**Solving the Advection Equation – Results**

**Introduction:**

This report provides an overview of HPC and its implementation in solving the Advection partial differential Equation, here Advection refers to the transport of a scalar field by the bulk motion of a fluid. Advection can be described by the continuity equation,

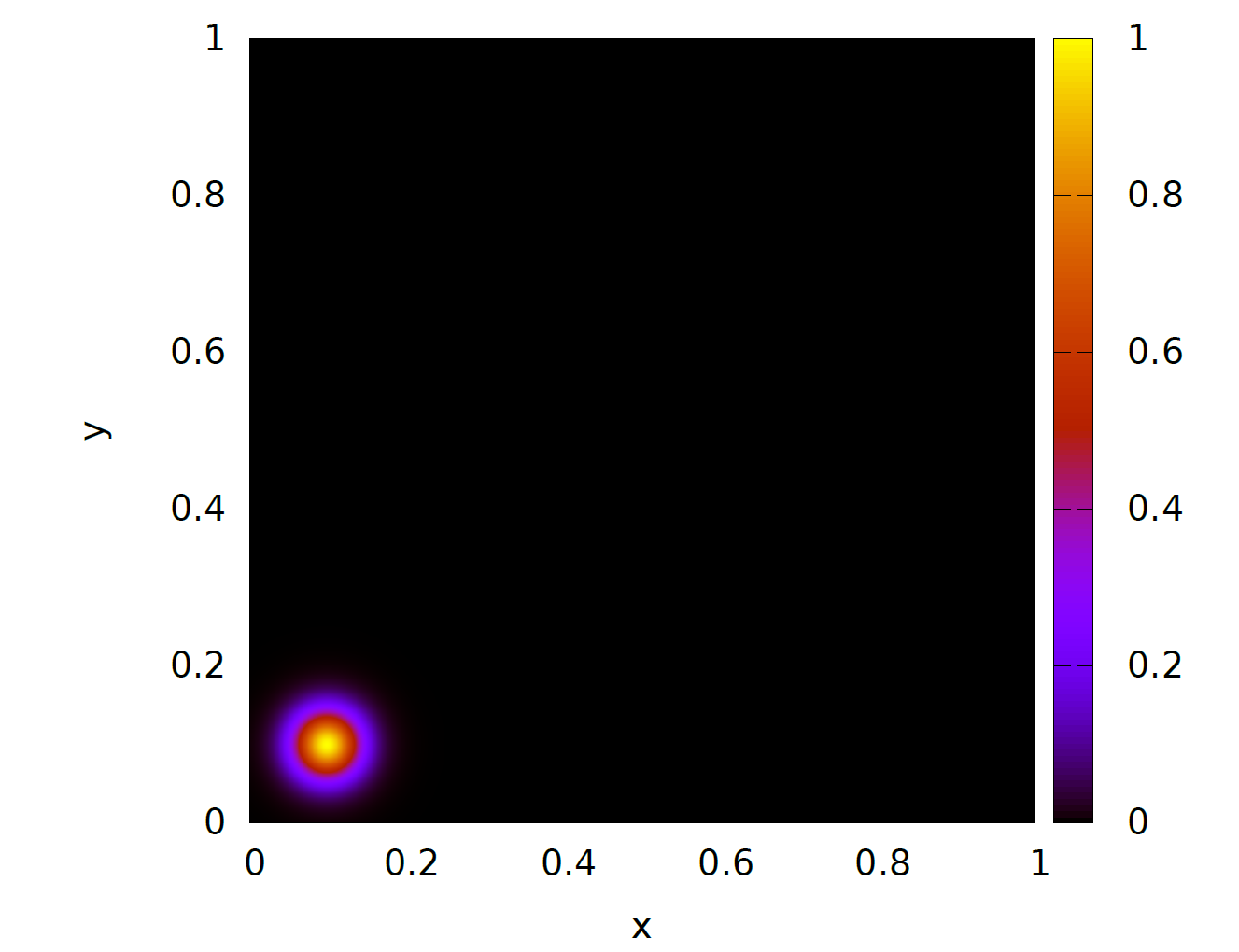
where u is the scalar field, is the velocity of the fluid and t is time. This program uses the method of finite difference to approximate solving this equation. The finite difference expression is given by

