EUROCC - National Competence Centres in the framework of EuroHPC

EuroHPC-04-2019: HPC Competence Centres

Emerging Technologies: Rust in HPC

Laura Moran and J. Mark Bull
EPCC, University of Edinburgh

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 951732. The JU receives support from the European Union’s Horizon 2020 research and innovation programme and Germany, Bulgaria, Austria, Croatia, Cyprus, Czech Republic, Denmark, Estonia, Finland, Greece, Hungary, Ireland, Italy, Lithuania, Latvia, Poland, Portugal, Romania, Slovenia, Spain, Sweden, United Kingdom, France, Netherlands, Belgium, Luxembourg, Slovakia, Norway, Switzerland, Turkey, Republic of North Macedonia, Iceland, Montenegro

Table of Contents

1 Introduction

The work in this technical report has been performed to highlight how the programming language Rust performs in a HPC environment. A case study in computational fluid dynamics (CFD) was used to compare Rust to both C and Fortran. The CFD simulation was performed in serial and parallel (using OpenMP parallelisation) across a range of problem sizes. In section 2, Rust as a programming language is discussed: the motivation behind it, the advantages and disadvantages of the language, and previous work in benchmarking Rust for HPC. Section 3 outlines the specific CFD problem being solved in this study, and how the solution is found using Jacobian iterators. Section 0 shows the results of the serial and parallel work performed in the three languages and discusses these. Finally, Section 5 contains a brief summary of the work in this study.

2 Rust language

2.1 About the language

Rust had its first stable release in May 2015 [1]; it is a general-purpose programming language that focuses on concurrency, type safety and performance. A main feature of this is memory safety without a garbage collector. During compilation, a "borrow checker" tracks the lifetime of all references in a Rust program to ensure memory safety and prevent data races. Rust was designed to be "as efficient and portable as idiomatic C++, without sacrificing safety" [2]
The Stack Overflow Developer Survey [3] has named Rust the "most loved programming language" every year from 2016-2022, and in 2022 it shared the title of "most wanted technology" with Python. A number of big companies have incorporated Rust into their systems, including Facebook, Discord, Dropbox, Amazon and Google.
A large advantage of Rust for HPC is the avoidance of data races: a condition which happens when two (or more) threads access a shared variable without synchronisation and at least one of the threads writes to the variable. These bugs are difficult to replicate due to depending on specific circumstances: Rust ensures these circumstances never arise by enforcing strict ownership of variables.

2.2 Previous HPC work

- Performance vs Programming Effort between Rust and C on Multicore Architectures: Case Study in N-Body [4]: a study showing that Rust and C (after optimisations of both) perform similarly in double precision but C is better at optimising mathematical operations in single precision. Moreover, the authors argue that Rust produces code that is easy to maintain and more compact than similarly low-level languages so the programming effort is lower than that for C.
- Rust programming language in the high-performance computing environment [5]: compares multiple implementations of a finite difference stencil code and the performance between Rust, Fortran and C++. It shows that Rust can perform at the same level as the older languages.
- What can the programming language Rust do for astrophysics? [6]: a different N-body simulation study, specifically re-writing fundamental parts of Mercury-T, a Fortran code which simulates the dynamical and tidal evolution of multi-planet systems. This

paper shows that Rust can reach the same performance as the original Fortran implementation of the code, without the concern of data races.

This study is based on a simple computational fluid dynamics simulation, originally written for a Masters level course; the details of the simulation are in the following section.

3 Computational fluid dynamics

3.1 The problem

Computational fluid dynamics (CFD) problems are based on the mechanics of fluid flow, liquids and gases in motion. They are often solved on HPC systems: parallelising these computations allow larger and more complicated problems to be solved. For a computer to simulate these systems, the equations describing the motions must be discretised onto a grid. There are several discretisation methods, the one chosen here is the finite difference method. This solver states that the value at any point in the grid is a combination of the values at neighbouring points. The solver used is described in more detail in Section 3.2.

