2. Parallel Computing Using a Multi-core CPU

While improvements in CPU clock speed (speed increases) for modern personal computers have somewhat plateaued, CPUs featuring multiple cores and threads—incorporating multiple processing units within a single chip—have emerged, and compilers supporting parallel computing via OpenMP (Reference 22) have become commonplace. The most efficient way to parallelize DSMC calculations is to divide the flow field into a number of regions equal to the number of threads and assign each region to a specific thread; however, this necessitates the transfer of molecules moving across boundaries between regions. Most previously reported parallelizations of DSMC calculations utilize this method; while limited to simple flow fields, reports indicate that computation speed is roughly proportional to the number of CPUs (or cores). Alternatively, one could consider parallelizing the DSMC method by targeting individual computational routines. While this approach makes it difficult to achieve complete independence between parallelized tasks—potentially limiting the expected performance gains—once the method is established, it can be applied to accelerate virtually any DSMC calculation with only minor modifications to the program code. Adopting this latter approach, this report outlines a method for parallelizing individual routines within a DSMC program using the Intel Fortran Compiler V12, and presents the resulting speed-up and analysis outcomes. The DSMC method comprises five primary computational routines—molecule movement, cell assignment, molecule reordering, inter-molecular collisions, and flow field sampling—which are executed repeatedly in that specific order. The parallelization procedures for all these routines are described below.

2.1 General Aspects of Parallel Programming: Let us consider a parallel execution loop consisting of N iterations by dividing it into a single specific loop (referred to as the "self-loop") and the remaining N-1 loops (referred to as "other-loops"). In this parallel processing setup, although the actual calculations within different threads can be performed independently (in parallel), they share the same memory locations for data. Consequently, a problem arises—which we will term "variable reference conflict"—when a thread processes a variable and writes the result back to the original

memory location, but another thread accesses that same variable before the write-back is complete, leading to incorrect results. The objective of this study is to develop a high-speed parallel DSMC program while preventing such conflicts. In OpenMP, the standard procedure to prevent variable reference conflicts involves making the following declarations immediately before the DO loop:

!$OMP PARALLEL DO PRIVATE(…, …, …)
!$OMP REDUCTION (+: …, …, …)

Specifically, variables used independently within the parallel computation are declared inside the parentheses following `PRIVATE`, while variables used for operations spanning parallel threads—such as calculating a sum—are declared inside the parentheses following `REDUCTION`.

2.2 Generation of Uniform Random Numbers: In parallel DSMC programming using 16 threads, at least 16 independent sequences of uniform random numbers are required. For this study, we utilized two independent random number sequences with a period of 2^{46}. We established initial values at points dividing each period into 8 or 9 segments and generated 17 sequences of uniform random numbers (with one reserved for sequential execution) starting from these points.

2.3 Method for Assigning Molecules or Cells to Threads: DSMC calculations involve the parallel execution of independent computations based on individual molecules or cells; however, these must be assigned to the 16 threads as evenly as possible. If random numbers are not used within a loop, one can simply parallelize the DO loop iterating over molecules or cells; however, when random numbers are involved, it is best to determine the thread number (an integer from 1 to 16) using `OMP_GET_THREAD_NUM() + 1` and obtain the corresponding random number sequence based on this value. Furthermore, to balance the computational load across threads, different scheduling strategies are appropriate: `Schedule(guided)` is suitable for molecular movement, as the computational load for moving individual molecules does not vary significantly; conversely, `Schedule(dynamic)` is preferable for inter-molecular collisions, as the

number of molecules per cell can vary widely, potentially leading to load imbalances between cells.

2.4 Parallelization of Molecular Movement: In this calculation routine, individual molecules generally move through space independently, so parallelization can be achieved by dividing the total number of molecules into 16 groups. However, discarding molecules that have flowed out of the computational domain requires reorganizing the arrays storing molecular data—a process that cannot be easily parallelized. Since the number of discarded molecules was relatively small in this instance, the parallel computation for spatial movement merely tagged the molecules to be discarded, while the actual removal process was performed sequentially outside the parallel computation block. Regarding the movement of molecules entering from the boundary, while this is typically handled after the movement of existing flow-field molecules is complete, in this implementation, molecules were generated on the boundary beforehand, and all molecules—including these new ones—were moved through space within a single parallel computation step.

2.5 Parallelization of Cell Identification: This calculation is independent for each molecule, allowing for easy parallelization and accelerated performance. To further enhance processing efficiency, the cell identification step was performed concurrently with the parallel computation of molecular spatial movement.