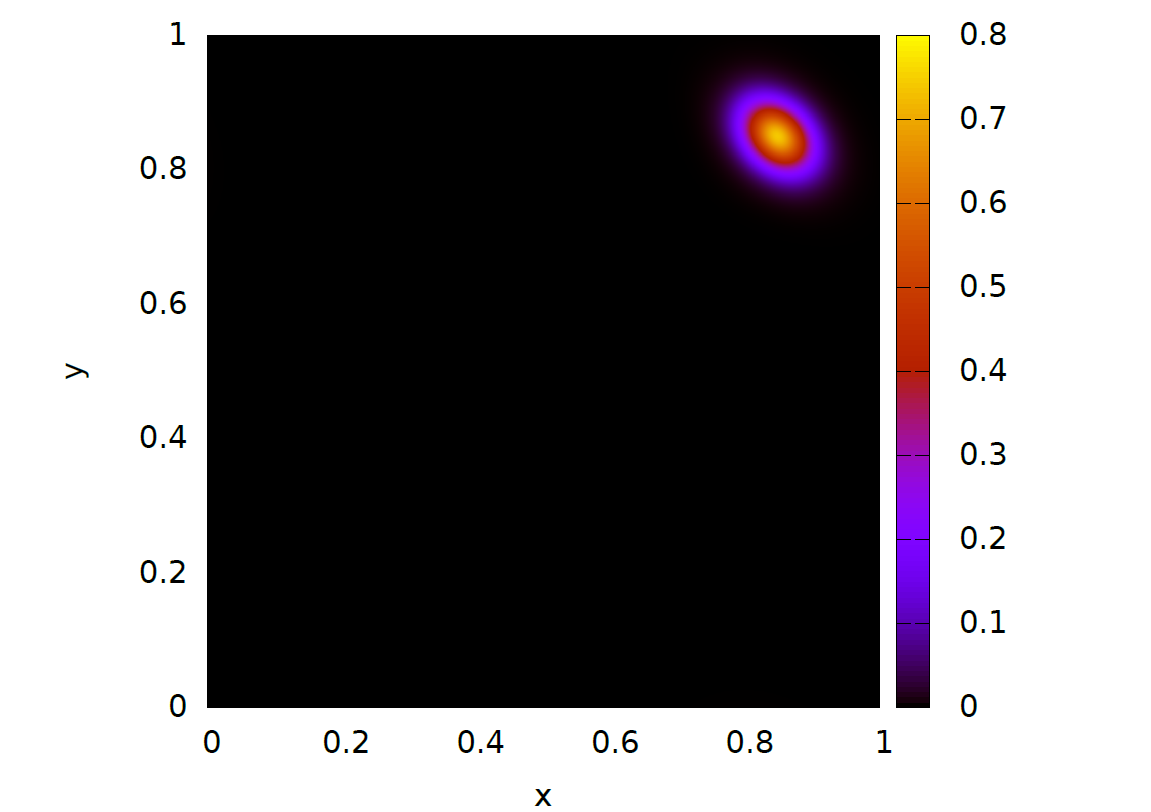
The parameters and initial conditions for this equation can be expressed mathematical as

The solution to this equation is implemented in a C program and uses OpenMP to allow parallelisation. The speed up and efficiency of the program when placed on ISCA (University of Exeter’s HPC system) is investigated and interpreted, drawing specific attention to parallel speed up and efficiency of solving this equation with 1500-time steps over 16 processors.

**What does this program do?**

This C program provides a solution to the Advection equation above using the method of finite difference. The solution is provided over a timestep of 1500 and a problem size of 1000x1000. The initial conditions as well as the boundary conditions are provided using the above formulas. The program is written to ensure parallelisation when placed on the University of Exeter HPC system, ISCA. The parallelisation of this program is provided by OpenMP, to ensure the best use of all the processors available.

**Plots of u(x,y) both with initial conditions and after 1500 time step.**

***Figure 1***: Initial conditions, showing u(x,y) at t=0.

***Figure 2***: u(x,y) after 1500 time steps with ∆t=0.05.

**How program was built and how tests were run (1/2 page)**

This program was built using the C programming language for a problem of size 1000x1000 points and compiled on ISCA compute node: 2x 8-core Intel Xeon E5-2640 v3 @ 2.60GHz, using the GCC 8.2.0 compiler. The results were averaged over three runs. The results omit the time taken to write to file.

