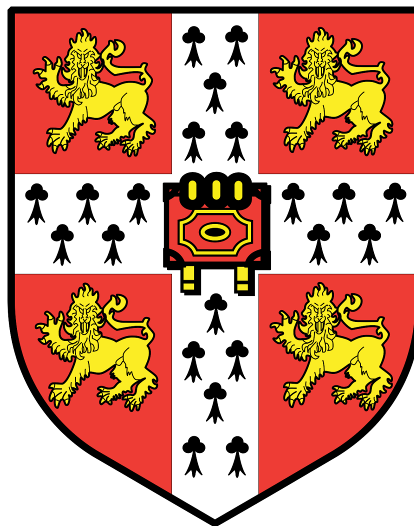
Investigation of the scalability and correctness of the random ray method

Final dissertation

Max Forbes



MPhil Nuclear Energy
Department of Engineering
University of Cambridge
Dr. Paul Cosgrove and Prof. Andrew Davis
August 25, 2021

Declaration

This dissertation is submitted for the degree of Master of Philosophy. This dissertation is substantially my own work and conforms to the University of Cambridge's guidelines on plagiarism. Where reference has been made to other research this is acknowledged in the text and bibliography. By Cambridge Department of Engineering policy, this work does not exceed 15000 words.

Acknowledgements

I would like to sincerely thank Dr. Paul Cosgrove, without whose tireless help this project would not have been possible. His willingness to impart his (seemingly infinite) knowledge, and part with his valuable time, are what allowed me to complete this project to the best of my ability. I would like to also thank my industrial supervisor Prof. Andrew Davis who provided the industry knowledge and fusion know-how for this project to succeed.

Finally I would like to thank both my family and my partner, Liv, for their endless support and belief. I believe it is their patience and motivation that is responsible for my sanity today, and for that I am extremely grateful.

This work was performed using resources provided by the Cambridge Service for Data Driven Discovery (CSD3) operated by the University of Cambridge Research Computing Service (www.csd3.cam.ac.uk), provided by Dell EMC and Intel using Tier-2 funding from the Engineering and Physical Sciences Research Council (capital grant EP/P020259/1), and DiRAC funding from the Science and Technology Facilities Council (www.dirac.ac.uk).

Abstract

Efforts to solve key problems facing nuclear fusion system development such as radiation damage and tritium breeding benefit from the use of the fastest and most accurate neutron transport solver available. Given the state of modern computational hardware, the run time of a solver is heavily dependent on its parallel scalability. In this project the scalability and correctness of the random ray method (TRRM) and discrete ordinates method were analysed. Strong and weak scaling performance was observed through the completion of various sized simulations on up to 256 threads using mini app implementations of the methods. Results showed that TRRM achieved a maximum speed up of 54.49±0.82 on 256 threads, and discrete ordinates achieved a maximum speed up of 7.26±0.39 on 256 threads. Both method exhibited poor weak scaling performance due to a combination of load imbalance, memory bottlenecks and serial fractions of code that slowed with problem size. Correctness assessments involved the simulation of three different geometries with two discrete energy groups designed to investigate how each method dealt with down scattering and neutron streaming. These simulations were compared to a reference solution calculated using Serpent [1]. It was found that ray effects inhibited the ability of discrete ordinates method to accurately represent neutron streaming, and TRRM results differed slightly from the reference solution due to the necessary use of different interaction cross sections. It was also found that discrete ordinates more accurately simulated thermal neutron production within materials, on average 0.45% larger than the reference solution compared to a mean difference of 3.13% from the TRRM solver.

Contents

1 Introduction **7**
- 1.1 Motivation . . . 7
- 1.2 Background . . . 8
  - 1.2.1 The random ray method . . . 9
  - 1.2.2 Deterministic - S_N discrete ordinates . . . 13
  - 1.2.3 Stochastic - Monte Carlo . . . 15
- 1.3 Scalability . . . 17
  - 1.3.1 Strong scaling . . . 17
  - 1.3.2 Weak scaling . . . 18
  - 1.3.3 Parallel programming . . . 18
- 1.4 Review of previous work . . . 20
  - 1.4.1 Parallel scaling of TRRM . . . 20
  - 1.4.2 Parallel scaling of S_N discrete ordinates . . . 23

2 Code modifications **25**
- 2.1 Minray . . . 25
  - 2.1.1 Fusion geometry loading . . . 25
  - 2.1.2 Shannon entropy . . . 26
  - 2.1.3 Source averaged relative error . . . 27
  - 2.1.4 Parallel implementation . . . 28
- 2.2 Kripke . . . 28
  - 2.2.1 Adding I/O capabilities . . . 28

3 Methodology **29**
- 3.1 Geometry and cross section generation . . . 29
- 3.2 Assessing scaling . . . 34

4 Results and discussion **35**
- 4.1 Scalability . . . 35
  - 4.1.1 Strong scaling . . . 35
  - 4.1.2 Weak scaling . . . 39
  - 4.1.3 Compiler dependency . . . 41
- 4.2 Correctness . . . 43
  - 4.2.1 Beryllium shell . . . 43
  - 4.2.2 Beryllium shell - Streaming paths . . . 47
  - 4.2.3 KANT . . . 50

5 Conclusions **54**

List of Figures

1 A flux distribution highlighting the ray effect. 15
2 An image demonstrating how data is loaded from memory to cache lines in shared-memory architecture [2]. 20
3 A plot of speedup against thread count for ARRC. Adapted from [3]. 21
4 Weak and strong scaling results of OpenMOC adapted from [4]. 22
5 Weak and strong scaling results of discrete ordinates adapted from [5]. 24
6 A plot of SARE with iteration number before meeting the threshold in Minray. 27
7 An example of the beryllium shell geometry generated using Serpent. 30
8 An example of the beryllium tunnel geometry generated using SERPENT. 31
9 An example of the KANT benchmark geometry provided by CCFE. 32
10 Down scaled material distributions of the three geometries. . . 33
11 A plot of parallel speedup against thread number in strong scaling assessments of Minray on a KNL CSD3 node and a best fit of Amdahl's law. 36
12 A reduced plot of parallel speedup against thread number in strong scaling assessments of Minray on a KNL CSD3 node alongside an example of ideal strong scaling. 37
13 A plot of parallel speedup against thread number in strong scaling assessments of Kripke on a KNL CSD3 node and a best fit of Amdahl's law. 38
14 A plot of strong scaling results for both Kripke and Minray against thread number. 39
15 A plot of the weak efficiency of Minray and Kripke against number of threads run on a CSD3 Skylake node. 40
16 A plot of the strong scaling speedup of Minray against number of threads when compiled with both GCC 5.4.0 and ICC Version 2021.1. 42

17 A plot of the weak efficiency of Minray against number of threads when compiled with both GCC 5.4.0 and ICC Version 2021.1. 42
18 A pseudo color plot of the normalised slow flux calculated by Kripke in the beryllium shell geometry. 44
19 A pseudo color plot of the normalised slow flux calculated by Minray in the beryllium shell geometry. 44
20 A pseudo color plot of the normalised slow flux reference solution calculated by Serpent in the beryllium shell geometry. . 45
21 A plot of the normalised radial flux distribution for all three methods in the beryllium shell geometry. 46
22 A pseudo color plot of the normalised fast flux calculated by Kripke in the modified beryllium shell geometry with streaming paths. 47
23 A pseudo color plot of the normalised fast flux calculated by Minray in the modified beryllium shell geometry with streaming paths. 48
24 A pseudo color plot of the normalised fast flux reference solution calculated by Serpent in the modified beryllium shell geometry with streaming paths. 48
25 A plot of the normalised radial fast flux distribution for all three methods in the modified beryllium shell geometry with streaming paths. 49
26 A plot of the absolute error in normalised fast flux solutions calculated by Minray and Serpent in the modified beryllium shell geometry with streaming paths. 50
27 A pseudo color plot of the normalised fast flux calculated by Kripke in the KANT benchmark geometry. 51
28 A pseudo color plot of the normalised fast flux calculated by Minray in the KANT benchmark geometry. 51
29 A pseudo color plot of the normalised fast flux reference solution calculated by Serpent in the KANT benchmark geometry. 52
30 A plot of the error in normalised fast flux solutions calculated by Minray and Serpent in the KANT benchmark geometry. . . 52
31 A plot of the absolute error in normalised fast flux solutions calculated by Minray and Serpent in the KANT benchmark geometry. 53

List of Tables

1 Energy boundaries for the two group energy discretisation used throughout correctness assessments. 30
2 A table documenting the problem dimensions simulated for each thread number in weak scaling assessment. 35

1 Introduction

1.1 Motivation

Since the observation of radiation damage in the very first commercial fission reactors of the 1970's, research into materials capable of withstanding the intense radiation of a fusion plasma has been active [6]. The breeding of tritium for fuel use in D-T fusion reactors is also paramount to their success. For these and many other problems, fast and accurate neutron transport simulation is necessary to assess the large number of potential system designs, that differ only slightly in geometric and material composition.

When compared to traditional nuclear fission systems, there are a number of key differences that affect which method of neutronics simulation is most accurate and computationally efficient for fusion geometries [7]:

- More anisotropic neutron scattering
- Fixed neutron source
- Neutron streaming paths

Most often in fission reactor simulation, neutrons are produced through fission and the neutron source is proportional to the scalar flux. The ratio between the number of neutrons born in one generation to those born in the next is calculated, known as eigenvalue calculations. In fusion simulation however, fixed source calculations are performed in which the neutron source is independent of flux. In order to accurately simulate neutron streaming, a fine angular representation is required. Fusion systems also show more anisotropic neutron scattering, which must be dealt with by the method used.

Typically methods can be divided into two categories, stochastic and deterministic, each offering its own benefits and drawbacks. Stochastic methods are those that produce results with an element of statistical uncertainty, typically a result of the use of random sampling in calculations. In contrast, deterministic methods are those that always produce the same results, based on the input, with no statistical noise. Due to the high resolution interaction data that is afforded by its continuous energy calculations, and its ability to include accurate geometric descriptions, Monte Carlo is most often used for neutron transport simulation in fusion geometries and is considered the

de facto standard. Monte Carlo is not without fault, however, as stochastic methods introduce both long simulation times and statistical errors [8]. Alternatively deterministic approaches such as discrete ordinates offer faster calculation times, but do not offer the spatial or energetic fidelity of Monte Carlo methods and fail to represent the small streaming effects often present in fusion systems. They also tend to have a higher memory footprint than Monte Carlo simulations, and often exhibit worse parallel scaling.

Moores law accurately projected how the number of transistors present on an integrated circuit would double every two years, however it has been found that power draw grows non-linearly with this performance improvement [9][10]. Recent developments in computer architecture have focused on the optimization of power per performance, with a lessened focus on the performance of individual nodes. These developments have led to the number of cores present on CPU's increasing, and the serial computational capabilities of each of these cores stagnating between successive product launches from manufacturers such as Intel and AMD. As a result, there has been a shift in responsibility for performance, from a reliance on hardware manufacturers, to software developers producing parallel software that benefits from the larger number of cores available [11]. This effect, coupled with the need for quick neutron transport simulation with the aim of rapidly improving fusion reactor design, means that the most scalable method of neutron transport simulation for fusion geometries would allow for the most efficient use of both time and resources on modern hardware. Therefore, investigations into the scalability and correctness of each method of neutron transport are necessary to permit the most advantageous approach to the obstacles facing the success of nuclear fusion system design [12].

The goal of this project is to evaluate and compare the scalability and correctness of both the discrete ordinates method and a hybrid stochastic and deterministic method known as The Random Ray Method (TRRM). Scalability will be assessed by recording the run time of each method when simulating a variety of problem sizes on an varying number of threads. Correctness will be assessed by comparing scalar flux solutions of the methods to a reference solution calculated with a Monte Carlo code.

1.2 Background

The focal point of neutron transport simulation is the solution of the steady state neutron transport equation [13]. Typically this takes the form

$$\Omega \cdot \nabla\psi(\vec{r}, \vec{\Omega}, E) + \Sigma_T(\vec{r}, E)\psi(\vec{r}, \vec{\Omega}, E) = Q(\vec{r}, \vec{\Omega}, E) \tag{1}$$

where ψ is the neutron flux vector, $\vec{r}$ is the position vector of the flux, $\vec{\Omega}$ is the angle of travel of neutrons, Σ_T is the total cross section and Q is a neutron source term. In fission problems Q often consists of both a scattering and fission term, however a fusion source term consists of scattering and, within the plasma, a fixed source term. There are a variety of solution methods to (1), a number of which are explained below.

1.2.1 The random ray method

The random ray method (TRRM) of neutron transport simulation is somewhat a hybrid between stochastic and deterministic techniques. Although stochastic by definition, the method is based upon the Method of Characteristics, a deterministic method for solving the neutron transport equation [14].

