ABSTRACT

YESSAYAN, RAFFI ALEXANDER. Improvements to the THOR Neutral Particle Transport Code on High-Performance Computing Systems via Acceleration, Parallelization, and Performance Analysis. (Under the direction of Dr. Yousry Y. Azmy).

As the availability and capability of high-performance computing (HPC) resources has grown, so has the demand for high-fidelity simulations. Unfortunately, for methods like the three-dimensional discrete ordinates approximation to the neutron transport equation, the cost of high-fidelity simulations climbs rapidly because of the high dimensionality of the phase space. Simulations can reach into the billions of degrees of freedom and require weeks to run on unaccelerated, serial implementations of the method. For many applications, this run time is intractable. To address this, modern codes must implement more advanced algorithms, such as iterative acceleration and parallelism to reduce run times and better capitalize on computing resources. When implemented efficiently and given sufficient resources, these methods can reduce a two-week simulation to hours or minutes. However, these advanced methods may also introduce complications. Depending on the implemented iterative method, acceleration techniques, such as diffusion synthetic acceleration, can be resource heavy, while others, such as Chebychev acceleration, can be unstable for certain problems. Parallelism introduces the requirement for inter-process communication and, for certain methods, asynchronicity.

Modifications and analysis have been performed on the THOR neutral particle transport code to implement the above improvements. THOR is intended to enable simulation of complex geometries in a variety of scenarios, ranging from reactor simulation to non-proliferation and, as such, must provide capabilities for a variety of applications. This work describes four major efforts to improve THOR's capabilities and bring it closer to production level. First is the implementation of Chebychev power iteration acceleration, a low resource cost method that reduces iteration count ~2x *versus* the existing fission source extrapolation and roughly comparable to some implementations of the high-memory usage JFNK solver. This implementation also includes control logic designed to improve the stability of the method for challenging problems. Second is the implementation of angular domain decomposition parallelism. This synchronous parallelism allows THOR to better utilize the resources of mid-scale computing systems with 10s to 1000s of processors and can fully replace the serial inner/outer iteration solver of THOR. For typical THOR simulations, this results in speedups

of ~10x to ~100x depending on available resources and problem parameters. The third item, in pursuit of optimizing parallel performance, is the development of a modular parallel performance model. This model captures the run time of THOR’s mono-energetic, zeroth spatial order, unaccelerated inner/outer iteration solver to within ~5% for reasonably large problems. The model is designed such that future modifications to the THOR code or to the parameters of interest (e.g. adding group or spatial order dependence), can be easily added. This makes the performance model a powerful tool not just for evaluating the efficiency of existing code, but also for performing comparative studies and localizing inefficiencies in later additions to THOR. A long-term goal of the THOR project is to enable massively parallel spatial domain parallelism on 10^5+ processors. The implementation of the spatial domain decomposition (SDD) algorithms on unstructured meshes is a massive undertaking beyond the scope of this work. However, Task four lays the groundwork for this capability by analyzing the communication inefficiencies present in PIDOTS, a three-dimensional Cartesian mesh code which implements a parallel Gauss-Seidel SDD algorithm. This analysis demonstrates that the high count, small-message communication paradigm used by PIDOTS leads to communication slowdowns on the order of 10x for 10^5 processors and identifies steps that can be taken to mitigate this effect before implementing a similar parallelism in THOR.

Improvements to the THOR Neutral Particle Transport Code on High-Performance Computing Systems via Acceleration, Parallelization, and Performance Analysis

by
Raffi Alexander Yessayan

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Master of Science

Nuclear Engineering

Raleigh, North Carolina

2018

APPROVED BY:

Dr. Yousry Azmy Committee Chair	Dr. Dmitriy Anistratov
Dr. Edward Gehringer	Dr. Sebastian Schunert

DEDICATION

To my Mom and Dad, Lynn and Arthur Yessayan, for a lifetime of unconditional love, guidance, support, and encouragement.

BIOGRAPHY

Raffi Alexander Yessayan was born in Charlotte, North Carolina on July 21st, 1993 to Lynn and Arthur Yessayan. He lived in Charlotte until being accepted at North Carolina State University in Raleigh, North Carolina. During his undergraduate career, he pursued significant coursework in both nuclear engineering and computer science. Additionally, for two years, he worked as a teaching assistant in both introductory computing and Fortran programming courses. In 2015, he graduated Summa Cum Laude with a Bachelor of Science in nuclear engineering, minors from both the French and Computer Science departments, and having completed the University Honors Program.

For his graduate coursework, he remained in North Carolina State University's Department of Nuclear Engineering and began studying optimization and parallelization techniques for unstructured mesh neutron transport codes under the direction of Dr. Yousry Y. Azmy. He is a recipient of the Consortium for Nonproliferation Enabling Capabilities Graduate Fellowship.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Yousry Y. Azmy, for his support and guidance throughout my graduate career. I would also like to thank Dr. Sebastian Schunert for contributing his insight and experience to my project, especially with regards to the inner workings of THOR. Finally, I would like to thank my parents for always providing support and encouragement throughout my academic career.

The work of the author is based upon work supported by the Department of Energy National Nuclear Security Administration under Award Number(s) DE-NA0002576.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 Introduction

1.1 The Neutron Transport Equation

The neutron transport equation describes the evolution of the neutron distribution over a six-dimensional phase space formed by position, direction of motion, and energy. This distribution is referred to as angular flux and, in this thesis, will be denoted as $\psi(\vec{r}, \widehat{\Omega}, E, t)$, where $\vec{r}$ is the position of the particle in cartesian (*x*,*y*,*z*) space within a volume $d\vec{r}$, $\widehat{\Omega}$ is the angular description of the particle's direction of travel within a cone $d\widehat{\Omega}$, E is the particle's energy within a band dE, and t is the time of evaluation. A standard representation of this equation is given below [1]:

$$\begin{aligned}
&\frac{1}{v}\frac{\partial}{\partial t}\psi\big(\vec{r}, \widehat{\Omega}, E, t\big) + \widehat{\Omega} \cdot \vec{\nabla}\,\psi\big(\vec{r}, \widehat{\Omega}, E, t\big) + \sigma_t(\vec{r}, E)\psi\big(\vec{r}, \widehat{\Omega}, E, t\big) \\
&= \int_0^\infty dE' \int_{4\pi} d\widehat{\Omega}'\ \sigma_s\big(\vec{r}, \widehat{\Omega}' \to \widehat{\Omega}, E' \to E\big)\psi\big(\vec{r}, \widehat{\Omega}, E, t\big) \\
&+\frac{\chi(E)}{4\pi k}\int_0^\infty dE' \int_{4\pi} d\widehat{\Omega}'\ \nu\sigma_f(\vec{r}, E)\psi\big(\vec{r}, \widehat{\Omega}, E, t\big) \\
&+q_{external}\big(\vec{r}, \widehat{\Omega}, E, t\big)
\end{aligned} \tag{1}$$

In Eq. (1), v represents the neutron speed, while $\sigma_t, \sigma_s, and\ \sigma_f$ represent, respectively, the neutron cross sections for total interaction, scattering, and fission. The average number of neutrons per fission event is given by ν and the resulting fission spectrum is given by χ.

In this form, the first line of the equation represents the neutron sinks, in order: the change in population with time, the out-streaming of particles, and interaction of particles. The second line represents the scattering source, or the influx of particles into a volume of phase space resulting from scattering interactions in other regions $\big(\vec{r}', \widehat{\Omega'}, E', t\big)$ that deposit a particle into $\big(\vec{r}, \widehat{\Omega}, E, t\big)$. The third line represents the contributions of fission events that result in a neutron in $\big(\vec{r}, \widehat{\Omega}, E, t\big)$. Note that here, the fission distribution, $\chi(E)$, is taken to be independent of the original energy, E' and that fission is assumed to be isotropic, i.e., it emits particles evenly amongst the 4π directions of the unit sphere. The fourth line provides an arbitrary external source of neutrons. The form given in (1) represents a combination of the two forms represented in the THOR transport code, k-eigenvalue and fixed source. In THOR, for the former, k is part of the solution and can be any positive real value, while $q_{external}$ is taken to be zero. For the latter, k must be less than 1 for a solution to exist and $q_{external}$ is a problem parameter.

While (1) provides a clean expression for the neutron transport equation, it is several steps removed from a form convenient to implement into a numerical code. First, for the purposes of this work, all problems will be taken to be steady state. As such, the time derivative of the flux is zero and the corresponding term can be removed. Second, the continuous variables in space, angle, and energy must be discretized, along with the integrals in angle and energy. Multiple methods exist by which to perform this discretization. One method, which is implemented by THOR, is the method of Discrete Ordinates [1], or the S_n method, which divides the spatial volume into elements, the angles into a set of discrete ordinates, $n \in 1..N$, and the energy range into a set of energy groups, $g \in 1..G$. The name S_n stems from the subscript n denoting the level of the angular quadrature used to approximate the angular integrals.

Rewriting Eq. (1) in this form will require several stages. First, is the treatment of the angular variable, Ω, via the method of Discrete Ordinates. In this method, the continuous variable Ω is replaced with the finite set of directions, Ω_n, where $n = 1..N$. Via numerical quadrature, this set of angles, paired with appropriate weights (w_n), can be used to replace integrals over Ω with sums over $w_n\Omega_n$. Finally, the scattering cross section, σ_s, which is a function of both the incoming and outgoing angle, must be represented using an expansion in spherical harmonics. Eq. (2) shows the Discrete Ordinates form of the transport equation with continuous space and energy [1].

$$\widehat{\Omega}\nabla\psi_{\mathrm{n}}(\vec{r},E)+\sigma_t\psi_{\mathrm{n}}(\vec{r},E)=\sum_{l=0}^{L}\sum_{m=-l}^{l}\Upsilon_{l,m}(\widehat{\Omega}_n)\int_0^\infty\sigma_s(\vec{r},E'\to E)\phi_l^m(\vec{r},E)dE$$
$$+\frac{\chi(E)}{4\pi k}\int_0^\infty\nu\sigma_f\phi(\vec{r},E)\,dE+\frac{S_n(\vec{r},E)}{4\pi} \qquad (2)$$

Here, ψ_n is the angular flux of the specific angle Ω_n and $\Upsilon_{l,m}$ is the spherical harmonic of degree l and order m. All other variables subscripted with *n* are the same quantities as before, except now they only represent the component along direction *n*.

Next is the discretization of the energy domain. This will be done via the multigroup method, which subdivides the energy range into a finite number of groups, *G*, and defines a set of *G* transport equations. This replaces integrals over energy with sums over each group. For quantities which are energy dependent, such as cross sections, the group average value is defined as the flux weighted average over the energy range of the group. An example of the multigroup discrete ordinates equations for continuous space is shown below [2].

$$\widehat{\Omega}\nabla\psi_{\mathrm{n,g}}(\vec{r})+\sigma_{t,g}\psi_{\mathrm{n,g}}(\vec{r})=\sum_{l=0}^{L}\sum_{m=-l}^{l}\sum_{g\prime=1}^{G}\Upsilon_{l,m}(\widehat{\Omega}_n)\sigma_s^{g'\to g}(\vec{r})\phi_{l,g'}^m(\vec{r})$$
$$+\frac{\chi_g}{4\pi k}\sum_{g\prime=1}^{G}\nu\sigma_{f,g'}\phi_{g'}(\vec{r})+\frac{S_{n,g}(\vec{r})}{4\pi} \qquad (3)$$

Here, $\psi_{g,n}$ is the angular flux of the specific group-angle combination. In this form, the solution is only obtained along one ordinate, $\widehat{\Omega}_n$, and one group, g, at a time.

The final phase dimension to discretize is space. There a variety of methods by which to do this. However, in general, this step is performed by dividing the spatial domain into a set of mesh cells. Each cell in the mesh has constant material properties and, as a result, constant cross section. Next, a set of relations are defined to relate the cell center and outbound fluxes to the flux incident on the incoming face(s) of the mesh cell volume. As discussed in [1] and [2], these relations can take many forms depending on the needs of the application.

As a result of this discretization, for a given angle, the flux in a cell is only dependent on values local to that cell, sources and cross sections, and the flux incident on the cell from its upstream neighbors. This allows for the use of a mesh sweep to evaluate the flux in all cells, given a starting boundary with a known incident flux.

The known boundary, combined with the cell material information, can be used to solve for the flux along a given ordinate in the cell and then propagate a cell-edge flux to the downstream mesh neighbor. When this process is repeated across the entire mesh, it is referred to as a sweep. After every angle has been swept, a new iterate of the flux profile in the domain can be generated. As the next section will describe, this iteration process continues until a sufficiently converged value of the flux solution is produced.

1.2 Iterative Neutron Transport Solvers

Moving from the mathematical expression of the numerical neutron transport equation to a workable algorithm is relatively straightforward. Solution of the transport equation typically uses a set of two nested iterations: outer and inner. A flow diagram of these processes is given in Figure 1. As it shows, the inner iteration is used to converge the in-scatter source, while the outer iteration is used to converge the up-scatter source. By sweeping energy groups from highest to lowest energy, the down-scatter source for group g is known at the start of the group g solve. In problems with fission, the outer iteration is augmented with an update to the fission source. This augmentation forms a power iteration and k_{eff} update.

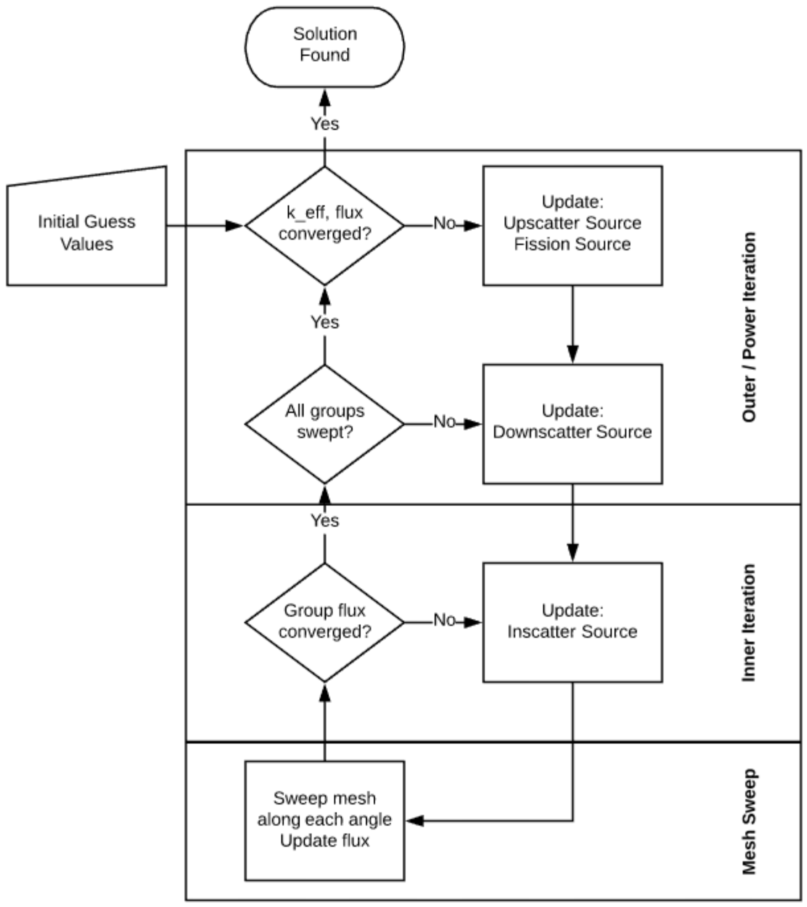

Figure 1: Flow diagram of iterative transport solution.

Also of note is the lowest level of the iteration hierarchy, the mesh sweep. In this operation, the current source and flux values are used to calculate a new flux iterate by moving along each cell of the mesh, starting from a boundary with known in-flux. In this way, each cell is solved using the known solution from its upstream neighbor. This sweep is performed along the entire mesh for each discrete ordinate in the angular approximation. However, individual sweeps are independent of one another until they are all completed. Once all sweeps have completed, an updated value for the angular flux moments can be generated.

1.3 The THOR Neutral Particle Transport Code

The THOR neutral particle transport code is maintained at NCSU in collaboration with Idaho National Laboratory (INL). It is a Fortran 95 application designed to solve the three-dimensional discrete ordinates approximation discretized by unstructured tetrahedral meshes. As a spatial approximation, it implements the arbitrarily high order transport method of the characteristic type (AHOTC). The specifics of this implementation are beyond the scope of this work and are detailed in [3].

To the best of author's knowledge, THOR is unique in its implementation of this arbitrarily high-order method on unstructured, tetrahedral meshes. As such, it provides a useful testbed for the implementation and evaluation of modifications to the AHOTC method in unstructured geometries. At the beginning of the author's graduate program, THOR maintained the following features:

- Serial solvers for both *k*-eigenvalue and external source problems using a traditional outer/inner iteration scheme
- Support for arbitrary energy groups, angular quadrature refinements, and spatial expansion order
- A Jacobian-Free Newton-Krylov Solver for *k*-eigenvalue problems
- Fission source extrapolation for acceleration of the outer iteration

Using these features, the THOR code could be used to evaluate a variety of scenarios relevant to the modern nuclear engineering field. These include reactor core modeling [4], source localization, and the evaluation of sub-critical weapons-grade material. However, the neutron transport equation can be an extremely expensive problem to solve since it is discretized in 6 variables – 3 directional, 2 angular, and 1 energy. Refinements to a given problem can lead to massively increased computational costs. For small problems, this scaling can yield several thousand degrees of freedom (DOF) and solving times on the order of seconds. Even for many iterations of parameter studies, these costs are quite minor. However, for larger problems, these costs can rapidly become intractable. For the largest problem run in THOR to date, a high-resolution model of the Advanced Test Reactor (ATR) at Idaho National

Lab (INL) [5], this scaling yielded approximately four billion DOF. This problem had a solution time of ~2 weeks and was simply too time-intensive to run repeatedly. Unfortunately, as evaluating a high-resolution reactor model, such as the ATR problem, forms a cornerstone of the verification and validation plan for the THOR code, this posed an obvious challenge. Based on the need to repeatedly and rapidly run problems on scales up to that of the ATR case, it was determined that several modifications would need to be made to THOR.

1.4 Desired Improvements

As the ATR problem had demonstrated, the then-current implementation of THOR was insufficient to tackle the massive problem sizes resulting from the desire for high-fidelity results. Fortunately, there were a variety of possible modifications that could be made to THOR to significantly speed up the solver.

However, there were also several limiting factors that restricted this field of choices. Most important was the fact that previously completed work from another researcher had verified the existing code via the method of manufactured solutions [6]. This meant that it was desirable to introduce changes that encapsulated, rather than modified, existing pieces of code. Maintaining this separation of functionality would allow for improvements to be made to the code's speed without risk of impacting verified code. The second constraint was related to the future plans for THOR. Since many planned projects related to the capabilities of the non-JFNK solvers, it would be ideal if the power iteration solver saw the greatest benefit from the implemented changes.

From these needs and constraints, a goal could be developed – reduce the runtime of the outer/inner iterative solvers as much as possible without significantly modifying existing routines and without imposing undue increases in resource utilization.

1.5 Selected Improvements

Based on the goal defined in section 1.4, there were two obvious paths for improving the performance of THOR. First, implementation of a new acceleration scheme better than fission source extrapolation and, second, the parallelization of one of the phase space dimensions of the neutron transport equation.

For the first, a new acceleration method, there were again two options. In an outer/inner solver schema, acceleration schemes can be applied to either (or both) the outer or inner iterations. Several powerful inner iteration schemes exist which sharply decrease the required number of inner iterations. Unfortunately, these are also typically more resource intensive, requiring much more memory, and would require implementing significant modifications to already-verified code. Additionally, *k*-eigenvalue problems in THOR are dominated by outer iterations. Some outer accelerations, such as the already implemented Fission Source Extrapolation, are comparatively cheap and require only a short calculation at the end of an outer iteration cycle. This small implementation and runtime overhead is balanced out by the comparatively small acceleration they produce.

In the end, since eigenvalue problems are common for THOR, the lightweight, non-intrusive nature of outer acceleration was found to be a better fit for the current needs. But, given the expected performance of outer accelerations, it was also decided that simply implementing a new outer acceleration would not provide a sufficient increase in performance.

To complement the outer acceleration some aspect of the code would also need to be parallelized. From the variables involved in the transport equation, this left three options, parallelism in space, angle, or energy.

Parallelism in energy is not ideal as the resulting algorithm is asynchronous due to energy group coupling and, since the number of groups is typically small, limited in scalability.

While also asynchronous, spatial parallelism is a tempting option. The sheer number of mesh cells in most problems allows for massive parallelization. Unfortunately, implementing this parallel structure on unstructured meshes is a major undertaking and well beyond the scope of this work.

Finally, there is angular parallelism. Like spatial decomposition, the angular domain grows in cost at an above-linear rate. However, unlike in the other options, angular decomposition is synchronous. Angles are evaluated independently for most of the inner iteration process. During the sweep, each angle is evaluated separately; and, at the end, aggregate measures are generated form the completed result vector. Because of this, it would be a relatively non-invasive task to implement an angular decomposition into a functioning transport solver.

Based on this brief sampling of the available options for acceleration and parallelization, an outer acceleration paired with angular parallelization best suits the needs of the THOR project. The implementation and evaluation of the former will be detailed in Chapter 2, while the latter will be described in Chapter 3. Additionally, further evidence from literature will be provided to justify each choice.

1.6 Thesis Outline

Each of the following chapters in this thesis will present work from single thrust area of the THOR improvement project. For each thrust area, an overview of the existing literature will be provided, followed by a discussion of the methodology, results, and conclusions related to the work

Chapter 2 will discuss the development of the outer acceleration scheme decided on in the previous sections. Chapter 3 will address the implementation of angular parallelization. Next, Chapter 4 will present a compartmentalized method for parallel performance evaluation of the THOR code. Chapter 5 will present an analysis of the performance of parallel codes on a high-contention HPC system. Chapter 6 will highlight a variety of software management improvements to the THOR project and explain their importance in the role of a production-level neutronics code. Finally, Chapter 7 will provide conclusions and an overview of the net contribution of this project.

2 Iterative Acceleration Methods

2.1 Introduction

As determined in sections 1.4 and 1.5, the optimal acceleration technique for this project was not the one which demonstrated the largest increase in computational speed, but rather the one which provided a reasonable amount of speedup for a relatively low cost in implementation complexity and invasiveness. This section will review several common inner and outer acceleration schemes. However, eigenvalue calculations are dominated by outer iterations and several light-weight outer acceleration algorithms are readily available. In the end, inner accelerations were deemed sub-optimal for our needs and work shifted focus to outer acceleration methods. From this pool, Chebychev acceleration was selected. While there was literature, [7], and anecdotal experience, [8], indicating that Chebychev was not an optimal method, it was clearly documented in [9] and similar in implementation to the fission source extrapolation already implemented in THOR. These benefits were deemed sufficient to outweigh the negatives. In the end, with some modification to the standard algorithm, Chebychev acceleration proved surprisingly effective and yielded speedups significantly better than fission source extrapolation and roughly equal to some versions of the THOR JFNK solver.

2.2 Literature Review

2.2.1 Fission Source Extrapolation

As described in [10], error-mode based fission source extrapolation (FSE) is one of the most simplistic outer iteration acceleration schemes. However, this does not prevent it from providing a useful degree of speedup in most problems. FSE is based on the fact that, as an unaccelerated problem converges, the relative change in the fission density of the problem approaches a constant. By identifying this relative change constant, an acceleration term can be generated. Mathematically, this is achieved through the following steps. First, define the iteration q fission density error as the sum over all spatial cells:

$$e^{q} = \sum |D^{q} - D^{q-1}| \tag{4}$$

where D^q is the fission density in iteration q. The fission density is calculated simply as the sum of all fissions in all groups. Next, this error term is used to evaluate an estimate of the eigenvalue and of the acceleration term as:

$$\lambda^{q} = \frac{e^{q}}{e^{q-1}} \tag{5}$$

$$\Theta^{q} = \frac{\lambda^{q}}{1-\lambda^{q}} \tag{6}$$

This extrapolation term is then applied to both the scalar flux and the fission density:

$$D^{q'} = D^{q} + \Theta^{q}(D^{q} - D^{q-1}) \tag{7}$$

$$\phi^{q'} = \phi^{q} + \Theta^{q}(\phi^{q} - \phi^{q-1}) \tag{8}$$

The authors of [10] mention that this approach is highly effective in converging *k*-eigenvalue problems and problems with significant up-scatter. However, under some cases, a stable value of Θ will not be found and the acceleration scheme will not prove effective.

