Solving the Boltzmann Equation at 61 gigaflops on a 1024-Node CM-5

L. N. Long
The Pennsylvania State University, University Park, PA 16802 lnl@cac.psu.edu

J. Myczkowski
Thinking Machines Corporation, 245 First Street, Cambridge, MA 02142 jacek@think.com

Abstract

This paper documents the use of a massively parallel computer, specifically the Connection Machine CM-5, to solve the Boltzmann equation to model one-dimensional shock wave structure, a boundary layer, and general 3-D flow fields. The Bhatnagar-Gross-Krook (BGK) model for the collision term combined with a finite difference scheme was used to model the flow. This collision term requires accurate knowledge of the density, temperature, and mean velocity. Great care must be taken in their calculation to insure conservation, which proved to be the most difficult part. The algorithm, however, is well suited to the Connection Machine, and accurate results were obtained with great efficiency. The 1-D version of the code (which actually models a 5-D problem in phase space) was optimized for the CM-5 and sustained 61 gigaflops on a 1024-node CM-5.

Introduction

The theory of gas dynamics may be approached from two directions: the macroscopic or the microscopic. In the macroscopic approach, one considers the gas to be a continuum and employs equations such as Navier-Stokes using aggregate quantities, for example density, velocity, and temperature. This approach works well for a relatively high density, however for low density, or rarefied, problems, where the mean free path is of considerable length compared to some dimension of the system, this continuum approach breaks down. For instance, in order to model flows in the low density of the upper atmosphere, such as the flow over the Space Shuttle upon reentry or the flow in microdevices, one cannot use the continuum approach of Navier-Stokes. The Navier-Stokes equations are only valid when the mean free path is much smaller than the characteristic lengths in the flow. For these cases, one must treat the gas microscopically, that is, consider it to be a conglomeration of particles instead of a continuum. This more general situation can be described by the Boltzmann equation.

The Boltzmann equation is an integro-differential equation in terms of the particle distribution function, f(**x**,**v**,t). Unfortunately, the Boltzmann equation, even for the simplest cases, is quite difficult to solve analytically or numerically. In some cases, numerical schemes have involved various simplifications specific to the given problem in order to reduce the computing resources needed. The only general approach has been the direct simulation Monte Carlo method (DSMC) [1], which involves random sampling to calculate various parameters. This method requires significant computing resources. The DSMC method is very communication intensive [2] and may never fully utilize the power of vector or parallel computers. The other major drawback to DSMC is that it is essentially a first-order accurate, explicit method [3]. More traditional numerical schemes may be able to achieve second-order accuracy with an implicit scheme, which may offer dramatic improvements in efficiency. This will be especially true if the newer algorithms can more effectively use massively parallel computers, which is true of the present algorithm.

The algorithm used here was specifically designed for data parallel computers. Drawing on the authors experience with DSMC [2,3] and continuum codes [4,5] on parallel computers, a new algorithm was developed [6,7] that exploits the power of massively parallel computers for rarefied gas dynamics.

A key difference between parallel and serial/vector computers is their memory organizations. Most massively parallel architectures use distributed memory. Serial and vector computers rely on shared memory. The distributed nature of parallel computer memory introduces additional constraints into the programming model by requiring a careful data layout and an understanding of the costs of

interprocessor communications. Since most algorithms designed to solve scientific problems will require communication, it is crucial to understand the additional complexity introduced by distributed memory. An effective algorithm for a massively parallel computer must satisfy a number of conditions. Some of the more important are locality and regularity of the communication pattern. On conventional architectures the number of flops (floating point operations per second) is a good indicator of algorithm efficiency. This not the case on parallel computers, where interprocessor communications do not contribute to useful flops but do contribute to the total execution time.

It should always be remembered that a code that achieves a high flop rate on a parallel computer is not necessarily the *best* algorithm. Another algorithm that uses more communications at the expense of the floating point hardware may converge faster and prove to be a better algorithm. The ultimate measure of performance is how long it takes to solve a problem. Codes that do achieve very high peak speeds though are useful as guidelines for optimization and for illustrating peak obtainable performance levels.

