Hybrid parallel solution to the non-linear Schödinger equation in 3D using finite differences

Matthew Richards, 44805599

November 14, 2019

Contents

Chapter 1

Problem Description & Serial Implementation

1.1 Problem Statement

The Schrödinger project involves solution of the Gross Pitaevski equation, a nonlinear extension of the time dependent Schrödinger equation. It is a PDE in 3 spatial dimensions alongside time and can be used to model Bose Einstein condensates (BECs). A BEC is a collection of atoms cooled to near absolute zero, which form a fifth state of matter (the others being solid, liquid, gas and plasma). To describe the approach taken to obtain a solution to the Gross Pitaevski equation, the key steps are discussed first in one dimension for the sake of clarity. The formal mathematical statement of the problem is as follows:

$$\begin{cases} i\hbar\dfrac{\partial}{\partial t}\psi(\boldsymbol{x},t) = -\dfrac{\hbar^2}{2m}\nabla^2\psi(\boldsymbol{x},t) + V(\boldsymbol{x})\psi(\boldsymbol{x},t) + g|\psi(\boldsymbol{x},t)|^2\psi(\boldsymbol{x},t) \\ \psi(\boldsymbol{x},0) = \varphi(\boldsymbol{x}) \\ \quad |\varphi(\boldsymbol{x})| \to 0 \text{ as } |\boldsymbol{x}| \to \infty. \end{cases} \tag{1.1}$$

where ψ is complex valued and $g \in \mathbb{R}$. $V(\boldsymbol{x})$ is some external potential applied to the system which is independent of the system's evolution (however may still be time dependent) To simplify things further, we consider rescaling of the units such that $m = \hbar = 1$ and let $\widehat{V}$ collectively denote the external potential V and the nonlinear interaction term $g'|\psi(\boldsymbol{x},t)|^2$. Expressing this in 1D we have.

$$i\frac{\partial}{\partial t}\psi(x,t) = -\frac{1}{2}\frac{\partial^2}{\partial x^2}\psi(x,t) + \widehat{V}(x)\psi(x,t). \tag{1.2}$$

1.2 Implementation

1.2.1 Finite Difference Method Theoretical Background

The central concept for the approach taken is the finite difference approximation to the derivative of a function, which are derived from the Taylor series. We use the forward and backward differences for the first derivative, which come from manipulating the Taylor expansion about x of $f(x+h)$ and $f(x-h)$ respectively:

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h) \qquad f'(x) = \frac{f(x) - f(x-h)}{h} - \mathcal{O}(h). \tag{1.3}$$

The central difference expression for the second derivative produces a more accurate result and obtained from the expansion of $f(x+h) - f(x-h)$ about x yielding the following:

$$f''(x) = \frac{f(x-h) - 2f(x) + f(x+h)}{h^2} + \mathcal{O}(h^2) \tag{1.4}$$

We can then substitute these approximations into (1.2) to yield the following discrete approximation. We note that extension of finite differences from derivatives to partial derivatives is straightforward and can easily be seen from the limit definition of a partial derivative;

$$i\frac{\psi_j^{n+1} - \psi_j^n}{\Delta t} \approx -\frac{1}{2}\frac{\psi_{j-1}^n - 2\psi_j^n + \psi_{j+1}^n}{(\Delta x)^2} + \widehat{V}(x_j)\psi_j^n, \tag{1.5}$$

where j denotes the spatial index, n the temporal index and Δx and Δt are the steps sizes for the respective domains. In this notation, $x_j = x_l + j\Delta x$ where x_l is the leftmost grid point. Note that the first order expression is used for the temporal partial derivative as this means our equation is able to be rearranged such that $\psi(x_j, t+\Delta t)$ can be determined purely in terms the previous time step t. This is required, as we only have the initial condition $\varphi(x)$ for a single time interval, namely $t = 0$. The above schema is called the forward time centred space approximation and can be solved to yield an explicit solution for ψ_j^{n+1}. The problem with such schema is that for some problems it is unconditionally unstable, as errors in the solution are iteratively amplified. In other cases, a restrictive condition is placed on the relationship between the step sizes Δx and Δt to ensure stability. A potential solution to this is to use the backwards difference for the time derivative, which produces the implicit method as follows:

$$i\frac{\psi_j^{n+1} - \psi_j^n}{\Delta t} \approx -\frac{1}{2}\frac{\psi_{j-1}^{n+1} - 2\psi_j^{n+1} + \psi_{j+1}^{n+1}}{(\Delta x)^2} + \widehat{V}(x_j)\psi_j^{n+1} \tag{1.6}$$

This is now a coupled system of N_x equations in N_x unknowns and requires the solving a matrix system at each time step. Whilst no longer unstable, this method is still not particularly accurate. The Crank-Nicholson method uses the fact that the LHS of (1.5) and (1.6) are the same, so $\frac{1}{2}((1.5) + (1.6))$ will yield a mixed implicit explicit scheme with error of $\mathcal{O}(N_x^2 + N_t^2)$ [1] (pg 385):

$$i\frac{\psi_j^{n+1} - \psi_j^n}{\Delta t} = \frac{1}{2}\left(-\frac{1}{2}\frac{\psi_{j-1}^{n+1} - 2\psi_j^{n+1} + \psi_{j+1}^{n+1}}{(\Delta x)^2} + \widehat{V}(x_j)\psi_j^{n+1}\right) + \frac{1}{2}\left(-\frac{1}{2}\frac{\psi_{j-1}^n - 2\psi_j^n + \psi_{j+1}^n}{(\Delta x)^2} + \widehat{V}(x_j)\psi_j^n\right)$$

Rearranging to put the $(n+1)$ terms on the left and (n) terms on the right:

$$\psi_j^{n+1} - \frac{i}{4}\frac{\Delta t}{(\Delta x)^2}(\psi_{j-1}^{n+1} - 2\psi_j^{n+1} + \psi_{j+1}^{n+1}) + \frac{i\Delta t}{2}\widehat{V}_j^{n+1}\psi_j^{n+1} = \psi_j^n + \frac{i}{4}\frac{\Delta t}{(\Delta x)^2}(\psi_{j-1}^n - 2\psi_j^n + \psi_{j+1}^n) - \frac{i\Delta t}{2}\widehat{V}_j^n\psi_j^n \tag{1.7}$$

To tidy up notation, let $\alpha := \frac{i}{4}\frac{\Delta t}{(\Delta x)^2}$, and $\beta := \frac{1}{2}i\Delta t$:

$$\psi_j^{n+1} - \alpha(\psi_{j-1}^{n+1} - 2\psi_j^{n+1} + \psi_{j+1}^{n+1}) + \beta\widehat{V}_j^{n+1}\psi_j^{n+1} = \psi_j^n + \alpha(\psi_{j-1}^n - 2\psi_j^n + \psi_{j+1}^n) - \alpha\widehat{V}_j^n\psi_j^n \tag{1.8}$$

We can then express these coupled systems as a matrix system;

$$\begin{bmatrix} 1+2\alpha+\beta\widehat{V}_1^{n+1} & -\alpha & 0 & \dots & 0 \\ -\alpha & 1+2\alpha+\beta\widehat{V}_2^{n+1} & -\alpha & \ddots & \vdots \\ 0 & -\alpha & 1+2\alpha+\beta\widehat{V}_1^{n+1} & \ddots & 0 \\ \vdots & & \ddots & \ddots & -\alpha \\ 0 & \dots & 0 & -\alpha & 1+2\alpha+\beta\widehat{V}_{N_x}^{n+1} \end{bmatrix} \begin{bmatrix} \psi_1^{n+1} \\ \psi_2^{n+1} \\ \vdots \\ \vdots \\ \psi_{N_x}^{n+1} \end{bmatrix} - \alpha \begin{bmatrix} \psi_l^{n+1} \\ 0 \\ \vdots \\ 0 \\ \psi_r \end{bmatrix}$$
$$= \begin{bmatrix} 1-2\alpha-\beta\widehat{V}_1^{n} & \alpha & 0 & \dots & 0 \\ \alpha & 1-2\alpha-\beta\widehat{V}_2^{n} & \alpha & \ddots & \vdots \\ 0 & \alpha & 1-2\alpha-\beta\widehat{V}_3^{n} & \ddots & 0 \\ \vdots & & \ddots & \ddots & \alpha \\ 0 & \dots & 0 & \alpha & 1-2\alpha-\beta\widehat{V}_{N_x}^{n} \end{bmatrix} \begin{bmatrix} \psi_1^{n} \\ \psi_2^{n} \\ \vdots \\ \vdots \\ \psi_{N_x}^{n} \end{bmatrix} + \alpha \begin{bmatrix} \psi_l^{n+1} \\ 0 \\ \vdots \\ 0 \\ \psi_r \end{bmatrix}. \tag{1.9}$$

For our particular application, we consider wave functions which disperse as $x \to \infty$, and so $\psi_l^n = \psi_r^n = 0 \,\forall\, n$, which removes the vector of constants. In vector notation we can write this as:

$$\begin{aligned} A_l \boldsymbol{\psi}^{n+1} &= A_r \boldsymbol{\psi}^n \\ \implies \boldsymbol{\psi}^{n+1} &= {A_l}^{-1} A_r \boldsymbol{\psi}^n \end{aligned} \tag{1.10}$$

1.2.2 Translation to Code

This is enough background to begin discussing the actual implementation. For every time period we construct the matrix system in (1.9/1.10), where we note that $A_r\boldsymbol{\psi}^n$ does not need to be constructed as a matrix (2D array); the matrix vector product can be computed in a for loop, where only the resultant vector need be stored. The matrix A_l is tridiagonal and as such is encoded as 3 arrays. The main diagonal is updated with the new potential for each correspondent time period. As a very minor optimisation, the above system has identical upper and lower diagonals so we can save a small amount of memory. These are then passed to a tri-diagonal matrix system solver to obtain $\boldsymbol{\psi}^{n+1}$.

1.2.3 Non-linear Potential Approximation

This procedure is marginally complicated by the fact that the potential $\widehat{V}_j^{n+1} = V(x_j) + g\big|\psi_j^{n+1}\big|\psi_j^{n+1}$ is dependent on ψ_j^{n+1}; which is a component of the vector we are trying to solve for. To alleviate this, we approximate $\widehat{V}_j^{n+1} \approx V(x_j) + g|\psi_j^n|\psi_j^n$, which is exact for vanishingly small Δt. We use this approximation to determine some $(\boldsymbol{\psi}^{n+1})'$. This yields an iterative update schema (Note that $\mapsto$ should be read as yielding a result, it does not indicate assignment):

$$\begin{aligned} \widehat{\boldsymbol{V}}^{n+1}[\boldsymbol{\psi}^n] &\mapsto (\boldsymbol{\psi}^{n+1})' \\ \widehat{\boldsymbol{V}}^{n+1}[(\boldsymbol{\psi}^{n+1})'] &\mapsto (\boldsymbol{\psi}^{n+1})'' \\ \widehat{\boldsymbol{V}}^{n+1}[(\boldsymbol{\psi}^{n+1})''] &\mapsto (\boldsymbol{\psi}^{n+1})''' \\ &\vdots \end{aligned}$$

Practically, we achieve this by wrapping the above mentioned matrix construction in a for loop, introducing an auxiliary array to hold the current $(\boldsymbol{\psi}^{n+1})^{(i)}$ and judiciously swap array pointers. To evolve through time, this is all nested within a for loop to avoid extra copies.

