Improved MCNP Memory Locality by Neutron Grouping

A Thesis

Presented in Partial Fulfillment of the Requirements for the

Degree of Master of Science

with a

Major in Nuclear Engineering

in the

College of Graduate Studies

University of Idaho

by

Aaron Bly

May 2014

Major Professors: Akira Tokuhiro, Ph.D. and Robert Hiromoto, Ph.D.

Authorization to Submit Thesis

This thesis of Aaron Bly, submitted for the degree of Master of Science with a major in Nuclear Engineering and titled "Improved MCNP Memory Locality by Neutron Grouping," has been reviewed in final form. Permission, as indicated by the signatures and dates given below, is now granted to submit final copies to the College of Graduate Studies for approval.

Major Professors: ________________________ Date______________
Akira Tokuhiro

________________________ Date______________
Robert Hiromoto

Committee
Member: ________________________ Date______________
Fatih Aydogan

Department
Administrator: ________________________ Date______________
Lee Ostrom

Discipline's
College Dean: ________________________ Date______________
Larry Stauffer

Final Approval and Acceptance

Dean of the College
of Graduate Studies: ________________________ Date______________
Jie Chen

Abstract

This research presents new code for Monte Carlo N-Particle (MCNP) to achieve an improved time during criticality calculations. Modifications implementing the grouping and sorting of neutrons takes advantage of memory locality by processing all neutrons in a group to achieve the temporal reuse of cross section data. This prevents unnecessary data lookups. Various groupings and their results are compared.

The modified code utilizing neutron energy groups provided the best result of a 16.7% ± 0.5% speedup for a criticality determination of a two slab tank experiment. This is a savings of 2 ½ hours for a system that normally takes approximately 15 ½ hours to execute. The code implemented was chosen to require minimal modifications to the MCNP program thus avoiding the need to rewrite a new version. Verification and validation is still needed in order to show that a speedup using neutron groups can be achieved in all cases.

Acknowledgements

The author would like to thank his wife and family who have put up with long study hours and for supporting him in every way. He couldn't have done it without them. Also he is indebted to Dr. Robert Hiromoto for his guidance and insights during the programming of the changes in the code. He would also like to thank Dr. Akira Tokuhiro for his review and helpful comments in writing this document.

Table of Contents

Authorization to Submit Thesis ii
Abstract iii
Acknowledgements iv
Table of Contents v
List of Figures vii
List of Tables viii
Chapter 1: Introduction 1
Chapter 2: Neutron Interaction with Matter 5
2.1 Introduction 5
2.2 The Neutron 5
2.3 Concept of Cross Sections 5
2.4 Neutron Interactions 6
2.5 Variation of Cross Sections with Neutron Energy 9
2.6 Effective Neutron Multiplication Factor 10
2.7 Summary 10
Chapter 3: Monte Carlo N-Particle 11
3.1 Introduction 11
3.2 MCNP Input Cards 11
3.3 KCode Calculation 12
3.4 Cross Section Data 12
3.5 MCNP Flow 13
Chapter 4: Improvement of Memory Locality Techniques 18
4.1 Introduction 18
4.2 Memory 18
4.3 Compiler Optimizations 19
4.4 Inlining 20
4.5 Register Utilization 20
4.6 Prefetching Operands 21
4.7 Loops 21
4.8 Scalar Optimizations 22
4.9 Vectorization 23
Chapter 5: Application of Neutron Groupings 24
5.1 Introduction 24

5.2 Source Generation Modifications 24
5.3 Material Bins 25
5.4 Energy Bins 26
5.5 New MCNP Flow 27
5.6 Neutron Groups and Cache Effects 32
5.7 Code Changes and Results for Energy Groups 35
5.7.1 Store Neutron in Energy Group 35
5.7.2 Arrays of Memory Pointers 40
5.8 Code Changes and Results for Material/Energy Groups 42
5.9 Side by Side Comparison of Code Set Results 49
5.10 Complex Configuration Designs 51
Chapter 6: Conclusion 53
References 55
Appendix A 57
Appendix B 63
B.1 Introduction 64
B.2 trnspt.F90 64
B.3 newSource.F90 65
B.4 Energy Group banked_particle_mod.F90 65
B.5 Energy Group hstory.F90 69
B.6 Energy Group bankit.F90 69
B.7 Material/Energy Group banked_particle_mod.F90 71
B.8 Material/Energy Group hstory.F90 76
B.9 Material/Energy Group bankit.F90 77
Appendix C 80