2.2.2 Coarse Mesh Rebalance

Coarse Mesh Rebalance (CMR), discussed in [11, 12, 13], is an inner acceleration scheme that can be applied to the neutron transport equation. As the name implies, the method involves superimposing a coarse mesh atop the existing fine mesh used by the transport solver. The current cell-average flux iterate is then accelerated by generating a multiplicative term called the rebalancing factor. In essence, this method uses a lower fidelity, faster solve to provide an improved guess to the higher-order solver.

Overall the methodology is rather simple. As described by [13] for the slab geometry case, a set of balances are generated such that the flux in the coarse mesh satisfies:

$$\psi^{l}_{n,j+\frac{1}{2}} = \begin{cases} \psi^{l-\frac{1}{2}}_{n,j+\frac{1}{2}} * F^{l}_{j}, \mu_n > 0 \\ \psi^{l-\frac{1}{2}}_{n,j+\frac{1}{2}} * F^{l}_{j+1};\ \mu_n < 0 \end{cases} \tag{9}$$

where the values for flux at the cell edges, $\psi^{*}_{n,j\pm\frac{1}{2}}$, are calculated using the balance relationship implemented in the given code and the values of F^{l}_{j} are the acceleration coefficients. When applied to all cells in the fine slab mesh, this forms a tridiagonal matrix which can be solved to yield the F^{l}_{j} terms. Having determined a value for the acceleration term in each cell, an accelerated iterate of the cell-center flux can be generated as:

$$\psi^{l}_{n,j} = \psi^{l-\frac{1}{2}}_{n,j} * F^{l}_{j} \tag{10}$$

As the condition of two edges per cell in 1D yields a tri-diagonal matrix, it can be inferred that higher dimensional problems will yield an increasingly wide block-diagonal structure representing the edge coupling between cells.

While the CMR method has been a mainstay of simple, effective acceleration methods, it presents several problems that make it unfit for our objectives described previously.

First, the presence of the matrix-solve means that CMR will have a significantly higher per-iteration cost than non-matrix methods, both in memory and time utilization. While this is not problematic for small problems, it will become a critical concern for extremely large problems, which are both time and memory limited.

Further complicating the use of CMR concerns its application to unstructured meshes. In the simple example described above, a fine slab mesh was coarsened into a less fine slab mesh. This same-geometry coarsening allows for an easy implementation and a simple mapping of the flux variables at cell edges. Applying this same routine to THOR would prove challenging since an arbitrary group of tetrahedrons cannot be coarsened into another tetrahedron. This difficulty could be avoided by using the same mesh for both the fine and coarse meshes. Unfortunately, this approach makes the method inflexible as it would only allow for that single configuration.

Several sources detail the process of overcoming the difficulties of unstructured-mesh CMR [14, 15]. However, in both, the solution was to implement a degree of regularity to the mesh. In doing so, the mesh was unstructured below some level of detail and structured above. An example of this is provided from openMOC, [15], in Figure 2. Note that the implemented acceleration is coarse-mesh finite difference, not CMR. Regardless, the system for handling unstructured meshes is the same.

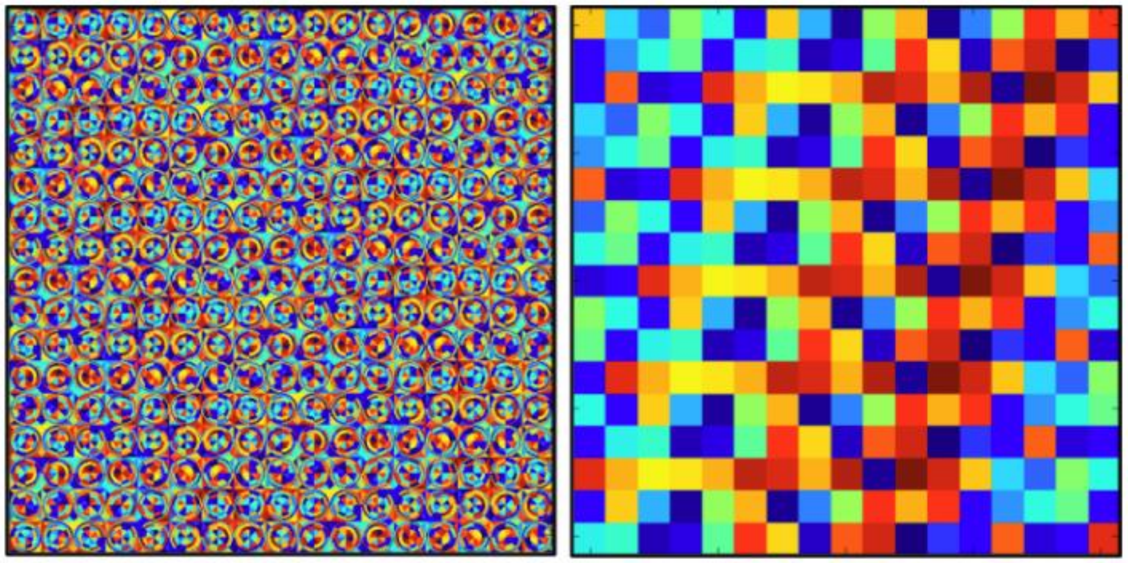

Figure 2: Example of level-of-detail based structured/unstructured mesh, from [16].

As can be seen on the left of Figure 2, even though the mesh is unstructured, it does possess a degree of regularity stemming from the repetitive nature of the unit cell lattice. This regularity is capitalized on to provide matching boundaries for CMR. Similarly, in [14], the fuel structure is coarsened into the repetitive hexagonal assembly shape.

Unfortunately, due to the desired arbitrary-geometry nature of THOR, it would be impossible to guarantee mesh repetitions of this sort in every problem without artificially inserting them. As such, CMR is not a viable method for this application given its current needs.

2.2.3 Diffusion Synthetic Acceleration

Alcouffe [7], describes a trio of Diffusion Synthetic Acceleration (DSA) methods for use in multi-group, discrete-ordinates problems. The three treatments are designed to provide a stable, highly-effective acceleration method. These stability improvements make DSA more efficient than unaccelerated transport iterations under essentially all conditions except extreme cases, such as unbounded heterogeneity [13]. This degree of improvement would provide a massive boon to THOR.

DSA is characterized by the utilization of a neutron diffusion operator to accelerate the inner iterations of the neutron transport solution. From a high-level viewpoint, the results of an inner transport solve are used as the inputs to a diffusion-like solve. The diffusion solve provides a greatly increased rate of convergence of the flux iterates. A significant benefit of this method is that it is effective under most conditions, including those where true diffusion would not be [7].

However, like CMR, the solution of DSA acceleration requires the creation of an additional matrix system of equations in the iterative correction terms, consuming both additional memory and execution time per iteration. However, given the results demonstrated in [7], this increase in per-iteration cost would be easily outweighed by gains in iteration count reduction. The comparison in [7] shows DSA consuming anywhere from ~3 to ~10 times fewer iterations than unaccelerated transport. Additionally, the method proves similarly effective when compared to Chebychev and CMR.

These substantial speedups, coupled with the evidence of guaranteed convergence provided by [7] would seem to indicate that DSA is the optimal acceleration to implement into THOR. However, as shown by Azmy [17], the guarantee of stability can be lost in multidimensional problems as material heterogeneity becomes unbounded. Furthermore, as [7] and [18] discuss, DSA is most easily applied to fixed source problems. Applying it to general eigenvalue problems can prove challenging. Since THOR primarily evaluates eigenvalue problems, this weakness, coupled with the desire to modify verified code structures as little as possible, rendered DSA suboptimal to implement for this project. There is still a significant desire to implement it in the future as project needs and constraints evolve. But, for now, it is not the optimal choice.

2.2.4 Chebychev Acceleration

From a review of the literature, Chebychev acceleration seems to be often used, but rarely described. It is mentioned as an option or used for efficiency comparisons by [7], [9], and [11], but the implementation is only fully described in Hébert’s work, [9]. Like fission source acceleration, Chebychev acceleration does not require a matrix evaluation [11]. Instead, it uses an estimate of the dominance ratio to modify the next flux iterate using a superposition of old iterates. This makes it extremely cheap to implement in terms of run time costs since it only involves a set of uncoupled expressions.

In the method described in [9], Chebychev acceleration is implemented as a cyclical process. First, a series of unaccelerated iterations are used to loosely converge the dominance ratio. Then, some fixed number of accelerated iterations are performed, with each one using the dominance ratio estimate as an acceleration parameter. After the fixed number of accelerations is complete, relations based on the Chebyshev polynomials are used to extract the equivalent un-accelerated dominance ratio. This new dominance ratio is used to drive a new cycle of accelerated iterations. At each step in the process, the Chebychev relationships are used to generate two terms, α and β, which are combined with the old iterative flux estimates to provide an improved new flux estimate.

In the same publication, Hébert compares the Chebychev method to a proposed variational technique based on Rayleigh Ratios. This technique proves slightly more involved than Chebychev acceleration to implement but yields only marginally better results in terms of iteration count.

Similarly, in [7], Chebychev is used as one of a variety of acceleration methods for benchmarking. Of note from these other methods are the unaccelerated, Coarse Mesh Rebalance, and Diffusion Synthetic Acceleration results. As would be expected from the earlier discussion, the DSA inner acceleration scheme requires far fewer iterations and is more stable than any other method. However, it has the most complicated implementation and highest per-iteration computational cost. Of the remaining methods, Chebychev is generally able to converge faster than both CMR and unaccelerated inner iterations. However, there is one case each [7] where Chebychev fails to outperform the other methods. These failures are attributed to insufficient convergence in the initial guess value for the dominance ratio. Both [11] and [9] emphasize that the method may become unstable if the initial dominance ratio guess is poor.

2.3 Implementation

The Chebychev acceleration method implemented in THOR follows closely the form developed in [9]. A summary of this method, along with the modifications made to it in our implementation is presented here.

First, a set of unaccelerated iterations is performed while estimating the dominance ratio as the 2-norm of the flux difference in iteration k and $k-1$:

$$\sigma^{k+1} = |\phi^{k} - \phi^{k-1}|_2 \tag{11}$$

When the dominance ratio estimate σ^{k+1} exceeds 0.5, a Chebychev acceleration cycle is triggered and the current iteration, k, is designated n^*. In this cycle, m Chebychev acceleration steps are executed. During each of these steps denoted with an iterative index p, a pair of acceleration parameters, α and β, are generated and applied to the flux prediction.

$$\phi^{n^*+p+1} = \phi^{n^*+p} + \alpha\left(\phi^{n^*+p} - \phi^{n^*+p-1}\right) + \beta * \left(\phi^{n^*+p-1} - \phi^{n^*+p-2}\right) \tag{12}$$
$$p = 1, .., m$$

$$\alpha = \begin{cases} \dfrac{2}{2 - \sigma^{n^*+1}} & p = 1 \\ \dfrac{4}{\sigma^{n^*+1}} * \left(\dfrac{\cosh[(p-1)]}{\cosh(p\gamma)}\right) & p > 1 \end{cases} \tag{13}$$

$$\beta = \begin{cases} 0 & p = 1 \\ \left(1 - \dfrac{\sigma^{n^*+1}}{2}\right) - \dfrac{1}{\alpha^{(p)}} & p > 1 \end{cases} \tag{14}$$

$$\gamma = \cosh^{-1}\left(\frac{2}{\sigma^{n^*+1}} - 1\right) \tag{15}$$

The acceleration defined above can be applied for *m* iterations. After this, the acceleration cycle ends and an update to the dominance ratio must be performed to allow for the re-initiation of the Chebychev scheme. The process of reconstructing the unaccelerated dominance ratio at the conclusion of a Chebychev cycle is as follows:

$$E = \frac{\left|\left|(\phi^{n^*+m} - \phi^{n^*+m-1})\right|_2\right|}{|(\phi^{n^*+1} - \phi^{n^*})|_2} \tag{16}$$

$$E^* = \left[C_{m-1} \cdot \frac{\left(2 - \sigma^{n^*+1}\right)}{\sigma^{n^*+1}}\right]^{-1} \tag{17}$$
$$C_{m-1}(x) = \cosh[(m-1) * \cosh^{-1}(x)]$$

The two factors, E and E^*, represent the achieved and theoretical error reduction for a Chebychev cycle of *m* steps. Next, they are combined to provide an estimate of the unaccelerated dominance ratio via:

$$\sigma^{n^*+m+1} = \frac{\sigma^{n^*+1}}{2} \cdot \left[\cosh\left(\frac{cosh^{-1}\left(\frac{E}{E^*}\right)}{m-1}\right) - 1\right] \tag{18}$$

Assuming the accelerated iteration has remained stable, the new dominance ratio can be used immediately as the σ^{n^*+1} value of a new Chebychev cycle of length m.

Unfortunately, this method was found to be frequently unstable for a variety of test cases. As a result, several modifications were made to the algorithm proposed by [9] outlined above and the modified version was implemented in THOR.

The most significant of these modifications was to recalculate the theoretical error reduction, E^*, during every iteration, replacing the value of m with the current value of p. Using this continuous update of the $E^{*,(p-1)}$, the effectiveness of the Chebychev cycle can be determined.

If, after a user-determined number of cycles, the achieved error reduction is less than the $p - 1$ theoretical error reduction, the cycle is deemed ineffective and aborted. From evaluation of the code performance, ineffective cycles seemed to occur when initial dominance ratio estimate was poorly converged. As such, when a cycle is aborted a configurable number of power iterations are queued up prior to the start of the next Chebychev cycle. By doing this, the acceleration benefits of Chebychev can be maintained while at the same time actively monitoring for poor convergence.

In addition to this algorithmic change, general stability was improved by tightening the criteria under which Chebychev iterations start. This was accomplished by requiring a small number of power iterations to be performed after the initial $\sigma > 0.5$ condition is reached. This delays the onset of the Chebychev system, but also tends to decrease the number of aborted cycles.

2.4 Results

The original results from the implementation of this acceleration scheme were presented in [4]. For that evaluation, the performance of the newly implemented Chebychev method was compared to the existing unaccelerated, fission-source accelerated, and JFNK solvers. To evaluate the effectiveness of the methods, a high-dominance ratio benchmark problem, $\sigma \approx 0.995$, was selected from the literature. This benchmark took the form of a homogenous cube with a side-length of 200cm and material properties: $\sigma_t = 1, \sigma_s = 0.7, \nu\sigma_f = 0.39\ cm^{-1}$. A one-dimensional version on this configuration is described in [19]. On this test domain, a monoenergetic flux was converged to 10^{-7} in scalar flux and 10^{-8} in eigenvalue.

For comparison purposes, the default implementations of the THOR unaccelerated and fission source accelerated solvers, see [6], were used. These simply implement the methodologies described previously with no special modifications.

Additionally, 3 varieties of the THOR JFNK solver routine were exercised. These three methods were JFNK-1 (2), JFNK-1 (4), and JFNK-2. The first two evaluate the nonlinear function using a single outer and either 2 or 4 inner iterations. These limitations on the number of iterations are designed to minimize the memory footprint of JFNK so that it can be applied to larger problems without becoming memory constrained.

JFNK-2 abandons this memory constraint and allows for the construction of up to a 30-dimension Krylov subspace. Unfortunately, this method is extremely memory intensive, with each dimension of the subspace requiring an additional copy of the angular flux moments for the entire domain to be stored. This is obviously an intractable method for large problems, but it can be used to provide significant speedup on otherwise challenging problems with small domains.

As shown in Table 1, the implemented Chebychev acceleration performed quite well compared to the non-JFNK transport solvers and performed reasonably well when compared to the JFNK solvers with limited memory footprints. Note that, as is typical for THOR, two inner iterations were performed per outer iteration for each of the three power iteration based solvers.

Table 1: Comparison of THOR solver methods [4].

Method	Power/Newton Iteration count (Krylov It. Count)	Transport Sweep Count	Speedup
Power Iteration	2714	5428	1
Fission source extrapolation	1109	2218	2.5
Chebychev acceleration	667	1334	4.1
JFNK-1 (2)	8 (679)	1376	3.9
JFNK-1 (4)	7 (522)	2120	2.6
JFNK-2	7 (649)	657	8.3

From these promising speedup results, Chebychev acceleration fulfills the objectives set forth for acceleration of THOR’s outer iterations. It provides a significant speedup in general problems and does not incur significant calculation or storage overhead. The method provides significant improvement over fission source extrapolation while maintaining a comparable per-iteration computational cost and only requires the storage of an additional scalar flux vector (as well as assorted scalars). Additionally, the method provides a roughly equivalent speedup, lower-memory alternative to the JFNK-1 (2/4) methods. This serves a dual benefit. First, it allows for larger problems to be solved, and second, it allows for transport iterations to be observed directly by developers for debugging. By its nature, the JFNK solver is comparatively minimalist in terms of in-process output compared to the more fully-featured iterative solver output.

Unfortunately, Chebychev acceleration does not provide a perfect approach. As has been noted several times in this chapter, the method is more prone to instability than the unaccelerated and fission source solvers.

This instability stems from the inability of the Chebychev relations to provide diminishing flux iterates. This is shown in the plot of α and β as a function of dominance ratio in Figure 3. Since there is no point in the region where Chebychev acceleration is allowed that provides negative acceleration factors, the method will always increase the flux. This will eventually lead to instability in the solution if a value higher than that in the limit solution is generated.

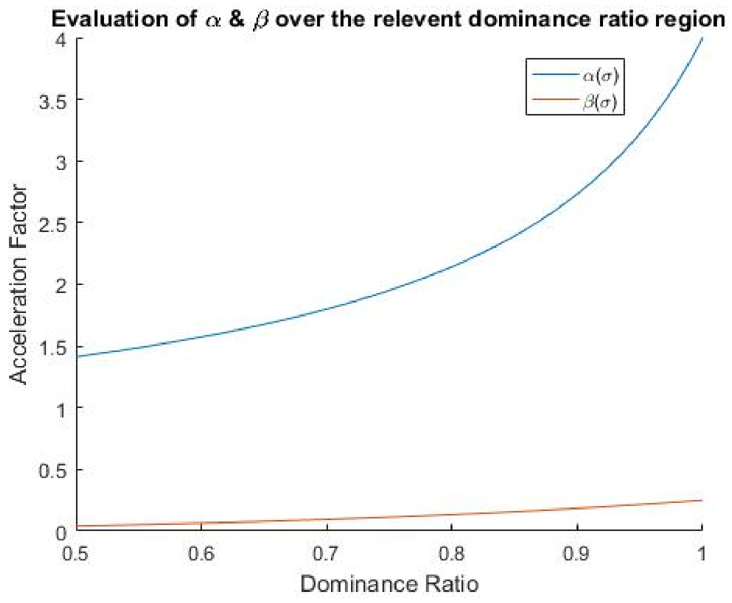

Figure 3: Behavior of Chebychev Acceleration Factors.

This acceleration behavior was the motivating factor behind the addition of the logic to detect “ineffective” cycles. When the system begins to overshoot, the error climbs rapidly. As such, when an increase in error is detected, the code stops the Chebychev iterations and uses power iterations in an attempt to correct the divergence. While not completely effective, this modification allowed the Chebychev routine to remain stable in several test cases where the original algorithm [9] otherwise failed. The tradeoff in iteration count is unfortunate, but worthwhile to increase the odds of convergence.

This instability precludes Chebychev acceleration from being used as a general first-choice algorithm. However, it does allow it to be used to accelerate problems with known solutions for scenarios such as benchmarks or to perform parameter studies on a problem for which it is determined to be stable. This ability to rapidly re-run test cases is critical to maintaining the ability to implement and debug new THOR features related to the evaluation of high-fidelity models.

2.5 Future Work

It is likely that the Chebychev routines will remain unchanged for the near-term. The currently implemented version fulfills the needs of ongoing research and performs sufficiently well. But, as time allows, or as needs change, there are several improvements that could be made to the algorithm. From discussions with [8], it appears that the stability and performance of the Chebychev acceleration method can be dramatically improved through the addition of further pre- and in-acceleration metrics. These metrics can be used to evaluate the readiness of the problem for (further) acceleration and drive modified unaccelerated iterations to better prepare the problem for Chebychev acceleration cycles.

Additionally, based on the discussion presented in the literature review, it would be a significant improvement to THOR’s performance if DSA were implemented. This is a desired feature for THOR and is under consideration for implementation in the coming years.

3 Parallel Domain Decomposition

3.1 Introduction

At the beginning of this project, THOR was a serial code. Given the availability of computing systems from moderate (10s to 100s of processors) to massive (10^5+ processors) in the contemporary scientific computing environment, it seemed obvious that THOR could benefit greatly from at least some degree of parallelism. As discussed in Chapter 1, the desired features of the acceleration precluded energy and spatial domain decomposition. As such, Angular Domain Decomposition (ADD) was pursued. Given that THOR's typical problems utilize angular quadratures of order S_2 through S_{16}, ADD is useful on the medium granularity parallel scale. This is ideal for its current stage of development as it keeps the code flexible enough to run on personal workstations as well as on mid-scale HPC systems.

3.2 Literature Review

3.2.1 Energy Domain Decomposition

As described by [20], energy domain decomposition (EDD) faces two distinct problems. The first is one of scaling and the second one of synchronicity.

Firstly, of the three parallelizable variables in a neutron transport problem, the number of energy groups tends to be one of, if not the, smallest. This severely limits the degree of parallelism that can be achieved. For a typical THOR problem, EDD would be limited to ~10 processors or, in the extreme case a few tens of groups. This lack of distributable work means that the maximum speed-up from the method is inherently limited simply due to the low number of processors which could be used.

While the first issue could be overcome if a given problem demanded many energy groups, the issue of asynchronicity is more difficult to overlook. As discussed by [20] and [21], the EDD method is inherently asynchronous. In other words, the more processors that are involved in the operation, the more iterations that are needed. This creates a situation in which the per-iteration gains are penalized by losses in total iteration count.

EDD works by decomposing the loop over all energy groups, i.e., assigning each group to different processors, just above the inner iteration level in Figure 1. This allows the calculation for each energy group to proceed in parallel with each processor calculating an in-scatter source and performing a one-group, all-angles sweep. Unfortunately, as the in-scatter source is typically composed of both up-scatter and down-scatter components, this means that no energy group will be operating with fully updated scattering source values. Thus, further iterations will need to be performed to converge the in-scatter sources between the various groups.

While this method would be possible to implement into THOR within the constraints laid out in Chapter 1, EDD was deemed sub-optimal for implementation. Given the desire to preserve the existing solver characteristics, a synchronous method is preferred. Additionally, while EDD could offer a moderate speedup, concerns were raised about the limited scalability of the method since THOR is typically used on few-group problems.

3.2.2 Spatial Domain Decomposition

Moving to the other end of the spectrum in scalability is spatial domain decomposition (SDD). In this decomposition, the spatial mesh is decomposed and spread between several processors. Now, rather than being responsible for a portion of the transport iteration, each processor is responsible for an entire transport sweep/solve on a reduced domain. The solutions from the partial domains are communicated and iterated over to produce a converged solution for the entire domain.

A variety of SDD methods exist. However, since this topic will be revisited in Chapter 5, focus will be given to the parallel Gauss-Seidel method in [22]. In this method, the problem domain is subdivided into a set of processor domains, where each processor domain contains some number of red-black striped sub regions. An example of this is shown in Figure 4. Following the decomposition, the solution proceeds in an iterated three step process. Solutions are converged by repeatedly sweeping over the red sub-domains, updating the boundary fluxes, sweeping over the black cells, updating the boundary fluxes, and then performing a global domain convergence check using the most current values for all cells.

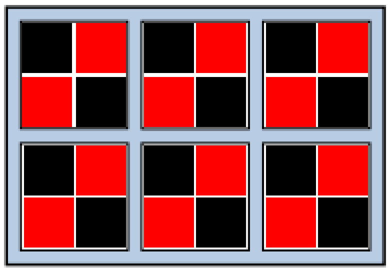

Figure 4: Example PGS-SDD Domain.

Unfortunately, due to the need to propagate data between subdomains, this method is also asynchronous. However, as the size of the spatial mesh is typically very large, thousands to millions of cells, this decomposition provides significant opportunity for parallelization. The results shown in [22] indicate scaling well into the tens of thousands of processors.

As mentioned, alternatives to the above PGS method exist. One class of these focuses on parallelizing the sweep operation by identifying data dependencies between cells and then parallelizing operations over cells with no unresolved dependencies. One of the earliest examples of this is the Koch-Baker Alcouffe (KBA) algorithm, which identifies a number of parallel task pipelines for a sweep and then assigns a processor to one of each of the resulting sets of mesh cells [23]. Significant work has been conducted to further improve this style of spatial parallelism, with research focusing on identifying optimal scheduling algorithms [24], [25], [26]. These algorithms attempt to identify the most parallel configuration of mesh-angle pair sweeping orders. By allowing multiple starting points and mesh-angle pairs simultaneously, efficiency far greater than that of KBA can be achieved. Unfortunately, much of this work is developed for Cartesian meshes, which have inherently known and straightforward dependency graphs. For unstructured grids, dependency graphs can become far more complicated and even include cycles. This makes the development of an optimal scheduler for unstructured meshes a daunting task.