1.2.4 Adaptation to 3D

Finally, this procedure must be adapted to handle three spatial dimensions. We introduce the notation $\delta_m^2\psi_j^n$ to denote the central difference approximation to $\frac{\partial^2}{\partial m^2}\psi_j^n$ and revisit Equation 1.8 including the terms from the gradient in 3D:

$$\psi_j^{n+1} - \alpha(\delta_x^2\psi_j^{n+1} + \delta_y^2\psi_j^{n+1} + \delta_z^2\psi_j^{n+1}) + \beta\widehat{V}_j^{n+1}\psi_j^{n+1} = \psi_j^n + \alpha(\delta_x^2\psi_j^n + \delta_y^2\psi_j^n + \delta_z^2\psi_j^n) - \alpha\widehat{V}_j^n\psi_j^n$$

Clearly, this will no longer produce a tri-diagonal system of equations to solve as $\delta_m^2\psi_j^n$ contains the points neighbouring j in the mth spatial dimension. Taking the approximation suggested in Numerical Recipes [2] (p. 1052-1053), we break the update rule into sequentially applied linear operators. This means that we only deal with one spatial component of the gradient term at a time. As such, we now solve 3 consecutive linked 1D problems in each time period:

$$\begin{cases} \psi_j^{n+1} - \alpha(\delta_x^2\psi_j^{n+1}) + \beta\widehat{V}_j^{n+1}\psi_j^{n+1} = \psi_j^n + \alpha(\delta_x^2\psi_j^n) - \alpha\widehat{V}_j^n\psi_j^n & \psi_j^n \mapsto \dot{\psi}_j^n \\ \dot{\psi}_j^{n+1} - \alpha(\delta_y^2\dot{\psi}_j^{n+1}) + \beta\widehat{V}_j^{n+1}\dot{\psi}_j^{n+1} = \dot{\psi}_j^n + \alpha(\delta_y^2\dot{\psi}_j^n) - \alpha\widehat{V}_j^n\dot{\psi}_j^n & \dot{\psi}_j^n \mapsto \ddot{\psi}_j^n \\ \ddot{\psi}_j^{n+1} - \alpha(\delta_z^2\ddot{\psi}_j^{n+1}) + \beta\widehat{V}_j^{n+1}\ddot{\psi}_j^{n+1} = \ddot{\psi}_j^n + \alpha(\delta_z^2\ddot{\psi}_j^n) - \alpha\widehat{V}_j^n\ddot{\psi}_j^n & \ddot{\psi}_j^n \mapsto \psi_j^{n+1} \end{cases} \tag{1.11}$$

Collectively these operations produce $\boldsymbol{\psi}^{n+1}$, noting that the dot notation is an indication of an iterative update, rather than the time derivative as in common convention. To apply this in code, local arrays consisting of the sequenced 1D problems are created from the master 3D array (which is just a 1D array with 3D indexing). These arrays are then passed to a routine which solves a single time step evolution in 1D. Whilst some of these additional copies could be saved, memory can only be contiguously accessed with reference to one spatial dimension. It was decided consistency in this approach was more worthwhile than minute memory optimisations. The use of local 1D arrays also provides the benefit that the single dimensional solve does not need to compute indices in 3D space to solve a 1D problem.

The potential passed to the 1D solver only contains the contribution term from that particular dimension, not the entire potential. For symmetric potentials, the contribution is equally weighted between the three spatial dimensions. Although the potential could be applied in an aggregate fashion, this approach made the most sense with the specification of independent dimensional trap frequencies.

1.2.5 Considerations on Non-linear Potential Iterations in 3D

The iterative procedure to determined the unknown future time in the nonlinear potential is applied similarly to the 1D case described in Section 1.2.3, excepting that a single iterative loop wraps all 3 steps. This means that there is only additional memory overhead to store the wave function at the previous full step and arrays of the partial spatial evolutions operations are not stored.

Some considerations have been given as to how to implement the iterated component around the 3D partial solves. Initially, the following sketched procedure was attempted to try and iteratively update the nonlinear potential (for a single time step):

```
for k=1:n_iter
  do_x_one_dim_evolution(ψ_old, ψ_iter_prev) ↦ ψ̇
  do_y_one_dim_evolution(ψ̇, ψ_iter_prev) ↦ ψ̈
  do_z_one_dim_evolution(ψ̈, ψ_iter_prev) ↦ ψ_iter_new
  ψ_iter_prev = ψ_iter_new
end
```

Whilst this does appear to work to an extent, there is some numerical dissipation of the norm (for 1000 time steps it reduced to 0.997). This is a marginal difference but not ideal. This does however make sense, as $\boldsymbol{\psi}_{\text{iter\_new}}$ is used in the iterative side for every single step. However this is the iterative approximation to the entire new wave function, not the directional contribution of each sub-step. One obvious alternative is to introduce a $\boldsymbol{\psi}_{\text{iter}}$ for each partial step of the way and to use these iteratively. However this will introduce the memory overhead of needing to store each of these vectors. Another simple solution would be to only pass the iterative $\boldsymbol{\psi}$ vector to the first direction, where the full components of the wave function contribution are expected. However, this means that any iterative development in the the second and third directions is just flow on from the first, they are non iterative steps (although they depend on an iterative one).

Eventually it was decided that it would be the most accurate to iteratively evolve each dimensional step in isolation, as presented in the following pseudo-code:

```
for k=1:n_iter
do_x_one_dim_evolution(ψ_old, ψ_iter_prev) ↦ ψ_iter_prev
end
ψ̇ := ψ_iter_prev
for k=1:n_iter
do_y_one_dim_evolution(ψ̇, ψ_iter_prev) ↦ ψ_iter_prev
end
ψ̈ := ψ_iter_prev
for k=1:n_iter
do_z_one_dim_evolution(ψ̈, ψ_iter_prev) ↦ ψ_iter_prev
end
ψ_new := ψ_iter_prev
```

However, this was not actually implemented as this leaves no convenient way to aggregate the iteration error for the specified output quantities. Either the individual iterative errors would need to be summed or a an additional copy of the entire wave function after a full 3 fold iteration would need to be stored to compare against. So the solution implemented was to pass the iterative term only into the first directional evolution, and have the subsequent feed through update the other directions.

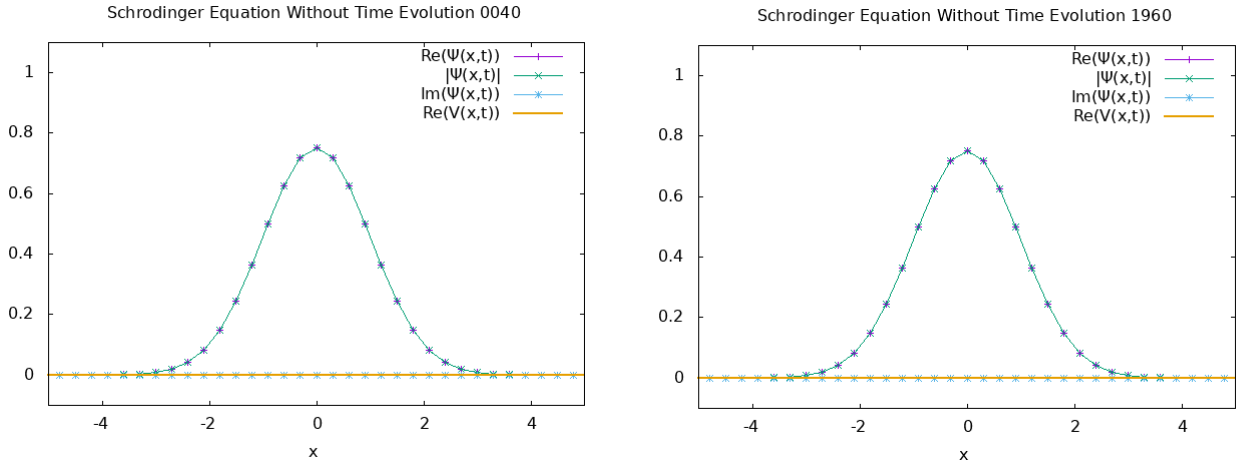
1.2.6 Initial Conditions and Normalisation

To incorporate the initial condition, the array of $\boldsymbol{\psi}$ values is populated at each grid point according to the specified $\varphi(\boldsymbol{x})$, before the time loops begin. The resulting $\boldsymbol{\psi}(t = 0)$ is then numerically integrated and rescaled such that it has a norm of 1. This is important as it means that $|\boldsymbol{\psi}|^2$ has a meaningful interpretation as a probability density between 0 and 1.

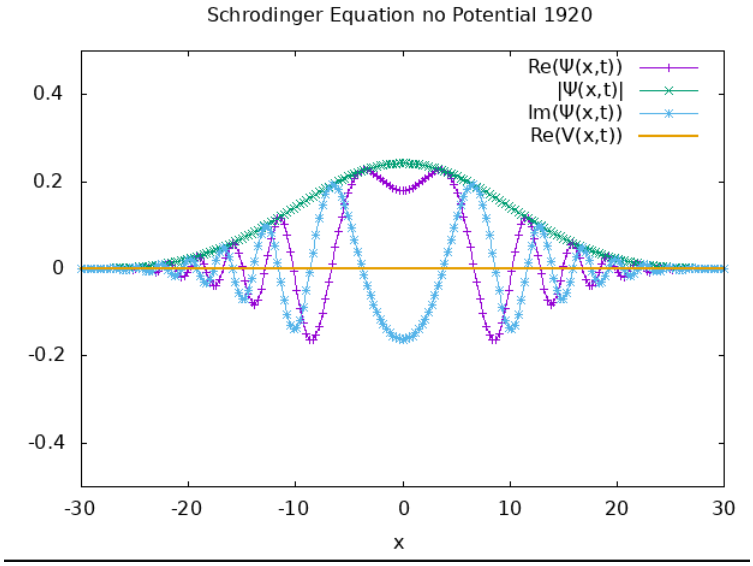
1.3 Verification of Implementation

To verify the implementation, the 1D case was first checked as it forms the basis for the 3D solution. Additionally, the wave function can actually be visualised in 1D and there are analytical solutions for certain potentials in 1D. The following test cases all use a gaussian initial condition of the form $f(\boldsymbol{x}; \sigma_0) = \sqrt{\frac{1}{\sqrt{\pi}\sigma_0}} \exp\left(-\frac{\boldsymbol{x}\cdot\boldsymbol{x}}{2\sigma^2}\right)$ (which is also rescaled numerically). Additionally, for the sake of consistency, these tests are taken with zero external potential, $\sigma_0 = 1$, $x_l = -60$, $x_r = 60$, $N_x = 299$ (the number of internal grid points), $dt = 0.005$ and $N_t = 2000$ unless otherwise specified.

1. $dt = 0$: The point of this is to verify memory is being indexed correctly and that the system is static with no time evolution. In the below image we see that the code does mirror the expected behaviour, slices of the output animation are taken to display the static nature of the the solution. The numbers in the title of the plot reflect an animation frame corresponding to the time step (out of 2000 steps).



2. Parameters as listed: For this case we still have no additional energy introduced into the system and so expect to see heat equation type dispersion. Additionally, we expect that the the norm of the wave function would be conserved as we are not eliminating particles from the system. This corresponds to the output produced, there is wave dispersion over time (the initial wave packet is identical to the $dt = 0$ case). The norm is also conserved to at least 8 digits.



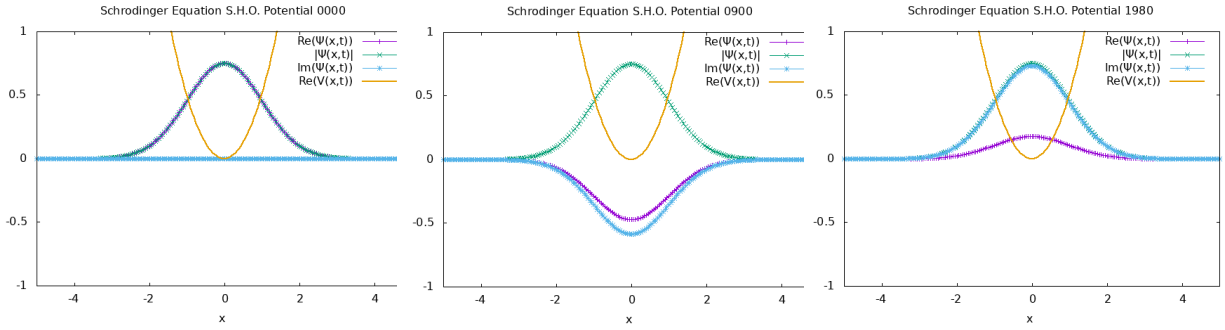
3. The next case introduces a potential to the system, namely the harmonic oscillator:

$$V(x) = \frac{1}{2}\omega^2 x^2$$

This is a good test case as so far we have not tested the code with a nonzero potential. Additionally, the harmonic oscillator has time independent analytic solutions. According to [3], in non-dimensionalised form, the solution is:

$$u_0(x) = \left(\frac{\omega}{\pi}\right)^{\frac{1}{4}} \exp(-\omega x^2/2)$$

in ground state. We note that the leading order behaviour is the decaying exponential profile which is time independent. Running the code with $\omega = 1$, we get the following plots



Note that these use 399 points with x boundaries of ± 10. We see that the wave function is time independent as would be hoped, suggesting the potential is included correctly. It is also worth noting that due to this configuration, the displacement squared is stable, $\langle x^2(t) \rangle \in (0.4997, 0.5] \, \forall \, t$ and the standard deviation correspondingly is stable around $0.707 \approx \frac{1}{\sqrt{2}}$. Since this would make a good comparison case in 3D, (as we would expect this behaviour to generalise to higher dimensions) it was examined what happens to these quantities on crude grid that would be more feasible to solve over in higher dimensions. More notable fluctuations were exhibited with the wave function appearing to "bounce". The square of displacement ranged between 0.34 and 0.5 and standard deviation ranged between 0.58 and 0.707.

