MA40177: Scientific Computing
Assignment 2

Handed out: Friday, April 8th 2022
Due in: Friday, May 6th 2022 (*by 17.00 at Assignment 2 - Submission Point on Moodle*)

– –

This assignment is worth 50% of the total assessment for MA40177 and should take an average student about 20 hours to complete (provided they have done all the problem sheet questions and tutorial exercises). Marks given in square brackets below indicate the marks available for each part. Please provide answers in the spaces below. You may word process your work if you wish, but please still insert it into the marked spaces. No marks will be lost if it is not word processed, provided it is legible. All programs should be written in `FORTRAN95`. All real arithmetic should be done in double precision (`kind = 8`). When writing/modifying subroutines, use the exact order of parameters specified in the questions and use the given filenames.

You should not discuss the details of your work with anyone else. The work which you hand in must be your own. You should be prepared to explain anything which you write to an examiner if asked to do so. In particular, if it is discovered that all or part of your code has been copied, both parties involved risk a severe penalty and might lose all their marks on the assignment.

IMPORTANT: When you have finished the assignment, please put all the relevant files (i.e. those you copied over and those you wrote) into a directory with a distinct name that identifies you as the author, e.g. `assignment2_skl27`. Zip up the directory into a single file using the command

```
tar czvf assignment2_skl27.tgz assignment2_skl27
```

and then upload the tgz-file that you obtain **and** a PDF file of your written work on the course Moodle page at Assignment 2 - Submission Point.

1 Two-dimensional Gray-Scott equations

In the second assignment we consider a two-dimensional model of the nonlinear reaction-diffusion, focusing on aspects of parallel computations. As previously, we are looking for concentrations u, v of two substances U, V, but now they depend on two variables $(x, y) \in [0, 1]^2$ and satisfy the two-dimensional partial differential equations

$$\frac{\partial u(x,y,t)}{\partial t} = -uv^2 + F(1-u) + D_u \Delta u \qquad \text{on } x, y \in [0,1], \quad t \in [0, T] \tag{1}$$

$$\frac{\partial v(x,y,t)}{\partial t} = uv^2 - (F+k)v + D_v \Delta v \tag{2}$$

$$u(1,y,t) = u(0,y,t), \quad v(1,y,t) = v(0,y,t),$$
$$u(x,1,t) = u(x,0,t), \quad v(x,1,t) = v(x,0,t) \qquad \text{(Boundary conditions)} \tag{3}$$

$$u(x,y,0) = u_0(x,y), \quad v(x,y,0) = v_0(x,y). \qquad \text{(Initial conditions)} \tag{4}$$

Here, Δ is the Laplace operator acting on bivariate functions as $\Delta u = \partial^2 u/\partial x^2 + \partial^2 u/\partial y^2$. As previously, we fix the diffusion coefficients to $D_u = 2 \cdot 10^{-5}$ and $D_v = 10^{-5}$, but we will vary the *feed rate* of the first substance $F > 0$, and the *kill rate* of the second substance $k > 0$.

1.1 Discretisation in space

We use again the finite difference method, extended to two variables. We divide the domain $[0,1]^2$ into $n = m^2$ cells of size $h \times h$ each, where $h = 1/m$. Then we define the discrete fields

$$u_{i,j} = u(ih,\ jh,\ t), \quad v_{i,j} = v(ih,\ jh,\ t), \qquad i,j = 1,\ldots,m. \tag{5}$$

Since the Laplace operator is a sum of univariate derivatives, we can use the one-dimensional finite difference scheme, and approximate

$$\frac{\partial u^2}{\partial x^2} \to \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h^2}, \qquad \frac{\partial u^2}{\partial y^2} \to \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h^2},$$

and similarly for Δv (recall the **Tutorial on Poisson's** equation). The periodic boundary conditions (3) imply that

$$u_{m+1,j} = u_{1,j}, \quad u_{0,j} = u_{m,j}, \qquad u_{i,m+1} = u_{i,1}, \quad u_{i,0} = u_{i,m}, \tag{6}$$

$$v_{m+1,j} = v_{1,j}, \quad v_{0,j} = v_{m,j}, \qquad v_{i,m+1} = v_{i,1}, \quad v_{i,0} = v_{i,m}. \tag{7}$$

Collecting discrete values $u_{i,j}, v_{i,j}$ into vectors

$$\mathbf{u}(t) = \begin{bmatrix} u_{1,1} & u_{2,1} & \ldots & u_{m,1} & u_{1,2} & \ldots & u_{m,2} & \ldots & u_{1,m} & \ldots & u_{m,m} \end{bmatrix}^\top \tag{8}$$

$$\mathbf{v}(t) = \begin{bmatrix} v_{1,1} & v_{2,1} & \ldots & v_{m,1} & v_{1,2} & \ldots & v_{m,2} & \ldots & v_{1,m} & \ldots & v_{m,m} \end{bmatrix}^\top \tag{9}$$

we can write the central differences as the matrix-vector products $\boldsymbol{\Delta}\mathbf{u}, \boldsymbol{\Delta}\mathbf{v}$ with the matrix

$$\boldsymbol{\Delta} = \frac{1}{h^2} \begin{bmatrix} \mathbf{D} & \mathbf{I} & \mathbf{0} & \ldots & \mathbf{I} \\ \mathbf{I} & \mathbf{D} & \mathbf{I} & & \\ & \ddots & \ddots & \ddots & \\ & & \mathbf{I} & \mathbf{D} & \mathbf{I} \\ \mathbf{I} & \ldots & \mathbf{0} & \mathbf{I} & \mathbf{D} \end{bmatrix} \in \mathbb{R}^{m^2 \times m^2}, \tag{10}$$

where $\mathbf{I} \in \mathbb{R}^{m \times m}$ is the identity matrix, and

$$\mathbf{D} = \begin{bmatrix} -4 & 1 & 0 & \ldots & 1 \\ 1 & -4 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -4 & 1 \\ 1 & \ldots & 0 & 1 & -4 \end{bmatrix} \in \mathbb{R}^{m \times m}. \tag{11}$$

The nonlinear reaction terms in (1),(2) can be written similarly to the first assignment

$$\mathbf{N}_u(\mathbf{u},\mathbf{v}) = -\mathbf{u} \odot \mathbf{v} \odot \mathbf{v} + \mathbf{f} - F\mathbf{u}, \qquad \mathbf{N}_v(\mathbf{u},\mathbf{v}) = \mathbf{u} \odot \mathbf{v} \odot \mathbf{v} - (F+k)\mathbf{v},$$

where $\mathbf{f} = \begin{bmatrix} F & \cdots & F \end{bmatrix}^\top \in \mathbb{R}^{m^2}$, and "$\odot$" is the elementwise product, but now over the larger vectors (8), (9).