List of Figures

Figure 2-1: Scattering Event 7
Figure 2-2: Absortion Event 8
Figure 2-3: Types of Cross Sections 8
Figure 2-4: Cross Section vs Neutron Incident Energy 9
Figure 3-1: Overview MCNP Process Flow 14
Figure 3-2: Detailed MCNP Flow 16
Figure 3-3: MCNP Process Pseudo Code 17
Figure 4-1: Loop Code Motion 22
Figure 4-2: Exponentiation Replaced by Multiplication 22
Figure 4-3: Multiplication Replaced by Addition 23
Figure 5-1: Energy Group Determination 26
Figure 5-2: New Overview MCNP Process Flow 28
Figure 5-3: New Detailed MCNP Flow 30
Figure 5-4: New MCNP Process Pseudo Code 31
Figure 5-5: MCNP Cache Hit/Miss 1 32
Figure 5-6: MCNP Cache Hit/Miss 2 32
Figure 5-7: MCNP Cache Hit/Miss 3 33
Figure 5-8: Neutron Group Cache Hit/Miss 1 34
Figure 5-9: Neutron Group Cache Hit/Miss 2 34
Figure 5-10: Banked_Particle Complex Type 36
Figure 5-11: Energy Group Arrays 37
Figure 5-12: Neutron Storage in Groups 38
Figure 5-13: Neutron Retrieval from Groups 39
Figure 5-14: Neutron Storage with Pointers 40
Figure 5-15: Array of Pointers Showing Gather/Scatter 42
Figure 5-16: Geometry of Criticality Design used in Research 43
Figure 5-17: Material Cells 44
Figure 5-18: Material Group Lookup 45
Figure 5-19: Material/Energy Group Arrays 46
Figure 5-20: Material/Energy Group Neutron Storage and Retrieval 47

List of Tables

Table 2-1: Neutron Energy Ranges 9
Table 5-1: Run Time Comparisons.......... 49
Table 5-2: k_{eff} Comparisons.......... 51

Chapter 1: Introduction

The ability to model nuclear systems that reach criticality and determining which configurations will provide the desired outcome (safety margin and performance) is a complex and, oft times, long process. Monte Carlo simulations can provide results that give a close approximation to real historical and experimental data for neutron transport. Software has been developed to provide a means to simulate the Monte Carlo methods for more detailed designs.

Software programs, such as Monte Carlo N-Particle (MCNP), have been created to provide a way to automate the calculations. However, due to the lengthy computation time that the Monte Carlo method requires it is still difficult to use in every day modeling. Smith (2003) tells of interest by the reactor physics community in performing a full-core Monte Carlo analysis. Smith predicted that a full-core Monte Carlo calculation would take 5,000 hours on a 2-GHz PC and, according to Moore's Law, could not be performed in less than one hour until the year 2030. Martin (2007) re-analyzed the topic at the 2007 ANS Mathematics & Computation Conference and concluded that it would be 2019 when such a full reactor core calculation could be accomplished in one hour by making use of a 1500-core processor. The availability of a 1500-core processor isn't a reality to every engineer.

Research and development has been pursued to develop ways to speed up the calculations allowing engineers the ability to take advantage of computer–based modeling without having to wait for long periods of time. Many have worked on modifying MCNP to optimize its performance and thus speed up the process.

