Direct Simulation Monte Carlo Analysis of Rarefied Flows on Parallel Processors

Richard G. Wilmoth*
NASA Langley Research Center, Hampton, Virginia 23665

A method for executing a direct simulation Monte Carlo (DSMC) analysis using parallel processing is described. The method is based on using domain decomposition to distribute the work load among multiple processors, and the DSMC analysis is performed completely in parallel. Message passing is used to transfer molecules between processors and to provide the synchronization necessary for the correct physical simulation. Benchmark problems are described for testing the method, and results are presented that demonstrate the performance on two commercially available multicomputers. The results show that reasonable parallel speedup and efficiency can be obtained if the problem is sized properly to the number of processors. It is projected that, with a massively parallel system, performance exceeding that of current supercomputers is possible.

Introduction

THE direct simulation Monte Carlo (DSMC) method of Bird[1] has achieved acceptance in recent years as a viable means for analyzing complex flowfields in rarefied and transitional flow regimes. The main reason for this popularity is the method's ability to analyze multispecies, real-gas, reacting flows under conditions where neither continuum (e.g., Navier-Stokes-based solutions) nor free-molecular analyses are applicable. The method has been applied to a wide variety of problems, including hypersonic, blunt-body re-entry and high-altitude rocket plume flows. The general validity of DSMC analysis has been demonstrated in a number of cases by comparisons with experimental data and, in the continuum and free-molecular limits, by comparisons with other analysis methods.

Despite the success of DSMC for modeling physically complex flow problems, the method has not received more general use because of the large amounts of computing time required. A typical, one-dimensional (1-D) or two-dimensional (2-D) problem requires at least a day of processing on a dedicated minicomputer or microcomputer, and computing times of 20 or more Cray-2 hours are often required for even a relatively simple three-dimensional problem. For blunt-body re-entry problems at near-continuum conditions, the computing time scales approximately as the inverse of the mean free path and, therefore, becomes prohibitive at low altitudes.

Recently, more attention has focused on ways of reducing these computing times, including development of new algorithms that take advantage of current supercomputer architectures.[2] An approach currently being pursued by a number of researchers is the application of parallel processing.[3,4] The DSMC algorithm of Ref. 1 is readily amenable to parallel processing, and with the improvements now becoming available both in parallel computer hardware and software, there is significant potential for obtaining dramatic improvements in speed.

The purpose of the present paper is to examine the problems associated with performing DSMC analysis in a parallel computing environment and to provide some benchmark results that may be useful in assessing potential performance based on currently available parallel computing systems. A method for dividing the analysis into tasks that can be shared among multiple processors will be outlined. Two benchmark problems will be described, and results obtained by running these benchmarks on a variety of parallel and nonparallel computers will be presented. These benchmarks represent actual DSMC analyses of 1-D and 2-D problems that have been tested in sequential and parallel modes on machines ranging from personal computers to supercomputers (e.g., Cray 2). A discussion of the merits of currently available parallel systems will be presented in terms of both hardware and software (e.g., compilers).

Conventional Direct Simulation Monte Carlo Method

The DSMC method of Ref. 1 has been described extensively in the literature.[1,5] The method consists of simulating a gas by thousands of molecules whose positions, velocities, and internal states are stored and modified with time as the molecules move, collide, and undergo boundary interactions in simulated physical space. A time step is chosen such that the movement and collision processes can be assumed to be uncoupled over the duration of the step. The calculations are started from an initial state such as a vacuum or uniform equilibrium flow. The simulation time is identified with physical time in the real flow, and all calculations are treated as unsteady. When boundary conditions appropriate for steady flow are imposed, the solution is the asymptotic limit of the unsteady flow. A physical space computational cell network is required for the selection of collision pairs and the sampling of flow properties.

A simplified flowchart showing the major elements in the conventional sequential algorithm is given in Fig. 1. After the initial state of the gas is set, the method repeatedly cycles through the following steps: 1) movement of individual molecules, 2) indexing (or sorting) the molecules to determine their cell location, 3) selection of collision pairs and calculation of postcollision properties, and 4) sampling of molecules within cells to determine average macroscopic properties. The method is essentially a computer simulation of the molecular gas motion and relies heavily on pseudorandom number sequences for simulating the statistical nature of the underlying physical processes. The details of the algorithm are such that data variables are typically accessed from com-

Presented as Paper 89-1666 at the AIAA 24th Thermophysics Conference, Buffalo, NY, June 12–14, 1989; received Sept. 28, 1989; revision received April 18, 1990; accepted for publication Aug. 4, 1990.

*Aerospace Engineer, Aerothermodynamics Branch, Space Systems Division. Member AIAA.

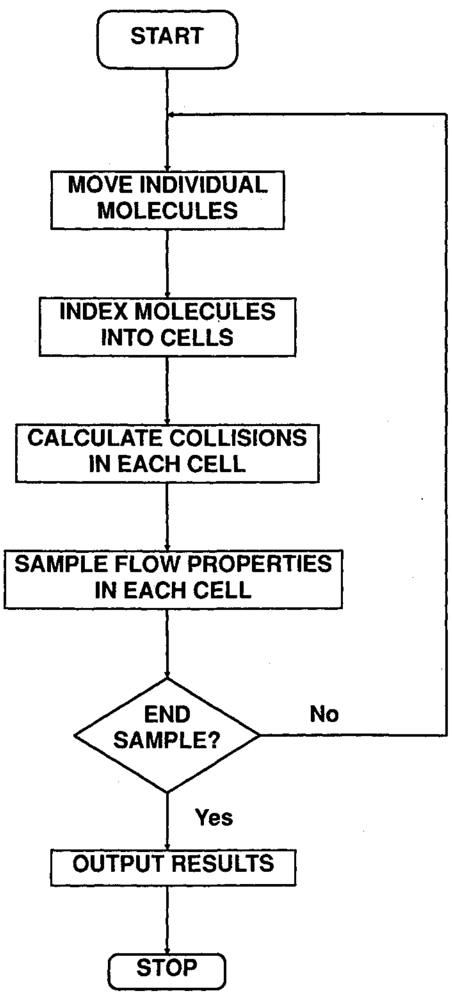
START
MOVE INDIVIDUAL MOLECULES
INDEX MOLECULES INTO CELLS
CALCULATE COLLISIONS IN EACH CELL
SAMPLE FLOW PROPERTIES IN EACH CELL
END SAMPLE?
No
Yes
OUTPUT RESULTS
STOP

Fig. 1 Conventional DSMC flowchart.

puter memory in a random sequence. Therefore, the method as formulated by Bird is not easily amenable to the vectorization or pipelining features available on modern supercomputers such as the Cray 2. However, since the motion of an individual molecule is considered to be independent of all other molecules during the movement phase and the collisions of molecules within a cell are considered to be independent of the collisions in other cells during the collision phase, there is a great deal of "natural" parallelism in the algorithm.

