2D Parallelised FDTD code

Akash Dasgupta

H.H Wills Physics Laboratory, School of Physics, University of Bristol

ABSTRACT

We demonstrate a parallel 2D Finite Difference Time Domain (FDTD) code, implemented using the message passing interface (MPI) library. The code utilises the 'Perfectly matched layer' (PML) absorbing boundaries. The code was used to demonstrate0 electromagnetic phenomenon such as lensing, diffraction and evanescent wave tunnelling. We demonstrate the effect on simulation runtime of simulation size and number of cores available, as well as discuss various optimisations and load balancing across processes carried out.

Introduction

In the world of computational electrodynamics, there are several solvers that may be used depending on the nature of the problem being simulated. If the final spectrum of the simulation is of interest for example, a frequency domain solver is ideal. However, in order to model transient effects, a 'Finite difference time domain'(FDTD) method is necessary. FDTD is a 'grid type' solver. The space is segmented into a grid, with the electric and magnetic fields assigned to each point in space. For each time iteration, the magnetic field and the electric field are sequentially updated, in what has been described as a 'leapfrog' fashion. In this way one can model the time evolving dynamics of oscillations in the the electric and magnetic field. Physical objects are modelled on such a grid by changing the characteristics of the points the object encompasses. The method is able to produce effects such as evanescence and pulsed light sources in a very natural way, and can be used to create movies of the time evolution of electromagnetic waves.

This report outlines the development of a scalable 2D FDTD code. The scalability is intrinsically linked to code runtime. Faster code means that more simulations can be run in the same given time, and larger simulations which otherwise would take impractically long to run can be realised. Such speed-ups can be achieved by utilising 'parallel computing', taking advantage of the multiple computing cores available in modern computers and super computers to accelerate the execution of the code.

This code was developed in C++, with a visualisation script developed in Python. The Tests described in the report are run on the University of Bristol Super computer, 'BlueCrystal Phase 3', which is equipped with 16 core 2.6 GHz SandyBridge CPUs and a hybrid 10 gigabit ethernet and 56 gigabit InfiniBand communication structure. Jobs may be run across different CPU nodes to utilise their combined core count. The effectiveness of the scalability of the code using such a system is discussed in explored report.

1 FDTD Theory

Light in a wave interpretation can be understood as mutually sustained oscillations in the electric and magnetic fields. The dynamics of electromagnetic waves through some medium is given by Maxwell's equations:

$$\nabla \times \mathbf{E} = -\frac{\mu_0[\mu_r]}{c_0}\frac{\partial \mathbf{H}}{\partial t}, \quad \nabla \times \mathbf{H} = \frac{1}{c_0}\frac{\partial \mathbf{D}}{\partial t}, \; \mathbf{D} = \varepsilon_0[\varepsilon_r]\mathbf{E}, \tag{1}$$

where $\mathbf{E}$ and $\mathbf{H}$ are the general 3D electric and magnetic fields respectively, ε_0 and μ_0 the electric permittivity and the magnetic permeability of free space. $[\varepsilon_r]$ and $[\mu_r]$ are the relative permittivity and permeability tensors, and is a function of the medium at some point in space.

The form of Eq.1 can be simplified significantly for the 2D system we seek to model. Here, any $\partial/\partial z$ terms in the curl vanish. Further, we will limit ourself to modelling isotropic materials (which have diagonal permittivity

and permeability tensors). Under this schemes the equations simplify into 2 modes: A transverse electric (TE) and Transverse magnetic (TM) mode. The code developed only simulates the TE mode, but converting it to do TM is a matter of switching labels, the dynamics are identical. For the TE mode, the maxwell equations simplify to:

$$\begin{aligned} \frac{\partial \tilde{E}_z}{\partial y} &= -\frac{\mu_{xx}}{c_0}\frac{\partial H_x}{\partial t}, \qquad -\frac{\partial \tilde{E}_z}{\partial x} = -\frac{\mu_{xx}}{c_0}\frac{\partial H_y}{\partial t}, \\ \frac{\partial H_y}{\partial x} &- \frac{\partial H_x}{\partial y} = \frac{1}{c_0}\frac{\partial \tilde{D}_z}{\partial t}, \end{aligned} \tag{2}$$

where:

$$\tilde{E} = \sqrt{\frac{\varepsilon_0}{\mu_0}}E \quad \tilde{D} = \frac{1}{\sqrt{\mu_0\varepsilon_0}}D. \tag{3}$$

The normalised electric field $\tilde{E}$ is chosen for the sake of precision. The magnetic field is many orders of magnitude smaller than the electric, and so to avoid loss of precision in numerical operations involving both, the normalisation is applied.

1.1 Finite difference and the Yee Grid

FDTD is implemented by approximating Maxwell's equations through finite difference. Taking for example the curl H term we can write:

$$\nabla \times \vec{H}\big|_{t+\Delta t/2} = \varepsilon \frac{\vec{E}\big|_{t+\Delta t} - \vec{E}\big|_t}{\Delta t}. \tag{4}$$

Note that the calculated curl of H actually exists in time halfway between the the two electric field samples. Rearranging we get the recursive relation:

$$\vec{E}\big|_{t+\Delta t} = \vec{E}\big|_t + \frac{\Delta t}{\varepsilon}\left(\nabla \times \vec{H}\big|_{t+\Delta t/2}\right). \tag{5}$$

Importantly, in order to update the electric field we must use samples of the magnetic field half a time step ahead. To calculate the curl on a grid space by finite difference the same applies: in order for the curl to exist at the same point in space as the calculated field, we must sample half a grid step either side. This is facilitated by the Yee Grid[1], where each unit cell holds the electric field, and 2 components of the magnetic field placed at a half space between (Fig.1a). For a Yee grid, with matrix element indices i and j, the curls at a point are given by:

$$\begin{aligned} C_x^E\big|_t^{i,j} &= \frac{\tilde{E}_z\big|_t^{i+1,j} - \tilde{E}_z\big|_t^{i,j}}{\Delta y}, \quad C_y^E\big|_t^{i,j} = \frac{\tilde{E}_z\big|_t^{i,j+1} - \tilde{E}_z\big|_t^{i,j}}{\Delta x}, \\ C_z^H\big|_{t+\frac{\Delta t}{2}}^{i,j} &= \frac{\tilde{H}_y\big|_{t+\frac{\Delta t}{2}}^{i,j} - \tilde{H}_y\big|_{t+\frac{\Delta t}{2}}^{i,j-1}}{\Delta x} - \frac{\tilde{H}_x\big|_{t+\frac{\Delta t}{2}}^{i,j} - \tilde{H}_x\big|_{t+\frac{\Delta t}{2}}^{i-1,j}}{\Delta y}. \end{aligned} \tag{6}$$

using the literature notation in the form: $(\nabla \times \mathbf{E}) \cdot \hat{x} = C_x^E$

