Project 2: Parallelizing Fluid Simulations

1 Computational Fluid Dynamics

In this project you will parallelize a provided serial computational fluid dynamics code. This code solves the Navier-Stokes equations using standard numerical techniques that are suitable for solving turbulent flows. The particular model solves a benchmark problem that simulates the breakup and dissipation of a large eddy structure and is a problem that is used to evaluate the suitability of fluid solvers in the simulation of the dynamic details of turbulent flows. The benchmark case is the Taylor-Green vortex problem. Note, to parallelize this solver you do not need to know the specifics of the Navier-Stokes equations, or turbulence, or even the numerical methods used to solve them. You only will need to understand the structure of the code and how the algorithm manipulates data. For some background I will provide brief description of the equations being solved and the numerical solution methods that the solver employs. This section can be skipped but might provide some helpful background information for some students.

1.1 Solving the Navier-Stokes Equations

The Navier-Stokes are similar to Newton's Laws of motion for continuous media such as liquids and gases. One can think of the set of equations as one of a momentum balance of fluid particles. If we have wish to simulate a fluid we can do this by dividing space up into a number of individual cells where the change in the momentum of the fluid at any region of space is governed by changes in pressure or frictional forces between fluid elements caused by the viscosity of the fluid. The transfer of momentum through the motion of fluid particles is governed by so called inviscid terms of the Navier-Stokes equations and the frictional transfer of momentum is governed by the viscous terms. The state of the fluid element will be described by three variables: 1) the pressure, denoted p, 2) the x component of velocity denoted u, 3) the y component of velocity denoted by v, and 4) the z component of the velocity vector denoted by w. The evolution of these variables in time will be controlled by balancing the flows of momentum through these faces where the flow of momentum through the face is known as the momentum flux. The difference in the momentum flux will determine the rate of change of the momentum (velocity) of the fluid within the cell. Thus the difference between fluxes entering a cell and leaving the cell will result in a net change in momentum of the fluid that remains in the cell. The fluxes can be approximated numerically. For the solver used in this project, the computation of the flux at any one cell will require the information from two cells to the left of the face and two cells from the right. To find the change of momentum in any given cell will require the computation of all faces of the cell (6 for a 3-D cubic shaped cell), and as a result, the rate of change of the momentum within a cell will require access to a stencil of 12 neighboring cell

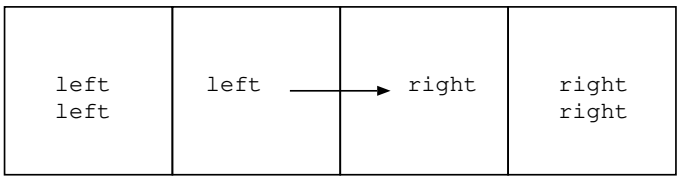

Figure 1: Cell Access for Face Flux Computation in 1-D

values in addition to itself. The data access for any given cell is illustrated in figure 1, and the resulting access for computing the rate of change for any given cell is given in figure 2.

Note that a flux represents a transport of momentum from one cell to the next, so fluxes are subtracted from one neighbor and added to the next. The total of the flux contributions to each cell provides is then used to determine how the cell state variables change in time. There are many methods that can be used to formulate an approximation of the flux that passes through a cell face. The details of how that happens will not be discussed here. Instead, you will be given a specific formulation that is implemented in the form of a serial program.

1.2 Developing the simulation program

The simulation program will simulate fluid in a periodic box. This means that the box's left most boundary will see an image of its right most boundary, the top boundary will see an image of its bottom boundary and so forth. The domain itself will be divided into a matrix of cells $ni \times nj \times nk$. To make room for a layer of two ghost cells needed to implement the periodic boundaries the overall systems of cells will be simulated in a box $(ni+4) \times (nj+4) \times (nk+4)$. The implementation of the algorithm will require a data structure that is a 3-D array where the size of the array is determined by user input. Since C and C++ don't provide general purpose run time sized multidimensional array facilities, we will create this array by explicitly computing the transformation of a triplet of array indices to a one dimensional array index. So we allocate a single dimensional array of size $(ni + 4) * (nj + 4) * (nk + 4)$ and then we can convert (i, j, k) array locations to the one dimensional array index using the formula

$$indx = (i + 2) * (nj + 4) * (nk + 4) + (j + 2) * (nk + 4) + (k + 2) \qquad (1)$$

This gives a row major ordering consistent with C arrays where the k indices in the array are in consecutive memory locations. The plus 2 in the formula, for example the $(i + 2)$ is there so we can naturally loop over the simulated cells of the array using a loop from $i = 0 \cdots ni - 1$ while accessing $i = -1$ and $i = -2$ will access the ghost cells used to implement periodic boundary conditions. This

