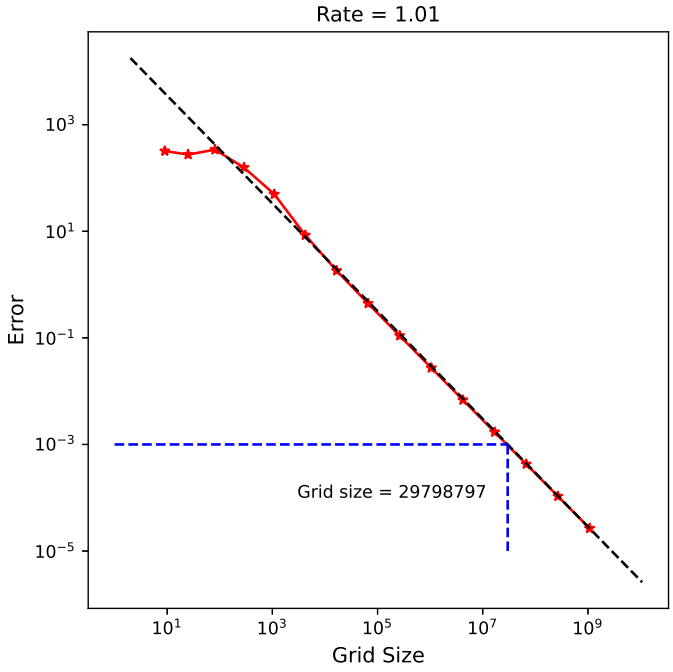

Figure 1: Approximation error vs grid size

Exercise 3 - Jonny Spence

Consider the problem of approximating, via the trapezoidal rule, the double integral

$$\int_0^b \int_0^a x\sin(x^2) + y\sin(y^2)\mathrm{d}x\mathrm{d}y = \frac{1}{2}\Big(- b\cos(a^2) - a\cos(b^2) + a + b\Big).$$

For a general function $f(x, y)$ the trapezoidal rule reads

$$\int_0^b \int_0^a f(x,y)\mathrm{d}x\mathrm{d}y \approx \Delta x\Delta y \sum_{i=1}^{N}\sum_{j=1}^{N} \frac{f(x_{i-1}, y_{j-1}) + f(x_{i-1}, y_j) + f(x_i, y_{j-1}) + f(x_i, y_j)}{4},$$

where we assume the points x_i, y_j for $0 \leq i, j \leq N$ form an $(N+1) \times (N+1)$ uniform mesh on $(0, a) \times (0, b)$.

To test the convergence of this numerical method, we first run the trapezoidal rule for several values of N and plot the resulting approximation error against the number of grid points $(N+1)^2$. For the experiments, I assume $a = b = 10$ (my code converged too slow for $a = b = 100$). Results are plotted in Figure 1 as the red line. The dashed black line has gradient $-$Rate, for Rate ≈ 1 calculated by linear regression through the data, indicating the method satisfies

$$\text{Error} \propto \text{Grid Size}^{-1}.$$

In particular, the blue dashed line shows that we obtain error 10^{-3} when using $\approx 2.98 \times 10^7$ grid points. If we continue the dashed line, we find that the method reaches machine round off error $\approx 10^{-16}$ using around 2×10^{20} grid points.

We consider the use of MPI parallelism to improve the methods performance. To split the calculation among several processors, consider splitting the region $[0, a] \times [0, b]$ in two ways: Into rectangles e.g. $[x_0, x_1] \times [0, b]$ and into squares e.g. $[x_0, x_1] \times [y_0, y_1]$ where $x_1 - x_0 = y_1 - y_0$. If we use np processors,