Much like DSA was suited for acceleration, SDD seems ideal for parallelization. Unfortunately, there are several drawbacks that rendered it unsuitable for our purposes. Foremost amongst these is the sheer man-hour cost of implementation. SDD has been repeatedly demonstrated for structured meshes. However, it remains a relatively novel approach on fully unstructured meshes. Implementation of SDD in THOR would form the basis of a major project, likely larger in scope than the one detailed in this thesis.

3.2.3 Angular Domain Decomposition

Having eliminated EDD and SDD, the remaining variable-domain to decompose is angle. In angular domain decomposition (ADD), the mesh sweep is decomposed such that N mesh sweeps occur simultaneously, where N is the number of discrete ordinates in the quadrature set that correspond to explicit boundary conditions.

As [20] discusses, ADD provides a medium level of granularity. It is neither as coarse as EDD nor as fine as SDD, but instead provides for several 10s to 100s of discrete workflows. As shown from results in the GONT code, even an optimally implemented version of ADD is capped at providing significant speedup in the range of 100s of processors [20]. Beyond this point, the serial costs of the iteration begin to dominate and significantly decreasing speedups are observed. Based on this fact, it is not reasonable to expect any implementation of ADD to demonstrate the same scalability as SDD. However, since it is synchronous and more scalable than EDD, it should generally perform better than that method.

ADD's synchronicity stems from the fact that, during the sweep process, each angle of the angular flux matrix is independent from all others. This is true only in Cartesian geometry (no redistribution term) and when at least one boundary condition per dimension is explicit. It is not until the generation of the spatial distribution of the angular moments of the flux that data from multiple angles is needed by a given processor.

Once all angular sweeps have been completed, the ADD algorithm can use an *all-reduce* operation to combine and redistribute the partial results from each ordinate [20]. This provides a much simpler, and typically maximally compact, method for distributing the data. Compared to SDD, where a processor only needs data from its adjacent neighbors, the communication structure required by ADD is direct and typically requires no further logic than that present in standard communication libraries.

Methods do exist to further optimize the communication behavior of the ADD method [27]; however, as that work showed, the cost of communication typically is low enough that effort is better spent improving non-communication performance. Additionally, given the variety of HPC architectures and network interconnects that exist, it is unlikely that any specific modification will provide a general improvement on all systems.

Regardless of these concerns, ADD provides a perfect match to the needs outlined in Chapter 1. It provides a minimally invasive method for significant speedup on the scale of the HPC computer available to the author. Furthermore, since the decomposition is synchronous, it can be considered independently of the existing, verified, code. If the serial and parallel versions return the same result, then the code is unaffected.

3.3 Implementation

Implementation of angular domain decomposition into THOR was a relatively straightforward process, requiring none of the special handling that was seen with Chebychev acceleration. The implementation consists of three parts: the work splitting routine, the parallel section, and the communication phase.

Since the intended system for testing THOR is the Falcon HPC at INL, it was important that the ADD algorithm be designed for message passing systems, not shared memory architectures. Based on this need, the most straightforward path towards implementation was to utilize the Open MPI [28], library. In doing so, each execution of the program now allows for a specific number of processors to be assigned to it. The processors all perform the serial components of the code. However, when a parallel section is reached, the work is divided amongst processors as described in the next section.

3.3.1 Work Splitting Routine

In THOR, due to its use of the level-symmetric quadrature set, the number of angles in a problem is given by

$$N = n(n+2) \tag{19}$$

Where the n is the same quadrature order as given by the S_n in the definition of the discrete ordinates method. This means that in an order-n problem, there are N parallel tasks. Again, this is only true for 3D problems with explicit boundary conditions. To enable fair division of this workload, the optimal number of tasks per processor is calculated at startup based on the number of provided processors via:

$$N_{Optimal} = \left\lceil \frac{N}{p} \right\rceil \tag{20}$$

For executions where the number of processors, p, is a factor of the number of angles, it is a straightforward matter to assign $N_{Optimal}$ tasks to each processor. This provides an even distribution of work and leads to the minimum amount of wasted CPU time during the parallel portion. However, when the number of angles is not an integer multiple of the number of processors, work must be assigned such that there is only a one task difference between the most loaded and least loaded processor.

These needs are met by explicitly mapping a set of $N_{Optimal}$ angles to each processor. A more flexible approach to this mapping would be to simply allow each processor to pull an ordinate from a queue as it completed its previous one. However, it was judged that this method was too prone to race conditions, would require additional communication, and would require burdensome additional control logic. Because of this explicit mapping, a consistently slow processor will continue to negatively impact the solve speed throughout the entire sweep rather than allowing other processors to pick up its slack. However, since the expected use of this capability is the $N = p$ scenario, this effect is negligible.

3.3.2 Parallel Execution

The implementation of a parallel code on a message-passing architecture can be considered as a set of p independent programs that only interact via communication. As such, technically, all aspects of the THOR code now exist in parallel. This means that for each of the p instances, a full copy of all program data is available in local memory. However, the p independent programs may have different values for their variables.

Using the index assigned to each processor and the work splitting routine defined previously, the ADD implementation of THOR can algorithmically select the set of angles to be swept by a given processor. Returning to the flow diagram given in Figure 1, the modified ADD flow diagram is shown in Figure 5. For compactness, the outer iteration portion has been omitted as it remains unchanged.

Given the use of a message-passing paradigm, there exist p processes executing at all stages of the routine, including the serial portions. However, except for the three-way split after the "Split Angular Sweep" stage, all p processors are performing identical calculations. Once the code reaches the split, the work is divided amongst the available processors (a 3-processor solve is shown in Figure 5). Based on this explanation, the work in the serial section is performed p times, but by p processors. This means that execution time of the serial portion remains invariant to changes in p. Contrasting this is the parallel section, where a total of N tasks are split amongst p actors. This means that the expected execution time of the parallel section should decrease as $\frac{1}{p}$. The control flow also reflects the fact that this method is synchronous. From the point where work is assigned, to the point where flux is accumulated, there is no interdependency between the tasks. This means that a problem which converges in i iterations should do so for any value of p.

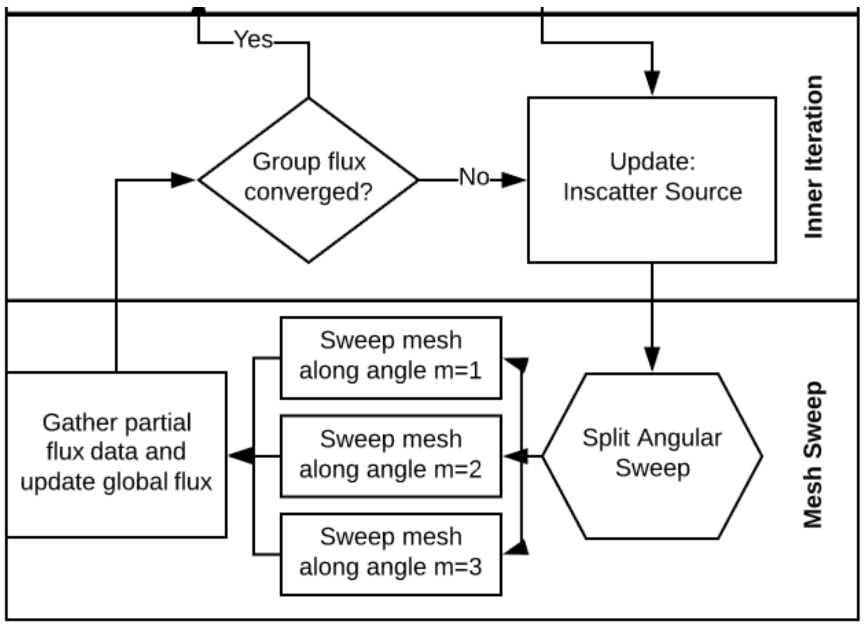
Yes
Group flux converged?
No
Update: Inscatter Source
Inner Iteration
Sweep mesh along angle m=1
Gather partial flux data and update global flux
Sweep mesh along angle m=2
Split Angular Sweep
Mesh Sweep
Sweep mesh along angle m=3

Figure 5: ADD Parallel Inner Iteration Scheme.

3.3.3 Communication

The remaining aspect of the parallel scheme is the recombination of the partial-sum of the angular fluxes. For the discrete ordinates method implemented in THOR, only the flux angular moments need to be retained between iterations. This allows for significant memory savings and provides a unique benefit during ADD communication. During the angular sweeps, rather than maintaining a large vector of angular fluxes, the data can be compacted into the typically much smaller number of angular moments necessary to compute the scattering source in the inner iterations. This is done on the fly by accumulating contributions from each computed angular flux in each cell as it is calculated into the angular moments for that cell. As such the accumulated flux angular moments for group g in cell i can be given as

$$\phi_{g,i}^{lm} = \frac{1}{8}\sum_{n=1}^{N} w_n Y_{lm}^{e}\left(\widehat{\Omega}_n\right) * \psi_{g,i}\left(\widehat{\Omega}_n\right) \tag{21}$$

where w_n is the weight of the n^{th} ordinate and $Y_{lm}^{e}(\hat{\Omega}_n)$ is the even part of the lmth spherical harmonic evaluated at the nth discrete ordinate.

Since, at the end of each processor's set of sweeps, that processor holds partial angular flux data for all cells in a single energy group, it is possible to directly recombine the partial angular fluxes into a complete scalar flux using a summing *MPI_AllReduce* operation. This operation can be visualized as a parallel sum in which each entry of every flux partial angular moment vector is added to its corresponding entry in all other matrices.

At the end of this operation, each of the p independent processes has a fully updated iterate of the flux angular moments matrix and the calculation proceeds to the next energy group or outer iteration as it would normally do in the serial program.

3.4 Results

As was originally presented in [4], the THOR ADD scheme proved a powerful tool for decreasing the run time of a variety of test cases. Chief amongst these was the ATR case previously mentioned in Chapter 1. The ATR benchmark utilized an S_4 quadrature, meaning that a theoretical maximum speedup of up to 24x was available to capitalize on using ADD. Testing showed an actual speedup of ~11x, reducing the total solve time from 14 days to 30 hours. Given the fact that a sizeable portion of the ATR problem's execution time was dedicated to I/O and serial data-manipulation activities, this is an acceptable speedup. Further speedup could likely be gained by capitalizing on the physical distribution of processors and I/O devices on Falcon and minimizing resource contention. However, this type of optimization would only prove useful on problems the size of the ATR benchmark, which are atypical cases.

The first step in assuring that the ADD algorithm had been implemented correctly was to compare the results from a variety of simple THOR test cases for both serial and parallel operation. These tests agreed, to iterative convergence precision, indicating that the ADD method had been implemented correctly and was not having an undue effect on the accuracy of the previously implemented serial solver.

Having demonstrated that the ADD parallelization was working, the next step was to evaluate the quality of its implementation by performing strong and weak scaling tests with increasing p. In strong scaling, the size of the problem is held constant while the number of processors increases. Given the way ADD divides the work comprising the parallel section, the optimal strong scaling curve would follow a $\frac{1}{p}$ trend. In weak scaling, the work per processor is held constant, but the number of processors increases. Here, optimal behavior would be a constant solution time. This would indicate that there is no contention between processors for shared resources, e.g., communication.

3.4.1 Scaling Studies

To evaluate these properties, a trio of homogenous cube problems were created with approximately 10K, 150K, and 400K tetrahedrons. At the time of testing, there did not appear to be significant variation between the per-cell execution times of each problem. As such, scaling plots were only created for the 10k tetrahedrons case. Later work, which will be detailed in Chapter 4, would return to these results and further analyze the per-cell computational costs.

Figure 6 through Figure 8 show the strong and weak scaling of the THOR ADD implementation for the 10k tetrahedrons sample problem at S_{12}.

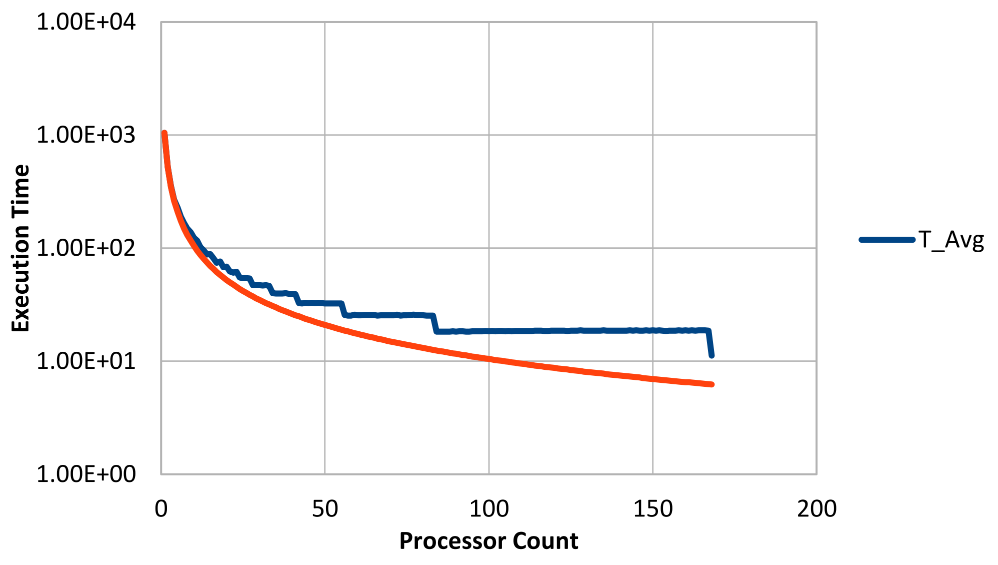

Figure 6: Comparison of Measured and Optimal Times for THOR ADD – Strong Scaling.

Figure 6 provides a strong scaling plot of THOR for the given problem configuration. The orange curve shows the optimal $\frac{1}{p}$ curve for ADD scaling, while the blue curve shows the average measured time over 3 executions of the same case. The stairstep behavior is an artifact of the work-distribution algorithm. Since, when p is not a factor of N, some processors are left idle, the number of processors increases while the amount of parallel action remains constant. This produces a stairstep pattern. As such, the point of closest approach to the optimal curve for any given plateau is the most relevant since it is unlikely that the code would be executed with a sub-optimal number of processors under normal conditions.

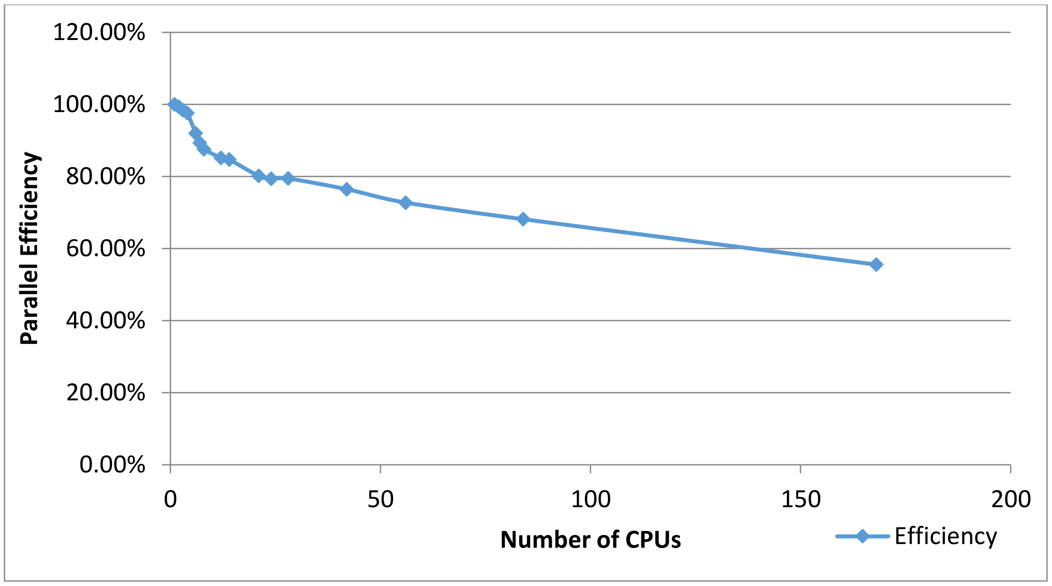

Figure 7: Strong Scaling for Angular Decomposition Problem plotting efficiency (%) versus the number of CPUs for S_{12} level-symmetric quadrature [4].

Figure 7 addresses the stair-stepping by simplifying the analysis to include only the optimal processor counts for each quadrature refinement. From here, it is more obvious that the code efficiency decays smoothly in the range $p > 24$ to an efficiency of ~55% at $N = 168$. While this final efficiency is lower than desired, it does reflect the fact that THOR was originally implemented as a serial code and that many data management operations that require system resources are repeated amongst all processors. Furthermore, this is in line with Amdahl's law, which states that the maximum speedup of a parallel code is bounded by the amount of serial work it must perform. This will be discussed in more detail in Chapter 4.

In the region $p < 24$, there appear to be several inflection points in the scaling behavior. This stems from the physical layout of the Falcon system, which will be addressed in more detail in section 4.3. For now, it is sufficient to note that for this test, there are 3 distinct communication regimes:

1. $p \leq 12$: An operation on this scale will reside on a single CPU of Falcon and not require access to the processor interconnect
2. $12 < p \leq 24$: At this scale, the code will use both processors on a single Falcon server, but not require the network interconnect.

3. $p > 24$: Here, the code resides on multiple servers and will require access to the network interconnect.

The MPI communication costs under these three regimes differ dramatically because of the amount of time it takes to propagate a message across the various interconnect media.

Next, weak scaling results were obtained for the same problem.

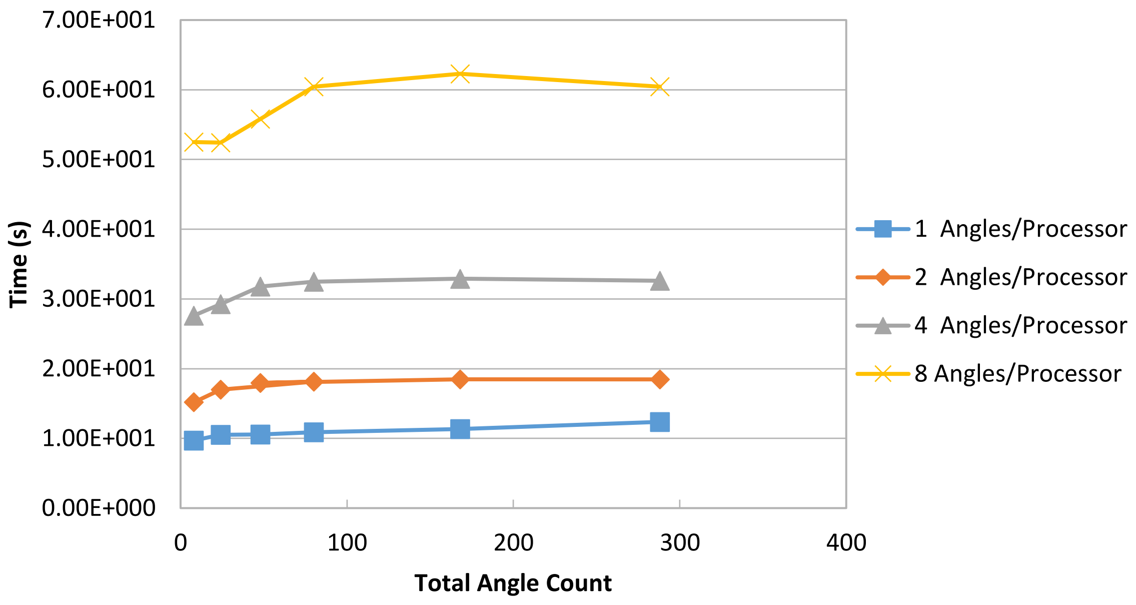

Figure 8: Weak Scaling Results for 10k Tet Mesh with THOR ADD.

Figure 8 provides weak scaling results for 1, 2, 4, and 8 angles per processor for S_2 through S_{16}. Again, the results present significant noise for lower values of p and smoothen out for $p \gg 1$. In the smooth region, the weak scaling traces are nearly flat, indicating a good weak scaling. However, the results are significantly better in the $p < 80$ region. This can again be attributed to changing execution costs as a function of network communication layer used. However, the fact that the issue becomes more severe as the work per processor increases may also indicate that this problem is incurring unexpected serial costs that scale with angle per processor.

3.4.2 Synthetic Communication Modeling

To complement the runtime analysis of THOR ADD, a synthetic model of its communication behavior was also created to provide preliminary confirmation that communication was behaving as expected. This synthetic model implemented the same communication structure as THOR but transmitted a matrix of randomized data. By doing so, this proxy model of the THOR ADD communication could be used to evaluate just the communication performance of the code. This step is designed to confirm that the communication behavior conformed with expected behavior given the Falcon network architecture.

The script, first introduced in [4], which implemented the proxy communication model generates a series of random datasets containing, in order of magnitude steps, between 10 and 10^9 floating point numbers. Then, for varying numbers of processors, the data would be sent using the same *MPI_AllReduce* command employed in THOR and the operation would be timed.

For cases where there was only 1 processor active per server, repetitions were performed simply to generate an average communication time. However, for cases where there were more than 1, but less than 24, active processors per server, these repetitions also served to homogenize the distribution of processors amongst the two CPUs per server.

Figure 9 and Figure 10 show the communication behavior measured from this synthetic benchmark. In both the single processor per server and multiple processors per server cases, the communication behavior can be defined via a two-regime model. First, in the region below 10^5 entries, the communication time is relatively constant. Above that point, the communication time grows linearly with data size. Furthermore, the spacing between the different traces steps up evenly, indicating that costs increase as a function of the depth of the communication tree. This behavior will be explored in much more detail in Chapter 5. For now, it is sufficient to confirm that the synthetic model yields some function for the communication time as

$$T_{comm}\left(N_{Bytes}, p\right) = \log_2(p) * \left(T_{constant} + T_{linear}\left(N_{Bytes}\right)\right) \tag{22}$$

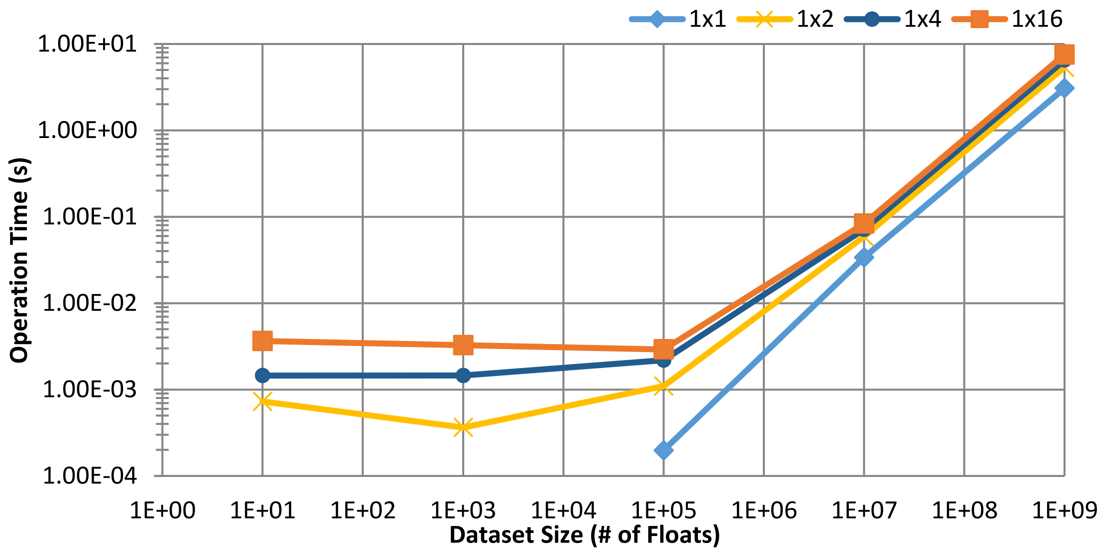

Figure 9: Measured communication time for minimally loaded nodes performing AllReduce on Falcon. Data sets are labeled as [# of active processors per node] x [# of Nodes], from [4].