Using this, we can write Eq.2 in the finite difference form:

$$\begin{aligned} H_x\big|_{t+\frac{\Delta t}{2}}^{i,j} &= H_x\big|_{t-\frac{\Delta t}{2}}^{i,j} - \left(\frac{c_0\Delta t}{\mu_{xx}\big|^{i,j}}\right) C_x^E\big|_t^{i,j}, \qquad H_y\big|_{t+\frac{\Delta t}{2}}^{i,j} = H_y\big|_{t-\frac{\Delta t}{2}}^{i,j} + \left(\frac{c_0\Delta t}{\mu_{yy}\big|^{i,j}}\right) C_y^E\big|_t^{i,j}, \\ \tilde{D}_z\big|_{t+\Delta t}^{i,j} &= \tilde{D}_z\big|_t^{i,j} + (c_0\Delta t) C_z^H\big|_{t+\frac{\Delta t}{2}}^{i,j}, \qquad \tilde{E}_z\big|_{t+\Delta t}^{i,j} = \left(\frac{1}{\varepsilon_{xx}\big|^{i,j}}\right) \tilde{D}_z\big|_{t+\Delta t}^{i,j} \vec{D}(\omega) = [\varepsilon_r]\vec{\tilde{E}}(\omega) \end{aligned} \tag{7}$$

These form the 'update equations', and for a simulation space. Iterating over i and j for each time step and applying these produces the time evolution of the simulated system.

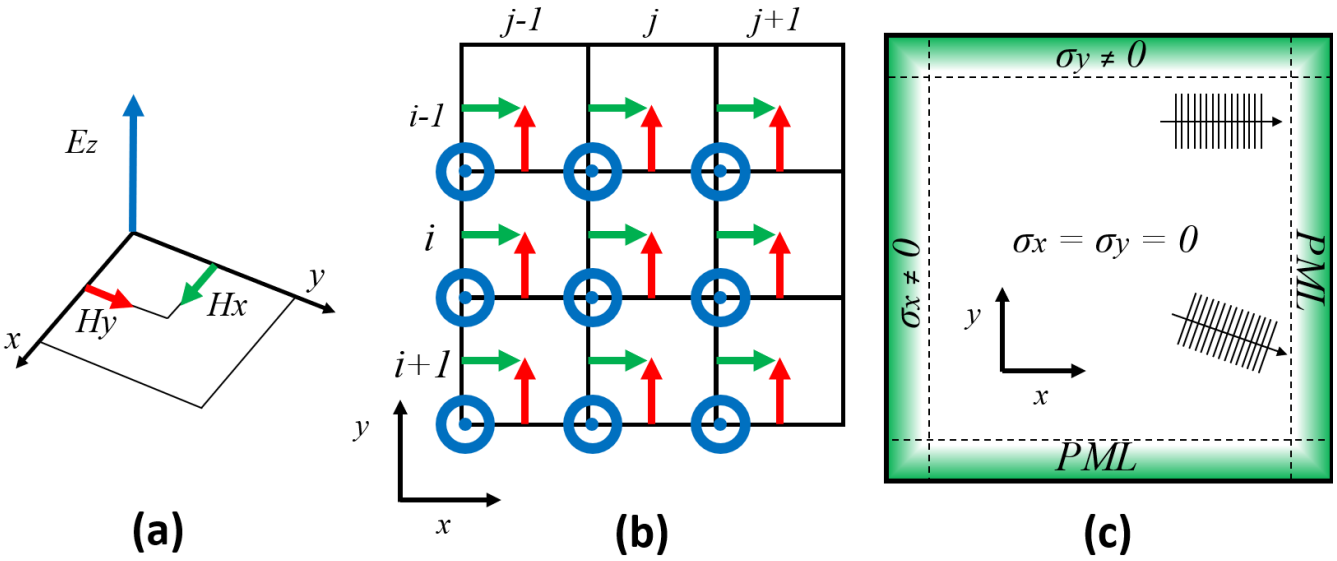

Figure 1. **(a)** Single unit cell of the Yee Grid. Electric field in Z sits on corners, and magnetic field in x and y placed half a space step from the Ez field. **(b)** Top view of Yee grid. **(c)** Schematic of PML implementation onto Yee grid. The edges are given conductivity in x and y as indicated, which is graded from the start of the region to the edge

1.2 The Perfectly matched layer

In order for simulated waves to 'leave' the simulation space at the boundaries, its common to use Berenger's[2] 'Perfectly matched layer' (PML) boundaries. A PML region is set up around the edges (Fig.1c), which absorb any incoming waves from any direction without reflection. The region is governed by the frequency domain equations:

$$\begin{aligned}
\nabla \times \vec{\tilde{E}}(\omega) &= -i\omega \frac{[\mu_r]}{c}[s]\vec{H}(\omega) \\
\nabla \times \vec{H}(\omega) &= \frac{i\omega}{c_0}[s]\vec{D}(\omega) \\
\vec{D}(\omega) &= [\varepsilon_r]\vec{\tilde{E}}(\omega)
\end{aligned} \tag{8}$$

where:

$$[s] = \begin{bmatrix} \frac{s_y s_z}{s_x} & 0 & 0 \\ 0 & \frac{s_x s_z}{s_y} & 0 \\ 0 & 0 & \frac{s_x s_y}{s_z} \end{bmatrix} \qquad \begin{aligned} s_x(x) &= 1 + \frac{\sigma'_x(x)}{i\omega\varepsilon_0} \\ s_y(y) &= 1 + \frac{\sigma'_y(y)}{i\omega\varepsilon_0} \\ s_z(z) &= 1 + \frac{\sigma'_z(z)}{i\omega\varepsilon_0} \end{aligned} \tag{9}$$

and $\sigma'_x(x), \sigma'_y(x)$ are some conductivity of the x and y edge regions respectively. In 2D, $\sigma'_z(z)$ may be ignored. These conductivities are made to gradually increase approaching the edge (Illustrated in Fig.1c). The full update equation is very complicated, and is omitted for simplicity. For illustration consider just Hx. Its update equation is:

$$j\omega\left(1 + \frac{\sigma'_y}{i\omega\varepsilon_0}\right)H_x(\omega) = -\frac{c_0}{\mu_{xx}}\left(1 + \frac{\sigma'_x}{i\omega\varepsilon_0}\right)C_x^E(\omega), \tag{10}$$

which can be brought into the time domain via a Fourier transform:

$$\frac{\partial H_x(t)}{\partial t} + \frac{\sigma'_y}{\varepsilon_0}H_x(t) = -\frac{c_0}{\mu_{xx}}C_x^E(t) - \int_{-\infty}^{t}\frac{c_0\sigma'_x}{\varepsilon_0\mu_{xx}}C_x^E(t')dt'. \tag{11}$$

