Parallel Implementation of the Direct Simulation Monte Carlo Method For Shared Memory Architectures

Da Gao* and Thomas E. Schwartzentruber†

Department of Aerospace Engineering and Mechanics, University of Minnesota, Minneapolis, MN 55455

Parallel implementation of a three-dimensional direct simulation Monte Carlo (DSMC) code is detailed that employs shared memory system using Open Multi-Processing (OpenMP). Several techniques to optimize the serial implementation of the DSMC method are also discussed. The synchronizations in OpenMP, as well as the related critical sections, have been identified as major factors that impact the OpenMP parallel performance. Methods to remove such barriers in the OpenMP implementation of the DSMC method are presented. For dual-core and quad-core systems, speedups of 1.99 and 3.74, respectively, are obtained for the OpenMP implementation. It is also reported that memory fetching and data communication within the same node but across sockets needs further improvement in order to achieve acceptable scalability for clusters of multi-socket, shared-memory architectures.

I. Introduction and Motivation

At the turn of the century, Knudsen\(^1\) showed that the transition regime, for which the mean free path of gas molecules becomes comparable with the geometric dimension of the flow, cannot be described by continuum hydrodynamics. Such flows are characterized by Knudsen numbers of order one and are found in a variety of fields such as high-altitude aerodynamics, aerosol dynamics, vacuum-pump operation, and nano flows. Invented as a numerical method for simulating the Boltzmann equation gas flows in the transition regime, the direct simulation Monte Carlo (DSMC) method\(^2\) has become a widely used computational tool. In contrast to continuum methods where macroscopic averages and transport closure models are used, within a DSMC simulation, mass, momentum, and energy are transported through molecular movement and collisions. As a result, the DSMC method accurately simulates flows ranging from continuum to free-molecular. However, the larger computational cost associated with the DSMC method for transitional and continuum gas flows can be substantial, especially for full-scale macroscopic gas flows of engineering interest in the near-continuum regime.

For this reason, improving parallel scalability of the DSMC method on large work-station clusters is an ongoing research area. Existing parallel implementations of the DSMC method are quite few and include the DS2V/3V codes,\(^3\) the MONACO code,\(^4\) the DAC\(^5\) code, and the SMILE\(^6\) code, as examples. None of these existing DSMC implementations take advantage of the multi-core-per-node systems that have been an industry trend for many years. As the computational power of computing resources steadily increases, the main drawback of the DSMC method (its computational cost) may become less and less significant if optimal usage is made of the available resources.

Current computational resources already enable DSMC simulations involving hundreds of millions of particles. Continually evolving hardware architectures such as multi-core systems and graphical processing units (GPUs) necessitate continued research including both hardware and software aspects. As DSMC simulations grow, so too does the code’s complexity in terms of CPU usage and memory management including dynamic memory allocations/deallocations and data structures. Furthermore, DSMC simulations for problems involving large density gradients (found in hypersonics, rocket plumes, and electric propulsion) require adaptive mesh refinement, variable local time steps, and other techniques to control the number of particles per cell in order to maintain both accuracy and efficiency. Generally speaking, the required memory is proportional to the number of the particles simulated. As millions of particles dynamically flow into the simulation domain, collide with each other, move to different positions, and flow out of the simulation domain, dynamic memory allocations and deallocations are required in order to achieve an optimal usage of the memory. On the other hand, much more complex data structures are necessary in order to

---

*Research Associate, Member AIAA. Email: dagao2008@gmail.com.
†Assistant Professor, Member AIAA. Email: schwartz@aem.umn.edu.
efficiently store and manage the information related to these millions of particles in the simulation. For example, the derived data structure in Fortran 90 can be used to package different elementary data structures such as doubles, integers, pointers, linked lists, and arrays together into one data structure representing a DSMC cell. An understanding of these complexities and their impacts on the optimal usage of modern multi-core architectures would be valuable not only for DSMC code development, but also for chip designers to provide better hardware support and design for real scientific computing applications in engineering.

Compared to distributed memory parallel computing which always employs the message passing interface (MPI),\(^7\) shared memory parallel computing has the potential to avoid communication overhead inside multi-core nodes that comprise larger clusters. As will be discussed in this article, typical nodes may contain multi-core processors plugged into separate sockets. For example, many nodes now contain two quad-core processors in separate sockets for a total of eight core-processors per node. Only a few years ago the standard was two dual-core processors in separate sockets for a total of four core-processors per node. The industry trend is not just to increase the number of core-processors per node, but to increase the number of core-processors per socket. Therefore, a combined shared/distributed memory parallelization may maximize the usage of the modern multi-core architecture computational resources, enhance the scalability, reduce the communication overheads, and reduce the number of domains compared to a pure distributed memory parallelization with MPI. For example, the combined shared/distributed memory parallelization for a cluster of nodes each containing eight cores could potentially require eight times fewer parallel domains than a pure distributed memory parallelization with MPI.

