Project 2: Parallel Fluid Dynamics Simulation

John Bradley

April 14, 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, as it would prevent cache contention between threads. Additionally, there are many nested loops, and the Ptolemy system does not have `OMP_NESTED` set by default, thus the use of nested `#pragma omp for` declarations should be avoided, as enabling nested parallel sections may not be beneficial. Careful partitioning of parallel sections will be needed.

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. Furthermore, this prevents threads from trying to access memory locations that may be in the same cache line.

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`) are calculated using loops that may have overlapping read/write access.

Creating a critical, or locked section, can ensure that more than one thread does not update the variables at the same time, thus avoiding a race condition. However, this will likely result in a performance impact as the entire block is prevented from being entered by another loop. In fact, early testing during development of the code showed that this does create a significant performance impact.

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 * iskip`, `j * jskip`, `k * kskip`). This is true for the three loops calculating the vectors in each direction. Furthermore, the section of each loop where the vectors are accessed includes one of the three skip directions.

```
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;
```

Ensuring that the outer most loop isn't incrementing the direction that is being callculatd should be sufficient to prevent a race condition.

```
#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;

        // calculations

      }
    }
  }
}
```

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 the direction of the skip variable is the same as the inner two loops.

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.

The final code as written will be ran a total of 10 times each over a varying number of OpenMP threads. The results will be collected, and the parallel runs compared against the single-threaded runs to ensure race conditions did not occur (i.e. reproduceability).

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 estimation 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

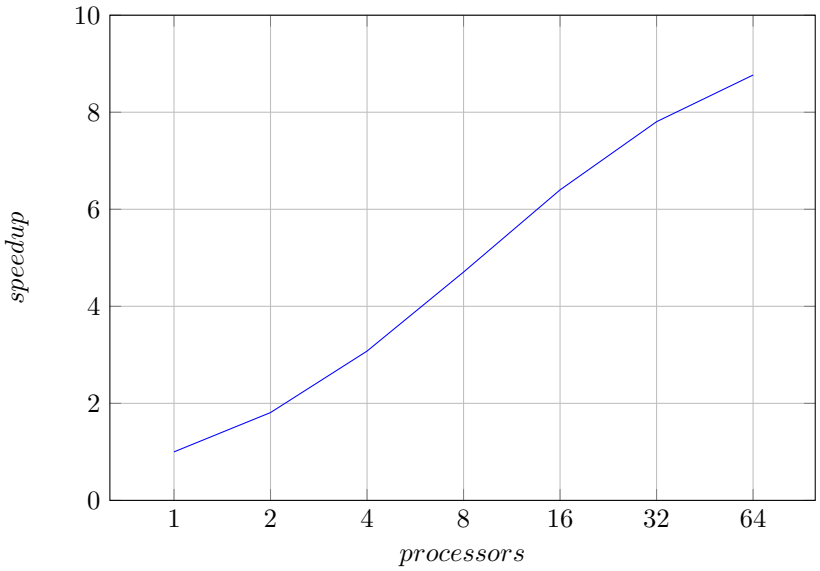

Figure 1: Estimated Speedup

The estimated speedups are listed in Table 1 and plotted in Figure 1. Communication and parallelization overhead is not deeply considered as much of the time each of each thread should be spent within nested loops.