Parallel Direct Simulation Monte Carlo Method

Review of Current Parallel Hardware

Before describing the current parallel DSMC approach, some of the currently available types of parallel computing systems will be reviewed. A number of features are used to classify parallel computer systems, and each of these features has a direct impact on the types of algorithms that can be used effectively. Most systems can be classified as either single-instruction–multiple-data (SIMD) or multiple-instruction–multiple-data (MIMD) machines. Vectorization can be thought of as a special subsystem of the SIMD type that uses parallel data access techniques to provide a stream of data on which a single operation can be performed. MIMD-type machines consist of multiple processors that can simultaneously perform a set of multiple instructions on the same or different data sets. To resolve memory conflicts on MIMD machines, each processor accesses either global memory that is shared between all processors or local memory that is private to that processor. In the shared memory approach, memory access is managed by a combination of special system software and special hardware that is designed to avoid conflicts and to provide data synchronization. Therefore, direct communication between processors is not generally required. In the local memory approach, each processor can access directly only the data residing in its local memory. Therefore, explicit communications between processors (either direct or indirect) are required and usually must be specified by the applications programmer.

A final category that should be considered is the problem granularity allowed by a particular system. Problem granularity refers to the size or number of operations that can be performed independently before either data must be exchanged between processors or some form of synchronization must occur. In "fine-grain" parallelism, only a few instructions are executed simultaneously in parallel, whereas in "coarse-grain" parallelism, the number of instructions may constitute a complete subroutine or even a complete program module. As will be shown later in this paper, the problem granularity directly affects the parallel performance and generally must be optimized for each problem on the particular computer system involved. In SIMD machines, granularity is fundamentally limited to a single machine instruction, although in principle, streams of instructions can be constructed. Therefore, SIMD machines appear to be best suited to fine-grain parallelism. In local memory MIMD machines, the maximum "grain size" is limited only by the local memory size, and problem granularity can be controlled more readily by the applications programmer through algorithm design and programming techniques.

An analysis of Bird's DSMC method shows that each of the preceding features has certain advantages for different parts of the problem. For example, improvements in performance have been reported by Usami et al.[6] using moderate changes to the basic DSMC algorithm to allow vectorization of the movement portion and certain other elements of the program. McDonald and Baganoff[2] have reported a highly simplified movement and collision algorithm that allows for extensive vectorization. Furlani and Lordi[3] and Goldstein et al.[4] reported significant reductions in computing times using parallel processing on local memory hypercubes with relatively coarse-grained parallelism. In this paper, the focus is on the latter approach. Coarse-grained parallelism on local memory MIMD computers seems to be better suited for "retrofitting" to existing DSMC computer programs and allows changes in the physical simulation model to be more readily incorporated.

Current Approach

The parallel DSMC approach selected for this study is based on using domain decomposition to distribute the physical computational space over multiple processors and using message passing to communicate information between the processors. The actual implementation was written in the C programming language and uses software developed by Seitz et al.,[7] in which a host program operating under a runtime system called the "Cosmic Environment" on a UNIX host executes all nonparallel tasks and provides overall control of the application program. All parallel tasks are performed on multiple processors (called nodes) under a node operating system called the "Reactive Kernel." Message-passing facilities are provided for both host-node and node-node communications. The Cosmic Environment/Reactive Kernel (CE/RK) system has been implemented on a number of multicomputers, including the Intel iPSC/2 and Symult Systems Series 2010. The CE system also includes a feature called "ghost cubes," which allows a collection of network-connected UNIX hosts (or even a single UNIX workstation) to fully simulate the multicomputer.[7] This feature allows one to develop and test software under the CE environment before actually running it on the multicomputer.

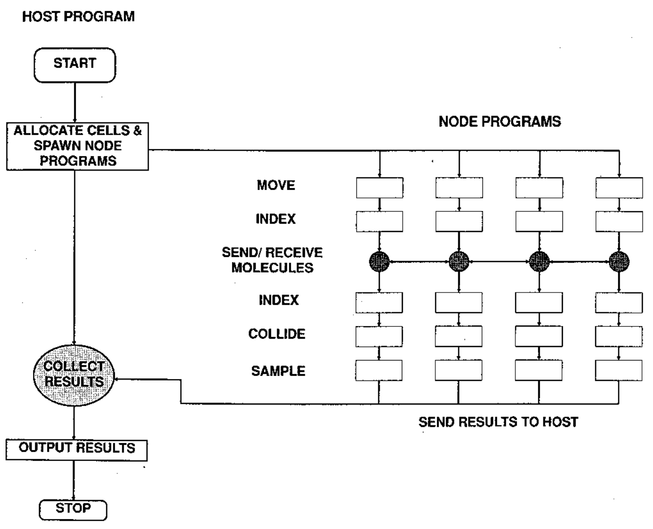

Fig. 2 Parallel DSMC flowchart.

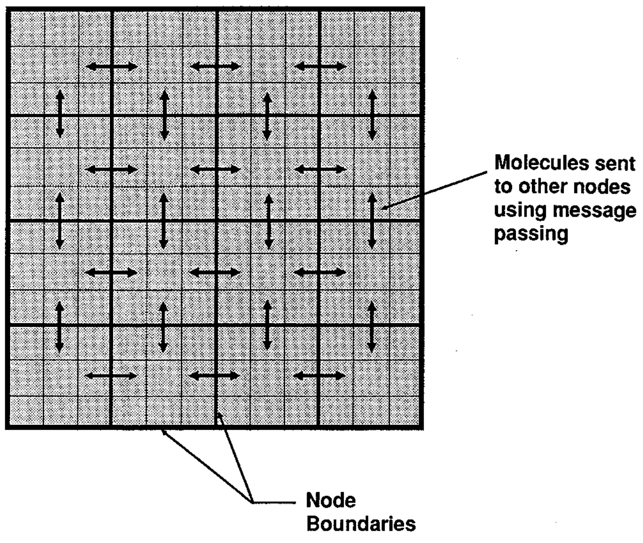

Fig. 3 Typical domain decomposition for parallel DSMC calculation.

The implementation of a parallel DSMC approach using domain decomposition is quite straightforward on the CE/RK system. Figure 2 shows a flowchart of the algorithm used. The host program performs all initialization tasks, allocates the computational space among nodes, spawns the node programs, and sends the initial data to the node programs. Each node program looks very much like the original sequential DSMC program, with the exception of the message passing routines used to send and receive molecules between nodes and an additional indexing step.