The present work is a first step towards such a hybrid shared/distributed memory parallelization of the DSMC method. The article focuses first on serial optimization followed by the shared memory parallel implementation (using OpenMP\(^8\)) of a 3-level Cartesian mesh DSMC code designed for 3D flows over complex geometries called the Molecular Gas Dynamic Simulator (MGDS) code.\(^5\) The purpose of the paper is to show how a state-of-the-art DSMC code employing complex data structures and dynamic memory allocation can be multi-threaded via OpenMP. The article is organized in the following manner. In Section II, optimization of the serial MGDS implementation is detailed. In Section III, the OpenMP parallel implementation is detailed, followed by the OpenMP parallel performance in Section IV. Finally conclusions are presented in Section V.

II. Serial Optimization

Prior to presenting the parallel optimization using OpenMP, the serial version of the MGDS DSMC code is first profiled and optimized. The MGDS code employs a 3-level adaptive Cartesian grid for the geometry model. An important aspect is that two distinct data structures are maintained. The Geometry data structure contains the 3-level Cartesian grid description including all information required to move a given particle through the grid. This Geometry data structure requires little memory storage and has the potential to be stored in its entirety on each partition of large parallel simulations. The detailed particle and cell data is stored in a separate Cell/Particle data structure that must be partitioned due to large memory storage requirements. The MGDS code employs an adaptive mesh refinement (AMR) technique and a cut-cell technique for imbedding arbitrary triangulated surface meshes into a simulation. The AMR and cut-cell subroutines are called only occasionally and therefore are not relevant to the parallel implementation of the core MGDS algorithm. Complete details of the data structures and algorithms employed by the MGDS code are found in Ref. 8 and will be referred to frequently in the present article.

The serial version of the MGDS code is profiled in order to determine which functions called within the code require the most computational time. The results should be representative of a general implementation of the DSMC method. Table 1 presents a summary of the initial profiling of the MGDS code. It shows the specific functions that require approximately 5\% or more of the total simulation time. The first two functions, particle indexing and ray-trace movement both deal with the movement of simulation particles. On a Cartesian mesh, a particle can simply be moved along its velocity vector for the full time step and afterwards its new coordinates can be efficiently indexed into the appropriate cell. This technique is depicted in Fig. 1(b). On the 3-level Cartesian mesh used in the MGDS code, three successive indexing calculations must be performed to locate the particle in the correct level-1 (L1), then level-2 (L2), and finally level-3 (L3) cell. Alternatively, the ray-trace function is called recursively to detect the time-to-hit each face in a cell by a given particle. This requires a dot product between the particle velocity vector and the cell-face normal vectors. The particle is then moved for the minimum computed time which corresponds to point at which it first crosses the cell boundary. This technique is depicted schematically in Fig. 1(a). The ray-trace function is then called recursively to move the particle for the remainder of the time step, possibly moving through multiple cells. Since particle indexing on an unstructured non-Cartesian mesh is computationally expensive, the ray-tracing technique is generally preferred on non-Cartesian meshes. However, since the MGDS code utilizes variable time steps within each
cell, the ray-tracing technique is actually used to move particles through the 3-level Cartesian mesh. For the purposes of profiling the relative costs of these two functions, both are used in producing the results in Table 1. Specifically, the ray-trace function is used to move particles to the intersection points on the cell face, however, the particle is then immediately re-indexed in order to locate it in the new cell. Thus every recursive ray-trace move is followed by a re-indexing of the particle. The sample sums function counts all particles, samples momentum and energy in each coordinate direction, and samples internal energies within each cell. The collision pairs function evaluates the local collision rate using the variable-hard-sphere (VHS) model and randomly selects appropriate pairs of particles for collision. The collision translational energy function performs elastic collisions between collision pairs with a random scattering angle. The main DSMC loop excludes the time spent in the other major functions and mainly includes copying particle data from the cell’s linked-lists into static arrays that the collision and sampling functions operate on. Finally, the global sort function is discussed in detail in upcoming sections.

