High Performance Computing, Project

Pranav Rao, 15008121, UCL

March 29, 2019

Chapter 1

Stage 1

1.1 Introduction

The task was to write a finite difference scheme using OpenCL to solve the poisson equation:

$$-\nabla \cdot (\sigma(x,y)\nabla)\, u(x,y) = f(x,y) \tag{1.1}$$

for $(x, y) \in [0,1] \times [0,1]$ with boundary conditions $u(x,y) = 0$. A poisson equation is a partial differential equation that has many broad applications, such as describing the potential field created by a given set of charges or masses. As a result given the potential field, it is possible to determine the electric/gravitational field that caused it.

After the equation had been discretised and implemented in OpenCL, scipy iterative solvers, gmres and bicgstab were to be used to find solutions to the above equation. The solutions from the finite difference scheme were then validated by a finite element method implemented using FEniCS. Finally the rate of convergence of the various iterative solvers were studied and analysed.

1.2 Method

1.2.1 OpenCL Finite Difference Discretisation

The finite difference discretisation that was implemented in OpenCL for the poisson equation was as follows:

$$\begin{aligned}\nabla \cdot (\sigma(x,y)\nabla)\, u \approx{}& \frac{\left(\sigma_{i+1/2,j}\frac{(u_{i+1,j}-u_{i,j})}{h}\right) - \left(\sigma_{i-1/2,j}\frac{(u_{i,j}-u_{i-1,j})}{h}\right)}{h} \\ &+ \frac{\left(\sigma_{i,j+1/2}\frac{(u_{i,j+1}-u_{i,j})}{h}\right) - \left(\sigma_{i,j-1/2}\frac{(u_{i,j}-u_{i,j-1})}{h}\right)}{h}\end{aligned} \tag{1.2}$$

In the above equation $\sigma_{i+1/2,j}$ is taken as the average of $\sigma_{i,j}$ and $\sigma_{i+1,j}$.

This discretisation uses a 5 point stencil to evaluate each point of the solution vector u. This means $u^{n+1}_{i,j}$ depends on $u^n_{i,j}$, $u^n_{i+1,j}$, $u^n_{i,j+1}$, $u^n_{i-1,j}$ and $u^n_{i,j-1}$, where n is the n^{th} iteration of the solution.

Equation (2) was implemented into an OpenCL kernel, using pyopencl. The python function containing the OpenCL kernel, which returned u_{n+1} was then passed to the `scipy.sparse.linalg .LinearOperator` function to convert the function into a linear operator object.

The boundary conditions were implicitly set in the kernel by the following method: if $(i, j = 0)$ then $(u_{i-1,j},\ u_{i,j-1} = 0)$ or if $(i, j = N - 1)$ then $(u_{i+1,j},\ u_{i,j+1} = 0)$, where N is the number of unknowns.

The number of unknowns, N, was 2 less than the number of discretisation points, because the boundary conditions were known, and as a result not passed to the kernel whilst solving the poisson equation.

1.2.2 Scipy Iterative Solvers

The Scipy iterative solvers used were GMRes and BiCGSTAB, which both required a LinearOperator objeect as an argument, which in this case was the python function running the OpenCL kernel.

1.2.3 Validation using FEniCS

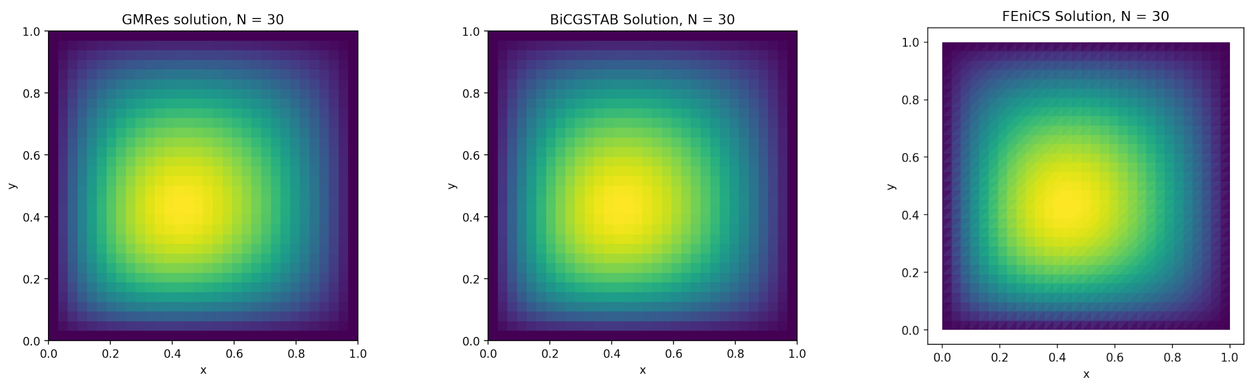
The finite element discretisation that was implemented in FEniCS. To discretise the poisson equation (1) we multiply the right and left hand side with a test function v which satisfies the same boundary conditions as u. By applying Green's theorem and noticing the second term vanishes due to boundary conditions of u the following expression is determined:

$$\int_\Omega [\sigma(\vec{x})\nabla u(\vec{x})]\cdot v(\vec{x})dx = \int_\Omega f(\vec{x})v(\vec{x})dx \tag{1.3}$$