For this benchmarking exercise, a simple and scalable problem was chosen. This problem is based on work covered in the EPCC Masters course *Practical Introduction to HPC* and the Fortran and C codes are from this course. The problem pertains to a fluid in a box: a box is created with a fluid entering in one side and leaving via another, as seen in Figure 1. For simplicity, the assumption of zero viscosity is used meaning that the fluid cannot form whirlpools. The problem is considered solved when a steady state is found.

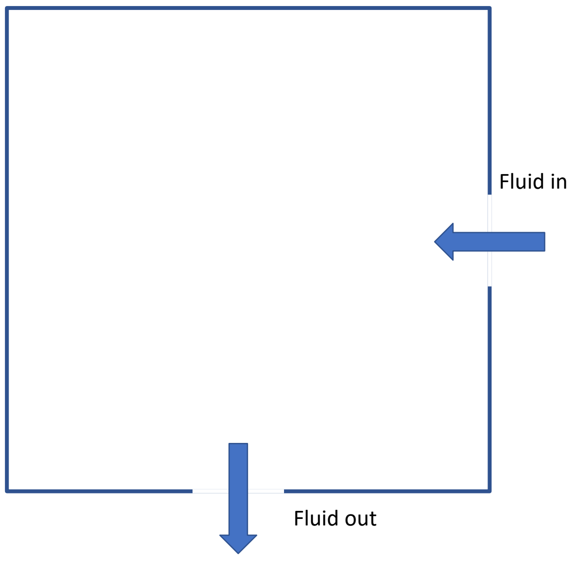

Figure 1. Diagram showing the CFD simulation set-up.

3.2 The solution

The problem described in Section 3.1 can be solved via the 2D stream function, ψ: the stream function is the volume flux through a curve defined by a start and an end point in an incompressible flow velocity field.

At zero viscosity, ψ satisfies:

$$\nabla^2\psi = \frac{\partial^2\psi}{\partial x^2} + \frac{\partial^2\psi}{\partial y^2} = 0. \tag{1}$$

As mentioned in Section 3.1, a continuous system described by partial differential equations (such as the stream function) can be discretised onto a grid for a computer to simulate correctly. The discretisation approach chosen here is the finite difference method. The differential equation can be expressed in the finite difference method as:

$$\psi_{i-1,j} + \psi_{i+1,j} + \psi_{i,j-1} + \psi_{i,j+1} - 4\psi_{i,j} = 0. \tag{2}$$

The Jacobi method can then be used to solve this finite difference equation by repeatedly averaging the value at each point in the grid with its four nearest neighbours. The process repeats until the solution is found, when the values at all points remain unchanged by the averaging process. Pseudocode to demonstrate how this looks in C can be seen in Figure 3.

The code involves a “base size” of the variables e.g., box size, widths of inlet and outlet. It is possible to scale the problem size via the command line when running the code, thus allowing a range of problem sizes to be run. The time taken to run these different problem sizes between the different languages can be compared. The output of the code is composed of velocity vectors that can be plotted on a 2D grid to show the steady-state solution at the end of the iterations. An example output is shown below in Figure 2.

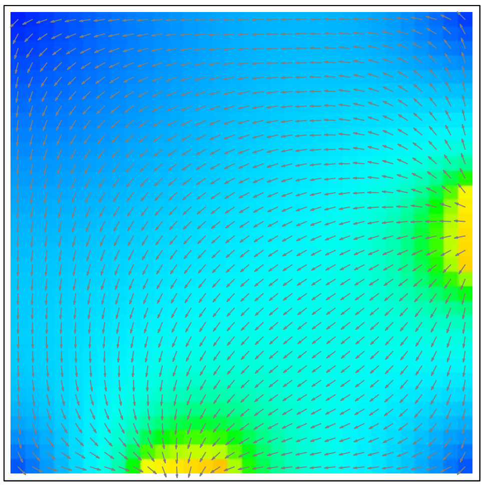

Figure 2. Example output for the fluid in a box CFD simulation. The inlet is on the right hand side and the outlet is on the bottom edge.

