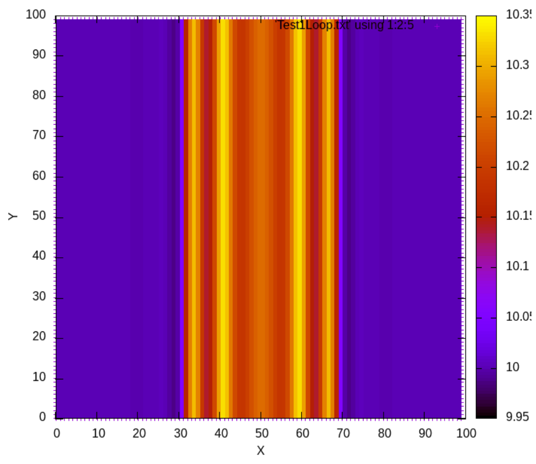
High Performance Computing (AERO 70011)

Parallel Computing Report

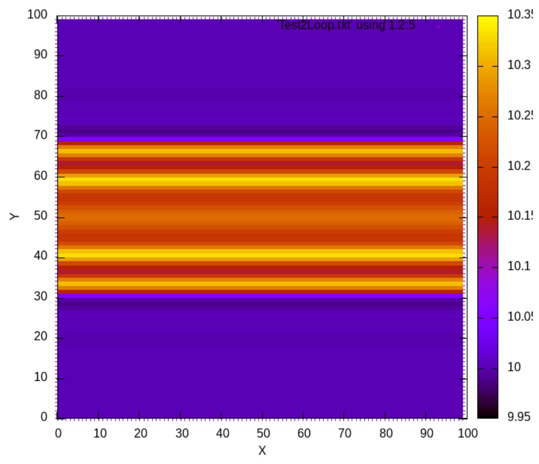
by Nicholas Yap (01703465)

August 29, 2023

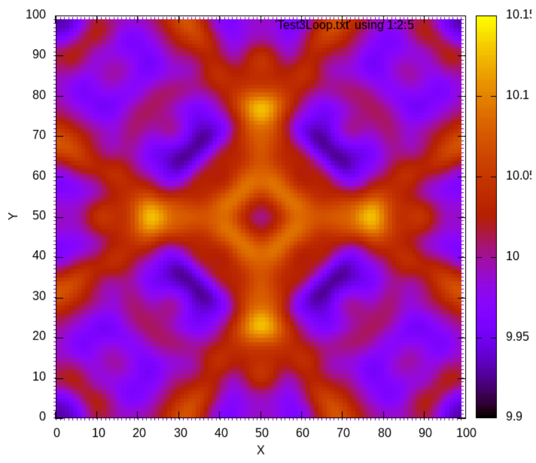
1 Question 1- Simulation Results



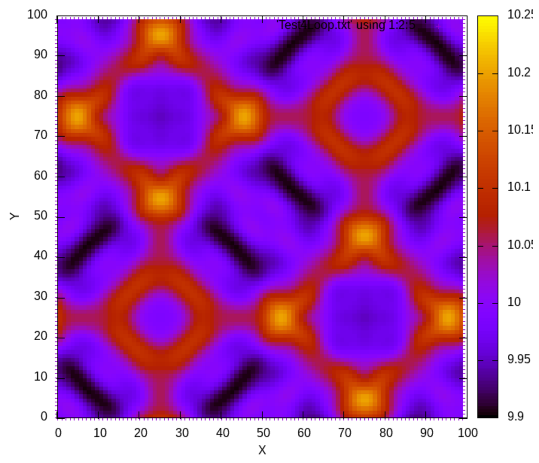
(a) Test Case 1



(b) Test Case 2



(c) Test Case 3



(d) Test Case 4

Figure 1: 2D Contour Plots

2 Question 2- Optimisation Approach

Some steps were taken to optimise both the serial and parallel performance of the code. The loop-based evaluation is conducted using two nested loops to loop through the rows and columns of each property.
The steps taken to optimise the code in series include:

- Utilising multipliers wherever possible instead of division operators, as C++ reads multiplications more simply.
- Whenever possible a `const` was used when declaring constants
- Reducing computation by assigning exact values to constants
- Applying the product rule to the derivatives $\frac{dhu}{dx}$ and $\frac{dhv}{dy}$ to reduce the number of call to derivatives and function and eliminate the need to store more matrices