Whilst implementing this in FEniCS the RHS of the above equation is represented as $a(u, v)$ and the LHS is represented as L.

To validate the finite difference method that was written using OpenCL, $\sigma(\vec{x}) = 1 + x^2 + y^2$ and $f(\vec{x}) = 1$ were chosen as our field and test functions. Both the finite difference method (using gmres and bicgstab) and finite element method were then run using $N = 30$. Figure 1.1 shows the output u for all three methods.

Figure 1.1 passes the "eye-test" that the finite difference method developed matches with FEniCS. To further support this statement, the FEniCS output of u was exported and viewed in Paraview.



(a) GMRes (b) BiCGSTAB (c) FEniCS

Figure 1.1: Final converged u values for the 3 methods.

The values of specific indices were compared between GMRes, BiCGSTAB and FEniCS. The absolute values of u at the chosen indecies is shown in table 1.1 and the percentage difference between each finite difference method and finite element method is shown in table 1.2.

u(i,j)	**GMRes Values**	**BiCGSTAB Values**	**FEniCS Values**
u(1,1)	0.0019155	0.0019155	0.0019157
u(10,10)	0.0427900	0.0427903	0.0427966
u(22,16)	0.0339292	0.0339296	0.0342288
u(15,15)	0.0459224	0.0459230	0.0462709

Table 1.1: Comparing the values of u determined by GMRes BiCGSTAB and FEniCS, for N=30

Table 1.1 shows that gmres and bicgstab converge to almost exactly the same value, with all $u(i,j)$ being consistent to 5 decimal places. This allows the conclusion, that the iterative solvers from scipy that were used converge to the same value.

When comparing the finite difference methods with the finite element method, the values of $u(i,j)$ are consistent to 5 decimal places near the boundary, and 2 decimal places near the centre. The reason for this is that the for both finite difference method and finite element method, the boundary was defined as 0, as a result the $u(i,j)$ values near the boundary are significantly consistent with each other. This small error near the boundary builds up with each successive element away from the boundary, and near the centre the values are least consistent.

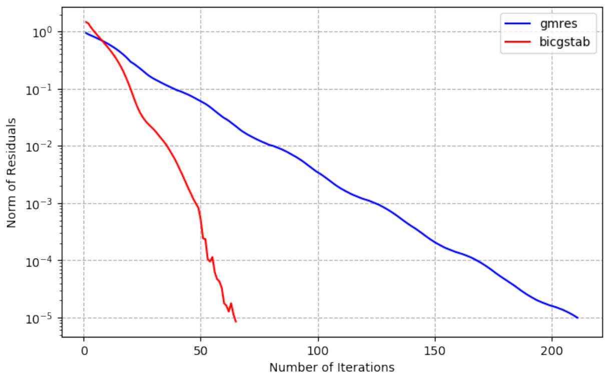
u(i,j)	**GMRes Values**	**BiCGSTAB Values**
u(1,1)	0.010%	0.010%
u(10,10)	0.015%	0.015%
u(22,16)	0.87%	0.87%
u(15,15)	0.75%	0.75%

Table 1.2: Percentage error of GMRes and BiCGSTAB compared to FEniCS, for N=30

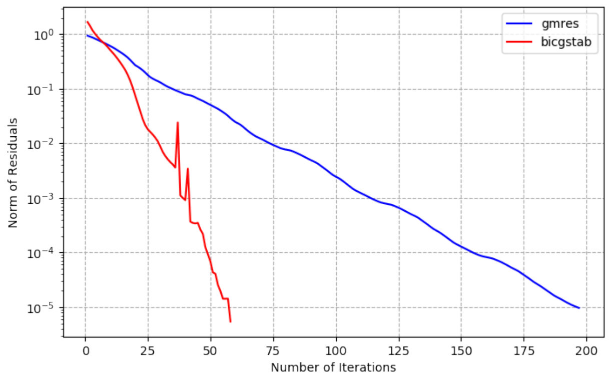
This inconsistency between FEniCS and the iterative solvers can be seen extremely clearly in 1.2 which shows the percentage differences. However for $u(16, 16)$ the absolute difference is the highest, but the percentage difference is not. This is due to the $\sigma(\vec{x})$ that was chosen not being symmetrical, and as a result the u value at $u(22, 16)$ is smaller than that at $u(16, 16)$.