1.2 Time stepping

In the code of this assignment, we will use the Compressed Row Storage (CRS) of matrices, and hence we have to sacrifice the possibility of using direct LAPACK solvers in implicit time schemes. However, the acceleration offered by parallel computing allows us to take smaller

Algorithm 1 Heun method for reaction-diffusion equations

Input: initial vectors $\mathbf{u}_0 = \mathbf{u}(0), \mathbf{v}_0 = \mathbf{v}(0)$, time step size $\tau > 0$, final time $T > 0$.
for $\ell = 0, \ldots, \lceil T/\tau \rceil - 1$ **do**
  Compute $\begin{aligned} \mathbf{r}_u &= -\mathbf{u}_\ell \odot \mathbf{v}_\ell \odot \mathbf{v}_\ell + \mathbf{f} - F\mathbf{u}_\ell + D_u \boldsymbol{\Delta} \mathbf{u}_\ell, \\ \mathbf{r}_v &= \mathbf{u}_\ell \odot \mathbf{v}_\ell \odot \mathbf{v}_\ell - (F+k)\mathbf{v}_\ell + D_v \boldsymbol{\Delta} \mathbf{v}_\ell. \end{aligned}$
  Compute $\begin{aligned} \hat{\mathbf{u}} &= \mathbf{u}_\ell + \tau \mathbf{r}_u, \\ \hat{\mathbf{v}} &= \mathbf{v}_\ell + \tau \mathbf{r}_v. \end{aligned}$ ▷ Predictor
  Compute $\begin{aligned} \hat{\mathbf{r}}_u &= -\hat{\mathbf{u}} \odot \hat{\mathbf{v}} \odot \hat{\mathbf{v}} + \mathbf{f} - F\hat{\mathbf{u}} + D_u \boldsymbol{\Delta} \hat{\mathbf{u}}, \\ \hat{\mathbf{r}}_v &= \hat{\mathbf{u}} \odot \hat{\mathbf{v}} \odot \hat{\mathbf{v}} - (F+k)\hat{\mathbf{v}} + D_v \boldsymbol{\Delta} \hat{\mathbf{v}}. \end{aligned}$
  Compute $\begin{aligned} \mathbf{u}_{\ell+1} &= \hat{\mathbf{u}} + \tau(\hat{\mathbf{r}}_u - \mathbf{r}_u)/2, \\ \mathbf{v}_{\ell+1} &= \hat{\mathbf{v}} + \tau(\hat{\mathbf{r}}_v - \mathbf{r}_v)/2. \end{aligned}$ ▷ Corrector
end for
Return $\mathbf{u}(T) \approx \mathbf{u}_\ell$, $\mathbf{v}(T) \approx \mathbf{v}_\ell$.

time steps, which render explicit time schemes stable. In particular, we are still employing the Runge-Kutta order-2 Heun method, but now apply it to the entire right hand side of

$$\frac{d\mathbf{u}(t)}{dt} = \boldsymbol{\Delta}\mathbf{u} + \mathbf{N}_u(\mathbf{u}, \mathbf{v}), \tag{12}$$

$$\frac{d\mathbf{v}(t)}{dt} = \boldsymbol{\Delta}\mathbf{v} + \mathbf{N}_v(\mathbf{u}, \mathbf{v}). \tag{13}$$

The pseudocode can be summarised as shown in Algorithm 1. The Heun method is stable as long as $|\lambda|\tau < 2$ for all eigenvalues λ of the Jacobian of (12), (13). In our case, the largest eigenvalue is dominated by that of the Laplace matrix (10). A slightly underestimated (ignoring $\mathbf{N}_u$) stability criterion is thus $\tau_u < 1/4$, where $\tau_u = D_u\tau/h^2$, the Courant number.

2 The Assignment

Copy over the files from the directory `$MA40177_DIR/assignment2`. They are designed to solve numerically the model problem defined above. However, the subroutines in `timestepping.f90` which implements Algorithm 1, and in `rhs.f90` which computes the right hand side of (12), (13), are empty. It will be your task to implement them. You will also need to parallelise sequential parts of the code in `create_matrix.f90`, `initial.f90`, `matmult.f90`, and in the main program `grayscott.f90`.

Part I: Implementation

Q1: Parallelisation of Laplace matrix and initial state

Study the subroutine `create_matrix` in the file `create_matrix.f90`, and the subroutine `initial` in the file `initial.f90`. Identify what lines are specific for the sequential code, and what parts would remain unchanged in a parallel implementation. **Modify both** files such that the matrix and initial state are constructed efficiently in parallel. *Hint: these tasks are easier than it looks like, i.e. only a few lines actually need changing.*

[4 points]

Q2: Parallelisation of the matrix-vector product

The subroutine `Mat_Mult` in the file `matmult.f90` contains the sequential code for multiplication of a CRS matrix by a vector. **Modify** this code to make it parallel and efficient. *Hint:* you can build upon the `sparsegather` routine from Problem Sheet 6, but you need to adapt it to the Gray-Scott model (1)–(4).

[4 points]

Q3: Computation of nonlinear terms and Right-Hand Side of (12), (13)

Implement a subroutine `RHS(u,v,F,k,Du,Dv,Delta,ru,rv)` in the file `rhs.f90` which takes

- `u,v`: derived type storage of parallelised real solution vectors $\mathbf{u}, \mathbf{v}$,
- `F`: real feed rate $F > 0$,
- `k`: real kill rate $k > 0$,
- `Du,Dv`: real diffusion coefficients $D_u > 0$ and $D_v > 0$,
- `Delta`: derived type CRS storage of parallelised real matrix $\mathbf{\Delta}$.