The node program works as follows:

1) Movement of molecules is performed as in the sequential method, except that only those molecules residing in the subdomain for a given node must be moved by that node.

2) Molecules are indexed as before.

3) All molecules destined for another node are collected in a message buffer, and that buffer is sent as a single message to the appropriate node. (Each node contains a list of the domain boundaries for all nodes so that it knows where to send the message.) If no molecules are destined for a particular node, the message buffer is still sent but contains a zero counter indicating that the remainder of the buffer is empty. Each node then waits to receive a message from each of the other $(N - 1)$ nodes. This provides the synchronization necessary to provide the proper physical simulation.

4) Molecules must then be indexed again in each node, since new molecules are not sorted into cells on arrival. In the present program, all molecules are simply reindexed. A more efficient approach might be to use some type of insertion sort so that arriving molecules are indexed to the proper cell as they arrive.

5) Collisions are calculated as before.

6) Sampling is performed as before.

Although not shown in Fig. 2, steps 1–6 are repeated for the desired number of time steps as in the sequential algorithm. Upon completion of all time steps, the node programs send all samples back to the host program, which calculates and outputs the desired macroscopic properties.

In the present study, no special attention has been given to balancing the work load among the processors. A simple domain decomposition technique is used to distribute evenly the physical domain among the nodes. This approach allows the domain boundaries to be defined as part of the initialization by the host program and to remain fixed throughout the DSMC simulation. A typical domain decomposition for a square domain with evenly spaced computational cells is shown in Fig. 3. Although this approach was found to be satisfactory for the simple benchmark problems studied so far, a more elaborate method will be needed for efficient solutions of problems of a more general nature. For example, in simulations of typical blunt-body entry flows where the number density is highly nonuniform, some sort of adaptive procedure will be required to adjust both the cell sizes and the domain boundaries as the solution evolves with time. Additional discussion of the effects of load balancing on the present work will be presented subsequently.

Definition of Performance Parameters

Two parameters commonly used to measure the performance of parallel computer systems are *speedup* and *efficiency*. Speedup is defined as the ratio of the time required to run a particular application on one processor to that required to run on N processors, i.e.,

$$\text{Speedup} \equiv t_1/t_N \tag{1}$$

Efficiency is then defined as

$$\text{Efficiency} \equiv \text{Speedup}/N \tag{2}$$

and is just the fraction of the maximum possible speedup obtainable with N processors. A simple relation that is often used to determine the maximum possible performance of a particular algorithm is Amdahl's equation, given by

$$t_N = \frac{pt_1}{N} + (1 - p)t_1 \tag{3}$$

where p is the fraction of the calculation that is performed in parallel. Using this relation, the speedup is given by

$$\text{Speedup} = N/[p + (1 - p)N] \tag{4}$$

and the efficiency by

$$\text{Efficiency} = 1/[p + (1 - p)N] \tag{5}$$

These relations show the limitations imposed when the calculations are not performed completely in parallel, i.e., $p < 1$. It can be shown that the speedup approaches an asymptotic limit of $1/(1 - p)$ and the efficiency approaches zero as the number of processors, N, approaches infinity. For example, with 99% parallelism, the maximum possible speedup is 100.

From the preceding discussion, it is obvious that parallel algorithms in which $p = 1$ are the most desirable so that no inherent limit is placed on the speedup by the algorithm itself. The DSMC method presented in the previous section is designed to achieve 100% parallelism (except for initialization and final output). However, in practice, there are two other factors that limit the performance of a given algorithm—*load balancing* and *communications*. Load balancing refers to the even distribution of the computational time among the processors, and perfect load balancing is approached as

$$t_1 \bmod N \rightarrow 0 \tag{6}$$

Another way of stating the effect of load balancing is to say that the minimum computational time, t_N, on N processors can be no smaller than the largest time, $t_{\max}$, spent on any one of the N processors. Thus, perfect load balancing is also approached as

$$(t_{\max}/t_{\text{avg}}) - 1 \rightarrow 0 \tag{7}$$

where t_{avg} is the average time spent over all processors. Therefore, the quantity $t_{\max}/t_{\text{avg}}$ provides a measure of the degree of load balancing.

Communications are an important consideration in the design of parallel algorithms, particularly in message-passing systems typically used on hypercubes. Even for 100% parallelism and perfect load balancing, the total time to complete a given problem must be augmented by the time to perform the necessary communications between processors (and host processes if necessary). Thus, for $p = 1$,

$$t_N = (t_1/N) + t_{\text{communications}} \tag{8}$$

and communications ultimately limit the maximum performance that can be achieved. The communication time depends on the actual hardware bandwidth, the message-passing software, and the message algorithm used in the application program. For the CE/RK system, a general expression for the communication time required for a single message is given by Seitz et al.,[7]

$$t_{\text{message}} = t_0 + aL + bD \tag{9}$$

where t_0 is called the *latency time* or the time required to send a 0-byte message between nearest neighbor nodes, L is the length of the message (in bytes), D is the distance (in nodes) over which the message must be sent, and a and b are constants. The total communications time is then

$$t_{\text{communications}} = \text{no. of messages} \times t_{\text{message}} \tag{10}$$

For typical systems, t_0 is on the order of a few hundred microseconds and generally is the dominant factor in Eq. (9). Since the single message time is controlled by the system software and communications hardware, it is important that the application program minimize the number of messages sent. For the parallel DSMC algorithm used in the current study, the total number of messages sent per time step is $N(N - 1)$. However, since messages are effectively sent in parallel, the actual communications time needed in Eq. (8) is

$$t_{\text{communications}} = (N - 1) \times t_{\text{message}} \tag{11}$$

The ratio $t_1/t_{\text{communications}}$ is a measure of the problem granularity and will be discussed later.

Benchmark Problems

Two benchmark problems, illustrated in Fig. 4, were selected for testing the parallel DSMC method. The first is the so-called 1-D Rayleigh problem described by Bird.[5] It consists of a gas initially at rest over an infinite flat plate. At time $t = 0$ the plate is given a velocity U_w and a temperature T_w. (All physical quantities in this paper are nondimensionalized by the appropriate initial gas quantities, i.e., velocities by the

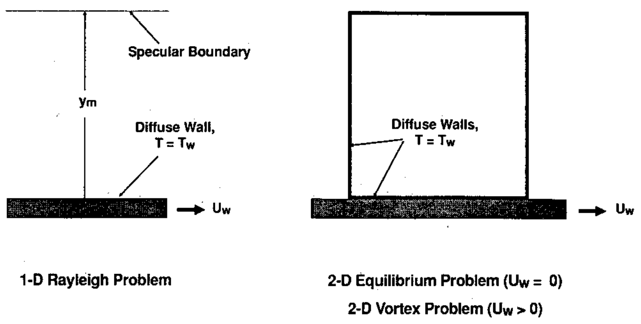

Fig. 4 Sketch of DSMC benchmark problems.

most probable speed, temperatures by the initial gas temperature, physical time by the mean time between collisions, and distances by the mean free path in the undisturbed gas.) This problem was selected because an actual DSMC computer program for solving this problem has been given by Bird[5] and because it has been used as a sequential benchmark on a variety of other computers by the author.