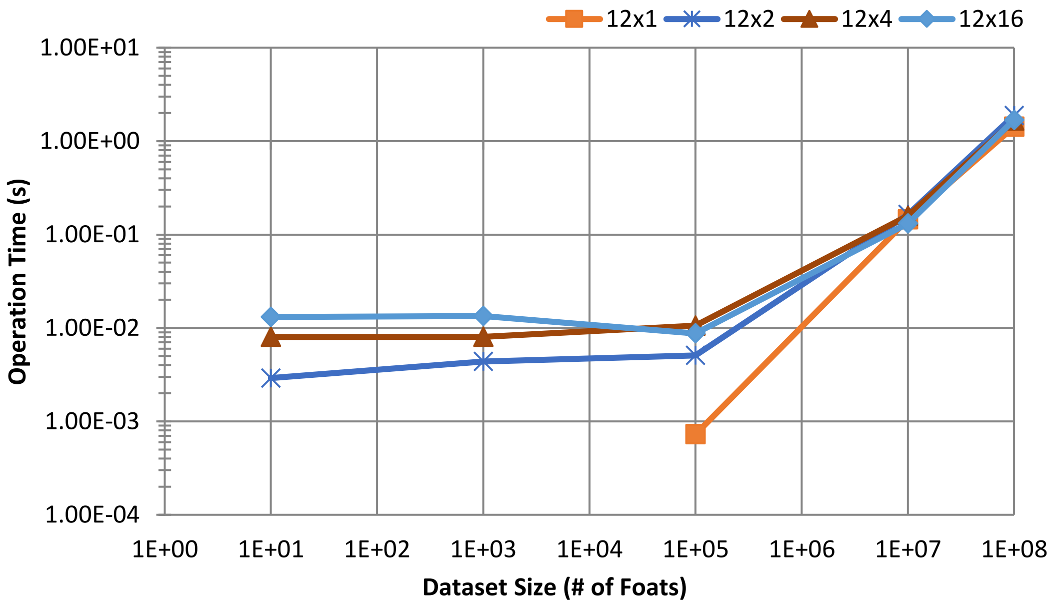

Figure 10: Measured communication time for partially loaded nodes performing AllReduce on Falcon. Data sets are labeled as [# of active processors per node] x [# of Nodes], from [4].

3.5 Future Work

Significant work will be presented in the following chapters that continue the analysis of the THOR ADD implementation as well as of the Falcon communication network. However, at this point, it is important to highlight a few details related to future work on the THOR angular domain decomposition implementation.

As has been discussed so far, the ADD algorithm implemented in THOR is functionally complete. However, since it was retrofitted into an existing serial code, some compromises had to be made. Chief amongst these were modifications made to preserve the behavior of the restart capability and debug I/O writing. As of now, both systems only operate in a serial mode. After all data is recommunicated, all the processors stop and idle while the root node performs the necessary I/O operations. This fix preserves the capability, but it introduces a significant serial overhead into a parallel code. It would be interesting to see future development work targeted at implementing parallel write versions of these routines that could mitigate some of the overhead cost. Note that, for the results presented in this section, these features were disabled entirely.

Looking further ahead, it would be an extremely powerful addition to THOR to see spatial domain decomposition implemented. This capability would give THOR a variety of scaling methods and allow it to be used optimally in the serial, parallel, and massively parallel regimes. However, as discussed, implementing any SDD algorithm in the fully unstructured mesh domain of THOR would prove to be a major undertaking and no doubt require almost a complete rewrite of the existing code.

4 Parallel Performance Modeling

4.1 Introduction

As discussed in the previous chapter, the THOR angular domain decomposition parallelization scheme allowed for the possibility for significant speedups if hardware resources were available. This ability to more fully utilize the computational resources available to the author proved extremely useful in reducing the time required to conduct further analysis.

Yet, while the availability of these speedups was appreciated, questions remained about the effectiveness of the implementation and whether the efficiency curve shown in Figure 7 could be improved. With these questions in mind, it was decided that a parallel performance model (PPM) would be a necessary and powerful tool for conducting any further analysis of the code's multiprocessing capabilities. This model would allow us to predict the runtime for THOR on the Falcon HPC given basic information about the problem parameters, e.g., mesh size, quadrature order, *etc*.

If designed and implemented correctly, this tool can be used as a sanity check to ensure the code behavior was in line with expectations. Additionally, once the PPM is available as a tool, it can be used to benchmark the benefit of modifications to the code as compared to previous versions. This capability will prove critical if THOR undergoes major overhaul or has further parallel functionality implemented in the future.

4.2 Literature Review

4.2.1 Amdahl's Law

While parallel performance models by their nature are specific to a program and machine combination, a language has developed to help express concepts that are shared between all parallel codes and platforms. High-level references to some of these quantities, such as efficiency, appeared in the previous chapter in self-explanatory ways. Now, before delving deeper into parallel performance analysis, some time will be taken to formalize these definitions.

The easiest of these concepts is that of serial and parallel operations. The former of those is a set of operations whose completion time is invariant to the number of processors assigned. The second is a set of operations whose total completion time decreases as some function of the number of assigned processors.

The speedup factor of a code is the measure of this decrease in parallel time and is defined as

$$S(p) = \frac{T(p = 1)}{T(p)} \tag{23}$$

From this naïve definition, given a sufficient number of processors, the speedup factor of the code will approach infinity if we assume that $T(p)$ is monotonically decreasing with p. This would imply that a sufficiently parallel program runs instantaneously. Under realistic conditions, this simple speedup model will always fail. This stems from the fact that no code is truly fully parallel. They all contain some serial section(s) and other parallel section(s). For example, under the THOR ADD scheme, all operations except for the mesh sweep are serial, while the sweep is parallel.

Using this more real-world description, a better definition of speedup can be obtained. This form is known as Amdahl's law and provides the basis for understanding the scalability of parallel codes.

$$S(p) = \frac{T_{serial} + T_{parallel}}{T_{serial} + \frac{T_{parallel}}{p}} \tag{24}$$

As can be seen here, the asymptotic behavior of this expression is quite different from that of Eq. (23). As the number of processors grows to infinity, the speedup becomes limited by the serial portion of the code, with a maximum value of $\frac{T_{serial}+T_{parallel}}{T_{serial}}$.

Based on this, it is clear that no realistic code will obtain perfect scaling for an arbitrarily large number of processors. It is therefore important to be able to quantify the region in which a code is most effective and to understand the marginal cost and benefit of dedicating additional resources to a specific computation.

4.2.2 PPMs for Transport Codes

The development of parallel performance model as evaluative tools for neutron transport codes is not new. Significant amounts of effort have been expended to characterize the performance of existing codes on their primary target machines.

In [29], Azmy constructed a PPM for the P-NT code on the intel iPSC/2 hypercube. The development of this model involves both the implementation of new parallel sweeping functionality as well as a large parameter space search evaluating the performance of the code over a variety of parameter combinations in mesh size, quadrature level, and processor count. The data from this parameter search is used to construct an analytical model of the code performance and a matching quantitative expression.

This model is validated by comparing its predictions to a variety of new code executions and then used to forward predict the capabilities of the code. From this analysis, Azmy concludes that the implemented parallel scheme possesses adequately efficient scaling to be used with 100s of processors.

The developmental technique described in [29] proves quite similar to the one described in this thesis as it is a highly effective method for evaluating the performance of a parallel code of this complexity. As Ref. [29] states, this type of modeling would likely be impossible for a more complex system. The number of uncertainties introduced would be beyond the scope of evaluation via parameter study and would undoubtedly require a degree of statistical modeling. Analysis of this type will be addressed in detail in chapter 5.

Whereas [29] constructs a PPM as a tool to evaluate the predicted performance of a single method on a specific machine and determine its optimal degree of parallelization, [30] uses its parallel performance model as a tool by which to select the optimal communication scheme for a set of given parameters. This PPM is intended to be machine agnostic and instead is designed to highlight the advantages and disadvantages of the presented parallelization and communication schemes.

While the authors reach a similar conclusion regarding angular domain decomposition and spatial domain decomposition as those detailed in the previous chapter, they do so via a quantitative approach that provides an explicit tool by which to evaluate each method. In doing so, this work removes much of the guesswork that goes into evaluating the performance of different parallel schemes before implementation.

In addition to this, [30] evaluates the advantages and disadvantages related to using a global spanning tree communication vs a more customized approach via the bucket algorithm. In this analysis, the authors conclude that, while the global tree communication is easily implemented, it suffers from significant loss of parallel efficiency due to the number of idle processors at each step of the communication. The bucket algorithm attempts to rectify this by more evenly distributing work amongst the processors so as to have as few idle as possible. While it does require more work to implement, it also allows for more parallelism in the communication phase of the solve.

4.3 An Overview of the Falcon HPC System

While some details of the Falcon High Performance Computer at Idaho National Lab have been presented so far, the following analysis will require a far more complete understanding of its architecture and capabilities. As such, this section aims to provide details that will be relevant in the coming pages. Specific details regarding Falcon's physical and network architecture were gathered via communication with [31].

When this research was conducted, Falcon was a SGI built, ~25,000 processor general computing system. While the system is not entirely homogenous, it does contain a homogenous subset of ~20,000 processors. It utilizes an InfiniBand interconnect with a stated bandwidth of 56Gbit/s. Rather confusingly, these ~25k cores are embedded into what SGI refers to as an "Enhanced 7D Hyper-Cube" by abstracting the nodes of the hypercube such that each node represents a cluster of system hardware. As it is a 7D hypercube, no node is more than seven communication hops away from any other node in the system. However, due to the sub-node arrangement of the system, the actual number of hops to any given processor is the number of node hops, plus an additional node-to-server hop.

For the purposes of the results detailed in this chapter and the next, all code was executed on the 20K processor homogenous subset. However, as a result of the topology of Falcon, this means that communication effects are influenced by the full 25k processor, 7D system.

4.3.1 Physical Layout of Falcon

This section will provide a brief overview of the physical layout of the Falcon HPC. This will also serve to introduce key terminology for the system.

Starting from the processor level and moving up, the 19,872-processor homogenous portion of Falcon is arranged as detailed in Table 2. Each server contains 128GB of RAM to be shared by all processors in the server.

Table 2: Summary of Falcon Hardware Layout.

	Region 1	Region 2
Processors	19,872	5,184
Processors per CPU	12	16
CPUs per Server	2	2
Servers per Chassis	9	9
Chassis	92	18

4.3.2 Network Topology of Falcon

From a topology perspective, Falcon technically implements a heterogeneous, partially filled 7D hypercube.

The system is heterogeneous since, because of its “enhanced” nature, the bandwidth between various dimensions of the hypercube is not constant. This modification by SGI is intended to provide additional bandwidth to codes which fully exercise Falcons computational resources.

As there are not 128 accessible nodes, the system does not provide a complete 7D hypercube. This stems from the fact that, without special permission, one node is reserved for the management and scheduling unit and the two hardware regions (12 and 16 processors/core) are kept distinct.

Each pair of nodes of the Falcon hypercube is defined to be a hardware module referred to as an “IRU”, or individual rack unit. These hardware clusters consist of a chassis / router pair with a 4x link to each of its neighbors. Each IRU is connected to 7 others to form a 3D unit with homogenous interconnect bandwidth. These 3D units are then interconnected with heterogeneous links to form the 4th through 7th dimensions of the hypercube. A pictographic representation is available in Appendix A.

Ignoring the higher dimensions, it is important to note that, even within the homogenous 3D space, a single IRU consists of 18 servers sharing a 4x link, where each server is then distributing its access to the link amongst 24 processors.

4.3.3 Other Falcon Notes

Having discussed the hardware and network specifics of Falcon, the last remaining item is the job management software. Like most HPC systems, Falcon implements a queue manager. Under optimal conditions, this manager attempts to allocate jobs to the most compact, in terms of network distance, set of processors that are available at the time of execution. This is intended to minimize resource contention caused by other codes operating simultaneously on the machine.

However, under realistic conditions, the HPC system utilization level is too high to guarantee locality, i.e., a diameter of $\log_2 p$. When this occurs, the servers assigned to a user can extend beyond the minimal hypercube of dimension $\log_2 p$. From a code execution perspective, this makes little difference. However, from a performance modeling perspective, this implies that the cost of any given communication between two participating processors is, to some degree, random and can experience dramatically varying levels of network latency and contention. The effects of this behavior will be explored in Chapter 5.

4.4 Complications Stemming from the THOR Cell Solver

As most of this thesis has addressed modifications to THOR designed to encapsulate, but not modify, the existing solver routines, there has not yet been a discussion of one unique feature of the code's cell solving kernel. This kernel forms the lowest layer of the iterative solver operation and is designed to operate on a single cell and along a single discrete ordinate at a time. This kernel uses inbound angular fluxes to the cell-angle combination, the cell's dimensions and nuclear data corresponding to the material occupying its volume, to compute that cell's flux angular moments and outbound angular fluxes.

In a structured mesh, there is a degree of regularity to the orientation of cells and thus, the sweeping pattern along these cells is mapped to its Cartesian indices depending on the octant of the corresponding discrete ordinate. This leads to a unique (or near unique) set of cells solving algorithms that need to be implemented. For example, in a cubic mesh, the orientation of the cell faces and the number of up/downstream neighbors is always known.

In an unstructured mesh, these benefits can be lost. Looking at the tetrahedral meshes used by THOR, it is easy to observe that tetrahedrons may have any orientation with respect to a given discrete ordinate in 3D space. This significantly complicates the process of identifying the upstream and downstream values which are inherently known in the structured domain.

Rather than designing a dynamic system to address this problem, [3] details the implementation of a set of six cases that the cell solver selects from. This allows for the random orientation of mesh cells to be reduced into a set of six solver configurations. Once a configuration is identified, the mesh cell, for the purposes of the solve calculation, is split into several sub-tetrahedrons, known as *canonical tetrahedrons*. These six decompositions are shown in Figure 11.

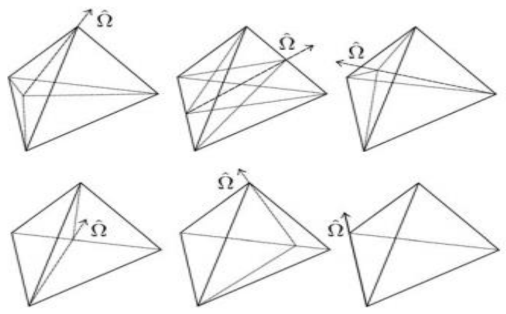

Figure 11: Possible Configurations for Canonical Tet Decomposition, from [32].

Since these decompositions lie along known axes, the cost of the cell solve is greatly reduced compared to the dynamic version. However, since the cell solver routine is opaque to the rest of the code, a degree of uncertainty is introduced. In each of the six decompositions, a different number of canonical tetrahedrons can be generated. This leads to each cell having a variable solution cost based on its orientation with respect to the discrete ordinate. This cost can range from 2x to 4x the cost of a canonical cell solve and cannot be determined during normal operation of the code. Since these decompositions are related to both the mesh configuration and quadrature set, they will have to be considered in the development of the PPM.

4.5 Methodology

As discussed in previous chapters, THOR implements a variety of options, ranging from different acceleration techniques to entirely different solvers. Producing a single performance model that covers all possible configurations is an overwhelming task. As such, the model was designed to evaluate a limited subset of THOR's capabilities.

The PPM, as developed, is intended to provide assessment of the cost of a single unaccelerated, power iteration. This choice reflects the fact that the power iteration solver is the most frequently used and, consequently, is the most mature. The implemented accelerations were disregarded since, as outer accelerations with minimal computational costs, they would have little influence on the single iteration cost. Additionally, while THOR is capable of multi-group and arbitrary order spatial evaluations, the model is designed to capture mono-energetic problems with zeroth order local spatial expansion of flux variables. The limitation on energy group count was intended to shrink the parameter space for analysis and stems from the reasonable assumption that, for a reasonable number of groups, the cost of a G group problem scales approximately linearly with G. However, note that as G grows very large, the rate of increase will become super-linear.   The decision to limit spatial expansion order reflects the fact that the zeroth order is a commonly used expansion order for current THOR exercises. If this changes in the future, it would be a relatively simple matter to retrofit the performance model to include higher spatial orders. The final simplification that was made was to disregard the cost of setup and wrap-up times. For small problems, these times are entirely negligible. For larger problems, such as the $4 \cdot 10^9$ degrees of freedom ATR problem described previously, these times can be significant when compared to the single iteration time. In this case, the total cost of setup and wrap-up was nearly equal to the cost of a single outer iteration. However, when the full problem is run, even in parallel, the cost of these operations in insignificant compared to the total runtime.

To help mitigate the long-term penalties of excluding THOR features and capabilities, the model is designed as a modular construct. In this fashion, each major component of the code is composed of the time it takes to perform its tasks plus the time taken by its child tasks. By constructing it this way, future modification to the code can be added or substituted into the model without disrupting the evaluation of the other functional components. This nesting is shown schematically in Figure 12.

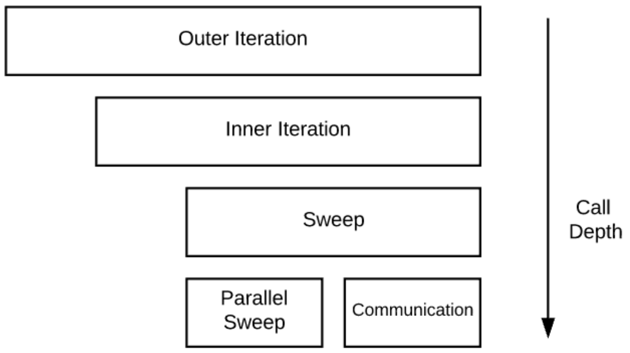

Figure 12: Hierarchical Timing Model Design, adapted from [33].

The second benefit of this design is that the most hardware-dependent portion of the timing model, communication, is isolated from the rest. This allows the timing model to be converted to different communication techniques rather easily and, more interestingly, allows it to be transferred to new hardware with little effort. This process of moving to new hardware would involve modifying the communication model and updating the hardware timing constants for the other levels via a new, possibly less thorough, set of test cases.

Based on this model design, the solver algorithm, and the description of Falcon provided previously, a preliminary model, originally given in [33], was developed for validation against measured execution times:

$$T_{solve} = \tau_{\text{const}} + N_{mesh} * (\rho * [\tau_{outer} + \tau_{inner}] + \tau_{sweep} + \left\lceil \frac{n*(n+2)}{p} \right\rceil * \tau_{parallel}) + f(\tau_{comms}, p), \tag{25}$$

where, as before, n is the quadrature order, p is the processor count, ρ is the number of angular moments, and $N_{mesh}$is the mesh size in number of cells. The various τ coefficients represent the system specific time constants for each routine.

As expected, this preliminary model shows the parallelism implemented via ADD. Here, all operations, including some portion of the sweep routine work, are serial except for the parallel section of the sweep and the communication instructions. It is also important to note that the scaling for the parallel portion does not perfectly reflect the work distribution function given in Chapter 4 in that it assumes a number of processors that divides the number of discrete ordinates with fixed global boundary conditions. Formally the scaling is given by $\left\lceil \frac{n(n+2)}{p} \right\rceil$ to account for sub-optimal choices of the number of processors, where we assumed all fixed global boundary conditions. However, as discussed previously, it was considered unlikely that a user would assign non-optimal numbers of processors to a given problem.

Finally, there is the communication function provided in Eq. (25). Based on the use of MPI's *AllReduce* spanning tree function and an understanding of the data transmitted, a preliminary system communication model can be expressed.

$$f(\tau_{comms}, p) = 2 * \log_2(p) * \left(\tau_{comms} + \frac{N_{mesh} * \rho}{\beta}\right), \tag{26}$$

where β is the system bandwidth. The 2x multiplier is used to indicate that the communication operation is two-way. There is both a send and receive action that must occur for the operation to complete. The $\log_2(p)$ term represents the time taken to establish a spanning tree between p processors, while the $\frac{N_{mesh}*\rho}{\beta}$ term represents the time taken to transmit the data over a network connection whose bandwidth is β. This communication model is partially confirmed by the results from the synthetic model presented in section 3.4.2. By taking the measured time plots generated there and combining them with the concepts expressed in the communication function, Eq. (26), one can easily identify the two regimes, tree building dominated and data transfer dominated.

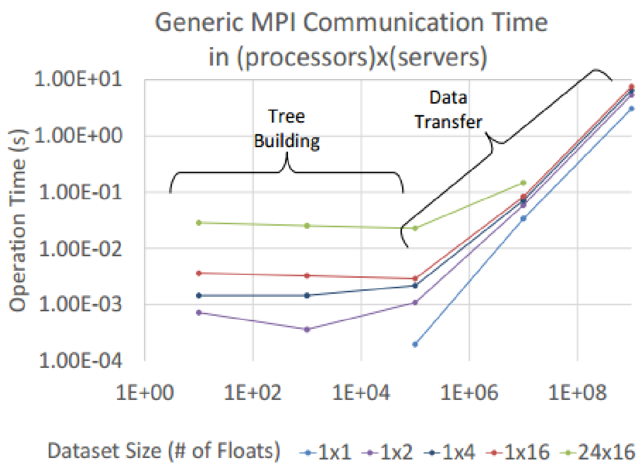

Figure 13: Updated representation of AllReduce Behavior on Falcon, adapted from [33].

4.6 Results and Model Validation

4.6.1 Evaluating the Influence of Canonical Tetrahedron Subdivision

In order to model the influence of the canonical tetrahedron subdivision on the computation's cost described earlier in this chapter, it was necessary to evaluate the average number of canonical tetrahedrons per cell over a wide range of problem configurations. To this end, five model problems were selected to quantify this uncertainty. The first three were the homogenous cube problems used in the testing of the ADD algorithm. This presents simple geometry cases with ~10K, ~150K, and ~400K tetrahedrons. The fourth model, designed to test very small problems, was a coarse mesh model of the Godiva benchmark [34] comprised of 274 cells. Finally, moving to the other end of the spectrum an unfolded version (i.e. full geometry with vacuum boundaries) of the C5G7 benchmark [35] was selected to test larger problems with more complex configurations. This problem has ~20 million mesh cells.

For each case, problems were run with a sequence of quadrature refinements, keeping track of the number of canonical splits of each tetrahedron-angle pair during the sweep, then calculating the average number of canonical tetrahedrons per cell. These results are presented in the following tables.

For the simple cube geometry, cases were run at S_2, S_4, S_8, S_{12}, and S_{16}. Table 3 summarizes the smallest, middle, and largest of these sets. As is evident from the data, there is essentially no variation in the number of tetrahedrons, which remains steadily at ~3.66 canonical tetrahedrons per mesh tetrahedron. This result stems from the fact that the decompositions which produce 3 and 4 sub-cells are the most likely as they have the least restrictive orientation constraints on the parent cell.

While the average number of canonical tetrahedrons remains essentially constant, there also appears to be a miniscule upward drift as the quadrature order increases. However, as long as the standard deviation of this change remains at such a small level, it can be safely disregarded for all relevant problem sizes. This means that the parallel performance model can disregard the variable cost of canonical cell splitting without significantly impacting its predictive capability.

Table 3: Simple Cube Test - Canonical Tet Variation [33].

# Angles	# Cells	Avg. Subcells	Std. Dev
8	8,859	3.66836	
80	8,859	3.66836	0
288	8,859	3.66836	
8	151,562	3.66812	
80	151,562	3.66814	1.53E-05
288	151,562	3.66815	
8	194,332	3.66802	
80	194,332	3.66804	1.15E-05
288	194,332	3.66804	
8	426,885	3.66813	
80	426,885	3.66820	4.73E-05
288	426,885	3.66822	

Next, the Godiva benchmark was used to provide yet another independent mesh, but this time on the smaller side of the spectrum. Much like the results for the smallest simple cube test, Godiva shows no significant variation in the number of sub-tetrahedrons and yields the same average number (Table 4). Based on the upward trend in the C5G7 results (Table 5) it is possible that the average number of sub-tetrahedrons for that problem will also asymptote to ~3.66. This question is left for later investigation.

Table 4: Godiva - Canonical Tet Variation [33].

# Angles	Mesh Size	Avg. Subcells	ΔAvg
8	274	3.66836	
80	274	3.66836	0
288	274	3.66836	

While the results of the simple cube and Godiva tests were promising, they did raise several concerns regarding generality of the conclusion. First was the lack of material boundaries. It is possible that, in a sufficiently large volume, the cell orientations average out. However, if material boundaries are introduced, these will force cell orientation to conform to some constraints. The second concern was that, by performing uniform refinements of the subcubes, the results from different refinement levels were not independent of one another.

The C5G7 model addresses both concerns by introducing material boundaries and a new mesh. Unfortunately, due to its size, only S_2 and S_4 quadratures could be evaluated. As Table 5 shows, change in the average number of canonical tetrahedrons is much more significant for this problem. However, it is still relatively small in terms of the other uncertainties present in the PPM, especially those related to communication. As such, it can still be disregarded. But, it may be necessary to revisit this effect in the future if a specific problem configuration is found which exhibits the same properties in a more pronounced fashion.

Table 5: C5G7 - Canonical Tet Variation [33].

# Angles	Mesh Size	Avg. Subcells	ΔAvg
8	20,617,414	3.618	
24	20,617,414	3.640	0.022

Based on the data collected from these 5 cases, it was determined that the number of sub-tetrahedrons created in each THOR problem was stable enough that it could be assumed constant in the model.