2.6 Parallelization of the Molecular Sorting Process: This routine, which rearranges molecules into a "Molecular Cross-Referencing Array" (LCR(K)) on a cell-by-cell basis, involves interdependencies between molecule indices and cell indices; consequently, variable access conflicts arise, making straightforward parallelization difficult. However, if a core part of the program cannot be parallelized, it becomes a bottleneck that prevents significant speedups. Therefore, we attempted to accelerate the process using a specialized parallel computing method. To explain this method, it is first necessary to understand the molecular sorting process itself. As shown in Figure 1, this process involves arranging all molecules according to the cells to which they belong to create a "one-dimensional reference array." By

placing molecules in contiguous array positions for each cell, it becomes easy to select pairs of molecules within the same cell—which are candidates for collisions—using random numbers.

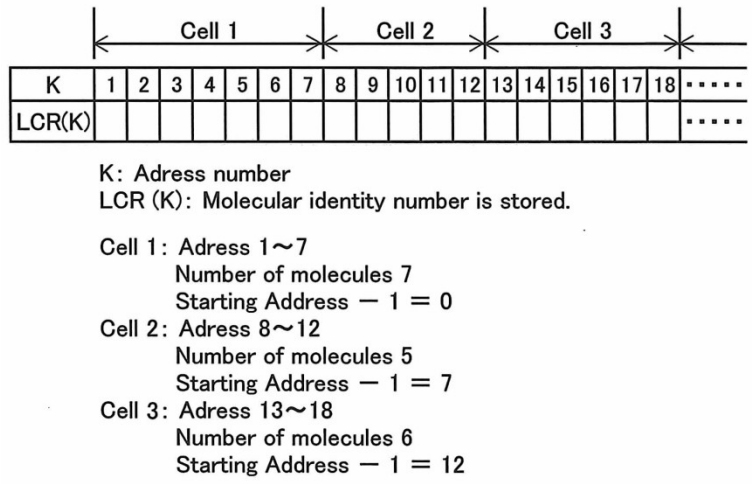

Figure 1: Example of a molecular reference sequence

To create this reference array, the total number of molecules is divided by the number of threads to assign a specific set of molecules to each thread; initially, each thread generates a reference array limited to its assigned molecules through parallel computation. By concatenating the partial reference arrays generated by each thread on a cell-by-cell basis, the required "molecular reference array" is obtained. However, this "concatenation process" also requires careful design to maximize the use of parallel computation. While the method described above necessitates a large array—sized according to the product of the number of cells and the number of threads—it offers the advantage of using the same program for any DSMC analysis.

2.7 Parallelization of Intermolecular Collisions: Since calculations in this routine are independent for each cell, parallelization is feasible. Furthermore, as the proportion of total computation time dedicated to intermolecular collisions increases with higher densities, optimizing this routine contributes most significantly to overall speedup. As previously

mentioned—particularly in axisymmetric analyses like the one here—the number of molecules per cell varies widely, meaning the computational load is not uniform across cells; therefore, it is advisable to specify "Schedule (Dynamic)" to prioritize assigning tasks to available threads.

2.8 Parallelization of Flow Field Sampling: As the scale of computation grows, the time required for this routine becomes a significant portion of the total execution time, making it a prime target for acceleration. In this study, we parallelized the process on a cell-by-cell basis using the molecular reference array LCR(K)—generated during the earlier molecular reordering step—and further improved efficiency by integrating this sampling into the intermolecular collision calculation loop.

2.9 Ordering of Molecular Information Arrays: When initial values are assigned to the array variables storing the positional coordinates and velocity components of all molecules, the data is organized sequentially by cell number. However, as the molecules subsequently move through space, the molecular information within these arrays loses this cell-based order (which is precisely why the reference array LCR(K) is required). However, it was determined that calculation speeds are clearly faster when the array variables storing molecular information are organized according to cell order (though strict adherence to cell order is not required); therefore, a procedure was added to reorder these array variables after all molecules have undergone spatial movement ten times. Notably, this procedure is effective not only for parallel computing but yields even greater benefits in sequential computing; in the analysis with a pressure ratio of 50, the speed increased by approximately 1.45 times for parallel computing, whereas it increased by approximately 1.7 times for sequential computing.