After implementing the code in series, multiple optimisation steps were taken to further optimise the code. These are as follows:

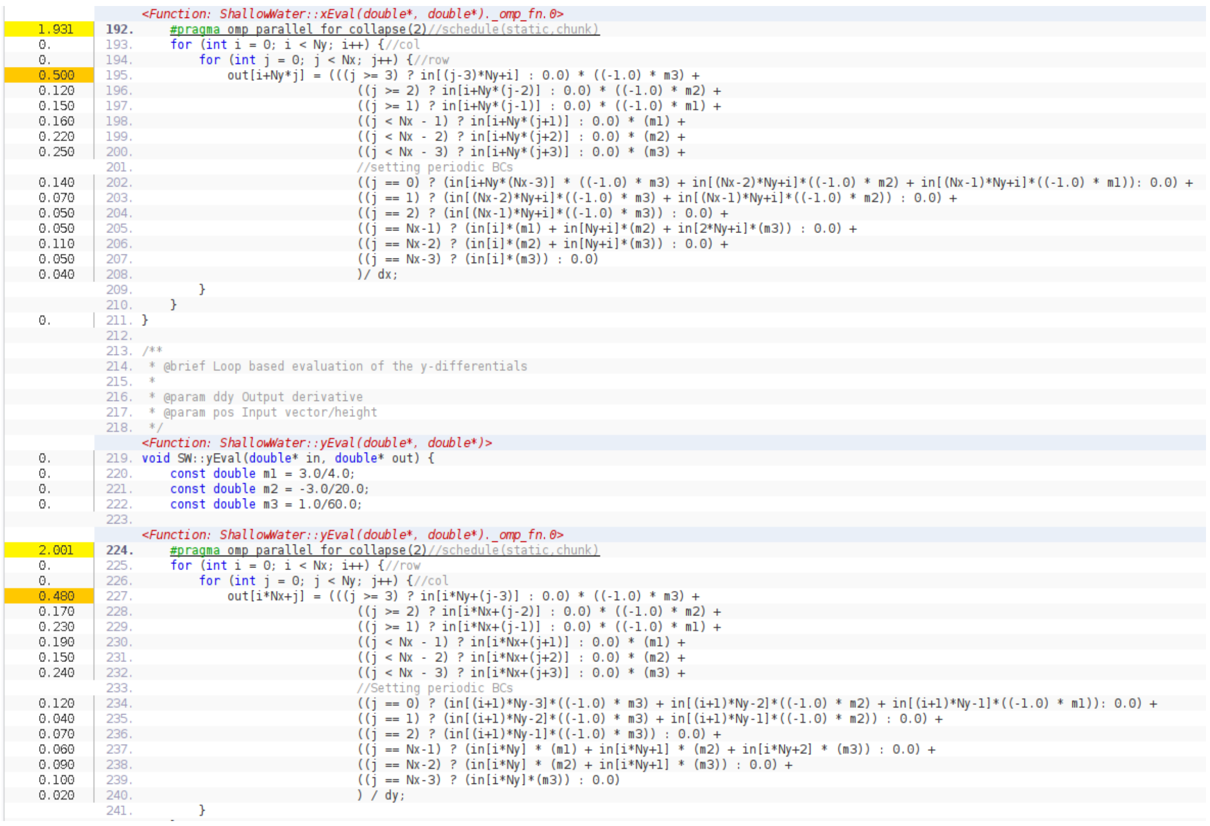
- Adding the highest level of optimisation, `-O3`, to make the compiler improve the performance of the code
- Adding clauses to OpenMP directives

The runtime after implementing these improvements are documented in Table 1. These runtimes were averaged from 10 runs of the code using test case 4, and the number of threads used was set to 8 for the testing (`--np = 8`).

Optimisation	Loop(s)	Time Reduction (%)	BLAS(s)	Time Reduction(%)
Initial Implementation	5.096	-	8.616	-
Parallelisation	1.192	76.6	1.814	79.0
`schedule` clause	1.132	5.0	1.939	6.9
`nowait` clause	1.089	3.8	1.890	2.5
Adding -O3 flag	1.084	0.5	1.847	2.3
`collapse` clause	0.710	34.5	-	-

Table 1: Runtimes for Loop and BLAS based approaches.