The Boltzmann equation

The Boltzmann equation governs the seven-dimensional particle distribution function, $f(\mathbf{x},v,t)$, and from f, one can calculate the macroscopic flow quantities. This distribution function governs the number density of molecules in physical ($\mathbf{x}$) and velocity ($\mathbf{v}$) space. It may also be a function of time. The general form of the Boltzmann equation for a monatomic gas and no external forces can be written

$$\frac{\partial f}{\partial t} + \mathbf{v}\cdot\nabla f = \int\int\int_{-\infty}^{\infty}\int_{0}^{4\pi}\left[f(\mathbf{v}^*)\, f(\mathbf{w}^*) - f(\mathbf{v})\, f(\mathbf{w})\right] g\, I\, d\Omega\, D\mathbf{w}$$

where g is the relative velocity between two particles having velocity $\mathbf{v}$ and $\mathbf{w}$ before collision and $\mathbf{v}^*$ and $\mathbf{w}^*$ after collision, I is the differential collision cross-section for scattering into the elementary scattering angle $d\Omega$, and $D\mathbf{w}$ is the differential element of volume in $\mathbf{w}$-space. The right hand side is known as the collision term and accounts for changes in the distribution function due to particle collisions. Calculating the collision term is the main difficulty in solving the Boltzmann equation. Simplifications of this term have been developed for various cases. Any model for this term must: (i) conserve mass, momentum and energy, (ii) yield a Maxwellian distribution at equilibrium conditions, and (iii) tend to drive a nonequilibrium distribution towards the Maxwellian distribution.

For some flows, the Bhatnagar, Gross, and Krook (BGK) model [8,9] of the collision term can be used. The basic idea of the BGK equation is that a detailed description of the collision process may not be necessary and that a larger scale approximation may be appropriate. Fairly simple collision models have been used in DSMC with some success (e.g. the variable hard sphere model).

The BGK equation was used in this study,

$$\frac{\partial f}{\partial t} + \mathbf{v}\cdot\nabla f = \alpha\,(f_0 - f)$$

where α = collision frequency and f_0 is the local Maxwellian

$$f_0 = \frac{n\,\beta^3}{\pi^{3/2}}\, e^{-\beta^2(\mathbf{v}-\mathbf{u}_0)^2}$$

where n, $\mathbf{u}_0$, and $T = 1 / (2R\beta^2)$ are the local density, mean velocity and temperature, respectively, and R is the gas constant. One can find n, $\mathbf{u}_0$, and T from the following moments of f,

$$n = \int_{-\infty}^{\infty}\int\int\int f\, D\mathbf{v}$$

$$n\,\mathbf{u}_0 = \int_{-\infty}^{\infty}\int\int\int \mathbf{v}\, f\, D\mathbf{v}$$

$$2\,R\,T = \frac{1}{\beta^2} = \frac{2}{3n}\int_{-\infty}^{\infty}\int\int\int (\mathbf{v}-\mathbf{u}_0)^2\, f\, D\mathbf{v}$$

In the limits of zero and infinite Knudsen numbers, this equation duplicates the Boltzmann equation exactly. For finite Knudsen numbers it is an approximation to the Boltzmann equation and yields a Prandtl number, Pr, of unity. One can approximate the viscosity-temperature relation using, for example, the Sutherland viscosity law. If this is done, however, the thermal conductivity will be slightly in error since Pr = 1.

Algorithm

To find a solution, f, of the BGK equation, an algorithm was designed specifically for data-parallel computers.

This algorithm [6,7] has the following features :

1. Upwind finite differences for the spatial derivatives.
2. Runge-Kutta time-stepping scheme for the time derivative.
3. Gaussian quadrature for the integrals over velocity space.
4. Least squares method for conservation.