MCNP has been modified by rewriting the code to work on parallel processors. McKinney and West (1993) researched the idea of running on an IBM RS6500 cluster with speedups of 13.3x faster. Others have applied MCNP to parallel processors as well. Hadjidoukas et al. (2010) achieved speedup of results of 61.16x with a parallel processor code rewrite of the MC4 software and Carstens (2004) got a speedup of 20 – 30x with a Beowulf Cluster of parallel processors.

Another optimization process has been to work with Graphics Processing Units (GPU) to achieve their results. Brown, et al. (2012a) reports speedups of up to 33.3x and 64.0x using a single or dual GPU environment. Nelson (2009) stated in his thesis of speedup results, on a GPU converted code, of 23.91x over standard central processing unit (CPU) runs. Meanwhile Gong, et al. (2011) reported speedups of a factor of 16.3 – 23.67x compared to single core CPU. Brown, et al. (2012b) stated that in order to take full advantage of GPU technologies, many challenges related to the hardware and software must be carefully understood and addressed. Furthermore, it must be kept in mind that some of the deficiencies and constrains in existing GPUs will likely be mitigated in future-generation products, leading to exponentially improved and perhaps unexpected computing power when compared with its CPU counterparts.

MCNP has shown increased speed by being rewritten to allow vectorization of the calculations by Brown and Martin (1984). Their results show speedups of at least 20 – 40x over scalar calculations utilizing the vectorized code.

All this previous research into optimizing MCNP involves intensive rewrites of the program to take advantage of the increased speed. Also, the rewritten code is very specific to the system it is running on and would not be able to be run on another system unless it had the same configuration.

Brown and Martin (1987), in using a vectorized code to implement Monte Carlo methods, discussed that speedups are achieved from the organization of neutrons into groups they called supergroups. These supergroups allow for the use of the same cross section data without having to jump around memory looking for the data when the neutrons don’t have similar characteristics. Brown and Martin also mentioned the issue of moving the data that represents the neutron from group to group counteracting the benefits of having the groups. As cross section data has to be removed from memory in order to handle the memory position swapping of the neutron. In order to counter that problem they discussed placing pointers in the groups that directs the neutron data’s location and only moving the pointers from group to group. These methods take

advantage of temporal and spatial locality of data in memory and are the main topic of the research discussed here.

Siegel, et al. (2013) also investigated the need for memory optimizations resulting in better use of locality in memory. Siegel, et al. state the cross section probabilities are strongly dependent on the precise energy of the neutron, and thus, as a neutron jumps around in energy from interaction to interaction, the calculation involves frequent, nearly random access to very large read-only lookup tables, something which presents significant performance challenges when executing simulations on modern CPUs. The significance of the need to explore memory optimizations can also be seen in their discussion that for robust reactor calculations the cross section data loads can consume up to 85% of the total application time, and typical integration times of thousands of particles per second can make highly detailed calculations impractical. The program used to test their theory was OpenMC.

The commonality of all the approaches described above requires an extensive restructuring of the Monte Carlo particle transport code as currently implemented in the MCNP program.

The objective of this thesis is to demonstrate that a more optimal sequential MCNP performance can be obtained with minimal programming effort by organizing the neutrons in to groups.

The methods discussed in this document have been applied with slight modifications to the original MCNP code allowing it to run on various systems without the need to be rewritten each time you want to change the system that you are using. Also, more engineers can benefit from a faster running code on their normal desktop computers.

To provide enough background information to the reader, a brief discussion on the basic nuclear engineering aspect of neutrons and cross section interactions is provided. The MCNP code is describe and used as the basis for comparing structural programming changes made in this thesis. Details of the various methods used are

provided. Finally, the process and reasoning behind the choices made for the additions to the MCNP code and the results achieved are illustrated.

Chapter 2: Neutron Interaction with Matter

2.1 Introduction

A brief explanation of concepts regarding the interactions of neutrons with fuel and non-fuel materials is given in this chapter. For the reader well versed on interactions please skip to the chapters where the Monte Carlo N-Particle (MCNP) code will be discussed.