For the given $\mathbf{u}$ and $\mathbf{v}$ in an appropriately parallelised form, the subroutine should compute and return the right-hand sides

$$\mathbf{r}_u = -\mathbf{u} \odot \mathbf{v} \odot \mathbf{v} + \mathbf{f} - F\mathbf{u} + D_u \mathbf{\Delta} \mathbf{u}, \qquad \mathbf{r}_v = \mathbf{u} \odot \mathbf{v} \odot \mathbf{v} - (F + k)\mathbf{v} + D_v \mathbf{\Delta} \mathbf{v}.$$

in the same parallelised form in the derived-type variables `ru` and `rv`, respectively. Use `Mat_Mult` subroutine developed in the previous question.

[6 points]

Q4: Parallel time stepping algorithm

Implement the subroutine `timestepping` in the file `timestepping.f90` which implements the Heun method from Algorithm 1. The subroutine should take the following arguments:

- `u,v`: derived type storage of parallelised real solution vectors $\mathbf{u}, \mathbf{v}$,
- `tau`: real time step $\tau > 0$,
- `T`: real final time $T > 0$,
- `F`: real feed rate $F > 0$,
- `k`: real kill rate $k > 0$,
- `Du,Dv`: real diffusion coefficients $D_u > 0$ and $D_v > 0$,
- `Delta`: derived type CRS storage of parallelised real matrix $\mathbf{\Delta}$.

Hint: use `RHS` routine developed in the previous question. For simplicity, you can assume that T is a multiple of the time step ($T = \ell\tau$ for some $\ell \in \mathbb{N}$), and that the number of grid cells m in x and y is a multiple of the number of processes `nproc`. Make sure first that your code works on one process (partial credit will be given for a working sequential code.) Make your code efficient and use as few vectors as possible.

[6 points]

Q5: Main program and Makefile

Modify the main program in `grayscott.f90` such that the code can run in parallel. Make it more user–friendly by allowing the user to specify $m, \tau, T, F, k, D_u, D_v$ (in this order) in a file `input.dat`. **Modify the Makefile** such that it compiles parallel code correctly. Verify that your program works first on one processor and returns positive concentrations. You can use the Fortran subroutine in `save_fields.f90` and the Python script `visualise.py` to plot the solution to a PDF file. Bear in mind that `save_fields.f90` is a sequential code.

In addition, **write** a 0.5-1 page **report** describing how you implemented, modified and *tested* the codes, *what* and *why* you did to make the code efficient and correct.

[6 points]

Insert your report on implementation here

My plan for this part of the assignment was to get the code working and then later optimise. I have outlined how I optimised and made the code more efficient in each section as well as written how it was all tested.

I started with Q1. Each processor only needed to be able to have $\frac{m^2}{nprocs}$ rows of the matrix and each of the vectors of u and v. Given the sequential version of `create_matrix`, only one line needed to be edited in the end. This was the index for the do loop. It needed to go from the start to the end of the locally stored part of Delta. This was the only edit to the code that I actually made (apart from the additional commenting). Moving onto the changing of `initial.f90`. This was a very similar process and the only part of the code that needed editing was again the indexing over the do loop. This was so only the locally stored part of u and v were populated. In addition to this change, I added a v_{inv} variable to be able to calculate the inverse of v2 before the do loop and just use this throughout. This was in the hope of saving time on divisions later especially when m increases to really large values. This was at the cost of the storage of one double but given the size of Δ, u and v, it was a worthwhile cost. Both `create_matrix.f90` and `initial.f90` were not changed after these edits. There was not a noticeable way to optimise these at all. For testing of this part of the code, I edited `grayscott.f90` to save the matrix Δ instead of u and v and not perform any other task. This allowed me to focus on Δ and visually see if it was correct. This was checked against the original sequential code. Visually, it looked correct and this was enough of a confirmation for now. This was further tested when `Mat_Mult` was tested. This testing process was repeated with `initial.f90`, where u and v were saved and checked against the correct outputs for the sequential code provided.

Moving onto Q2. For this section I had to start from scratch, as I hadn't done the `sparsegather` routine before this. My idea was to build upon the code that was already there for the sequential case. In order to do this, I would need to send and receive the values from u and v that were going to be used by the other processors. This was done with the use of the `MPI_Send` and `MPI_Recv` subroutines. I also had to call in the comm rank and the comm size to be able to complete the code (`MPI_Send` and `MPI_Recv` were later updated to `MPI_SendRecv`). Then to start the program, the send and receives were performed and then the original `Mat_Mult` algorithm was called. This ensured that the correct elements were in the bands of the local u and v. To test this subroutine, I used a combination of Matlab and Fortran. In Matlab, I wrote a function to multiply a given vector by the Δ matrix (this was the full form of Δ and not the CRS version). This was then able to be compared to the version in Fortran. This was also compared to the sequential version again. Also additional tests were performed for different initial conditions for u and v. This proved particularly useful as I had an indexing error and setting both u and v equal to all 1's allowed me to pinpoint where this error was coming from (turns out it was from a little overlap in the `MPI_Send` function and the -4 from Δ was being used twice in some rows). In all these tests, I started off using a very small m (m=4) and worked my way up for first the single processor case and then for 2 processors and continuing up from there. This allowed me really to find exactly where the errors were coming from. There were still recurring errors after this so I employed the use of the Fortran debugger to solve these issues. I was able to set breakpoints where I thought the errors were and pinpoint exactly what was going on (turns out it was in a test program I wrote and I was getting NaNs from).

Q3 was about implementing a subroutine `rhs.f90` which would calculate r_u and r_v. As explained in the code this was done by calculating r_u and r_v in 3 steps: the first partial calculation, calling daxpy to continue the partial calculations and then a call of `Mat_Mult` to add the additional Δ multiplication onto the end. Originally this was done with a do loop to calculate the first partial.