Table 1. Profiling summary

<table>
<thead>
<tr>
<th>Functions</th>
<th>Percentage of the total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>particle indexing (via Cartesian geometry)</td>
<td>24</td>
</tr>
<tr>
<td>raytrace move</td>
<td>16</td>
</tr>
<tr>
<td>main DSMC loop</td>
<td>15</td>
</tr>
<tr>
<td>sample sums</td>
<td>7</td>
</tr>
<tr>
<td>coll pairs</td>
<td>5</td>
</tr>
<tr>
<td>global sort</td>
<td>4</td>
</tr>
<tr>
<td>coll trans</td>
<td>2</td>
</tr>
</tbody>
</table>

The profiling results listed in Table 1 correspond to a simulation of free-stream argon flow with collisions through a rectangular box 30 cm long by 10 cm by 10 cm. The domain has four symmetry surfaces and one inflow and outflow surface in the x-direction. The simulation is carried out at a temperature of 50K and a density of $1.0 \times 10^{-5} \text{ kg/m}^3$. The free stream velocity is 200 m/s in the x-direction and the resultant steady-state solution contains approximately 1 million simulation particles and the profiled portion involves the final 8000 sampling timesteps.

The main result seen in Table 1, is that indexing on a 3-level Cartesian mesh is considerably expensive. If particles were moved using ray-trace only and cell-connectivity was used to place particles in neighboring cells, then particle

![Figure 1. Particle movement algorithms in DSMC.](image)
indexing could be completely eliminated with significant computational savings. For this reason, the default movement procedure in the MGDS code is ray-trace movement for particles in all cells. Table 1 also reveals that the computational time spent in the collision functions is relatively low. However, it should be noted that the simulation used for the profiling involved only elastic collisions between particles of the same species. In more complex DSMC simulations where internal energy exchange and chemical reactions between multiple species are computed, the time spent in the collision functions will increase but typically remain below 50%.

Using the profiling information from Table 1, the MGDS code is further optimized, resulting in a speedup of approximately 2.46. The major aspects where efficiency was improved are discussed in detail in the following subsections and a summary is listed in Table 2.

Table 2. Optimization and speedups

<table>
<thead>
<tr>
<th>Optimization Techniques</th>
<th>Speedups Achieved</th>
</tr>
</thead>
<tbody>
<tr>
<td>cartesian vs ray-trace move</td>
<td>1.69</td>
</tr>
<tr>
<td>reduction of the indexing function</td>
<td>1.26</td>
</tr>
<tr>
<td>cache usage enhancement</td>
<td>1.07</td>
</tr>
<tr>
<td>global sort algorithm improvement</td>
<td>1.08</td>
</tr>
<tr>
<td>replacing double with float</td>
<td>1.81</td>
</tr>
</tbody>
</table>

A. Cartesian vs Ray-trace Move

Obviously, using the ray-trace function to recursively move a particle through the mesh is more computationally expensive than simply translating the particle for one complete time step. Indeed, the average time spent in the ray-trace function per particle was determined to be 13.3 times longer than that spent simply translating a particle for the test simulation considered. However, if indexing is included, the speedup of the overall simulation is much less. Specifically, the ratio of the total simulation times when using the ray-trace function plus re-indexing into neighbor cells, compared to when particles are simply translated (Cartesian move) and then re-indexed was determined to be 1.69. This is a result of the fact that particles are re-indexed after each ray-trace move. This may be more than once per timestep for many particles, whereas if simply translated, particles are only indexed once. Thus the re-indexing function is called less when using a simple Cartesian move, and the move itself is more efficient than a ray-trace move.

B. Reduction of Particle Indexing

As seen in Table 1, the subroutine to determine a particle’s L3 cell index is quite expensive since each three-dimensional coordinate must be indexed successively on each of the three Cartesian levels. Moreover, the subroutine is heavily called because once a particle is moved to a neighboring cell (via ray-trace), the index of the destination cell is determined. Since the mesh geometry remains fixed for extended periods of time, the cell-connectivity could be computed once and stored within each cell. The serial version of the MGDS code profiled above was modified to store the neighbor cell indexes as variables in each cell thereby reducing the call to the expensive indexing subroutine. Based on the profiling results of Table 1, the ideal speedup (with no indexing) for the test simulation would be 1.32. The actual speedup obtained with the modification was 1.26 (as listed in Table 2). The reason is that the 3-level Cartesian mesh used in the MGDS code has no cell-face alignment restrictions between L2 and L3 cells. Specifically, within each L2 cell the connectivity between L3 cells is strictly Cartesian, however, connectivity across L2 and L1 cells is not as trivial. Thus if a particle crosses either a L2 or L1 cell boundary, it is still re-indexed in the current MGDS implementation. Thus calls to the re-indexing function are significantly reduced, but not entirely eliminated. It is further noted that after adaptive mesh refinement (AMR), all particles are re-indexed since the dimensions of all cells may have changed. Details of the adaptive mesh refinement can be found in Ref.8

