CloverLeaf Description

The System of Equations and their Numerical Solution

CloverLeaf is a mini-app that solves the compressible Euler equations on a Cartesian grid. Each cell stores three values: energy, density, and pressure. A velocity vector is stored at each cell corner. This arrangement of data, with some quantities at cell centres, and others at cell corners is known as a staggered grid. CloverLeaf currently solves the equations in two dimensions, but three dimensional support will be added in an upcoming release.

The compressible Euler equations are a set of three partial differential equations that describe the conservation of energy, mass and momentum in a system. CloverLeaf produces a second-order accurate solution using explicit finite volume methods. It first performs a Lagrangian step, using a predictor-corrector scheme to advance the solution forward by a calculated time delta. This step causes the mesh to move with fluid velocity, so an advective remap is used in order to return the mesh to its original state. A second-order Van Leer scheme is used, with the advective sweep being performed in the x and y directions for the energy, mass and momentum. The initial sweep direction alternates between steps, providing second order accuracy. The flow direction mush be calculated during the remap to allow data from the "upwind" direction to be used. Although the deformation of the grid does not actually move cell vertices, the average velocity on a cell face is used to approximate a flux through each face for the advection of material.

The compressible Euler equations form a hyperbolic system and therefore generate discontinuities in the form of shock waves. The second-order approximation will fail at these discontinuities (since the ...) and cause "ringing" in the solution. To avoid this, an artificial viscous pressure is used, which makes the solution first order in the presence of shock waves. This preserves monotonicity in the solution, by behaving as a simple addition to the pressure.

The timestep control uses the maximum sound speed as an upper bound for the time delta. The timestep is thus limited to the time it would take for the the highest speed sound wave to cross a cell. The timestep is then multiplied by a safety factor to preserve the stability of the solution. The timestep control contains two further tests: one to ensure that a vertex can't overtake another as the mesh deforms, and one to ensure that a cell cannot deform such that it's volume becomes negative.

In order to close the system of equations, we use an equation of state, which calculates the pressure and sound speed in a cell, given its energy and density. CloverLeaf uses the ideal gas equation of state with a gamma value of 1.4.

Currently, CloverLeaf only solves for a single material, although multiple states (pressures, densities, and velocities) of this material can exist in the problem domain. Support for multiple materials will be added into a future release.

The Implementation of the Algorithm

The algorithm is straightforward to implement if a serial compute architecture is assumed. However, current and future architectures are not designed to be programmed is this fashion, and hence, CloverLeaf has been designed to perform well in a number of areas: memory accesses, data locality, compiler optimisations, threading, and vectorisation.

The computation in CloverLeaf has been broken down into "kernels" -- low level building blocks with minimal complexity. Each kernels looks over the entire grid and updates one (or some) mesh variables, based on a kernel-dependent computational stencil. Control logic within each kernel is kept to the minimum possible level, allowing maximum optimisation by the compiler. Memory is sacrificed in order to increase performance, and any updates to variables that would introduce dependencies in the kernel are written into copies of the mesh. Each kernel is also written so that every cell can be updated independently of any other, allowing the kernels to be threaded or vectorised easily.

Boundary Cells and Halo Exchange

At the edge of the computational domain boundary conditions are used to close the solution. Extra "halo" cells around the mesh provide data for the computational stencil when required. These halo cells are not, and do not need to be, updated by the computational kernels. Data in the halo cells is filled in one of two ways: (1) at a boundary between processors, data is simply copied from the cells held by one processor, into the halo cells of the processor holding the adjoining portion of the mesh, and (2) at the edge of the computational domain, the cells are filled using a physical boundary condition. CloverLeaf currently only uses a reflective physical boundary condition.

Implementations

The underlying strategy behind the development of CloverLeaf was to keep the code base as simple as realistically possible, with low-level kernels perform the computational work without using many levels of function calls. The baseline version of the code has been written in such a way as to facilitate porting to any arbitrary language or architecture. Language and vendor specific extensions such as (Fortran BLAH or vector intrinsics) were avoided in case these would inhibit other implementations.