This was edited later to use Fortran's inbuilt element-wise vector multiplication and further changed to reuse values already calculated for r_v in r_u to save on time on memory calls and additional flops. Similarly a daxpy call was added too to optimise this part of the code. I was lucky enough not to have errors here but I still tested the code against the sequential case to confirm that the r_u and r_v were properly calculated. For this `grayscott.f90` was edited again to perform the rhs call instead of the timestepping one and then the r_u and r_v were outputted and saved. This allowed for easy comparison between Matlab results and previous ones in Fortran.

Q4 was about implementing the Heun method in `timestepping.f90`. This was virtually a carbon copy of Algorithm 1. Line 3 was performed using `rhs.f90`, line 4 with daxpy, line 5 with `rhs.f90` again and line 6 with two calls to daxpy for the addition of r_u and $\hat{r_u}$ and r_v and $\hat{r_v}$ as well as the addition to $\hat{u}$ and $\hat{v}$ and multiplication by $\frac{\tau}{2}$. This was only tested in conjunction with the final `grayscott` program. This was due to the `timestepping` algorithm being the main part of the `grayscott` program and the outputting was already coded into the `grayscott` program so there was no additional need to write a custom test program for this.

Q5 was about modifying the main program in `grayscott.f90` to be able to run in parallel. First I implemented the reading and broadcasting of the input variables from the `input.dat` file. Also MPI functionality was added throughout and I ended up calling `MPI_Init` to initialise the setup as well as calling the comm rank and comm size. The `input.dat` file was then read and the results broadcasted from the node with $myid = 0$. All the local indices for u,v and Δ's ibeg and iend were calculated and set. Delta,u and v were then populated and then the time stepping algorithm was performed. The final u and v were then gathered using `MPI_Gather` onto the node with $myid = 0$ and then saved using the `save_fields` subroutine. The program was then finalised and the vectors deallocated. While testing this originally, I had forgotten to add a `MPI_Gather` call for the u and v and ended up just with the first little chunk of the output after using `visualise.py`. Luckily this was an easier error to fix! To test this, I started by setting the values for T and τ small enough that only one iteration would occur. This was to be able to test `timestepping` to prove that this was properly working. This was tested for a wide range of values for u and v which I already had the answers for. This allowed me to be able to push the limits of this part of the code and make sure that the answers were correct still. From this, I then added another processor and tested on 2. This allowed for checking for certain segmentation errors. The same process of checking against known values was done and then the number of processors was increased suitably.

Part II: Theory

Q6: Performance Model

Estimate the number of FLOPs and the number of memory references in one step of Algorithm 1 as functions of m. Using the values of t_{flop}, t_{mem}, t_{lat} and t_{word} from the **Timings handout**, estimate the time t_{seq} of one Heun step as a function of m.

Measure the total time and deduce the time of one time step. Compare the theoretically predicted time with the measured sequential time for $m = 128$ and $m = 512$.

Now estimate in a similar way the communication time for each process in one Heun step, again as a function of m and the number of processes P. Plot the theoretically predicted computational time t_{par} for the parallel implementation of Algorithm 1 and the parallel efficiency $E(n, P)$ (where $n = m^2$) for both $m = 128$ and $m = 512$, and the numbers of processes $P = 1, 2, 4, 8, 16, 32$. (*You can use any plotting software.*)

How do the results change if the performance improvements from Q10 are implemented?

[6 points]

Insert your answer here and attach plots on separate sheets

We will assume a perfect caching result here and use it as a lower bound to our practical results. We have used a similar system to `sparsegather` for our `matmult.f90`. The difference is the addition of the b vector after and hence:

$$\begin{aligned} t_{Mat\_Mult} &= t_{sparsegather} + t_{vector\_addition} \\ &= (13 * t_{mem} + 10 * t_{flop}) * n + 2 * t_{lat} + 2 * m * t_{word} + n * t_{mem} + 2 * n * t_{flop} \\ &= (14 * t_{mem} + 12 * t_{flop}) * n + 2 * t_{lat} + 2 * m * t_{word} \end{aligned} \tag{14}$$

We have the additional $t_{vector\_addition}$ due to the multiplication by alpha and the accessing of b once (not necessary to do this twice assuming perfect caching). Calling `RHS` once is:

$$\begin{aligned} t_{RHS} &= t_{ru/rv_population} + 2 * t_{daxpy} + 2 * t_{Mat\_Mult} \\ &= (5 * t_{mem} + 4 * t_{flop}) * n + 2 * (3 * t_{mem} + 2 * t_{flop}) * n + 2 * ((14 * t_{mem} \\ &\qquad + 12 * t_{flop}) * n + 2 * t_{lat} + 2 * m * t_{word}) \\ &= (39 * t_{mem} + 32 * t_{flop}) * n + 4 * t_{lat} + 4 * m * t_{word} \end{aligned} \tag{15}$$

Finally we add up the time for one iteration of `timestepping`:

$$\begin{aligned} t_{timestepping} &= 2 * t_{RHS} + 6 * t_{daxpy} \\ &= 2 * ((39 * t_{mem} + 32 * t_{flop}) * n + 4 * t_{lat} + 4 * m * t_{word}) \\ &\qquad + 6 * (3 * t_{mem} + 2 * t_{flop}) * n \\ &= 96 * n * t_{mem} + 76 * n * t_{flops} + 8 * t_{lat} + 8 * m * t_{word} \\ &= 96 * m^2 * t_{mem} + 76 * m^2 * t_{flops} + 8 * t_{lat} + 8 * m * t_{word} \end{aligned} \tag{16}$$

So for the times given in the handout for this the sequential code we get:

Q6 Theory (T=30)

nrprocs	M = 128	Per iteration	M = 512	Per iteration
Practical 1 proc	**0.5504**	0.0005504	**8.7436**	**0.0087436**
Theoretical (without lat)		0.000510001152		0.008160018432
Theoretical (with lat)		0.000513939392		0.008168211392

Figure 1: Table of results from testing

We observe here that our times (both theoretical and in practice) are of the same magnitude with the theoretical time a slight faction less than the practical one. (Note here that these times are for T=30 due to the long calculation time of T=3000). For the communication time for each process in one Heun step is independent of P as always $2*m$ elements are sent over per process and `Mat_Mult` is repeated 4 times. The latency time is also the same per processor. These times are in total:

- $t_{comm} = 4 * (2 * t_{lat} + 2 * m * t_{word}) = 0.00000787648$ (m=128)
- $t_{comm} = 0.00001638592$ (m=512)

Insert your answer here and attach plots on separate sheets

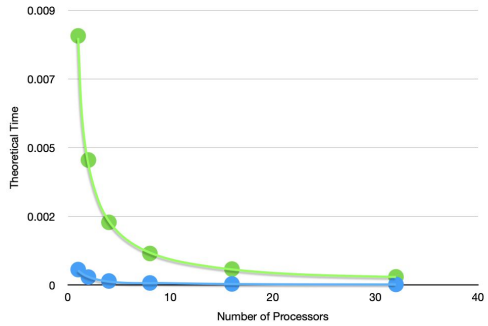

Figure 2: Theoretical Time against P

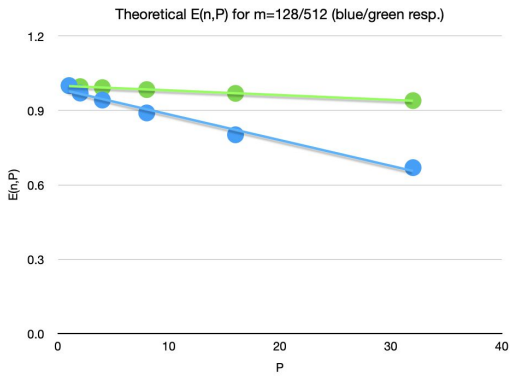

Figure 3: Theoretical Parallel Efficiency against P

P	Theoretical Time		Theoretical Efficiency	
	m=128	m=512	m=128	m=512
1	0.000510001152	0.008160018432	1.0	1.0
2	0.000262877056	0.004096395136	0.97003740029712	0.995999917132995
4	0.000135376768	0.002056390528	0.941818082109923	0.99203170809392
8	0.000071626624	0.001036388224	0.890034186170774	0.98418939966651
16	0.000039751552	0.000526387072	0.801857296037144	0.968870968016481
32	0.000023814016	0.000271386496	0.669250243218113	0.939621461489375

Figure 4: Table of results from testing from plots

We can see that for higher m, we get that the theoretical parallel efficiency remains high over a wider range of processors. In other words, the problem is strongly scalable , particularly for larger problem sizes.

For Q10, we don't expect the theoretical time to go down but rather the practical one due to the removal of the creation of Δ. Also the non-blocking MPI calls will cause the time to go down. The time improvements are compared in Q10 with plots to show the % reduction in time against the number of processors and the size of m.

Part III: Consistency and Scaling Tests

In this section, use $m = 128$, $\tau = 0.4$, $T = 3000$, $F = 0.04$, $k = 0.059$, $D_u = 2 \cdot 10^{-5}$, $D_v = 10^{-5}$ unless otherwise specified.

Q7: Empirical error analysis

Run your code (you can use one processor in this question) for $\tau = 0.05, 0.1, 0.2, 0.4$, printing the solution $u^{(\tau)}_{n/2,n} \approx u(1/2, 1, T)$, corresponding to each value of τ. Compute the error estimates $u^{(\tau)}_{n/2,n} - u^{(2\tau)}_{n/2,n}$ and the experimental order of convergence (EOC)

$$\text{EOC}_\tau = \log_2 \left(\frac{u^{(\tau)}_{n/2,n} - u^{(2\tau)}_{n/2,n}}{u^{(2\tau)}_{n/2,n} - u^{(4\tau)}_{n/2,n}} \right).$$

Populate the following table with the values you obtained.

τ	$u^{(\tau)}_{n/2,n}$	$u^{(\tau)}_{n/2,n} - u^{(2\tau)}_{n/2,n}$	EOC_τ
0.05	0.694865230052	0.000003152845	-1.988900553
0.1	. 0.694862077207	0.000012514726	-1.976557626
0.2	. 0.694849562481	0.000049252069	—
0.4	. 0.694800310412	—	—

Plot the solutions $\mathbf{u}, \mathbf{v}$ by invoking `python visualise.py` in the command line after running your program for $m = 128$. Copy `solution.pdf` to your H: drive and attach it below.

Comment on the results here and attach the plot on a separate sheet

As we double τ, we have that the difference between consecutive values of $u^{(\tau)}_{\frac{n}{2},n}$ quadruples. Hence the EOC_τ reduced by a factor of approximately -2 each time that τ is halfed. This implies that the convergence rate is $\mathcal{O}(\tau^2)$. This is again consistent with the work done in the first coursework.

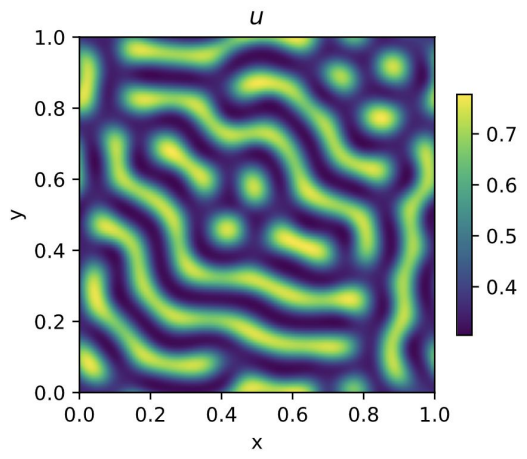

Figure 5: U solution

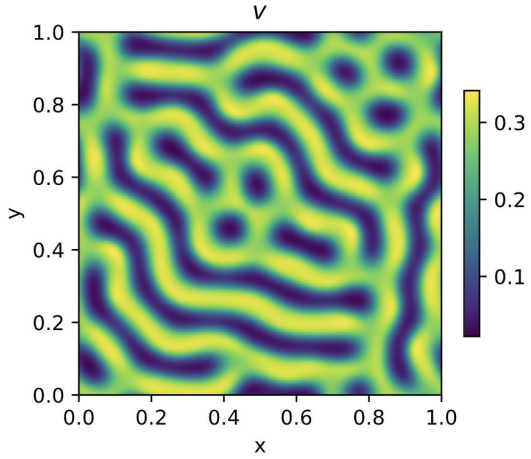

Figure 6: V Solution

[3 points]

Q8: Strong scaling

Set $\tau = 0.03$, $T = 300$, $m = 512$ and calculate the solution on $P = 1, 2, 4, 8, 16, 32$ processes. Verify that the result does not change significantly (compared to Table in Q7) for different P. Measure the time spent in one step of Algorithm 1 for different numbers of processes (hint: measure the total time and divide by the number of steps). Plot both the time and the parallel efficiency $E(n, P)$, where $n = m^2$. Repeat the experiment for two different values of m (e.g. $m = 128$ and $m = 512$). Compare the measured times to the theoretical estimates in Q6.