4.6.2 The Communication Model

The next major piece of the timing model to evaluate was the communication section. As discussed previously, Falcon's architecture introduces a considerable number of runtime effects on this component of execution time, namely processor locality, contention, and bandwidth. Unfortunately, for a general user, the effective value of these parameters is completely opaque. As such, to at least represent some portion of the possible combinations, all timing measurements for the communication component were conducted by repeating the exact same case multiple times then averaging the measured times.

First, to confirm that the behavior of THOR conformed to that of the synthetic communication model presented in section 3.4.2, the simple cube problem was run repeatedly on Falcon with allocations of size 2^0 through 2^6 processors. The communication times were tabulated and, for the repetitions within each case, standard deviations were generated. The resulting data aligned quite well with the synthetic model results for a similar communication data size, with communication times in the range of 10^{-2} to 10^{-3}s for the smallest case. The standard deviations within these cases were typically on the order of 10^{-4} or smaller. However, it was observed that, infrequently, a specific case would consume far more time than expected, leading to deviations on the order 5 to 10% of the elapsed time. As these effects would disappear under repeated runs at different times, the spikes are attributed to system level effects beyond the scope of this model and will be disregarded. This does indicate that the general load level of the HPC system can have some effect on the performance of a specific code running on it. While relatively minor for this case, the next chapter will explore a far more elaborate case where the impact is more significant.

Next, it is important to address the degree of variation between runs of the same problem on different allocations. This behavior is not reflected in the model, but instead represents an irreducible amount of error that will be present and can be demonstrated in the following plots. Noting that the data at each point was collected on a different randomly allocated sub-system, Figure 14 shows the magnitude of the communication time and how its behavior can vary as a function of allocation size. In this plot, the simple cube problem was run using an S_8 quadrature for a variety of processor counts. In the $p = 288$ case, a system allocation of 288 processors was requested, regardless of the number used by THOR. In the other case, the allocation size was matched to the number of utilized processors. While the relative cost of each communication is likely a result of other system effects, the difference in trend can be explained as a function of the processor distribution in the allocation. For the orange trend, the processors will always form the same spanning tree on the hypercube since the number of available processors is the same as the number of tasked processors. However, in the $p = 288$ case, there are far more configurations for the processors to fall into since many different spanning trees of size $p < 288$ can be constructed from a set of 288 processors. As a result, between various executions of the code, the locality of the processors can change much more dramatically as the nodes in the spanning tree change positions in network-space.

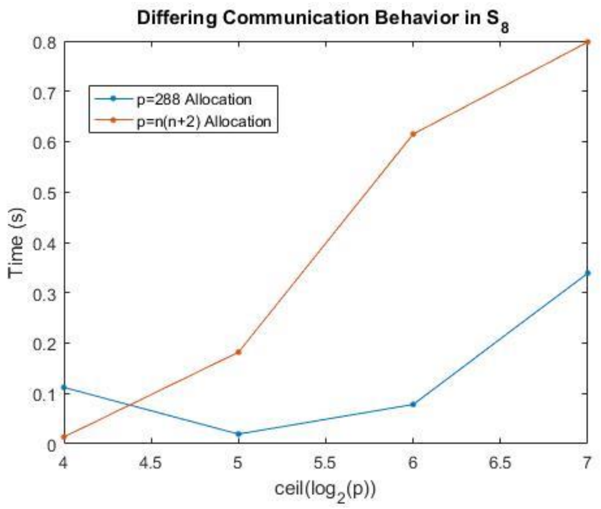

Figure 14: Differing communication trends for p=288 and p=n*(*n+*2)* for S_8, adapted from [33].

To emphasize this point, Figure 15 shows the same experiment for a S_{12} problem. Here, the optimal configuration and the $p = 288$ configuration form spanning trees with only a single-level difference. As such, there are significantly fewer substantively different configurations for the processors to fall into when an oversized allocation is requested. These results further emphasize the point expressed in the discussion of efficiency – parallel codes should be utilized with a logical number of assigned processors, both in the execution stage and during the resource provisioning.

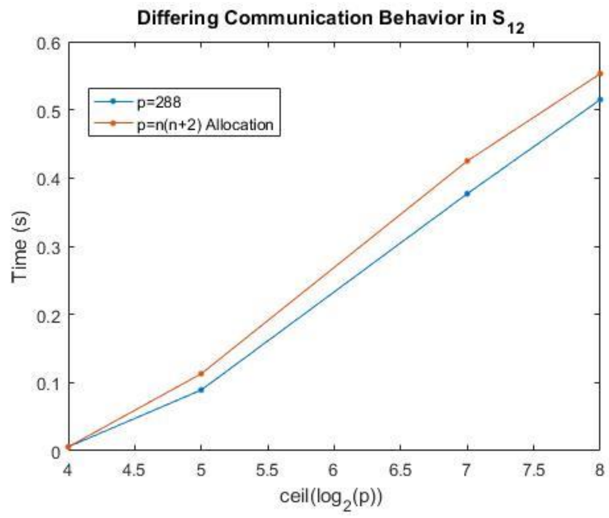

Figure 15: Differing communication trends for p=288 and p=n*(*n+*2)* for S_{12}, adapted from [33].

Using only data collected on appropriate processor allocations, a fit for the measured communication data was performed. As expected, this fit demonstrates a clear $\log_2(p)$ profile and, as such, conforms to the preliminary model. However as shown in Figure 16, significant outliers still exist. But, these outliers are most significant at very low values of p, where the total time contribution from communication is at its least.

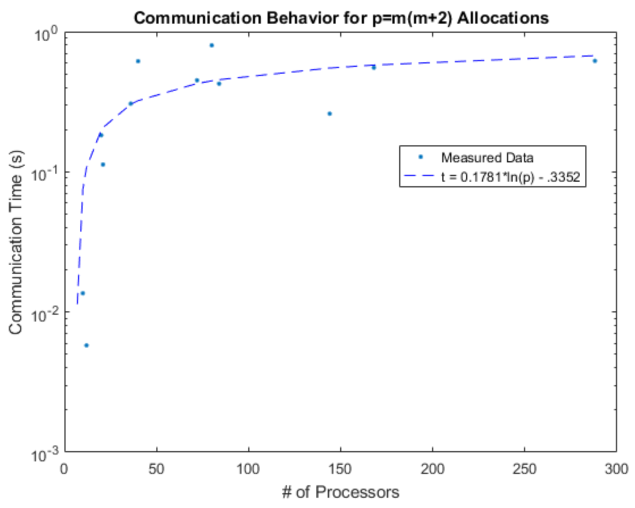

Figure 16: THOR/HPC communication time fit, adapted from [33].

4.6.3 The Parallel Sweep Model

The last remaining item to complete the model is the parallel sweep section. As discussed previously, this is the only parallel section of the code and should demonstrate a clear $\frac{1}{p}$ time trend for $p \leq n(n+2)$. Above that point, the additional processors will contribute no speedup and measured times will flatten out. To confirm that this trend was present, the smallest refinement of the simple cube test was again used with $n = 2, 4, 6$ and $p = 1, 2, 4, 6, 64$. As shown in Figure 17, these assumptions are confirmed. Results from all three quadrature refinements clearly follow the expected trend. The dashed regions indicate those points where the constraint of $p \leq n(n+2)$ was met. Note that the more this constraint is violated, the farther the times drift from their expected trend. Additionally, since, in the sweep, the total time scales linearly with the number of angles (N, not n) one should expect the slope of the three lines to be equal. This trend is confirmed by the fitting functions presented in the figure.

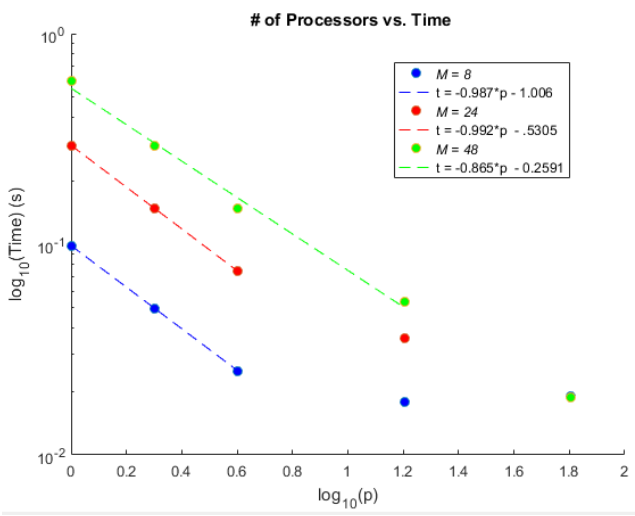

Figure 17: 1/p relationship between processor count and parallel sweep time, *adapted* from [33].

By manipulating the data presented in Figure 17, another confirmatory plot can be produced. Figure 18 shows the measured time as a function of total number of angles for $p = 1, 4$. Here, the ideal result would be one in which the slope of the $p = 4$ curve was exactly $\frac{1}{4}$ of the slope of the $p = 1$ curve. As the fitting functions show, this is indeed the case. From these two metrics, it is reasonable to assume that the ADD algorithm has been implemented properly and will exhibit the correct behavior, at least within this section of the code.

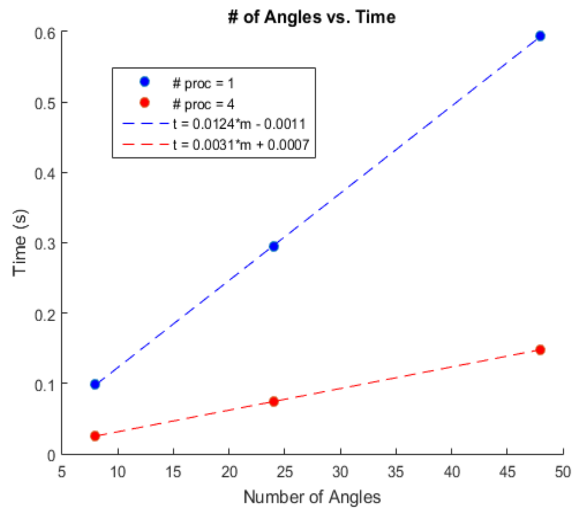

Figure 18: Linear relation between time and number of angles for a fixed value of p, adapted from [33].

Working with this collected data, it should now be possible to measure a quantity known as the grind time, which for this model, represents the execution time per cell, per angle:

$$T_{grind} = \frac{time(N, N_{mesh})}{N * N_{mesh}} \tag{27}$$

A grind time of $\sim 2.30 \cdot 10^{-6}$ was found for $p = 1$ and one of $\sim 2.33 \cdot 10^{-6}$ was evaluated for the $p = 4$ cases. This difference of ~1% is greater than the runtime variance of ~0.5% and will be accounted for in the next section.

4.6.4 Evaluating the Model

With the major components of the sweep evaluated, it is now possible to produce a unified model for the parallel sweep time and evaluate it.

First, the model was used as a self-prediction tool. Predictions of the sweep times for the cases described above were generated using the average of the two grind times and compared against measured run times. These results are tabulated below and show an acceptable level of agreement between the predicted and actual measures.

Table 6: Percent Error for Interpolation Cases, from [33].

Case p=4	Meas. Time (s)	Model Time (s)	Difference (%)
S2, 500k tet	1.53	1.50	1.9%
S4, 500k tet	4.61	4.51	2.3%
S6, 500k tet	9.20	9.02	2.1%

Now that the model had been determined to be self-consistent, a second, larger set of test cases was selected. These cases, by design, were selected to require the timing model to extrapolate well beyond the scope of its design cases. In order to yield extrapolation in both quadrature order and cell size, S_2 through S_{16} were evaluated for the largest (~400K tetrahedrons) mesh of the simple cube geometry. These results are shown below with measured data as points and model trends as lines.

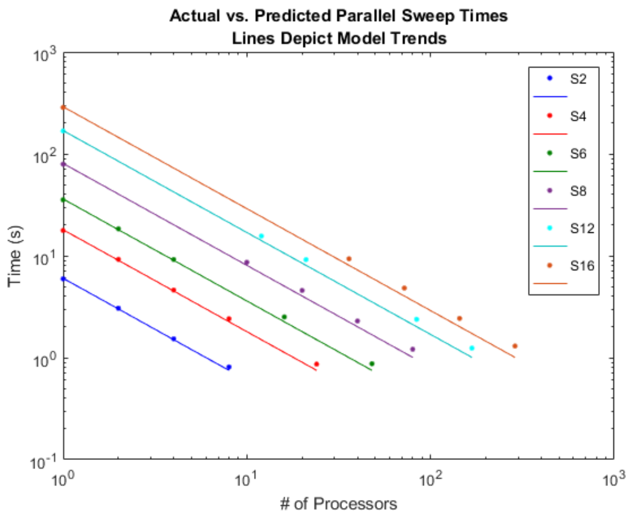

Figure 19: 400k tetrahedrons mesh model predictions vs measured, adapted from [33].

In Figure 19, the model produces increasingly poor results as the number of processors increases. This effect appears to be independent of the number of angles as the error at every discrete value of p is approximately the same. Regardless, even by $p = 288$, the results were far too poor to use in any effective fashion. Error by that point had climbed to ~25%.

This increase in error stems from the same root cause as the ~1% error measured in the grind times in the previous section. There exists a processor-count dependent factor which has not yet been accounted for that has a minute, but cumulative effect on system performance. This effect is captured and fitted in Figure 20, which shows that the effect increases monotonically with p. Based on this result, it is likely that this growth effect comes not from THOR, but from some lower-level aspect of the parallelization scheme, such as increased memory contention. While not explored in this work, it is possible that this increase stems from that same source as the slowdowns discussed in the next chapter. This would further indicate that latency-based efficiency losses are present even at the low 100s of processors.

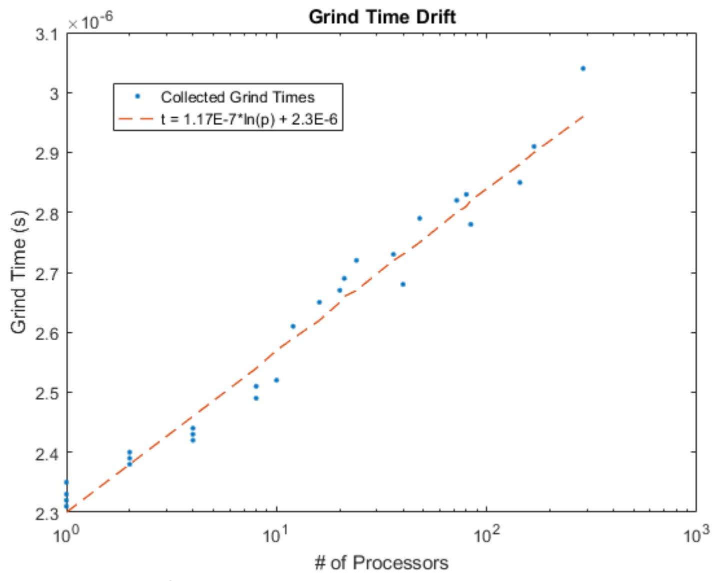

Figure 20: Evaluation of processor count dependence in grind time, adapted from [33].

Recreating Figure 19 with this correction factor added to the grind time calculation yielded a much more accurate sequence of predictions, with errors of only ~2-3%. This is an acceptably low error range and indicates that the model is successful in extrapolating out the performance of THOR.

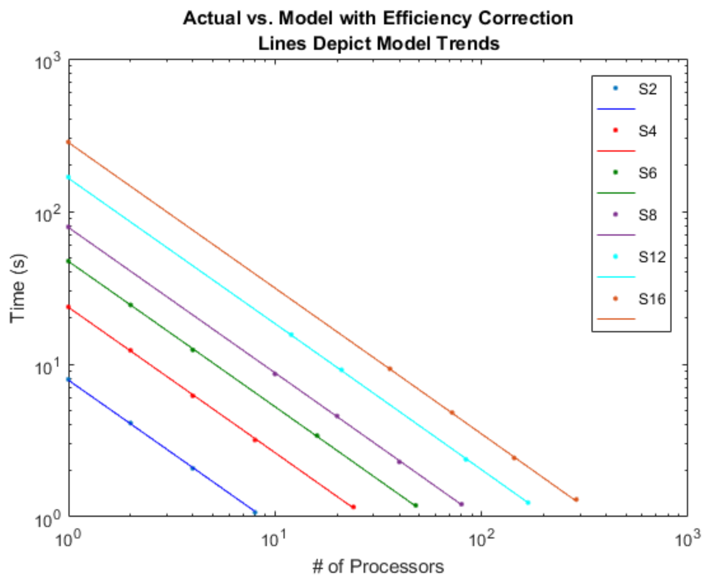

Figure 21: Amended Model Cases, adapted from [33].

However, so far, the model has only been used to predict the results of the simple cube test. Much like in the evaluation of canonical tetrahedrons division, only using a single mesh increases the chance that mesh dependent effects are not properly accounted for.

To correct this, two further meshes were analyzed. The first was a ~15K cell version of the Takeda-IV [36] benchmark and the second was a ~2K cell Godiva benchmark. As expected, results for these cases were not as good as those for the simple cube test. However, they were still relatively accurate. These results are tabulated below. For the Takeda-IV problem, the average error is ~5%, which is only slightly higher than the simple cube error and still quite applicable for typical use. However, the results for the Godiva model are considerably worse. While not as bad as the uncorrected sweep model, these errors are on the order of 10%. But, as this is the smallest mesh evaluated and all of the errors are in the form of overprediction, this may be explained by hardware effects, such as caching, that benefit from the decreased data volume.

Table 7: Actual vs. Model Results for Takeda-IV & Godiva Cases.

Case		**Time (s)**	**Model Time (s)**	**Difference (%)**
Godiva, 3k tet				
S8	*p*=40	1.17E-02	1.36E-02	16%
	p=80	5.88E-03	6.78E-03	15%
S12	*p*=84	1.24E-02	1.36E-02	10%
	p=168	6.28E-03	6.78E-03	8%
S16	*p*=144	1.24E-02	1.36E-02	10%
	p=288	6.40E-03	6.79E-03	6%
Takeda, 15k tet				
S8	*p*=40	6.71E-02	7.28E-02	9%
	p=80	3.44E-02	3.65E-02	6%
S12	*p*=84	6.80E-02	7.29E-02	7%
	p=168	3.55E-02	3.65E-02	3%
S16	*p*=144	7.04E-02	7.30E-02	4%
	p=288	3.70E-02	3.65E-02	1%

4.6.5 The Unified THOR Parallel Performance Model

The last step was to combine the various model components. An observant reader will notice that no discussion has been dedicated to evaluating the inner and outer sweep timing constants. This is not an oversight. For all of the cases described in this chapter, the computational time of these routines was negligible. The non-computational time in those routines is dedicated to updating the screen to keep the user apprised of execution status. Still, both of these were negligibly small, so the decision was made to neglect them. It is likely that this will remain true for nearly all THOR-relevant problem configurations as the accumulated cost of sweeping and communicating per iteration should grow far faster than that of the iteration control logic.

Figure 22 presents a final evaluation of the model on the 500K tetrahedrons simple cube problem using the form described in equation (28). As the number of processors increases, one can observe a slight leveling off of the trends. This begins to occur as the sweep cost and the communication cost become increasingly comparable. This asymptote represents the speedup ceiling for the problem.

$$T_{solve} \approx \frac{n*(n+2)}{p} * N_{cells} * T_{grind}(p) + f(comm) \tag{28}$$

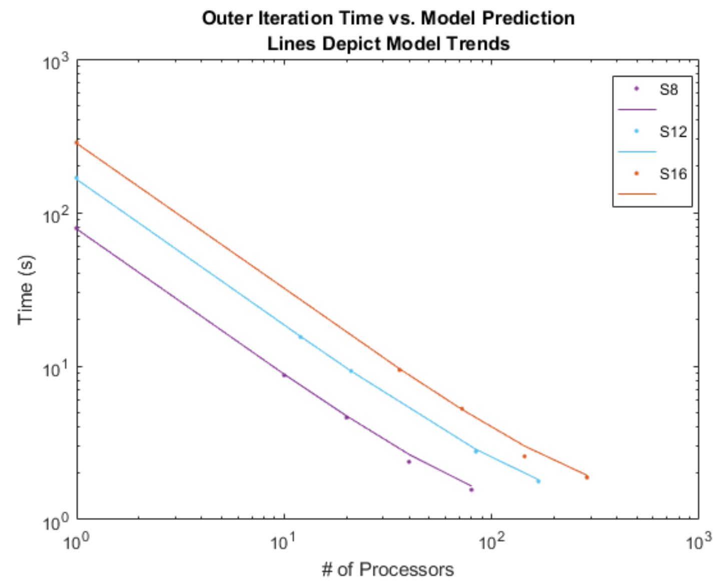

Figure 22: Final Model Predictions for Simple Cube, adapted from [33].

4.6.6 Future Work

As established, the THOR PPM provides an accurate tool for the evaluation of code run time. This allows it to be used as both a predictive and evaluation tool. In the first case, it may be used to estimate the total time an iteration will take based on the problem parameters. With that estimate, and knowledge of the dominance ratio and convergence limits, the total runtime may also be estimated. As an evaluation tool, the PPM can be used to detect if modifications to the code have introduced unexpected inefficiencies or to determine the relative performance of different problem cases.

In its current state, the PPM is limited to the power iteration solver and zeroth spatial expansion order. As mentioned before, while it does not explicitly handle multi-group problems, the scaling should be linear in g. An obvious source of future improvements would be to implement these missing parameters. That would allow the model to fully predict the behavior of the unaccelerated power iteration solver. This idea could be further extended by creating modular additions for the fission source and Chebychev acceleration routines. Longer term improvements could include modeling the behavior of THOR's JFNK solver or the behavior of yet unimplemented functionality.

The more fully the PPM can characterize THOR, the more application it will have as a decision-making tool in the software engineering process. Much like in the work described by [30], a PPM for each of the major THOR components could help a user to determine the optimal solver for a given problem configuration on hardware with known performance parameters.

5 Parallel Communication Effects

5.1 Introduction

In this part of our work we move away from analyzing the behavior of the THOR code and will, instead, provide an in-depth study of the behavior of communication cost on a multi-user multiprocessor approaching the massively parallel regime like the Falcon HPC. As we describe in later sections in more detail, this analysis was motivated by a combination of factors stemming from questions about the scaling performance of the Cartesian mesh, S_N code, PIDOTS, written by Zerr [22]. In that code, unexpectedly substantial decrease in parallel efficiency was observed as the number of processors increased. Our interest in these questions arises from their general nature and the likelihood that analogous effects will afflict THOR's communication procedures in future extensions.

In this chapter we will present four primary findings. The first is that the irregularities first noticed by Zerr in [22], namely the varying refinement-dependent trends in weak scaling and the unexpected loss of efficiency at high p, can be recreated on the Falcon HPC. The second is that some of these effects follow logically from the structure of the communication scheme. Third, that the communication behavior of a code running on the Falcon HPC changes dramatically depending on the machine's utilization at run time. And, fourth, that these changes in loading can have a significant impact on the execution of codes which rely on large numbers of point-to-point communications.

5.2 Literature Review

5.2.1 The PIDOTS Code

PIDOTS, the Parallel Integral Discrete Ordinates Transport Solver, is a Fortran 90 based code that implements the Integral Transport Matrix Method (ITMM) for solving the steady state, one-speed neutron transport equation with isotropic scattering on a spatially decomposed Cartesian meshes. This combination of solver and spatial domain parallelism was intended to provide a novel approach to massively scalable, structured mesh transport problems.

As this section focuses on communication behavior, the full details of the ITMM are beyond the scope of this review. A more detailed derivation is given in [22]. At a high level, this method replaces the inner iteration scheme with a new operator called the integral discrete ordinates transport matrix that allows for explicit, i.e. non-iterative, evaluation of the solution. Conceptually this is achieved by constructing the dense, discrete-variable transport matrix acting on the vector of cell-averaged scalar fluxes with the right-hand side comprised of the vector of cell-averaged, isotropic fixed source. To begin, one can define a single scalar flux iterate update as:

$$\phi^{\nu} = J_{\phi}(\phi^{p} + \Sigma_s^{-1} q) \tag{29}$$

Here, ϕ^{ν} is the updated scalar flux iterate, ϕ^{p} is the previous iterate, and $\Sigma_s^{-1} q$ is the material scattering source. The Jacobian of the iteration, J_{ϕ}, is defined as:

$$J_{\phi} = \frac{\partial \phi^{\nu}}{\partial \phi^{p}} \tag{30}$$

However, as the number of iterates trends to infinity, the solution converges to a solution of the form shown in equation (31), where I is the identity matrix.

$$\phi^{\infty} = \left(I - J_{\phi}\right)^{-1} J_{\phi}(\Sigma_s^{-1} q) \tag{31}$$

This implies that, if J_{ϕ} can be constructed, the converged solution can be directly computed. The construction is accomplished using a single mesh sweep in all discrete directions in which a given cell's cell-averaged scalar flux is coupled to the cell-averaged angular fluxes of its upstream neighbors. When this is done for all angles, the net effect is to couple the scalar fluxes of every mesh cell to the scalar flux in all mesh cells yielding a full matrix operator.

