCode	2558 (9.84%)	2480 (9.51%)	multhrphy
State:broadPhase	11304 (43.47%)	2448 (9.41%)	multhrphy
ig9icd64.dll!0x00007ffbff10d9a5	2239 (8.61%)	2239 (8.61%)	ig9icd64
ig9icd64.dll!0x00007ffbff10d991	1758 (6.76%)	1758 (6.76%)	ig9icd64
msvcp140d.dll!0x00007ffbffc245cd	640 (2.46%)	640 (2.46%)	msvcp140d
msvcp140d.dll!0x00007ffbffc35778	551 (2.12%)	551 (2.12%)	msvcp140d

3.2.3. Comparison of Parallel version to Serial version. The comparison between the parallelized and serial versions of collision detection utilizes speedup as a key metric across varying data set scales. As demonstrated in Table 2 and Figure 3, interesting trends emerge. Specifically, a data set scale of 100 yields a speedup of 1. Upon increasing the data set scale to 200, the speedup peaks at 2.39. Beyond this point, however, the speedup shows a diminishing trend as the data set scale continues to rise. As outlined in Gustafson's Law in Section 3.2.3, maintaining or increasing the speedup could be achievable through an augmentation in the number of threads in tandem with the increase in data set scale.

Table 2. The parallelized collision detection, to serial version using speedup as metric in different data set scale. And the number of logical threads is 4.

Data set scale	Seq(microseconds)	Omp(microseconds)	Speed-up
100	501	502	1
200	6078	2539	2.39
400	73801	32466	2.27
600	272861	145405	1.88
800	548801	329028	1.67

DOI: 10.54254/2755-2721/40/20230654

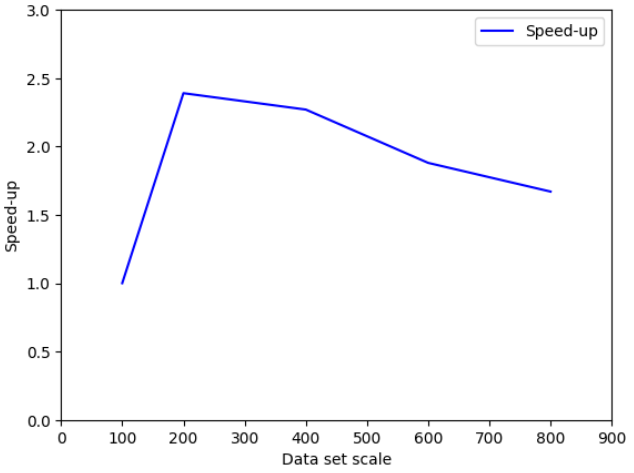

Figure 3. The parallelized collision detection, to serial version using speedup as metric in different data set scale. And the number of logical threads is 4 (Photo/Picture credit: Original).

*3.2.4. Parallel Scalability.*Performance improvements from the parallel-optimized version were evaluated using speedup as the metric, with scalability tested across varying configurations. The data set size remained constant at 200. Both Table 3 and Figure 4 demonstrate the relationship between speedup and the number of threads employed. Notably, maximum speedup is achieved at four threads, reaching a value of 2.39. Any increase in the number of threads beyond this point results in diminishing returns on speedup. This can be attributed to the CPU's architecture, as outlined in section 4.2.1, which reveals the presence of four logical cores. Consequently, deploying more than four threads yields a less significant advantage from parallelization, while concurrently escalating the overhead associated with it.

Table 3. The performance improvements of our parallel optimization version using speedup as metric in different scalability. And the data set scale is fixed at 200.

Number of Threads	Seq(microseconds)	Omp(microseconds)	Speed-up
1	6078	6078	1
2	6078	3835	1.58
3	6078	3095	1.96
4	6078	2539	2.39
5	6078	3243	1.87
6	6078	3828	1.59
7	6078	2823	2.15
8	6078	2265	2.68
9	6078	3741	1.62
10	6078	3491	1.74