This is approximated as above through finite difference, as well as a running sum of the curls to facilitate the integral.

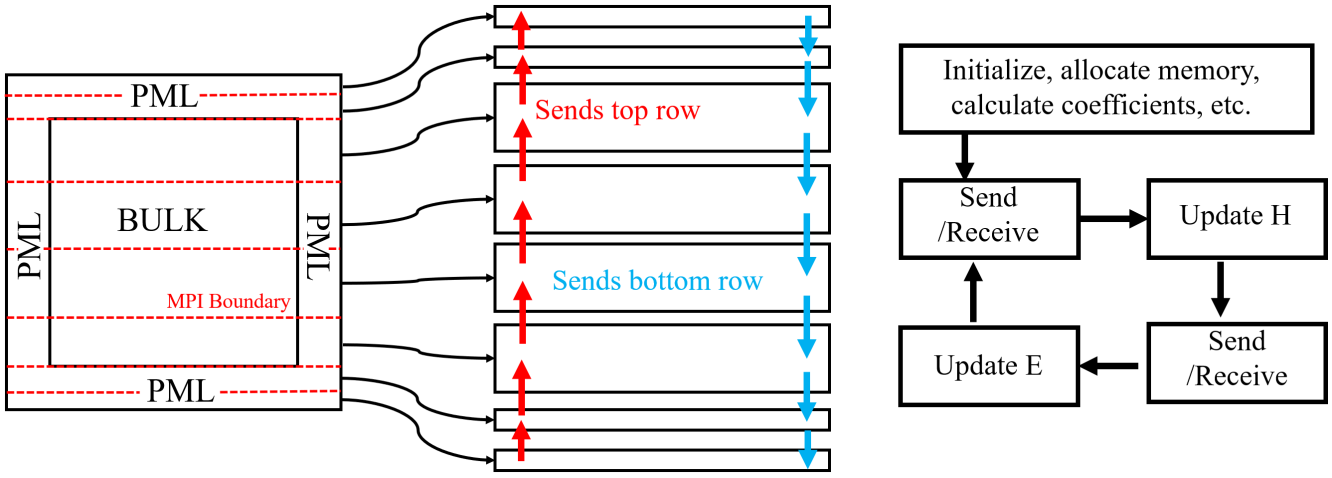

Figure 2. Schematic of an MPI parallelised workload. The Simulation space is segmented into rows, with communication between virtually 'adjacent' processes. The main iteration loop consists of sending/reciving the data from the edge rows to the adjecent processes, updating the magnetic field

2 Implementation

2.1 Parallelisation

In order to make the simulation scalable, and to make full use of the multiple cores and CPU nodes available, parallelisation is imperative. While never truly a 1:1 scaling in terms of process count to speed, breaking the compute task down into jobs that may be run alongside each other over multiple CPU cores can still reduce the simulation runtime by orders of magnitude. 2 available libraries were considered: The OpenMP and MPI libraries.

The 'Message passing interface', or MPI, is a distributed memory scheme. The process run with their own assigned memory. From the point of view of the operating system they are completely independent programmes. However, in such an implementation there will inevitably need to be some communication between processes. Since communication latency is usually much greater ($\sim \mu s$ of infiniband) than the time per CPU cycle ($\sim ns$), this can become a noticeable performance bottleneck.

OpenMP is a shared memory scheme. The code runs as a single programme, but is able to run certain tasks on different CPU cores, which all share the same memory pool. This can subvert the communication latency of MPI. However OpenMP is not scalable across CPU nodes: A CPU is only able to access the memory on the motherboard its physically plugged into. Further, this means that each core competes with the others for access over the same memory bus.

For the base of the code MPI was chosen to handle the parallelisation. The restriction to 16 processes only on a purely OpenMP code was judged unsuitable. Volakis[3] demonstrated in 2001 an implementation of an MPI based FDTD code. In it the simulation space was segmented into sub-grids, with each MPI process responsible for one segment. Our implementation follows a similar process, but with one key difference. Instead of segmenting the simulation space into grids, we segmented our space in rows, which seems more intuitive and complies with C++'s row major best practice. An illustration of the parallelisation can be found in Fig. 2. Each process sends its top and bottom row to the process 'above' and 'bellow' it in the virtual topology. The main time iteration loop consists of updating the magnetic field, sending/receiving the updated values, and then updating the E field.

'Hybrid parallelization', which combines OpenMP and MPI has been reported[4] and was attempted here. One can obtain some of the benefits of both schemes, by initiating MPI processes which span over several CPU nodes, each utilising OpenMP to parallelise across the CPU cores. This was unsuccessful in our case,actually leading to a slowdown with addition of OpenMP cores (Fig.3). Su et.al's[4] implementation mentions the non triviality of implementing OpenMP. To update the electric field the magnetic field must be sampled in both the x and y direction