3. Acceleration via Parallel Computing

The effectiveness of parallel computing was investigated for an underexpanded jet of monatomic Argon (Ar) gas in an axisymmetric flow field. Tables 1 and 2 present results for a pressure ratio of 50 calculated using the U-system, showing the difference in computation time between

sequential execution on a single core and parallel execution on eight cores (16 threads). In cases where multiple calculation routines are combined during parallel computing, the total time is shown. Table 1 displays results from a simplified program that does not include the flow field upstream of the orifice (approximately 58 million molecules), while Table 2 shows results from a program that includes the upstream flow field and supports gas mixtures (approximately 140 million molecules). The values in the tables are dimensionless, with the total time for all parallel computing routines normalized to 1.0. For sequential computing, results are shown for both the code specifically written for sequential execution and the code written for parallel execution when run sequentially. Although it is surprising that there is little difference in computation time between the different program codes during sequential execution, this is likely because other cores may be assisting the computer with tasks other than the actual numerical calculations, even during sequential processing. Regarding the current calculations, intermolecular collision computations account for a significant portion of the total execution time (over 90%). Since parallelization accelerated the collision computations by a factor of 7 to 8, the overall computation speed also increased by a similar factor; this represents a reasonably effective outcome for an 8-core CPU.

Figures 2 through 5 illustrate the impact of the number of threads on computation speed for the case corresponding to item ① in Table 1. The computer's BIOS settings allow for limiting the number of usable cores and toggling Hyper-Threading Technology (HTT). Consequently, while using 8 cores with HTT disabled results in 8 threads, using 4 cores with HTT enabled also yields 8 threads; this results in 16 distinct configuration scenarios but 12 variations in the number of threads (see the horizontal axis of the figures). The vertical axis represents "speed-up," defined as the ratio of the parallel code's speed at a given thread count to the speed of the sequential code (using a single thread). Similar calculations were also performed for the case in Table 2 (involving approximately 2.4 times the number of molecules and 1.05 times the number of cells), but since they yielded nearly identical trends, they are omitted here.

Table 1: Comparison of computation times for DSMC routines between sequential and parallel calculations, expressed as dimensionless values with the total parallel computation time set to 1 (program without an upstream computational region)

	Sequential ① (Sequential Code)	Sequential ② (Pallalel Code)	Parallel
Movement	0.166	0.34	0.057
Cell ident.	0.188		
Sorting	0.06	0.08	0.029
Collision	6.6	6.6	0.91
Sampling	0.14		
Whole	7.2	7.0	1.0

Table 2: Comparison of computation times for DSMC routines between sequential and parallel calculations, expressed as dimensionless values with the total parallel computation time set to 1 (program with an upstream computational region; supports gas mixtures)

	Sequential ① (Sequential Code)	Sequential ② (Pallalel Code)	Parallel
Movement	0.074	0.185	0.022
Cell ident.	0.098		
Sorting	0.02	0.11	0.012
Collision	7.5	7.9	0.97
Sampling	0.13		
Whole	7.8	8.2	1.0

Figure 2 compares the total execution times for all computational routines; the blue bars represent the case without HTT, while the red bars represent the case using HTT with two threads per core. In this instance, the computational speed increases with the number of threads, regardless of whether HTT is used. However, as the thread count approaches the maximum of 16, the rate of speed increase slows, resulting in a speedup of approximately 7.4 times at 16 threads. While the value indicated in Table 1 (item ①) was 7.2 times, the figure of approximately 7.4 times observed in Figure 2 is likely due to the following: in Table 1, tasks other than the DSMC calculation (such as Windows OS operations) were handled by other threads with spare capacity; conversely, in the comparison shown in Figure 2, all tasks—including the DSMC calculation—had to be performed solely by the allocated threads, thereby increasing the computation time of the single reference thread. Figure 3 shows the results for the total time spent on intermolecular collisions and flow-field sampling. Since this calculation

involves a flow field with relatively high density—where intermolecular collision calculations account for a large proportion of the workload—the results in Figure 3 significantly influence the overall routine results, leading to similar trends in Figures 2 and 3. Although this routine is parallelized by collision cell, the number of molecules assigned to each cell varies, making it likely that computational loads will differ among threads; this is considered one of the reasons why the rate of speed increase slowed as the thread count approached its maximum.