In the Method of Characteristics (MoC) the PDE neutron transport equation is solved along characteristic tracks in the form of ordinary differential equations. The characteristic form of (1) can be derived by parameterising

$$\vec{r} = \vec{r_0} + s\vec{\Omega} \tag{2}$$

where $\vec{r_0}$ is a reference location for a track, and s is the distance along a track at angle $\vec{\Omega}$. This allows the transport equation to be written in its characteristic form, that is spatially dependent only on s

$$\frac{d}{ds}\psi(s, \vec{\Omega}, E) + \Sigma_t(s, E)\psi(s, \vec{\Omega}, E) = Q(s, \vec{\Omega}, E) \tag{3}$$

which can be discretised in angle and energy, and so the angular dependence is removed for individual tracks, and energy is discretised using a multi group approximation, where neutrons are divided into energy bins and Equation (3) is solved for each. It is also possible to discretise (3) spatially, by dividing the geometry into homogeneous regions with a flat source called FSRs (Flat Source Regions). This relation can be simplified to an attenuation and accumulation of angular flux along a single characteristic segment between $s^{'}$ and $s^{''}$. This is done by applying an integrating factor solution to Equation (3) given the previous assumptions, and allows the change in angular flux along a given segment k across an FSR i to be expressed as

$$\Delta\psi_{k,g} = \left(\psi_{k,g}(s^{'}) - \frac{Q_{i,g}}{\Sigma^{T}_{i,g}}\right)(1 - e^{-\tau_{k,i,g}}) \quad (4)$$

where the optical length $\tau_{k,i,g}$ is the product of the FSR's macroscopic transport cross section and the segment length $\tau_{k,i,g} = \Sigma^{T}_{i,g}(s^{''} - s^{'})$ [15]. By assuming isotropic scattering, the source $Q_{i,q}$ of an FSR can be expressed as

$$Q_{i,g} = \frac{1}{4\pi}\sum_{g^{'}=1}^{G}\Sigma^{S}_{i,g^{'}\rightarrow g}\phi_{i,g^{'}} + F_{i,g} \quad (5)$$

where $\Sigma^{S}_{i,g^{'}\rightarrow g}$ is the macroscopic scattering cross section from group $g^{'}$ to g, $F_{i,g}$ is the neutron source term arising from fusion, typically zero in regions outside of the plasma, and $\phi_{i,g^{'}}$ is the scalar flux in group $g^{'}$ and FSR i, defined as the integral of $\psi_{g^{'}}$ over all angles.

The scalar flux in each FSR can be calculated using the following expression

$$\phi_{i,g} = \frac{4\pi}{\Sigma_{i,g}}(Q_{i,g} + \frac{1}{4\pi}\sum_{k}\omega_{k}\Delta\psi_{k,i,g}) \quad (6)$$

where ω_k is the weight associated with segment k, corresponding to the angular quadrature, calculated by normalising the segments length with the total distance covered by all tracks. These three equations (4), (5) and (6) constitute the iterative procedure that underpins both MoC and TRRM. The two methods differ in how track quadrature, the distribution of track starting positions and angles, is selected. In a typical MoC solver the quadrature is pre-determined and static, the origin and angle of tracks is chosen to suit the systems' geometry. In TRRM however, the choice of quadrature is stochastic, and the origin and direction of travel of each track is randomly sampled upon each iteration [16]. This random sampling introduces another step to computation, but removes the need for quadrature to be accessed every iteration from storage, altering the scalability of TRRM in comparison to MoC.

The algorithmic structure of the fixed source iteration cycle of the random ray method can be seen in Algorithm 1.

Iterations can be active or inactive, and the maximum number of iterations is a sum of the two. The first iterations are inactive, this allows the scattering source to converge without affecting the mean flux. When inactive iterations are complete, active iterations begin and the flux of each iteration

Algorithm 1 The random ray power cycle

```
Initialise all scalar fluxes to 1.0
while Iteration ≤ Maximum number of iterations do
    Update sources with old scalar flux (Equation (5))
    Transport sweep (Algorithm 2)
    Normalise scalar flux to total track distance
    Add source to scalar flux (Equation (6))
    if Active iteration then
        Add scalar flux to accumulator
    end if
end while
```

contributes to a mean. It is necessary to compute a mean solution, as the generation of tracks in TRRM introduces random error to the result of each iteration. This is dealt with statistically through the use of a mean. The number of active and inactive iterations can be set by the user. Alternatively an automatic technique to determine if the source is converged can be employed.

One such technique is the monitoring of the Shannon entropy of the source on each iteration [17]. Shannon entropy is a single scalar value, that itself is of no physical importance, but converges with the source of a problem [16]. This is calculated by subdividing the number of FSR's in the problem into N larger chunks known as the Shannon entropy mesh. For each mesh cell the following expression is evaluated

$$P(n) = \frac{V(n)}{S} \sum_{g}^{G} [\Sigma_g^s(n) \phi_g(n)] \tag{7}$$

where $V(n)$ is the volume of mesh cell n, and $S(n)$ and $\Sigma_g^s(n)$ are the scattering source, and scattering cross section of group g within mesh cell n respectively. By accumulating this value $P(n)$ for all mesh cells, the Shannon entropy of the system can be determined using the expression

$$H_{src} = -\sum_{n}^{N} P(n) \log_2(P(n)) \tag{8}$$

where the Shannon entropy of a given problem is a fixed value, and is always

between $H_{src} = 0$ for a point source problem and $H_{src} = \log_2 N$ for a flat source distribution where again N is the number of mesh cells used [18]. Once the Shannon entropy of the problem is deemed to have converged, iterations become active, and the fluxes contribute to the mean.

The majority of the TRRM power loop can be completed in parallel. The most computationally intensive portion of the loop is the transport sweep algorithm seen in Algorithm 2. It should be mentioned that the power cycle and transport sweep of fixed source MoC are similar to Algorithms 1 and 2, with the notable differences of source calculation and ray generation.

Algorithm 2 TRRM Transport Sweep

```
for Ray ≤ Maximum number of rays do
    Distance travelled D = 0
    Generate random ray position and angle
    while D < Termination distance do
        Calculate distance to next cell s
        Attenuate along s
        D = D + s
    end while
end for
```

Termination distance is set by the user and is the imposes a maximum length on a track. On a single iteration the sweep follows each characteristic along its length, calculating the change in flux of every segment in a track until the termination distance is reached and moving on to the next track. The flux of an FSR is calculated by combining the flux calculations of all segments that have passed through it and normalising by the total length of tracks covered in that iteration. This results in a flux solution which, whilst flux is not converged, is used to update the source terms prior to beginning the following iteration. In both MoC and TRRM a large number of tracks with small separation produces the most accurate results, but this requires a larger number of tracks to be followed in each transport sweep, presenting a situation in which the accuracy benefits of increased fineness must be balanced with a negative impact on run times. Due to the random ray generation each iteration, TRRM can accumulate a large total number of track angles and positions over the course of a simulation, with the computation of fewer tracks per iteration. Another benefit of TRRM is that it removes reliance on accessing track segment data from storage which can act as a bottleneck to

efficient parallel performance, improving the potential scalablity of TRRM. In typical MoC the set of angles and positions in the quadrature are chosen that best suit a problems geometry, in order to produce the most accurate result. The accumulation of track angles and positions with each iteration in TRRM means that this method can be readily applied to any geometry. Each problem does not require the preparation of a bespoke track quadrature before hand, removing a further aspect of non-computational difficulty.

A disadvantage to the application of TRRM to fusion systems is the approximation of an isotropic scattering source. As mentioned in Section 1.1, anisotropic scattering is commonplace in fusion systems, and thus necessitates the use of correction methods to account for this approximation.

1.2.2 Deterministic - S_N discrete ordinates

The S_N or discrete ordinates method of solving the neutron transport was first used for radiative heat transfer calculations by Chandresekhar in astrophysics before being applied to nuclear reactor cores at Los Alamos [19, 20]. The neutron transport equation shown in (1) is dependent on 6 variables: 3 position, 2 angular, and 1 energy. The approach of the S_N method is to discretise the transport equation in each of these variables.

Energy discretisation is performed using a multi group approximation and angular and spatial variables are also both discretised the same way as TRRM and MoC. Unline MoC, spacial discretisation in discrete ordinates is most often uses the diamond difference scheme, a numerical approximation that relates the flux at the surface of the mesh cell to the average flux throughout the cell volume [21]. This allows the transport equation to be represented by a large set of linear algebraic equations that can be solved when represented in matrix form [22]. Due to their large size, they can not be solved directly, but instead an iterative approach is taken. The general form of the equation used in discrete ordinates method is

$$L\psi^{i+1} = S\psi^{i} + Q \tag{9}$$

where i is the notation used for the iteration number, L is a leakage operator, S is a scattering operator, and q is a source term. This equation is used first with a guess for the flux of the system, ψ^0. It is then solved using the source iteration technique, using the algorithmic structure detailed in Algorithm 3. On each iteration this process returns a flux distribution that is closer to the

final value than the last, with no stochastic noise. For this reason an average of results is unnecessary, and the iteration procedure can be terminated once the flux solutions of successive iterations differ by less than a tolerance set by the user.

Algorithm 3 Discrete ordinates iteration loop

while Scalar flux not converged **do**
  Calculate scattering $\phi = S\psi^i$
  Add fixed source $\phi^i_{out} = \phi^i + Q$
  Perform transport sweep (invert L) $\psi^{i+1} = L^{-1}\phi^i_{out}$
end while

In order for the inversion of L to occur, a sweep across the mesh is completed. This transport sweep loops across the entire mesh structure between neighbour cells, and can be completed using a number of algorithms. The transport sweep algorithm is the manner in which the code iterates across all cells, and the choice of transport sweep algorithm can be expected to have an effect on the scalability of the method. This is because different transport sweeps will perform a different number of calculations per variable.

The method of discrete ordinates introduces a number of issues. Firstly, it involves the discretisation of six variables, resulting in very large number of unknowns [23]. In order to accurately represent the large and complex fusion geometry, a correspondingly large and fine mesh is required. All of these unknowns occupy a space in memory, and so memory bottlenecks are common in discrete ordinates methods, affecting their scalability. Unlike Monte Carlo and TRRM calculations, in which the same approach can be taken with a wide number of different systems, the S_n method must be altered for each problem it simulates: the choice of transport sweep, spatial and angular discretisation as well as the energy group structure should all be chosen with relevance to the problem, necessitating reliable heuristics or experience on the part of the user.

A final disadvantage to S_n discrete ordinates is known as the ray effect first observed by Lathrop in 1968 [24]. This is a fundamental issue with the use of angular discretisation, where in fact the angular nature of radiation is continuous [25]. This is independent to the spatial discretisation of the problem, and results in a distorted flux distribution in the direction of angles treated by the discretised solver. An example of this can be seen in Figure 1.

A discrete ordinates simulation of a point source within a sphere of beryllium, in which too few (8) polar angles were used, resulting in a non-physical flux distribution.

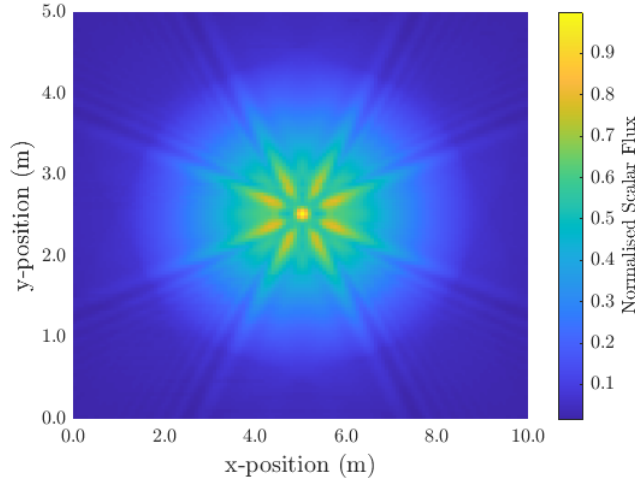


Figure 1: A flux distribution highlighting the ray effect.

This can often be remedied through the use of a larger number of angles in the discretisation (increasing computational cost), but can also sometimes be a result of the geometry itself. Systems with isolated scattering regions or an isolated source region within an absorbing or void medium can exhibit the ray effect more readily than others, and often fusion systems fall victim to having a source region within a void material [26].

1.2.3 Stochastic - Monte Carlo

The Monte Carlo method was first derived during nuclear weapons research and involves the implicit solution of (1) [27]. There exists a probability distribution that a given particle will travel a certain distance before collision, and when that collision occurs, there is a probability associated with each of the potential interactions of this particle. By the generation of (pseudo) random numbers, these distributions can be sampled and a particle can be simulated throughout its lifetime. Although this may not represent the exact lifetime of any given particle, the case is that given the simulation of a sufficiently large number of particles, the average behaviour of the entire system can be determined [28].

Monte Carlo codes allow for the exact representation of system geometry. This is often done by dividing the system into arbitrarily complex cells, each made of a given material possessing tabulated interaction cross sections that are not subject to the multi group approximation. A particle is generated, its location and the interaction cross sections of its enclosing cell and its neighbours are determined. Using this data the travel distance of the particle is determined, by sampling a random number. When moved to this new location, the program samples the potential interactions of the collision, and determines the resulting particles. If a scattering interaction, this would include altering the particles' energy and direction, or in the case of a fission interaction this would include adding neutrons to the system.