4 Results

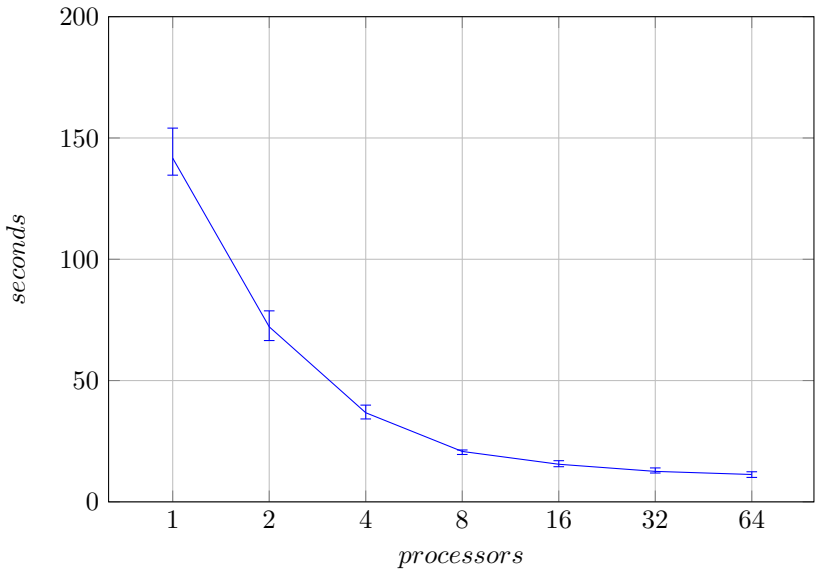

Figure 2: Mean Execution Time

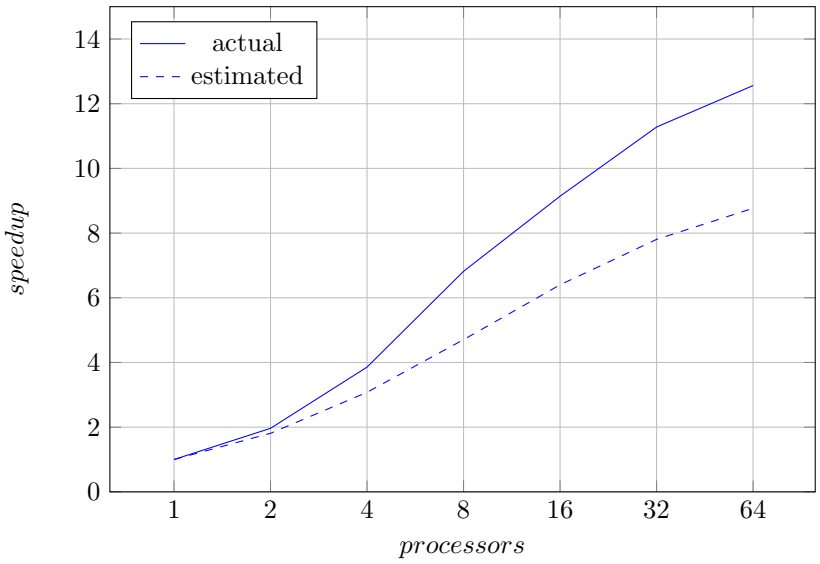

Figure 3: Actual vs Estimated Speedup

Figure 2 shows mean execution time of the 10 runs for each count of OpenMP threads. Speedup in Figure 3 is calculated using the mean execution time as t_1 and t_p (respectively for both serial and parallel values).

A `diff` performed between the single-threaded runs and every subsequent run with greater processor values confirmed that no race-conditions existed.

5 Synthesis

Speedup, as shown in Figure 3 exceeded what was originally expected using the estimation that 90% of the code was parallelized. For the 64 processor run, the actual speedup was:

$$S_{actual} = \frac{t_1}{t_6 4} = \frac{141.693}{11.280} = 12.561$$

This exceeds both the estimated speedup of 8.787 and S_{upper} of 10. Using the actual speedup, the calculated serial fraction is:

$$f = \frac{\frac{p}{S_{actual}} - 1}{p - 1} = \frac{\frac{64}{12.561} - 1}{64 - 1} = 0.065$$

A serial raction of 6.5% is not unreasonable, as the originally estimated fraction was 10%.

At around 8 processors, the reduction in execution time begins to taper off, and ultimately by 32 processors, the speedup begins to reach an upper limit.

6 Conclusion

The results are entirely satisfactory and within reason. The initial estimate was fairly conservative, stating that 90% of the code was parallelizable. The actual portion may have been marginally greater, or cache effects may have contributed to the additional speedup observed, offsetting any potential overhead from implementing OpenMP.

The results also indicate that the methodologies to avoid performance penalties by declaring a critical, or atomized, section within the `computeResidual` function was correct. This allowed the program to actually achieve (and exceed) expected speedup.

Finally, the strategy of parallelizing the outermost loops where possible, and in the case of `computeResidual`, reordering the order of loops performed, successfully prevented a race condition.