C. Cache Usage Enhancement

The caches are the small high speed memories that are placed between the processor and the main memory. An effective usage of the caches can in some cases dramatically improve code performance because accessing memory from the cache is much faster than accessing the main memory. The cache size (presently varies between 2-6MB) is much smaller than the main memory, therefore, it can only hold a subset of the main memory’s contents. As a result,
if the data is not stored in the cache when it is required for a computation, the data has to be accessed and loaded from the main memory, which is slow and the access is referred to as a “cache miss”. On the other hand, if the data is stored in the cache, then there is no need to access the main memory, which is fast and the cache access is a “cache hit”. The cache hit ratio is defined as the total number of cache hits divided by the total number of cache accesses. It is a measure of the cache performance. A higher cache hit ratio means a better cache performance, and usually leads to a faster execution of the code.

Complex data structures are utilized by the MGDS code in order to handle dynamic memory allocation as the number of particles and the number of cells (during AMR) change significantly during a DSMC simulation. Again, the reader is referred to Ref. 8 for complete details. In particular, all required cell data is stored in a derived data structure in Fortran 90. In addition to cell data, this structure contains two pointers to the head and tail of a doubly-linked list. Each link points to a derived particle data structure that contains the complete information for each particle. It is important to note that the cell data structure only contains the head and tail pointers of this list, however the actual particle data is scattered throughout memory and only “linked” via pointers. As a result, even when particles linked within the same cell are moved and collided, all computations will incur cache misses due to the random storage of these data in the main memory. To enhance the cache usage, a single additional data structure is added to the MGDS code, called a large “cache-cell” structure. Instead of holding head and tail pointers, this data structure contains pre-allocated large static arrays for particles and potential collision pairs, in addition to other required cell data. The arrays are made conservatively large (allocated memory for 10,000 particles currently). However, only one such data structure is ever allocated and this single large cache-cell structure fits within standard cache memory sizes. In this manner, before each DSMC cell is processed, all cell and particle data is first copied into the cache-cell structure. This ensures that all required cell and particle data is actually located in a single continuous block in main memory. This entire block is loaded into the cache memory (as a so-called “cache-page”), and all operations that occur within a single cell are performed using only memory stored in the cache. Since the core of any DSMC implementation revolves around repeated operations on all particles within one cell at a time, such cache usage enhancement could potentially result in large efficiency gains.

Comparing the total simulation time for the profiling test case when using the cache-cell structure versus without, reveals an overall simulation speedup of 1.07. Interestingly, preliminary results reveal the collision functions to have very substantial speedups (approximately 100 times faster when using a cache-cell structure). Conversely, the ray-trace function shows a barely noticeable speedup. The dramatic cache performance difference between the collision function and the ray-trace function could possibly be explained as follows. The collision functions require only information stored in the cache-cell and nothing else. Thus it is possible that all computations involving collisions within a cell are completely performed using cache-memory access only. Whereas, when a particle moves to an adjacent cell within the ray-trace algorithm, information from the new cell must be loaded from main memory. Advanced compiler profiling options capable of reporting the frequency of cache misses exist, and are likely a necessary tool for continued optimization and testing of this strategy.

D. Global Sort Algorithm Improvement

The separation of the Geometry data structure and the Cell/Particle data structure within the MGDS code\(^8\) enables the complete movement of all particles within each cell independently of all other cells in the simulation. During this process, only particles’ positions are updated and they remain linked in their original linked-lists. After all cells have been processed within the main DSMC loop, all particle positions have been updated including a variable representing their destination cell. What remains after the main DSMC loop is to re-link all particles whose destination cell does not correspond to the cell in which they are currently linked. The most basic implementation of this global sorting subroutine checks every particle in each cell to see if the particle should be placed in a different cell. This is inefficient since many particles remain inside the same cell during a given time step and more importantly, because the work required to determine the destination cell has already been completed within the ray-trace function. Ideally, the global sort function should only processes particles guaranteed to move (far fewer pointer operations would be involved). The theoretical speedup gain should be the ratio of the total number of particles in the cell to the number of moved particles in the cell. Generally speaking, about 20 to 25 percent of the particles in a cell move to a new cell during each DSMC timestep and therefore the theoretical speedup for the global sorting subroutine is about 4.0 to 5.0.