2.2 The Neutron

An atom consists of 3 sub-atomic particles called electrons, protons, and neutrons. Electrons are described as circling the nucleus of the atom while protons and neutrons make up the nucleus. Neutrons are necessary within an atomic nucleus as they bind with protons to hold the nucleus together via the nuclear force or binding energy. While bound neutrons in nuclei have the possibility to be stable, free neutrons are unstable and undergo decay, thus the cause for radiation.

The neutron was discovered in 1932 by James Chadwick, and in 1933 it was hypothesized by Leo Szilard to be the cause for nuclear chain reactions, and that nuclear reactions can self-perpetuate. During the 1930's, when nuclear fission was discovered, it became clear that if the process also produced free neutrons, this could produce the neutrons needed for a fission chain reaction. The kinetic energy of the fission fragment is the method that generates the energy, in the form of heat, used in energy conversion in a nuclear power plant.

2.3 Concept of Cross Sections

A cross section is the effective area that governs the probability that a nuclear reaction will occur. From nuclear physics the cross section is used as the probability of an interaction event between a neutron traveling through a material and that material's nuclides.

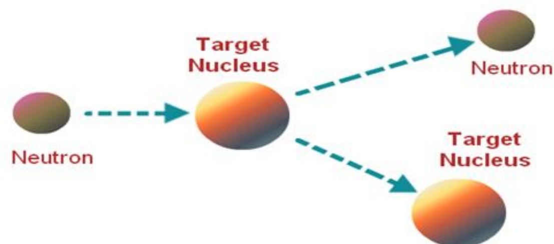
Cross sections are dependent on various parameters: energy of the neutron involved, material in which the neutron is traveling, energy of the material atoms, and the relative angle between neutron and target material nuclide. These parameters influence the cross sections by either increasing or decreasing the probability of the interaction events.

2.4 Neutron Interactions

Interactions of a free neutron traveling through other materials can be divided into two major phenomena with corresponding cross sections: scattering and absorption. The sum of these cross sections is called the total cross section. When a neutron is scattered by a nucleus, its speed and direction change but the nucleus is left with the same number of protons and neutrons it had before the interaction. The nucleus will have some recoil velocity and it may be left in an excited state that will lead to the eventual release of radiation. When a neutron is absorbed by a nucleus the atom will enter into an excited state and will attempt to reach a stable state by either releasing energy in the form of radiation or fission can be induced.

Scattering events, see Figure 2-1, can be subdivided into elastic and inelastic scattering. In elastic scattering the total kinetic energy of the neutron and nucleus is unchanged by the interaction. During the interaction, a fraction of the neutron's kinetic energy is transferred to the nucleus thus slowing down the neutron. Inelastic scattering is similar to elastic scattering except that the nucleus undergoes an internal rearrangement into an excited state from which it eventually releases radiation.

Elastic



Inelastic

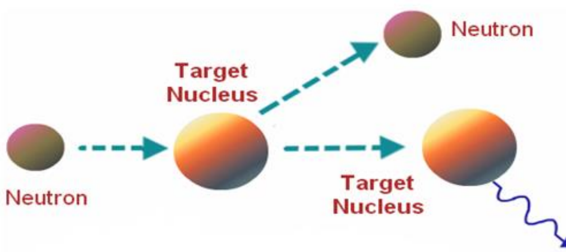
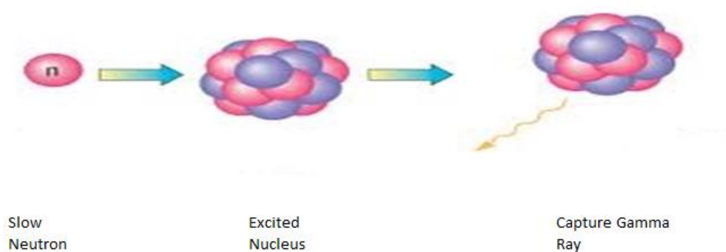


Figure 2-1: Scattering Event

Instead of being scattered by a nucleus, the neutron may be absorbed as shown in Figure 2-2. A variety of emissions may follow. The nucleus may rearrange its internal structure and release one or more gamma rays. Charged particles may also be emitted. The more common charged particles are protons, deuterons, and alpha particles. The nucleus may also rid itself of excess neutrons. The emission of only one neutron is indistinguishable from a scattering event. If more than one neutron is emitted, the number of neutrons now moving through the material is larger than the number present before the interaction; the number of neutrons is said to have been multiplied. Finally, there may be a fission event, leading to two or more fission fragments (nuclei of intermediate atomic weight) and more neutrons. Rinard, (1991).