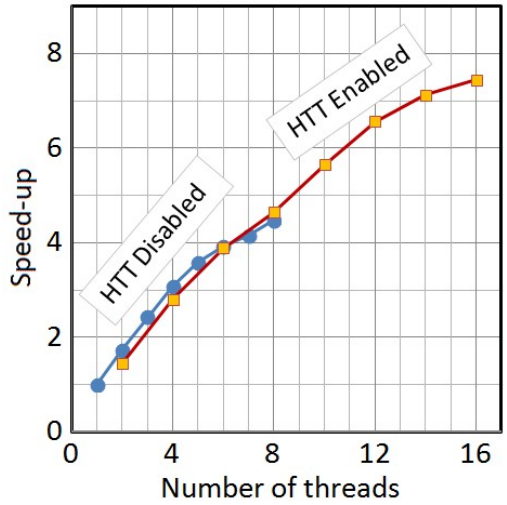

Figure 2: Number of threads vs. computation speed (total of all computational routines)

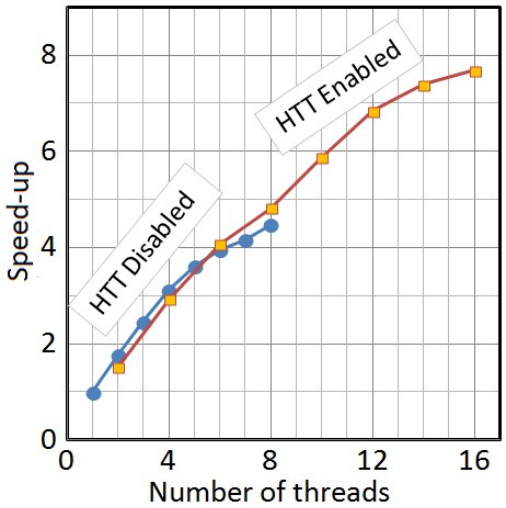

Figure 3: Number of threads vs. computation speed (intermolecular collisions and flow-field sampling)

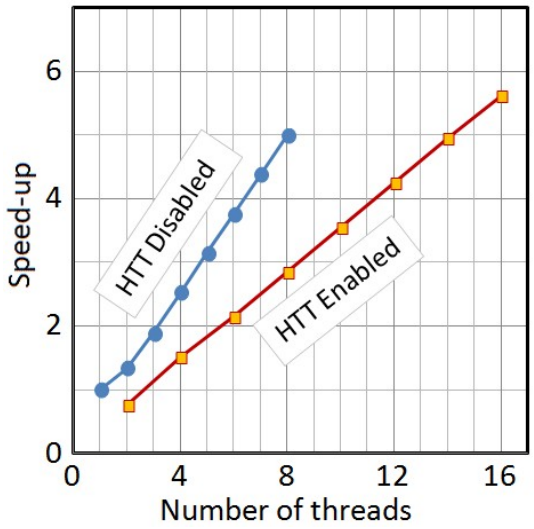

Figure 4: Number of threads vs. computation speed (molecular movement and cell assignment check)

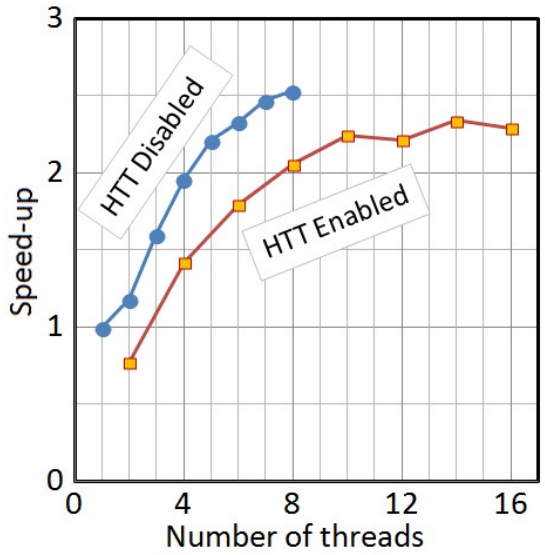

Figure 5: Number of threads vs. computation speed (molecular reordering process)

Figure 4 compares the total time spent on molecule movement and determining the cell to which each molecule belongs. This routine parallelizes straightforward calculations performed for individual molecules; given that the computational load per molecule is nearly uniform and the number of molecules is large, the computational speed increases in

proportion to the number of cores. However, when comparing HTT-enabled and HTT-disabled configurations for the same number of cores, the effect of enabling HTT—effectively doubling the number of threads—is minimal. Figure 5 shows the section corresponding to the molecular reordering process; because this part—which could not be parallelized in the original program—was parallelized using a specialized technique, the resulting speedup is modest. However, since this section accounts for less than 3% of the total computational workload in this example, it has little impact on the overall speed improvement. Speed gains plateau when the number of threads exceeds 10, causing the execution speed with HTT enabled to fall below that with HTT disabled. There appears to be room for further improvement in the parallelization of this section. Regarding the cases with a single core shown in Figures 4 and 5, the fact that performance is slower with HTT enabled than with HTT disabled is likely because the baseline case (one thread, single core, HTT disabled) utilizes sequential code, whereas the other cases utilize parallel code, which inadvertently resulted in slower execution.