In order to be able to only sort the particles that are moved into different cells, two requirements must be met: (1) The number of particles in each cell that are to move out of the cell must be known prior to the global sort procedure; (2) The doubly linked list of the particles in each cell must be pre-sorted into two groups: particles that need to be sorted into different cells and particles that stay in the same cell. For the implementation, we added a variable iMoved in the cell data to keep track of each cell’s number of moved particles. We also put the particles that need to be sorted
to different cells at the head of the list while locating the particles that stay in the same cell at the tail. In this manner, particles are automatically “pre-sorted” into two groups. As a result, during the global sorting function, only the particles that are required to be sorted will be moved into the correct cells. For the profiling test simulation, a speedup is 1.08 for the overall simulation time was found, while a speedup of 1.15 was found for the global sort function itself. The low speedup is caused by the fact that for the profiling test simulation, most particles (≈100%) moved to different cells and therefore most particles in each cell need to be sorted. An additional simulation was performed where the simulation time step was lowered by a factor of 10. This caused fewer particles (now approximately 27% of all particles) to move to a new cell during each timestep. The speedup increased to 2.6 for the modified global sort function over the original basic sorting implementation. The speedup is not ideal (about 3.7 theoretically) since some additional computation has been added to the ray-trace function in order to pre-sort the particles within each linked-list.

E. Double Precision Variables

In addition, we compared the computation cost of using all double versus all float precision variables. Results show that with only float used, the profiling test simulation ran 1.81 times faster compared to a version where only double precision was used. This finding suggests limited use of double precision variables. However, when dealing with mesh sizing and particle movement on the order of the mean-free-path, double precision may be required to prevent round off error from impacting simulation accuracy for near-continuum flows.

III. OpenMP Parallel Optimization

The OpenMP (Open Multi-Processing) library is a parallel programming model for shared memory multi-core processors. It supports multithreaded programming through language (such as Fortran and C/C++) pragmas (compiler directives). With the OpenMP pragmas, the compilers can identify sections of the code that are intended to run in parallel. The execution of the OpenMP program is initiated and controlled by a single thread, referred to as the “master thread”. When the master thread encounters parallel portions, it will create additional threads, often referred to as “slave threads”. Together, these threads execute the program in parallel. As all threads complete the work in the parallel portion, slave threads disappear and the master thread continues any serial execution until the end of the program, as shown in Figure 2.

![Figure 2. Sketch of the runtime execution of the OpenMP parallel program](image)

A. General Parallel Implementation

A few core issues central to achieving high parallel performance are coverage, granularity, load balancing, locality, and synchronization. Coverage is the percentage of a program that can be made parallel. Granularity refers to how much computational work is required within each parallel region before communication among processors is required. Amdahl’s law,\(^{10}\) which dictates the maximum parallel efficiency attainable in terms of the coverage and granularity of a program, suggests that the DSMC method is very suitable for parallelization. In most DSMC implementations, the majority of computational time is spent within a large loop that processes all cells one at a time. In the MGDS code, each cell can be processed independently of all others\(^{8}\) and therefore has a high degree of coverage. Furthermore,
within each cell, the generation, movement, collision, and possible chemical reaction of particles requires significant computational expense. As advanced physical models are incorporated into DSMC, the computational time spent per cell will continue to increase and thus the granularity of the DSMC method is also quite high. Load balancing refers to how evenly balanced the work load is among different processors. This is certainly a challenge for the DSMC method since cells contain different numbers of particles, some of which react or collide with surfaces and some which don’t. Thus simply partitioning an equal number of cells to each processor may not achieve acceptable load balancing. Locality essentially means how efficiently the cache memory is used by the program. This certainly affects the serial performance of the program, however, as discussed in upcoming sections, may also affect the performance on shared-memory parallel architectures. Finally, synchronization refers to points in the program where all processors must halt execution and communicate information among different processors before resuming execution.

The goal for successful OpenMP implementation is to fully parallelize the main loop over all L3 cells and also fully parallelize the global sorting function which re-links all particles within their new destination cells each time step. The challenges in parallelizing these DSMC procedures using OpenMP all revolve around the complex data structures and dynamic memory allocation inherent in a state-of-the-art DSMC implementation. Specifically, dynamic memory allocations and deallocations during each DSMC time step, the doubly linked-list data structure containing particle information, and the use of pointers to reference both particle and cell data structures each require special handling for shared-memory (OpenMP) parallelization. For example, even if a pointer variable is initialized on each thread as a private variable, the memory to which the pointer references is still shared by all the threads. Without modification of the DSMC algorithm and data structure, this memory access would create a synchronization point since the memory location can only be accessed by one thread at a time. Such a synchronization point drastically reduces the parallel performance.