Attach scaling plots on a separate sheet

[4 points]

Q9: Weak scaling

Keep $\tau = 0.03$ and $T = 300$. Starting from $m = 128$, successively double the number of points m in each direction and at the same time increase the number of processes by a factor of four (i.e. keep the problem size on a single process fixed). Perform this experiment for $P = 1, 4, 16$ and for $P = 2, 8, 32$ processes and measure the time spent in one step of Algorithm 1. Plot this time as a function of the number of processes P and plot the scaled efficiency $E_s(n, P)$.

Attach scaling plots on a separate sheet

[4 points]

Write a short report, summarising your results in Q8 and Q9

Starting with Q8, I have done the testing for m=128,512. Below are the tables related to this and the plots (blue representing m=128 and green, m=512. This is true for this whole section).

Q8 Strong scaling - m=128

P	u_{n/2,n}^{\tau}	Time taken	Time/step	Speedup	Par eff
1	0.999979875139	5.3472	0.00053472	-	-
2	0.999979875139	2.6012	0.00026012	2.05566661540827	1.02783330770414
4	0.999979875139	1.4222	0.00014222	3.75980874701167	0.939952186752918
8	0.999979875139	0.8113	0.00008113	6.59090348822877	0.823862936028596
16	0.999979875139	0.5191	0.00005191	10.3009054132152	0.64380658832595
32	0.999979875139	0.4889	0.00004889	10.9372059725915	0.341787686643484

Figure 7: Table for m=128

Q8 Strong scaling - m=512

P	u_{n/2,n}^{\tau}	Time taken	Time/step	speedup	Parr eff
1	0.999979515163	85.5560	0.0086	-	-
2	0.999979515163	48.3082	0.00483082	1.77104508137335	0.885522540686675
4	0.999979515163	23.8527	0.00238527	3.58684761054304	0.89671190263576
8	0.999979515163	11.7623	0.00117623	7.27374748135994	0.909218435169993
16	0.999979515163	5.9232	0.00059232	14.4442193408968	0.90276370880605
32	0.999979515163	3.0011	0.00030011	28.5082136549932	0.890881676718538

Figure 8: Table for m=512

Insert your answer here and attach plots on separate sheets

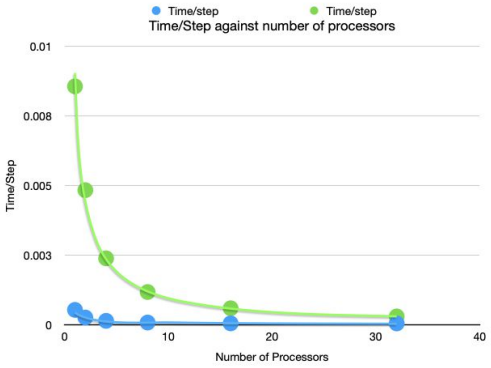

Figure 9: Time per step against number of processors

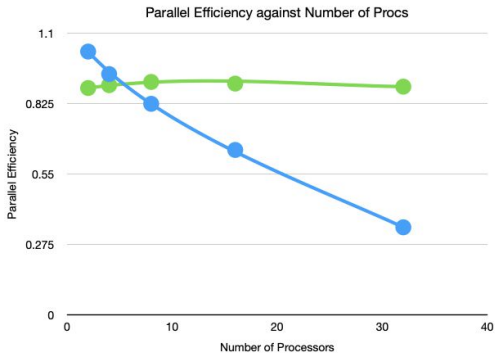

Figure 10: Parallel Efficiency against processors

This is what we expect. Noting the value of u that is in the tables, it doesn't change significantly and this is a good sign that the parallelisation process has worked properly. Comparing the theoretical times to those shown here, we get again that they are really similar in both cases (m=128 and m=512). Interestingly the times are m=512 are closer to the theoretical and even more interestingly the actual time for m=128,P=2 actually beats the theoretical one. This is because computers don't work with perfect caching and have better ways to optimise. We get that for the larger problem size (m=512) the parallel efficiency remains high as the number of processors increases. This peaks for 8 processors and starts to tend off and will go down as the processor count increases. For the small problem size (m=128) we have that the parallel efficiency tends towards 0 much faster. In conclusion we have that for larger problem sizes, the codes scales much more strongly. (In the parallel efficiency plot, the values for P=1 are left out for clarity (they would be equal to 1 otherwise)).

For small problem sizes we have that the speed-up reduces quickly compared to Amdahl's optimal speed-up and this implies that this does not strongly scale and an appropriate number of processors to use for this problem size is one or two (this having the efficiency over 80%). For the larger problem size this scales nicely against Amdahl's ideal and remains at approximately 80% for a wide range of processors and this implies that the number of processors can be increased without being less efficient.

For Q9, the results are in the two tables below and the two plots below them.