3.3 Experimental method

The work to benchmark the Rust implementation of the code outlined in Section 3.2 comprises of two main efforts: serial implementation and parallelised. The serial implementation was run for eight different problem sizes: 1, 2, 4, 8, 16, 32, 64 and 128 times the base set-up.

To parallelise the code, an OpenMP implementation was used. For Rust, the rayon crate was used. This crate guarantees the absence of data races by using mutable references. These references allow all threads to see the updated values. More details can be found in the documentation for this crate: https://docs.rs/rayon/latest/rayon/. Rayon works in a similar way to OpenMP so was deemed suitable for the comparison work in this study; the main difference is that instead of always executing in parallel, it dynamically chooses whether to do so based on available system resources at the time of execution.

The results of running these simulations in serial and parallel across different problem sizes and three languages are outlined in the following section. All of the simulations were run on a single node on Cirrus, a HPC system at EPCC. Each node of Cirrus has two 2.1 GHz, 18-core Intel Xeon E5-2695 (Broadwell) series processors. Each node has 256 GB of memory shared between the two processors, with one NUMA region for each processor. There are three levels of cache, configured as follows:

- L1 Cache 32 KiB Instr., 32 KiB Data (per core)
- L2 Cache 256 KiB (per core)
- L3 Cache 45 MiB (shared between 18 cores)

```
int main
{
  //main array
  double **psi;
  //temporary version of main array
  double **psitmp;

  //allocate array

  psi    = (double **) arraymalloc2d(m+2,n+2,sizeof(double));
  psitmp = (double **) arraymalloc2d(m+2,n+2,sizeof(double));

  //zero the psi array
  for (i=0;i<m+2;i++)
    {
      for(j=0;j<n+2;j++)
    {
      psi[i][j]=0.0;
    }
    }

  //set the psi boundary conditions
  boundarypsi(psi,m,n,b,h,w);

  //begin iterative Jacobi loop
  for(iter=1;iter<=numiter;iter++)
  {
      //calculate psi for next iteration
        for(i=1;i<=m;i++)
          {
             for(j=1;j<=n;j++)
           {
             psinew[i][j]=0.25*(psi[i-1][j]+psi[i+1][j]+psi[i][j-1]+psi[i][j+1]);
               }
            }

      //copy back
        for(i=1;i<=m;i++)
    {
      for(j=1;j<=n;j++)
        {
          psi[i][j]=psitmp[i][j];
        }
    }
}
```

Figure 3. Listing showing pseudocode for the finite difference method Jacobi iterator to solve the stream function.

4 Results

4.1 Serial results

The serial programs in C, Fortran and Rust were each run for eight problem sizes and the execution time for the number of iterations recorded. Each problem size was run with 5000 iterations, which was enough for even the largest problem size to converge on a solution. The resulting times are plotted below in Figure 4, using a log scale to allow the differences at the smaller problem sizes to be seen more clearly. Each problem size was run 10 times, the means with their standard deviations have been plotted. The error bars have been omitted as they were too small to see, demonstrating little difference between the runs which is encouraging to see. The three languages differ the most at the smaller problem sizes, with Fortran running a little faster than C, which in turn runs faster than Rust. However, the execution times of these small scale factors are so tiny that this is not a concerning result.