Thus one must discretize the independent variables $\boldsymbol{x}$, $\boldsymbol{v}$, t of $f(\boldsymbol{x},\boldsymbol{v},t)$.

Mapping to the CM-5

To solve the finite difference equation on the CM, one must first decide how to distribute the physical and velocity mesh over the processors for optimal utilization of the CM. This requires a close look at the operations performed at each step in the iteration process.

An explicit scheme will calculate $f(x_i, \boldsymbol{v}, t_{i+1})$ from terms such as

$f(x_{i-1}, \boldsymbol{v}, t_i)$, $f(x_i, \boldsymbol{v}, t_i)$, $f(x_{i+1}, \boldsymbol{v}, t_i)$, *and* $f_0(x_i, v, t_i)$,

where f_0 is the local Maxwellian distribution. The scheme used here distributes the spatial dimension across the processors. Calculating f at the next time step then involves iterating over the velocity space points. Some interprocessor communication is necessary since the difference approximation to $\partial f/\partial x$ requires f at adjacent mesh points, but this is minimal and only involves nearest neighbor communication. Also, since the integration to calculate n, T, and $\boldsymbol{u}_0$ is done over all the velocity space for each physical space point, each processor's memory contains all the necessary data to calculate these moments and no interprocessor communication is necessary. This is a tremendous advantage and results in nearly optimal use of the CM.

Quadrature and conservation

For each point in physical space, one needs to keep track of a full 3-D mesh of points in velocity space to represent the distribution function at that point. This distribution is then used to approximate the integrals for calculating n, $\boldsymbol{u}_0$, and T. The macroscopic quantities n, $\boldsymbol{u}_0$, and T are given by the three moments of f shown earlier. Over the discretized space, any number of integration schemes can be used. The algorithm implemented was Gaussian quadrature. Seven points were used in each direction for a total of 343 points in velocity space, which is equivalent to using a 13th order polynomial in each velocity direction. It is important to find n, $\boldsymbol{u}_0$, and T such that calculation of the moments of f_0 by quadrature gives exactly the same result as that for f by quadrature. Thus conservation of mass, momentum, and energy is accomplished, i.e. the zeroth, first, and second moments of f-f_0 must be zero.

This is accomplished by requiring that :

(1) The moments of f_0 and the moments of f are exactly equal.
(2) The "equilibrium" distribution is close (in a least squares sense) to a Maxwellian.

That is, n, $\boldsymbol{u}_0$, and T are computed from the moments of f (by quadrature). These macroscopic quantities then define a Maxwellian f_M. This Maxwellian is then corrected so that the above two requirements are satisfied. This is accomplished using the following algorithm [10,11] :

$$f_{0_k} = f_{M_k} + A^T \left[A\, A^T \right]^{-1} A \left(f_k - f_{M_k} \right)$$

For a 1-D flow, A is a 3 x 343 matrix representing the quadrature weights (in 3 velocity dimensions) for the three moments and the subscripts k on the distribution functions denote points in velocity space. For 3-D flow A would be a 5 x 343 matrix, so the problem is not changed significantly when going to 3-D problems.

Now, the BGK collision term can be written

$$\alpha \left(f_{0_k} - f_k \right) = \alpha \left(A^T M - I \right) \left(f_k - f_{M_k} \right)$$

where

$$M = \left[A\, A^T \right]^{-1} A$$

and I is a 343x343 identity matrix. Note that A is a constant matrix and the product of A^T and M could be computed and stored as a front-end array, but this yields an inefficient scheme. The inefficiency results due to the numerous broadcasts required from the front-end to the CM and the large number of operations required, $O(343^2)$. Doing the operations this way would also continually break the computation pipeline, so the ALUs would not be able to achieve peak speed.