This process is repeated for a large number of particles. The more particles that are simulated, the smaller the variance in the calculated values of interest such as reaction rates or dose estimates. This is a result of the central limit theorem, relating the variance of a mean to the number of data points used in its calculation

$$\sigma^2 = \frac{1}{N} \tag{10}$$

where σ is the standard deviation in the mean and N is the number of particles sampled [29]. The benefit of increasing the number of particles simulated in order to reduce variance must be balanced with the added computational intensity of the problem. In the past, technological limitations on the number of simulated particles resulted in the necessary use of variance reduction techniques, designed to reduce the uncertainty in results whilst simulating the same number of particle lifetimes. These techniques are also necessary in deep penetration problems. Where a small fraction of simulated neutrons are able to reach far from the source, or through optically thick regions of geometry, resulting in larger statistical fluctuations in these areas of the geometry [30]. This is known as the low sampling phenomenon [31] and is of particular relevance to fusion, where accurate dose rate estimates in the edges of the reactor are necessary for tritium breeding calculations, as tritium breeding often occurs in blankets surrounding the reactor. Monte Carlo is inhibited by the access of large amounts of data from memory. The energy-dependent cross section data that is used by Monte Carlo solvers must be accessed, and then interpolated. In fact a relatively small fraction of the computational time used in Monte Carlo is calculations, the majority is memory access [32].

1.3 Scalability

The scalability of a program is its ability to utilize a larger number of available processors. There are two main approaches to assessing the scalability of an algorithm or program.

1.3.1 Strong scaling

Strong scaling is the speedup associated with running the same size problem, that is, the same number of unknowns, with a larger number of processors. Whilst the total work remains constant the number of threads increases, and so in parallel portions of code a smaller volume of work is completed by each thread. This results in a shorter run time for the program, and the resulting speedup is expressed as

$$S_t = \frac{R(1)}{R(t)} \tag{11}$$

where $R(1)$ is the run time on a single thread, and $R(t)$ is the run time on t threads. This value allows comparison to be made between the run time improvements of different thread numbers. An ideal program would scale linearly, and the program would speed up by a factor of t when run on t threads. This is not possible at large values of t however, as a process cannot be completed entirely in parallel, and the communication speed between threads and memory begins to limit the speedup. It is desirable that the chosen solver exhibits good strong scaling, as faster solution to the neutron flux in a given geometry would allow faster iteration upon system designs.

In 1967 it was proposed by Amdahl that the speedup of software in strong scaling assessments is limited by the fraction of code that can only be completed in serial, this is known as Amdahl's Law [33]. The run time of a process on one thread, known as the serial run time, is split into serial and parallel fractions, s and p. The total run time can not be reduced below the serial fraction of the serial run time, even as the number of threads approaches infinity. Amdahl's Law predicts the speedup of a process for t threads

$$S_t = \frac{1}{s + p/t} \tag{12}$$

given that the fraction of run time in serial and parallel regions is known when run on one thread. Therefore for a process to exhibit the greatest speedup

on N threads, it is desirable that the serial fraction of code is minimal.

1.3.2 Weak scaling

In weak scaling the problem size of a process increases alongside the number of processors. The result is a constant workload per thread, but the solution of a larger problem overall. In the scope of this study, an increased problem size would constitute a finer spatial representation of a geometry. A program that dealt perfectly with weak scaling would show a run time that is constant when the number of threads increases. In reality, the time in a parallel region of a process might remain constant, but the serial portion of the program might take longer, as a larger number of variables must be processed. Furthermore, the time taken to create and schedule threads increases with thread number and is not considered in perfect weak scaling. Weak scaling was first proposed by Gustafson in 1988, an ammendment to Amdahl's Law with the observation that a faster method of calculation would naturally be used to process a larger problem, rather than reducing run time on problems already solvable [34]. Gustafson's Law takes this approach to express the speedup of a program facing an increased problem size as

$$S_t = s + pt \tag{13}$$

where the symbols each have the same meaning as in Amdahl's Law. It is assumed that parallel portions of code speed up linearly with the number of threads N, and that the scalar portions of code remain constant with problem size. Weak scaling analysis observes the effect of thread number t on the weak efficiency

$$E_t = \frac{R(1)}{R(t)} \tag{14}$$

which is identical to Eq (11), but is called weak scaling efficiency due to the varying problem size.

1.3.3 Parallel programming

There are two standard approaches to parallel programming that differ in the memory architecture used: shared memory systems and distributed memory systems. In distributed memory systems processors complete work on a local portion of memory available only to that processor. Communication

between processors must be explicit meaning that implementation is often more complex for the developer, but it can offer a higher level of performance, particularly in weak scaling assessments. Due to the limited time frame of this investigation, parallelism was implemented using a shared memory architecture, which offers a much faster development cycle at the cost of potential performance hurdles, namely race conditions and false sharing.

A race condition is a phenomenon in shared-memory parallel programs in which two or more threads attempt to modify a portion of shared memory at the same time [35]. This can be avoided by synchronizing the action of threads, allowing only one to perform a section of code at once, known as atomics. This prevents bugs and anomalous results being produced by race conditions, but can act as a bottleneck for performance, so are avoided to offer the greatest speed up.

In shared memory parallel architectures, the data being used by each processing unit is loaded into a cache from memory. Suppose a system is using two cores, Core A and Core B, and both cores are executing a process on different pieces of data, but these data points reside on the same cache line within the memory. As they are performing in parallel each loads the same area of memory into their respective caches, seen in Figure 2. Core A might perform its process on its portion of the memory first, tagging this cache line as modified, and in turn, tagging the memory in the Core B cache as invalid. As a result, before Core B can perform the process on its data, Core A's cache line must be forced back to memory, before Core B re-loads the modified memory. This process of offloading and loading memory is time consuming, and can result in a hit to performance, without causing incorrect results, and is known as false sharing [36]. It can be avoided by storing data with buffers so two array points don't reside on the same cache line, or by scheduling processes so operations are not performed simultaneously on data that resides on the same cache line.

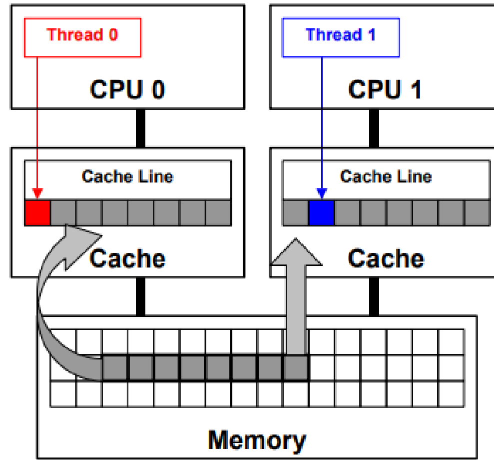


Figure 2: An image demonstrating how data is loaded from memory to cache lines in shared-memory architecture [2].

A final disadvantage of shared memory parallelism is the limit it places on the number of cores available to a process. Where distributed memory systems are used, a program can be run on more than one CPU offering a greater potential maximum number of processors. Shared memory limits the processors used to one CPU, and as a result the maximum number of threads that can be used is limited by the hardware.

1.4 Review of previous work

As discussed in Section 1.1, the parallel performance of a neutron transport solver is an important factor in choosing which method is most appropriate. As a result, investigations into the strong and weak scaling performance of the methods described in Section 1.2 have been performed by others on geometries that are not relevant to nuclear fusion. This allows a comparison to be made between literature values and the data obtained throughout this investigation.

1.4.1 Parallel scaling of TRRM

In 2018 a full scale TRRM solver known as ARRC (A Random Ray Code) was used to simulate a number of benchmark fission eigenvalue problems [3]. Within this analysis, the strong scaling of the Random Ray method was

observed by recording ARRC runtime on up to 44 physical cores, and up to 88 threads via hyperthreading. This research was performed on a dual socket Intel Broadwell-EP Xeon E5-2699 v4 node, and parallelism in ARRC was shared memory. The resulting plot of strong scaling can be seen in Figure 3.

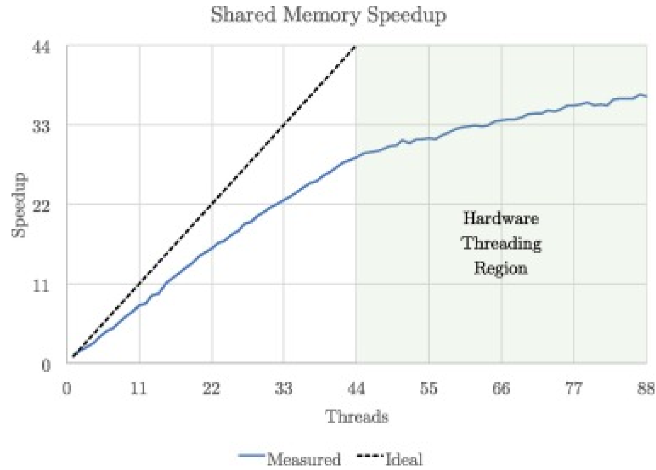
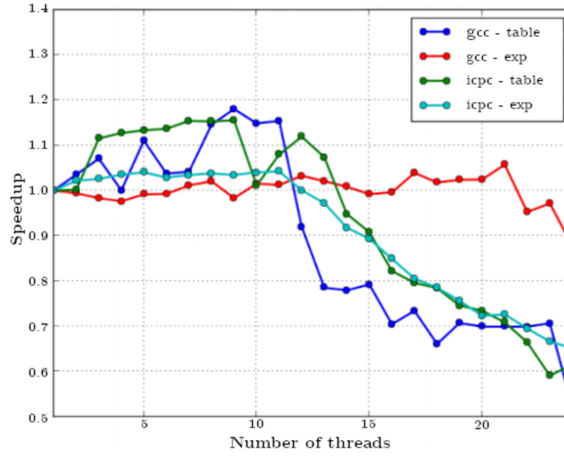


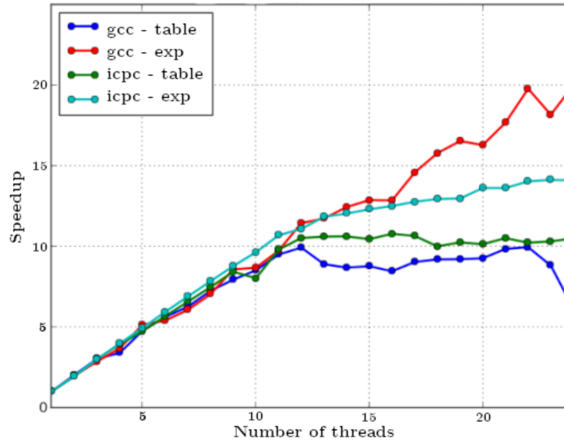
Figure 3: A plot of speedup against thread count for ARRC. Adapted from [3].

This shows a maximum speedup achieved of 37 on all 88 threads, but the speedup curve appears not to have plateaued, and so higher speedup is likely on a larger number of threads.

As explained in Section 1.2.1, the random ray method is based upon the method of characteristics, and as a result is likely to largely mimic its scalability before memory access becomes the rate limiting factor on a large number of cores. A 2016 paper investigated the strong and weak scaling of an MoC solver known as OpenMOC, for a number of different compilers on two Intel i7/E5-2620 processors, each with 12 physical cores providing a maximum 24 hardware threads [4]. The results of this investigation can be seen in Figure 4.



(a) A plot of weak scaling efficiency of OpenMOC against number of threads.



(b) A plot of strong scaling speedup of OpenMOC against number of threads.

Figure 4: Weak and strong scaling results of OpenMOC adapted from [4].

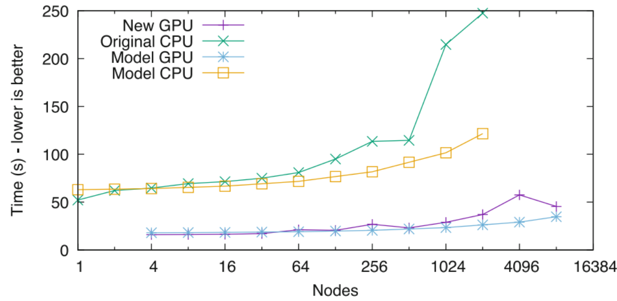
In these plots table and exp refer to the method of exponential calculation being used to evaluate the attenuation term in the MoC solution. Table is the use of tabular interpolation, and exp is the use of the compilers inbuilt exponential function. These results were obtained by simulating the same eigenvalue fission problem. This might affect scalability, as an eigenvalue calculation requires an addition step to calculate the fission source in each FSR. Furthermore, TRRM is expected to have a similar similar scalability

than MoC. This is because the functional difference between the two is the generation of new rays every iteration, as opposed to the reading of this information from memory.

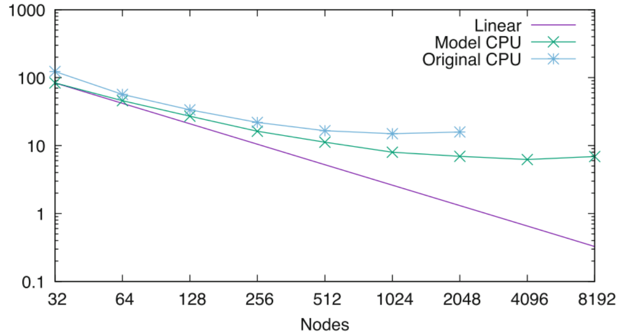
Two key observations can be made based upon these results. Firstly, the compiler used can have a sizeable effect on the scaling properties of a program. Secondly, an MoC solver can achieve close to linear strong scaling at less than 24 threads, and an almost constant run time can be achieved in weak scaling.