It is noted that most of the improvements were made to the loop application of the code, and this showed in the profiler, where the `cblas_dgemm` calls added up to about 5 times the computing time of the loop based calculations.

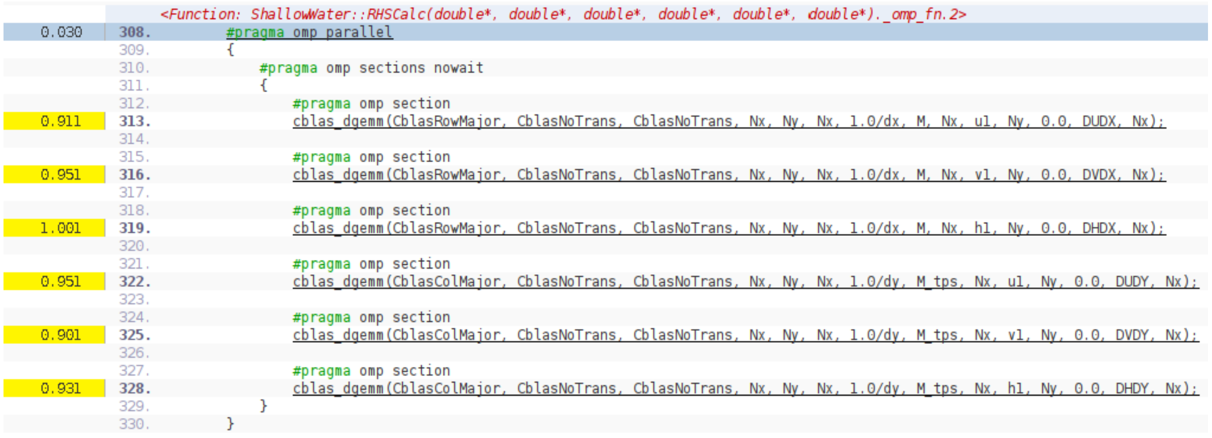


```
                <Function: ShallowWater::xEval(double*, double*)._omp_fn.0>
1.931       #pragma omp parallel for collapse(2)//schedule(static,chunk)
0.          for (int i = 0; i < Ny; i++) {//col
0.              for (int j = 0; j < Nx; j++) {//row
0.500               out[i+Ny*j] = (((j >= 3) ? in[(j-3)*Ny+i] : 0.0) * ((-1.0) * m3) +
0.120                               ((j >= 2) ? in[i+Ny*(j-2)] : 0.0) * ((-1.0) * m2) +
0.150                               ((j >= 1) ? in[i+Ny*(j-1)] : 0.0) * ((-1.0) * m1) +
0.160                               ((j < Nx - 1) ? in[i+Ny*(j+1)] : 0.0) * (m1) +
0.220                               ((j < Nx - 2) ? in[i+Ny*(j+2)] : 0.0) * (m2) +
0.250                               ((j < Nx - 3) ? in[i+Ny*(j+3)] : 0.0) * (m3) +
                                    //setting periodic BCs
0.140                               ((j == 0) ? (in[i+Ny*(Nx-3)] * ((-1.0) * m3) + in[(Nx-2)*Ny+i]*((-1.0) * m2) + in[(Nx-1)*Ny+i]*((-1.0) * m1)): 0.0) +
0.070                               ((j == 1) ? (in[(Nx-2)*Ny+i]*((-1.0) * m3) + in[(Nx-1)*Ny+i]*((-1.0) * m2)) : 0.0) +
0.050                               ((j == 2) ? (in[(Nx-1)*Ny+i]*((-1.0) * m3)) : 0.0) +
0.050                               ((j == Nx-1) ? (in[i]*(m1) + in[Ny+i]*(m2) + in[2*Ny+i]*(m3)) : 0.0) +
0.110                               ((j == Nx-2) ? (in[i]*(m2) + in[Ny+i]*(m3)) : 0.0) +
0.050                               ((j == Nx-3) ? (in[i]*(m3)) : 0.0)
0.040                               )/ dx;
                }
            }
0.      }

        /**
         * @brief Loop based evaluation of the y-differentials
         *
         * @param ddy Output derivative
         * @param pos Input vector/height
         */
                <Function: ShallowWater::yEval(double*, double*)>
0.      void SW::yEval(double* in, double* out) {
0.          const double m1 = 3.0/4.0;
0.          const double m2 = -3.0/20.0;
0.          const double m3 = 1.0/60.0;

                <Function: ShallowWater::yEval(double*, double*)._omp_fn.0>
2.001       #pragma omp parallel for collapse(2)//schedule(static,chunk)
0.          for (int i = 0; i < Nx; i++) {//row
0.              for (int j = 0; j < Ny; j++) {//col
0.480               out[i*Nx+j] = (((j >= 3) ? in[i*Ny+(j-3)] : 0.0) * ((-1.0) * m3) +
0.170                               ((j >= 2) ? in[i*Nx+(j-2)] : 0.0) * ((-1.0) * m2) +
0.230                               ((j >= 1) ? in[i*Nx+(j-1)] : 0.0) * ((-1.0) * m1) +
0.190                               ((j < Nx - 1) ? in[i*Nx+(j+1)] : 0.0) * (m1) +
0.150                               ((j < Nx - 2) ? in[i*Nx+(j+2)] : 0.0) * (m2) +
0.240                               ((j < Nx - 3) ? in[i*Nx+(j+3)] : 0.0) * (m3) +
                                    //Setting periodic BCs
0.120                               ((j == 0) ? (in[(i+1)*Ny-3]*((-1.0) * m3) + in[(i+1)*Ny-2]*((-1.0) * m2) + in[(i+1)*Ny-1]*((-1.0) * m1)): 0.0) +
0.040                               ((j == 1) ? (in[(i+1)*Ny-2]*((-1.0) * m3) + in[(i+1)*Ny-1]*((-1.0) * m2)) : 0.0) +
0.070                               ((j == 2) ? (in[(i+1)*Ny-1]*((-1.0) * m3)) : 0.0) +
0.060                               ((j == Nx-1) ? (in[i*Ny] * (m1) + in[i*Ny+1] * (m2) + in[i*Ny+2] * (m3)) : 0.0) +
0.090                               ((j == Nx-2) ? (in[i*Ny] * (m2) + in[i*Ny+1] * (m3)) : 0.0) +
0.100                               ((j == Nx-3) ? (in[i*Ny]*(m3)) : 0.0)
0.020                               ) / dy;
                }
```