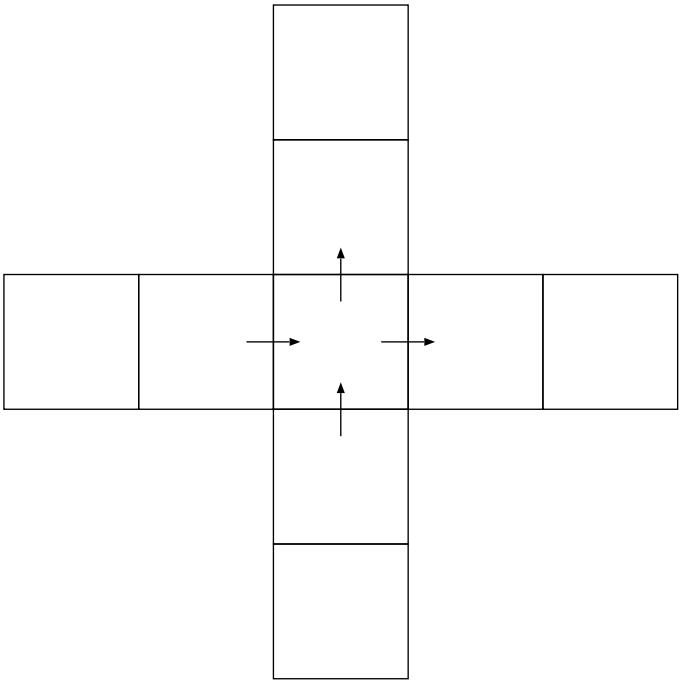

Figure 2: Stencil for cell residual computation in 2-D

index calculation is simplified to the following formula in the program:

$$indx = k_{start} + i * i_{skip} + j * j_{skip} + k * k_{skip} \tag{2}$$

where $k_{start} = 2 * i_{skip} + 2 * j_{skip} + 2 * k_{skip}$, $i_{skip} = (nj + 4) * (nk + 4)$, $j_{skip} = (nk+4)$, and $k_{skip} = 1$. Using this notation we can also write offset from the current index such that if $indx = (i, j, k)$ then $indx + n * j_{skip} = (i, j + n, k)$ and so on. This technique will be used to iterate over the mesh of fluid elements in the program.

1.3 Major Procedures

```
void setInitialConditions(float *p, float *u, float *v, float *w,
  int ni, int nj, int nk, int kstart,
  int iskip, int jskip,float L) ;
```

This procedure initializes the initial state of the fluid variables, p, u, v, and w. The initial conditions are set up to solve the 3-D Taylor-Green vortex problem. Note, due to first touch policies, this loop will likely determine the ownership memory pages to cores. From a performance point of view, it may be useful that the way the loops are partitioned here match the major computational loops that follow.

```
void copyPeriodic(float *p, float *u, float *v, float *w,
  int ni, int nj, int nk , int kstart,
                  int iskip, int jskip) ;
```

This procedure implements the periodic boundary conditions by copying data into the ghost cells from the opposing periodic face. There are three loops in the routine that copies x axis facing boundaries, y axis facing boundaries, and z axis facing boundaries.

```
void zeroResidual(float *presid, float *uresid, float *vresid,
                  float *wresid,
  int ni, int nj, int nk ,
                  int kstart, int iskip, int jskip) ;
```

This routine zeros out the residual variables where the fluxes will be summed. Since the fluxes are accumulated, they need to be initialized to zero to get consistent results. Note, that due to first touch policies, this routine is likely to set thread ownership of memory, similar to the initial conditions. The best performance will probably be achieved if the partition of these loops to threads matches the partition of the computationally intensive loops that follow it.

```
void computeResidual(float *presid,
                     float *uresid, float *vresid, float *wresid,
     const float *p,
     const float *u, const float *v, const float *w,
```

```
    float eta, float nu, float dx, float dy, float dz,
    int ni, int nj, int nk, int kstart,
    int iskip, int jskip) ;
```

This routine accumulates the momentum fluxes to the cells using the inviscid and viscous fluxes. Internally the routine does this by executing three loops for the x axis oriented faces (i dimension), the y axis oriented faces (j dimension) and the z axis oriented face (k dimension). Note that in this routine each flux is summed to two cells (Specifically subtracted from one and added to the other). As a result, when the loops are parallelized it is possible to create race conditions if one is not careful when parallelizing using *OpenMP*.

```
float
computeStableTimestep(const float *u, const float *v, const float *w,
    float cfl, float eta, float nu,
    float dx, float dy, float dz,
    int ni, int nj, int nk, int kstart,
    int iskip, int jskip) ;
```