1.3 Analysis

To analyse how the iterative solvers perform the norm of the residuals was plotted after each iteration for both gmres and bicgstab, shown in figure 1.2. The norm of the residuals for gmres was straightforward to calculate, because the `callback` function uses the vector of residuals. For bicgstab however it was more complex because the `callback` function uses the vector of the current u values. In order to calculate the norm of the residuals the formula: $\frac{||f-Ax||_2}{||f||_2}$ was used.



(a) Residuals for $\sigma(\vec{x}) = 1 + x^2 + y^2$



(b) Residuals for normally distributed sigma.

Figure 1.2: Residual plots for GMRes and BiCGSTAB for N = 30

It is seen that bicgstab converges ins ignificantly lower number of iterations for both defined and normally distributed σ. The line for gmres, in both cases is smoother than that of bicgstab, this is due the the fact that gmres with each successive iteration is attempting to minimise the residual.

A similar trend is noticed in figure 1.3, where bicgstab converges significantly faster, in this case in 8 times less number of iterations.

Figure 1.4 is a different performance measure of the iterative solvers, it shows the relationship between the number of discretisation points and the number of iterations for convergence. Once again it is evident that bicgstab is significantly faster than gmres.

The exponential curves were fitted for both iterative solvers, by taking the `np.log10` of number of iterations and number of discretisation points, which resulted a linear relationship, and then using `scipy.stats.linregress` to fit a line. For gmres the fitted line was: $y = 1.881407x - 0.458444$ and for bicgstab the fitted line was: $y = 1.018744x + 0.296047$. To plot the lines as exponential function,

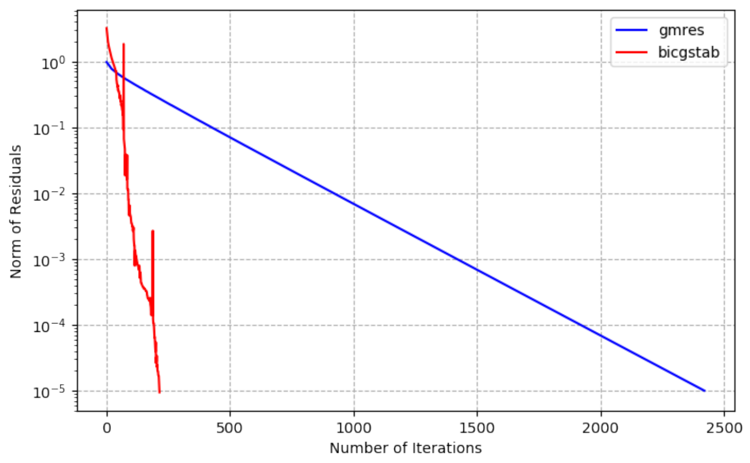


Figure 1.3: Residual plot for GMRes and BiCGSTAB for N=100 for normally distributed σ

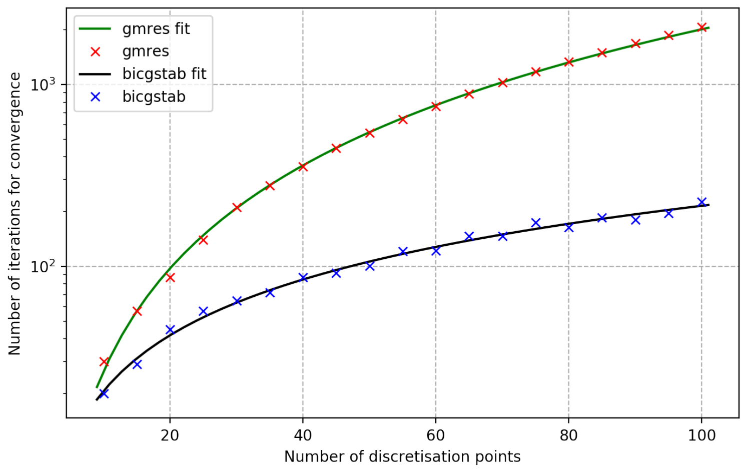


Figure 1.4: Convergence relationship with Discretisation points for gmres and bicgstab.

10^y was calculated and plotted.