Figure 2: Profiler for Loop section



```
                <Function: ShallowWater::RHSCalc(double*, double*, double*, double*, double*, double*)._omp_fn.2>
0.030           #pragma omp parallel
                {
                    #pragma omp sections nowait
                    {
                        #pragma omp section
0.911                   cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, Nx, Ny, Nx, 1.0/dx, M, Nx, u1, Ny, 0.0, DUDX, Nx);

                        #pragma omp section
0.951                   cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, Nx, Ny, Nx, 1.0/dx, M, Nx, v1, Ny, 0.0, DVDX, Nx);

                        #pragma omp section
1.001                   cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, Nx, Ny, Nx, 1.0/dx, M, Nx, h1, Ny, 0.0, DHDX, Nx);

                        #pragma omp section
0.951                   cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, Nx, Ny, Nx, 1.0/dy, M_tps, Nx, u1, Ny, 0.0, DUDY, Nx);

                        #pragma omp section
0.901                   cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, Nx, Ny, Nx, 1.0/dy, M_tps, Nx, v1, Ny, 0.0, DVDY, Nx);

                        #pragma omp section
0.931                   cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, Nx, Ny, Nx, 1.0/dy, M_tps, Nx, h1, Ny, 0.0, DHDY, Nx);
                    }
                }
```

Figure 3: Profiler for BLAS section

3 Question 3- Loop or BLAS

To evaluate spatial derivatives for the shallow water equations, both loop and BLAS based calculations were used to evaluate their effectiveness in computing the task at hand. This was done using the `--calc(default == 1)` arg in the command line, where the user is able to select a `loop(1)` or `BLAS(2)` based approach.
As noted in Table 1, the loop based approach was overall faster than the BLAS based approaches, even after parallelising. This is because the loop based calculations allow for a more efficient workload distribution across threads. The BLAS based approaches may also be slower due to the nature of the task, which is not exactly a linear algebra problem. Moreover, the overhead of calling the BLAS library alongside its need to multiply many values by 0 may lead to it being more inefficient. These

conclusions are also backed up by Figures 3 and 2, where one `cblas_dgemm` call is equivalent to both loops combined.