4. Finally we test the full Gross-Pitaevski equation (Schrödinger with the interaction non-linearity) by searching for a soliton solution as suggested. For this case, we initialise the wave function with

$$\psi(x, 0) = \varphi(x) = \frac{1}{\cosh\left(\frac{\pi}{2\sqrt{3}} \frac{x}{\sigma}\right)}$$

and set $g = -1$. Without being overly specific for the choice of width σ_0, setting it equal to 2.5 yields a solution where $\langle x^2(t) \rangle$ is stable at 3.125 for the first 1600 intervals and $\sigma(t_n) = 1.767$ for the same range. So it appears that the cubic nonlinearity can produce a standing wave as would be by the theory, and it is anticipated that this convergence would improve with iterations for convergence.

5. Moving on to the 3D case, many of these tests can be directly repeated, albeit with the inability to visualise the wave function, as it becomes a complex function of 4 variables. Again we initially test the $dt = 0$ case and find that the wave function is unitary and does not disperse as would be hoped. For the 3D tests we use a small gird of 39 points between -20 and 20. For the sake of simplicity the grid is constructed uniformly in every dimension. Given that the initial condition is a gaussian and symmetric this is sensible. Checking a non-zero dt with this crude grid yields dispersion as would be expected and retains the unitary norm. In fact, it comparing with the 1D case, the values at each iteration are directly correspondent to those in 1 dimension. As additional verification, the squared displacement and standard deviation are also output in the y direction as well as x. These values are are symmetric as would be expected.

6. Considering the harmonic oscillator in 3D with the crude grid, there is notably more fluctuation in displacement squared and standard deviation than in the above presented

case. Comparing the 3D and 1D output when run on the same spatial grid, the numerical output comparing is now marginally different. Despite this, there is a clear direct correspondence that can be seen the plot of the standard deviations. The 1D x and 3D x and y direction curves are basically indistinguishable when plotted on the entire time domain.

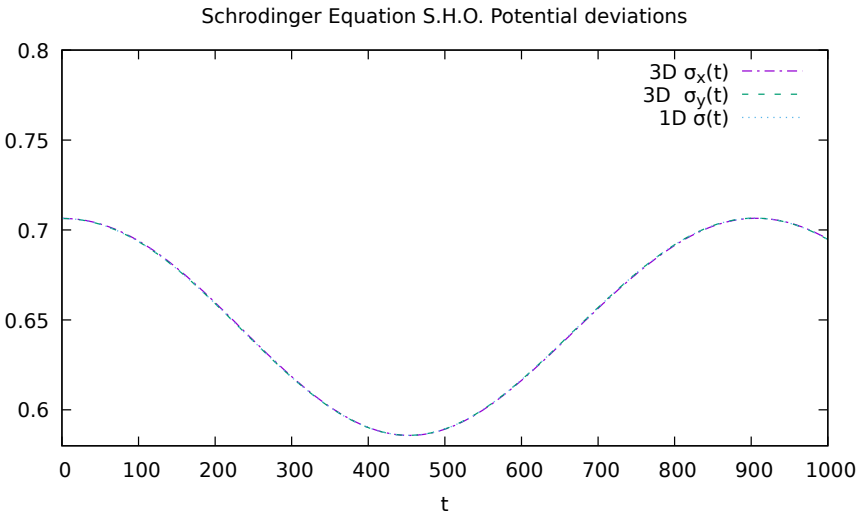


Figure 1.3

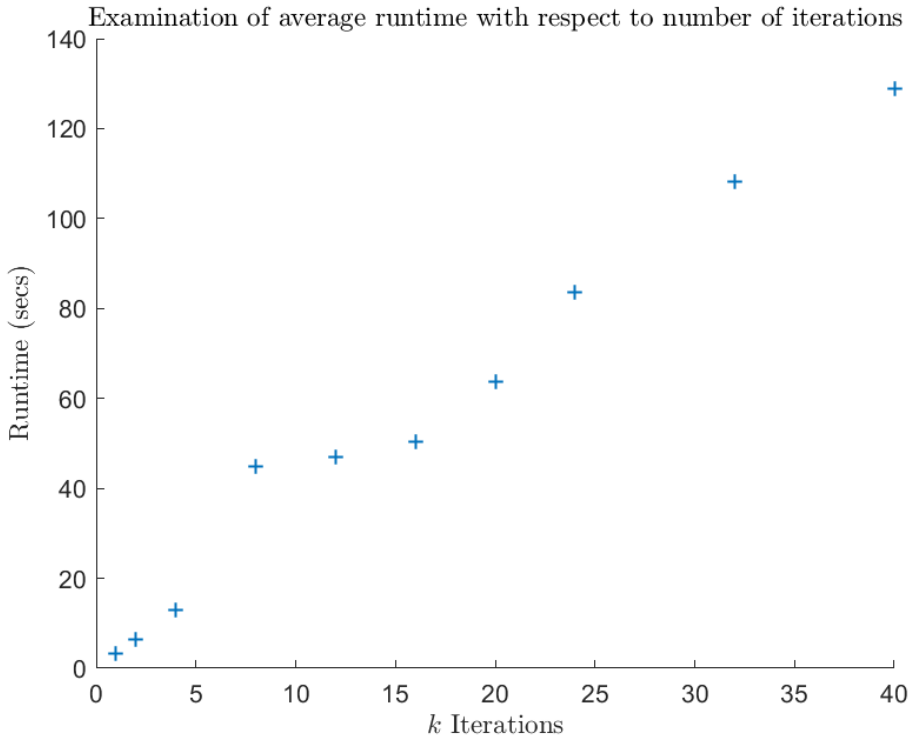
1.4 Performance

Optimiser Flag	None	-O1	-O2	-O3	-O4
Average Wall Time (s)	146.0	76.0	75.7	72.6	71.6

Table 1.1: Average Reported Wall Time in Seconds

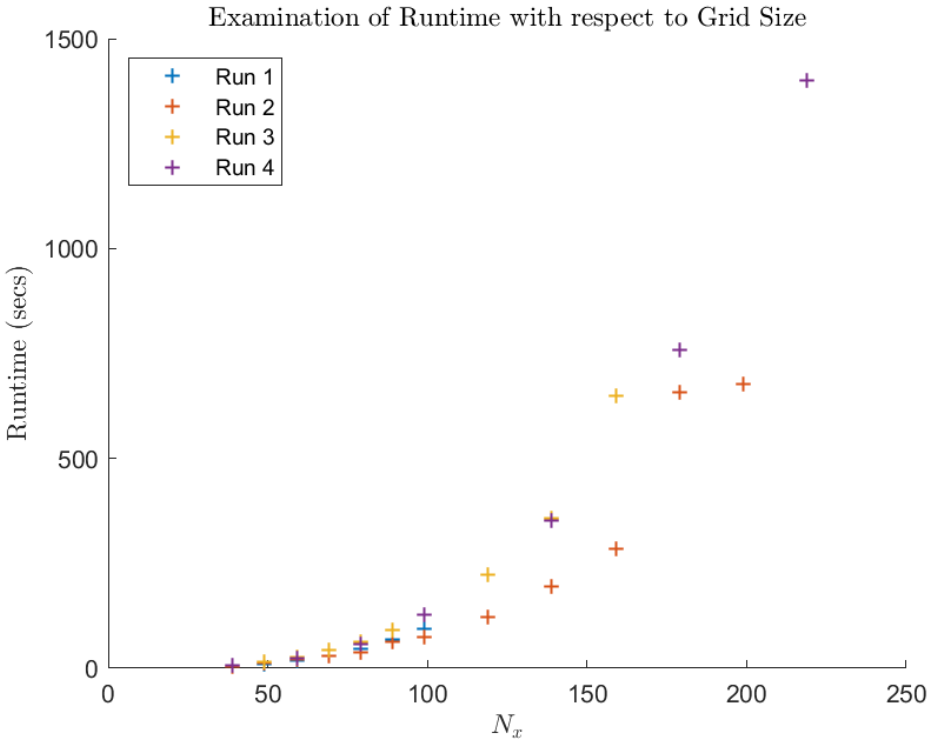
It was briefly considered to investigate whether there is a worthwhile trade-off between compile time and code run time. It was found that compilation was well less than a second for each compiler flag so certainly worthwhile to compile with the optimise flags. This is especially important given that the run time halves. These timings were taken on a symmetric grid with $N_x = 59$ and 1000 time steps.

We now consider the variation of the actual parameters of the program. First we consider the number of iterations k used to approximate the nonlinear potential. Examining the code, we would expect that it runs k times slower as it is just another nested loop. Profiling the results, we see the following:



This is a clear linear trend which corresponds with the nature of the additional iterative process as a for loop.

Next we turn our attention to the scaling of the grid. To simplify matters we consider a symmetric grid of equal spacing and position in all 3 dimensions. These tests were run with only 250 time steps in order to reduce the run time and make larger grids run in a reasonable amount of time in a single threaded setting. We see that the scaling is nonlinear, which is reasonable given that we are building the grid in 3 dimensions. However, the scaling does not appear to be as severe as $\mathcal{O}(N_x^3)$. It does however highlight that parallelisation would be of significant benefit for problems of non trivial size in order to alleviate the scaling issues. There unfortunately is non-trivial fluctuations in the curvature between timing runs, the different runs did not occur during a continuous time interval unfortunately.



References

[1] Sauer, Timothy, "Numerical Analysis", Pearson Education, 2012.
[2] Press, William H., Teukolsky, Saul A., Vetterling, William T., Flannery, Brian P., Numerical Recipes The Art of Scientific Computing, Cambridge University Press, 2007, 3rd Edition.
[3] University of California, San Diego, PHYSICS 130A QUANTUM PHYSICS, "The 1D Harmonic Oscillator", 23/4/13, Accessed 14/08/19 `https://quantummechanics.ucsd.edu/ph130a/130_notes/node153.html`.

Chapter 2

Parallel Implementation

2.0.1 Addendum to Chapter 1

As a brief note, the serial implementation is largely the same as that presented in the first milestone. The main differences are that the iterative scheme employed was updated to that discussed in the second code pseudo-code snippet in Section 1.2.5, which at the time was reasoned to be the most effective. Additionally, the input arguments and memory allocation were updated to allow the specification of non spatially homogeneous grids. Otherwise, only refactors and clarifications have been made, no algorithmic changes are present in the serial implementation.

2.1 Parallel Implementation Description

As a brief outline how this section proceeds, it is first discussed how OpenMP is well suited to the problem, though the need to use MPI in addition for scalability complicates this. Then the MPI implementation is explained before proceeding to discussion of the hybrid approach. The MPI parallelisation dictates two logical paradigms as to how OpenMP should be used to speed up processing on each rank.

It is immediately clear that the Schrödinger problem is well suited to parallelisation as within each time period and directional evolution, independent one dimensional evolutions of the wave function occur for every grid point in the space orthogonal to the 1D solve direction. The nested for loop structure of this means that OpenMP can easily be applied effectively as once each thread is allocated memory to compute 1D solves, the outer spatial loop can be directly parallelised without any race conditions.