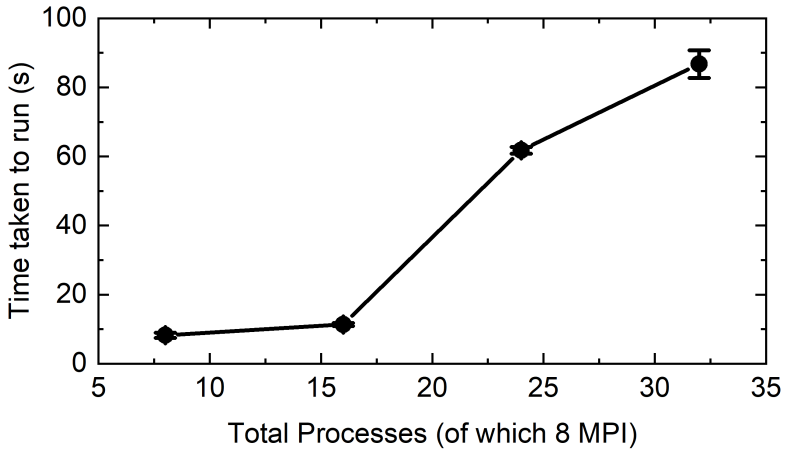

Figure 3. Attempt at 'Hybrid parallelization', running a simulation of a 800×800 point source simulation for 10000 iterations. 8 MPI processes were used to distribute the task across CPU nodes, with the remaining process being OpenMP threads. In this implementation the OpenMP threads are actually detrimental to performance

for example. Memory is arranged on the physical NAND flash in 1D rows, and so pulling data along rows and columns involves a fairly complicated memory lookup. The overhead associated with all these memory calls in OpenMP, occurring over each core trying to access similar memory locations can be an issue. They also mention how 'cache misses' caused a problem in their implementation. Solving these require fundamentally re-writing the update equations, implementing lookup tables to reduce cache misses as well as careful trial and error tuning, which was not possible to implement within the scope of this project.

2.2 PML Load balancing

A comparison of the update equations of the PML and the bulk region reveals the PML region has a much greater computational load. In the MPI row segmentation scheme, for a slice far from the top and bottom edges, most of the calculation will be with the 'bulk' update functions, only needing to use the PML update functions at left and right the edges. However rows in the top PML regions will be significantly slower than the bulk, and can become bottlenecks.

In order to prevent PML bottlenecking, more processes per unit area should be assigned to these regions than the bulk. This is determined by a 'task density density ratio', which is the ratio of the number of cores assigned per unit length (in the y direction, since we are segmenting lengthwise) of the two regions. A task density of 2 for example means there are twice as many tasks per unit length in the PML region.

Investigating different ratios reveal that a task density of 2 is optimum (Fig.4). The time taken to process number graph in Fig.4 for a ratio of 1 (no load balancing) shows a step like shape. The time taken to run remains the same despite adding more cores, because the simulation is bottlenecked by the PML region. The 'jumps' are when the processor count is high enough to naturally cause a segmentation line to pass through the PML region. A ratio of 2 remedies this, but increasing it above 2 is seen to have the opposite effect, with less cores available for the bulk. For this reason the ratio was set as two for all following tests.

2.3 C++ specific optimisations

Some optimisation specific to the C++ was explored. C++ differs from C most notably through its utility of clases. Initially the code was written in an 'object oriented' way, with each point on a Yee grid being implemented as a class. While this is intuitive and easier to maintain, timing this against an implementation with pure arrays (that is, 4 arrays holding Hx, Hy, Ez and Dz) showed a small but still noticeable speed-up (Fig.5a), and so the final code utilised this in favour of classes.

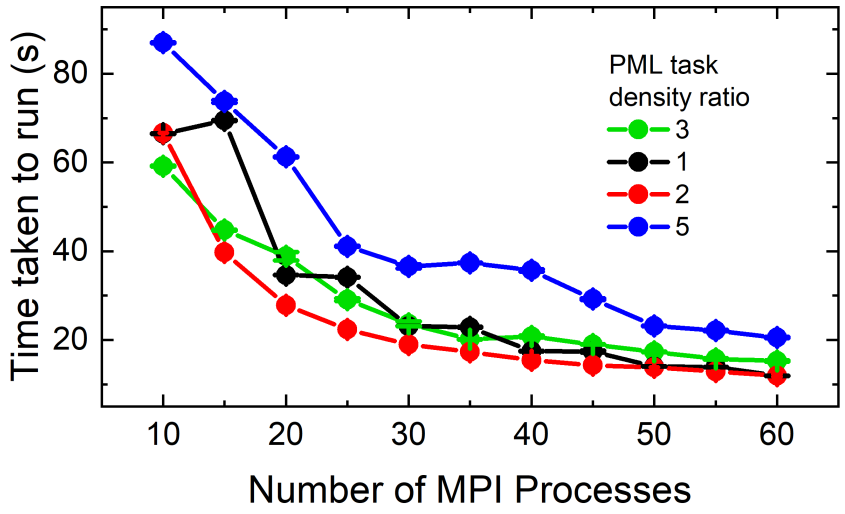

Figure 4. Time taken to run simulation at different MPI process numbers, for different PML task ratios. Simulation used was a sin wave point source on a 1000×1000 grid, at 10000 iterations. A ratio of 2 seems to be optimum