4 Question 4- Parallelisation Approach

Two options for parallelism were presented, being OpenMP and MPI(Messing Passing Interface). OpenMP is an API used to add multi-threaded shared memory parallelism to applications. MPI on the other hand uses message passing to parallelise. The objective of parallelisation is to reduce runtime by splitting workload across multiple threads of the CPU.
For this application, OpenMP was preferred, as the complexity of the tasks could impose some latency on the communication, leading to deadlocks. Thus, OpenMP was the chosen form of parallelisation. The main benefit of MPI is also that it is able to run on multiple machines, which was not a requirement for this assignment. . Open MP is also used due to its suitability for the problem, as the main computational hurdle is at the runge-kutta. It was found as well that OpenMP worked well with the loop based calculations, making it much faster than the BLAS based RHS calculations.
The directive `#pragma omp parallel` was used. These directives are called mainly in the functions `TimeIntegrate` and `RHSCalc`, which also calls them in the `xEval` and `yEval` functions. Parallel section constructs and parallel loop constructs are used.
The `#pragma omp parallel for` directive allows for a subset of rows or columns to be assigned to each thread. This form of parallelisation was used in the `TimeIntegrate` function to calculate the k-values for the runge-kutta 4. It was discovered however, that without any clauses, adding parallelisation to this section of the code resulted in no improvements, even increasing runtime by 2.5% on average. This directive was also used in the `xEval` and `yEval` functions which uses loops to evaluate the spatial derivatives.
To parallelise the BLAS section of the `RHSCalc` function, the `#pragma omp sections` directive was used to parallelise this section of the code.
As discussed in Section 2, clauses were used in line with the `#pragma omp parallel` directives. The `schedule(static,chunk)` clause was placed with the for loops in all functions initially. This clause split the workload into roughly equal chunks, and the static clause assigns each thread to a specific chunk. While the threads may idle when finishing its workload, each iteration of the loop consists of the same operations, so workload per iteration is mostly consistent.
As the `#pragma omp sections` directive was used a number of times, the `nowait` clause was used in this setting. This clause was used to assign workload in a round-robin system. This clause suggests that each node is updated by each thread with no implicit synchronisation to boost performance.
Finally, the `collapse(2)` clause was tested to be implemented alongside the nested loops in the `xEval` and `yEval` functions to evaluate the spatial derivatives. It was noted that, while the collapse clause was added only to the loop based solver, it was the most effective clause as shown in Table 1

5 Question 5- Parallel Scaling

By implementing parallelisation using OpenMP, it is worthwhile investigating the effect of increasing the number of threads and its effect on the runtime of the code, and hence its performance.
The effectiveness of the parallelisation was measured using the runtime of the code using different numbers of threads. This was timed using `chrono::steady_clock::time_point begin = chrono::steady_clock::now()` and `chrono::steady_clock::time_point end = chrono::steady_clock::now()`. These runtimes were then verified using the *Oracle Developer Studio*. The test was run on test case 4, where 10 tests were conducted and their results averaged. These tests were conducted on both loop based and BLAS based calculations. These plots are plotted in Figures 4 and 5. It can be noted that neither plot shows exact inverse proportionality, which is to be expected, as the overhead to split the workload between the threads increases with the number of threads leading to decreased performance.
For the loop based calculations, as shown in Figure 4, the code showed a decrease in runtime until 9 threads (`--np = 9`), with an average runtime of 0.724s. However, past 10 threads, the overhead due to the increased parallelisation catches up with the split workload, causing the code to slow down drastically. A similar trend is seen with the BLAS method, as shown in Figure 5, where 7 threads (`--np = 7`) showed the lowest runtime of 1.821s. It is noted however that the `collapse` clause, as

discussed in Section 4, had the biggest effect of any clause, and was not applied in the BLAS applications.