For the OpenMP implementation, an input file “OpenMP.in” allows the user to input the desired number of threads. Based on this number of threads, the work is partitioned either statically or dynamically among the threads using OpenMP library commands. Each thread has been identified with an added integer variable named iThread within the MGDS code. This thread id is then used to differentiate the data specific to each thread so that all threads can concurrently work together independent of one another. The variables in the parallel portion of the code can be declared as either shared variables or private variables. A private variable means each thread maintains a private copy of the variable for the duration of the parallel construct. Private variables are used to facilitate computations whose results are different for different threads. A shared variable means all threads in the parallel region share access to the variable. Sharing variables among threads makes interthread communication quite simple: threads send data to other threads by assigning values to shared variables and receive data by reading values from them. However, such shared memory is the major source for data dependences and errors because some threads may not be able to receive the latest updated value when multiple threads are writing to the same shared memory address.

B. Removal of Synchronization Points

A major challenge for the OpenMP implementation of the DSMC method is to coordinate the execution of multiple threads that operate on shared variables because such data races could easily lead to wrong results if one thread is reading a variable while another thread is modifying it. On the other hand, such coordination mechanisms, often referred to as synchronization in OpenMP, may force the execution of the code to be serial in nature. For example, without modification, the OpenMP library functions may automatically create a critical section that forces a memory location to be accessed one thread at a time. As a result, the parallel performance may be significantly reduced if even a single synchronization point exists in the OpenMP code. All synchronization points have been successfully removed from the main DSMC loop and particle sorting algorithms within the MGDS code through the following important modifications:

1. Shared memory access to standard data types such as double, float, and integer, is moderated through the reduction clause in the OpenMP library. For example, the total number of particles is a global variable that requires an update after each time step is complete. When multiple threads are processing and counting their own particles, any attempt to update the global number of particles variable would create a critical section synchronization point where only one thread could update this variable at a time. Instead, if such global variables are declared as “reduction” variables, OpenMP can remove the critical section and maintain efficient parallel performance. Essentially through the reduction clause, OpenMP automatically creates private copies of the shared variable for each thread and combines them together according to the operation specified in the reduction clause at the end of the parallel construct. The advantage of this OpenMP capability is little-to-no modification of source code is required to handle shared-memory access to such variables by multiple threads. The disadvantages are that this capability incurs some overhead (although far less than the creation of a synchronization point), and more importantly, that this capability does not exist for more
complex data types and data structures. Pseudocode of the reduction clause for the global number of particles variable is listed below:

```
$OMP Reduction (+::NumTotParticles) 
NumTotParticles = NumTotParticles + NewParticles
```

(2) Sometimes, a complex derived data structure is shared among different threads, which without source code modification, would induce a critical section and synchronization point. For such data structures, the above reduction clause in OpenMP is not applicable. The previously described cache cell data structure is such an example. In order for each thread to process its current cell and particles contained within, the program must load all data into the cache cell data structure. Clearly, each thread needs a separately allocated cache cell to work with if the loop is to be fully parallelized with no synchronization points. The solution is to allocate an array of cache cell data structures (the array size is equal to the number of threads) so that each thread can work on an element of the array (indexed by the variable iThread) without any data dependence for memory accessed by other threads. Thus, any operations that involve allocation, deallocation, accessing, or writing to a temporary data structure variable, require an array sized to the number of threads in order to completely remove all synchronization points. Since the number of threads is small compared to the total number of such data structures allocated (for example the total number of cells), the additional memory requirement is negligible.

(3) The global sort function does involve data dependency between cells and therefore modification must be made to remove associated synchronizations. Specifically, if the particles in different cells are to be sorted by different threads simultaneously, then multiple threads may attempt to add particles to the same linked-list at the same time. In order to remove all data dependencies, an additional array of linked-lists is added (allocated) within each cell data structure. Thus in addition to the single doubly linked-list of particles contained within each cell (see Ref.8), additional linked-lists are added to accept incoming particles from cells being processed by other threads. Again the size of the array of linked-lists is equal to the number of threads. In this manner, when linking particles to destination cells, each thread adds the particle to its own list within the destination cell data structure. That is, each thread only accesses and writes to its specific index in the array of linked-lists. Thus, particles within a given cell can be re-sorted into destination cells without accessing the memory used by other threads. This data independence enables the entire particle sorting function to be parallelized with no synchronization points. After all particles are re-linked to the correct incoming linked lists, a global linking of these lists can be fulfilled by simply reassigning the head and tail pointers of each list, thereby recovering the single list containing all current particles within each cell.