Q9 Weak Scaling

m	p	u_{n/2,n}^{\tau}	Time taken	Time/step	Weak scaling efficiency
128	1	0.999979875139	5.3472	0.00053472	1
256	4	0.999979572050	5.6445	0.00056445	0.94732925857029
512	16	0.999979515163	6.6425	0.00066425	0.804998118178397

128	2	0.999979875139	2.5931	0.00025931	1
256	8	0.999979572050	2.8239	0.00028239	0.91826906051914
512	32	0.999979515163	3.0809	0.00030809	0.841669641987731

Figure 11: Table of results from testing

Write a short report, summarising your results in Q8 and Q9

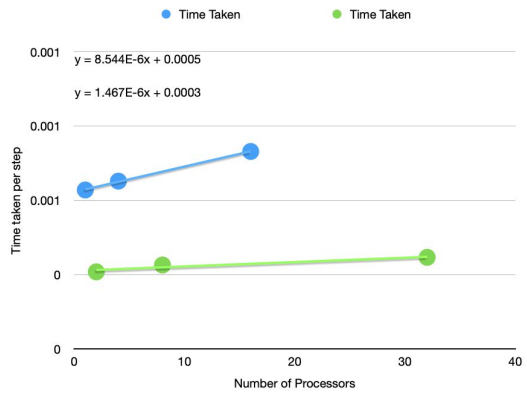

Figure 12: Time/Step against P

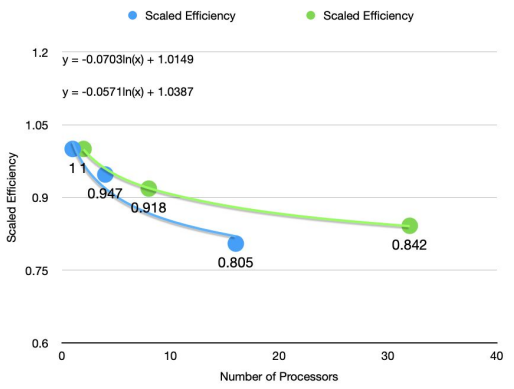

Figure 13: Scaled Efficiency against P plot

Looking at the plots we can see from the left one that we get that the time increases though very slightly (equations given for the trendlines). The parallel efficiency stays high and seems to follow a logarithmic trend (as the trendlines suggest). The `timestepping` algorithm is very memory-bound and hence we see that adding cores keeps the efficiency high. For applications that scale perfectly weakly, the work done on each node remains the same as the scale of the machine increases. I.e. we are solving progressively larger problems in the same time as it takes to solve the smaller ones. This is mirrored in the time plot as the gradient of the trendlines is very very low and this implies that the time taken to perform the task stays approximately the same and hence the task is almost perfectly weakly scaling.

Q10: Enhancements

Improve the performance of your code. For example you might want to consider the following:

- Adjust the parallel vector data structures presented in the lectures such that each process requires memory for only its local part as well as for the two halos (above and below).
- Overlap calculations and communications using non-blocking MPI sends and receives. You can also try to make communication more efficient by sending several vectors in one message.
- Memory references and divisions are around ten times more expensive than multiplications, additions or subtractions. Exploit this fact by writing a matrix-free version of the algorithm. To do this write a new subroutine for applying the matrix $\mathbf{\Delta}$ to a vector that avoids explicit storage of the matrix and too many divisions.

For any performance enhancement you make, write a short report on how you implemented it, and provide plots or tables that confirm the improvement.

[7 points]

Insert discussion of further enhancements and attach plots here.

To do this, I started with the first task. Originally, I used the format (lower halo, local part, upper halo) but swapped over to (local part, lower halo, upper halo). This was because the readability of the code was much easier. Implementing this improvement was mostly changing the indices throughout the code. In `grayscott.f90` the only changes were to the allocation sizes and the `MPI_Gather`. The allocation size is now: number of rows + 2*m. `MPI_Gather` needed to be adjusted to not copy over the halos, just copy over the first nrows (=number of rows) elements. Next I edited `initial.f90`. This was done again to adjust over the indices. I added a variable k which adjusted the indices from u%ibeg to 1 and u%iend to nrows resp. Within `timestepping.f90`, the ru, ruhat, rv and rvhat vectors were edited to be of the size nrows. This removed the unnecessary storage that wasn't used. Also the daxpy calls here were edited to be over the appropriate bounds. Again this was done in `rhs.f90`. For this first part, I edited `matmult.f90` to work with these shortened vectors. This was done by again changing the indices. This was all that was edited for the first part of the improvements. To check that the code was functional, I was able to compare this to the previous part of the assignment. This was done step at a time to pinpoint errors that arose. So I started with 1 processor and small m and worked up to 32 processors with m=512.

The next improvement that I made was the third suggestion. This was to remove the implicit creation of Δ. I did this before the previous suggestion as this would be able to take advantage of the non-blocking MPI calls. I first did this by adding the 3 center u_i's and then adding the halos later. This was effective but I was accessing each b(i) twice and this was inefficient on the whole (the old code was left in `matmult.f90`, commented out). To combat this, I separated out the if statements to allow for 3 different cases. When i is greater than m and less than nloc-m+1 we have the case that the data in the halos is not used at all. This is then able to be calculated while we wait for the additional halo information for the other cases. I explicitly wrote out all the lines of code in `matmult.f90`. This was to remove the Aurow variable and remove the accessing time of Aurow multiple times. Through testing, I found out that the halo data was on the processor way before the center part of b%xx was calculated. (This was true for nprocs<m). So therefore there was no downtime on the processor. To avoid divisions, instead of using $\frac{1}{h^2}$, m^2 was used. Additionally, I inputted m into the `Mat_Mult` function as well as the parent functions. This allowed me to not have to calculate it in `matmult.f90` removing a costly sqrt operation.

The second suggestion was implemented the same time as the third one. As stated, the halo information could be sent over using the non-blocking calls for `MPI_Send` and `MPI_Recv` during the calculations of the middle part of b%xx without any downtime on the processor. `MPI_Wait` was then called between the two halves of the code and it then carried on with the calculations of the other parts of b%xx. The testing throughout this section was done by comparing the outputs for different values of τ, T, etc against the output of the first part of this assignment. As stated in the first paragraph, testing the improvements was done in comparison to the non-improved version.

(From here the blue line represents the standard code and the green line represents the improved. This is the case when they both appear on the same graph). To test and confirm the improvement, I set the number of processors equal to 32 and changed the value of m in multiple of 64 up to 512. This is shown in the table and plots below.

Insert discussion of further enhancements and attach plots here.