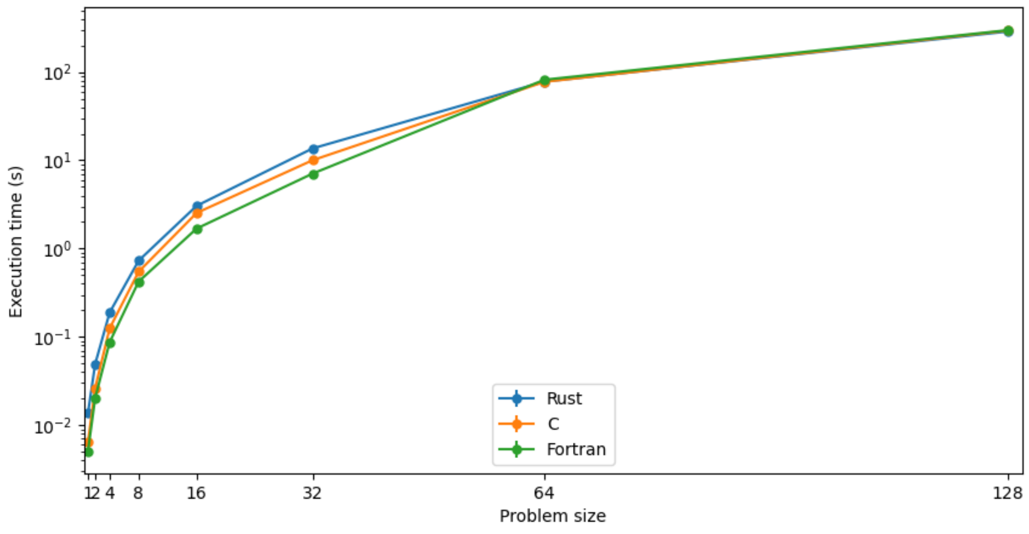

Figure 4. Plot displaying the execution time against the problem size for the serial code, for C, Fortran and Rust.

The execution times of the serial code relative to C are plotted in Figure 5, so the C results are the horizontal line at 1.0. It is clear from Figure 5 that at the smallest problem sizes Fortran and C are closer in execution time than Rust. Once the scale factor reaches 32, Fortran and Rust are approximately the same as C, with Fortran being slightly faster and Rust continuing to be a little slower. From problem size 64 onwards, the three languages take around the same time to execute. This is likely due to the code becoming memory bound and reaching the hardware limit of memory bandwidth.

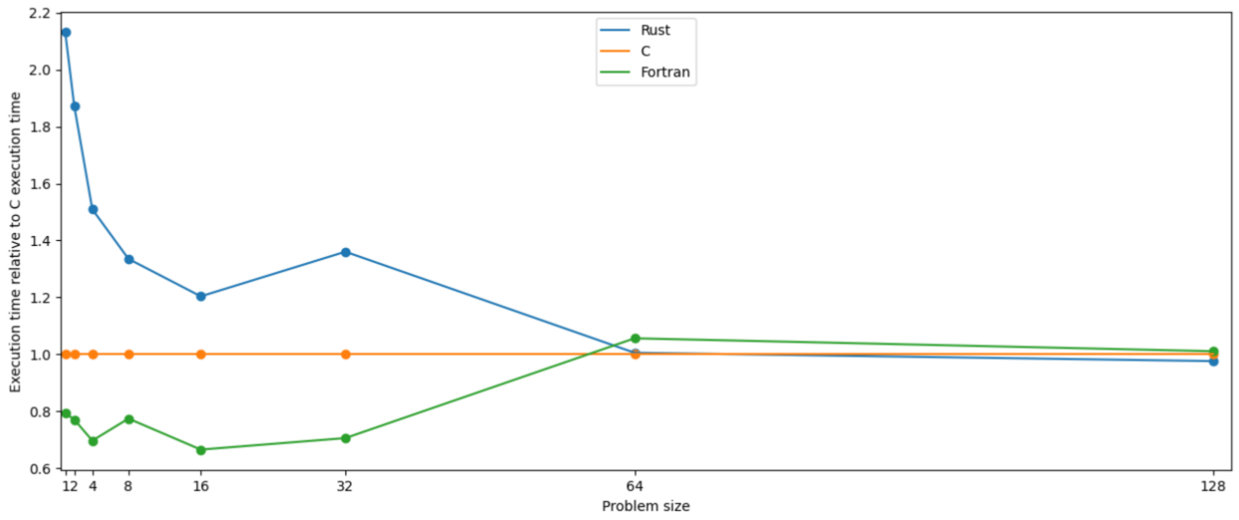

Figure 5. Plot showing the relative execution times of C, Rust and Fortran when compared to C for a range of problem sizes.

