Project 2: Parallel Fluid Dynamics Simulation

John Bradley

April 12, 2024

1 Introduction

This report aims to document the implementation and analysis of a program that utilizes OpenMP to implement a computational fluid dynamics model suitable for modeling behavior of turbulent flows. This report will cover the methods and techniques utilized for implementing the fluid dynamics model using OpenMP, analyze and estimate the expected performance of the implementation, present the results from experimentation, compare and contrast the expected performance to the observed results, and conclude with insights and possible changes that could be made to the implementation

2 Methods and Techniques

Many of the functions in the code provided deal with sequential sections of memory, and allowing a single thread to access these sequential memory areas would be beneficial. Additionally, there are many nested loops. The Ptolemy system does not have `OMP_NESTED` set by default, thus the use of nested `#pragma omp for` declarations should be avoided. To do so, careful selection of partitioning of parallel and serial sections need to be made to exploit sequential memory access.

For example, in the `setInitialCOnditions` function, the inner-most nested loop contains:

```
for(int k=0; k<nk; ++k) {
  int indx = offset + k;
  float dz = (1./nk)*L;
  float z = 0.5*dz+k*dz - 0.5*L;

  u[indx] = 1.*coef*sin(x/l)*cos(y/l)*cos(z/l);
  v[indx] = -1.*coef*cos(x/l)*sin(y/l)*cos(z/l);
  p[indx] = (1./16.)*coef*coef*(cos(2.*x/l)+cos(2.*y/l))
     *(cos(2.*z/l)+2.);
  w[indx] = 0;
}
```

For each iteration of the loop, `indx` is the sum of `offset`, defined in the outer loop, and the value of `k`, which is incremented each iteration of the loop. If the loop was parallelized, each thread may access non-sequential sections of the `u`, `v`, `p`, and `w` arrays. If we, instead, parallelize the loop that contains the "`k`" loop, then we can exploit this memory access pattern. This effect is more pronounced the further out this parallelization is declared. Similarily, this pattern of access is present across many of the functions in the program, and can be exploited in the same way.

However, this naïve approach isn't applicable to all situations. For example, in `computeResidual`, the residual flux vectors (e.g. `presid`, `uresid`, `vresid`, and `wresid`) may have overlapping read/write access, and the variable will likely need to be locked or careful partitioning of the loops be performed. In this project, atomization will be used, or, in cases where there are multiple variables, `omp_set_lock`.

```
// openMP lock
omp_lock_t lock;
omp_init_lock(&lock);

#pragma omp parallel
{
#pragma omp for
for(int i=0; i<ni+1; ++i) {
  const float vcoef = nu/dx;
  const float area = dy*dz;
  for(int j=0; j<nj; ++j) {
    int offset = kstart+i*iskip+j*jskip;
    for(int k=0; k<nk; ++k) {

...

    omp_set_lock(&lock);
    presid[indx-iskip] -= pflux;
    presid[indx]       += pflux;
    uresid[indx-iskip] -= uflux;
    uresid[indx]       += uflux;
    vresid[indx-iskip] -= vflux;
    vresid[indx]       += vflux;
    wresid[indx-iskip] -= wflux;
    wresid[indx]       += wflux;
    omp_unset_lock(&lock);

...
```

This ensure that more than one thread updates the variable at the same time to avoid a race condition. However, this will likely result in a performance

impact as the entire block is prevented from being entered by another loop. Looking closer at the `offset` value and the two indexes for each component of the vector, the index of each vector's array is dependent on `k` and the skip direction (`i`, `j`, `k`). This this dependent direction isn't parallelized, this should avoid a race condition without having to use variable locking. A little bit of loop restructuring can help with this as well.

```
#pragma omp parallel
{
  #pragma omp for
  for(int j=0; j<nj; ++j) {
    const float vcoef = nu/dx;
    const float area = dy*dz;
    for(int i=0; i<ni+1; ++i) {
      int offset = kstart+i*iskip+j*jskip;
      for(int k=0; k<nk; ++k) {
        const int indx = k+offset;

...
```

Simply swapping the `i` and `j` loops in the first vector calculation kept the integrity of the data while eliminating the need for variable locks. The second and third calculation in `computeResidual` do not need to have the loops reordered as they are not parallelizing two index locations next to each other.