1.4 Conclusion

The implementation of the 5 point stencil finite difference scheme was successful in converging to the same values as the finite element scheme implemented in FEniCS. Upon analysing the convergence plots of gmres and bicgstab, it was noted that for this specific problem bicgstab was performing significantly better.

Chapter 2

Stage 2

2.1 Introduction

The aim of the second stage of the project was to write an OpenCL implementation of a forward difference scheme which would model diffusion. The OpenCL kernel from stage 1 was modified to carry out the time evolution.

2.2 Method

2.2.1 OpenCL implementation

For the diffusion process to occur, a time discretisation needed to be added. The forward difference scheme that was used to calculate each successive time step was:

$$u(x,y,t+\Delta t) = \frac{du(x,y,t)}{dt}\Delta t + u(x,y,t) \tag{2.1}$$

where Δt is the discreet time step and $\frac{du}{dt}$ is the first derivative of u with respect to time and is defined by the following equation:

$$\frac{du(x,y,t)}{dt} = \nabla\cdot(\sigma(x,y)\nabla)u(x,y,t) \tag{2.2}$$

The field $\sigma(x,y)$ that was chosen was a normally distributed random field.

Equation (2.2) is almost exactly the same as the 5 point stencil that was implemented in OpenCL for stage 1, with the only difference being the negative sign. As a result each successive time step can be calculated by modifying the kernel from stage 1, multiplying the result by the time step Δt and adding the current value of u.

By combining eq (2.1) and eq (2.2) we get the equation:

$$u_{n+1} = (1 + \Delta t A)u_n \tag{2.3}$$

where n is the n^{th} timestep. The eigenvalues for the operator are then defined as; $1 + \Delta t\lambda_k$. This method is extremely sensitive to the Δt that is chosen, and only values of Δt which satisfy this condition: $|1 + \Delta t\lambda_k| < 1$ converge. All λ_k for this operator are real and negative, and hence Δt would have to be positive, which makes sense in terms of the axis of time. The value of Δt is then bounded by $\Delta t < \frac{2}{|\lambda_{max}|}$, where λ_{max} is the eigenvalue with the greatest magnitude. The value of λ grows as a function of $\frac{1}{h^2}$, where h is the spatial discretisation, and as a result Δt should be a function of h^2.

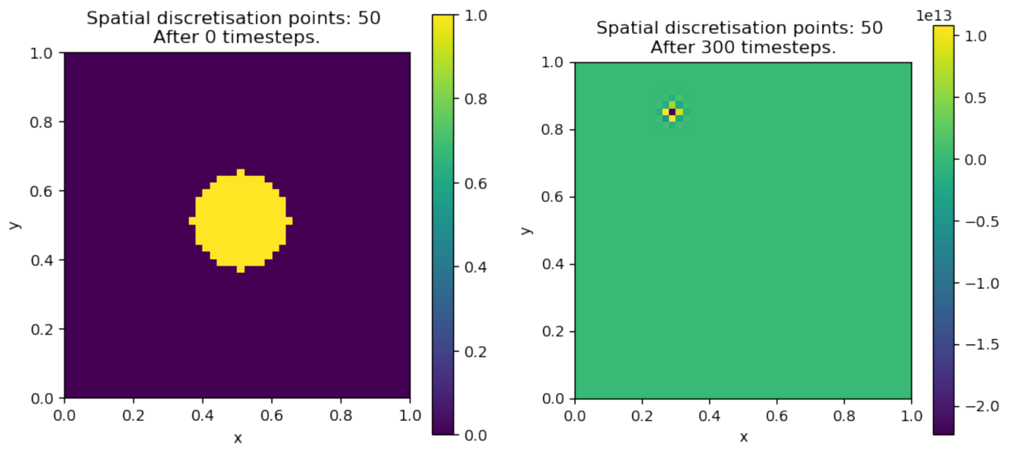


Figure 2.1: Time evolution plot for N=50, which explodes due to the Δt which was being too large. With circular initial weight.

The function of Δt that was chosen was: $\Delta t = \frac{h^2}{2}$ because it was the most consistent value for which the time evolution function did not explode. Fig 2.1 shows the evolution which blows up, in this case $\Delta t = h^2$ was chosen, with an initial circular weight. A similar result is shown in fig 2.2 which has the same parameters except for the initial weight being the shape of a square.

Fig 2.3 and fig 2.4 show the successful diffusion process, run with $\Delta t = \frac{h^2}{2}$.

2.3 Conclusion

To successfully model the time evolution diffusion process a small Δt needs to be chosen. The disadvantage for this is the fact that it requires many small time steps, which only get smaller as the size of our problem increases. As a result to witness diffusion many time steps need to be calculated, and this increases as a function of N^2 where N is the number of spatial discretisation points.