The difficulty in parallelising this problem however, is in devising a means to subdivide the problem such that it can be solved on separate nodes in communication with MPI. As can be seen in Equation 1.9 we have a coupled system of equations wherein the determination of $\boldsymbol{\psi}^{n+1}$ depends on the entire previous time step vector $\boldsymbol{\psi}^{n}$. Whilst the explicit side matrix can be easily decomposed into sub columns, where the overlapping entries are communicated between nodes, the matrix inversion poses a problem, as inverting the matrix requires the complete, non-partitioned system. As such, to directly compute the inverse, the subdivided explicit solution and implicit matrix diagonals would need to be broadcast to a single rank, the inverse computed locally and the result broadcast back. This would mean a significant proportion of the computational expense would occur in serial, and would also incur non-trivial communication costs. Additionally the whole problem must then fit in memory on a single rank, although this is unlikely to be the limiting factor. To overcome this, an iterative procedure to determine the matrix inverse is adopted, using the previous time solution. This allows calculation to predominantly occur on each rank separately.

2.1.1 MPI Iterative Matrix Inversion

To explain this produce, a simple example is discussed, using a 4x4 grid split across 2 ranks. Obviously this is far too coarse to produce meaningful physical results, however it suffices for explanation. The matrix system below is identical to Equation 1.9, where we have omitted the boundary condition vectors as they are zero, and the right hand side has been computed and denoted as $\boldsymbol{\psi}^r$. For ease of notation we let $\gamma_i := 1 + 2\alpha + \beta\widehat{V}_i^{n+1}$ to denote the main diagonal terms of the implicit matrix:

$$\left[\begin{array}{cc|cc} \gamma_1 & -\alpha & 0 & 0 \\ -\alpha & \gamma_2 & -\alpha & 0 \\ 0 & -\alpha & \gamma_3 & -\alpha \\ 0 & 0 & -\alpha & \gamma_4 \end{array}\right] \begin{bmatrix} \psi_1^{n+1} \\ \psi_2^{n+1} \\ \psi_3^{n+1} \\ \psi_4^{n+1} \end{bmatrix} = \begin{bmatrix} \psi_1^r \\ \psi_2^r \\ \psi_3^r \\ \psi_4^r \end{bmatrix}$$

$$\iff \left[\begin{array}{cc|cc} \gamma_1 & -\alpha & 0 & 0 \\ -\alpha & \gamma_2 & 0 & 0 \\ 0 & 0 & \gamma_3 & -\alpha \\ 0 & 0 & -\alpha & \gamma_4 \end{array}\right] \begin{bmatrix} \psi_1^{n+1} \\ \psi_2^{n+1} \\ \psi_3^{n+1} \\ \psi_4^{n+1} \end{bmatrix} = \begin{bmatrix} \psi_1^r \\ \psi_2^r \\ \psi_3^r \\ \psi_4^r \end{bmatrix} + \alpha \begin{bmatrix} 0 \\ \psi_3^{n+1} \\ \psi_2^{n+1} \\ 0 \end{bmatrix}$$

Since the top two and bottom two equations have no dependence on one another, we can re-write this as a coupled system:

$$\begin{cases} \begin{bmatrix} \gamma_1 & -\alpha \\ -\alpha & \gamma_2 \end{bmatrix} \begin{bmatrix} \psi_1^{n+1} \\ \psi_2^{n+1} \end{bmatrix} = \begin{bmatrix} \psi_1^r \\ \psi_2^r \end{bmatrix} + \alpha \begin{bmatrix} 0 \\ \psi_3^{n+1} \end{bmatrix} \\ \begin{bmatrix} \gamma_3 & -\alpha \\ -\alpha & \gamma_4 \end{bmatrix} \begin{bmatrix} \psi_3^{n+1} \\ \psi_4^{n+1} \end{bmatrix} = \begin{bmatrix} \psi_3^r \\ \psi_4^r \end{bmatrix} + \alpha \begin{bmatrix} \psi_2^{n+1} \\ 0 \end{bmatrix} \end{cases}$$

Noting that $\psi_i^{n+1} \approx \psi_i^n$ (the previous time step wave function, not the explicit side solution ψ_i^r) for $\Delta t << 1$, we have

$$\begin{cases} \begin{bmatrix} \gamma_1 & -\alpha \\ -\alpha & \gamma_2 \end{bmatrix} \begin{bmatrix} \psi_1^{n+1} \\ \psi_2^{n+1} \end{bmatrix} \approx \overbrace{\begin{bmatrix} \psi_1^r \\ \psi_2^r \end{bmatrix}}^{(A)} + \alpha \overbrace{\begin{bmatrix} 0 \\ \psi_3^n \end{bmatrix}}^{(B)} \\ \begin{bmatrix} \gamma_3 & -\alpha \\ -\alpha & \gamma_4 \end{bmatrix} \begin{bmatrix} \psi_3^{n+1} \\ \psi_4^{n+1} \end{bmatrix} \qquad \underbrace{\begin{bmatrix} \psi_3^r \\ \psi_4^r \end{bmatrix}}_{\text{iter. static}} \qquad \underbrace{\begin{bmatrix} \psi_2^n \\ 0 \end{bmatrix}}_{\text{iter. dep.}} \end{cases}$$

This structure can be interpreted as iteratively providing a new set of "boundary conditions" on each rank based on the neighbouring ranks, which then form the modified right hand side expression. The split matrix system is something that can be solved locally on each rank, provided that the appropriate values the term (B) are communicated to and received from the neighbouring ranks. Note that for local matrices with more than two rows, there will be internal columns which need not be shared. We can improve the quality of this approximation by iteratively solving for a second new time period solution $(\boldsymbol{\psi}^{n+1})'$, where we replace (B) with $\boldsymbol{\psi}^{n+1}$ terms used instead of $\boldsymbol{\psi}^n$. Additionally, the main diagonal entries γ_i are updated as they contain a dependence on future potential $\widehat{V}_i^{n+1}$, which we were already iteratively approximating in this fashion in the serial code. Due to this structure, we can perform both of these iterative steps in conjunction with one another.

In order to implement this splitting of the problem, there are asynchronous MPI send and receives in order to exchange the entry in the outer columns for each matrix. Similarly, send and receives are performed in order to compute the explicit side solution $\boldsymbol{\psi}_r^n$, where the first and last entry in this vector for each rank depends on $\boldsymbol{\psi}^n$ from the rank prior and after respectively.

This is shown diagrammatically for the specific 3D generalisation taken in Figure 2.1. The barrier wait to receive the asynchronous results can be delayed until after the main diagonal is updated, based upon $\boldsymbol{\psi}^{(n-1)}$ which does not depend on the results of communication at the current iteration.

Inversion Scheme in 3 Dimensions

Given that this procedure applies to a 1D problem, consideration must be given as to how this procedure will be generalised to 3D. There are three logical schemes for how this could be done, corresponding to the number of dimensions MPI communication occurs in. In theory, it would be best for the scaling of communication costs to use MPI to subdivide the space in all spatial dimensions, creating local "cubes" for each rank. This means that increasing the number of ranks used will decrease side lengths of the cubes and thereby the communications costs for each rank. However it was opted to only split the domain in one dimension using MPI. This scheme will result in smaller individual slices when more ranks are used, but the communication cost for each rank is the same regardless of the number of ranks. This was opted for partially for ease of implementation, as whilst the communication process is the same, there is considerable room for error in arranging and determining the correct rank neighbours in 3 dimensions. This would also limit the number of usable rank configurations, which is important in practice and for scalability testing. The main reason however, which is discussed in Section 2.3.1 is the difficulty of using OpenMP within the local MPI partition in an efficient, non-trivial manner, given that the inversion scheme requires linked sub-problems to be solved in synchronisation.

The approach of partitioning the domain in one dimension only gives rise to the following communication schema. For each (y, z) grid position, we compute a 1D solve in x, where each rank works on the local partition of the grid. For each iteration (and the initial computation of the explicit side solution), the boundary points for each rank are communicated to the neighbouring ranks, to allow computation of the updated deformation of $\boldsymbol{\psi}^r$. The exception to this are the edge ranks where no such communication occurs on the exterior sides. After all convergence iterations are complete, the next (y, z) point is solved for until all x updates are complete.

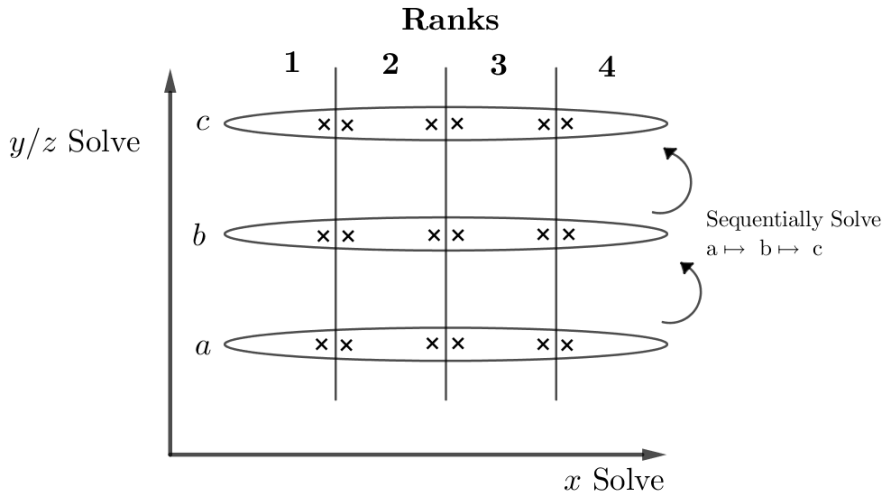


Figure 2.1: Depiction of the MPI communication schema utilised. Each ellipse is representative of a single one dimensional solve in the x direction, where the crosses indicate that boundary points on each rank, which must be communicated to their correspondent neighbour to complete these solves. Thus the progression from a to b reflects an update of the corresponding (y, z) grid point, where another 1D solve must be performed.

Inversion Scheme Convergence

In order to ensure the correctness of the iterative matrix inversion algorithm, the convergence of the algorithm was briefly investigated. For this context, the main diagonal was considered without a future dependence and the initial values for the right hand side α vector were taken to be zero, as there is no convenient way to generate an initial guess to a problem in isolation. Brief numerical experiments seemed to indicate convergence of this algorithm for tridiagonal systems was related to the diagonal dominance of the matrix; the magnitude of the diagonal element being greater than the sum of the magnitude of other elements in the row. This did not appear to be a strict condition as matrices marginally violating this property would converge to the true solution after a sufficiently large (often excessive) number of iterations. Fortunately, the implicit matrix for this problem is always diagonally dominant as $|\gamma_i| = |1 + 2\alpha + \beta\widehat{V}_i^{n+1}| > |\alpha|$ if β is small. Since $\beta \propto \Delta t$ will always meet this this in order to produce reasonable physical results. It was found that given we have a good starting guess for the right hand side terms from the previous time period, less than 5 iterations were sufficient to converge up to double precision with the serial results.

2.1.2 Additional MPI Uses

As a result of the above splitting of the problem, reduction operations are required to compute the output values at each time period as they depend on the entire problem space. Also, reduction is required in a similar fashion to determine the numerical normalisation rescaling that is done at the first time step in order to keep the wave function unitary. Finally, MPI broadcasts are used to read the problem input on one rank and send the parameters to all others.

2.2 Hybrid Approaches

To begin, the trivial uses of OpenMP are discussed. To ensure that the wave function is unitary when initialised, a local norm is computed on each rank then MPI reduced to determine the global norm. Similar reduction occurs to compute the output quantities for each time period. Both of these processes consistent of independent operation in 3 nested for loops (one for each dimension) so OpenMP reduction can be easily applied to speed up the computation of local values to be used in the MPI reduction. OpenMP can also be used for the initialisation of the arrays used for local 1D solves although, the communication cost would only be worthwhile if the number of threads is particularly high. Given that this is determined when a job is allocated and not compile time, it was decided not to use OpenMP here.