The second benchmark problem is actually a 2-D variation on the Rayleigh problem consisting of a square box whose wall temperature can be set at time $t = 0$ to something different than the initial undisturbed gas and which has a lower wall that can be given a nonzero velocity (similar to the Rayleigh problem). With a nonzero lower wall velocity, a vortex should develop in the box. However, by setting the wall temperature to 1 and the lower wall velocity to 0, the problem simply represents a gas in equilibrium whose macroscopic quantities should remain constant with time. This problem has a number of desirable properties for benchmarking a parallel DSMC method. First, the number of molecules should, on average, remain constant within each cell. Second, the number of collisions in each cell should, on average, remain constant (excluding wall interactions, which, of course, will be more frequent for those cells adjacent to walls). This means that load balancing should be reasonably good (at least on average). Finally, by varying the wall temperature and lower wall velocity, it will be shown that granularity and load balancing can be altered, and those effects can be studied in a systematic manner.

Flowfield Results

One-Dimensional Rayleigh Problem

The 1-D Rayleigh problem was run for the conditions $T_w = 1.6$ and $U_w = 2.0$. The same total number of time steps were used for all runs, and ensemble-averaged results were obtained from the sampled data at several intermediate time values. A comparison of the U-velocity distributions for a sequential calculation and a parallel calculation is shown in Fig. 5 for three time values. The parallel case was run using a 4-node ghost cube on a single Sun workstation. The parallel results are in good agreement with the sequential results (which are also in good agreement with results for those same conditions given by Bird[5]).

Two-Dimensional Equilibrium Problem

Since the mean properties are constant for the 2-D equilibrium problem ($U_w = 0$), the most interesting check of the parallel method is obtained by verifying the collision frequency as a function of temperature. For the hard-sphere collision model used in these calculations, the collision frequency should be proportional to the square root of the temperature. By setting the wall temperature to a value higher than the initial gas temperature, the gas will be heated by the wall and should equilibrate to the wall temperature after sufficient collisions. This provides at least a rough check that the physical simulation is correct in the parallel method, since a faulty communications algorithm can yield a collision frequency that differs from $\sqrt{T_w}$. For example, it can be shown

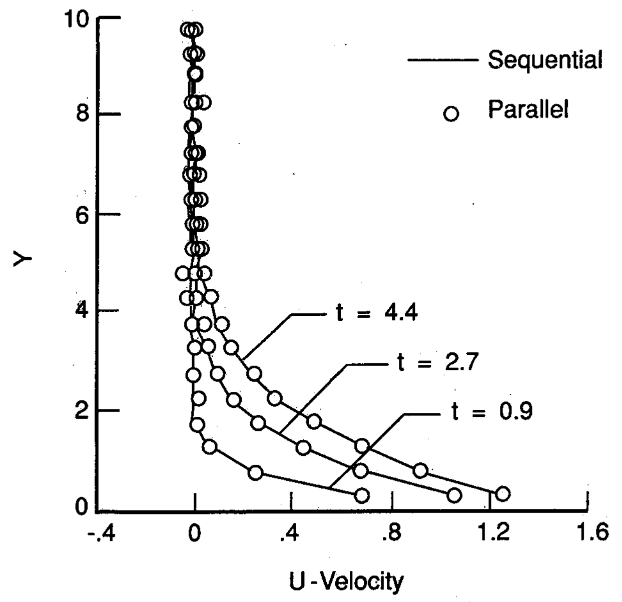

Fig. 5 Comparison of sequential and parallel DSMC results (1-D Rayleigh problems).

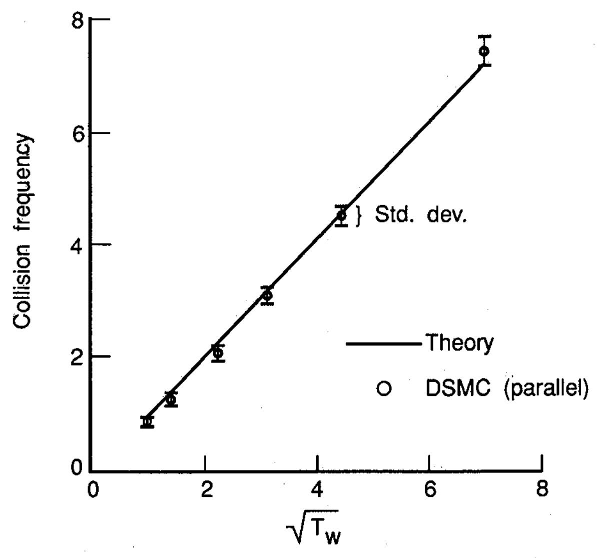

Fig. 6 Comparison of predicted collision frequency with theory for hard-sphere collision model (2-D equilibrium problem).

for the domain decomposition algorithm used that if *no* molecules were allowed to transfer between nodes, the average collision frequency within the complete box would be given by

$$f_{\text{collision}} = [4(\sqrt{N} - 1)\sqrt{T_w} + [N - 4(\sqrt{N} - 1)]\sqrt{T_0}]/N \tag{12}$$

for $N = 4$, 16, 64, etc., where N is the number of processors and T_0 the initial temperature of the gas (always equal to 1 in current nomenclature). Thus, for $N \geq 16$, there would be interior nodes containing molecules that would never *see* the walls and would remain at the initial temperature, T_0.

Figure 6 shows a comparison of the collision frequency predicted by the parallel method with that for an ideal hard-sphere collision model. The parallel calculation was performed on an Intel iPSC/2 using 16 nodes, and the agreement with theory is very good.

Two-Dimensional Vortex Problem

By making the lower wall velocity nonzero in the previous problem, a vortex flow can be generated in the box. A sample set of velocity vectors for $U_w = 2.0$ is shown in Fig. 7. Although no data are available for comparison, the qualitative appearance of the vortex seems reasonable. Other values of wall velocity were run, and the results are used later in this paper to demonstrate the effects of nonuniform load balancing among the processors. Those results show that the location of the center of the vortex depends on the magnitude of the lower wall velocity, as expected.