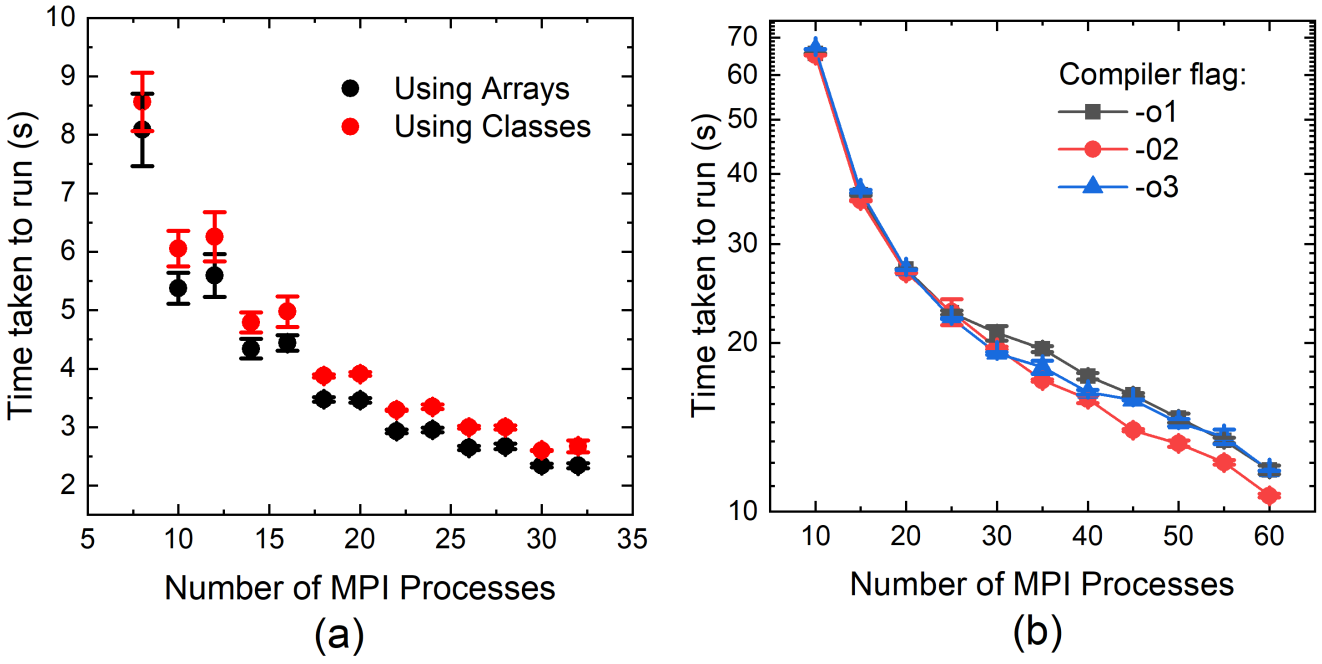

Figure 5. **(a)** Time taken to run with number of cores for code implementation using both classes and c style arrays. The C style arrays show a small performance improvement, not having the overhead involved with classes. Test done simulating a point sin wave source, with 300×300 grid with 10000 iterations. **(b)** Time taken to run vs number of cores when compiled with different g++ compiler flags. The curves are fairly similar, with the 'o2' flag being slightly faster for high processor count. Test done simulating a point sin wave source, with 1000×1000 grid with 10000 iterations.

The g++ compiler has a whole host of possible optimisations. These can be turned on via the '-o1', '-o2' and '-o3' flags. The increment in number indicates additional optimisations being turned on over the previous. The full details of what specific optimisations are turned on can be GNU foundation documentations. Exploring the effect of these on our code however showed very little effect. By a very thin margin '-o2' seemed to show the most benefit (Fig.5b), and as this is the recommended flag to use in most cases by the GNU foundation this was used to compile the programme for our testing

2.4 General Optimisation

Some general steps were taken to optimise the code. In general the philosophy adopted was to try and handle all computationally expensive operations, such as if statements and modulo operations outside of the main update loop. Values dependant on time, such as the electric field value of the source and time dependant coefficients were calculated beforehand and placed in arrays which were simply read out in the main loop. User changeable parameters, such as whether to turn I/O on or off, were handled by preprocessor flags to avoid if statements in the main loop.

In many MPI based workloads it is common to have a master rank which handles job allocation and gathers the data from all the processes. In our implementation we opted to not follow this model. Each rank in our code has 'equal footing', carrying out a similar computational load, and outputs the data in its own file. These can be read out and stitched back together in a visualisation programme fairly easily. This avoids unnecessary communication between processes: Each process only needs to send and receive 2 rows worth of data per time iteration.

3 Simulation results

3.1 Numerical Stability

The grid step of the simulation will be limited both by the size of the objects in the simulation, as well as the maximum frequency. A rule of thumb used in RF signal sampling, which we will appropriate here for our purposes, is there must be at least 10 points sampled across half the shortest wavelength in order to properly resolve the waveform.

The timestep is tied to the grid step by the numerical stability rule[5]:

$$\Delta t \leq \frac{\Delta x}{2c_0}. \tag{12}$$

Essentially this enforces that for a wave moving between 2 points on the grid, its effective speed dose not overtake that of light. When enforced, the method is able to produce natural looking waveforms (Fig.6a). If we relax this rule and allow for larger timesteps, we see numerical instability (Fig.6b). This manifests itself as a mass of high frequency noise which grows in amplitude and propagates across the simulation space. The amplitude eventually causes overflow of the double precision buffer, and an area of 'Not a Number' (NaN) values fill the grid.

3.2 Demonstration of physical phenomenon

The ability to generate point source waves is demonstrated in Fig.6. In this section we demonstrate some more interesting scenarios simulated by the code, in a somewhat indulgent attempt to demonstrate its capabilities. Objects can be placed in the simulation space, by tweaking the relative permittivity of the points they encompass. We only modelled non magnetic materials here, but of course magnetic materials may also be modelled by changing the permeability.

Lensing is demonstrated, by creating a biconcave (Fig.7a) and planar convex (Fig.7a) lens. For the former a plane wave source was incident, and a very clear convergence of the wave front on a focal point is seen. The spot is about 200nm in size, which is the incident wavelength, and is about 1 micron from the lens axis, consistent with the 2 micron radius of curvature. The latter had a radial point-source wave front incident on it, and was able to collimate it into plane waves.

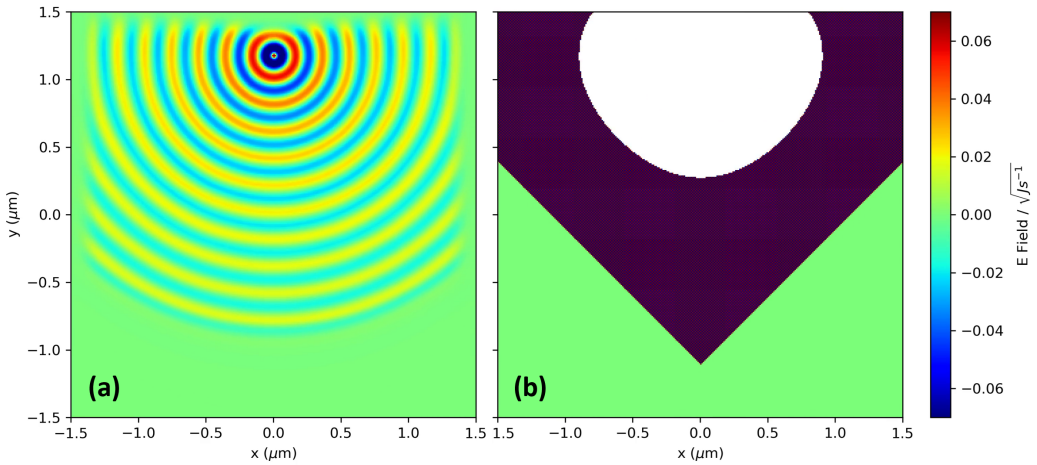

Figure 6. Simulation of a 200nm point wave source, with $\Delta x = 10nm$ and **(a)** $\Delta t = \Delta x/2c_0$, **(b)** $\Delta t = \Delta x/c_0$. The numerical instability for the larger timesteb is clear. The propagating area has a very high frequency noise, which grows exponentially. The white area is where the values have caused buffer overflow, and created a propagating area of NaN

Total internal reflection and evanescent waves can also be created (Fig.7c), by creating a small gap of lower refractive index. The simulated plane wave total internally reflects, and evanescent waves extending into the gap visible. By reducing the gap size, tunnelling of the wave is also seen.

