Continuous Assessment (1)

086696

March 2020

1 Introduction

This report investigates how parallel programming can be applied to find numerical solutions to partial differential equations, in particular, the advection equation. Tests have been carried out on the ISCA supercomputer to demonstrate the parallel speedup and efficiency as a function of the number of processors when the program is run. The performance gains by parallelising the program are demonstrated and explored. The program was run on the University of Exeter's supercomputer known as ISCA. The 6,000 core system is compromised of a range of compute resources: the traditional cluster (128 GB nodes) is complimented by two large memory (3TB) nodes, Xeon Phi accelerator nodes and GPU (Tesla K80) compute nodes.

2 Program

The program under investigation finds numerical solutions to the advection equation in two dimensions. The advection equation describes the transport of a scalar field by the bulk motion of a fluid. The spatial derivatives are approximated using a finite difference expression.

$$\frac{\partial u}{\partial t} \approx -v_x \frac{u_{i,j} - u_{i-1,j}}{\Delta x} - v_y \frac{u_{i,j} - u_{i-1,j}}{\Delta y} \tag{1}$$

The change in x and the change in y are the spacing between grid points in the x and y directions, and i,j are the indices of the grid points in the x and y directions. This gives the rate of change of u which can then be used to time step the solution by continuously updating the value of u by adding the previous value to the calculated rate of change.

$$u_{new} = u_{current} + \frac{\partial u}{\partial t} \Delta t \tag{2}$$

The computational domain is a two dimensional grid consisting of 1000x1000 grid points. The initial conditions are a gaussian with width $\sigma = 0.03$ centered on $(x_0, y_0) = (0.1, 0.1)$. The initial conditions are expressed mathematically as follows.

$$u(x,y) = exp(-\frac{(x-x_0)^2 + (y-y_0)^2}{2\sigma^2}) \tag{3}$$

The program under test first initialises a 1000x1000 two dimensional array using the initial conditions given. It then proceeds to update these values by finding the rate of change of u at each timestep for a total of 1500 timesteps. At the beginning of each iteration an identical copy of the two dimensional array is made so that when the next loop which updates the values is parallelised, there is no chance of a data race occurring as loop iterations are now independent. After the timesteps have been completed, the final values of u for each coordinate are written to a dat file in order for gnuplot to produce a plot.