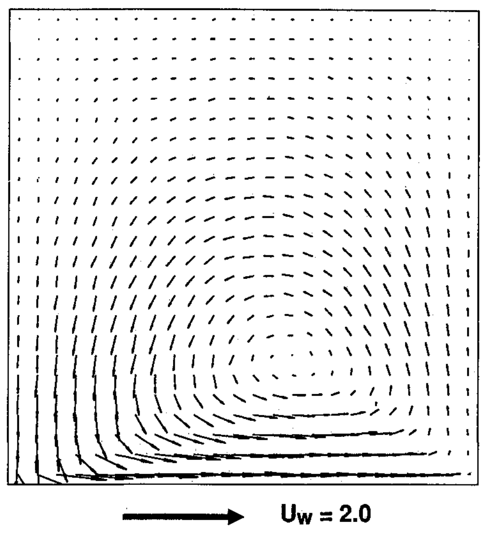

Fig. 7 Velocity vectors for 2-D vortex problem: U_w = 2.0.

Benchmark Performance Results

Sequential Results

The computational times for the sequential versions of the 1-D Rayleigh problem and the 2-D equilibrium problem on various computers are given in Tables 1 and 2, respectively. These results are not intended to measure the absolute relative performance of the various machines but are shown mainly to provide a reference for comparison with the parallel results. However, it is important to note that the results depend not only on the computer hardware but also on the compiler and operating system. Most of the systems shown used some variation of a UNIX-like operating system, and the compilers were used with the maximum level of optimization that would

Table 1 Sequential results—1-D Rayleigh problem

	Time, s	
Computer	Fortran	C
Cray YMP	19	42
Cray 2S	22	61
Cray 2	27	75
Cyber 205 (VSOS)	41	—
Convex C210	59	147
DECStation 3100	85	93
Cyber 860 (NOS)	95	—
Sun SPARCstation 1	116	187
Gould NP1	154	329
Sun 4/280	182	318
HP 9000/800	212	334
Sun 3/260 w/FPA	256	440
MicroWay Monoputer (MS-DOS*)	379	—
Sun 3/160 w/FPA	390	670
Cyber 173 (NOS)	472	—
VAX 11/785 (VMS)	556	645
Definicon DSI-20 (MS-DOS*)	598	—
MicroVAX II (VMS)	698	—
Sun 3/260 w/68881	845	1008
VAX 11/780	1309	1505
IBM PC/AT w/80287 (MS-DOS)	3960	—

Table 2 Sequential results—2-D equilibrium problem

Computer	Time, s	
	Fortran	C
Cray YMP	42	90
Cray 2S	52	147
Cray 2	65	161
Convex C210	133	407
DECStation 3100	180	231
Sun SPARCstation 1	267	411
Sun 4/280	403	680
Sun 3/260 w/FPA	657	1267
Sun 3/260 w/68881	2060	—

produce correct results for the system involved. (Two of the systems, the MicroWay Monoputer and the Definicon DSI-20, actually consisted of separate processor boards mounted in an IBM personal computer, and the host machine operating system was used only to transfer the programs into the memory of the auxiliary board.) As expected, the C version gave higher execution times than the Fortran version mainly due to the more extensive use of double-precision arithmetic in the C language. It is also important to note that none of the systems tested required any significant changes to the programs other than a few system-dependent calls, such as timing routines. All programs also used the same program-supplied random number generator rather than any random number routine supplied as part of the particular system libraries.

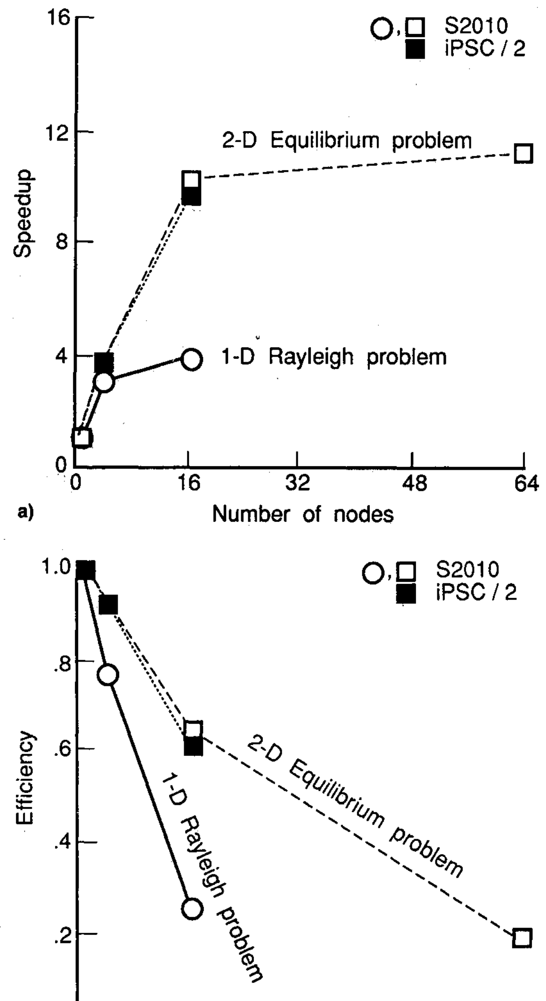

Fig. 8 Performance parameters of two benchmark problems: a) speedup; b) efficiency.

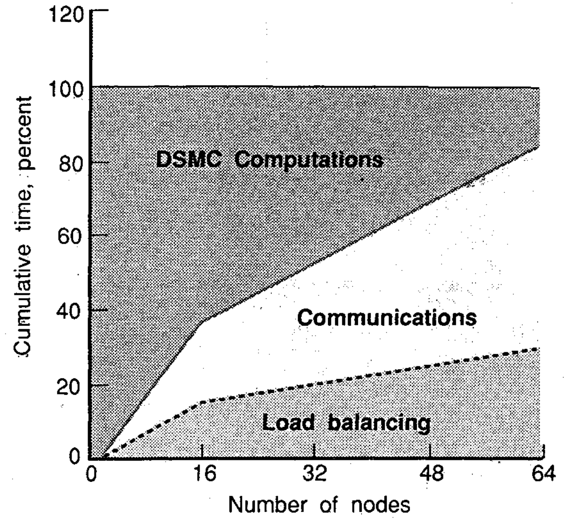

Fig. 9 Distribution of CPU time (2-D equilibrium problem).

Finally, for later comparison with the parallel results, it should be noted that most of the parallel systems tested in this study have individual processors with speeds roughly comparable to the slower of some of the workstation-class machines shown in the tables (e.g., Sun 3/260 w/68881).