Finally, by creating boundaries, diffraction patterns such as the famous double slit diffraction pattern can be created (Fig.7d). The interference pattern maxima and minima, with radial spacing, are clearly seen.

4 Scalability

4.1 Scaling with cores

In order to assess the speed-up with additional cores, an granular sweep of time taken to run a simulation with number of cores was conducted. Because of the way the code was constructed, it would be hugely detrimental to run with less than 8 cores. 8 is the lowest number of processes required maintain the PML task ratio of 2, which was judged optimum. Hence to ensure all the points on a number of cores to time graph are comparable, our tests begin at 8 cores.

The expected speed-up (serial time taken / parallel time taken) will be linear at first with number of cores, but this cannot go on forever. Eventually a 'diminishing returns' effect is seen. In 1967 (when 'handheld computing' depicted in episodes of Star Trek were considered unthinkably futuristic), Amdahl[6] proposed a scaling of speed-up, S, to number of compute units, N:

$$S = \frac{1}{(1-p) + \frac{p}{N}}, \tag{13}$$

$$S_{max} = S\big|_{N\to\infty} = \frac{1}{1-p} \tag{14}$$

where p was the proportion of the code that could benefit from parallelisation. We can interpret (1-p) to be not just the contribution from any serial parts of the code, but also the contributions of the MPI communication overhead and any other overheads. As the serial part of the code these (to the first order) act as a constant floor to the runtime.

Studying data of our code's runtime with number of cores shows that even with all that has changed in computing, Amdahl's scaling law (taking into account starting with 8 cores) is strikingly accurate at predicting the speed-ups and runtime (Fig.8). The points where new nodes had to be introduced are indicated on the graph. The curve seems

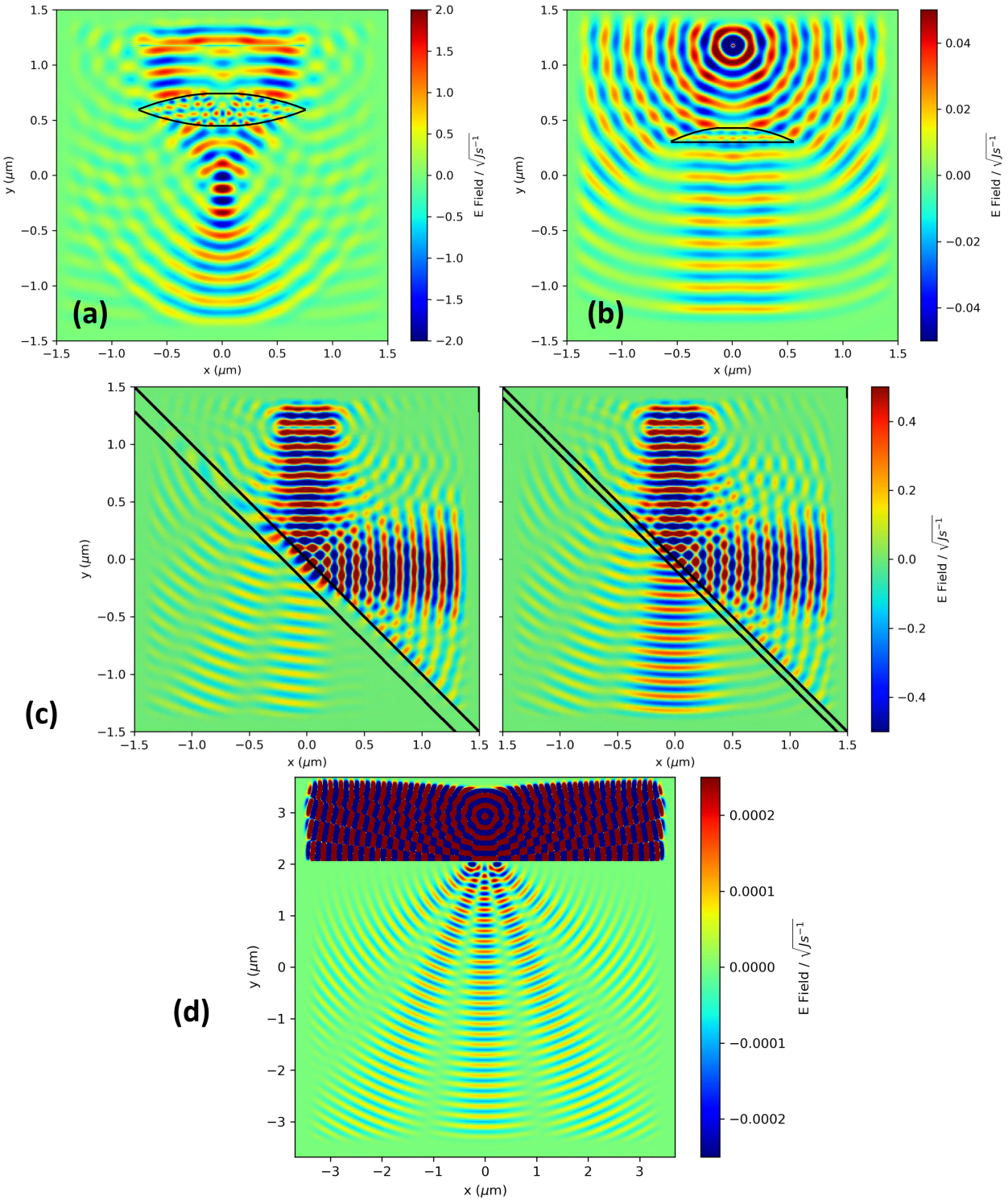

Figure 7. Demonstration of physical phenomenon, all with 200nm sources, and grid spacing of 10nm. **(a)** Focusing plane wave with bi-concave lens. The lens had a radius of curvature of 2 microns, 1.5 micron diameter and a refractive index of 4.7. **(b)** Collimation of radial light source by planar convex lens. The lens had a radius of curvature of 1.25 microns, and a diameter of 1.1 micron. **(c)** Evanescent tunnelling: The grid is given a refractive index of 2, except a small gap at 45^o sized 200nm (left) and 80nm (right). The evanescent waves are visible within the larger gap, and tunnelling occurs in the smaller gap. **(d)** Double slit diffraction: Slit width was 10nm, slit separation of 500nm. Note the diffracted waves have much lower amplitudes than the incident wave, hence the apparent saturation in that part of the image