By leaving the scheme in the form shown above, one can store copies of the matrices A^T and M on each processor (2058 numbers). Storing the square matrix $A^T M$ would not be practical since it would amount to 343^2 numbers per processor. Keeping A^T and M separate also reduces the number of operations required, O(343). The

performance advantages of storing these matrices on the CM are enormous. It allows one to compute the collision terms without performing any communication. And these terms represent the most compute intensive portions of the algorithm.

Code optimization

The CM-5 node design [12] is centered around a standard bus. To this bus are attached a RISC microprocessor, a CM-5 Network Interface, and floating point hardware. All logical connections to the rest of the system pass through the Network Interface (NI). The node memory consists of standard DRAM. The arithmetic hardware consists of four vector units (VU), one for each memory bank, connected separately to the node bus. Each VU is a memory controller and computational engine controlled by a memory-mapped control-register interface.

There are two places in the FORTRAN 90 (CMF - Connection Machine Fortran) code that have been optimized. The first optimization can be described as a "stencil" operation. This is illustrated with the code fragment below.

```
      real, array ( pnodes, xnodes)     :: f, fold
cmf$ layout f(:serial, :news), fold(:serial, :news)
       do np = 1, pnodes
            coef  = rkc(irkc) * up(np) / dx * dt
           if ( up(np) .gt. 0.0 ) then
              f(np,2:xnodes) = fold(np,2:xnodes) - coef *
( f(np,2:xnodes) - f(np,1:xnodes-1) )
           else
            f(np,1:xnodes-1) = fold(np,1:xnodes-1) - coef
                *( f(np,2:xnodes ) - f(np,1:xnodes-1))
           endif
         enddo
```

where irkc, dx, and dt are assumed to be constant in this do loop. In this code fragment the first dimension of the arrays is completely contained within a processor (serial axis). The second dimension is spread across the processors (news axis). When written in CMF this code fragment is not executed optimally. The major problem is that communications operations involve data motion on-chip as well as off-chip (only off-chip data is necessary) and the fact that contextualization is expensive. We have rewritten this code fragment using assembler language (CDPEAC). In this code we have optimized communications in two ways. The physical (on-chip) part of the communication is vectorized across the serial dimension. For example if for vector unit (VU) p, first(p) and last(p) are the first and last indices along the news axis holding data stored on that VU then each VU first does:

```
t1 = f(:,first(p)-1)
t2 = f(:,last(p)+1)
```

This is implemented with calls to an optimized communication primitive. The on VU motion is incorporated into the computation (and thus entirely eliminated). The up array is broadcast to all the VU's.

The on VU kernel has the same basic structure as the fortran loop above (i.e. outer loop on the serial dimension, inner loop on the news dimension). There are two versions of the inner loop, one for the up(np) > 0.0 case and one for the other. Consider the up(np) > 0.0 case. Effectively we do:

```
trow(1) = t1(i)
trow(2:n) = f(i, first(p):last(p)-1)

f(i, first(p):last(p)) = fold(i, first(p):last(p) - coef *
   ( f(i, first(p):last(p)) - trow)
```

However we don't explicitly build the temporary, trow. In the vector code we have a vector register, Vf, which holds the direct access to f. If we explicitly constructed trow, the last 15 elements of the vector register holding it would be the same as the first 15 elements of Vf. Rather than duplicating them we simply put the one different element of trow in Vf[-1] and use Vf[-1] as the vector register for trow. Before starting the inner loop we put t1(i) into Vf[-1]. Each time we load a new piece of f into Vf we first copy Vf[15] to Vf[-1]. The vector loop takes 3 cycles per vector element, a subtraction (t = f - trow) with overlapped load from f, a sub-multiply (fold - coef * t) with an overlapped load from fold and a store with movement from Vf[15] to Vf[-1] overlapped. A similar approach is used for the up(i) <= 0.0 case.