2.2.1 Potential OpenMP Communication Options

There are two obvious ways that OpenMP could be applied; using threads to speed up the computation of a single 1D solve, or to perform a 1D solve on each thread concurrently. This corresponds to two perspectives in Figure 2.1 (ignoring for the moment that this includes MPI rank communication as well), the first is using OpenMP to speed up the solving the 1D sub-problems a, then b and so on. The second is to start the solving of a, b and c simultaneously on each OpenMP thread (generalising to the number of threads available), where the serial sub-problems are computed simultaneously.

Since there is a matrix inversion procedure in the 1D code, the latter approach would be preferable, as we would get a parallel speed up for the entirety of the sub-problem. The first approach would still receive a significant proportion of this speed up, as there are several parallel loops within, however threads would sit idle whilst the master handles the matrix inversion. In light of this, the second approach is used for the y and z directions, whilst we discuss the x direction in Section 2.2.3 . Another advantage of this approach is that there is less overhead in communication as the OpenMP parallel sections can occur in a single contiguous block. It should be mentioned that there will be some memory overhead involved in allocating memory for each thread to use privately for the independent simultaneous 1D solves. However, the necessity to store the previous time period 3D lattice will be problematic well before this additional memory usage is an issue.

2.2.2 OpenMP with no Rank Communication - y and z directions

Firstly, each of the 1D arrays used for the 1D problems is is allocated to be `nthreads` times the length of those in the serial code. This means each thread can access independent memory within each parallel section by using the thread number to offset the starting index in this collective array. This is done as opposed to locally allocating on each thread, as the parallel region is nested within the time loop, so the system would need to `malloc` and `free` multiple arrays on every iteration.

Once this is done, these two directional solves can be placed in a single parallel region, using the `#pragma omp for` construct to distribute the work, where separate indexing of memory is achieved using the `thread_offset` variable. As a clarifying point to the code, the pointer to the start of each thread's memory allocation is obtained as `&z_psi_old_local_1d[thread_offset]`, for the respective arrays. This was deemed marginally clearer than pointer arithmetic, indicating that the block of memory begins at the address indexed by the offset. It should be noted that these parallel solves call functions which additionally contain OpenMP directives, but these these are not used in this context. This is since this schema uses all threads to solve a sub-problem each, so there will be no spare threads to split up the internal functions which

have OpenMP directives. These calls are instead used by the x direction evolution which has to deal with the sequencing of MPI communication.

2.2.3 OpenMP with Rank Communication - x direction

The general outline for computing a 1D solution, where the direction being solved in is split across ranks is as follows:

```
1  init_psi_1d(ψ_old_3d) ↦ ψ_old_1
2  PARALLEL_MPI_exchange_boundaries(rank, ψ_old_1,...)
3  one_dim_evolution(rank, ψ_old_1,...) ↦ ψ_new_x
4    [ %contents of one_dim_evolution
5    ⟶ build_rhs_explicit(...)
6    ⟶ for k=1:n_iter
7           PARALLEL_MPI_exchange_boundaries(rank, ψ_old_1,...)
8           modify_rhs_explicit(...)% update with comm. results
9           build_new_main_diag(...)% update from past iter
10          solve_for_new_implicit_soln(...)
11   ⟶ end  ]
```

The difficulty in parallelising such a scheme is due to the MPI boundary exchange within the loop for the iterated convergence of the nonlinear solution and the matrix inversion. This exchange must be synchronised across ranks before the matrix inversion which poses a problem when introducing threading. Section 2.3.1 discusses the potential for a solution using of MPI's support for funnelled threading. Here we proceed with an alternative working implementation, where `one_dim_evolution` is called in a serial context and OpenMP is used to parallelise the component subroutines.

The construction of the right hand side explicit solutions $\boldsymbol{\psi}^r$ and the construction of the iteratively updated main diagonal can be directly parallelised using `#pragma_omp_for` as they only depend on the previous time period/iteration. This leaves the matrix inversion being performed in serial. This however is not unreasonable given that MPI communication is used to compute this inverse, as each rank only holds a local partition of the problem grid. In discussion with Michael, parallel matrix inversion algorithms don't seem to be appropriate given that these are only local inverses, which are then being iteratively updated with MPI to compute the global inverse. Furthermore, these parallel algorithms generally require the solution to be passed onto a single rank at some point, which is otherwise avoided by this approach. This is potentially an important distinction when scaling to solve large problem instances.

2.3 Optimisations & Challenges

2.3.1 Using Funnelled MPI Thread Support for Rank Communication

An alternative approach for using OpenMP here is to attempt to mirror that used in y and z, where multiple sub-problems are solved concurrently. This seems relatively straightforward in theory, by wrapping the entire procedure snippet in a parallel region and performing MPI rank communication from an OpenMP master block. The unfortunate caveat to this approach is that it requires OpenMP barrier synchronisation after such a block to ensure that all threads receive the new boundary values, before attempting to compute the local matrix inverse. OpenMP does not support barriers within `#pragma omp for` regions, which complicates matters significantly.

The potential solution to this is to manually split the loop into distinct segments for each thread, to emulate the `omp for` splitting, thus allowing for barrier synchronisations inside. This makes indexing rather complicated as the loop bounds must be determined manually, and we must have at least as many grid points as threads in the outer loop direction for our operations to make sense. Additionally, we now must compute a thread offset in the innermost loop as was done in y and z evolutions. Subsequently, the independent 1D memory allocations could then be indexed without overlap. The MPI communication function must also be modified to accept an array of grid points to exchange, as each of the threads needs to swap their own, specific boundary values at each synchronisation.

The main difficulty in attempting this procedure is ensuring that the correct values are communicated at the right time to the right neighbours, and are received by the corresponding thread on the neighbouring rank. Finally, it is noted that this may not even result in a speed up due to the barrier synchronisations required. There is potential that the successive synchronisations for OpenMP and MPI could compound slow calculations on a particular thread into affecting the whole program. There is an additional potential gain however as this would mean the entire time loop could run in a single parallel region, which would remove the tear-down and resumption costs on each time iteration.

2.3.2 OpenMP Usage & Speedup

Preliminary testing would indicate that in general OpenMP is marginally beneficial at best to the runtime of the implementation. It is hoped that this is due to the relatively small problem sizes being tested at present, and that for larger problems that will be tested in Milestone 3, the potential speed up will outweigh communication costs. Consideration was given as to whether the sections being parallelised were large enough to warrant the communication overhead, however most use cases are nested loops over 3D segments, so the threading should have a sufficiently large impact for larger problems.

2.3.3 OpenMP Collapse

Given the fact that the program contains several closely nested loops, (or loops that can be converted to closely nested loops by moving calculation of `y_curr` inside the inner loop) the affect of using the collapse clause for OpenMP for loops was considered. This condenses the iteration space of the indicated loops into a single iteration space, allowing for more effective loop scheduling. Given that static scheduling is used as the workload per iteration within the loop should be constant, it would be expected that the speed up is marginal at best, although may help in problems with non-spatially uniform grids. Preliminary testing on a problem which took just over 10 minutes to solve with the pure hybrid approach saw marginal improvements. Given this, it was opted to use collapse on all such loops, noting at the very least it should not hurt runtime.

2.3.4 Loop Unrolling

In a similar vein to the above, the `-funroll_loop` compiler flag was investigated to see if this had any performance improvements. Whilst manual loop unrolling is also a possibility, given that a significant proportion of the looped code is already unwieldy, multi-line function calls and updates, it was decided to just test the compiler flag, to spare the considerable cost to code readability for a non-guaranteed performance gain. Preliminary tests also indicate a marginal performance improvement over not using the flag.

2.3.5 Broadcast Optimisations

A minor optimisation was to abstract the reading of inputs and the producing of output at each time periods into separate functions, so that the messages could be packed into a single array message without bloating code. For the inputs, the integer input types are cast to doubles so that a single array broadcast can be sent to all ranks, rather than individually sending each input. After the broadcast, the quantities in the array are assigned their individual identifiers on each rank, where the integer values are cast back to integer types. In a similar fashion, the reduction operation to compute the norm and variance at each time step can be amalgamated into a single message to avoid superfluous communication costs.

2.4 Verification of Parallel Implementation

The verification of the parallel implementation of the solution to the nonlinear Schrödinger equation is relatively straightforward as for the most part it is has been directly ported from the 3D serial implementation. So by direct comparison it is is clear to establish that the parallel code is at least as correct as the serial 3D code. Given that there are 1D cases with analytical solutions and that the tests conducted in Milestone 1 (Section 1.3) showed a direct correspondence between the 1D code and the 3D generalisation, this would indicate the parallel code is correct by extension. Whilst different initial conditions and grid resolutions can be specified, this problem does not have a set of edge cases as such. The one possible exception would be the matrix inversion failing, however it has already been established prior (in 2.1.1) that this will not happen for diagonally dominant matrices, which we have in all reasonable cases. Furthermore, the algorithm is completely procedural; once an initial condition is established, the same logical steps are followed in every problem instance. There are minimal conditionals, and these only affect local program flow. Nonetheless, we still examine a sample of the test cases presented to verify the serial implementation, and sanity check the physical interpretation.

Harmonic Oscillator Revisited

For the sake of clear comparison, we present the following results comparing the Milestone 1 serial implementation and Milestone 2 parallel implementation, whilst noting that minor modifications of the serial code occur in the interim between these two points. However, as the following results indicate, there is not actually any change in results for the same input parameters. The one exception to this is with respect to the parameter for iterations, where marginal differences are exhibited for small choice of k. This is however to be expected given that the iterations are implemented in a more effective fashion, requiring less total iterations to converge. This discrepancy however disappears once sufficiently many iterations are applied such that results converge. For comparisons we use a grid of 39 points in all directions on a domain from -20 to 20, 5 iterative steps and a symmetric width parameter of 1, this is the same as in the Milestone 1 tests (Section 1.3).

Both the serial and parallel implementations produce unitary norms regardless of input parameter choice and so will not be visualised. The dimensional standard deviations are time dependent as would be expected and are compared as follows:

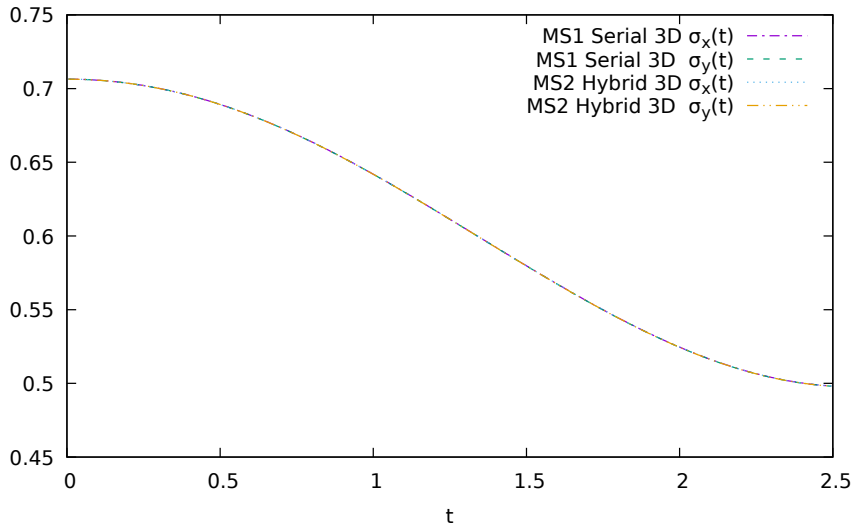


Figure 2.2: Gross Pitaevksi Equation with Simple Harmonic Oscillator Potential for $g = -1$.

As can be seen, all four lines are identical, which indicates that the procedures are equivalent. The fact that the quantity is the same in both x and y is unsurprising given the spherical symmetry of the potential. While the z direction also behaves similarly it has been suppressed to avoid further visual clutter.

Off Centre Initial Condition

Using otherwise the same initial conditions, the initial gaussian is offset to centre around (0.2,0,0) where we instead plot the horizontal displacement in each dimension. Similar to the above, we see that the parallel and serial code is indistinguishable. The behaviour here is physically sensible, the initial wave packet has nonzero potential as it is not centred with in the trap. The continuing oscillatory behaviour also seems reasonable given that there is no dispersion present in the system. The varying amplitude also makes sense given the nonlinear interaction.