1.4.2 Parallel scaling of S_N discrete ordinates

SNAP is a proxy application from Los Alamos National Laboratory based on the discrete ordinates transport code PARTISN. In 2016 work was completed that assessed the scalability of this mini-app, and the strong and weak scaling results can be seen in Figure 5 [5].



(a) A plot of run time in weak scaling tests of discrete ordinates against number of nodes.



(b) A plot of run time in strong scaling tests of discrete ordinates against number of nodes.

Figure 5: Weak and strong scaling results of discrete ordinates adapted from [5].

This data was recorded on Titan, a Cray XK7 supercomputer that uses AMD Opteron 6274 CPU's. MPI was used to implement distributed memory parallel processing and the problem being solved was an eigenvalue problem that used sub domains representing pins in a fission reactor. The results suggest that it is possible to achieve near linear scaling with discrete ordinates up to 256 nodes, and speedup is limited to around 11x at 4096 nodes. Weak scaling is near ideal up to around 64 nodes, before efficiency drops significantly as node number increases. This is explained in the research as being due to network considerations on Titan, limitations of communication speed between nodes.

2 Code modifications

The aim of this investigation was to assess both the scalabilty and correctness of the random ray and discrete ordinates methods, through both an analysis of strong and weak scaling, and the comparison of flux solutions in a number of fusion relevant geometries. Mini-apps are applications with smaller code bases then their full scale equivalents, and achieve smaller code base by sacrificing a number of features. This analysis was completed using mini-app implementations of TRRM and discrete ordinates, as this allowed the most flexibility in the development of code specifically for this investigation.

2.1 Minray

The random ray method mini app that was investigated is known as Minray [16]. Minray has a number of missing features that are required for correctness assessment in fusion geometries, and that affect the scalability of the code. Therefore the mini app was modified to solve a variety of fixed source fusion problems in parallel.

2.1.1 Fusion geometry loading

Primarily this investigation focuses on the use of TRRM for fusion relevant geometries. As a result, the most vital changes to Minray were how geometries were loaded, and how the source value in an FSR was determined. Minray is hard coded to solve a benchmark fission problem known as C5G7, but loads the distribution of materials, as well as the interaction cross section data for materials from external text files. A change in the naming system of said external files from 'moderator_scatter.txt' to 'M1scat.txt' meant that cross section data could be loaded recursively for an arbitrary number of materials.

By default Minray solves eigenvalue problems in which source calculation required the scalar flux dependent fission source in each FSR to be evaluated. Source calculation in Minray was modified to be fixed source using (5). This is done using a fixed source distribution file, similar to the material distribution file, that describes if a given FSR had a neutron source, for example if positioned within plasma.

A limit of the integrating factor solution to angular flux across an FSR seen in Eq (4), is that void regions with a transport cross section of 0 would

exhibit no attenuation. As a result, the flux solution in all void regions would be null, meaning the solution would ignore all streaming effects, and comparisons to other methods would serve no purpose. To avoid this issue, void regions were provided transport cross sections of $1 \times 10^{-10}\text{cm}^{-1}$ and same group scattering cross sections (i.e. $\Sigma^s_{g \to g}$) of $1 \times 10^{-10}\text{cm}^{-1}$. It is also important to note that Minray uses a compilers exponential functions when evaluating this attenuation term.

2.1.2 Shannon entropy

As a result of the stochastic nature of the random ray method, the calculation of scalar flux in each FSR possesses an element of random noise. To resolve this, stochastic methods such as Monte Carlo take an average of results, and provide a statistical error in calculated values. In Monte Carlo fixed source simulations, the source does not need to converge before averaging begins. Conversely, a disadvantage of solving fixed source problems with TRRM is that the source distribution must converge before averaging is started, to reduce the error in calculated values.

The Shannon entropy of the problem was monitored to determine if the scattering source had converged. To achieve this a moving window of size W is assessed. The Shannon entropy of the most recent W iterations is stored, and the source is deemed to have converged once the mean Shannon entropy of the second half of the window is within a standard deviation of the first half of the window. This is done to mimic the method Shannon entropy convergence detection used in ARRC [3].

The addition of source convergence detection through Shannon entropy calculation allows simulations to be run without identifying the number of inactive iterations beforehand. This is necessary, as the number of iterations before the source converges varies with the geometry being simulated, so automatic detection enables a larger degree of flexibility to the code. It also means that the problem being simulated might affect the scalability of the method. When the source is not converged, Shannon entropy is calculated and the mean scalar flux is not, whereas once converged, mean scalar flux is calculated and Shannon entropy is not. The difference in scalability of each of these processes and the number of iterations before and after source convergence would affect the scalability of the method as a whole.

2.1.3 Source averaged relative error

Once the source distribution of the system has converged, and the average scalar flux in each FSR is being tallied, the program must know when to terminate. Without modification, Minray requires the input of the number of active cycles to be completed by the program. To remove this, Minray was modified to include the calculation of source average relative error (SARE)

$$\text{SARE} = \frac{1}{N} \sum_{n}^{N} \frac{\sigma_n}{\phi_n} \tag{15}$$

where ϕ_n is the mean scalar flux in FSR n and σ_n is the standard deviation in the mean scalar flux of FSR n. When the SARE has dropped below a threshold set by the user, the scalar flux is deemed to have converged and the program terminates.

The result of such an approach can be seen in Figure 6. The value of SARE drops following the central limit theorem before meeting the threshold set by the user. SARE can jump on iterations where the randomly generated ray quadrature misses all FSR's with a fusion source, resulting in an anomalous flux calculation and an increase in the relative error of that iteration.

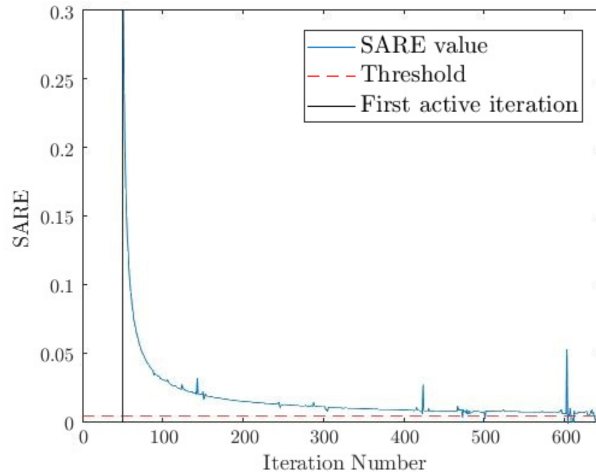


Figure 6: A plot of SARE with iteration number before meeting the threshold in Minray.

2.1.4 Parallel implementation

Minray is written in C, and parallel processes were implemented using OpenMP, a shared-memory parallel programming API. As parallelism through OpenMP is shared-memory, it allows for ease of code production, but can lead to performance issues outlined previously.

There is also a run-time overhead associated with the generation of new threads, and so if the perceived benefit from performing a process in parallel is smaller than this overhead, it is not beneficial overall to make the process parallel. This is of particular importance for smaller portions of the Minray algorithm, where phenomena like false sharing are likely to hinder performance when looping over FSRs and groups to update source terms or normalise calculated scalar flux. OpenMP directives such as atomics could prevent false sharing, but at the cost of a guaranteed performance bottleneck. As a result, for assessment in this investigation, all parallel portions of code in Minray were within the transport sweep: ray generation, ray tracing and flux attenuation. This also excludes mean and Shannon entropy calculation, so any potential effect of problem geometry on scalability discussed in Section 2.1.2 is removed.

2.2 Kripke

For comparison with Minray, a mini-app that uses the discrete ordinates method, Kripke, was used [37]. Kripke offered some key benefits. It was designed for investigation into the effect of how variables are organised on thread level parallelism and the implications on performance. Although this effect is outside the scope of this investigation, as OpenMP is already fully utilised in the Kripke source code it is ideal for the assessment of the strong and weak scaling properties of the discrete ordinates method. A few changes were required however for Kripke to be used to compare the correctness of the method.

2.2.1 Adding I/O capabilities

Originally Kripke is hard coded to solve the fixed source 3D Kobayashi radiation benchmark, problem 3i [38]. This benchmark contains a fixed source region, and so there were no necessary alterations to the power loop of Kripke. All changes made were for the loading of materials and cross sections, and

the output of flux solutions.

Since Minray is a 2D solver, the decision was made not to overhaul Kripke to also be a 2D solver, but instead to extrude 2 dimensional problems in the z-axis. Such an overhaul would entail a complete change to the transport sweep of Kripke. When a problem is extruded far enough in the z direction, the flux solutions of the most central 2D x-y plane mimic those of a 2D solver. The scaling of the solver also remains comparable with Minray. The strong scaling of a program does not depend on the problem size, and in weak scaling, doubling the number of elements in the x-y plane whilst leaving the number of elements in z constant still results in a doubling of total problem size. This allows the same 2D material distribution generated and used by Minray to be imported in Kripke. As a result, material IDs and interaction cross section files can also be reused, meaning the loading of Cartesian geometries into Kripke could be completed using similar functions to Minray, and involved only a conversion of syntax from C to C++.

In order to compare the correctness of the discrete ordinates method to TRRM, it was also necessary that the final flux solutions of Kripke were output to a text file. Upon each iteration, the scalar flux of the problem is calculated by Kripke and the relative change in flux from the previous iteration is determined. Once the relative change in neutron population of a problem has dropped below a threshold set by the user, the problem is deemed to have converged. At this stage the scalar flux in each spatial region is calculated by integrating angular flux over angles, and the values of the central most x-y plane are output to a text file.

3 Methodology

3.1 Geometry and cross section generation

The assessment of correctness in this investigation was completed on a number of geometries chosen to highlight desirable characteristics of a fusion neutron transport solver. Each of these geometries were run on both Kripke and Minray to compare the solutions of the two solvers, and compared to a reference solution calculated using Serpent, a popular Monte Carlo solver [1].

The first geometry used was a simple Be-9 shell, surrounded by vacuum, with a central point source. This was designed to assess how the solver dealt

with down scattering. This is of relevance to thermal tritium breeding designs, in which almost all tritium breeding is achieved by low energy neutrons [39]. A two group energy discretisation was used shown in Table 1.

Group	**Lower bound (MeV)**	**Upper bound (MeV)**
Slow	1.0×10^{-11}	1.0
Fast	1.0	20

Table 1: Energy boundaries for the two group energy discretisation used throughout correctness assessments.

The central source produced neutrons exclusively in the fast region of a two group spectrum, at 14 MeV. As a result, all neutrons present in the slow portion of the spectrum must be produced by down scattering in the beryllium shell. A plot of the geometry can be seen in Figure 7. This geometry allows for simple comparison of radial distributions between solvers, and a small number of input cross sections to be generated as there is only one region that is not vacuum.

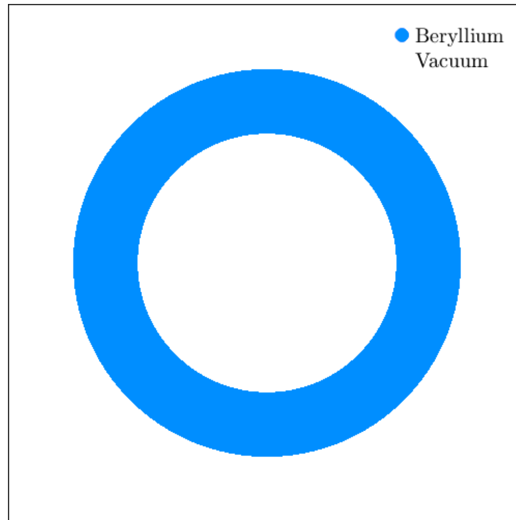


Figure 7: An example of the beryllium shell geometry generated using Serpent.

The second geometry used in this investigation is a modification of the simple beryllium shell. The shell thickness was increased, and a tunnel was

added through the sphere as seen in Figure 8. The goal of using this geometry is to investigate how both the random ray method and discrete ordinates deal with neutron streaming through the tunnels. As this investigation is using 2D Cartesian material distributions, this geometry allows simple comparison of the radial fast flux distributions through the tunnel calculated by Minray and Kripke with the reference solution from Serpent.

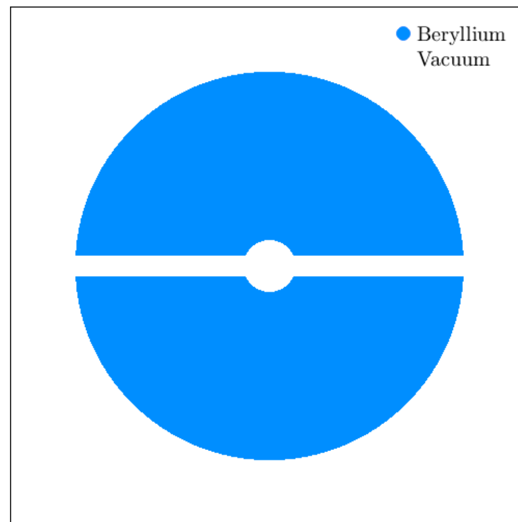


Figure 8: An example of the beryllium tunnel geometry generated using SERPENT.