Parallel Results

The parallel DSMC program was tested to an Intel iPSC/2 and a Symult Systems S2010 for both the 1-D Rayleigh problem and the 2-D equilibrium problem. These systems have similar characteristics, and both were tested at the Caltech Concurrent Computing Laboratory under the CE/RK system. The results presented subsequently are not intended to demonstrate the relative merits of these computers but are mainly intended to demonstrate the performance of the DSMC algorithm. Therefore, the timing measurements were made in such a way as to include the overhead associated with the application software itself as well as that associated with the CE/RK software and the communications hardware for each system.

Speedup and Efficiency

The speedup and efficiency measured for the 1-D Rayleigh problem and the 2-D equilibrium problem are shown in Fig. 8 as a function of the number of nodes (processors). Both problems show an increase in speedup with increasing number of nodes, but the 1-D results begin to level off beyond about 4 nodes while the 2-D results begin to level off beyond about 16 nodes. The maximum speedup obtained for the 1-D problem was about 3.9, whereas the maximum speedup obtained for the 2-D problem was about 11.3. Although the efficiency was less than 25% for the maximum number of nodes tested in both cases, the efficiency was better than 75% for the 1-D problem using 4 nodes and better than 60% for the 2-D problem using 16 nodes with only marginal decreases in speedup. The results demonstrate the importance of "sizing" the problem to fit the number of nodes.

Figure 9 shows the distribution of central processor unit (CPU) time spent for the 2-D problem as a function of the number of nodes with a breakdown into time spent doing the actual DSMC-related computations, communications, and load balancing. The distinction between communication and load-balancing activities was somewhat difficult to make, since time spent "waiting" for receipt of a molecule message is included in the measured communications time but is actually associated with load balancing and overall synchronization of the node processes. More than 50% of the total time is spent doing "noncomputational" activities when the number of nodes exceeds about 32.

Effect of Problem Granularity

The effective "grain size" could be increased over a reasonable range by increasing the wall temperature for the 2-D problem. Figure 10 illustrates the increase in computational and communications times with temperature on a 16-node iPSC/2. (The time attributed to load balancing was constant over this temperature range.) The time spent in doing DSMC-related computations increases (because of increased collision frequency) at a faster rate than that for communications time. Since the load balance is roughly the same, the speedup is expected to increase, and that is indeed the case as shown in Fig. 11. Increasing the ratio of computation time (per time step) to communications time resulted in an increase in the speedup from about 9.7 to 11.7.

These results illustrate the importance of "sizing" the problem to the system being used. That is, by increasing the amount of time spent doing useful computations relative to the amount of time needed to communicate data, more efficient use can be made of a given number of processors.

Effect of Load Balance

When the lower wall velocity in the 2-D box problem is nonzero, the average number density is higher near the right side of the box because of the vortex motion that is induced. Therefore, if the distribution of physical cells among processors is kept constant, the relative computational load balance should vary as the lower wall velocity is varied. Figure 12 shows the computational and communications times measured on a 16-node iPSC/2 as a function of the wall velocity. Since molecules are simply redistributed within the box, there is essentially no increase in the DSMC computational time. However, there is a significant increase in communications time as a result of nodes with fewer molecules having to "wait" for nodes with more molecules to complete a given DSMC time step. (The communications times were measured in such a way that this waiting time was included in the timing measurement along with the actual message communication time. There was no effective way to separate this waiting time using the CE/RK software.)

The CPU load distribution is shown in Fig. 13 for both zero wall velocity (the equilibrium problem) and a wall velocity of

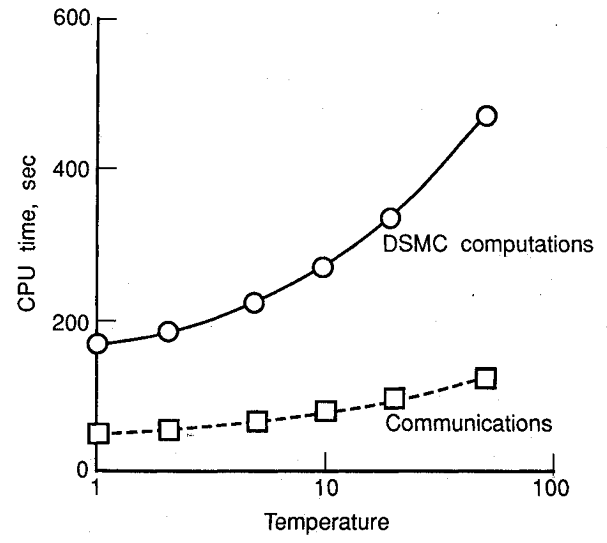

Fig. 10 Effect of temperature on problem granularity (2-D equilibrium problem on 16-node iPSC/2).

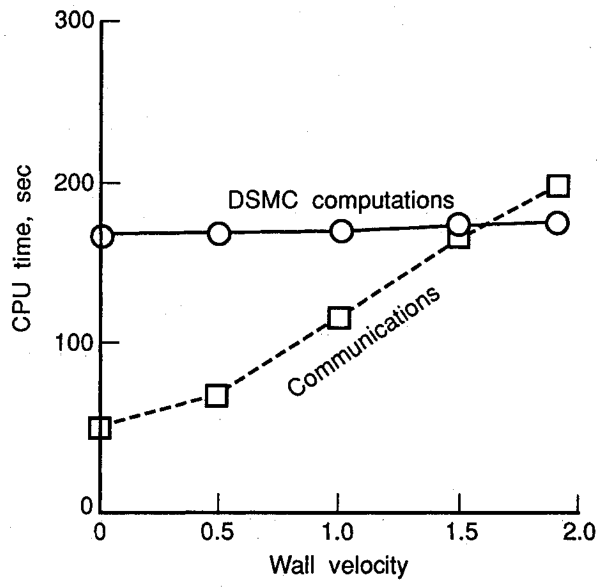

Fig. 12 Effect of wall velocity on CPU load balance (2-D equilibrium problem on 16-node iPSC/2).