In addition to implementing the ITMM method, PIDOTS also implements a spatial domain decomposition using a red-black Gauss-Seidel iteration scheme (a Parallel-Block Jacobi variation also exists, but it will not be analyzed here). This method was briefly described in section 3.2.2. As a brief recap, this method subdivides the problem domain amongst participating processors. On each processor, the domain is further divided into a set of sub-domains, colored red and black in a checkerboard pattern, each of which is composed of some number of mesh cells. Since this is a structured mesh, each division in the exercised tests is designed to be of the same size so the computational load is balanced across subdomains and processes.

The selection of the number of processor domains is a function of available resources, while the selection of sub-domain size stems from the desire to optimize the solving time. Larger subdomains will require fewer iterations to converge, while finer subdomains will have smaller (and faster to solve) ITMM operators. Note, since the ITMM solver removes the need to perform source iterations, these iterations are instead related to converging the sub-domain boundary angular fluxes.

While executing, PIDOTS simultaneously solves the ITMM matrix equation for a single color within each subdomain using incoming angular fluxes from neighbors with the other color. Then, it packages up the resulting, i.e., updated, angular flux information on the subdomain boundaries and propagates them to neighboring subdomains of the opposite color. For subdomains on different processors, this requires the use of MPI send and receive functions. This solve/communicate process repeats until the problem is converged and a full solution, namely cell-averaged scalar flux distribution is achieved.

5.2.2 HPC Topology & Communication

Due to the sheer number of different HPC hardware configurations, network topologies, and intended use cases, it is difficult to produce a singular analysis of HPC communication behavior. Instead, this section will focus on the class of systems which includes Falcon, our primary development HPC. To this end, discussion will be presented regarding distributed-memory systems with hypercube topologies.

As discussed in [37], distributed-memory systems are ones in which each node is a complete computer with its own processor, memory, and input/output devices. When multiple nodes of this type are joined via a network interconnect, a distributed-memory system, or cluster, is the result. Since the individual nodes are discrete computers, they must communicate via messages. This contrasts with shared-memory systems in which all the processors share hardware-level access to one another's memory. While shared-memory systems have inherently lower latency as a result of having this hardware access, distributed systems are far more scalable. This scalability results from the ability to simply add another node to the cluster with only changes to the network hardware. As a result, modern HPC clusters can have processor counts in excess of 10K-100K cores.

Given a distributed memory system of arbitrary size, there a variety of ways in which to interconnect the nodes. This is referred to as the network topology. Ref. [37] explores a variety of these topologies, while [38] examines the hypercube topology in more detail. A hypercube topology is represented as an n-dimensional cube where each vertex of the cube is a node in the compute system. Note here that, as discussed earlier, Falcon is not a true hypercube. First, each vertex of the hypercube represents a sub-cluster of compute nodes and, second, Falcon also has missing vertices in its 7th dimension. These differences cause Falcon to behave sub-optimally compared to a true hypercube and, as will be shown in the following sections, exacerbate issues stemming from communication.

As discussed in [38], the hypercube topology is intended to provide a scalable system with a good balance between maximum communication distance and interconnect number. The first of these metrics, maximum communication distance, is defined as the diameter of the topology. For a hypercube, it is $\log_2 n$, where n is the number of nodes. The number of connections is referred to as the degree. Ideally, a fully interconnect set of nodes would have a minimum diameter and a maximum degree. However, it is important to realize that the degree also represents the number of physical interconnects between each node. As such, topologies with too high of a degree are unsuitable due to physical constraints. For a hypercube, the degree is the same as the diameter, $\log_2 n$.

With the hardware and topology defined, the next key component of HPC operations is the communication method. As mentioned earlier, for distributed systems, this communication takes the form of discrete messages over a network interconnect. The most common system for these messages in production code is the MPI standard and its resulting libraries, such as Open MPI [28]. These libraries provide black-box solutions to the process of establishing a communication network and exchanging messages. Instead of dealing with the network communication systems directly, developers use the libraries to perform high-level operations such as point-point send and receive operations or collective communications such as *all_gather* or *all_reduce*. A send operation can be used to communicate a single piece of data from one node to another, while the more powerful operations allow for data collection and manipulation across entire subsets of the cluster.

5.3 Results

5.3.1 Recreating the PIDOTS Scaling Problem

When the parallel Gauss-Seidel (PGS) implementation of PIDOTS was evaluated, an unusual trend was noticed in the weak scaling behavior. As the spatial decomposition is asynchronous, an increase in iterations with both processor count and subdomain fineness is expected. This behavior was indeed observed. An example of these results, from a $h = 1.0\,cm, c = 0.99$ case is presented for four levels of subdomain refinement and processor counts ranging from 1 to $\sim 10^5$ adapted from [22] is presented below.

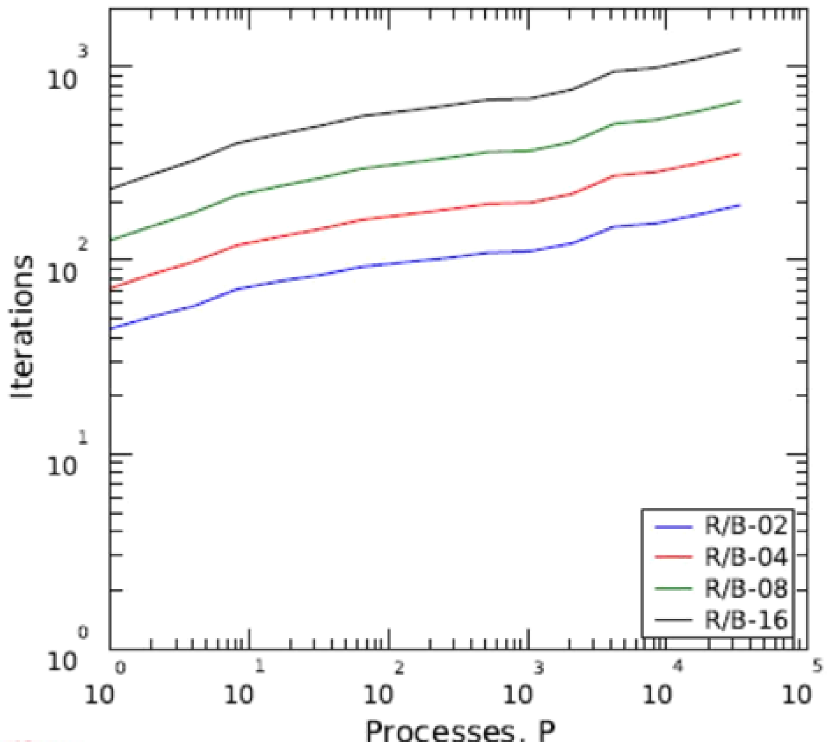

Figure 23: Original PIDOTS Iteration Count as a Function of Processor Count; adapted from [22].

In Figure 23, The notation R/B-# refers to the number of red/black subdomains per dimension of the processor domain. As such, R/B-02 has the coarsest subdomain sub-division and RB-16 has the finest. While this figure's behavior aligned well with the expected behavior, the resulting weak scaling behavior differed quite significantly. As can be seen in Figure 24, adapted from [22], the curves are no longer clearly separated. Instead, there are several crossover points and differing trends present.

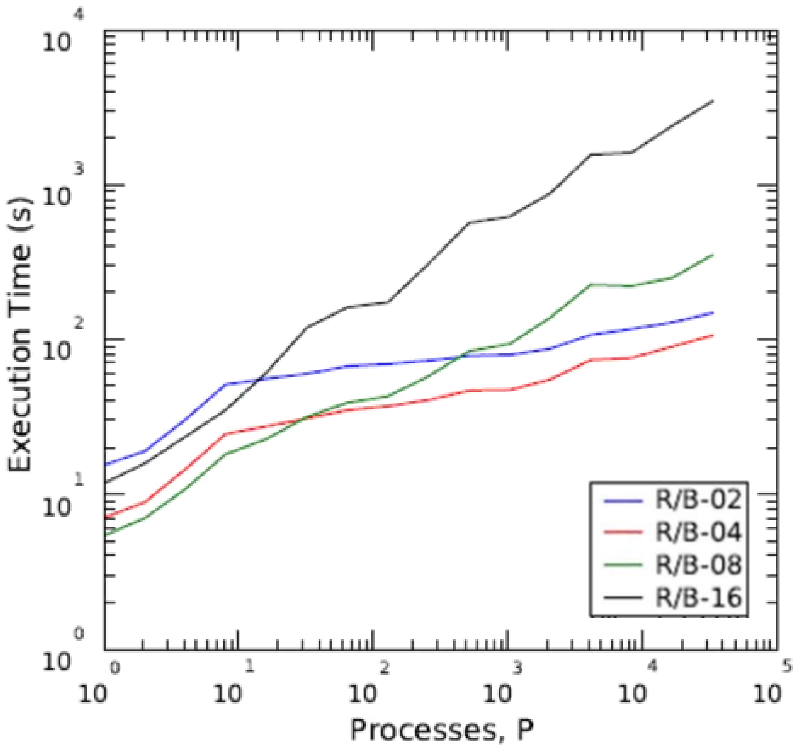

Figure 24: Original PIDOTS Weak Scaling Results adapted from [22].

To confirm that this behavior was not an artifact of the specific HPC that PIDOTS was run on, we recreated the performance tests on the Falcon system, albeit on fewer processors. These results, shown in Figure 25 and Figure 26, present the same information. Due to the availability of resources on Falcon, we only ran cases up to 10^3 processors. As can be seen, the iteration counts line up perfectly, indicating that the same problem was solved. While not identical, the weak scaling execution times show the same trends. The ordering of lines at $p = 1$ is identical and the same lines as in Zerr's results cross over. The major difference appears in the number of processors where these crossovers occur. For the Falcon results, the crossover points occur at higher values of p than they did in the original tests [22]. This can likely be attributed to different system parameters, such as processor speed and bandwidth. Regardless, from these plots, it is clear that this observed, and unexpected, behavior of the weak scaling is a persistent feature of the algorithm as implemented and not restricted to a specific platform.

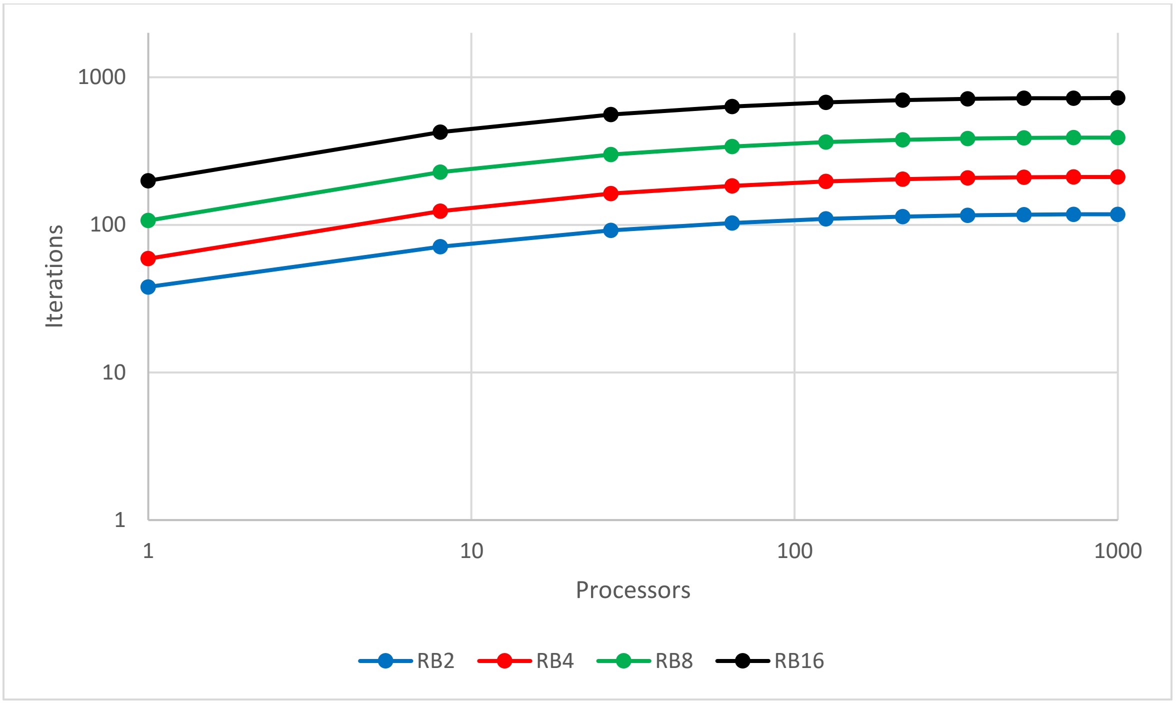

Figure 25: PIDOTS Iteration Trends from FALCON.

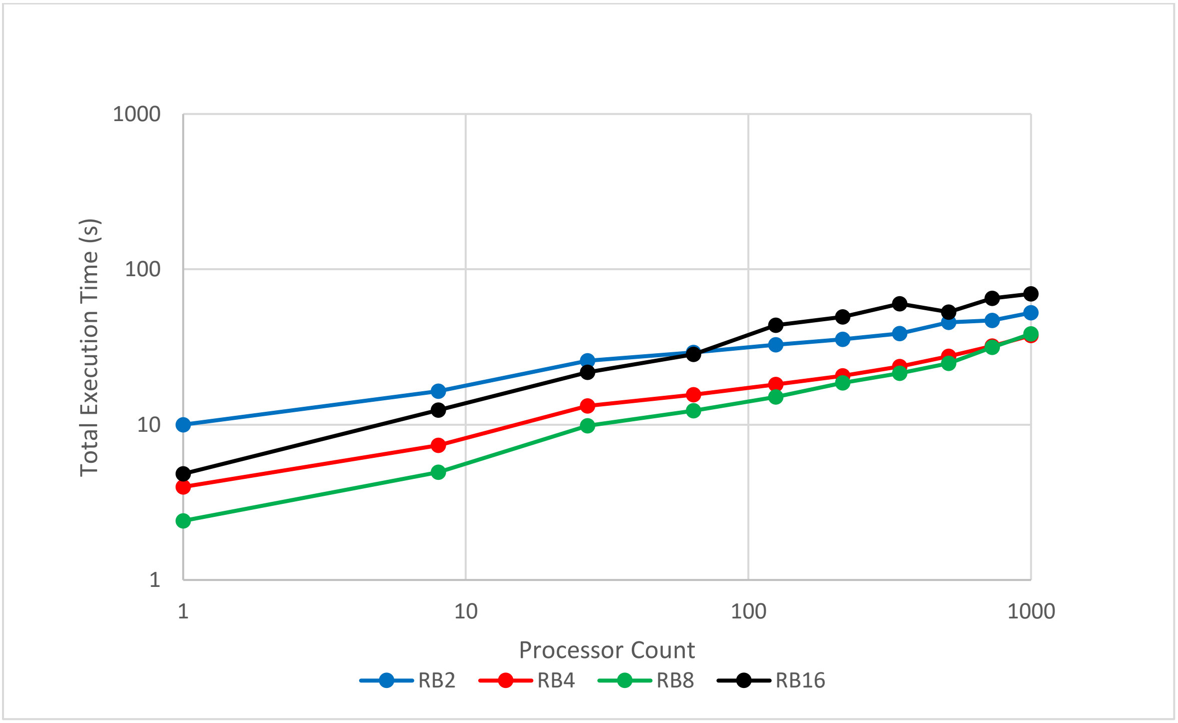

Figure 26: PIDOTS / Falcon Weak Scaling.

To supplement this analysis and provide a starting point for further investigation, the Falcon results were also used to create the curves shown in Figure 27, which depicts the execution time per iteration as a function of the number of processors. As the computational work per processor remains constant in a weak scaling test, the increase in the per-iteration cost should stem from communication. However, these costs grow far faster than expected, especially as the cases begin to approach $p = 10^3$. However, since the cost per iteration curves do not show the same crossovers, it is likely that the total effect is the result of two factors. In the following sections, each of these factors will be addressed.

This analysis was also performed on Fission, another INL HPC machine. As the results are essentially the same, these plots have been placed in Appendix B.

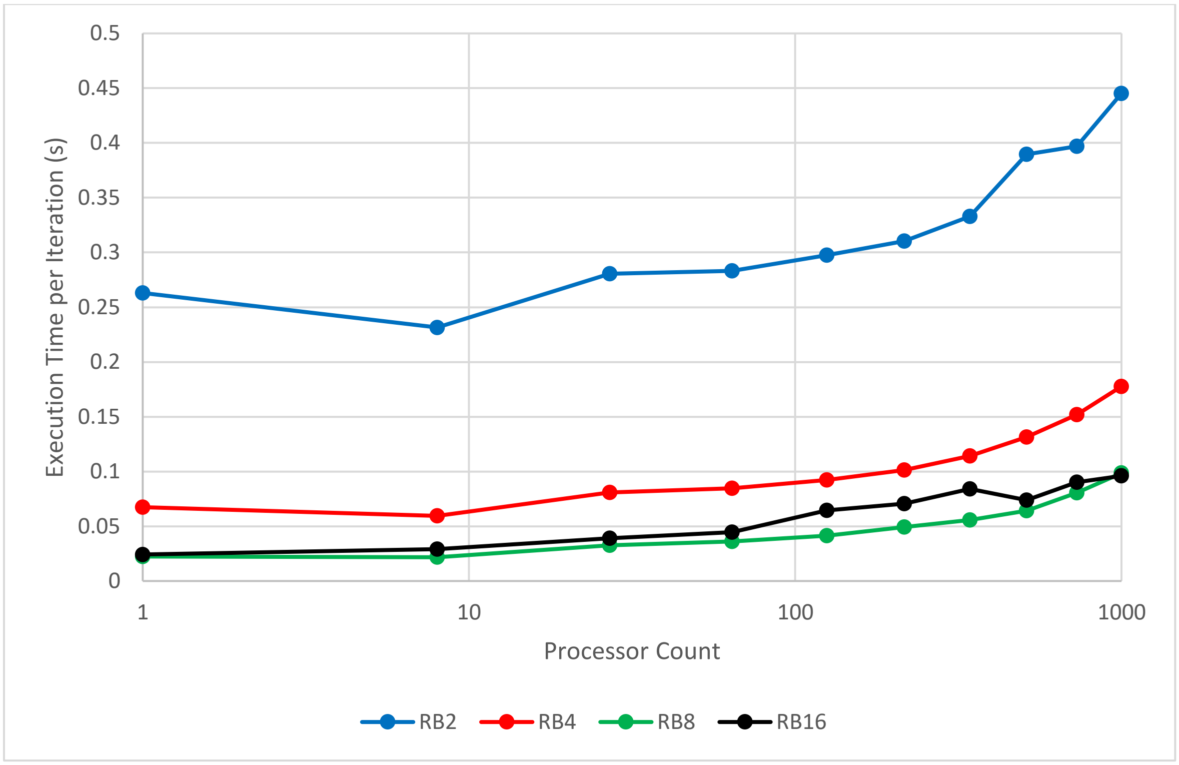

Figure 27: Falcon PIDOTS Time Per Iteration.

5.3.2 Justification of Crossover in the Weak Scaling Trends

As the weak scaling trends can be recreated on dissimilar computers, there must exist some feature of the PIDOTS algorithm which produces the crossovers. Additionally, as the processor count at which the crossovers occur varies from system to system, it is also necessary that this behavior be a function of some set of system parameters. The following model provides a high-level parameterization of the execution time of a single PIDOTS iteration as a function of the number of processors and the number of per-processor subdomains. In doing so, it should become clear, based on the tradeoff between computation and communication costs, how crossovers can occur.

A PIDOTs iteration can be subdivided into two distinct phases, solve and communication.

$$T_{iteration} = T_{solve} + T_{communication} \tag{32}$$

By defining w_L, the number of mesh cells per dimension of a processor domain, and s_L, the number of sub-domains per dimension of the processor domain, Eq. (32) can be expressed as a function of system and program parameters. Below, $\left(\frac{w_L^3}{s_L^3}\right)^3$ represents the amount of work required to invert the ITMM operator in each processor sub-domain. This is simply an order *n*^{3} matrix inversion operation applied to a matrix of dimension $\frac{w_L^3}{s_L^3}$. Then, for communication, each processor sub-domain must engage in three sends – one to each downstream neighbor – where each send has a data size corresponding to the number of cells on the sub-domain face, $\left(\frac{w_L}{s_L}\right)^2$.

$$T_{iteration} = c_1\left(\frac{w_L^3}{s_L^3}\right)^3 * s_L^3 + 3s_L^2 * \left(Latency + \frac{\left(\frac{w_L}{s_L}\right)^2}{Bandwidth}\right) \tag{33}$$

This evaluation can be further rearranged to produce a two-term expression in which, for a given computer system, the first term is purely a function of w_L and the second is a function of both w_L and s_L.

$$T_{iteration}(s_L) = \left(3 * \frac{w_L^2}{Bandwidth}\right) + \left(\frac{3 * Latency * s_L^8 + c_1 w_L^9}{s_L^6}\right) \tag{34}$$

When the data size of each send is large, this model will be dominated by the bandwidth term. However, as s_L increases, the number of messages increases and their size decreases. Consequently, the latency term will become increasingly significant. This behavior is seen in the second derivative of (34), where a minimum exists in the cost curve for a given mesh size. As such, the costs do not increase monotonically, and it is possible to experience crossover. This indicates that, for any given number of mesh elements, there exists a different processor sub-domain partition which optimizes computation time.

5.3.3 Exploring Communication Cost Growth on Falcon

To begin, it is important to understand some of the assumptions implicit in how most practitioners view communication in parallel codes. In general, communication is perceived to be purely a function of the machine hardware and of the user's code. No other external factors are considered. This essentially treats every piece of code as if it is running on a dedicated machine. For the most part, this is a completely sufficient description of communication. Delays that occur on this scale, like system interrupts or packet loss, are either rare or incur a small enough penalty relative to the execution time that they can be ignored. However, for this description to hold, three conditions must remain true. First, in systems where processors are not guaranteed maximum locality (i.e., they are scattered around the network depending on availability at the time of assignment), there must not be a significant amount of communication occurring as a result of other programs. If there is, the time to deliver a message becomes a function of the amount of traffic at the switch, including other executing programs.

Second, the contention for network resources from within the user’s program must be low. In Falcon, since the processors are two layers below the network nodes, this can occur if multiple servers on the same node all make network requests near-simultaneously. Finally, the assumption that typically incurred network slowdowns are negligible can be broken if the message size becomes sufficiently small. In this regime, the routing time, which is susceptible to influence form factors external to the code, dominates the communication cost.

In running large cases on PIDOTS on Falcon, all three of these assumptions are broken. Since Falcon runs near 100% utilization at all times, there is no guarantee of locality of the requested processors, especially for larger allocations, such as $p = 1000$. Since the majority of engineering codes executed on Falcon are, to some degree, parallel it can be expected that, without guaranteed locality, there is a high degree of inter-program contention for network resources at all times. Additionally, since a single switch services multiple servers, each of which contains 24 processors, it is reasonable to assume that self-contention for network resources will also be high. This can be confirmed by counting the number of sends in a single problem execution. For the $p = 64$, with 4,096 subdomains per processor case, the number of sends is ~93 million messages. Finally, the fine spatial decomposition achieved in the PIDOTS system, coupled with the fact that each sub-domain sends its own message, nearly guarantees that message sizes will be extremely small.

With these concerns in mind, the author contacted the HPC administrative staff at INL to gather information about Falcon’s communication cost curve [31]. No such curve was available. In fact, the only data that had been collected regarding communication costs was measured during an outage on a completely unloaded system. The measurements were used to test the point-to-point latency of pairs of servers or nodes at a time. At no point would more than 18 processors be communicating at once. From this aggregate data, the Falcon system was characterized to have a latency of $2.83 \pm 1.5\,\mu s$. This metric was collected using an InfiniBand diagnostic tool, so it represents a communication layer below that of the MPI library. As such, it is expected that measured costs will only represent a fraction of the final cost. Determining the magnitude of the cost increase from the InfiniBand layer to the MPI layer is beyond the scope of this work.

To capture the latency behavior of Falcon during periods of high utilization, we wrote a script to execute various calls to InfiniBand diagnostic tools and collect latency time distributions. For each execution of the script, a user defined number of processor pairs would execute the script and time the latency portion of 5 million repetitions of the send/receive operation.