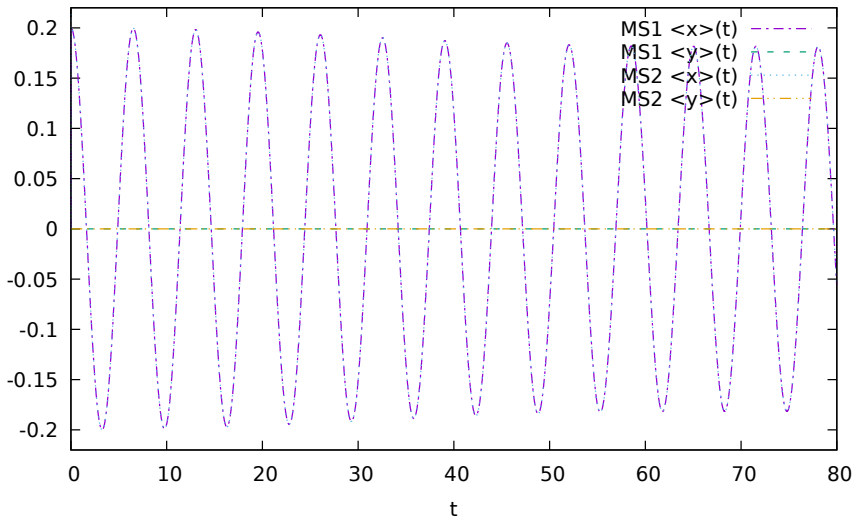


Figure 2.3: Gross Pitaevksi Equation with Simple Harmonic Oscillator Potential centred about (0.2,0,0) for $g = -1$.

Harmonic Trap in Two Dimensions Only

As a final test, the potential is left unconstrained in the x dimension by setting the component trap frequency ω_x to zero. With the lack of a confining potential, the expected behaviour is that the initial wave packet will disperse freely in the x direction. This is reflected in the code output - the dispersion is clearly exhibited on the time scale plotted as $\sigma_(x)(t)$ grows monotonically:

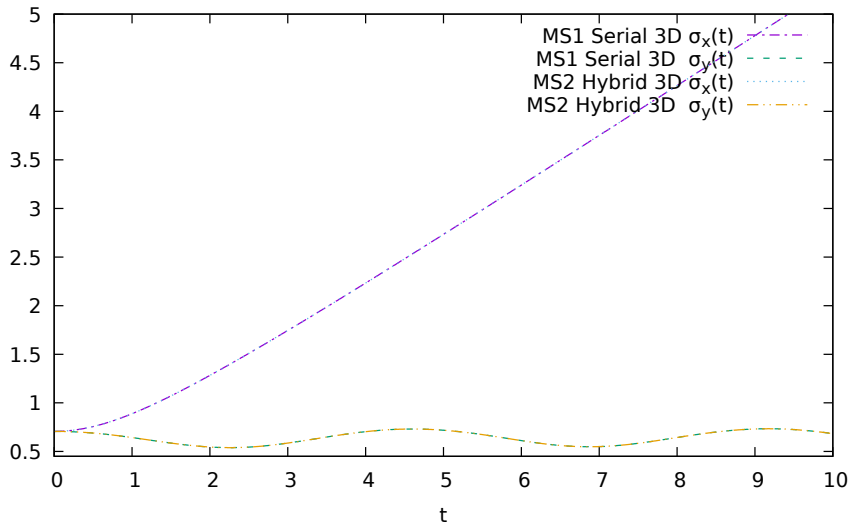


Figure 2.4: Gross Pitaevski Equation with Harmonic trap confinement in y and z directions, x unrestricted for $g = -1$.

2.5 Test Plan for Milestone 3

To test the overall performance of the parallelisation, the paradigms of strong and weak scalability are examined. We begin considering the former as it is clearer to see the performance scaling, however note that the latter is of more practical importance to this particular problem. Solution to PDEs are a problem in scientific computing where the grid domains and lattice size are largely determined by the physical application. The resolution however can be increased for more accurate results if more computing resources are available.

Although there are multiple input parameters which represent components of scalability. It was decided to focus on the grid resolution. This is since the time range for the desired solution will likely be constrained by the application, as will the accuracy required with the number of iterations. These parameters are kept standard at 500 time steps with $\Delta t = 0.01$ resulting in a 5 second time period being solved for. The parameter g is set to -1 and the gaussian width and directional frequencies are both set 1 in every direction. Unless otherwise specified, the fixed grid endpoints of $-20 \leq x \leq 20$ are used in each spatial dimension.

2.5.1 Strong Scalability Tests

We consider the speedup scaling of the problem for a fixed size instance where varying hardware resources are allocated to the program. Consideration was given as to what problem size is deemed most appropriate as a baseline for these scalability tests. It was decided to start with a problem of size $[60 \times 60 \times 60]$ grid points, which runs in about 90 seconds on the getafix cluster in serial. This is intended to be trivial enough to quickly test that job run scripts are working as intended and results are as expected. The intent is to then repeat tests for problems of size

90^3 and 140^3 which run in serial in about 5 and 20 minutes respectively. The basis for this is that this is hopefully a large enough problem that the parallelisation work is non-trivial, and the communication costs do not render any speedup obsolete. Additionally, keeping the runtime relatively short means that multiple MPI/OpenMP configurations can be tested relatively quickly. It is hoped that this might inform which configurations are most impactful and should be the focus of the rest of testing. Provided that these tests indicate that the parallelisation actually is producing a speedup, a test on a problem of size 180^3 would be conducted which ran in 50 minutes on the cluster. For each of these size configurations the following array of tests would be conducted. The priority listing is indicative of the order tests intended to be carried out in, so that if not all are completed, partial results are presentable. Note that this is of more importance for larger problem sizes. The variations in MPI would be performed first as preliminary testing seems to indicate this is more impactful than OpenMP, which makes sense given the use of parallelisation.

Priority		1	3	2
	MPI - Nodes $\times$ **Tasks**	**OpenMP # Threads**		
		1	4	8
1	$1N \times 1T$			
3	$2N \times 1T$			
5	$1N \times 2T$			
6	$2N \times 2T$			
2	$4N \times 1T$			
4	$1N \times 4T$			

Table 2.1: Proposed Strong Scalability Test Set

2.5.2 Weak Scalability Tests

For weak scalability, we are testing how the problem scales for fixed workload per processor. On this basis, it is proposed to test primarily with 2 OpenMP threads, unless there is another stand-out choice for this revealed by the strong scalability tests. It is reasoned that since this is a test of the scaling per processor, the number of threads of executions should be of secondary importance. With time permitting, the test suite would be repeated with 4 threads to hopefully confirm the scaling properties are similar. For this particular problem, we are in the fortunate position that problem sizes is well defined and easy controlled. The assumption for weak scaling is that the runtime T can be expressed as

$$T = A + B \cdot N,$$

that is the serial portion of code takes a fixed time to execute A regardless of size and the parallel portion scales linearly with problem size. The obvious question for the Schrödinger problem is how is the problem size N characterised, as the side length n or the volume n^3. It is proposed that the latter is most appropriate, although it is hoped that testing will confirm or reject this. The second consideration for strong scalability will be the limitation on the number of physical nodes available. On getafix, there are 6 nodes available on the cosc partition with 14 cores each, which limit to 2 cpus per task. Conflicting memory requirements or cluster availability will likely prevent doing tests of any reasonable size on anywhere near to this maximum number. To determine the problem size N_0, we work back from the fact that a grid of size 180 took ~ 50 minutes to run in serial. Noting 160 is the nearest factor of two which we can cleanly double or halve for dimension we take the following test set.

Size	**Set 1**	**Set 2**	**Set 3**
$80 \times 40 \times 40$	$1N \times 1T$		
$80 \times 80 \times 40$	$1N \times 2T$	$2N \times 1T$	
$80 \times 80 \times 80$	$1N \times 4T$	$2N \times 2T$	$4N \times 1T$
$160 \times 80 \times 80$	$1N \times 8T$	$2N \times 4T$	$4N \times 2T$
$160 \times 160 \times 80$	$2N \times 8T$	$2N \times 8T$	$4N \times 4T$
$160 \times 160 \times 160$	$4N \times 8T$	$4N \times 8T$	$4N \times 8T$

Table 2.2: Proposed Weak Scalability Test Set

The intent is that these tests will be performed sequentially for each set, noting that it would be reasonable to expect some non-trivial performance decline when transitioning from all on 1 node with local tasks to using multiple nodes. In the event that the parallel code is significantly faster than the serial code, these would be repeated with an accordingly scaled base problem size N_0. If this is not found to be the case, then a test of similar order of magnitude would be conducted to give a point of comparison regardless.

Chapter 3

Timing Results

3.1 Introductory Remarks

All timing results were obtained with the getafix cluster using the OpenMPI specification. The majority of all planned tests were completed, along with some additional inclusions. The main reason for the discrepancy here is that I was originally conservative in the test cases presented. Access to cluster resources was far less competitive than anticipated. I wrote python code to automate the production of slurm scripts for varying resource configurations and problem types which has meant I could quickly check whether the cluster was in use and submit jobs when it was available. Additionally the hardware configuration scoping was increased as I originally had noted that the Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz processors present had 14 cores, but decided it was worth utilising the fact they are capable of supporting 28 threads.

Another modification made was to consider more hardware configurations after the initial results actually showed reduction in runtime in parallel. This was facilitated by the fact that there were initially issues in getting OpenMP threads to behave in a non serial fashion in MPI programs. This contributed to the test sets in Milestone 2 being conservative, however solving this problem meant they could be freely extended.

Finally, it was decided to relax the decision of using a standard 500 time steps of step size $\Delta t = 0.01$ to solve for a 5 second time period. As was noted in the test plan, these parameters will likely be nearly completely constrained by the physical scenario being represented. It is also worth acknowledging that the time evolution is completely independent of the communication scheme, the same procedure is conducted in every single time step without variation. As such, it was anticipated that the program should scale linearly with the number of time steps, aside from initial input reading and memory allocation. This theorised relationship is investigated briefly in Section 3.2.4. It was decided to relax the imposed fixed time step constraint, allowing for the testing of larger grids, without compute times prohibitively long given the confines of deadlines.

3.2 Strong Scaling

For strong scaling, problem instances of grid sizes $90^3, 140^3$ and 400^3 were solved over various hardware configurations with the standardised grid domain and parameters detailed in Section 2.5. These configurations are consistent with the proposed tests, however a 60^3 toy problem instance was also proposed for debugging. In practice, this was revised to a trivial 10^3 instance for initial testing, and a full configuration test was not done. Given that the problem is trivially solved in serial, the communication costs are disproportionately large at this scale. The first two test grid sizes of 90^3 and 140^3 were solved over the original 500 step time frame, so the previously noted serial runtime of 5 and 20 minutes respectively are consistent. However the

400^3 instance only solved for 10 time steps evolutions and took 10 minute to solve in serial. To allow these to be directly compared, the runtimes in seconds are not presented, but instead speedup, or the ratio of the runtime with respect to the serial runtime are used. Note that the former is equivalent to the reciprocal of speedup, or "slowdown" and is used where parallel code is slower than the serial code. This avoids the confusing case of interpreting speedup < 1, as a proportion has clear meaning, regardless of whether it is fractional.

3.2.1 Smaller Grid Results

MPI		**OMP Threads**	
Nodes	**Tasks/ Node**	**1**	**4**
1	1	1.0	0.49
1	2	0.54	0.32
1	4	0.33	0.72
1	6	0.29	0.42
2	1	2.55	2.05
2	2	2.34	2.54
2	3	1.73	1.69
3	2	2.79	2.46
4	1	2.85	2.82
6	1	2.83	2.73

Table 3.1: Runtime as a proportion of serial runtime for a grid of 90^3. The serial configuration runtime was 410 seconds.

This test will not be analysed in any great detail, as it is relatively clear that the problem instance is far too small to benefit from multi-node MPI configurations. We can however notice that generally OpenMP threading does improve performance relative to the single threaded counterpart test. Additionally, it is interesting to note that the best results occur using MPI tasks without OpenMP threads, although the lack of thread configurations does not provide much insight here. This will be re-examined for larger problem instances where more thorough scaling tests are done.