Fortran and C

Each of the compute kernels was initially written in both Fortran and C. The code is identical in all but syntax and both versions should produce the same output, although this can be compiler dependent. The C kernels form the platform for the development of the CUDA and OpenCL versions of the code, and the Fortran kernels form the basis of the OpenACC implementation.

OpenMP

The OpenMP implementation uses OpenMP pragmas to add loop-level parallelism at a kernel level, in both Fortran and C. The outer loop is distributed between threads, and the inner loop is vectorised where possible. Affinity is essential to ensure data locality, and this must be dealt with on a system-by-system basis. Task-based parallelism with OpenMP will be added in a future release.

OpenACC

The OpenMP implementation was taken as the basis for this version. The main differences are that the data needs to be transferred to an attached device, using extra pragmas, and the both the inner and outer loops are threaded. To achieve a boost in performance over the CPU host the algorithm needed to be fully resident on the attached accelerator and not used as a "co-processor". Data exchange only takes please when halo exchanges are required, a global reduction is needed (e.g. to find the minimum timestep when using multiple accelerators) or there is user request to output state data, for example for visualisation.

Currently only the Fortran version is available as an OpenACC version, using the Cray Compiler. Other implementations of OpenACC are in an early stage of investigation and differences in the how the standard is interpreted make a single source version hard to produce at this moment in time. These are expected to converge as the standard and compilers mature.

Without pragmas included, the OpenMP and kernels versions have identical source.

OpenCL

The OpenCL implmention is close to completion. It uses each loop in the C kernels as an OpenCL kernel. The mesh data and all computation is resident on the device as in the OpenACC version. The advantage of OpenCL is that it can be run equally easily on a CPU. The main difference to the C coding is the "boiler plate" coding required to transfer data to and from the device and share data between kernels.

CUDA

The CUDA uses the C kernels as the base in a similar fashion to the OpenCL. However the boiler plate code differs significantly.

MPI

CloverLeaf is distributed by partitioning the computational mesh into rectangular chunks. This decomposition attempts to minimise the surface area between computational chunks to minimise the communications overhead. Neighbouring chunks line up exactly. A halo two cells deep is added around the mesh to allow data from neighbouring mesh chunks to be made available. This data is updated during a call to the halo exchange, when requested data is sent via MPI buffers and inserted into halo cells. This halo exchange is required at a number of points during the computational cycle. During each exchange only the required data is communicated to the required depth to further reduce communications cost.

Currently no further optimisations of the MPI code has been investigated. Multipe data fields are sent in separate messages and not packed into one message. Also, sends are posted before receives even though it is generally accepted that posting receives first is likely to be more efficient.

There are alternative implementions to using the halo exchange using MPI. One is to use co-array Fortran which should allow overlap of computation and communication. The use of shmem should also allow similar overlap

Heterogeneous Implementations

With heterogeneous nodes containing a CPU and attached accelerator it is possible in theory to use all computational elements. Currently the only functioning heterogeneous method is when the hybrid MPI/OpenMP can be run on all elements. A tasked based OpenMP implementation should be able to add another method. OpenCL should also be able to all available devices, though issues such as load balance will become important when performance of compute elements differs significantly.

Current Status

The OpenMP implementation seems well optimised though it usually just lags the flat MPI version by a small amount on most systems.

The code seems to vectorise all loops except for the upwind/downwind phase of

the advection routines. A modified advection kernel will now vectorise but this usually performs worse than the scalar version due to increased floating point operations and loop logic.

The OpenACC version fully threads in all kernels and scope for further optimisations seem limited after a detailed profiling of the code and the removal of bottle necks.

Comparisons of the OpenCL and CUDA versions will be possible in the near future.

Despite the unoptimised MPI coding, CloverLeaf weak scales very well to the order of 10,000 cores. The major test will be as this increases to 100,000 and then 1,000,000 cores.

The ability of flat MPI to outperform hybrid codes still stands for this class of application. Whether this will change as core counts per node and node counts increase is one of the purposes of this mini-application.