The successful removal of all critical sections and synchronizations is vital for parallel performance using OpenMP. The resulting threaded version of the MGDS code achieves good parallel scalability for both dual-core and quad-core systems, as discussed below.

IV. OpenMP Parallel Performance

We have tested several cases in both dual-core and quad-core multi-processor systems. Figure 3 shows the OpenMP parallel performance as a function of the number of threads for a test simulation. The same simulation conditions are employed to test parallel performance as listed in Section II for serial optimization. The only difference is that a smaller domain is simulated, resulting in approximately 50,000 total simulation particles. This is important since if threaded over multiple processors, each processor may now be simulating a relatively small number of particles. Thus the granularity may become small for parallel simulations with multiple threads. The dual-core system is the Intel Core2 Duo CPU E6750. The quad-core system is the Intel Xeon Quad Core CPU X5550. The ideal speedup as a function of the number of threads is also plotted with a solid red line in Figure 3 for the purpose of comparison. Several conclusions can be drawn: (1) Both dual-core and quad-core systems show a good speedup, 1.99 for the dual-core and 3.74 for the quad-core, respectively. (2) As the number of cores increases, the OpenMP performance slows down due to increasing overheads. (3) The quad-core system shows good scalability up to four threads.

Furthermore, the simulation on the two socket system was re-run using the “hyper-threading” capability on that architecture. Again each socket houses one Intel Xeon Quad Core CPU X5550, however, now each core-processor is “hyper-threaded” into two parallel processes, resulting in a total of 16 possible threads on this two-socket system. The speedup result as a function of the number of threads is plotted in Figure 4 for up to 12 threads. It is clearly seen that the scalability disappears after a maximum of 4 threads. The most likely reason for the reduced parallel scalability is the fetching of memory across sockets. Although all processors have fast memory access (compared to memory located on a separate node in a parallel cluster), memory access within the same socket is substantially faster.
Figure 3. OpenMP parallel performance for dual-core and quad-core processors

Figure 4. OpenMP parallel performance for multiple quad-cores in multiple sockets
Figure 5. Comparison of OpenMP parallel performance with and without memory fetching.

Figure 6. Conceptual sketch of the memory fetching across sockets.
than across sockets. In order to gain a better understanding of memory fetching and data communication phenomena across sockets, a conceptual sketch of both cases with and without memory fetching is presented in Figure 6. Figure 6 (a) shows two dual-cores that are housed in two sockets where the memory fetching and data communications across the sockets happens when four threads work together. On the other hand, Figure 6 (b) shows a quad-core in a single socket where there is no memory fetching and data communications across the sockets when four threads work with each other concurrently. It should be noted that when using a distributed memory parallelization approach (such as MPI), each processor is automatically associated with a partitioned portion of memory located on its own socket. However, for a shared-memory (OpenMP) implementation, if all core-processors on the node are to share memory, this memory may be physically located on either socket. To further confirm this hypothesis, the same simulation was tested with 2 dual-core processors in 2 sockets and then compared to a test using one quad-core in one socket. Thus both simulations employ four threads, but involve processors on separate sockets. Figure 5 shows that the speedup is drastically reduced with memory fetching across sockets compared with the case in which all memory is accessed within the same socket.