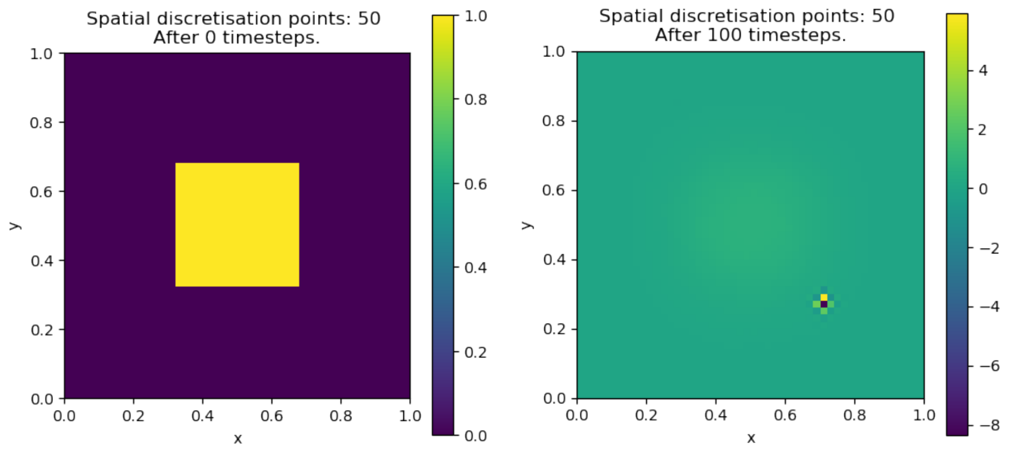


Figure 2.2: Time evolution plot for N=50, which explodes due to the Δt which was being too large. With square initial weight.

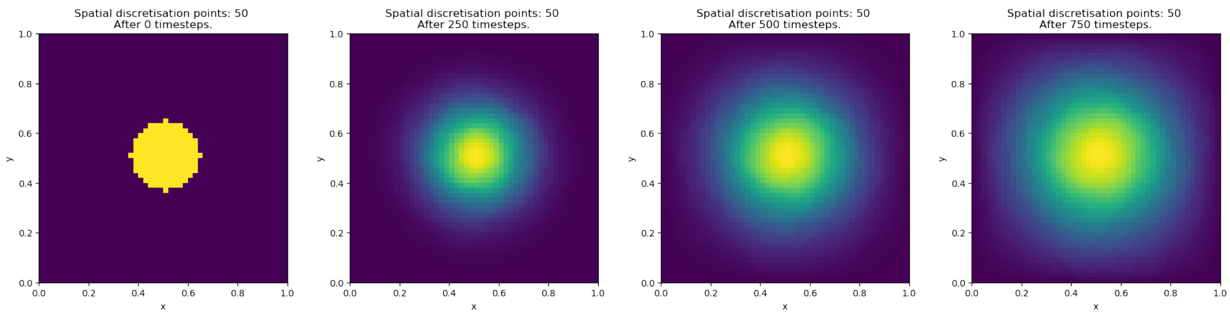


Figure 2.3: Time evolution plots for N=50 with circular initial weight.

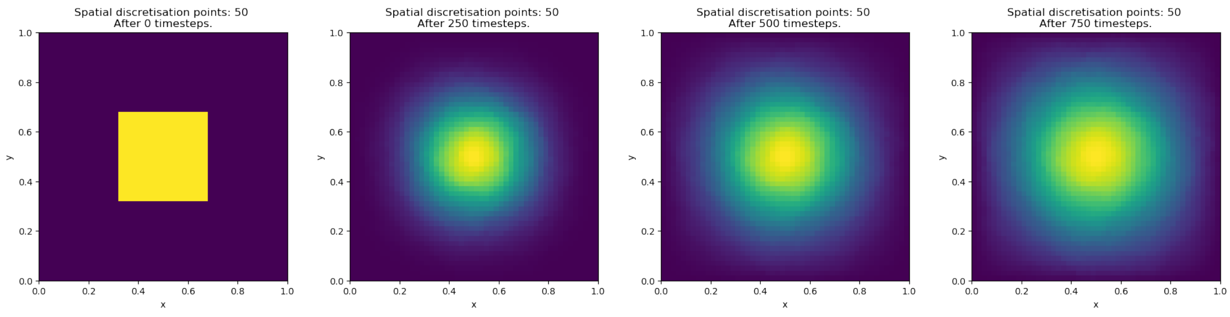


Figure 2.4: Time evolution plots for N=50, with square initial weight.