4.2 Parallel results

Satisfied that Rust runs as fast as C and Fortran in the serial implementation for problem sizes of sufficient size, the next step is to implement parallelisation. The source codes for C and Fortran contained OpenMP editions; a Rust code utilising the rayon crate described earlier was written based on the serial version. The parallel simulations were run for the same number of problem sizes as the serial, and for a range of thread counts. This resulted in too many plots to display so a suitable selection have been chosen for presentation and discussion here.

As in the serial runs, the number of iterations was set to be 5000, and ten runs were completed for each problem size and thread count in order to calculate a mean and a standard deviation. The variables chosen for graphical representation was number of iterations per second against thread count in order to see the speed-up gained by increasing the number of threads for a given scale factor. The smallest problem size was a scale factor of 1 (the same size as the base dimensions in the code) and the results can be seen in Figure 6.

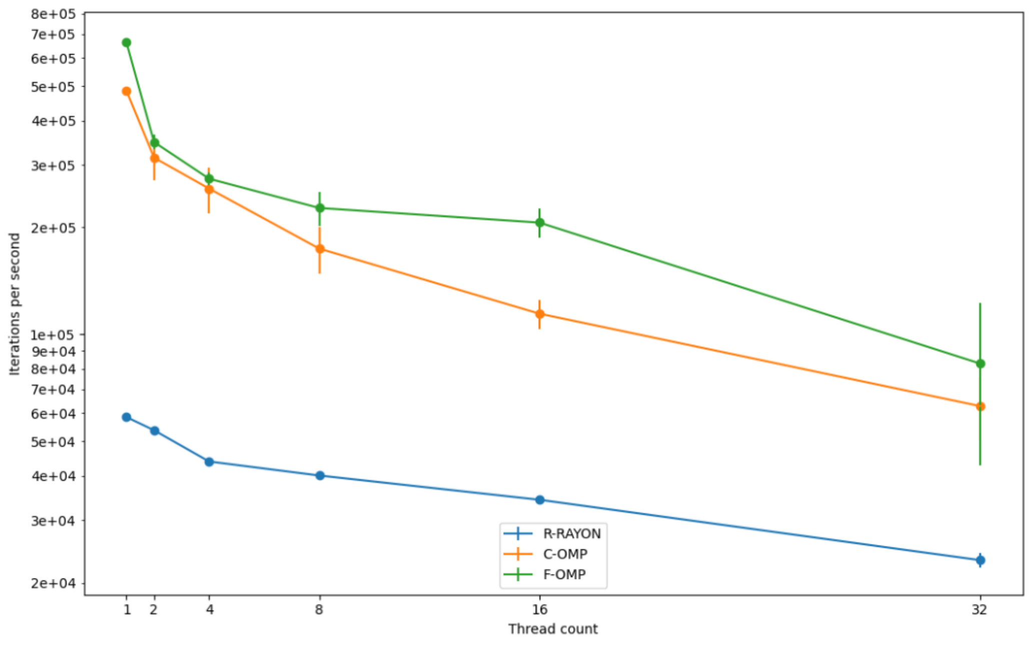

Figure 6. Plot showing the performance in iterations per second against number of threads for a scale factor of 1. Results are shown for Rust, C and Fortran.

Figure 6 shows that for the smallest problem size of 1, increasing the number of threads actually slows the simulation down. The overheads involved in splitting work in the parallel loops up and then synchronising at the end is greater than running all of the calculations on one thread. This trend is seen across all three languages, and it is also apparent that Rust is far slower than C and Fortran. This is due to the strict memory-safety enforced in Rust: altering the original array of values is forbidden and so a mutable reference must be made before any parallelisation can be performed. This disparity will be seen consistently across the next figures showing the larger problem sizes.