This routine computes the largest possible time-step that will produce a stable solution for each cell. The time-step that the simulation uses will be the minimum of these time-steps. Please note that the loop in this subroutine is performing a reduction and will require appropriate *OpenMP* annotations.

```
float
integrateKineticEnergy(const float *u, const float *v, const float *w,
                       float dx, float dy, float dz,
       int ni, int nj, int nk, int kstart,
       int iskip, int jskip) ;
```

This routine computes the integrated kinetic energy of the fluid system. This is performed by summing up the integrated kinetic energy of the fluid in each cell. The change of the fluid kinetic energy over time is an important parameter of this benchmark problem. Please note that the loop in this routine is performing a reduction and *OpenMP* parallelization will need to annotate the loop appropriately.

```
void weightedSum3(float *uout, float w1, const float *u1, float w2,
  const float *u2, float w3,
  int ni, int nj, int nk, int kstart,
  int iskip, int jskip) ;
```

This procedure computes the weighted sum of three arrays. It is used to implement the Runge-Kutta time integration procedure used in the solver.

These routines are used to evolve the state of the fluid in time. The basic procedure is to start the time integration by computing the stable time-step. The time integration procedure involves evaluating the residual functions at

three different solution estimations. To compute the residual, first the periodic boundary conditions are enforced, then the residual is zeroed out, and then the residual is computed by computing and summing fluxes for the six faces of each cubic cell within the fluid mesh. The solution is then updated using a weighted sum of previous solutions and the residual function. Once the final update is computed, the simulation time is advance and important data is collected, and the iteration repeats until the target time is achieved.

2 Parallel Programming Project

In this project you are to use *OpenMP* directives to parallelize the provided fluid solver. When parallelizing the code you should get the exact same results. The number of time-steps, the final simulation time, and the reported kinetic energy should be the same to every digit. If your parallel program obtains a result that is almost the same, then this is an indication that you have a race condition and thus your program is incorrect. Thus it is important that you develop your parallel program incrementally so that you can understand which directive created the race condition.

The parallelization of the code can be mostly achieved by using parallel for directives to parallelize existing loops. However, special care must be taken to make sure that race conditions are avoided. In particular when summing fluxes to cells in the loops of the `computeResidual` procedure, there may be overlapping reads and writes as the fluxes are added to the residual vector. You will need to determine a technique that either partitions threads such that these conflicts do not occur, or utilize atomics or other locking mechanisms to make sure that the race conditions are avoided. One approach is to carefully divide the loops such that parallel threads do not perform overlapping writes by careful partitioning of the loops. In general, avoiding race conditions for this routine without significant performance penalties can be a challenge.

Another potential source for race conditions will be in the procedure that computes the stable time-step, `computeStableTimestep`, and the procedure that computes the total system kinetic energy, `integrateKineticEnergy`. In the first procedure the stable time-step is the minimum over each cell stable time-step, and in the second the kinetic energy contribution from each cell is accumulated to a single value. Both of these operations require combining many values into a single value. Naively applying parallel loop directives to these loops will result in unexpected results.

Provided in the directory are scripts `run##.js` which will run larger problems on the cluster for extracting performance data ranging from 1 to 64 threads. Note that there are 64 physical cores on a node of the ptolemy cluster. From these scripts you should be able to compute the speedup and efficiency of your *OpenMP* implementation. In general, it should be possible to get reasonably good speedups. You should be able to get a parallel efficiency of better than 50% when running with 16 threads, with diminishing returns for larger number of threads.

As with the previous project, document your results in a formal report that will be submitted with your program. The same grading will rubric will be used as was utilized in the previous project.

3 Graduate Credit Work

If you are taking this course for graduate credit, then in addition to parallelizing the application with *OpenMP* you will need to use *MPI* in a hybrid fashion. Meaning that after you develop the *OpenMP* part of the project, you will need to develop an *MPI* version. The easiest way to accomplish this is to modify the program to distribute the three dimensional array across processors. Perhaps the easiest ways to do this would be to partition the `i` dimension of the array by dividing it evenly between processors. For example, when running a 64×64×64 problem on 4 *MPI* processes, each processor would be assigned 16 × 64 × 64 sub-block. Then the `copyPeriodic` code can be modified to use MPI `Sendrecv` or similar to exchange `(j, k)` planes between processors in a ring like structure. Additionally code that writes to file will need to be modified to write only from a master processor, and code that uses reductions will need to implement *MPI* reduction operations. The resulting code will be parallelized using a mixture of distributed memory and thread based parallelism. Since this implementation utilizes hybrid parallelism that will have different sources of parallel overhead, the execution behavior of the hybrid programming model can be evaluated by utilizing the 64 cores with a mixture of thread and distributed memory parallelism. The `runmpi#t#p.js` files are provided to run the distributed memory tests to explore different degrees of thread an distributed memory parallelism that exploits all 64 cores of the node.