The final geometry assessed in this investigation is known as KANT, a fusion benchmark used to investigate the treatment of neutron streaming by a given solver [40]. An example of KANT can be seen in Figure 9 to contain a number of materials, designed to emphasise streaming in the system.

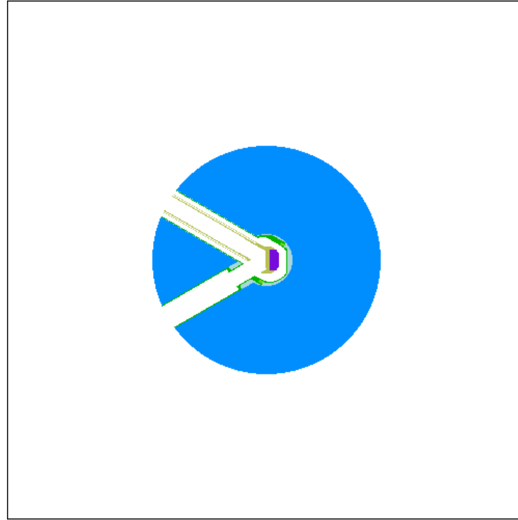
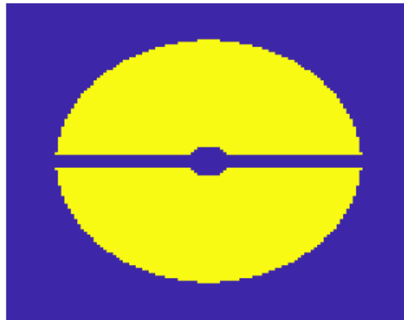


Figure 9: An example of the KANT benchmark geometry provided by CCFE.

Both the beryllium sphere and tunnel geometries were created using the CSG capabilities of Serpent to produce a .png of the geometry, which was later converted to a 2D material ID file with the use of a simple Python script. This script accepts an image of a geometry, and reads the RGB values of pixels. Subsequently numbering the first occurrence of a unique colour in the image, allowing the user to edit a file that assigns each colour in the image to a material ID. A separate script uses this conversion to produce a field of material IDs from the image, and offers the ability to downscale an image to reduce the number of FSRs present in the material distribution file, which is vital for weak scaling assessment. Examples of all three geometries, scaled down to 125x125 material regions can be seen in Figure 10.

(a) Beryllium shell



(b) Beryllium tunnel

(c) KANT

Figure 10: Down scaled material distributions of the three geometries.

These three geometries were each simulated with a central point source

in Minray, Kripke and Serpent. To this end, interaction cross sections were generated using Serpent for use in the other two solvers. This provided accurate cross section data, due to the energetic fidelity afforded by Monte Carlo. Both the beryllium shell and beryllium tunnel geometries were produced in Serpent, and so reference calculations were completed on full resolution geometries. The KANT geometry however was imported to Serpent in its down scaled form, by generating a grid of CSG surfaces to mimic the Cartesian geometry.

3.2 Assessing scaling

In order to evaluate the parallel performance of Minray, its strong and weak scaling properties were compared to those of Kripke. To assess the strong scaling of the programs, simulations of the same 250x250 cartesian cell geometry were performed on a range of different core numbers. This was achieved through the use of the Cambridge Service for Data Driven Discovery (CSD3). Access to the Intel Zion Phi KNL node allowed each solver to be run using up to 64 physical cores on a single node, and up to 256 threads via hyper threading. Each solver was run with 48 different thread counts, and for each thread count the simulation was run 10 times. This allowed a mean value to be obtained, and a statistical error in each result to be calculated.

To assess the weak scaling properties of the mini apps, the thread number used by both Minray and Kripke were increased in orders of 2, and problem size was increased proportionately. To achieve the doubling of a two dimensional square problem, in each dimension the number of cells was increased by a factor of approximately $\sqrt{2}$. The thread numbers and corresponding problem dimensions used in these weak scaling experiments can be seen in Table 2.

No. of threads	Problem dimensions
1	177×177
2	250×250
4	354×354
8	500×500
16	707×707
32	1000×1000

Table 2: A table documenting the problem dimensions simulated for each thread number in weak scaling assessment.

Inherently the problem size in each dimension must be an integer, and this prevents a perfect $\sqrt{2}$ scaling in each dimension. The problem sizes used in this experiment were within 1% of the exact value, and so are not expected to be a source of major error in the weak scaling assessment.

The compiler used to produce the Minray executable run on the CSD3 nodes could also have a measurable impact on the scalability of the program. For this reason, Minray was compiled both with Intel's ICC compiler, and with GNU's GCC compiler. The previously outlined weak and strong scaling assessment was performed for each, to investigate the effect that the compiler has on scalability. This comparison was not possible on Kripke, due to a bug encountered with one of Kripke's dependencies during compilation with Intel's C++ compiler ICPC.

4 Results and discussion

4.1 Scalability

All scalability assessments were performed on the beryllium shell geometry seen in Figure 7. As expressed in Section 2.1.4, dependency of scaling on problem geometry was removed, and so this choice was arbitrary.

4.1.1 Strong scaling

The speedup of Minray in strong scaling experiments can be seen in Figure 11. This data was collected with Minray compiled using GCC 5.4.0 and OpenMPI 1.10.7, and run on Intel Xeon Phi up to a maximum of 256 threads. Mean serial run time on this platform was 1086.65 s, and a maximum speedup

was achieved on 256 threads of 54.49 ± 0.82 at a run time of 20.17 s. The fit of Amdahl's law seen in Figure 11 was obtained through Chi-squared minimisation, and suggests that a fraction $p = 0.985$ of the run-time on 1 thread benefits from being run in parallel. This corresponds to a theoretical maximum speedup of ≈ 68.03 when run on an infinite number of cores, ignoring potential bottlenecks such as memory access.

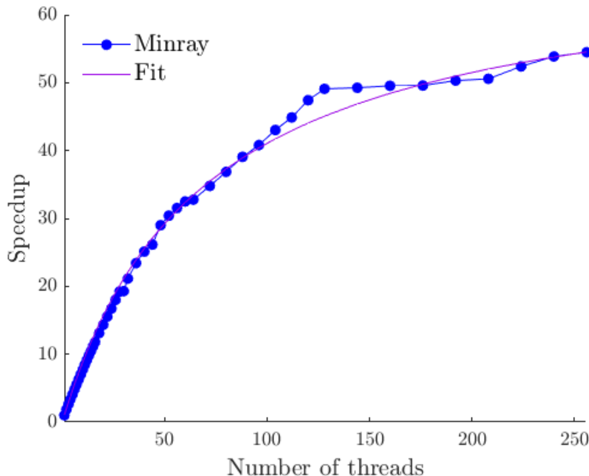


Figure 11: A plot of parallel speedup against thread number in strong scaling assessments of Minray on a KNL CSD3 node and a best fit of Amdahl's law.

These results correspond with calculations in literature. The previously achieved a speed up of 37 when TRRM is run on 88 threads solving an eigenvalue problem is 5.3% smaller than the 39.07 ± 0.26 calculated in this experiment on an equivalent number of threads solving a fixed source problem [3]. This suggests that fixed source TRRM is more scalable than the eigenvalue equivalent. This could be predicted, as fixed source simulations remove the need to compute a fission source.

A plot of the results in Figure 11 over a smaller range of threads can be seen in Figure 12 for a simpler comparison to the data in Figure 4b and Figure **??**. It is clear that Minray performs similarly to traditional MoC at less than 24 threads, which is expected due to the functional similarity between the two methods. It is expected that at higher thread counts, MoC would reach a lower speedup than TRRM, as memory bottlenecks would

dominate as a result of the ray information being read from memory.

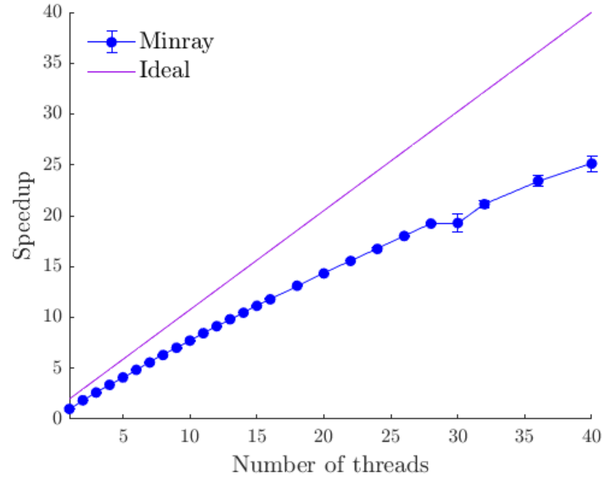


Figure 12: A reduced plot of parallel speedup against thread number in strong scaling assessments of Minray on a KNL CSD3 node alongside an example of ideal strong scaling.

The results of strong scaling analysis on Kripke can be seen in Figure 13. Kripke was also compiled with GCC 5.4.0 and OpenMPI 1.10.7 and run on a maximum of 256 Intel Xeon Phi threads. The mean run time on 1 thread was 116.18 ± 0.63 s, and the maximum speedup of 7.26 ± 0.39 was achieved on the maximum 256 threads with a mean run time of 16.00 ± 0.85 s. This fit of Amdahl's law predicts a maximum theoretical speedup of 7.58, with an estimated serial fraction of code $s = 0.132$.

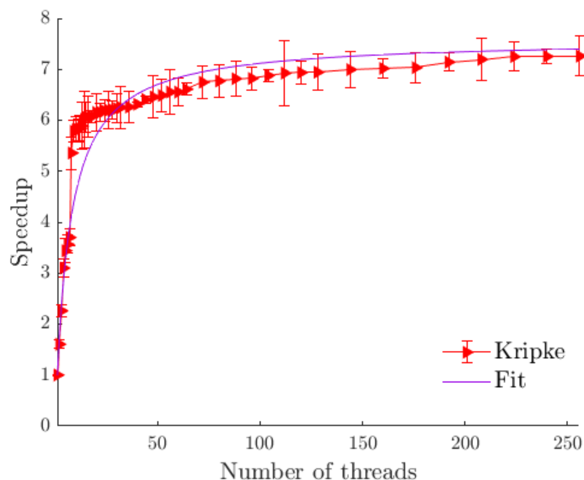


Figure 13: A plot of parallel speedup against thread number in strong scaling assessments of Kripke on a KNL CSD3 node and a best fit of Amdahl's law.

A comparison between the Kripke results from this experiment and other assessments of discrete ordinates reveals that the strong scaling of the method is relatively consistent. The run times presented in Figure 5b show that the maximum speed up of SNAP was ≈ 15, and that the speed up at 256 nodes was ≈ 10. This is greater than both the speed up on an equivalent number of threads, and the maximum predicted by the fit of Amdahl's law.

The likely cause of the early plateau of Kripke's strong scaling results is the dominance of a memory bottleneck. At fewer than 8 threads, the run time of Kripke is limited by the time taken for each thread to complete its work. As the thread number increases, the work per thread decreases, so too does the time each thread is working for. This results in a reduced run time, and a speed up with thread number. However, the time taken to access memory is constant, so if the computation time of each thread falls under this value, threads must wait for memory to be accessed before work can be completed, in essence imposing a maximum speed up on the program. This seems to occur in Kripke at 8 threads. This is likely also the reason that Kripke does not reach the speed up discussed in Figure 5b. Programs made parallel with distributed memory architectures are less prone to memory bottlenecks as each thread operates on its own memory, so the MPI implemented parallelism used in SNAP is able to achieve larger speedup.

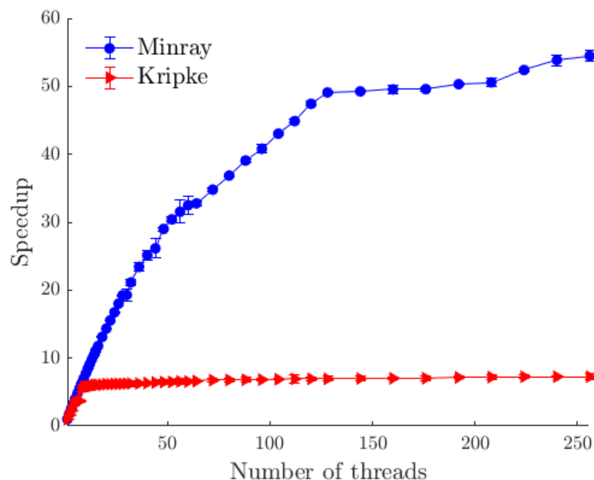


Figure 14: A plot of strong scaling results for both Kripke and Minray against thread number.