To handle the boundary conditions (i.e. not overwriting f(i, 1) when up(i) > 0.0 and not overwriting f(i,xnodes) when up(i) <= 0.0) we take advantage of fact that f is the same as fold in those positions. Figuring this out required information from other parts of the code than what we have above. Given that, on the first node we overwrite t1 with f(:,1) and on the last node we overwrite t2 with f(:,xnodes). So in such a boundary position the statement is eg:

```
f(i,1) = fold (i,1) - coef * (f(i,1) - t1(i))
```

and since t1(i) is the same as f(i,1) this reduces to

```
f(i,1) = fold(i,1)
```

which is fine since fold and f are the same at this position.

The second optimization is the rewrite in assembler (CDPEAC) of a matrix vector multiply operator. The code fragment is shown below :.

```
      DOUBLE PRECISION, ARRAY(xnodes) ::
            alt1, alt2, alt3
   DOUBLE PRECISION, ARRAY(pnodes, xnodes) ::
            aatia1, aatia2, aatia3, ftmp

CMF$  LAYOUT aatia1  (:SERIAL,:NEWS)
CMF$  LAYOUT aatia2  (:SERIAL,:NEWS)
CMF$  LAYOUT aatia3  (:SERIAL,:NEWS)
CMF$  LAYOUT ftmp    (:SERIAL,:NEWS)

      DO p=1,pnodes
        alt1(:) = alt1(:) + aatia1(p,:) * ftmp(p,:)
        alt2(:) = alt2(:) + aatia2(p,:) * ftmp(p,:)
        alt3(:) = alt3(:) + aatia3(p,:) * ftmp(p,:)
      END DO
```

The major consideration in this optimization is to reduce the number of memory references made by the code emitted by the CMF compiler. The CMF compiler was executing 6 flops per 10 memory references. The optimized version of that code achieves an asymptotic rate of 24 flops per 16 memory references.

Results

Reference 6 presented a one-dimensional version of the BGK algorithm and some shock wave structure calculations. Figure 1 shows an example simulation result, the velocity profile through a Mach=1.5 shock wave for a monatomic gas. These results were obtained in 2000-3500 time steps using 1024 spatial nodes. Also shown are the BGK results of Liepmann et al [13] for velocity and temperature. This version of the code was highly optimized to achieve the high gigaflop rates.

The new three-dimensional BGK code was used to simulate the Rayleigh problem (impulsively started boundary layer), in order to verify the formulation. Diffuse reflection solid surface boundary conditions were implemented. Figure 2 shows the profile of the velocity component perpendicular to the wall for the same conditions (supersonic, hot wall) that were used in Refs. 1 and 2. These results correspond to a non-dimensional time of five.

Qualitative results for two three-dimensional flows are presented in Figures 3 and 4. Figure 3 shows Mach number contours around a cubical body moving at Mach = 1.5 with a Reynolds number (based on body length) equal to 50. The bow shock and wake are clearly visible. The far-field boundary conditions also work well, since no reflection of the shock is apparent. Figure 4 shows Mach = 0.5 flow through a square duct, with a Reynolds number (based on duct length) of 1000. The very thick boundary layer is apparent. Both of these flows were computed using the same code, with minor changes to the boundary conditions.

Code performance

The 1-D version of the code was run on the Los Alamos National Labs 1024-node CM-5 and a 32-node CM-5 at Thinking Machines Corporation. The results are shown in Table 1 for 10 time steps. The program achieved roughly half of the peak speed of the CM-5 and demonstrated scalability.

It should also be pointed out that the unoptimized CMF code ran quite well, achieving roughly 40 gigaflops (31 % of peak speed) on a 1024-node CM-5. On a 32K CM-2, 2.9 GFlops were measured (32-bit precision). An 8K CM-200 achieved 1.0 (32-bit precision) and 0.7 (64-bit) precision) gigaflops. Scaling these results to a 10 MHz 64K processor CM-200, would yield approximately 8 gigaflops.