To further augment these results, a network interconnect map, provided through [31], was used as a lookup table to determine the distance, in number of network hops, between the communicating processors.

As can be seen in the figures below, on the loaded Falcon system, there are multiple outliers whose final time cost is many orders of magnitude higher than those of the typical cost. This effect is evident for both the two and three hop cases presented. In the three-hop case, the density of these outliers appears to increase rather significantly. Two and three hop cases are presented as these are, as expected from the system diameter, the most common network distances when selecting two nodes at random from Falcon.

In both cases, the measured data is divided by 1.5 μs as this is the apparent floor for communication. As such, all results are given as multiples of this floor.

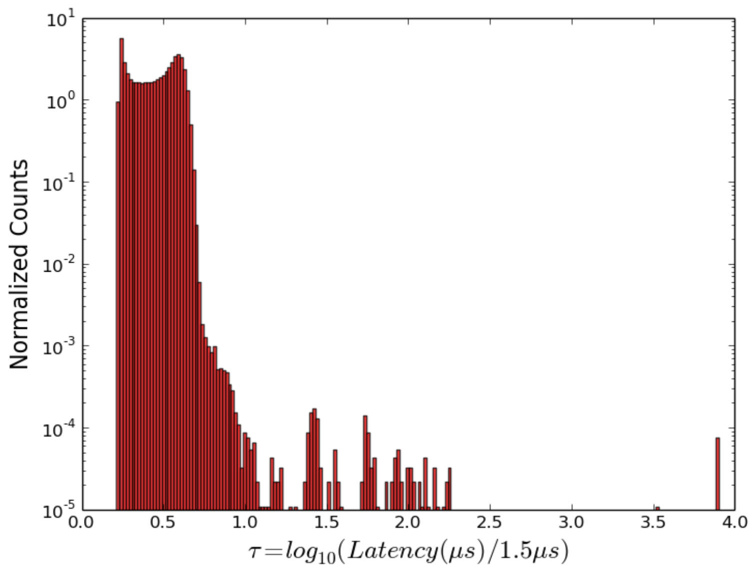

Figure 28: Histogram of 2-Hop Latency.

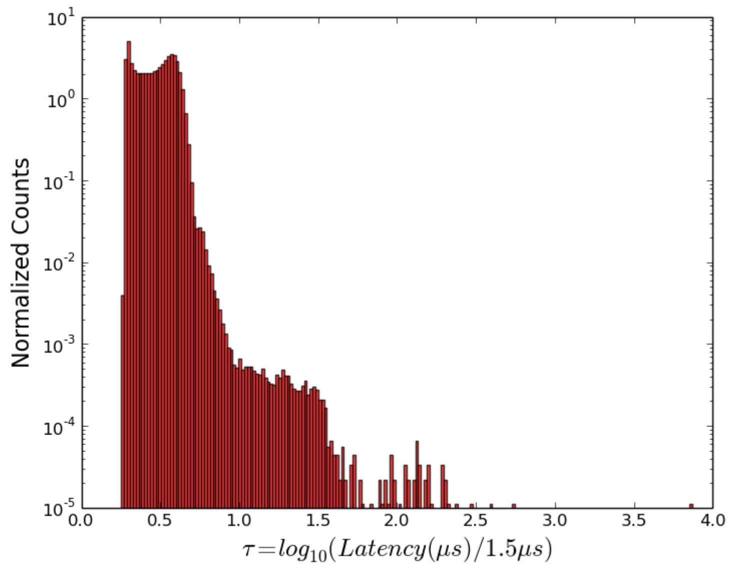

Figure 29: Histogram of 3-Hop Latency.

In Figure 28 and Figure 29, there are several common features. The first peak, occurring near 0.5, represents the standard latency time, i.e. the latency time resulting from a message neither incurring abnormal nor avoiding normal routing delays. Moving rightward, the second peak is associated with a frequent delay. This is likely self-contention as it is the only effect that should occur with a frequency approaching that of non-delayed communications. The region from this peak onward represents the slowdowns due to interactions among simultaneously executing programs at the node level. These effects stem from saturation of the switch links because of either high traffic or low traffic with much larger message sizes. As expected, when moving from the 2-hop case to 3-hop case, these network effects will grow more pronounced since there is an increased chance to encounter a node-level slowdown.

5.3.4 Modeling the Impact of the Latency Distribution

Having established that the network latency behavior of Falcon during periods of high utilization exhibits large outliers, it is now necessary to demonstrate that these outliers can have a significant impact on the execution time of a code. Doing this using PIDOTS would be difficult due to inability to control the conditions under which the code executes on Falcon. Instead, SimPy [39] was used to develop a discrete simulation model of PIDOTS communication behavior. This model is not intended to be a PPM for PIDOTS, but rather to serve as a case study for the accumulation of message latency delays. The PIDOTS communication style was used simply because an implementation of it was readily available for reference.

To create the simulation, a probability distribution function (PDF) of the Latency distribution is required. An average latency distribution was generated from the 2-hop and 3-hop measured data. To simplify the numerical representation of this PDF, the distribution was divided into 3 parts. The first section, designed to capture the first two peaks is a simple uniform distribution. The second region is defined as an exponential decay until a latency value of $20{,}000\ \mu s$ ($\sim\tau = 4.1$). From there on, the PDF is zero.

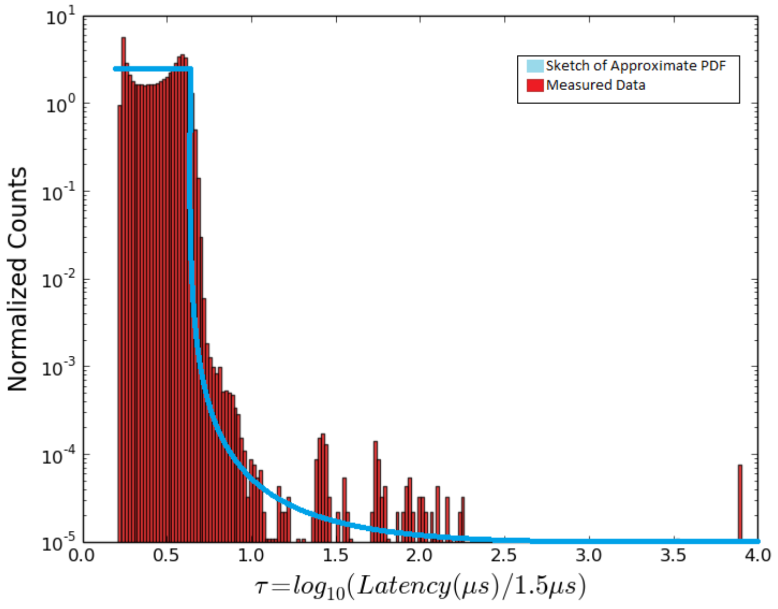

Figure 30: Sketch of Approximate PDF overlaid on Measured 2-Hop Data.

When the event simulation is run, a rejection scheme is used to sample an appropriate latency time for each communication that occurs in a single iteration of a problem of the specified size. This simulation was run for a variety of cases and a subset of the results are tabulated below. For all cases each test was repeated 100 times, except for $p = 32{,}768$, the first two refinements were run 20 times and latter two 5 times. The subdomains per processor linear column specifies the cubic root of the total number of subdomains per processor. The deterministic column represents the results if latency were fixed at the stated average value of $2.83\ \mu s$. The uniform column restricts the sampled latency values to the uniform region, roughly 1.5 μs to 5 μs. The full model column allowed for sampling of the entire distribution.

Table 8: Discrete Event Simulation Results.

Processors	Subdomains per Processor per Dimension	Deterministic Latency (μs)	Uniform Latency (μs)	Full Model (μs)
64	2	936	891	941
	4	1,147	1,136	1,194
	8	2,239	2,147	2,408
	16	5,629	6,649	7,851
512	2	907	929	1001
	4	1,108	1,132	2,189
	8	2,222	2,149	4,631
	16	5,636	6,686	11,503
4096	2	924	957	2,404
	4	1,102	1,142	4,839
	8	2,214	2,156	10,785
	16	5,650	6,712	19,909
32768	2	924	988	7,209
	4	1,103	1,156	13,896
	8	2,153	2,164	20,002
	16	5,712	6,727	28,146

As can be seen by looking across any row in the table, there is only a minor impact on the final iteration time when no outliers are allowed. However, when outliers are permitted, there is a sharp jump in the average predicted time. Since the PIDOTS communication scheme is, to an extent, ordered, a single slow process will introduce a slowdown in the measured time of all its dependent processes. This effect is cumulative over messages and across iterations. So, with the introduction of even a small number of large delays, significant deviation from the expected time profile can be observed. As this effect is observable even for low processor counts and will only be further magnified by the MPI overhead, it is reasonable to state that the effect is present, in a non-negligible fashion, in the operation of PIDOTS. The behavior of the $p = 64$ and $p = 32{,}768$ cases are plotted below as a visualization of the data in Table 8.

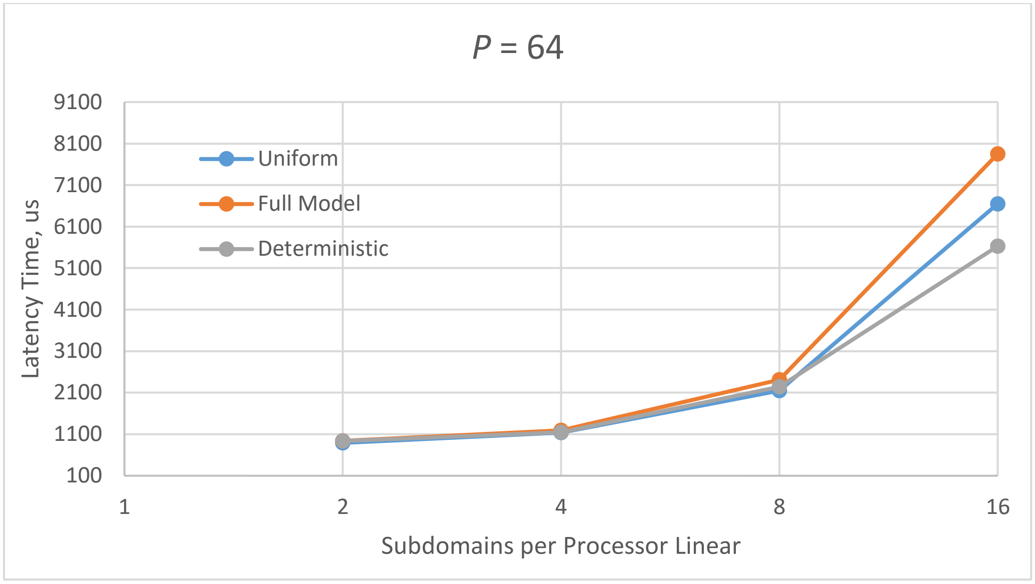

Figure 31: Comparing Latency Models for p=64.

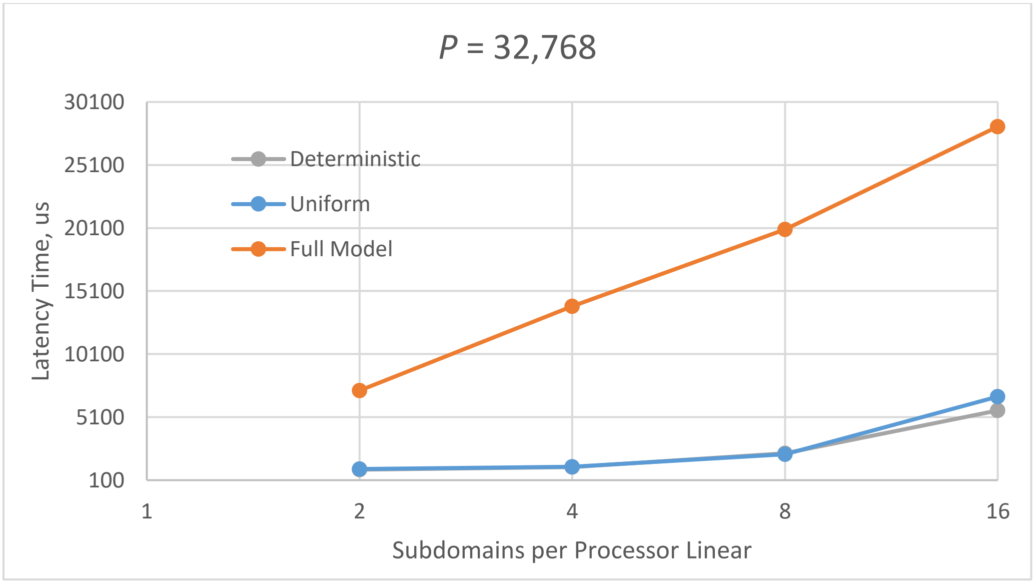

Figure 32: Comparing Latency Models for p=32,768.

5.4 Future Work

In this chapter, three distinct claims are made about the behavior of PIDOTS on general-use HPC systems. First, as demonstrated by Eq. (34), there exists a problem-dependent optimal value for processor sub-domain refinement, s_L. This allows for the weak scaling crossovers observed in both Zerr's work [22] and the results presented in this chapter. Second, under typical user conditions on the Falcon HPC, the distribution of communication latency times contains large, infrequent outliers. Since, as PIDOTS cases are increasingly refined, the data volume per message decreases, these outliers can lead to individual communication times several orders of magnitude longer than expected. Third, using the SimPy discrete event simulation and a conservative latency distribution, it can be shown that PIDOTS-like communication schemes send enough messages that encountering these high-latency delays is almost guaranteed. Furthermore, the degree of serialism inherent to a PIDOTS-like communication phase, coupled with the repeated occurrence of these latency spikes, can lead to large increases in the run time of the overall code.

Unfortunately, this discrete event simulation is not capable of simulating all the components which effect the true latency time. Nor is it capable of isolating the contributions of self-contention and external contention from the general model. It would be an interesting endeavor to continue the analysis of this model and to parameterize it in terms of system utilization, processor locality, and local network link structure. However, since these items could only further degrade the performance of the communication latency, this would likely only serve to further support the same conclusions reached so far.

Having established that the high-utilization latency behavior of Falcon can have a detrimental effect on the run time of massively parallel codes, it is now also necessary to modify the PIDOTS code to reflect a new communication style that helps to mitigate the latency effects. Based on the latency models presented in this chapter, that modification would likely involve sending fewer, larger messages. This would help to decrease the relative latency cost as well as lower the number of opportunities during which to encounter delays, somewhat analogously to the R/B08 and R/B16 cases. The implementation of such a system will be pursued in the near future.

Finally, it would be of great benefit to repeat this work on yet another HPC system, potentially one with a drastically different architecture, to see if the same trends can be observed. This would increase the generality and impact of the conclusions.

6 Software Management Improvements

Whereas previous chapters have presented research conducted in pursuit of improving and analyzing the capabilities of the THOR code and the Falcon HPC, this chapter will instead discuss the various improvements that have been made to the THOR project in terms of software management. These improvements are designed to help enforce code standards, maintain developer accountability, and rigorously test modifications to the code as required by modern standards of code development and maintenance.

6.1 Version Control

Version control is typically considered to be one of the primary requirements of a professionally maintained software project. Prior to the author's work with THOR, the code had been maintained via backups and archives kept by the various developers of the code. This did not stem from negligence on the part of previous developers, but rather from a variety of concerns. Since THOR has typically had a single primary developer at any given time, these concerns, coupled with the lack of need for a multi-author platform, effectively squelched any desire to implement version control.

The two primary concerns related to using a public-facing version control software both stemmed from issues related to code ownership and accountability. Since the source code is based at NC State University (NCSU), it was thought best to keep the code only available to NCSU affiliates and those specifically authorized by them. Compounding these concerns were long-term goals related to registering the finished code with the Radiation Safety Information Computational Center (RSICC) based at Oak Ridge National Laboratory. The desire to maintain a limited list of authorized users again forestalled thoughts of public version control.

However, NCSU maintains a Github Enterprise instance which resides behind the university's access control system and allows for projects to be made completely private save for whitelisted users. This functionality resolves concerns related to both the NCSU ownership interest and access accountability. As such, all code development is now directed through the

NCSU Github system. This provides version control as well as centralized issue management to all the current active THOR developers.

6.2 Modular Code Restructuring

In previous years, THOR has been a monolithic code. There was a single executable that controlled all the functions the code was capable of performing. As such, emphasis was placed on maintaining capabilities and avoiding regressions over implementing edge case features. This led to a variety of projects that needed to be implemented but could not be due to their impact on the existing code. Adding to this were the myriad unofficial scripts which were written to supplement the features or tool-chain of the core code.

While only nascent, the THOR project has undergone a paradigm shift recently. Now, rather than a single centralized program, developers are encouraged to contribute well-defined pre- and post-processors which can augment the abilities of the central THOR transport solver. This shift will hopefully drive a rapid increase in the number of supported data handling and solution post-processing capabilities. Furthermore, this should actively reduce the need for unofficial scripts and help to consolidate the capabilities maintained by each developer.

So far, implementation of these pre- and post-processors has been limited. Most of this work has gone into developing a mesh handling pre-processor to expand the types of meshes that can be converted into the THOR mesh format. Formalizing this process has both reduced the frequency of direct user input in the mesh generation process and has made THOR compatible with a variety of widely supported mesh formats, including Exodus II.

Near term plans involve further expanding the fixed source definition capabilities via a first-collision source interpreter as well as the abstraction of all solution analysis to a post-processor to allow for greatly increased flexibility in output generation.

6.3 Formal Testing

THOR has always maintained a small set of tests designed to ensure that the core functionality of the code does not regress due to new features. However, these tests were not explicitly designed to exercise the code, but rather were a simple assortment of conveniently small and varied benchmark and debugging scenarios. The goal of this initiative was to formalize the testing by establishing not only a set of test cases, but also a clear understanding of what each case was designed to test for.

Unfortunately, Fortran does not easily provide the capability to perform unit tests (i.e. tests which exercise small portions of the code exclusive of the rest of the project). Even where this is possible, the complexity of some of the required inputs would make unit tests infeasible. Instead, it was optimal to establish a small set of tests, each with a specific execution case in mind to exercise as broad a swathe of the code as possible.

These standardized tests were implemented, as well as a python testing harness. Now, performing the full test suite is a simple matter of executing a single command rather than manually checking over a series of benchmarks. Better still, since the cases are parameterized, it is possible to use a test suite to narrow down which aspect of the code is failing rather than simply reporting a test failure.

In terms of implementation, this standard has been strictly enforced for the new pre- and post-processor systems. The mesh pre-processor is chief among these, with a robust test suite which tests essentially all captured error states as well as the entirety of the execution logic.

Admittedly, for a more complex system, like the core transport solver, obtaining complete coverage is unlikely. However, effort is being invested to ensure all new functionality includes a sufficient degree of testing and that refactored functionality is amended with new tests.

6.4 Documentation and Testing Coverage Metrics

Promises of improved testing and code documentation are always appreciated. But, it is also easy to become distracted by an interesting problem and neglect proper code maintenance work. To assist with maintaining code standards across the entirety of the project, THOR also implements project-level automation for generating test coverage metrics and code documentation. These tools, coupled with the test harness can be used to generate project level reports with specific sections for each pre/post processor and the core solver. The code coverage reports are using the Unix Lcov routine [40] and the automated documentation is built using doxygen [41]. Together, these scripts help developers to identify portions of the project which do not conform to standards regarding test coverage or documentation.

Like with the testing program, it will be a major undertaking to bring all of THOR into compliance with these new standards. However, significant effort is being made to ensure all new code conforms to these standards and that existing code is brought into compliance as it is reviewed and modified. Hopefully, this will cement a new developer standard for the package and encourage future developers to maintain a similar, if not better, level of code maintenance.

6.5 Automated Build System

To cap off the process of automating the project testing, documentation, and coverage components, a Jenkins [42] based continuous integration server has been established with the ability to execute these scripts and interpret the resulting reports. While likely an overkill for a project of this size, it is the author’s hope that, by automating tracking of code compliance, it will become standard for future developers to strive to achieve the required metrics instead of just ignoring the availability of these tools to the ultimate detriment of the fruits of their labor.

6.6 Summary

In brief, recent modifications to the THOR project at a software management and maintainability level have been designed to enable a variety of powerful features that are critically important to a professional software project. These include formal code revision and versioning, along with issue tracking, using the NCSU Enterprise Github instance as well as the addition of a Jenkins based build, test, and document integration server. This server maintains the capability to exercise all tests maintained under the new THOR test harness format as well as to perform code coverage and report on the status of each test. Additionally, the server is capable of building doxygen based documentation to support code coverage in terms of both testing and documenting. Augmenting these new code maintainability requirements is a change in the structure of the THOR project. Now, with official support for pre/post-processors, the capabilities of the primary project should be able to more rapidly grow to meet the needs of various research projects within the group.

From a user perspective, these automated tools will help to streamline the installation process by establishing a single point installation script which can compile, test, and report to the new user. This should lower the entry barrier for new users and developers.

These new code maintenance standards do introduce greater upfront costs to the developers by requiring time investment in non-production aspects of the code. However, this upfront cost should be massively less than the accumulated lost time resulting from insufficient documentation or uncaught regressions as the THOR project continues to grow and evolve.

7 Conclusions

In this section a brief overview of each of the major works presented in this thesis will be provided and, as each chapter has already presented chapter-specific conclusions, brief final comments regarding the totality of the project will be made.

7.1 Summary

This work presented four primary chapters all centered about the concept of implementing, evaluating, and improving routines designed to increase the run time efficiency of the THOR neutron transport code.

In Chapter 2, discussion centered on the implementation of the Chebychev outer acceleration scheme into the THOR code. This outer acceleration provided a significant speedup compared to the unaccelerated solver, ~4x, while neither significantly modifying the existing code structure nor introducing significant resource utilization overhead. This method surpassed the existing fission source extrapolation scheme and proved roughly equal to some version of the THOR JFNK solver.

Chapter 3 highlighted the implementation of angular domain decomposition into the THOR project. Again, this addition provides the opportunity to obtain significant speedups by utilizing 10s to 100s of processors. The ADD implementation of THOR is now the standard working version of the code and will be the centerpiece of future development. Hopefully, future work will provide a massively parallel SDD scheme to supplement this effective, but only moderately parallel, algorithm.

The next chapter dove deep into the analysis of the THOR angular domain decomposition algorithm by developing a parallel performance model for the parallelized, unaccelerated power iteration scheme. This PPM was able to effectively predict the run time of the THOR code across all tested combination of mesh, mesh refinement, and quadrature order. Due to its modular construction, the PPM is easily modifiable and can be used as a diagnostic tool to identify code modules with inefficient or unexpected behavior.

In Chapter 5, focus shifted away from the THOR code and instead focused on the behavior of the PIDOTS massively parallel code on high-utilization HPC systems. This analysis, using both measurements and models, demonstrated that unaccounted for, rare, large delays in communication could have a cumulative effect on a running code and cause significant slowdowns at the macro level. While this work is not directly applicable to THOR in its current state, it provides information that will be critical to future development, especially as focus shifts to massively parallel schemes and larger processor counts.

Finally, Chapter 6 detailed improvements made to THOR intended to make the project more robust and professional from a software engineering point of view. While these improvements do not directly improve the results of the code, they should greatly improve its maintainability and usability in the long term.

7.2 Final Comments

The research conducted in this thesis, in addition to the software management improvements described in Chapter 6, have made great strides in moving THOR towards its ultimate goal of being a production level code.

The speedups made available through acceleration and parallelization have greatly broadened the scope of viable problems and have made THOR a more effective tool for research investigations into not only reactor analysis problems, but soon, also for evaluation of multiplicity in sub-critical weapons grade material and other non-proliferation scenarios.

The analysis performed through the PPM and the latency model have helped to improve the efficiency of the code and the THOR developers' understanding of network communications on modern HPCs. These tools will continue to provide benefit to THOR as further modification and upgrades are implemented.

The author looks forward to his continued work with the THOR project and its future development goals.

REFERENCES

[1] J. J. Duderstadt and L. J. Hamilton, Nuclear Reactor Analysis, John Wiley & Sons, 1976.

[2] G. Bell and S. Glasstone, Nuclear Reactor Theory, La Grange Park, IL: American Nuclear Society, 1970.

[3] R. Ferrer, "An Arbitrarily High Order Transport Method of the Characteristic Type for Unstructured Tetrahedral Grids," The Pennsylvania State University, State College, PA, 2010.

[4] R. Yessayan, Y. Azmy and S. Schunert, "Verification and Validation of the Tetrahedral Grid Radiation Transport Code THOR Based on the Advance Test Reactor Benchmark," in *Physor 2016*, Sun Valley, ID, 2016.

[5] "Advanced Test Reactor: Serpentine Arrangement of Highly Enriched Water-Moderated Uranium-Aluminide Fuel Plates Reflected by Beryllium," Idaho National Laboratory, Idaho Falls, ID, 2005.