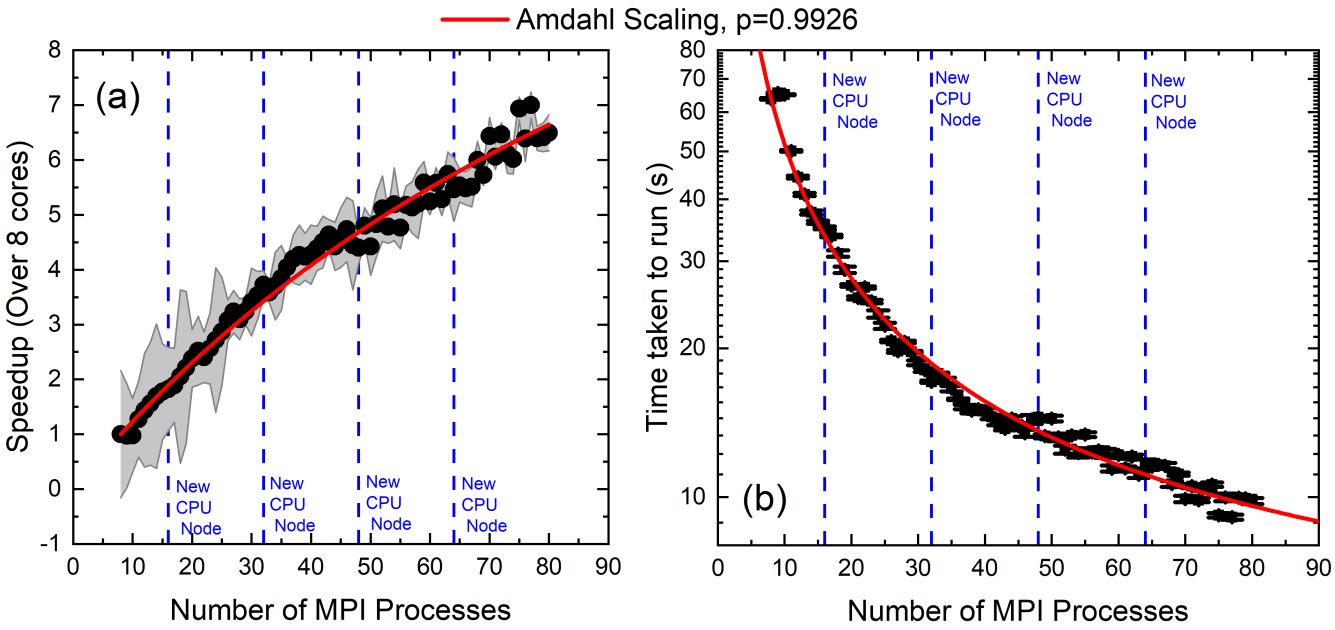

Figure 8. **(a)** Speed-up and **(b)** time taken to run point source sin wave simulation (1000×1000 for 10000 iterations)with number of cores given for MPI process. The lines where a new node was required to fulfil the requested number of cores is also indicated. Best fit for (a) was found to be a two phased exponential decay of time taken with processor number. The speed-up in (b) is ideal at first, but deviates at higher core count

to not interact with these 'boundaries' in any observable way, and so we can say inter-node communication did not effect the timings much (at least in the range tested). Hence given the time taken at any number of cores for a specific simulation we can predict its runtime at a larger number of cores. The p parameter for our code was found to be

$$p = 0.9926, \tag{15}$$

making our theoretical maximum speed-up:

$$S_{max} = \frac{1}{1 - 0.9926} \approx 135 = 17.8 \cdot S\big|_{N=8} \tag{16}$$

From this, as well as examining Fig.8a we can see a $10\times$ increase in core count from 8 to 80, yields a $7\times$ speed-up, which is not bad. However, approaching the 100 core mark the returns begin to diminish. Eq.14 combined wit Eq.16 predicts going up to 200 cores ($25\times$ the 8 core base) will only yield a $11\times$ speed-up over 8 cores. This may be beneficial for larger models, but for smaller ones, if core count is at a premium, staying bellow 100 cores seems reasonable.

4.2 Scaling with simulation size

Having determined the speed-up with number of cores, we must examine also the effect of simulation size. The 'size' in this case is quantified by the number of grid points (or 'Pixels') in our simulation. The number of Pixels is an area quantity, and scales with the square of the simulation dimensions.

Producing runtime vs number of core curves for different simulation areas (Fig.9a) shows that the time taken increases on the same order as the area, apparently shifting up on the time axis. Studying the shape difference by normalising all the curves to the largest (first) runtime in their series reveals that for larger sized simulations the shapes of the curves are very similar, following the Amdhal scaling described in the previous section.That discussion was based on a 1000×1000 grid, so we can say for larger simulation sizes the assertions made there are still valid. However it is also apparent that for smaller simulation areas, the scaling is less effective, and in the case of the