A plot for comparison of the strong scaling of Minray and Kripke can be seen in Figure 14. It is clear that Minray offers a larger maximum speedup than Kripke, although the two exhibit similar speed up at fewer than 8 threads. Whilst the speed up frames Minray as an obvious choice for performance, the run time differences must be considered, but unfortunately the run times of Kripke and Minray cannot be compared. This is because Kripke is a 3D solver being used to produce 2D solutions, and so is iterating upon a larger number of variables than would be necessary for a true 2D solution, that would be comparable in run time to Minray.

4.1.2 Weak scaling

The weak scaling results of both Minray and Kripke can be seen in Figure 15. Both solvers exhibit less than ideal scaling, although it is clear that Kripke outperforms Minray at all thread numbers. Both Minray and Kripke are outperformed by all of the solver methods highlighted in Section 1.4. There are a number of potential causes for the poor weak scaling of Minray. First is load imbalance, where the workload of a process is not evenly distributed amongst threads, but instead some threads are left with no work, and not contributing to the speed up of computation. This is possible in the way that parallelism is currently implemented in Minray. When a ray is randomly

generated, the number of FSRs that it intersects varies, depending on the angle of the ray sampled. Given that the number of segments in a ray varies, so too does the number of flux attenuation computations that must be performed, resulting in a computational workload that changes from ray to ray. Since task scheduling is static by default in OpenMP, i.e each thread is assigned the same number of rays to compute, it is possible that each iteration threads can be left with no work, having computed a number of rays with fewer than average segments. This could be solved by using dynamic task scheduling improving load balancing, at the cost of a larger time overhead associated with thread allocation. Further research into the impact of load balancing on Minray run times would help determine if dynamic scheduling would accelerate run times or improve weak scaling.

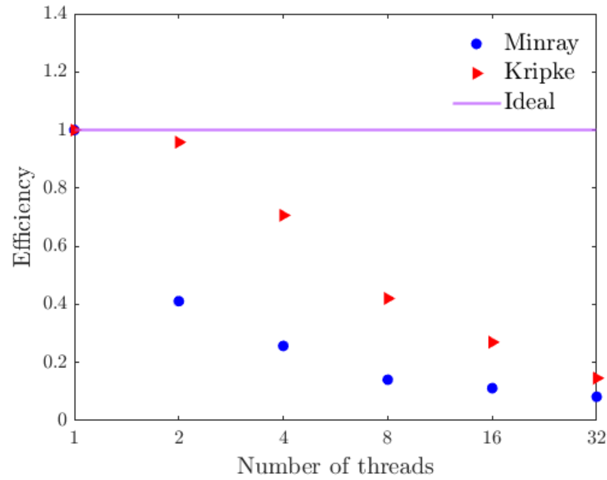


Figure 15: A plot of the weak efficiency of Minray and Kripke against number of threads run on a CSD3 Skylake node.

A second potential cause is the nature of the serial fraction of code in Minray. As explained in Section 2.1.4, portions of code in Minray that represent a smaller computational workload, that iterate over FSRs rather than rays were not made parallel. The workload of these serial code portions increases with cell number, and take a longer time to run as a result. It is a fundamental assumption of Gustafson's law, and the ideal of weak scaling, that the serial portion of code has a constant run time regardless of problem size, but in Minray this is not the case. Therefore, it is most likely that

the poor weak scaling shown in Figure 15 is not reflective of the random ray method, but of this particular implementation of TRRM. Furthermore an investigation into the effect of distributed and shared memory architectures on the weak scaling of TRRM would be beneficial. It is expected that a distributed memory application in which each thread is working on its own data set would offer better weak scaling, but the extent of these advantages is not known.

The poor weak scaling of Kripke could be explained in part by the occurrence of false sharing, as explained in Section 1.3.3. Implementation of OpenMP in Kripke does not make use of atomics, and so false sharing is likely. However, as the margin by which the efficiency of Kripke outperforms Minray decreases with the number of threads, an alternative is more likely. Memory bandwidth is the maximum rate at which data can be transferred from memory. If data is processed by threads at a rate greater than the memory bandwidth, run time is limited by the time taken for data to be transferred to threads. In weak scaling, as the problem size increases, so does the volume of data that must be read from memory. This results in a greater run time, and a lower weak scaling efficiency.

4.1.3 Compiler dependency

It can be seen in Figure 4 that the compiler choice can have an effect on the strong and weak scaling of an application. To this end, the strong and weak scaling of Minray when compiled by GCC 5.4.0 and ICC Version 2021.1 can be seen in Figure 16 and Figure 17 respectively.

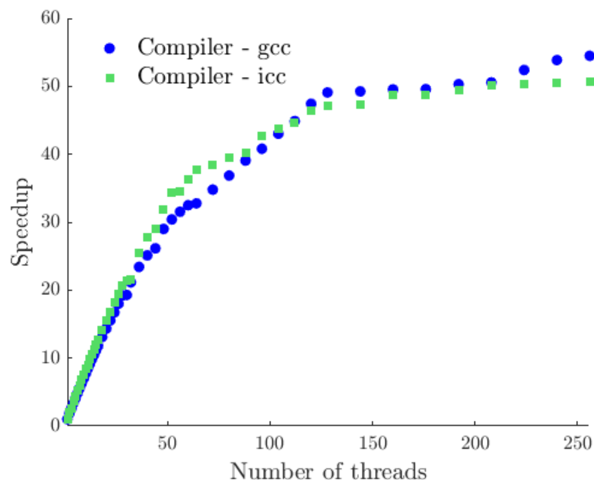


Figure 16: A plot of the strong scaling speedup of Minray against number of threads when compiled with both GCC 5.4.0 and ICC Version 2021.1.

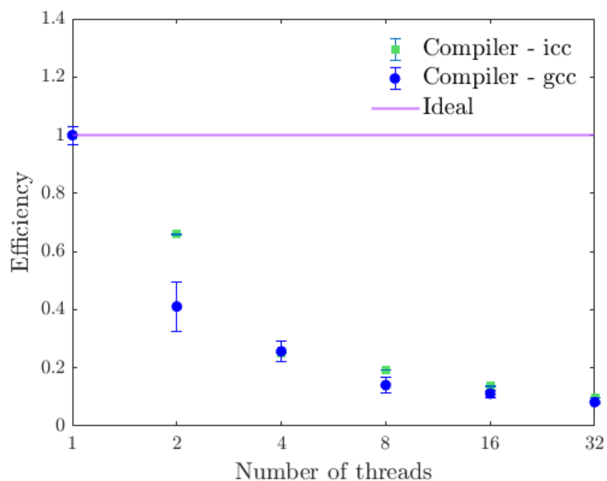


Figure 17: A plot of the weak efficiency of Minray against number of threads when compiled with both GCC 5.4.0 and ICC Version 2021.1.

When compiled by Intel's ICC compiler, a fit of Amdahl's law to strong scaling results suggests that the serial portion of code $s = 0.0147$. This is the same value as that calculated for the GCC compiled results, and summarises

that the compiler had no effect on the scalability of Minray in this case. It did have an effect on the mean run time however. The mean serial run time when compiled by ICC was 867.94 $\pm$ 0.83 s. This means Intel's compiler offers a 20.1% reduction in run time when compared to GNU, equivalent to a speedup of 1.25. An investigation into the effect of compiler and compiler flag choice in the use of Minray prior to real world use could therefore provide a reduction to run time on larger simulations than the 250x250 FSR run in this experiment.

4.2 Correctness

Minray returns a scalar flux in source regions several orders of magnitude larger than both other solvers and even the fixed source magnitude used by the code. The exact cause for this is unknown, although this effect is not present when source regions exist in materials that are not vacuum. Because of this, the error likely arises from the treatment of void materials and the small cross sections associated with them.

To allow comparison of distributions to the other solvers, the flux in the source region of Minray was 'chopped', and flux distributions of all solvers was divided by the maximum flux recorded by each solver.

4.2.1 Beryllium shell

Plots of the normalised slow flux solutions of the beryllium shell geometry in Kripke and Minray can be seen in Figure 18 and Figure 19 respectively. The reference solution generated using Serpent can be seen in Figure 20.

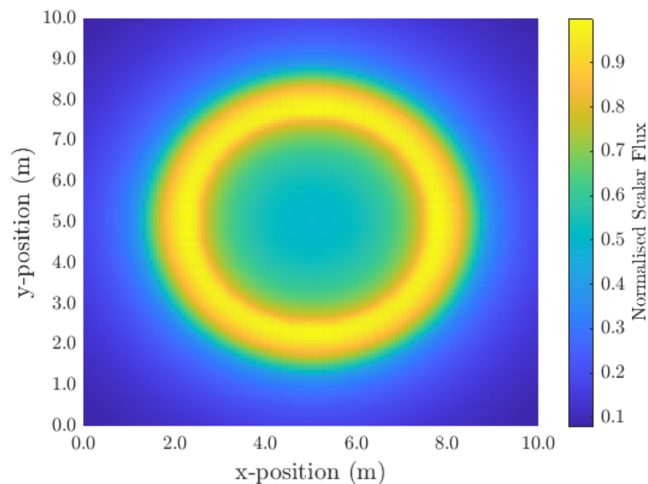


Figure 18: A pseudo color plot of the normalised slow flux calculated by Kripke in the beryllium shell geometry.

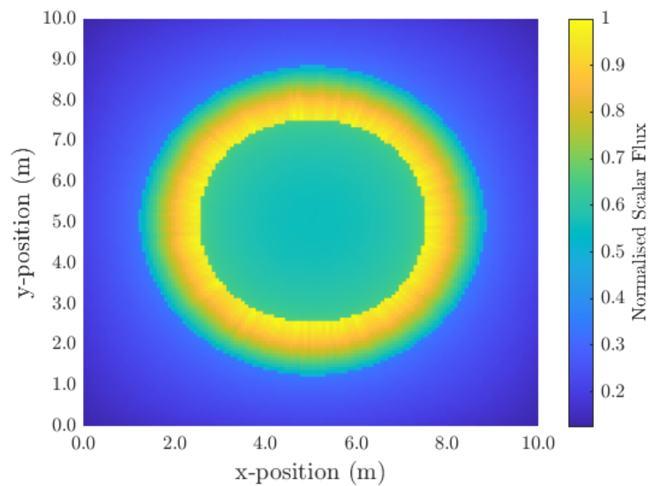


Figure 19: A pseudo color plot of the normalised slow flux calculated by Minray in the beryllium shell geometry.

A key observation of the Minray flux is its apparent discontinuous nature at material boundaries. This is non physical, and can be particularly relevant in situations where neutron flux at the surface of a material is important such

as in radiation damage calculations. Fast neutrons would be more pertinent to these calculations, and so this effect will be monitored in the fast neutron flux plots of Section 4.2.2.

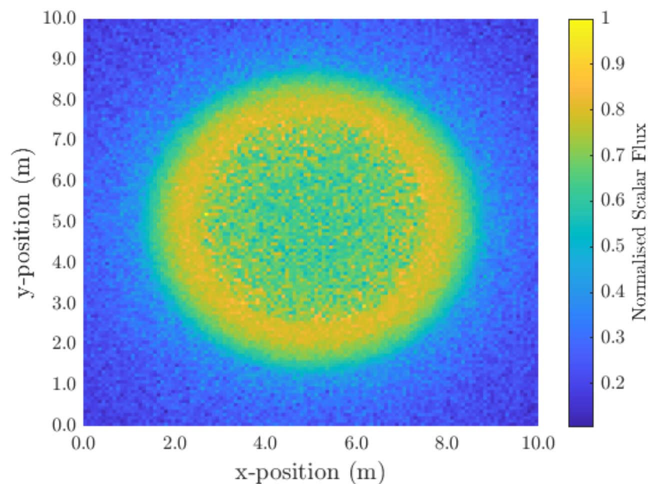


Figure 20: A pseudo color plot of the normalised slow flux reference solution calculated by Serpent in the beryllium shell geometry.

Qualitatively, all three flux solutions show the production of slow flux within the beryllium, and exhibit the scattering of slow flux into the central void, resulting in a higher flux in the central void region than outside the shell. Noise is clearly present in Figure 20, despite the simulation of 100000000 neutron histories.

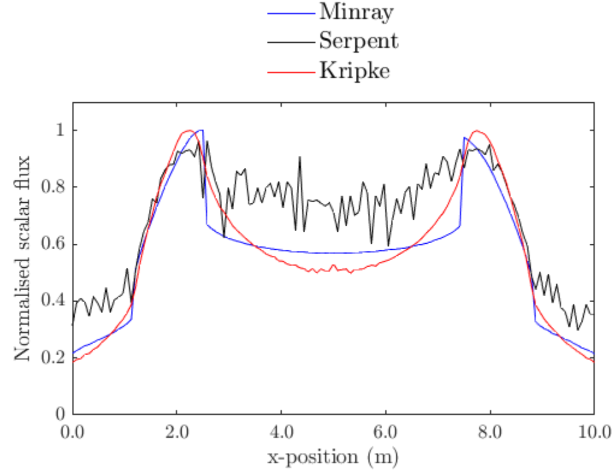


Figure 21: A plot of the normalised radial flux distribution for all three methods in the beryllium shell geometry.