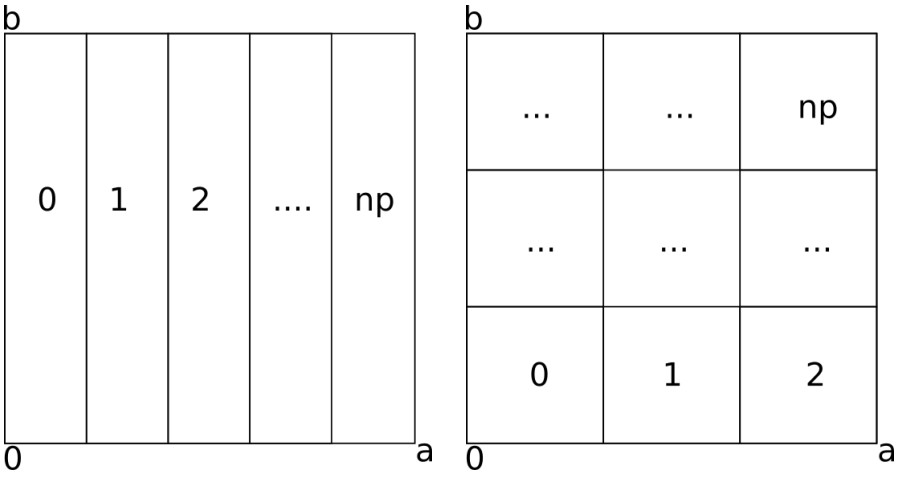

Figure 2: Division of grid into several components.

both methods are illustrated in Figure 2. We run the trapezoidal rule using both approaches for MPI, with several values $1 \leq \text{np} \leq 32$. When dividing the grid into squares, we impose that np is a perfect square, to avoid overlap between the regions. Since round-off error is achieved only for around 10^{20} grid points which is computationally demanding, consider instead a resolution $N_{\max} = (2^{15}+1)^2 \approx 10^9$, giving an error around 3×10^{-5} as seen in Figure 1. Validity of the methods is checked by observing both MPI implementations have the same error at this resolution. The run-time of each method against the number of processors is plotted in Figure 3. Note that both methods see a roughly linear decrease in computation time with increasing number of processors. This is to be expected since the size of the problem to be solved by each processor decreases linearly with the number of processors. Also, note that both methods of splitting up the grid have broadly comparable run times. This is also expected since when using the same number of processors, both methods supply the same number of grid points to each processor - the shape of the supplied region is of little relevance. However, when using square division of regions we impose that the number of processors is a perfect square, giving us much less control on the number of processors used. I would therefore suggest it is simpler to implement the rectangular sub-division. Finally, note that the run-time decreases slower when using more than 16 processors. This could potentially be due to the architecture of the 32 processors on a single node on cirrus.

The accompanying code is organized as follows:

- trapezoidal.cpp contains a serial implementation of the problem and produces the output of Figure 1.
- trapezoidal-par-nmax.cpp and trapezoidal-par-nmax-square.cpp contain an MPI implementation of the problem using each method of sub-division, at resolution $N_{\max}$.
- plots.py is a python implementation which uses the output of trapezoidal.cpp and several runs of the MPI functions to produce Figure 1 and 3.

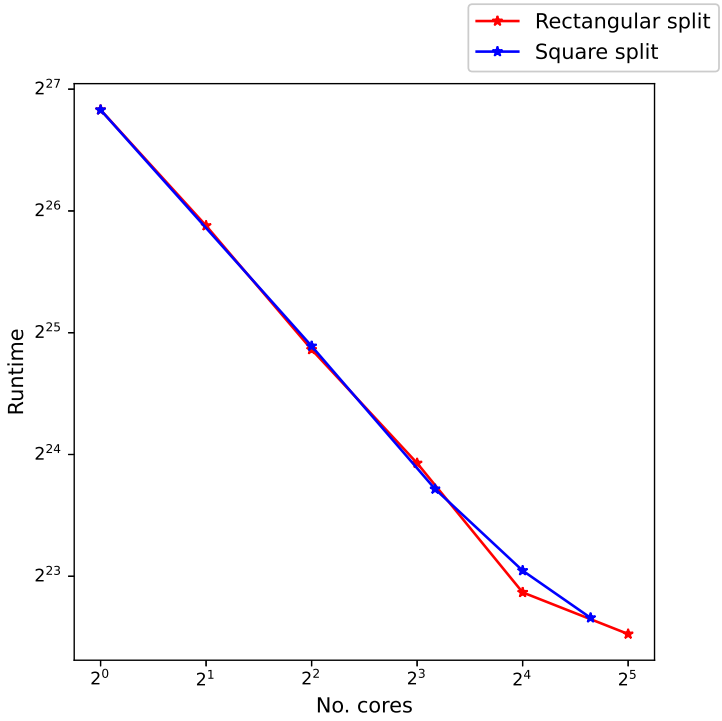

Figure 3: Run-time vs number of processors.