MPI		**OMP Threads**				
Nodes	**Tasks/ Node**	**1**	**2**	**4**	**6**	**8**
1	1	1.0	1.37	1.79	2.47	2.64
1	2	1.9	2.54	3.63	2.18	2.54
1	4	3.03	4.15	2.87	3.25	
1	6	4.63	0.99	4.03		
2	1	0.59	0.64	0.85	0.55	0.64
2	2	0.67	0.58	0.81	0.74	0.82
4	1	0.57	0.62	0.61	0.57	0.73
6	1	0.81		0.54		

Table 3.2: Speedup for a grid of 140^3 relative to the serial runtime of 25.5 minutes.

We see logical results for increasing the number of OpenMP threads, which decreases runtime generally. There are some exceptions, using 6 OMP threads performed worse than using 4 in some cases. Perhaps unsurprisingly, the best configurations occur utilising a single physical

node with multiple MPI tasks on that node. This allows the benefits of splitting the problems using MPI without introducing as significant of a communication cost. No configuration using multiple nodes performs better than the serial code. Given the relatively small instance size it seems feasible that the communication costs are more detrimental than the advantage of splitting the problem.

3.2.2 Large Grid Results

MPI		**OMP Threads**				
Nodes	**Tasks/ Node**	**1**	**2**	**4**	**6**	**8**
1	1	1.0	1.57	2.02	2.59	2.35
1	2	1.91	2.94	4.07	4.61	3.89
1	4	3.36	5.0	4.53	5.47	
1	6	4.37	3.68	6.08		
2	1	1.14	1.18	1.81	1.77	0.85
2	2	1.3	1.32	1.41	1.44	2.59
2	3	1.46	1.88	1.75		
2	4	1.51	1.29	1.23	1.76	
3	2	1.23	1.42	1.72		
4	1	1.29	1.28	1.35	1.39	1.55
4	4	1.41	1.39	1.45	1.58	
6	1	1.24	1.47	1.41		

Table 3.3: Problem speedup relative to serial performance for grid of dimensions 400^3. The problem set evolves through 10 time steps, and the baseline serial configuration ran in 600 seconds (10 minutes).

MPI		**OMP Threads**		
Nodes	**Tasks/ Node**	**1**	**2**	**4**
1	1	1.0	1.46	2.49
1	2	2.02	3.19	4.53
1	4	3.62	5.62	5.34
2	1	1.23	1.64	1.48
2	2	1.26	1.93	1.93
4	1	1.28	1.51	1.51

Table 3.4: Problem speedup relative to serial performance for grid of dimensions 400^3. The problem set evolves through 40 time steps, and the baseline serial configuration ran in 48 minutes. This reduced test set was included additionally to quantify the proposed linearity between runtime and time scale.

For the larger problem instance in Table 3.3, we see promising results, with all configurations excepting 2 MPI nodes with 8 OpenMP threads running faster than the serial implementation. This is in contrast to the 140^3 grid set, where only the single node results were faster than serial. The largest performance gain is still using a single node, with 4 OpenMP threads and 6 MPI tasks per node achieving a speedup factor of 6.08. This is still well short of the ideal speed up $S(N, p) = p$ given that we are using 24 times the resources of the serial code to achieve this.

Next, the speed up relative to the serial implementation is examined graphically for this large scale instance. The caveat is that as can clearly be seen from the raw data, there is

no semblance of equivalence between the single node and multi-node regimes. Consequently, it does not seem reasonable to refer to these homogeneously as "processors" on a single axis. Instead, slices of this space are considered.

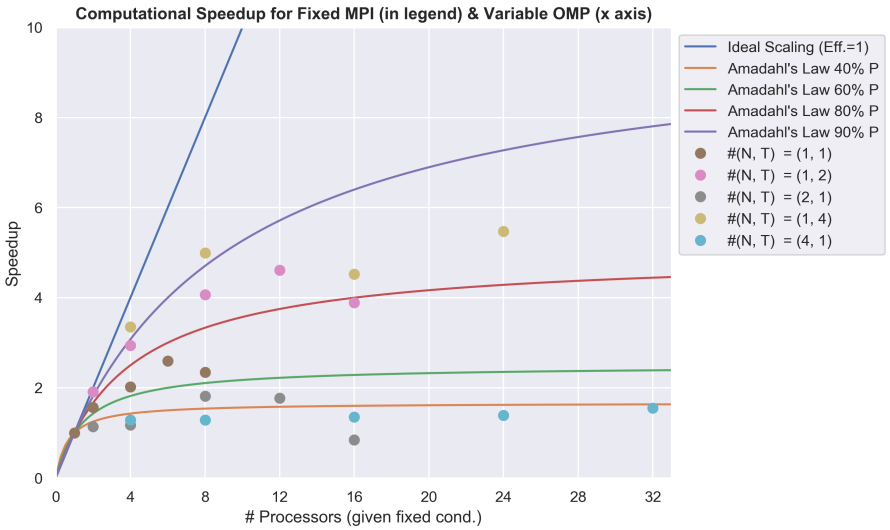


Figure 3.1: Computational Speedup for slices in the MPI node (N) and tasks per node (T) planes. Ideal scaling and various Amadahl's Law curves included for reference.

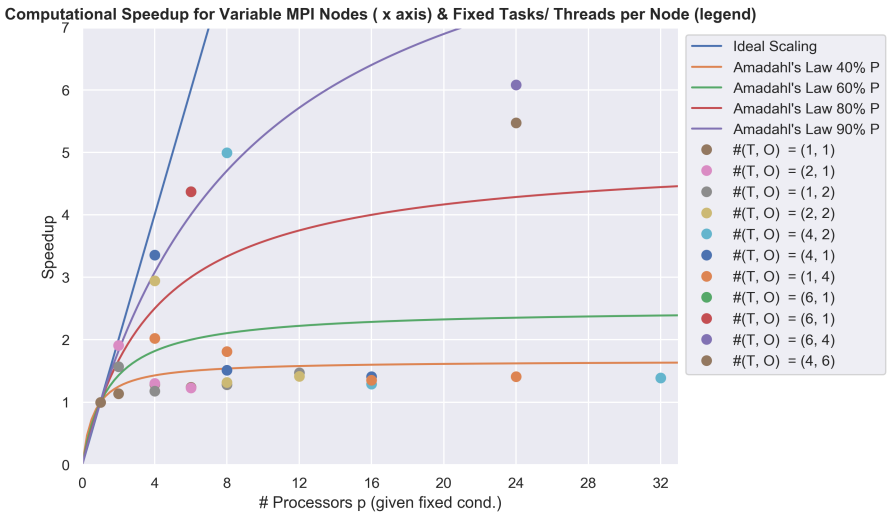


Figure 3.2: Computational Speedup for slices in the MPI tasks per node (T) and OMP threads (O) planes. Ideal scaling and various Amadahl's Law curves included for reference.

Examining the scaling slices it is evident that the program does not exhibit ideal strong scaling. This is perfectly reasonable, the code whilst heavily parallelisable inherently has a serial set up fraction of run time and significant communication costs. The results fall within the realm of reasonable given the expectation of Amadahl's law that there is an initial non-parallelisable fraction of code. Time scaling is actually important in this context as increasing the number of steps would reduce the serial fraction of execution time, which would be favourable in regards to Amadahl's law. The data points in Figure 3.2 where the number of MPI nodes is varied are more discrepant from this theorised relationship. As has already been discussed, this is likely attributable to the non-trivial costs that occur in transferring data between nodes, given the size of the problem.

3.2.3 Method Shortcomings in Relation to Strong Scaling

At this point it is worth reiterating the shortcomings of the parallelisation implemented. The use of domain slices rather than tiling results in increasing the number of MPI ranks leaving the communication cost for each rank constant. While the size of each rank's local grid is reduced, there is a point where increasing the number of ranks further only detriments performance. Given this, it is not unreasonable to expect negatively trending (or at least non-increasing) speed up when the number of nodes is increased.

3.2.4 Aside on Time scaling

As has been discussed, it was decided to focus testing on the problem scaling with respect to grid size, and fix all other parameters. Given fixed number of time steps was relaxed to allow the testing of larger instances, it is worthwhile briefly examining whether the theorised linear relationship between runtime and problem size is justified. Table 3.4 repeats a reduced selection of the 400^3 problem tests, solving over 40 timesteps, 4 times longer than the original test set. Generally there are no wild discrepancies, although the 2 MPI Node, 2 OpenMP thread tests seem to be notably quicker than in Table 3.3. There is an expectation that the runtime proportions in Table 3.4 should perhaps be slightly higher, as the serial initial fraction of code comprises a smaller percentage fraction of the total code execution time.

The most significant result however is that the runtime is roughly 5 times longer, despite the problem workload only growing 4-fold. It would be expected that the barrier synchronisations might amplify the affects of system fluctuations, but this does not explain the variability in the serial runtime. Aside from the initial serial component for set-up and memory allocation, the problem should be uniform for each time step, given that there is no branching conditional logic involved in the code. Given that this indicates the time scaling is not exactly linear, it is not ideal to have spatial scaling tests use different time scales. However, it does not appear to depart from this significantly, but only two sample scales does not resolve this confidently. This should be kept in mind when comparing results that use different time scales, although each individual set of tests is self consistent.

3.3 Weak Scaling

To examine weak scaling, the proposed tests starting from grid latices of $80 \times 40 \times 40$ and doubling the the volume and resources were fulfilled. Three different hardware configuration sets were tested, with each stemming from a different baseline hardware configuration. For all tests, the number of OpenMP threads was kept at a constant 1. This is perhaps a limitation of the tests, although it meant that all cores on a node could be used in MPI configurations. The scope of these tests was marginally extended, with the realisation that 28 was the ceiling on cores per node using hyper-threading.

After the additional inclusion of the 400^3 grid in the strong scaling tests, it was decided to expand the weak scaling tests scope. Subsequently, a test up to a maximum lattice size of 1024^3 was used, relaxing the 500 time iterations down to 10.

3.3.1 Exploring Hardware Configurations

Grid Size	Set 1		Set 2		Set 3	
80-40-40	1N-1T:	1.0				
80-80-40	1N-2T:	1.07	2N-1T:	5.08		
80-80-80	1N-4T:	1.5	2N-2T:	10.64	4N-1T:	12.48
160-80-80	1N-8T:	1.6	2N-4T:	8.72	4N-2T:	12.5
160-160-80	2N-8T:	23.5	2N-8T:	21.79	4N-4T:	22.51
160-160-160	4N-8T:	42.58	2N-16T:	39.47	4N-8T:	46.86
320-160-160	4N-16T:	47.42	4N-16T:	46.03	4N-16T:	45.04

Table 3.5: Runtime as a proportion of the base grid size test case. The 80-40-40 grid instance solved in 78 seconds over a period of 500 time steps.

Before discussing the graphical representation, there are some interesting points that can be observed directly. Firstly, for the smaller problem scales, the single node configurations are significantly faster. This is consistent with similar observation in the strong scaling tests.

It is seen as problem size increases this speed difference is less pronounced or non existent. This is likely related to the synchronisation in the MPI communication pattern. Each rank waits for messages sent and received from its neighbours before computing the local matrix inverse, and since these neighbours are circular, all ranks must at least be almost in sync. This means it is reasonable that if messages are still exchanged quickly on a node, the ranks may sit idle waiting for the multi-node communication to catch up. Another possible explanation is that as grid size size increase, and thereby the message size grows, cores nodes must use less than ideal memory locations to service each MPI rank. This could eliminate some of the benefits associated with intra-node communication.

Finally it is less than ideal to note the runtime variability between the final three tests in each column, which should have ideally been close to identical.