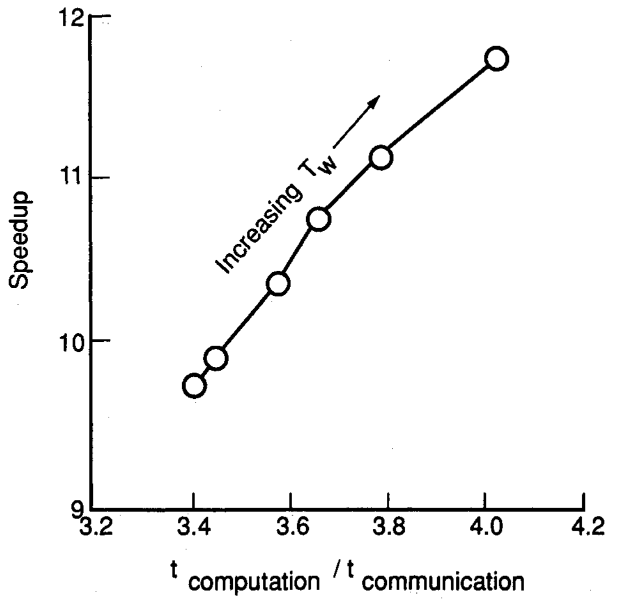

Fig. 11 Effect of problem granularity on speedup (2-D equilibrium problem on 16-node iPSC/2).

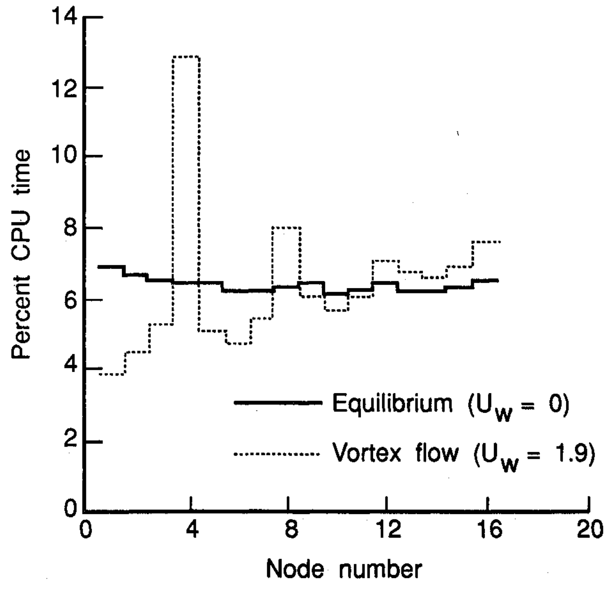

Fig. 13 CPU load distribution for 2-D equilibrium and vortex problem (16-node iPSC/2).

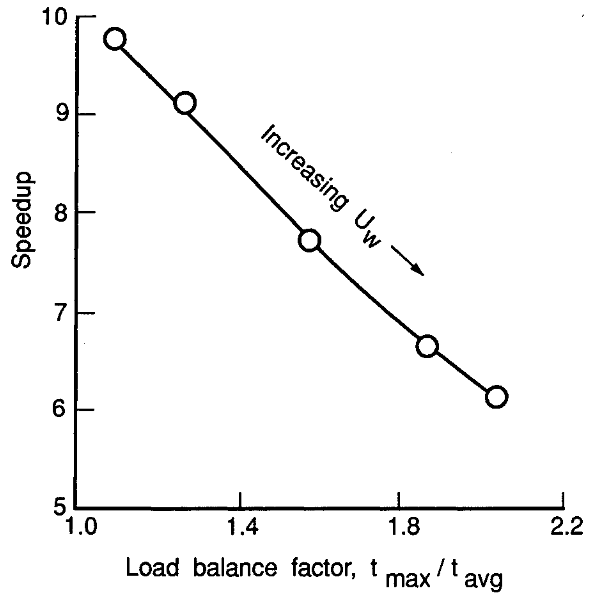

Fig. 14 Effect of load balance on speedup (2-D vortex problem on 16-node iPSC/2).

1.9. The CPU time includes only the time spent in actual DSMC-related computations and is shown as a percent of the total CPU time for the benchmark computation. The CPU distribution is relatively flat for zero wall velocity. However, for $U_w = 1.9$, the density is highest in cells contained within node 4 (the lower right-hand corner of the box in Fig. 3), and the CPU time required by this node is about twice the average time per node. As discussed earlier, the ratio of the maximum to average time per node provides a reasonable measure of the degree of load balancing, with a ratio of 1 corresponding to perfect load balance. Figure 14 shows that the measured speedup decreases in almost inverse proportion to this load balance factor.

Effect of Problem Size

As shown previously, for a given number of nodes, the speedup increases as the size of the problem relative to the communications time increases. To illustrate this effect further, an estimate of the constants in Eq. (9) was made, and this equation along with Eqs. (1), (8), and (10) were used to estimate the speedup for larger problem sizes relative to the benchmark problem. The results of these predictions are shown in Fig. 15 along with the actual measurements on the Symult S2010 and Intel iPSC/2. These predictions show that there is an optimum number of nodes (for maximum speedup) for a given problem size and that this optimum number increases with increasing problem size.

The results of these predictions were used to project the potential performance of a hypothetical hypercube in comparison with a modern supercomputer such as the Cray 2. Figure 16 shows the predicted CPU time for a hypercube with the number of nodes increasing with problem size compared with the predicted time for a Cray 2. (The CPU times for both predictions use actual measured values for a problem size of 1.) For problems only marginally larger than the 2-D benchmark used here, the Cray 2 should be somewhat faster, but for much larger problems (100–1000 times larger), the hypercube should be much faster. Note that the slope of the line predicted for the hypercube depends on the schedule used to increase the number of nodes. Although the scheduled increase in nodes with problem size used here is somewhat arbitrary, it was based on a conservative estimate of the speedup and efficiency that might be achieved in a realistic parallel system. For example, if the optimum number of nodes from Fig. 15 were used and perfect load balancing was assumed, the CPU time for the hypercube would remain constant with increasing problem size.

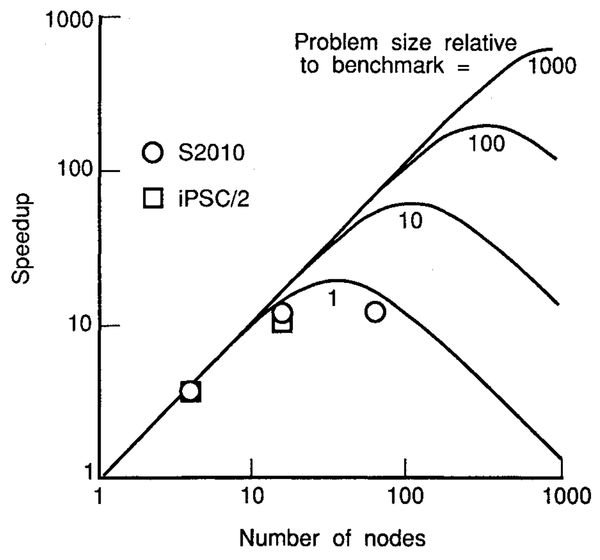

Fig. 15 Predicted speedup for different problem sizes (2-D equilibrium problem).