The 3-D version of the code has essentially the same collision term as the 1-D code (except A is 5x343 for 3-D). The only difference between the two codes is that the 3-D code has the full gradient term

$$v \bullet \nabla f$$

instead of the simple 1-D term $\partial f / \partial x$. The majority of the computations are in the collision term. So, while we have not completely optimzed the 3-D code, it is expected to sustain very high computational speeds.

Conclusions

The Connection Machine has proved itself a valuable tool for solving the BGK equation. The algorithm developed had a good balance of computation and communication, which made the solution on the CM very efficient. When enforcing the conservation laws, the CM proved very effective since all the processor computations turned out to be independent of one another and required no communication.

Knowledge of experimental trends and also a comparison

to results of past work gives confidence that the algorithm is properly modeling the physics of the gases. With the physics properly in place, this algorithm can now be developed further to model more complex problems.

Acknowledgment

This work was supported by the National Science Foundation (ASC-9009998) and The Pennsylvania State University. The authors would also like to acknowledge the significant contributions of M. Bromley, D. Dahl, R. Lordi, and R. Shapiro of Thinking Machines Corporation and M. Kamon [14] currently a graduate student at MIT.

References

1. G. A. Bird, *Molecular Gas Dynamics*, Oxford Univ. Press, Glasgow, 1976.

2. B. C. Wong and L. N. Long, "The DSMC Method on the Connection Machine," Computing Systems in Engineering, Vol.. 3 , No. 4, Dec., 1992.

3. L. N. Long, "Navier-Stokes and Monte Carlo Results for Hypersonic Flow," AIAA Journal, Vol. 29, No. 2, Feb., 1991.

4. L. N. Long, M. N. S. Khan and H. T. Sharp, "Massively Parallel Three-Dimensional Euler/Navier-Stokes Method," AIAA Journal, Vol. 29, No. 5, May, 1991.

5. L. N. Long, "Gas Dynamics on the Connection Machine," Proceedings of Parallel CFD '92, Rutgers, NJ, May, 1992.

6. L. N. Long, M. Kamon, J. Myczkowski, "A Massively Parallel Algorithm to Solve the Boltzmann (BGK) Equation", Computing Systems in Engineering, Vol. 3, , No. 4, Dec., 1992.

7. L.N. Long, B.C. Wong, and J. Myczkowski, "Deterministic and Non-Deterministic Algorithms for Rarefied Gas Dynamics," Proceedings of the 18th International Symposium on Rarefied Gas Dynamics, to appear in AIAA Progress in Astronautics and Aeronautics, 1993.

8. W. G. Vincenti and C. H. Kruger, *Introduction to Physical Gas Dynamics*, Krieger, Malabar, 1986.

9. C. K. Chu, "The High Mach Number Rayleigh Problem According to the Krook Model," *Rarefied Gas Dynamics*, ed. C.L. Brundin, pp. 589-605, 1967.

10. R. M. Freund, Private Communication, Boston, MA, Aug., 1991.

11. G. Strang *Introduction to Applied Mathematics*, Wellesley-Cambridge Press, Wellesley, 1986.

12. CM-5 Technical Summary, Thinking Machines Corporation, Cambridge, Massachusetts, Oct., 1991.

13. H. W. Liepmann, R. Narasimha and M. T. Chahine, "Structure of a Plane Shock Layer," The Physics of Fluids, Vol. 5, No. 11, November 1962.

14. M. Kamon, "A Massively Parallel Algorithm to Solve the BGK Equation for Shock Wave Structure," Penn. State Univ. Honor's Thesis, Engineering Science Department, May, 1991.

Table 1. 64-bit Performance results for 10 time steps of the 1-D BGK code on a 1024-node CM-5.

Processor Nodes	Number of Grid Points in X	Total Number of Unknowns	Floating Point Operations	CPU Seconds	Gigaflops	Percent of Peak Speed
32	16,384	5,619,712	$1.49 * 10^{10}$	7.57	1.97	48 %
1024	524,288	179,830,784	$4.76 * 10^{11}$	7.84	60.71	46 %