3 Plots of results

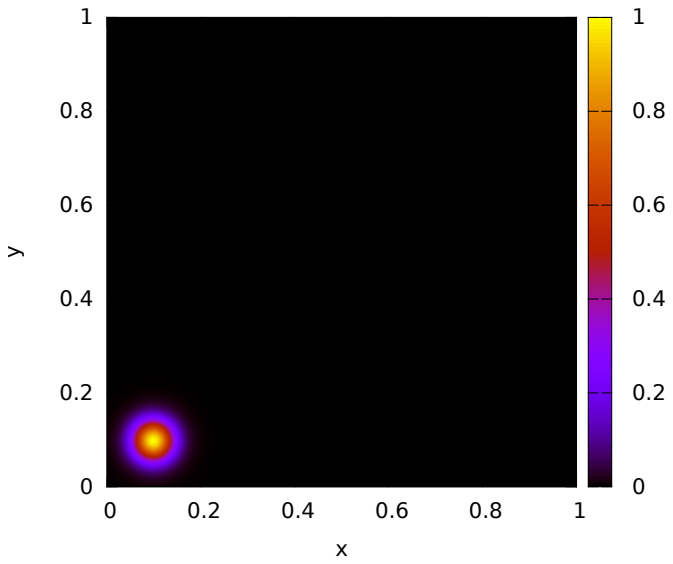

Figure 1: Initial Plot

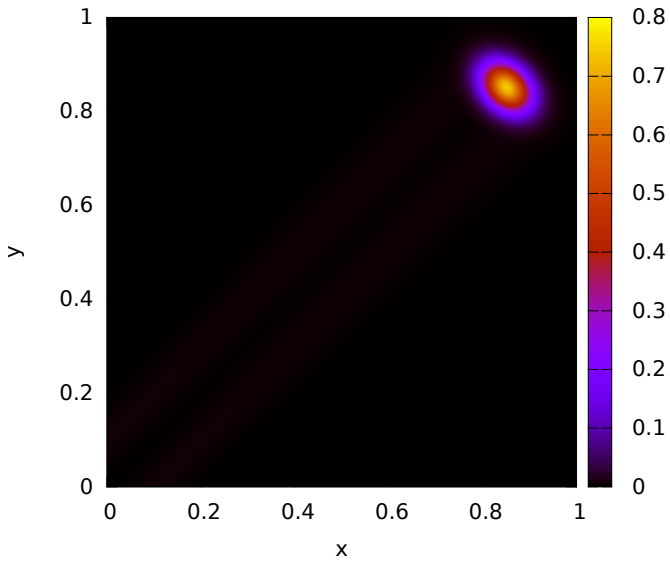

Figure 2: After 1500 timesteps

4 Building the program and running the tests

The serial version of the program was compiled using the command *gcc -o run ca.c -lm* This uses the gcc compiler version 8.1.0 and names the executable *run.exe* and links the math library to the program. The parallel version of the program was compiled using the command *gcc -o run -fopenmp ca.c -lm* which is the same as the serial compile command except it includes the flag *fopenmp* which links the OpenMP library which is used for the parallelisation.

To then submit this job to ISCAs queue a script file was used which requests the resources necessary to run the job. The file specifies the number of compute nodes for a specified length of time. The script also directs output to two files. *Ca.out* for stdout and *Ca.err* for stderr.

This script file is run with the command *msub* which submits the script file to the resource manager. This returns a job identifier which can then be used to check on the current status of the job. The program will produce a file *u.dat* in the directory the job was submitted from. This file can then be plotted using the command $gnuplot\ example_gnuplot_file$. This produces a pdf *u.pdf* containing the plotted grid.

The timing was measured differently for the serial and parallel versions. The parallel version was timed using the omp_get_wtime routine which returns the elapsed wall clock time in seconds. This is programmed as part of the *.c* file and is only available when compiled with OpenMP. The serial version was timed using the *time* command as part of the *pbs* script file. This gives the real-time, user CPU time, and system CPU time spent executing the program. The *real*

time is the time elapsed wall clock time taken by a command to get executed, while *user* and *sys* time are the number of CPU seconds that command uses in user and kernel mode respectively.

5 Results and Discussion

OpenMP	No. threads	Walltime (s)	Speed-up	Efficiency
No	1	19.3	1	1
Yes	1	19.3	1	1
Yes	2	10.1	1.91	0.96
Yes	3	7.0	2.76	0.92
Yes	4	5.5	3.5	0.88
Yes	5	4.8	4.02	0.80
Yes	6	4.1	4.71	0.79
Yes	7	3.7	5.22	0.75
Yes	8	3.4	5.68	0.71
Yes	9	3.1	6.23	0.69
Yes	10	2.9	6.66	0.67
Yes	11	2.8	6.89	0.63
Yes	12	2.6	7.42	0.62
Yes	13	2.5	7.72	0.59
Yes	14	2.4	8.04	0.57
Yes	15	2.3	8.39	0.56
Yes	16	2.2	8.77	0.55