The above results are for a specific simulation designed with a well balanced load distribution among threads for the loop over L3 cells. For practical DSMC simulations, the load distribution among threads may not be well balanced. As a result, an efficient load balancing mechanism is necessary for an OpenMP program. Fortunately, OpenMP provides two major scheduling methods for threading loops; static scheduling and dynamic scheduling. In a static schedule, the assignment of the iterations to each thread is set at the beginning of the loop and can not vary from one loop execution to another. On the other hand, in a dynamic schedule, not all portions of the loop are assigned to threads at the start of the loop. Instead, each thread requests the next portion (single or multiple iterations) after it has completed the work already assigned to it. Thus the assignment of the iterations to each thread can vary at runtime from one execution to another in a dynamic schedule. We have compared the effects of these two scheduling methods on the MGDS DSMC code and listed the results in Table 3. In the table, N is the total number of the L3 cells contained in the main DSMC loop (the number of iterations). P is the number of processors (threads), which is 2 for a dual-core processor in this case. We have simulated the test case using various “chunk sizes” set as N/P. The chunk size of N/P is well load balanced, while the chunk size of N/2P is not as balanced. Several conclusions can be drawn from the results: (1) For a well load balanced case, static scheduling is better than dynamic scheduling due to less overhead in the static schedule OpenMP procedure. (2) For simulations that are not inherently load balanced (such as the DSMC method), dynamic scheduling is better than static scheduling because the benefits gained through dynamic load balancing outweigh the extra overhead associated with the dynamic scheduling procedure in OpenMP. In large scale, practical DSMC simulations, it is likely impossible to specify a load-balanced static schedule prior to looping over all cells. Some reasons are listed below: (1) Some cells are inflow cells and require extra calculations compared to interior cells. (2) The distribution of the number of particles among various cells is not uniform, which means some cells will have more particles than other cells. Moreover, the distribution of the number of particles dynamically evolves. (3) The computational time associated with the particle moves, collisions, surface collisions, and possible chemical reactions could vary greatly in each cell and may not remain constant even within the same cell during the course of a simulation. This detailed information would be required before the start of the main DSMC loop in order to properly load balance using static scheduling. Thus dynamic load balancing in OpenMP is thought to be necessary for complex DSMC simulations.

<table>
<thead>
<tr>
<th>“chunk size”</th>
<th>static (speedup)</th>
<th>dynamic (speedup)</th>
</tr>
</thead>
<tbody>
<tr>
<td>N/P</td>
<td>1.99</td>
<td>1.98</td>
</tr>
<tr>
<td>N/2P</td>
<td>1.96</td>
<td>1.80</td>
</tr>
<tr>
<td>N/4P</td>
<td>1.70</td>
<td>1.87</td>
</tr>
</tbody>
</table>

V. Conclusions

To the authors’ knowledge, the present paper is the first effort to explore shared-memory parallelization on multi-core systems of a state-of-the-art DSMC code complete with dynamic memory allocation and complex data structures. Prior to parallel optimization, several serial optimization techniques are implemented and associated gains in efficiency
reported. Specifically, indexing a particle’s position on a 3-level Cartesian grid is found to be quite computationally demanding compared to using a ray-tracing technique together with known cell-connectivity to move particles through a multi-level Cartesian mesh. Furthermore, preliminary results indicate that if all required memory to process a single cell is located in a continuous block of memory (data locality) that is able to fit within cache memory, certain functions may run significantly faster. In particular, the collision functions within the DSMC loop may run entirely using cache memory only. However, the speedup for other functions such as particle movement, which inherently require information from multiple cells, is almost negligible. Further research using advanced compiler profiling tools for cache usage may enable modification of DSMC functions that rely solely on local data (within the cache) and result in considerable efficiency improvements.

A shared-memory (OpenMP) implementation of the DSMC method are also described in detail. All data dependencies and synchronization points within the main DSMC loop over cells and the global sort algorithm are successfully removed. The reduction command is used in OpenMP for standard integer, double, and float variables to remove data dependency, however this technique is not applicable for more complex data structures. In order for multiple threads to work with more complex data structures, arrays of these data structures must be allocated, where the size of the array is the number of parallel threads. This allows each thread to work on its own temporary data structure without accessing or writing to memory used by other threads. Finally, in order to thread the particle sorting algorithm, an array of incoming particle lists must be maintained (instead of just a single particle list). Again, the length of this array is equal to the number of threads. In this manner, multiple threads can add particles to the same cell data structure without attempting to access the same memory location, that is, each thread only accesses its designated element within the array. In addition, the advantages of dynamic scheduling in OpenMP for non load-balanced simulations over static scheduling is highlighted.

Overall, parallel speedups of 1.99 for the dual-core and 3.74 for the quad-core, respectively, are demonstrated for the OpenMP implementation of the MGDS DSMC code. Scalability is dramatically reduced when processors are located on different sockets within a given node. Memory fetching and data communication across sockets in a multi-socket system is shown to lead to a speedup factor of just over 4.0 when 12 processors are utilized that are located on two separate sockets. Nevertheless, the OpenMP implementation research presented in this article provides a foundation for future shared-memory parallelization techniques for DSMC and possible hybrid OpenMP/MPI implementations able to take full advantage of modern multi-core architectures.

Acknowledgments

This work is partially supported through a seed-grant from the University of Minnesota Supercomputing Institute (MSI). This work is also supported by the Air Force Office of Scientific Research (AFOSR) under Grant No. FA9550-04-1-0341. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the AFOSR or the U.S. Government.

References