Early Experiences Developing CFD Solvers for the Intel MIC Architecture
Ryan C. Hulguin and R. Glenn Brook

Introduction

The time to final solution with computational fluid dynamics (CFD) solvers greatly influences the rate that analysis and design decisions can be made. The increase in CPU clock speeds through the 1980’s and 1990’s has benefited the CFD community tremendously. The raw compute power has enabled the simulation of more accurate and more complicated flow problems. Recently CPU clock speeds have hit a plateau, and further improvements are now achieved through parallel computing. Many CFD solvers today run in parallel using a distributed memory environment over small clusters or even large supercomputer systems. The chips used in these systems only feature a small number of cores per processor. Intel has now introduced its many integrated core architecture (MIC) that combines many Intel CPU cores onto a single chip. Two CFD solvers are developed for use on the MIC software development platform known as Knight’s Ferry (KNF). Excellent scaling across the cores of the KNF card is observed.

Knight’s Ferry

KNF cards provided by Intel have up to 32 cores on a chip and they are based on the x86 instruction set [1]. Each core has a 512-bit vector unit that can perform 16 single precision floating point operations in a single instruction. Additionally, each core supports 4 hardware threads. The caches are kept coherent through a ring interprocessor network. The KNF cards operate on the PCI-Express bus and they come with either 1 or 2 gigabytes of GDDR5 memory. The particular KNF card used in this study has 32 cores and 2 gigabytes of memory.

Governing Equations

The first CFD solver developed is based on the Euler equations shown below:

\[
\frac{\partial \vec{Q}}{\partial t} + \frac{\partial \vec{F}}{\partial x} + \frac{\partial \vec{G}}{\partial y} + \frac{\partial \vec{H}}{\partial z} = \vec{0}
\]

\[
\vec{Q} = [\rho, \rho u, \rho v, \rho w, E]^T
\]

\[
\vec{F} = [\rho u, \rho u^2 + P, \rho u v, \rho u w, (E + P)u]^T
\]

\[
\vec{G} = [\rho v, \rho u v, \rho v^2 + P, \rho v w, (E + P)v]^T
\]

\[
\vec{H} = [\rho w, \rho u w, \rho v w, \rho w^2 + P, (E + P)w]^T
\]

where \( \rho \) is the density, \( \vec{u} = (u, v, w) \) are the velocities, \( P \) is the pressure, and \( E \) is the total energy per unit volume.

The Euler equations are used to simulate inviscid fluid flows and the particular solver developed is an unstructured 3d solver that has 5 state variables at each physical gridpoint.

The second CFD solver developed is based on the BGK model Boltzmann equation using a discrete velocity model [2] shown below:

\[
\frac{\partial f_k}{\partial t} + \vec{V}_k \cdot \frac{\partial f_k}{\partial \vec{x}} = v \cdot (f_k^{eq} - f_k)
\]
\[ f^{eq} = \frac{n}{(\pi T)^{3/2}} \left( -\frac{(\vec{V}_k - \vec{u})^2}{T} \right) \]
\[ v = \frac{8n T^{1-\omega}}{5\sqrt{\pi} Kn} \]

where \( f_k \) is the probability distribution function at discrete velocity \( k \), \( \vec{V}_k = (V_{x_k}, V_{y_k}, V_{z_k}) \) are the discrete molecular velocities, \( \vec{u} = (u, v, w) \) are the bulk (average) velocities, \( n \) is the number density, \( T \) is the temperature, \( \omega \) is the gas viscosity index, and \( Kn \) is the Knudsen number which characterizes the flow. The Knudsen number is defined as the ratio between the mean free path \( \lambda_{\omega} \) and the characteristic length \( L \).

The model Boltzmann equation is used to simulate rarefied gas flows and the particular solver developed is a structured 3d solver that has hundreds of thousands of state variables at each physical gridpoint.

**Numerical Algorithm**

Both solvers are developed using a novel point-iterative algorithm [3] to solve the nonlinear systems of equations with a near minimal memory footprint. Given a nonlinear set of equations, \( \vec{F}(\vec{q}) = 0 \), Newton’s method can be applied to give

\[ \vec{q}^p = \vec{q}^{p-1} - J(\vec{q}^{p-1})^{-1} \vec{F}(\vec{q}^{p-1}) \]

where \( p \) is the Newton iteration and \( J \) is the jacobian as defined below

\[ J_{k j} = \frac{\partial f_k}{\partial q_j} : 1 < k, j < n \]

Rearranging terms leads to

\[ J \Delta \vec{q}^p = -\vec{F} \]

where \( \Delta \vec{q}^p = \vec{q}^p - \vec{q}^{p-1} \).