Finally, some functions, such as `computeStableTimestep`, requires that a minimum be reduced from the parallelization portion. In C++, if this was a mathematical operation, such as summation or subtraction (such as in `integrateKineticEnergy`), this would be a trivial task using a built-in reduction clause. However, it is not. To prevent a race condition, a minimum local to the thread must be calculated, and once that is done, a critical section is added so that only one thread does a global (or team) comparison to set the final minimum value.

```
float minDt = 1e30;
#pragma omp parallel
{
  float local_minDt = 1e30;
  #pragma omp for nowait
  for(int i=0; i<ni; ++i) {
    for(int j=0; j<nj; ++j) {
      int offset = kstart+i*iskip+j*jskip;
      for(int k=0; k<nk; ++k) {
        const int indx = k+offset;

        // inviscid timestep
```

```
            const float maxu2 = max(u[indx]*u[indx],max(v[
                indx]*v[indx],w[indx]*w[indx]));
            const float af = sqrt(maxu2+eta);
            const float maxev = sqrt(maxu2)+af;
            const float sum = maxev*(1./dx+1./dy+1./dz);
            local_minDt=min(local_minDt,cfl/sum);

            // viscous stable timestep
            const float dist = min(dx,min(dy,dz));
            local_minDt=min<float>(local_minDt,0.2*cfl*dist*
                dist/nu);
          }
        }
      }

      #pragma omp critical
      {
        minDT=min(minDT,local_minDt);
      }
    }
```

The `nowait` clause was used so that the threads do not wait for other threads to reach the critical section before performing it.

Further parallelization could be done, with each parallelized function being further parallelized using MPI, however this is outside of the scope of the undergraduate assignment.

3 Analysis

Very little of the code is spent in serial execution, as all of the functions were able to be parallelized to a great extent. A generious extimation of approximately 90% of the execution time being parallelized gives us the serial fraction of:

$$f = \frac{W_S}{W_S + \sum W_K} = \frac{0.1}{1} = 0.1$$

based on the assumption that $t_1 = 1$. The upper bound of the speedup is:

$$S_{upper} = \frac{1}{f} = \frac{1}{0.1} = 10$$

The upper bound can be compared to the estimated speedup for the number of procesors that will be used using the following equations:

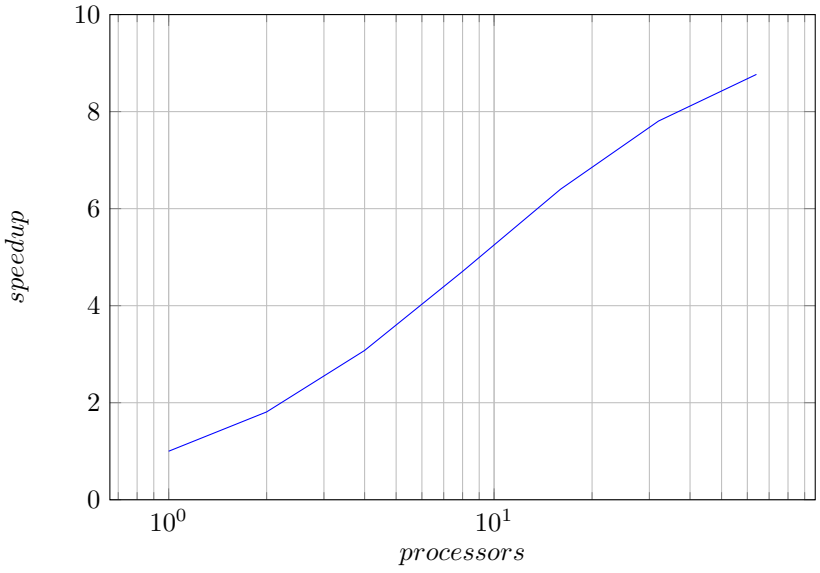
$$t_p = \left[f + \frac{(1-f)}{p} \right] \times t_1$$

$$S = \frac{t_1}{t_p}$$

Proc	Speedup
1	1
2	$1.\overline{81}$
4	3.077
8	4.706
16	6.4
32	7.805
64	8.767

Table 1: Estimated Speedup

Figure 1: Estimated Speedup



The estimated speedups are listed in Table 1 and plotted in Figure 1.

4 Results

5 Synthesis

6 Conclusion