Q10 - improvements table and plot - standard

Standard			Improved			Improvement %
m	time	Value	m	time	Value	
64	1.9676	0.618303498254	64	1.8008	0.618303498254	0.0847733279121773
128	3.9372	0.694865905056	128	3.1057	0.694865905056	0.211190693894138
192	5.7589	0.568202919775	192	4.5839	0.568202919775	0.20403202000382
256	9.7400	0.521959878723	256	7.0315	0.521959878723	0.278080082135524
320	13.0383	0.506346065817	320	9.8091	0.506346065817	0.247670325119072
384	18.2265	0.499703877838	384	13.4564	0.499703877838	0.261712341919732
448	23.1801	0.496268228609	448	17.5042	0.496268228609	0.24486089361133
512	29.1904	0.494252562985	512	22.4158	0.494252562985	0.232083150624863

Figure 14: Table of results from testing

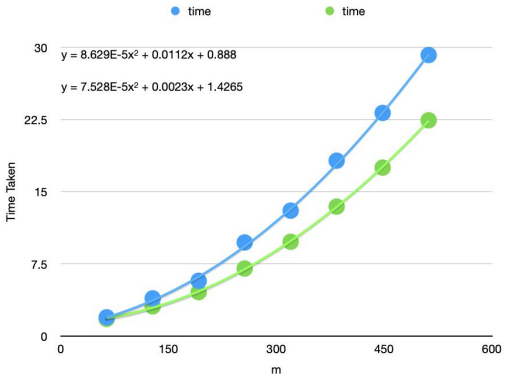

Figure 15: Time against m plot

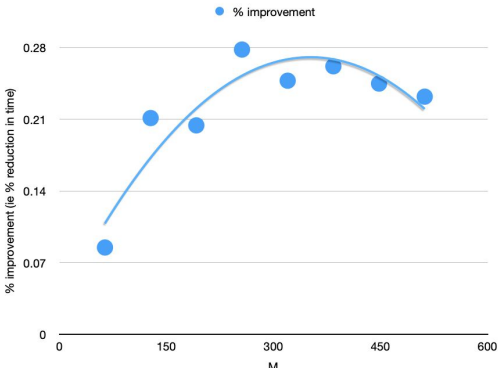

Figure 16: % improvement against m

As we can see here, initially as m increases so does the percentage reduction in time. This is due to the number of memory references decreasing rapidly relative to the size. Hence why for m=64, this is much lower as the difference is at its lowest. After m=320, there is a clear down trend in the % reduction in time. This is due to the communication between the processors increasing linearly and causing a decrease in the reduction of time. M was only tested up to 512 due to the problem not converging for values of m much bigger (due to $\tau > \frac{1}{4}$). On the plots above, polynomial trend lines are fitted and we can see from the equations on the first plot that the slopes are very close to linear. This is what we expect from out enhancements ans this is confirmed by the second plot being relatively flat for % values being around 25%.

If we do the same for a fixed m (128) and change the number of processors we get:

m=128

Standard			Improved			% reduction
P	time	u	P	Time	U	
1	53.3253	0.694865905056	1	39.9882	0.694865905056	0.25010829756232
2	25.6795	0.694865905056	2	20.3803	0.694865905056	0.206359158083296
4	13.9387	0.694865905056	4	11.1501	0.694865905056	0.200061698723697
8	7.7891	0.694865905056	8	6.2352	0.694865905056	0.199496732613524
16	4.8150	0.694865905056	16	3.7911	0.694865905056	0.212647975077882
32	3.7433	0.694865905056	32	3.0172	0.694865905056	0.193973232174819

Figure 17: Table of results from testing

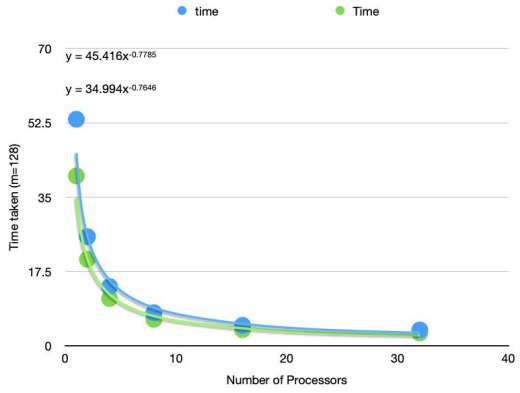

Figure 18: Time against #Processors

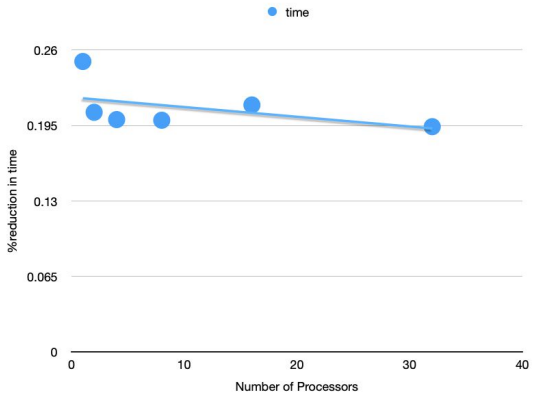

Figure 19: % improvement #Processors

We can see here that the improvement levels stay about the same with a reduction of above 20% for most of the results. This is a clear indicator that our improved code is in fact improved.

General Remarks

- Write routines and programs to test parts of your code.
- Support your observations with numerical results. You can also use graphs.
- Typical trends for convergence and cost with respect to some variable n are αn^{β} and $\alpha\beta^{n}$ for some constants $\alpha, \beta \in \mathbb{R}$.
- Use as many subroutines and functions as you deem necessary.
- Choose helpful names for variables, subroutines, functions, etc.
- Provide comments in your code where appropriate.
- Make sure your code is properly indented. Your text editor may be able to help you with this.
- "Programs must be written for people to read, and only incidentally for machines to execute.", Abelson & Sussman, Structure and Interpretation of Computer Programs

- Together with your Fortran code and Makefile, provide a `README` file explaining how to compile and run your programs.
- Try to make sure that it is easy to repeat the experiments you used to verify your code and to generate the results you use in your report.