Using the Jacobi method to solve the linearized system of equations gives

\[ (\Delta q^p)^m_k = \frac{1}{J_{kk}} \left[ -F_k - \sum_{j \neq k} J_{kj} (\Delta q^p)^{m-1}_j \right] \]

where \( m \) is the current Jacobi iteration. With some slight algebraic manipulation, the iterative update equation can be cast into a delta formulation as shown below

\[ (\Delta q^p)^m_k - (\Delta q^p)^{m-1}_k = \frac{1}{J_{kk}} \left[ -F_k - \sum_{j \neq k} J_{kj} (\Delta q^p)^{m-1}_j - J_{kk} (\Delta q^p)^{m-1}_k \right] \]

\[ \Delta[ (\Delta q^p)^m_k ] = \frac{1}{J_{kk}} [ -F_k - J(\Delta \vec{q}^p)^{m-1} ] \]

The novelty lies in the fact that the jacobian does not need to be explicitly calculated and stored. Instead it can be calculated implicitly through the use of dual numbers and Taylor series expansions in the dual space.

\[ J_k (\Delta q^p)^{m-1} \approx \frac{1}{h} \text{Dual}[F_k (\vec{q}^{p-1} + \epsilon h (\Delta \vec{q}^p)^{m-1})] \]
\[ F_k \approx \text{Real}[F_k (\vec{q}^{p-1} + \epsilon h (\Delta \vec{q}^p)^{m-1})] \]
\[ J_{kk} \approx \frac{1}{h} \text{Dual}[F_k (\vec{q}^{p-1} + \epsilon h \vec{e}_k)] \]
where $\varepsilon^2 \equiv 0$, $h \in \mathbb{R}$ is an arbitrary perturbation and $e^k$ is $k$th vector in the standard basis for $\mathbb{R}^N$.

Parallel Implementation

Data parallelism across the cores is achieved using OpenMP threads. More specifically, a `#pragma omp parallel for` directive is placed just outside the loop over all physical cells. Due to the nature of the Jacobi iterative method, all work in this loop is embarrassingly parallel. Additional loops are made parallel using OpenMP threads as well. Pseudocode for the solvers is shown below

```plaintext
Parallel for loop over gridpoints
  Set initial values
End parallel for loop over gridpoints
Loop over timesteps
  Apply boundary conditions
  Parallel for loop over gridpoints
    Reset values of dq to zero
  End parallel for loop over gridpoints
  Loop over number of Jacobi iterations
    Parallel for loop over gridpoints
      Solve for ddq via Jacobi iteration
    End parallel for loop over gridpoints
  End loop over number of Jacobi iterations
  Parallel for loop over gridpoints
    Update dq using ddq
  End parallel for loop over gridpoints
End loop over timesteps
End parallel for loop over gridpoints
```

Both solvers are compiled for native mode on the Intel MIC, and currently only use a single KNF card.

Test Problems

The first test problem uses the Euler solver to simulate a Sod Shock [4]. In a shock wave, the properties of a fluid change almost instantaneously. The standard Sod Shock starts off with a fluid at rest with the following initial conditions:

- $\rho_{\text{left}} = 0.0$, $u_{\text{left}} = 0.0$, $P_{\text{left}} = 1.0$ and $\rho_{\text{right}} = 0.125$, $u_{\text{right}} = 0.0$, $P_{\text{right}} = 0.1$

The Sod Shock is a popular test case for verifying a solver’s ability to appropriately capture shocks and contact discontinuities in unsteady fluid flows. The solution generated is shown below

![Unsteady Sod Shock](image_url)
The second test problem uses the Boltzmann solver to simulate a stationary shock wave. The standing shock is a traditional test case for CFD. The problem is initialized with a numerical discontinuity between two states. The upstream state is fast and cold, while the downstream state is slow and hot. The mach 1.55 shock profile for Argon [5] is simulated and the result is shown below.

![Normalized Density Profile](image)

The third test problem uses the Boltzmann solver to simulate a Couette flow. In Couette flows, gas is initially at rest between two infinitely long parallel plates. For this simulation [6], the left plate is stationary while the right plate moves at 300 m/s. The gas between the plates is initialized with the following values:

\[ \rho_0 = 9.28 \times 10^{-8} \text{ kg/m}^3, u_{x0} = u_{y0} = 0.0 \text{ m/s}, T_0 = 273.0 \text{K}, Kn = 0.1199 \]

Both plates are set at a temperature of 273.0K. Over time, the gas settles into a solution that does not change. Couette flow makes a great test problem to verify a solver’s ability to handle solid surfaces and moving boundary conditions. The steady state solution achieved is shown below.

![Couette Flow](image)

**Performance**

Each solver was run on the KNF card in native mode with a varying number of threads ranging from 1-128 to assess the scaling performance across the many cores. Speedup from the Sod shock test problem using both single and double precision is shown below.
32 is considered to be the ideal speedup since the KNF card used only has 32 physical cores. When 16 threads are requested, 92%-96% of the ideal speedup is achieved. It is interesting to note that when 96 threads are requested, 99% of the ideal speedup is achieved when using single precision. The speedup observed when using double precision, even when doubling the problem size, was not able to match that achieved when using single precision.

The other CFD solver based on the BGK model Boltzmann equation is put through the same study. Speedup from the Couette test problem run in single precision is shown below.

At 16 threads, only 75% of the ideal speedup is achieved and at 96 threads, only 63% of the ideal speedup is achieved. This is most likely due to the memory layout of the data, considering each grid point has hundreds of thousands of state variables associated with it. It is worth noting that both CFD solvers have not yet undergone extensive performance optimizations, and doing so may change the observed scalability.

**Future Work**

As stated earlier, the CFD solvers presented here have not yet undergone extensive performance optimizations, but they will be performed in the future. Also, the
current work investigates using the Intel MIC technology in native mode with OpenMP threads. Future work will investigate using multiple MIC cards with MPI communications between the cards, and also using the host cores in a heterogeneous manner.

Acknowledgments

The authors would like to thank both the National Institute for Computational Sciences (NICS) and Intel for making this research possible.

References