DOI: 10.54254/2755-2721/40/20230654

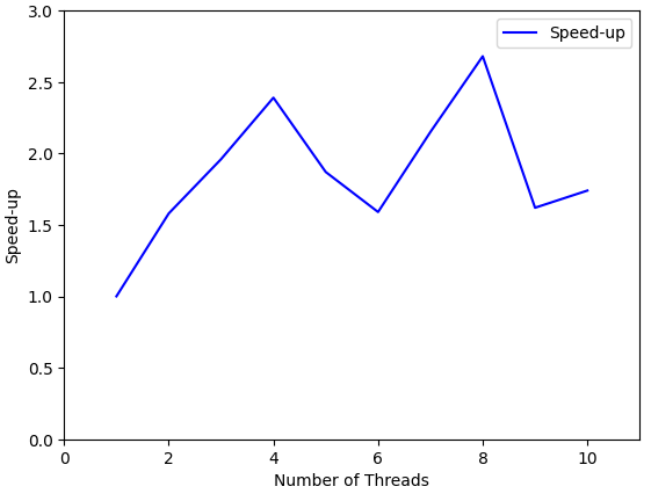

Figure 4. The performance improvements of our parallel optimization version using speedup as metric in different scalability. And the data set scale is fixed at 200 (Photo/Picture credit: Original).

4. Conclusion

In this study, the collision processing workflow is refined by introducing an additional stage, resulting in a total of three stages. This modification serves to minimize superfluous operations, ensuring that CPU time is optimally utilized. Specifically, the Broad Phase employs the Sort and Sweep algorithm, contributing to a significant reduction in time complexity. Furthermore, the study reveals that collision detection is the most resource-intensive component of the physics engine. Within this stage, the Broad Phase consumes the most computational time. By leveraging OpenMP for parallel optimization, the study achieves a twofold speedup in performance. To maintain this enhanced efficiency while accommodating increasing data set sizes, it is recommended to augment the number of available CPU cores.

References

[1] Vasheghani Farahani M, Foroughi S, Norouzi S, et al. Mechanistic study of fines migration in porous media using lattice Boltzmann method coupled with rigid body physics engine[J]. Journal of Energy Resources Technology, 2019, 141(12): 123001.

[2] Freeman C D, Frey E, Raichuk A, et al. Brax--A Differentiable Physics Engine for Large Scale Rigid Body Simulation[J]. arXiv preprint arXiv:2106.13281, 2021.

[3] Millington I. Game physics engine development[M]. CRC Press, 2007.

[4] Vasheghani Farahani M, Foroughi S, Norouzi S, et al. Mechanistic study of fines migration in porous media using lattice Boltzmann method coupled with rigid body physics engine[J]. Journal of Energy Resources Technology, 2019, 141(12): 123001.

[5] Quinn M J. Parallel programming[J]. TMH CSE, 2003, 526: 105.

[6] Culler D E, Dusseau A, Goldstein S C, et al. Parallel programming in Split-C[C]//Supercomputing'93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing. IEEE, 1993: 262-273.

[7] Gilles K. The semantics of a simple language for parallel programming[J]. Information processing, 1974, 74(471-475): 15-28.

[8] Chandra R. Parallel programming in OpenMP[M]. Morgan kaufmann, 2001.

[9] Huber J, Cornelius M, Georgakoudis G, et al. Efficient execution of OpenMP on GPUs[C]//2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE, 2022: 41-52.

DOI: 10.54254/2755-2721/40/20230654

[10] Yviquel H, Pereira M, Francesquini E, et al. The OpenMP Cluster Programming Model[C]//Workshop Proceedings of the 51st International Conference on Parallel Processing. 2022: 1-11.
[11] Doerfert J, Patel A, Huber J, et al. Co-Designing an OpenMP GPU runtime and optimizations for near-zero overhead execution[C]//2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 2022: 504-514.
[12] Pham B Q, Alkan M, Gordon M S. Porting fragmentation methods to graphical processing units using an OpenMP application programming interface: Offloading the Fock build for low angular momentum functions[J]. Journal of Chemical Theory and Computation, 2023, 19(8): 2213-2221.
[13] Silva H U, Lucca N, Schepke C, et al. Parallel OpenMP and OpenACC porous media simulation[J]. The Journal of Supercomputing, 2023, 79(8): 8425-8446.
[14] Marques S M V N, Serpa M S, Muñoz A N, et al. Optimizing the edp of openmp applications via concurrency throttling and frequency boosting[J]. Journal of Systems Architecture, 2022, 123: 102379.
[15] Da Silva H U, Schepke C, Lucca N, et al. Parallel openmp and openacc mixing layer simulation[C]//2022 30th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP). IEEE, 2022: 181-188.