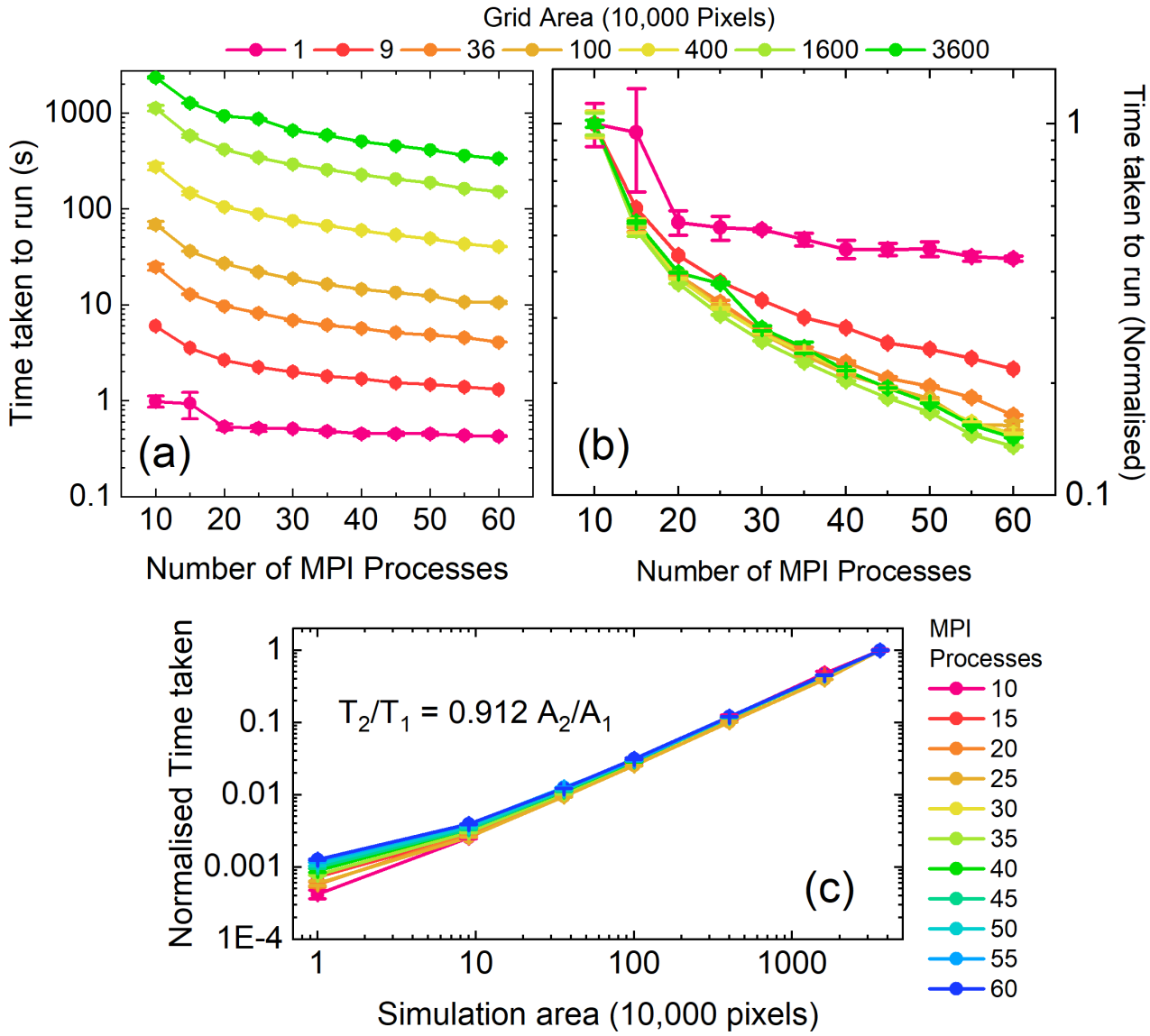

Figure 9. Scaling of simulation runtime with number of cores for different simulation sizes. Times calculated from ruining a point source sin wave for 10,000 iterations. **(a)** Plots without any scaling. The curves are relatively evenly spaced. **(b)** Plot of number of cores/time graph, normalised to the 1st time value. For smaller sizes the shapes differ but the general shape is the same for larger sizes. **(c)** Time taken to run normalised to the time taken for the largest area simulation, for different number of cores. Save for the first point, the same linear scaling with area is seen, regardless of number of cores

100×100 grid hits a floor and completely flattens out. This is likely due to each rank having such a small workload that the MPI communication floor is hit, preventing further speed-ups.

The spacing in (Fig.9a) seems te be fairly well correlated with the areas themselves. Studying this further we can produce a graph of the time taken as a function of area for different number of cores, normalised to the max value for easier comparison (Fig.9c). It can be seen that save for the very first point (which corresponds to the discussed 100×100 grid that hit a floor), the time taken scales linearly with area, and identically regardless of the number of cores. This makes intuitive sense, as the area determines the number of operations it must carry out per time iteration to update the whole space. By performing a linear fit, we can devise the area to time relationship:

$$\frac{T_2}{T_1} = 0.912 \left(\frac{A_2}{A_1} \right), \tag{17}$$

where A and T are the areas and time taken for a certain simulation at a certain number of cores. Interestingly, the gradient is slightly less than one. The linearity is already good news for scalability, as larger simulations will require proportionally increased resources, and the sub unity gradient just adds to this benefit.

Hence, different simulations will have different runtimes, but if we know the time taken for a certain sized simulation for a certain number of cores, by combining Eq.14, Eq.16 and Eq.17, we can predict the timings if the simulation is scaled up.

5 Conclusion

We developed a FDTD code, parallelised using the MPI library. We demonstrated the optimum load balancing scheme for dealing with the PML regions, and demonstrated the effect of various optimisations to the code. The code's capabilities were demonstrated in its ability to simulate electromagnetic waves, as well as physical effects such as lensing, evanescent tunnelling and double slit diffraction. Finally, it was shown to be scalable, showing good speed up with number of cores and a linear scaling with simulation space area.

References

1. Yee, K. S. Numerical Solution of Initial Boundary Value Problems Involving Maxwell's Equations in Isotropic Media (1966).
2. Berenger, J. P. A perfectly matched layer for the absorption of electromagnetic waves. *Journal of Computational Physics* **114**, 185–200 (1994).
3. Volakis, J. L., Davidson, D. B., Guiffaut, C. & Mahdjoubi, K. A parallel FDTD algorithm using the MPI library (2001).
4. Su, M. F., El-Kady, I., Bader, D. A. & Lin, S. Y. A novel FDTD application featuring OpenMP-MPI hybrid parallelization. In *Proceedings of the International Conference on Parallel Processing*, 373–379 (2004).
5. Taflove Susan Hagness, A. C. & London, B. I. Computational Electrodynamics The Finite-Difference Time-Domain Method Third Edition ARTECH HOUSE. Tech. Rep.
6. Amdahl, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings - 1967 Spring Joint Computer Conference, AFIPS 1967*, 483–485 (Association for Computing Machinery, Inc, New York, New York, USA, 1967).