The variability is low: submitting programs to the ISCA queue allows exclusive access to a compute node on ISCA so are not competing for resources.

The compiled ‘Add’ file was submitted to run on the ISCA HPC system by the running of a bash job script called ‘advection.pbs’ (see below), this file submitted the C program to be run on the ISCA system. After each run the line ‘Export OMP\_NUM\_THREADS=’ was changed to match the number of processors to be tested.

#!/bin/bash

# Number of nodes and processors per node requested:

#PBS -l nodes=1:ppn=16

# Time requested (in hh:mm:ss format):

#PBS -l walltime=00:05:00

# Name of stdout file:

#PBS -o Advection.out

# Name of stderr file:

#PBS -e Advection.err

# Queue name (do not change this):

#PBS -q training

# Account code (do not change this):

#PBS -A Research\_Project-IscaTraining

# Set the number of OpenMP threads to use

export OMP\_NUM\_THREADS=16

# Change into the directory the job was submitted from

cd ${PBS\_O\_WORKDIR}

# Load the most recent GCC module

module load GCC

# Run the "Advection" executable

/usr/bin/time ./Add

# End of file

**Table of results**

**Parallel Speed-up vs Efficiency:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| OpenMp? | #Threads | Walltime (s) | Speed-up | Efficiency |
| No  (Fastest Serial Run) | 1 | 76.0 | 1.00 | 1.00 |
| Yes | 2 | 38.7 | 1.96 | 0.98 |
| Yes | 4 | 19.7 | 3.85 | 0.96 |
| Yes | 8 | 10.8 | 7.03 | 0.88 |
| Yes | 16 | 6.06 | 12.5 | 0.78 |

ISCA compute node: 2x 8-core Intel Xeon E5-2640 v3 @ 2.60GHz

Compiler: GCC 8.2.0

Problem size: 1000x1000 points

Averaged over 3 runs, values to 3.S.F

Plots of parallel speed up and efficiency when placed on ISCA:

* Graphs are plotted using averages.

***Figure 3:*** Number of Processors vs Parallel Speed Up

***Figure 4:*** Number of Processors vs Parallel Efficiency

Interpretation of results:

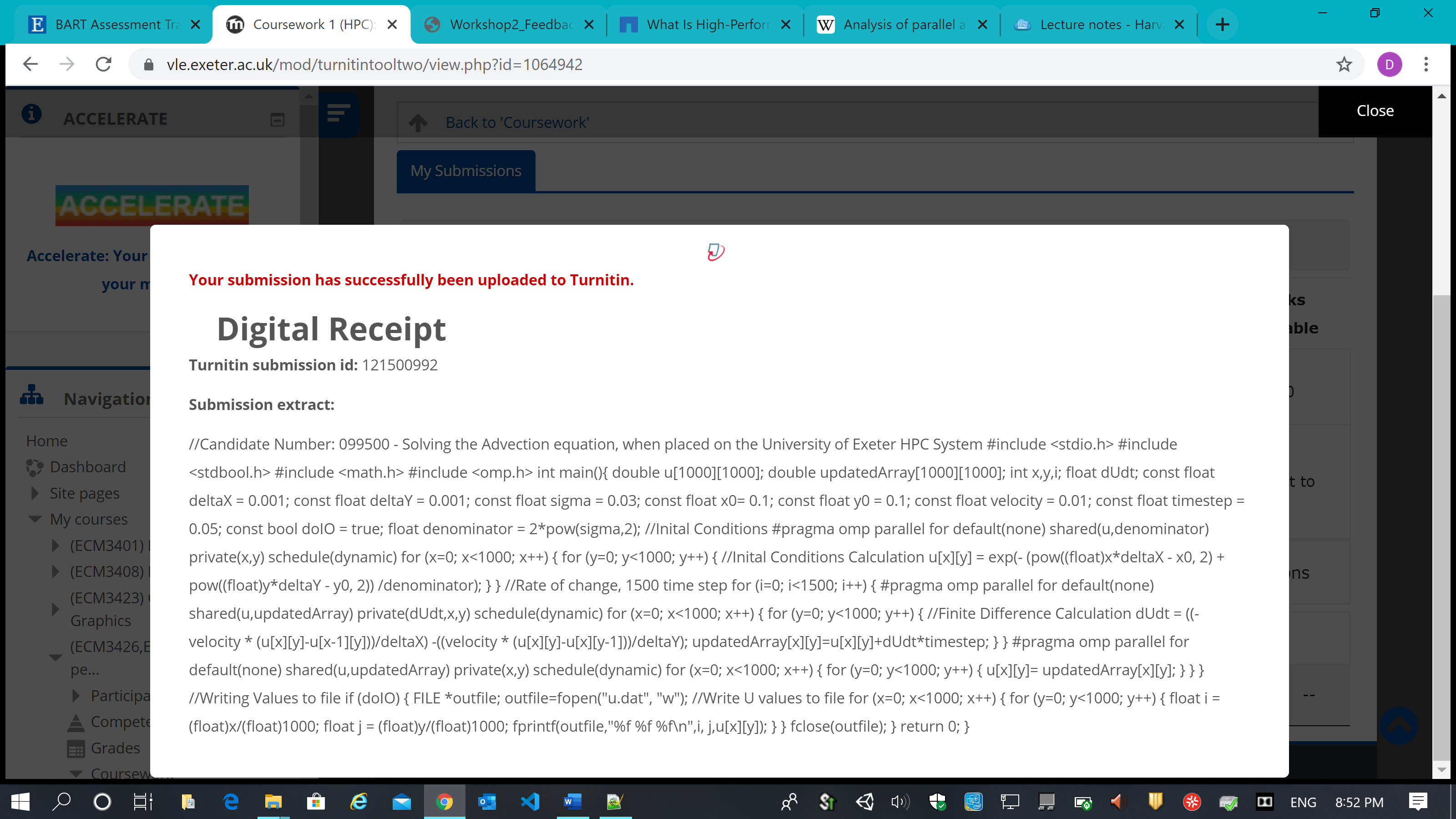
Analysis of parallel speed up and efficiency is interpreted under the assumption that a limitless number of processors is available. The results from the parallelisation of this program are summarised in the table and graphs above. Figures 3 shows the speed up results of running this program on 1 ISCA node varying the number of processors used. As the number of processors increases the time taken for the program to execute decreases. However, despite this increase in wall time, there is a smaller increase in the speed up. This difference is noticed since some parts of the program cannot be parallelised, and therefore will take a longer time to compute. The part of this program that takes the most time is the timestep iteration from 0 to 1500. As the timestep increases so does the computation time need to perform the calculation.

As the number of processors increases the efficiency of the program decreases, at first this decrease is subtle however after 8 processors this becomes more substantial. Introducing a dynamic loop schedule allowed the program to assign a thread to each iteration of the loop, when that thread finishes it is assigned the next iteration that hasn’t been executed yet. The loop over 1500 is the most computational expensive part of the program and therefore needs a dynamic loop schedule to ensure that iterations are executed efficiently and as fast as possible. Dynamic loop scheduling significantly improves the parallel scaling and ensures the program is well load balanced.

The program cannot be sped up due to the program having memory bound constraints. The program is limited by speed at which data can be accessed from main memory. Memory bound parts of the program cannot be sped up by using more cores in the same CPU. The program also exhibits compute bound parts which affect the rate at which processors can carry out arithmetic operations. This program consists of many floating-point operations therefore resulting in a longer computation time.

The program consists of many for loops which can be parallelised to ensure different iteration are execute by different threads when placed on an HPC system. Within the #pragma parallel for loops, the arrays are shared where constants and variables are private to ensure calculations remain consistent. These values need to remain private to ensure that different values for each loop iteration which are initialised within the parallel region. There is no data race within for loops to ensure that each iteration writes to a different array element. The for loop from 0 to 1500 for the timestep of this program cannot be parallelised since each iteration of the for loop depends on the previous one, which is illustrated by the finite different calculation.

To conclude, this program provides an effective solution to the Advection equation over a timestep of 1500. The program shows a increased speed up and efficiency when placed on the universities HPC system.



# References

*"What is high performance computing"*. (2020). Retrieved from Inside HPC: https://insidehpc.com/hpc-basic-training/what-is-hpc/

Acreman, D. D. (2020). *High Performance Computing and Distributed System Lectures.* University of Exeter .