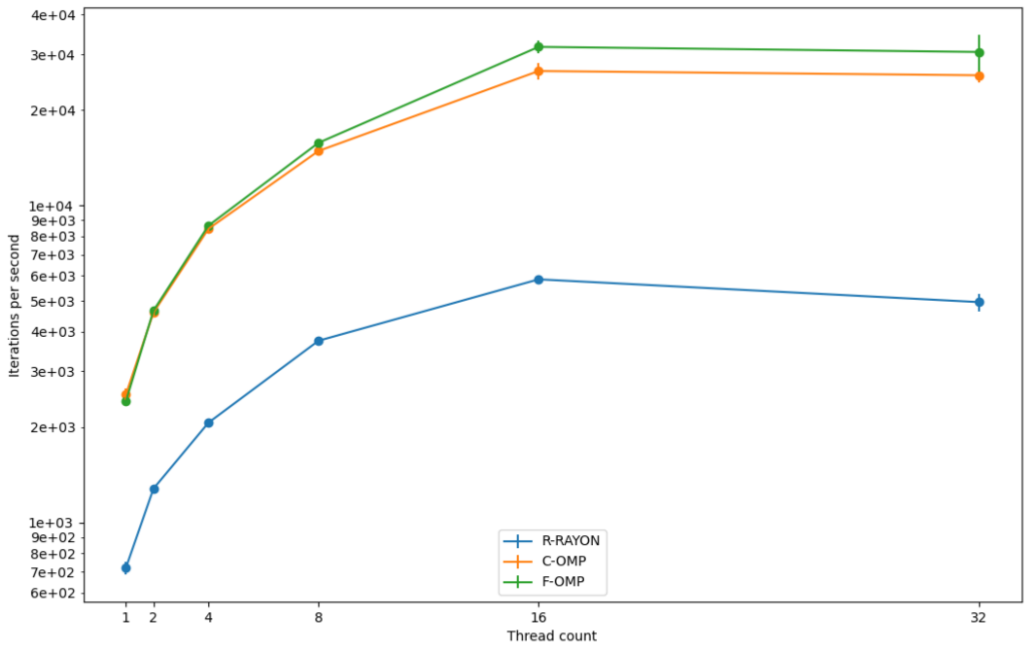

Figure 7. Plot showing the performance in iterations per second against number of threads for a scale factor of 16. Results are shown for Rust, C and Fortran.

Figure 7 shows the results of the same simulation for a problem size of 16. For this larger scale factor, increasing the number of threads working on the problem has increased the number of iterations performed per second, therefore reducing the runtime of the program. This effect is only seen until a thread count of 16, beyond which there is no improvement observed. Once reaching this threshold, the limiting factors are likely to be overheads of synchronisation, and the effects of running across two NUMA regions.

Finally, the largest problem size of 128 should also be examined, to get the full range as seen in the serial section of the study.