Table 1: Table of results

6 Plots of parallel speed-up and efficiency

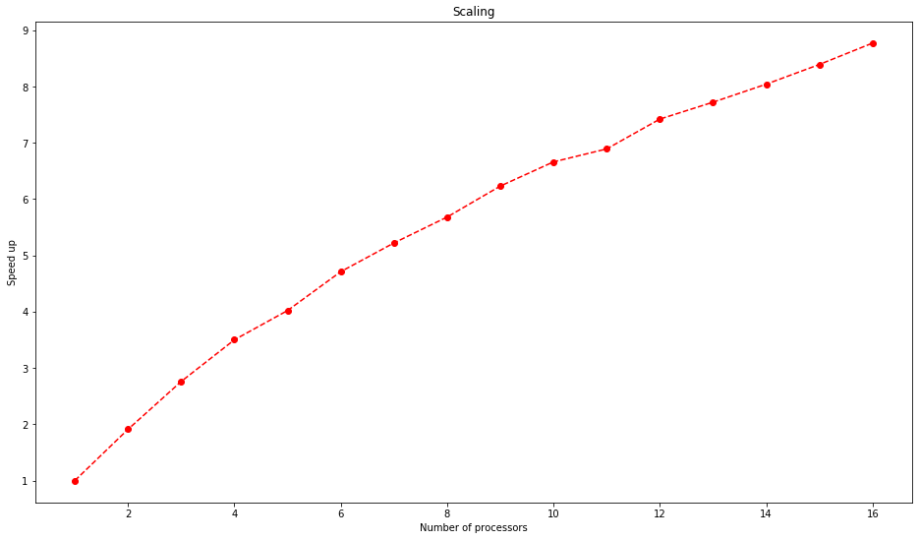

Figure 3: Speedup vs Number of Processors

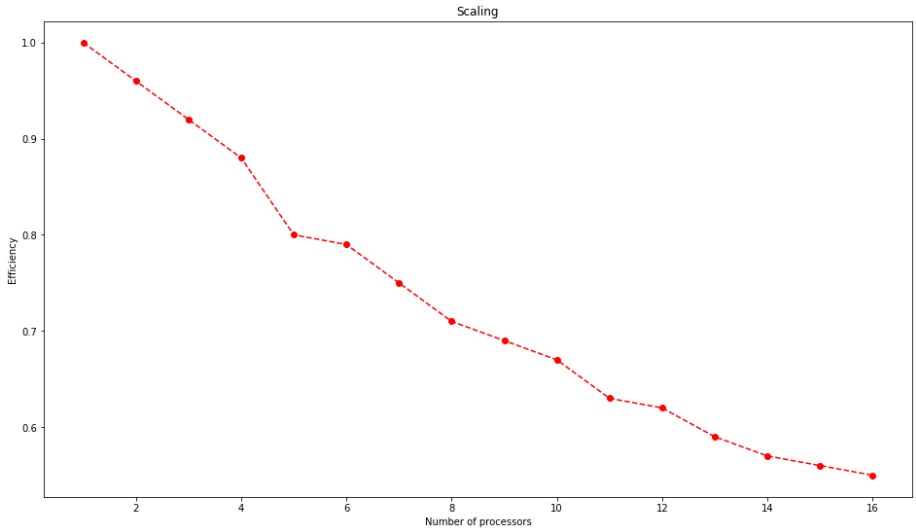

Figure 4: Efficiency vs Number of Processors

The results in table 1 were averaged over three runs of the program on a problem size of 1000 by 1000 points. The code was run on an ISCA compute node which

contains two, eight core Intel Xeon E5-2640 v3 @2.60Ghz processing units.

The results show us that parallelising the program has significantly reduced the time it takes for the program to run. The speed up is calculated as $S_N = T_1 \ / \ T_N$ where T_1 is the time for a serial run and T_N is the time to run on N processors. The efficiency is calculated as $E_N = S_N \ / \ N$ where S_N is the parallel speedup.

There is a very strong positive correlation between the speedup and the number of processors in use as shown in figure 3. The rate of change is consistent as the number of processors increases and does not show signs of dropping off. There is also a clear negative correlation between the efficiency and number of processors in use as shown in figure 4. The decrease in efficiency happens at a consistent rate, going from 0.96 with two threads down to 0.55 with sixteen threads.