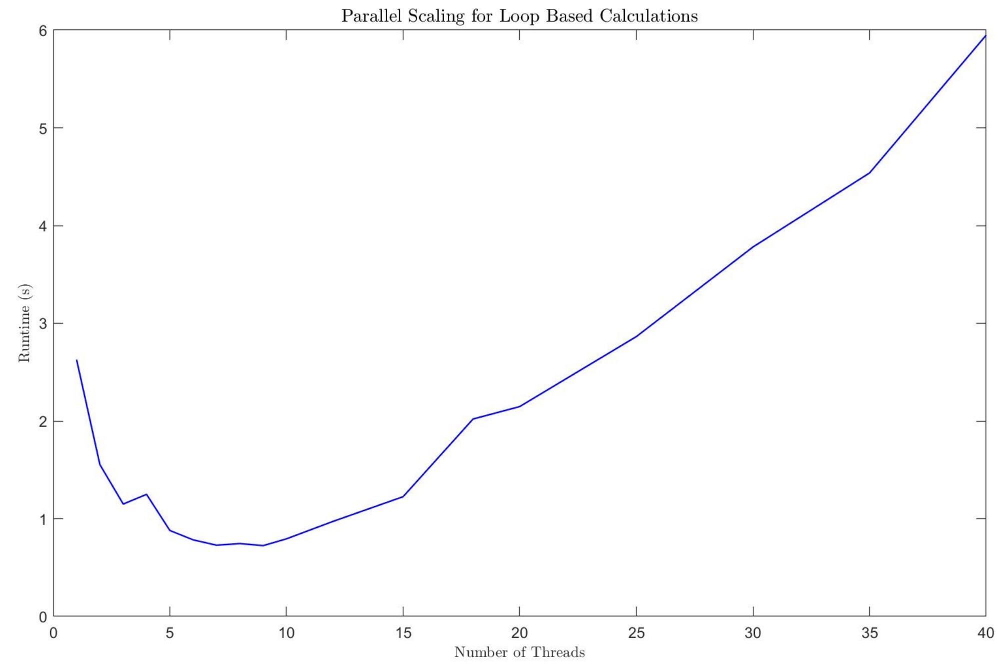


Figure 4: Loop scaling reached its lowest average runtime at `--np = 9`

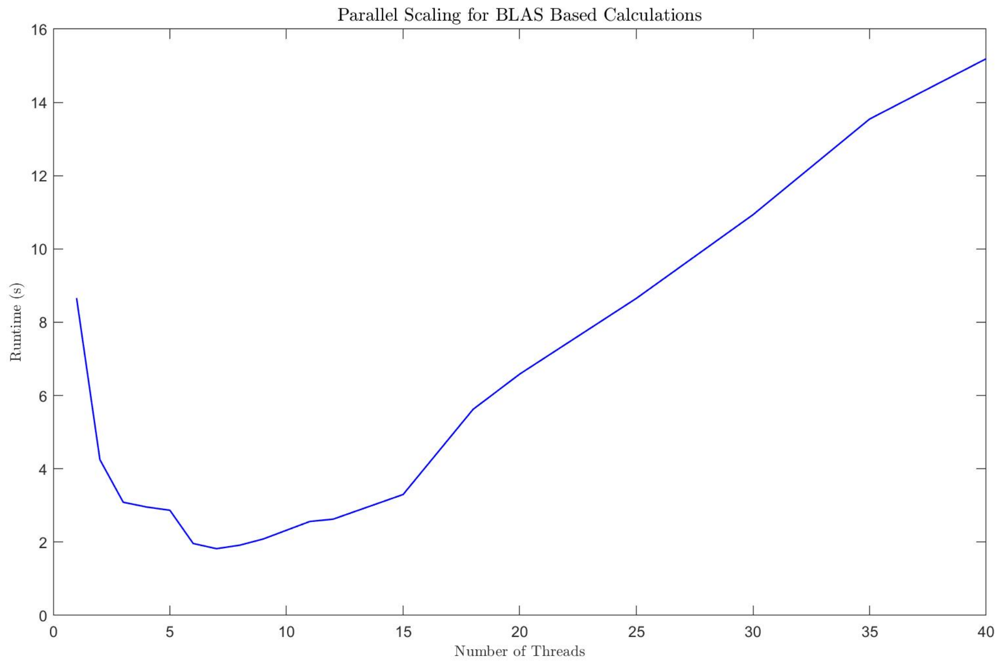


Figure 5: BLAS scalling recorded its lowest average runtime at `--np = 7`