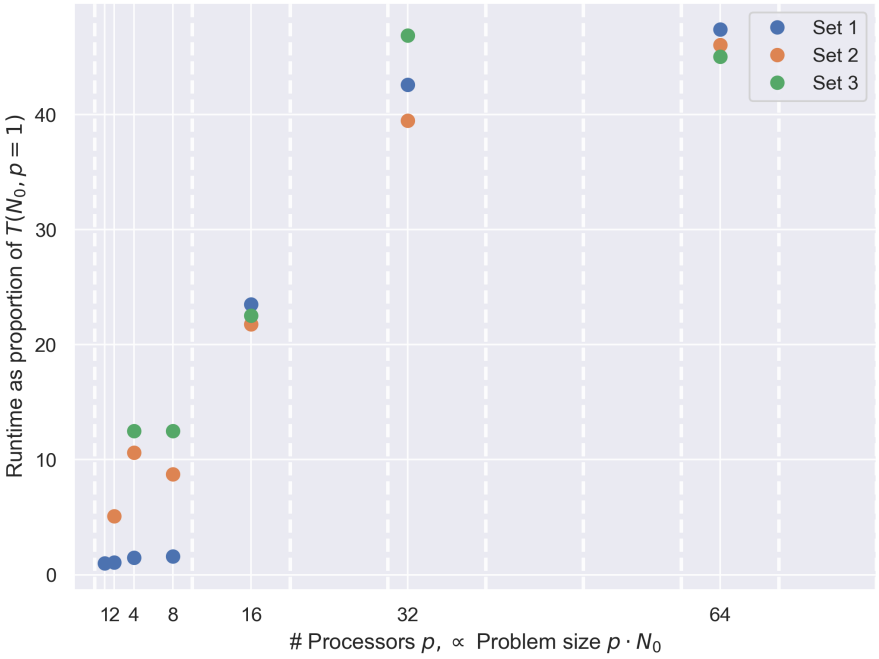


Figure 3.3: Plot of Runtime as a proportion of the serial runtime i.e. $\frac{T(N_0 \cdot p, p)}{T(N_0, p=1)}$ where $N_0 = 80 \times 40 \times 40$. Tests are conducted without using OpenMP threads so that the configurations can make use of the cores as MPI tasks on a node.

It is apparent that the performance falls well short of ideal weak scaling, as when more than one node is used there is a very significant performance decline, the runtime proportion does not remain constant at one. Given that the MPI message size is proportional to the x direction grid size (recalling MPI is using to slice the grid in the x domain), there are non trivial communication contributions and so is reasonable that this impact scaling. Another fact to note is that since the MPI splitting occurs in the x direction, doubling the dimension in y or z will inherently result in a restructure in resources and sub-problem size. Consider the following diagrams, where the z component of the grid has been suppressed for clarity:

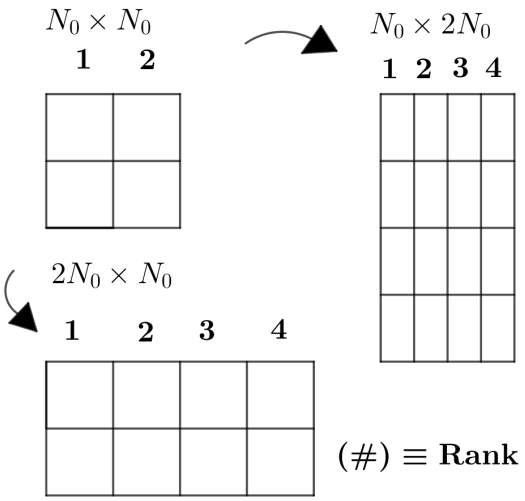


Figure 3.4: Sketch of problem subdivision depending on direction of expansion. Indicates that doubling in the x (first) direction results in the grid size per node persistent. Expanding in the y direction, where MPI splitting does not occur keeps the number of grid points per rank consistent, but not the grid dimensions.

It is seen that the direction of expansion does not behave symmetrically with respect to rank configuration. It is always preferable to have the largest problem dimension oriented in the x direction, and this has been implicit in the design of all tests. Increasing in the y and z direction, results in thinner slice sub-problems on each rank. In doubling the problem size, we necessarily must increase the problem size in both x and y, so it is reasonable that performance cost is non-uniform with respect to dimension.

It may have been more physically appropriate to choose square size lengths such that the volume doubles, as it seems unlikely that an application would call for finer grain spatial grid size in a single dimension (or would up-scale in an alternating fashion as in the tests). This however presents its own issues as the workload would not be fixed per processor, which is a key assumption for the weak scaling tests.

3.3.2 Larger Problem Instances

Given the expansion of the strong scaling tests to examine a problem of dimensions 400^3, it was deemed appropriate to also consider larger instances for the weak scaling tests. As was mentioned, these tests solve over a shorter number of time steps to run in a reasonable time period.

Grid Size	Set 4	
256-256-256	1N-1T:	1.0
512-256-256	1N-2T:	1.0
512-512-256	1N-4T:	1.07
512-512-512	1N-8T:	1.23
1024-512-512	1N-16T:	1.41
1024-1024-512	2N-16T:	5.7
1024-1024-1024	4N-16T:	14.29

Table 3.6: Runtime as a proportion of the base grid size test case; $256 \times 256 \times 256$. This instance solved for 10 time steps and took 3 minutes to solve.

For this test set with the larger baseline problem size we see a reasonable approximation to weak scaling for the smaller problem instances. This is unfortunately correspondent with the situation where the problem is solved in a single node regime and does not generalise beyond this area. As such, this is of limited importance when considering further up-scaling of the problem size.

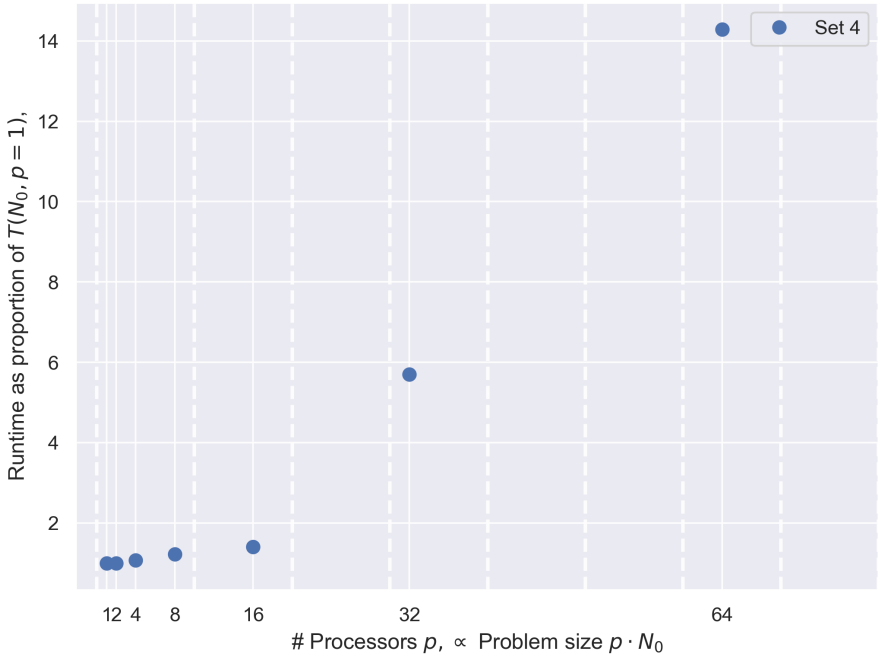


Figure 3.5: Plot of Runtime as a proportion of the serial runtime i.e. $\frac{T(N_0 \cdot p, p)}{T(N_0, p=1)}$ where $N_0 = 256 \times 256 \times 256$.

Figure 3.5 exhibits qualitatively similar performance to that exhibited in Figure 3.3 however it should be noted that the degradation in the runtime proportion is significantly lower, reaching a maximum of 14 times slower for the largest instance as opposed to the 45-fold slowdown. Whilst not immediately clear, this can be explained considering the consequences grid size has on parallel performance, which was illustrated Figure 3.4. Due to the use of domain slicing in the x direction, maximum grid size instance from Set 1-3 ($320 \times 160 \times 160$) is split amongst the $4 \times 16 = 64$ processors resulting in sub-problems of dimension $5 \times 160 \times 160$. This kind of asymmetric domain splitting is unfortunately inherent to the implementation, and could be avoided using a tiling scheme. Having only 5 points per rank means that 1D solves in the x direction are incredibly communication intensive, which does explain the blow-out in runtime. For the 1024^3 problem in Set 4, this issue is less severe, with 16 points in x per rank.

The sparsity of sample points limits further conclusions that can be drawn from this data. It should be noted that the apparent linear trend present in the plot is not significant, this is an artefact of the log scaling of the x axis, in line with the problem size and resources doubling.

Finally, it is emphasised that whilst these plots do not indicate ideal speedup, the solve times are still significantly faster in parallel than in serial. Although this is effectively a strong scaling test, it was attempted to solve the 1024^3 instance in serial, as a point of comparison to the 43 minute runtime in parallel. The memory allocation requirement for the job was in the range of $30 - 34$Gb and took 3 hours to solve. Although this is a feasible amount of memory for a single node to have access to on a cluster computer, it is clear that further upscaling dimensions in serial will eventually become a problem.

3.4 Conclusions on Scaling

From the timing experiments conducted, it is apparent that implemented parallel solution to the nonlinear Schrödinger problem does not show ideal weak or strong scaling. Despite this, the code does still scale, and speedups were noted across nearly all instances for appropriate hardware configurations. This speed however is not achieved in a particularly resource efficient manner. The largest speed up factor of 6.08 for the largest strong scaling test utilised 24 times the serial resources.

In general, it was determined that single node processing is preferable where appropriate. Once more than one node was required, the performance difference between MPI tasks on one node and using multiple nodes was less pronounced.

There was no distinct best configuration found in terms of core usage on a single node for MPI tasks or OpenMP threads. It seems generally, that these resources are better allocated to MPI tasks per node. Algorithmically this makes sense, as the MPI splitting into ranks allows calculation of a local partial matrix inverse on each rank, while this procedure is not parallelisable within the usage of OpenMP. This is in spite of the separate memory allocation for MPI which corresponds to larger communication overhead (relative to OMP).

One major advantage of the parallel implementation is the ability to distribute memory consumption. The parallelised code uses marginally more memory (communication exchanges, extra local 1D copies) overall, however this memory cost is spread across all MPI ranks. Consequently, this means the parallel implementation can solve problems too large for a serial approach to effectively handle. Whilst the currently unused portion of the grid could be written to disk and loaded, this would introduce considerable IO overhead. It is however anticipated that runtime scaling may become an issue before memory does. This is based on the fact that in terms of scaling for fixed problem size, the resource use to obtain speedup is quite inefficient. Additionally, the fact that performance is well short of ideal weak scaling would suggest that runtime will necessarily increase for larger problems with fixed amount of work per processor. The can be mitigated if the lattice spacing can afford to be more coarse, in order to reduce solve time.

3.5 General Conclusions

A hybrid parallel approach to solving the 3D non-linear Schrödinger equation was developed using finite difference methods. An approximation to decompose the the 3D dimensional solution into successive application of linear operators acting in single dimensions was used to reduce the computational expense of the problem. This allowed convenient use of both MPI and OpenMP for parallelisation, the former used to slice the domain in the x dimension, whilst

the latter was used to parallelise the solving in the remaining two (y and z), alongside speeding up the components of the MPI solve.

An attempt was made to modify the parallelisation to use funnelled MPI thread support such that OpenMP could be utilised in the x where the domain was split with MPI, to facilitate concurrent solves in that direction. Unfortunately, this approach was not able to be implemented successfully as the MPI thread communication required sequencing of barrier synchronisations on the OpenMP threads around the MPI communication. This means the implementation underutilises the OpenMP thread resources during the x direction evolution matrix inversions, where the MPI partitioning occurs.

The code was tested on a variety of problem instances to determine performance with respect to weak and strong scaling. The code does parallelise effectively, however less than ideal speedup was observed for strong scalability. Whilst runtime improvements were observed, the efficiency of resources to produce this was relatively low. For weak scaling, runtime when doubling the resources and problem size remained less than double the baseline problem size when scaling on a single node. Once multiple nodes were required, the proportionate runtime increased significantly. It was identified that this poor scaling behaviour is likely related to the use of domain slicing rather than tiling.

Forecasting towards massive problem instances, it was determined that runtime is likely to be a scaling issue before memory considerations. This is related to the scalability observed, but also that the memory consumption is constant with respect to problem time-scale. Only the current and previous time step are required to be held in memory, the previous states are discarded or written out to file. Practical applications will likely require significantly more (or larger) time steps than were tested here to produce physically meaningful results. As such, time scaling is likely to be the main limiting factor.