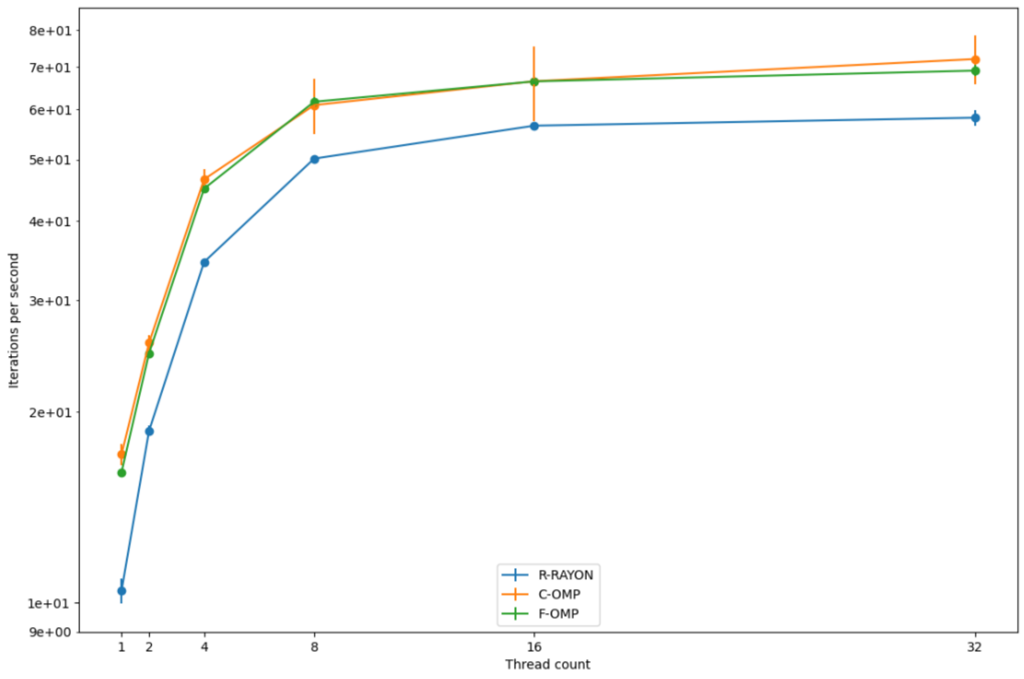

Figure 8. Plot showing the performance in iterations per second against number of threads for a scale factor of 128. Results are shown for Rust, C and Fortran.

Figure 8 shows the speed-up for the largest problem size run, a scale factor of 128. A similar trend to Figure 7 (scale factor 16) is seen, but the threshold for no further improvement is found quicker, by a thread count of 8. This is likely due to saturating the available memory bandwidth. Again, the disparity between Rust and the two older languages is apparent, however the scaling behaviour seen by increasing the thread count is similar across the three languages.

Another interesting feature across all three of the problem sizes displayed here (Figure 6, Figure 7 and Figure 8) is the difference in the error bars. All of the runs were submitted in batches and so the difference in the errors originates from the way the parallelisation in the languages work. The error bars for C and Fortran are far larger than that of Rust, or rather, Rust is more consistent in its execution times than the older languages. This is most likely due to the robustness of the memory safety in Rust and how the work-splitting is achieved. If there are threads waiting for others to complete before joining together this could make one run take longer than another for the same language. This could lead to the more consistent runtime observed between these simulations.

5 Conclusions

This study has shown how Rust performs compared to C and Fortran for a simple CFD exercise concerning the flow of fluid into and out of a square box in 2D. The flow is solved by discretisation of differential equations onto a grid by the finite difference method. In order to see how Rust scales on an HPC system, the code was run in both serial and parallel. The parallelisation was performed by OpenMP (for C and Fortran) and rayon (for Rust); the two are conceptually similar. The size of the box was altered over a large range to see how this affected the performance of the codes, and a range of thread counts were used for the parallel executions.

In serial, Rust has been proven to perform comparatively to C and Fortran, at sufficiently large problem sizes.

In parallel, the performance of Rust is slower than C and Fortran. This is determined to be due to how parallelisation is implemented. One of the main features of Rust is its attitude to memory safety and ensuring data races can never occur. This requires duplication of arrays in order to parallelise this problem, to create mutable references that can be altered by threads. This approach consistently adds to the execution time of the program. However, the effect of increasing threads is seen to be consistent across all of the languages, showing that Rust experiences the same improvements (or lack thereof!) as C and Fortran for these problems. It appears that the cost of extreme memory-safe practices is a slightly slower program, which is something to be considered when choosing which language is most appropriate for a given problem.

6 References

[1] R. documentation, May 2015. [Online]. Available: https://blog.rust-lang.org/2015/05/15/Rust-1.0.html.

[2] "pcwalton," [Online]. Available: http://pcwalton.blogspot.com/2010/12/c-design-goals-in-context-of-rust.html. [Accessed Dec 2022].

[3] StackOverflow, "Developer Survey Results 2022," 2022. [Online]. Available: https://survey.stackoverflow.co/2022/. [Accessed December 2022].

[4] M. Costanzo, E. Rucci, M. Naiouf & A. De Giusti, "Performance vs Programming Effort between Rust and C on Multicore Architectures: Case Study in N-Body," *2021 Latin American Computing Conference,* 2021.

[5] M. Sudwoj, "Rust programming language in the high-performance computing environment," *Bachelor Thesis,* 2020.

[6] S. Blanco-Cuaresma & E. Bolmont, "What can the programming language Rust do for astrophysics?," *Proceedings of the International Astronomical Union,* 2016.