A plot of the radial flux distribution of slow flux in the beryllium shell geometry for all three methods can be seen in Figure 21. This plot shows that within the shell Minray and Kripke both produce similar flux distributions to the reference solution. This is not the case in void regions however, in which both Minray and Kripke result in lower relative fluxes than the Serpent solution. On average Minray and Kripke were 18.75% and 18.58% smaller than the reference solution respectively. Within the beryllium shell region however, Kripke more closely represented the reference flux solution, with a result 0.45% larger than Serpent on average. Minray however resulted in a slow flux that was on average 3.13% smaller than the reference solution. The advantage of Kripke is in void regions near material boundaries, in which Minray calculates a flux much smaller than the reference solution.

No concrete conclusions can be drawn from this data however, as the statistical uncertainty of Serpent results in this portion of the spectrum is very large. In the Serpent simulation, all neutron lifetimes are produced in the faster portion of the spectrum, and so the number of samples in the slower part of the spectrum is inherently linked to the number of scattering collisions. This introduces a large statistical uncertainty in the flux results of the lower energy group. Future work that intends to assess the down scattering of TRRM and discrete ordinates would have to avoid the low sampling issues encountered in this project, perhaps using variance reduction

techniques.

4.2.2 Beryllium shell - Streaming paths

Pseudo color plots of the normalised fast flux solutions of the beryllium shell geometry in Kripke and Minray can be seen in Figure 22 and Figure 23 respectively. The reference solution generated using Serpent can be seen in Figure 24.

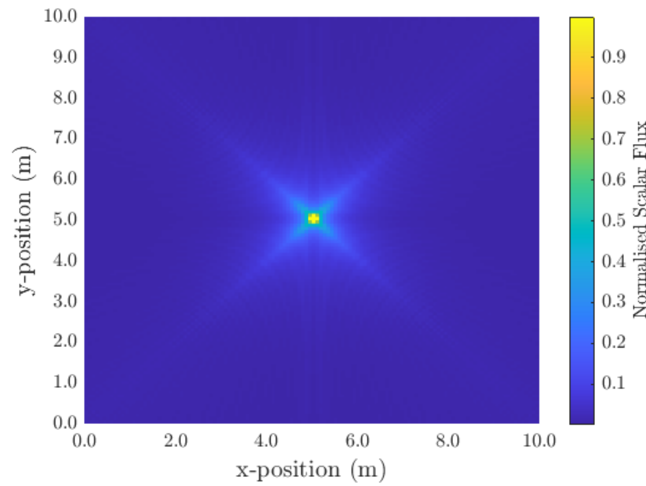


Figure 22: A pseudo color plot of the normalised fast flux calculated by Kripke in the modified beryllium shell geometry with streaming paths.

What is clear from Figure 22 is that the fast flux calculations using Kripke fall victim to the ray effect. This effect was present when a large number and combination of angles was used. This plot was calculated using 4096 total angles, with 64 polar and 64 azimuthal. The cause of the ray effect is explained in Section 1.2.2.

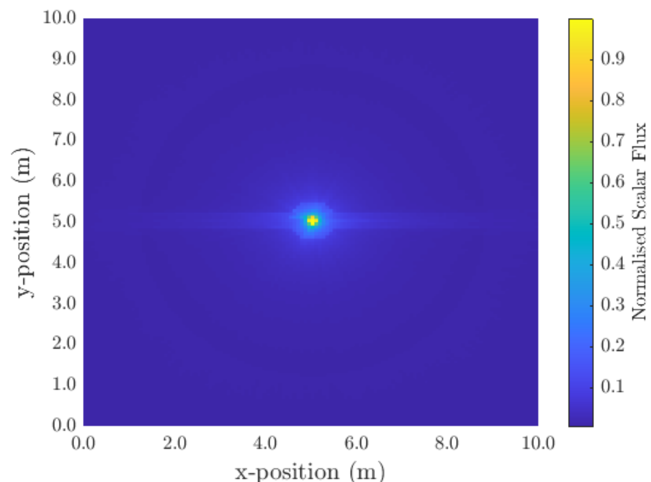


Figure 23: A pseudo color plot of the normalised fast flux calculated by Minray in the modified beryllium shell geometry with streaming paths.

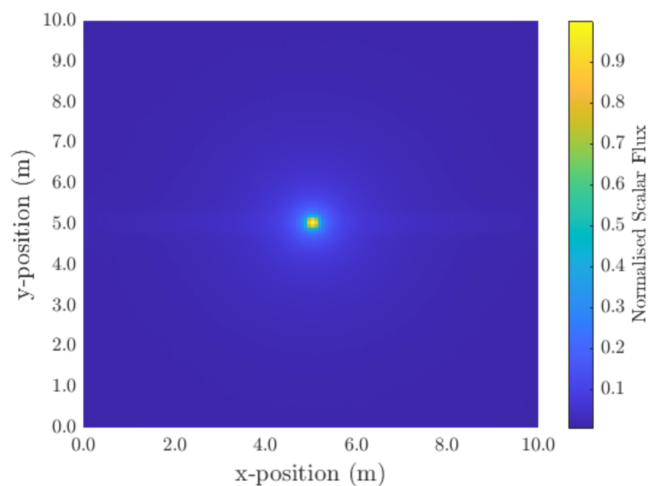


Figure 24: A pseudo color plot of the normalised fast flux reference solution calculated by Serpent in the modified beryllium shell geometry with streaming paths.

Both Figure 23 and Figure 24 display a larger fast flux along the streaming tunnels than in other directions. It is clear that the large volume of stochastic noise in slow serpent results is not present in the fast group. This is because

the number of samples is dependent on the fixed source, not on scattering, avoiding the low sampling phenomenon.

Figure 23 shows that fast flux is lower in all beryllium regions of the geometry. This too appears to be an effect of the discontinuous Minray solution. It is non physical for the fast flux to be lower within the material, closer to the source, than in void regions past this optically thick region.

A plot of the distribution of fast flux along streaming paths in the modified beryllium shell geometry for all three methods can be seen in Figure 25.

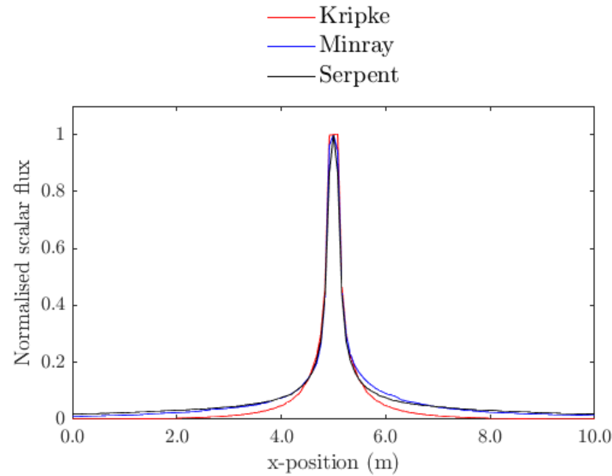


Figure 25: A plot of the normalised radial fast flux distribution for all three methods in the modified beryllium shell geometry with streaming paths.

This plot shows that as expected, Kripke (with ray effects present) underestimates the flux along streaming paths. One of the reasons discrete ordinates solvers are not common in fusion simulation, even without the ray effect, is that they fail to represent neutron streaming. This claim is supported by Figure 25. Minray more closely mimics the reference solution along these tunnels. On average the Minray solution was 12.91% smaller than the Serpent equivalent, and the Kripke solution was 66.03% smaller than the reference solution. As the material is the same in these measurements, this difference likely arises due to the void transport cross sections used by Minray. At shorter distances from the fixed source, more attenuation occurs in Minray, so calculated flux is larger than the reference solution, but drops more quickly with distance resulting in a smaller average flux.

A plot of the absolute error in Minray calculations compared to Serpent can be seen in Figure 26.

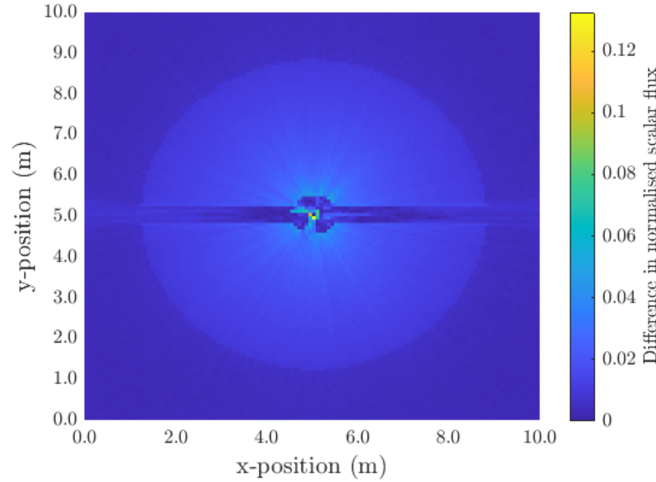


Figure 26: A plot of the absolute error in normalised fast flux solutions calculated by Minray and Serpent in the modified beryllium shell geometry with streaming paths.

This shows that in fact Minray was most accurate within streaming tunnel regions of the geometry. It also highlights a fingering effect in the Minray solution. This might be representative of a ray effect equivalent in TRRM, where too few rays are sampled each iteration, leading to anomalous rays that possessed a larger flux dominating even after a large number of iterations. It is predicted that a larger number of iterations or a larger number of rays per iteration would mitigate this effect at the cost of computational time, as rays that do not pass through the source would be able to 'spread' the flux of these anomalous rays throughout the geometry.

4.2.3 KANT

Plots of the normalised fast flux solutions of the KANT benchmark in Minray can be seen in Figure 28, and an equivalent plot for the reference Serpent solution can be seen in Figure 29. As shown in Section 4.2.2, Kripke experiences ray effects when simulating problems with a fixed source surrounded by void material. This is also the case in the KANT benchmark, as seen in Figure 27

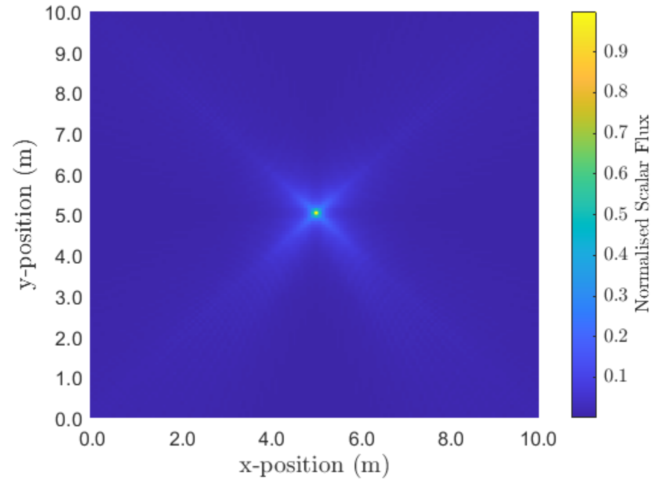


Figure 27: A pseudo color plot of the normalised fast flux calculated by Kripke in the KANT benchmark geometry.

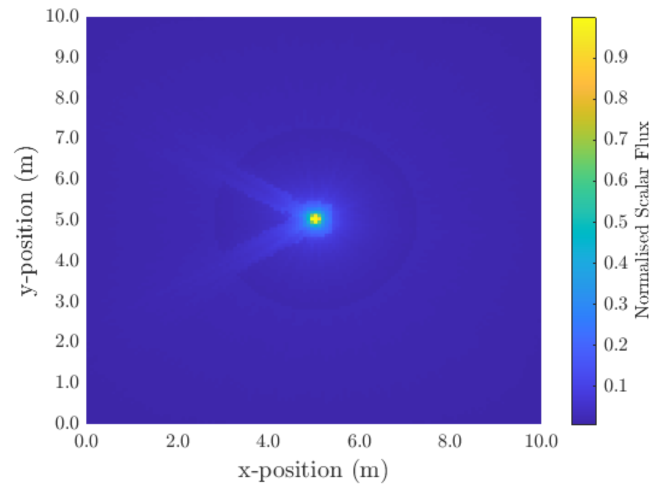


Figure 28: A pseudo color plot of the normalised fast flux calculated by Minray in the KANT benchmark geometry.

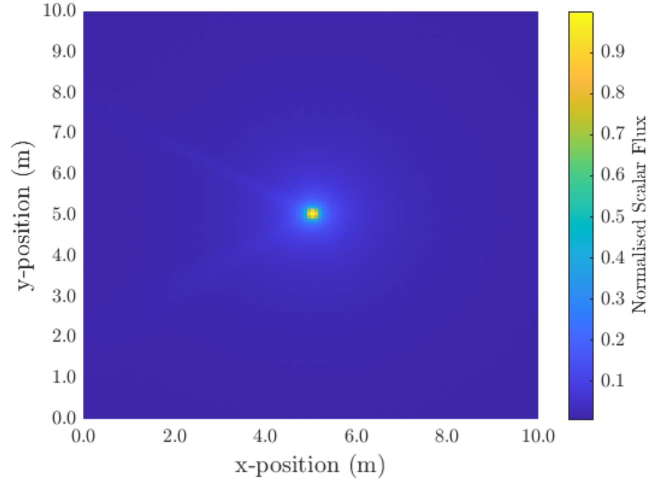


Figure 29: A pseudo color plot of the normalised fast flux reference solution calculated by Serpent in the KANT benchmark geometry.