[6] S. Schunert, Y. Azmy and R. Ferrer, "Verification of the Three-Dimensional Tetrahedral Grid Sn Code THOR," in *Proceedings of International Conference on Mathematics and Computational Methods Applied to Nuclear Science & Engineering (M&C 2013)*, Sun Valley, Idaho, USA, 2013.

[7] R. E. Alcouffe, "Diffusion Synthetic Acceleration Mehtods for the Diamond-Differenced Discrete-Ordinates Equations," *Nuclear Science and Engineering,* vol. 64, pp. 344-355, 1977.

[8] Y. Wang, *Methods for Improving Chebychev Acceleration Convergence.* [Personal Communication]. June 2015.

[9] A. Hébert, "Variational Principles and Convergence Acceleration Strategies for the Neutron Diffusion Equation," *Nuclear Science and Engineering,* vol. 91, pp. 414-427, 1985.

[10] W. A. Rhoades and D. B. Simpson, "The TORT Three-Dimensional Discrete Ordinates Neutron Photon Transport Code (TORT Version 3)," Oak Ridge National Laboratory, Oak Ridge, TN, 1998.

[11] E. E. Lewis and J. F. Miller, Computational Methods of Neutron Transport, La Grange Park, IL: American Nuclear Society, 1993.

[12] N. Cho and C. Park, "A Comparison of Coarse Mesh Rebalance (CMR) and Coarse Mesh Finite Difference (CMFD) Acceleration Methods for the Neutron Transport Calculations," Korea Advanced Institute of Science and Technology: Nuclear Reactor Analysis and Particle Transport Laboratory, 2002.

[13] Y. Azmy and E. Sartori, Nuclear Computational Science: A Century in Review, Springer Science & Business Media, 2010.

[14] N. Odry, J. Lautard, J. Vidal and G. Rimpault, "Coarse Mesh Rebalance Acceleration Applied to an Iterative Domain Decomposition Method on Unstructured Mesh," *Nuclear Science and Engineering,* vol. 3, no. 187, pp. 240-253, 2017.

[15] W. Boyd et al., "The OpenMOC Method of Characteristics Neutral Particle Transport Code," *Annals of Nuclear Energy,* vol. 68, pp. 43-52, 2014.

[16] OpenMOC, "Coarse Mesh Finite Difference Acceleration," 2014. [Online]. Available: https://mit-crpg.github.io/OpenMOC/methods/cmfd.html. [Accessed October 2017].

[17] Y. Azmy, " Unconditionally stable and robust adjacent-cell diffusive preconditioning of weighted-difference particle transport methods is impossible," *Journal of Computational Physics,* vol. 182, pp. 213-233, 2002.

[18] S. Schunert et al., "A flexible nonlinear diffusion acceleration method for the SN transport equations discretized with discontinuous finite," *Journal of Computational Physics,* vol. 338, pp. 107-136, 2017.

[19] T. Sutton, P. Romano and B. Nease, "On-the-fly Monte Carlo Dominance Ratio Calculation using the Noise Propagation Matrix," *Progress in Nuclear Science and Technology,* vol. 2, pp. 749-756, 2011.

[20] Y. Azmy, "Multiprocessing for Neutron Diffusion and Deterministic Transport Methods," *Progress in Nuclear Energy,* vol. 31, no. 3, pp. 317-368, 1997.

[21] B. Wienke and R. Hiromoto, "Parallel Sn Iteration Schemes," *Nuclear Science and Engineering,* vol. 15, pp. 49-59, 1985.

[22] R. J. Zerr, "Solution of the Within-Group Multidimensional Discrete Ordinates Transport Equations on Massively Parallel Architectures," The Pennsylvania State University, State College, PA, 2011.

[23] K. Baker, "An Sn Algorithm for the Massively Parallel CM-200 Computer," *Nuclear Science and Engineering,* vol. 128, p. 312, 1998.

[24] M. Adams et al., "Provably Optimal Parallel Transport Sweeps On Regular Grids," in *M&C 2013,* Sun Valley, ID, 2013.

[25] T. Bailey et al., "Validation of Full-Domain Massively Parallel Transport Sweep Algorithms," in *2014 American Nuclear Society Winter Meeting and Nuclear Technology Expo*, Anaheim, CA, 2014.

[26] W. Hawkins et al., "Efficient Massively Parallel Transport Sweeps," in *Transactions of the American Nuclear Society*, San Diego California, 2012.

[27] Y. Azmy, "Communication Strategies for Angular Domain Decomposition of Transport Calculations on Message Passing Multiprocessors," in *Joint International Conference on Mathematical Methods and*, Saratoga Springs, NY, 1997.

[28] E. Gabriel et al., "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation," in *11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, 2004.

[29] Y. Azmy, "Performance and Performance Modeling of a Parallel Algorithm for Solving the Neutron Transport Equation," *The Journal of Supercomputing,* vol. 6, pp. 211-235, 1992.

[30] J. Fischer and Y. Azmy, "Comparison via parallel performance models of angular and spatial domain decompositions for solving neutral particle transport problems," *Progress in Nuclear Energy,* vol. 49, no. 1, pp. 37-60, 2007.

[31] C. Garvey, *Information about the Falcon and Fission HPCs.* [Personal Communication]. 2015-2018.

[32] R. Ferrer and Y. Azmy, "A Robust Arbitrarily High-Order Transport Method of the Characteristic Type for Unstructured Grids," *Nuclear Science and Engineering,* vol. 172, pp. 33-51, 2012.

[33] R. Yessayan, Y. Azmy and S. Schunert, "Development Of A Parallel Performance Model For The THOR Neutral Particle Transport Code," in *M&C 2017,* Jeju, South Korea, 2017.

[34] D. Mosteller and J. Goda, "Analysis of Godiva-IV Delayed-Critical and Static Super-Prompt-Critical Conditions," in *M&C 2009,* Saratoga Springs, NY, 2009.

[35] E. Lewis, "Benchmark specification for Deterministic 2-D/3-D MOX fuel assembly transport calculations without spatial homogenization (C5G7 MOX)," Nuclear Energy Agency, 2008.

[36] T. Takeda and H. Ikeda, "3-D Neutron Transport Benchmarks," *Nuclear Science and Technology,* 1991.

[37] Y. Solihin, Fundementals of Parallel Multicore Architecture, Boca Raton, FL: CRC Press, 2016.

[38] G. Ostrouchov, "Parallel Computing on a Hypercube: An Overview of the Architecture and Some Applications," Oak Ridge National Laboratory, Oak Ridge, TN, 1987.

[39] SimPy, "Simpy: Discrete event simulation for Python," 2017. [Online]. Available: https://simpy.readthedocs.io/en/latest/. [Accessed 2017].

[40] LCOV, "LCOV - the LTP GCOV extension," 2016. [Online]. Available: http://ltp.sourceforge.net/coverage/lcov.php. [Accessed 2017].

[41] Doxygen, "Doxygen: Generate documentation from source code," 2017. [Online]. Available: http://www.stack.nl/~dimitri/doxygen/. [Accessed 2017].

[42] Jenkins, "Jenkins User Documentation," 2017. [Online]. Available: https://jenkins.io/doc/. [Accessed 2017].

[43] OpenMPI, "Open MPI Documentation," 2017. [Online]. Available: https://www.open-mpi.org/doc/. [Accessed 2017].

APPENDICES

APPENDIX A – FALCON TOPOLOGY

The basic building block of Falcon’s network topology is the IRU, as discussed in Section 4.3. This appendix will provide a brief graphical description of the way in which these blocks are assembled into the full topology.

All figures adapted from [31] and modified with explanations where necessary.

A basic IRU is composed of 18 nodes and 2 switches as shown below, with nodes as rectangles and switches as polygons. This forms the 1D structure of the topology.

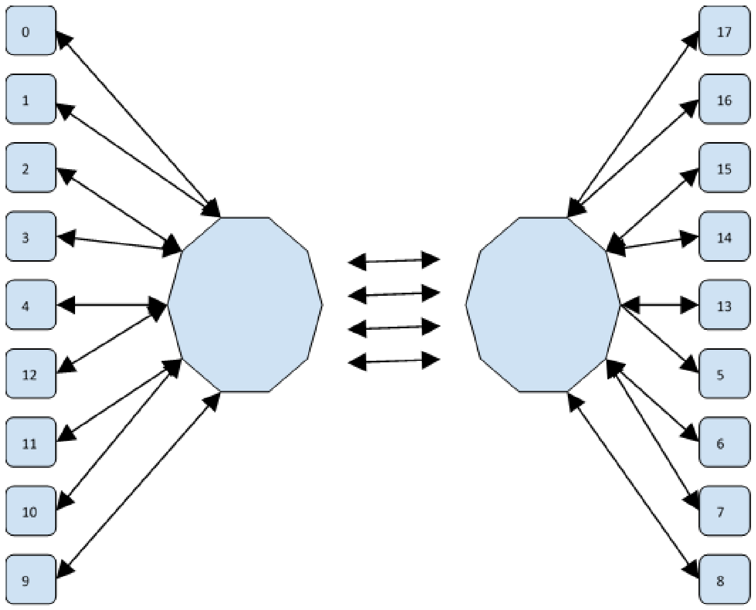

Figure 33: Falcon IRU Structure and 1D Topology.

From here, the 2D structure can be assembled by connecting multiple IRUs. Each switch is populated with four links to its neighboring switches. Stepping to 3D, multiple IRUs can be connected into grids, as shown below. Note here that each rectangle now represents an entire IRU and each line four links. This also corresponds to one physical rack of Falcon

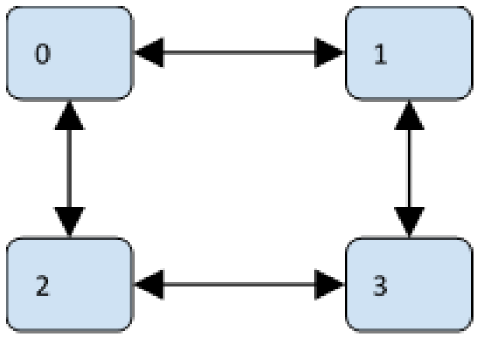

Figure 34: Falcon 3D Topology.

Joining two 3D assemblies results in a 4D assembly as shown below. Again, each arrow represents four links.

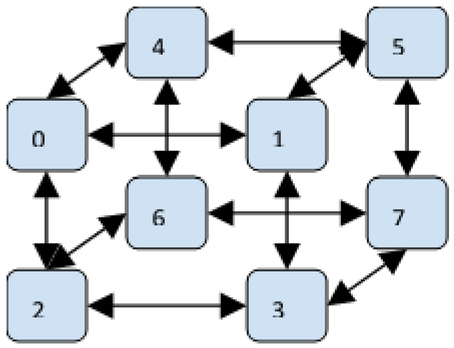

Figure 35: Falcon 4D Topology.

A 5D assembly is created by combining two 4D units and using eight links per interconnect. This introduces a layer of network heterogeneity.

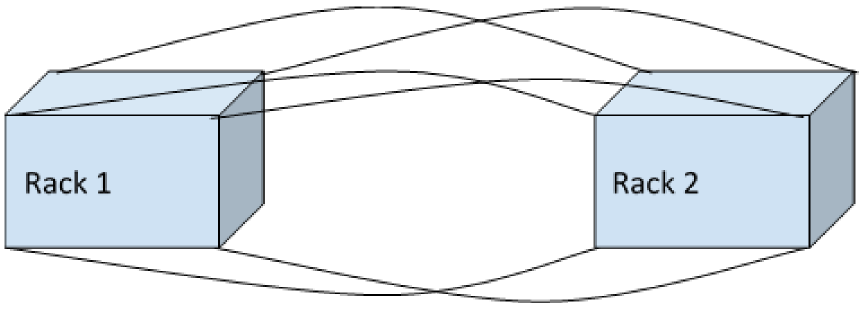

Figure 36: Falcon 5D Topology.

Again, 6D is simply the combination of two 5D units. However, there is again a network interconnect heterogeneity as shown below.

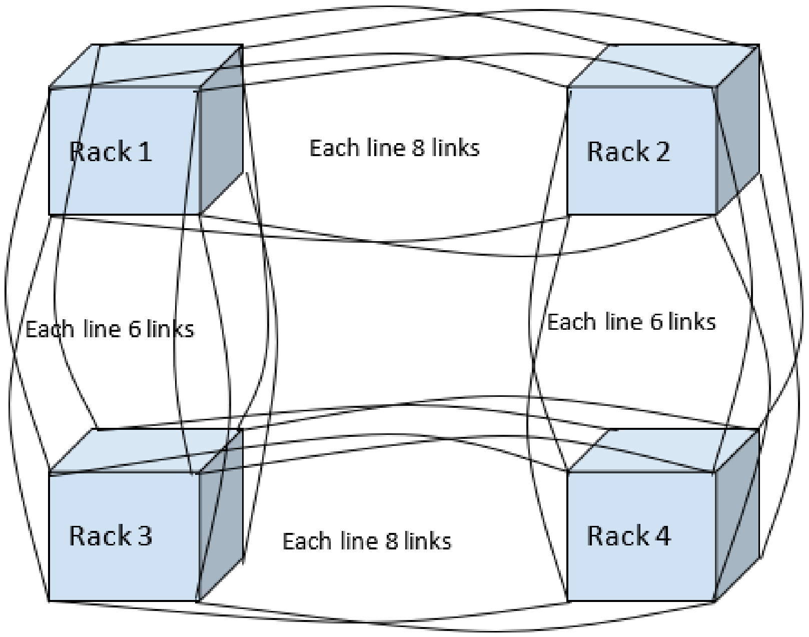

Figure 37: Falcon 6D Topology.

Finally, this 7D structure is created from two 6D units. However, there are insufficient nodes to do so. As such, the 7D structure is incomplete and heterogenous in hardware and link type as shown in the figure below. This makes the 7D structure only partially complete and, as such, degrades some of the functionality of a hypercube.

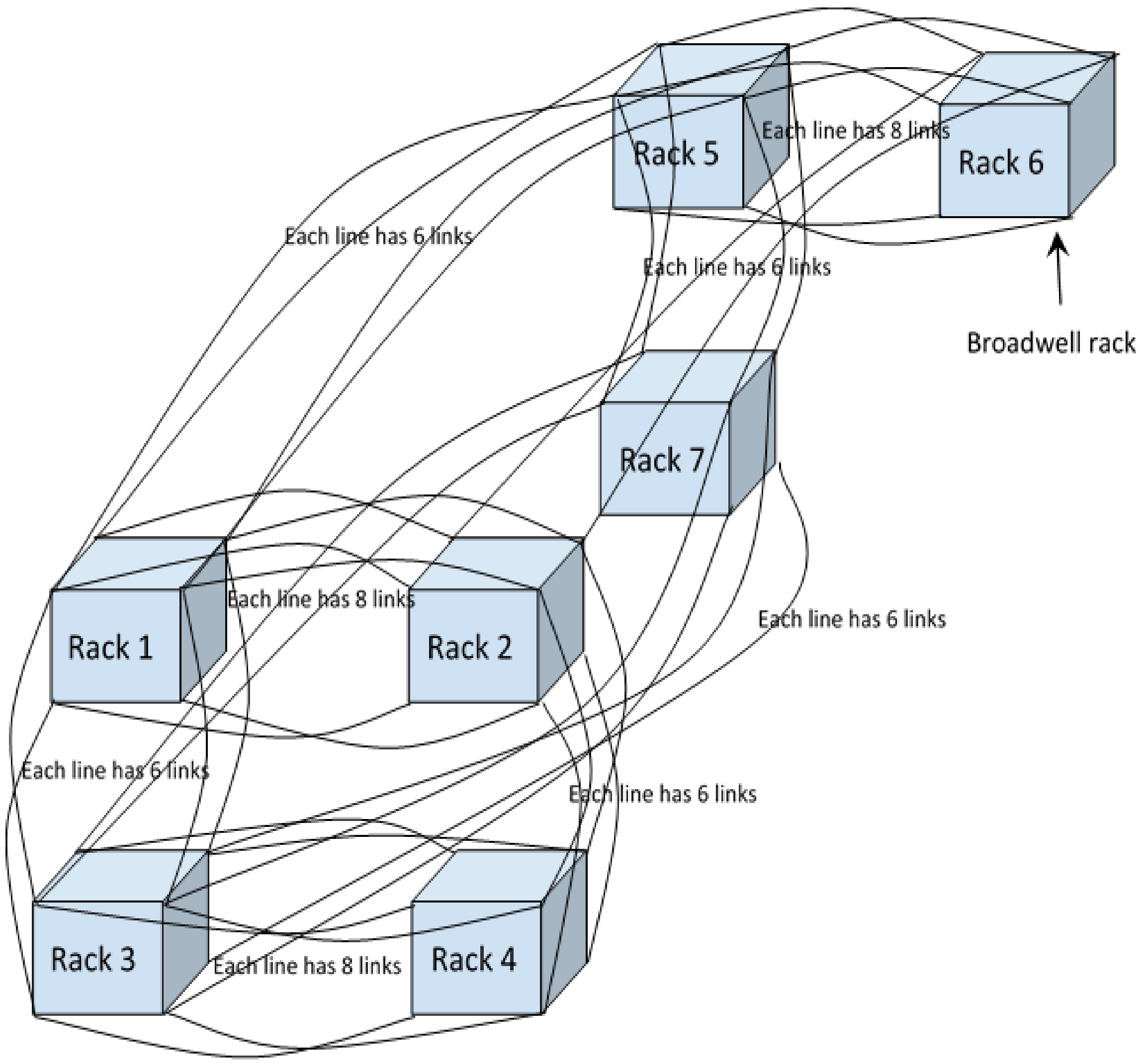

Figure 38: Falcon Partial 7D Topology.

APPENDIX B – FISSION HPC DATA

The Fission HPC at INL is the predecessor to Falcon. It is a 12,512-processor system with a Fat-Tree topology. The plots presented below are analogous to those in Section 5.3.1 with the exception that they were produced on Fission rather than Falcon.

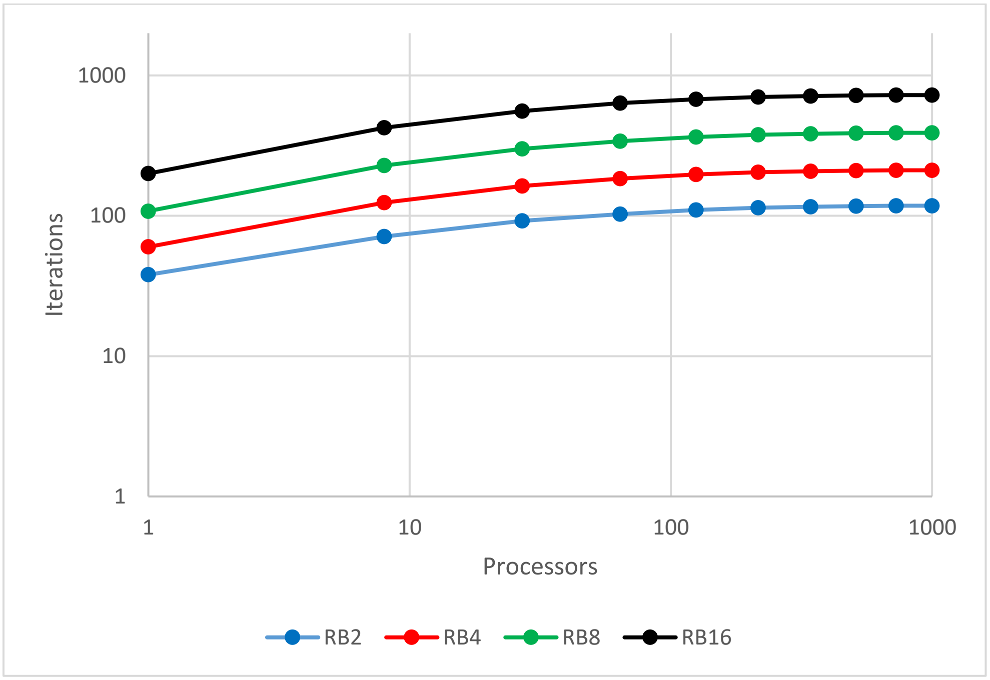

Figure 39: PIDOTS / Fission Iteration Behavior.

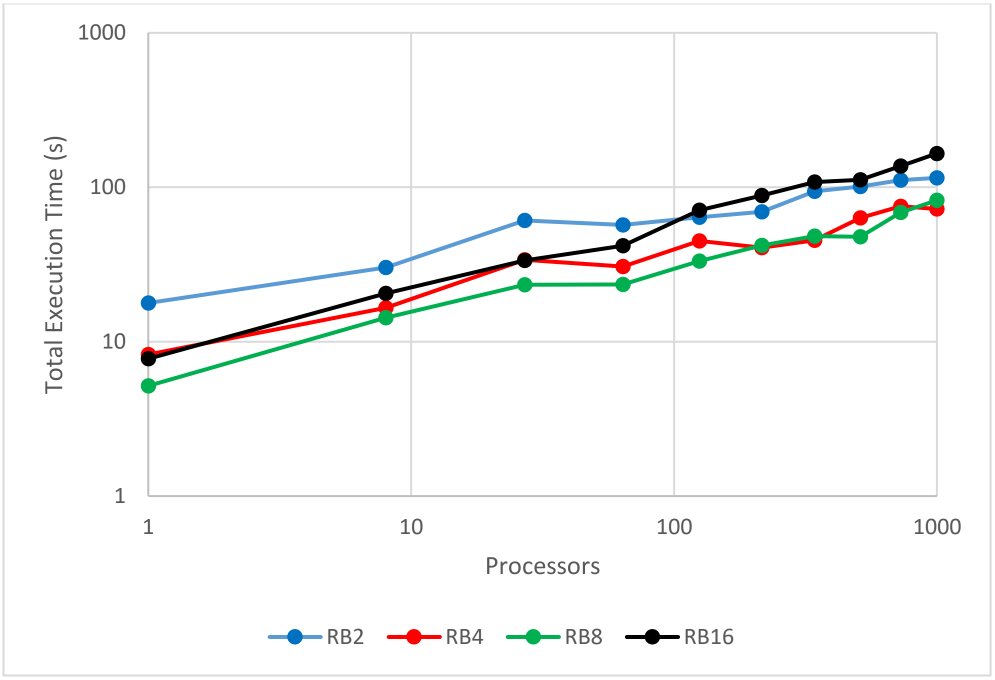

Figure 40: PIDOTS / Fission Total Execution Time.

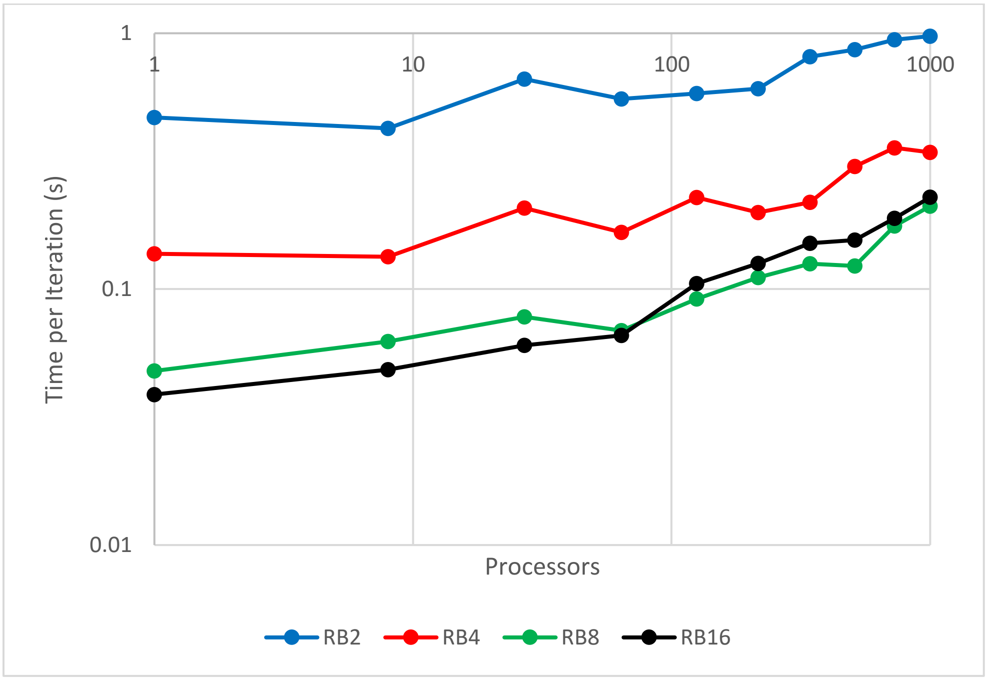

Figure 41: PIDOTS / Fission time per iteration.