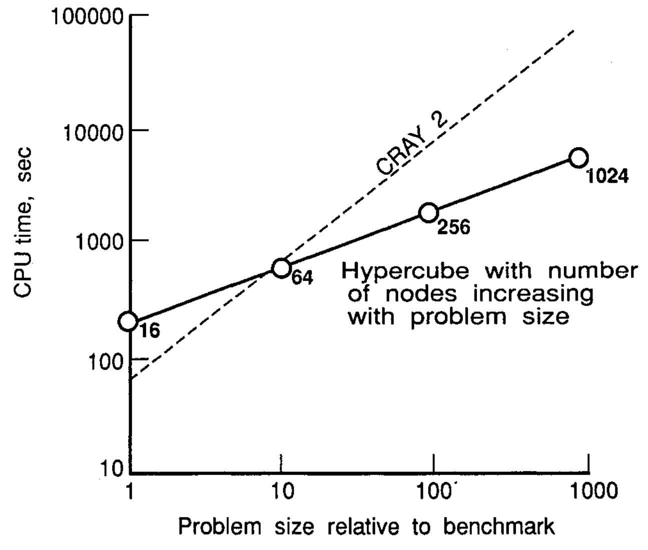

Fig. 16 Predicted CPU time as a function of problem size (2-D equilibrium problem).

Parallel Processing on Cray YMP

Some limited results have also been obtained using the *autotasking*[8] feature available on the Cray YMP. This feature provides for compiler directives to be inserted that specify portions of a program that may be executed in parallel on the multiple processors available on the YMP. The parallelism achieved was generally more fine-grained than that used for the hypercube tests. Autotasking was investigated only for the 2-D equilibrium problem, and only the Fortran version of the program was tested. The results of these tests are summarized in Table 3 along with selected results from the hypercube tests. Autotasking provided speedups of about 2–3 using eight processors on the Cray YMP. The speedup was found to depend strongly on the system load at the time of the run (i.e., number of other users, etc.) and, therefore, no firm measurement of the maximum hardware speedup was possible. However, the times for the Cray YMP are roughly one-tenth of those obtained on the other systems tested.

Concluding Remarks

A parallel direct simulation Monte Carlo (DSMC) method has been described that can be implemented in a relatively straightforward manner on local memory multiple-instruc-

Table 3 Parallel results—2-D equilibrium problem

Computer	No. of processors	Compiler	Time, s	Speedup
Cray YMP	8	Fortran w/autotasking	13–18	3.2–2.3
Symult S2010	16	Parallel C	226	10.3
Intel iPSC/2	16	Parallel C	261	9.7

tion–multiple-data types of parallel architectures using message passing to communicate between processors. The algorithm uses domain decomposition to distribute the work load among the processors so that the DSMC simulation is performed completely in parallel. Synchronization is necessary at each time step to provide the proper physical simulation.

This study has shown that significant speedups over single-processor sequential processing may be achieved. However, the actual speedup that may be obtained is strongly problem-dependent, and the problem must be properly sized to the number of processors to achieve efficient parallelism. For a two-dimensional benchmark problem, speedups of about 10 were achieved using a 16-node hypercube. By increasing the computational time relative to the communications time, speedups of almost 12 were obtained. Although the measured processing times were still greater than those achievable on supercomputers such as the Cray 2, a predicted performance comparison projected to larger DSMC problems shows that the parallel method should be able to provide computational times much less than the Cray 2 if the number of processors is properly sized to the problem. Therefore, the advantage of parallel processing in DSMC applications is not just in providing faster computations for current problems of interest but is also in allowing larger problems to be solved that are not currently practical.

The method described herein has been designed mainly to provide benchmark performance comparisons for parallel systems. It does not provide for efficient load balancing in general problem solving, and it is based on relatively coarse-grain parallelism. Further studies of the tradeoffs between the degree of parallelism, load balancing, and problem granularity are necessary to determine the optimum DSMC algorithm. With currently available parallel programming paradigms, the optimum method must provide some flexibility in these areas to take advantage of expected increases in unit processor speeds and communications bandwidth. A method that combines both parallel and vector processing is expected to provide the best performance.

Acknowledgments

The author wishes to express his appreciation to David Goldstein and Wen-King Su of Caltech for their assistance in using the Caltech Concurrent Computing Facilities, to Jeff Fier of Symult Systems for running some of the benchmark programs on the S2010, and to Andrea Overman of NASA Langley for assistance in using the autotasking feature of the Cray.

Use of trademarks or names of manufacturers in this paper does not constitute an official endorsement of such products or manufacturers, either expressed or implied, by NASA.

References

[1]Bird, G. A., "Monte-Carlo Simulation in an Engineering Context," *Progress in Astronautics and Aeronautics: Rarefied Gas Dynamics*, edited by Sam S. Fisher, Vol. 74, Pt. 1, AIAA, New York, 1981, pp. 239–255.

[2]McDonald, J. D., and Baganoff, D., "Vectorization of a Particle Simulation Method for Hypersonic Rarefied Flow," AIAA Paper 88-2735, June 1988.

[3]Furlani, T. R., and Lordi, J. A., "Implementation of the Direct Simulation Monte Carlo Method for an Exhaust Plume Flowfield in a Parallel Computing Environment," AIAA Paper 88-2736, June 1988.

[4]Goldstein, D., Sturtevant, B., and Broadwell, J. E., "Investigations of the Motion of Discrete-Velocity Gases," *Progress in Astronautics and Aeronautics: Rarefied Gas Dynamics*, edited by E. P. Muntz, D. P. Weaver, and D. H. Campbell, Vol. 118, AIAA, Washington, 1989, pp. 100–117.

[5]Bird, G. A., *Molecular Gas Dynamics*, Clarendon, Oxford, 1976.

[6]Usami, M., Fujimoto, T., and Kato, S., "Monte Carlo Simulation on Mass-Flow Reduction Due to Roughness of a Slit Surface," *Progress in Astronautics and Aeronautics: Rarefied Gas Dynamics*, edited by E. P. Muntz, D. P. Weaver, and D. H. Campbell, Vol. 116, AIAA, Washington, 1989, pp. 283–297.

[7]Seitz, C. L., Seizovic, J., and Su, W.-K., "The C Programmer's Abbreviated Guide to Multicomputer Programming," California Institute of Technology, Pasadena, CA, Computer Science Tech. Rept. 88-1, Jan. 19, 1988.

[8]*Autotasking User's Guide*, Cray Research, Inc., Rept. SN-2088, 1988.