To better understand how the results of Minray compare to the reference solution, a plots displaying the difference and absolute difference between the Minray and Serpent solutions can be seen in Figure 30 and Figure 31 respectively.

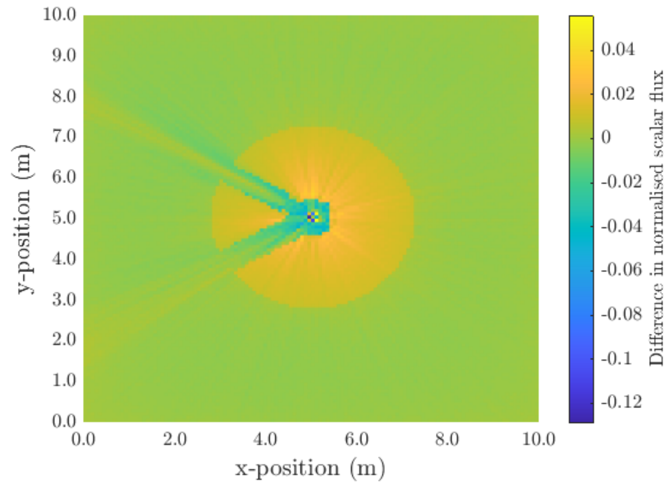


Figure 30: A plot of the error in normalised fast flux solutions calculated by Minray and Serpent in the KANT benchmark geometry.

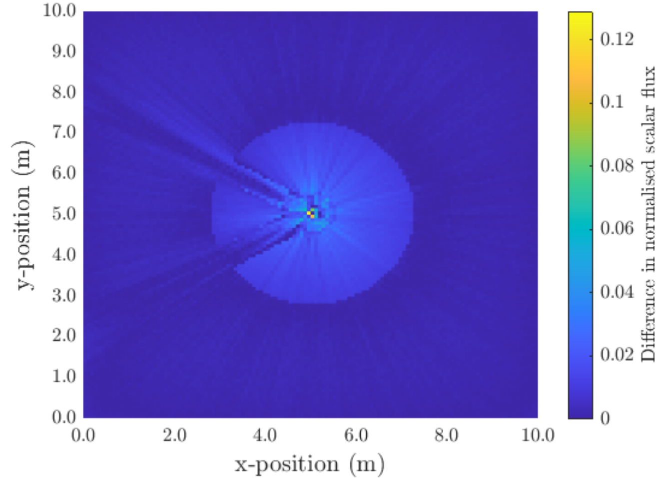


Figure 31: A plot of the absolute error in normalised fast flux solutions calculated by Minray and Serpent in the KANT benchmark geometry.

This difference was calculated by subtracting the normalised flux solution of Minray from that of the reference solution and so a negative (blue) value represents a region in which Minray calculated a larger flux, and a positive (yellow) value represents a region in which Minray calculated a smaller flux. Absolute error represents the magnitude of this error. These plots highlights how in neutron streaming paths closer to the neutron source, Minray calculates a flux solution that is larger than the reference solution. Further from the source along the same neutron streaming paths Minray calculates a smaller neutron flux than the reference solution. This could be an artifact of the $1{\times}10^{-10}$ transport cross section implemented to Minray in void regions. Which would means that more neutron flux is attenuated in void regions close to the source, and as a result neutron flux further from the source is lower. As the transport cross section is so small, a more likely source of this difference is that anisotropic scattering is not currently dealt with by Minray. As a result future experiments that apply techniques to either calculate anisotropic scattering directly in TRRM, or mitigate the effects of the isotropic scattering approximation, would be useful.

Figure 30 further emphasises the 'discontinuity error' encountered by Minray. In optically thick regions of the geometry, Minray calculates a much smaller fast flux than serpent and this effect is not present as soon as the void region is reached. The cause of this error is not known, and so further

investigation is required to understand what is causing, and potentially fix, this error.

5 Conclusions

This objective of this project was to measure the scalability and compare the correctness of the random ray and discrete ordinates methods of neutron transport simulation when applied to fusion relevant geometries. This analysis was completed through the modification of mini app implementations of the methods to operate in parallel on a shared memory architecture, automatically detect convergence and compute a fixed source problem.

The strong and weak scaling properties of the methods were assessed by varying problem size run on up to 256 threads on the CSD3 KNl node. It was found strong scaling experiments that Minray was capable of a maximum speed up of 54.49 ± 0.82 on 256 threads, and Kripke achieved a lower maximum speed up of 7.26 ± 0.39 on 256 threads. The speedup of Kripke was likely lower due to the impact of memory access bottlenecks. The weak scaling of both applications was poor, although Minray was outperformed by Kripke at all thread counts. The poor weak scaling of Minray is likely the result of load imbalance or a serial code portion that increases in run time with problem size. The effect of compiler choice on Minray strong and weak scaling was also assessed. It was found that whilst compiler choice had no effect on scaling, the use of Intel's ICC compiler resulted in a 20.1% reduction in run time when compared to GNU's GCC compiler. The same analysis could not be completed on Kripke due to a bug in a dependency when compiled with ICPC (the Intel C++ compiler). Further analysis on the effect of compiler choice, perhaps on a wider variety of compilers would be beneficial prior to real world Minray use.

Correctness assessments were performed on three geometries designed to test how each solver treats down scattering and neutron streaming in fusion relevant geometries. Each geometry was simulated Serpent, as well as Minray and Kripke, to provide a reference solution for comparison. These simulations used a two group energy discretisation, and all source neutrons were produced in the fast group of the spectrum.

It was found that problems simulated in Minray whose fixed source resided in a void region produced an error. The scalar flux in source regions was several orders of magnitude larger than other solvers. Further development of

Minray for fusion application should constitute treatment of this bug.

Kripke more appeared to more accurately calculate slow neutron flux produced by down scattering within a beryllium shell surrounding a point source, differing from the reference solution by 0.45% on average compared to a mean difference of 3.13% from the Minray solver. No meaningful conclusions can be drawn however, as a large, unforeseen statistical error was present in the reference solution due to the low sampling phenomenon. This geometry did however highlight that Minray produces a discontinuous flux solution at material boundaries, which should also be remedied in future work on Minray.

It was found that Minray outperforms Kripke in the treatment of fast neutron streaming. On average the Minray solution along streaming paths was 12.91% smaller than the reference solution. This because approximations made in the derivation of TRRM require non zero interaction cross sections, and so void regions in Minray were given larger cross sections than their Serpent equivalents. Kripke resulted in mean difference of 66.03% . This is likely a result of the ray effect present in the fast flux calculations, despite the use of 4096 discrete angles.

Future work should focus on the development of Minray to avoid the errors discovered in this investigation, although early signs indicate that the random ray method can simulate fast neutron streaming more accurately than the discrete ordinates method, whilst differing from Monte Carlo solutions. TRRM demonstrated impressive strong scaling on a shared memory architecture, but future work implementing distributed memory parallelism to the method could improve weak scaling of the method.

References

[1] J. Leppänen. Serpent–a continuous-energy monte carlo reactor physics burnup calculation code. *VTT Technical Research Centre of Finland*, 2013.

[2] Intel Software Engineering Team. Intel guide for developing multi-threaded application, 2011.

[3] J. R. Tramm, K. S. Smith, B. Forget, and A. R. Siegel. Arrc: A random ray neutron transport code for nuclear reactor simulation. *Annals of Nuclear Energy*, 2018.

[4] W. Boyd, A. Siegel, S. He, B. Forget, and K. Smith. Parallel performance results for the openmoc neutron transport code on multicore platforms. *The International Journal of High Performance Computing Applications*, 2016.

[5] T. Deakin, S. McIntosh-Smith, and W. Gaudin. Many-core acceleration of a discrete ordinates transport mini-app at extreme scale. In *International Conference on High Performance Computing*, 2016.

[6] J. Knaster, A. Moeslang, and T. Muroga. Materials research for fusion. *Nature Physics*, 2016.

[7] Y. Wu. *Fusion neutronics*. Springer, 2017.

[8] E. E. Lewis and W. F. Miller. Computational methods of neutron transport. 1984.

[9] G. E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 1998.

[10] D. A. Patterson. Future of computer architecture. In *Berkeley EECS Annual Research Symposium (BEARS)*, 2006.

[11] M. Annavaram, E. Grochowski, and J. Shen. Mitigating amdahl's law through epi throttling. In *32nd International Symposium on Computer Architecture (ISCA'05)*, 2005.

[12] F. Kong, Y. Wang, D. R. Gaston, A. D. Lindsay, C. J. Permann, and R. C. Martineau. A scalable multilevel domain decomposition preconditioner with a subspace-based coarsening algorithm for the neutron transport calculations. 2019.

[13] R. JJ. Stamm'ler and M. J. Abbate. *Methods of steady-state reactor physics in nuclear design*. Academic Press London, 1983.

[14] J. R. Askew. A characteristics formulation of the neutron transport equation in complicated geometries. Technical report, United Kingdom Atomic Energy Authority, 1972.

[15] W. Boyd, S. Shaner, L. Li, B. Forget, and K. Smith. The openmoc method of characteristics neutral particle transport code. *Annals of Nuclear Energy*, 2014.

[16] The Random Ray Method for neutral particle transport. *Journal of Computational Physics*.

[17] F. B. Brown. On the use of shannon entropy of the fission distribution for assessing convergence of monte carlo criticality calculations. In *ANS topical meeting on reactor physics (PHYSOR 2006)*, 2006.

[18] M. Nowak, J. Miao, E. Dumonteil, B. Forget, A. Onillon, K. S Smith, and A. Zoia. Monte carlo power iteration: Entropy and spatial correlations. *Annals of Nuclear Energy*, 2016.

[19] S. Chandrasekhar. *Radiative Transfer*. Dover Publications, 1960.

[20] B. G. Carlson. Solution of the transport equation by sn approximations. Technical report, Los Alamos Scientific Lab, 1955.

[21] D. F. Gill. Behavior of the diamond difference and low-order nodal numerical transport methods in the thick diffusion limit for slab geometry. Technical report, Knolls Atomic Power Laboratory (KAPL), 2007.

[22] M. Lahdour, T. El Bardouni, M. Mohammed, and S. El Ouahdani. The discrete ordinate method for angular flux calculations in slab geometry. *Heliyon*, 2019.

[23] M. L. Adams and E. W. Larsen. Fast iterative methods for discrete-ordinates particle transport calculations. *Progress in Nuclear Energy*, 2002.

[24] K. D. Lathrop. Ray effects in discrete ordinates equations. *Nuclear Science and Engineering*, 1968.

[25] J. C. Chai, H. S. Lee, and S. V. Patankar. Ray effect and false scattering in the discrete ordinates method. *Numerical Heat Transfer, Part B Fundamentals*, 1993.

[26] T. Nishimura, K. Tada, H. Yokobori, and A. Sugawara. Development of discrete ordinates sn code in three-dimensional (x, y, z) geometry for shielding design. *Journal of Nuclear Science and Technology*, 1980.

[27] N. Metropolis and S. Ulam. The monte carlo method. *Journal of the American Statistical Association*, 1949.

[28] P. K. Romano, N. E. Horelik, B. R. Herman, A. G. Nelson, B. Forget, and K. Smith. Openmc: A state-of-the-art monte carlo code for research and development. In *Joint International Conference on Supercomputing in Nuclear Applications*, 2014.

[29] D. C. Montgomery and G. C. Runger. *Applied statistics and probability for engineers*. John Wiley & Sons, 2010.

[30] Q. Pan and K. Wang. An adaptive variance reduction algorithm based on rmc code for solving deep penetration problems. *Annals of Nuclear Energy*, 2019.

[31] E. M. Gelbard and R. E. Prael. Monte carlo work at argonne national laboratory. Technical report, 1974.

[32] J. R. Tramm and A. R. Siegel. Memory bottlenecks and memory contention in multi-core monte carlo transport codes. In *Joint International Conference on Supercomputing in Nuclear Applications*, 2014.

[33] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, 1967.

[34] J. L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 1988.

[35] R. H. B. Netzer and B. P. Miller. What are race conditions? some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1992.

[36] W. J. Bolosky and M. L. Scott. False sharing and its e ect on shared memory performance. In *Proceedings of the Fourth symposium on Experiences with distributed and multiprocessor systems*, 1993.

[37] A. J. Kunen, T. S. Bailey, and P. N. Brown. Kripke-a massively parallel transport mini-app. Technical report, Lawrence Livermore National Lab.(LLNL), 2015.

[38] K. Kobayashi, N. Sugimura, and Y. Nagaya. 3d radiation transport benchmark problems and results for simple geometries with void region. *Progress in nuclear energy*, 2001.

[39] M. A. Abdou. Tritium breeding in fusion reactors. In *Nuclear data for science and technology*, 1983.

[40] U. von Moellendorff, U. Fischer, H. Giese, F. Kappler, R. Tayama, E. Wiegner, H. Klein, and A. Alevra. The karlsruhe neutron transmission experiment (kant): Spherical shell transmission measurements with 14 mev neutrons on beryllium. Technical report, 1996.