Capture



Fission

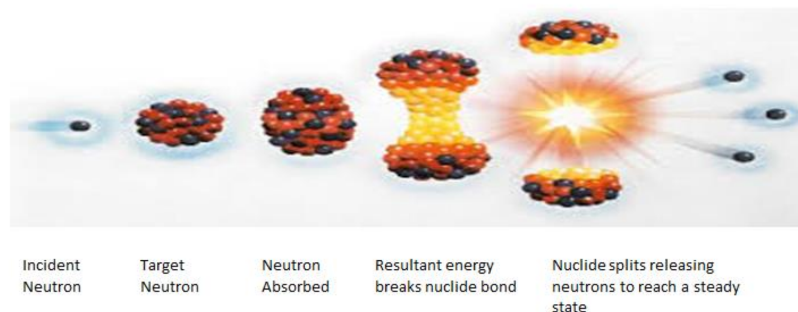


Figure 2-2: Absorption Event

Figure 2-3 shows the types of cross sections and how they are related to one another.

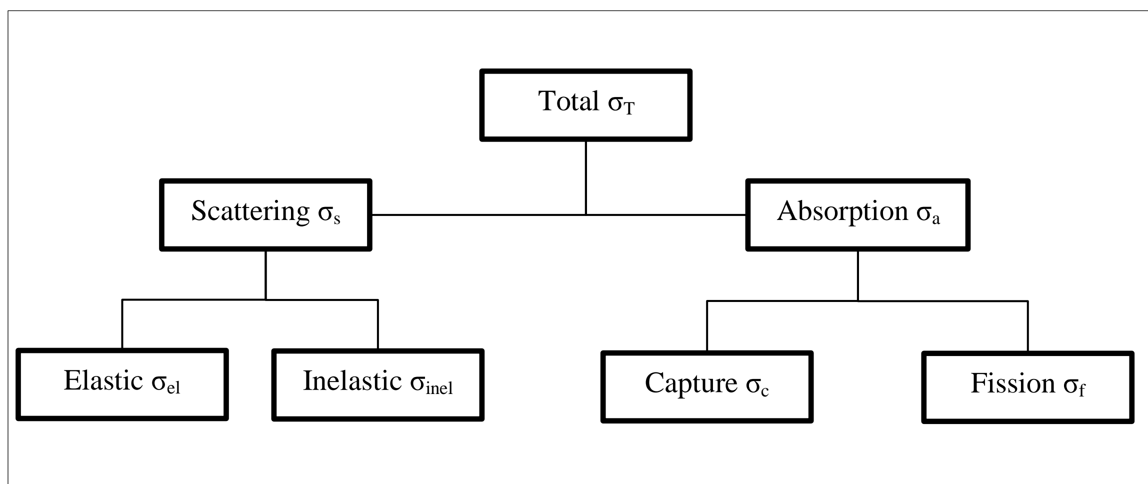


Figure 2-3: Types of Cross Sections

2.5 Variation of Cross Sections with Neutron Energy

There are four regions that a neutron can be classified under due to its energy: fast, resonance, epithermal and thermal.

Table 2-1: Neutron Energy Ranges

Neutron Energy Ranges	
Fast	100 keV to 10 MeV
Resonance	1 eV to 100 keV
Epithermal	0.1 eV to 1 eV
Thermal	0.025 eV (< 0.1 eV)

Figure 2-4 shows the relationship between cross section and incident neutron energy. For the most part the cross section decreases as an inverse of neutron energy (1/E).

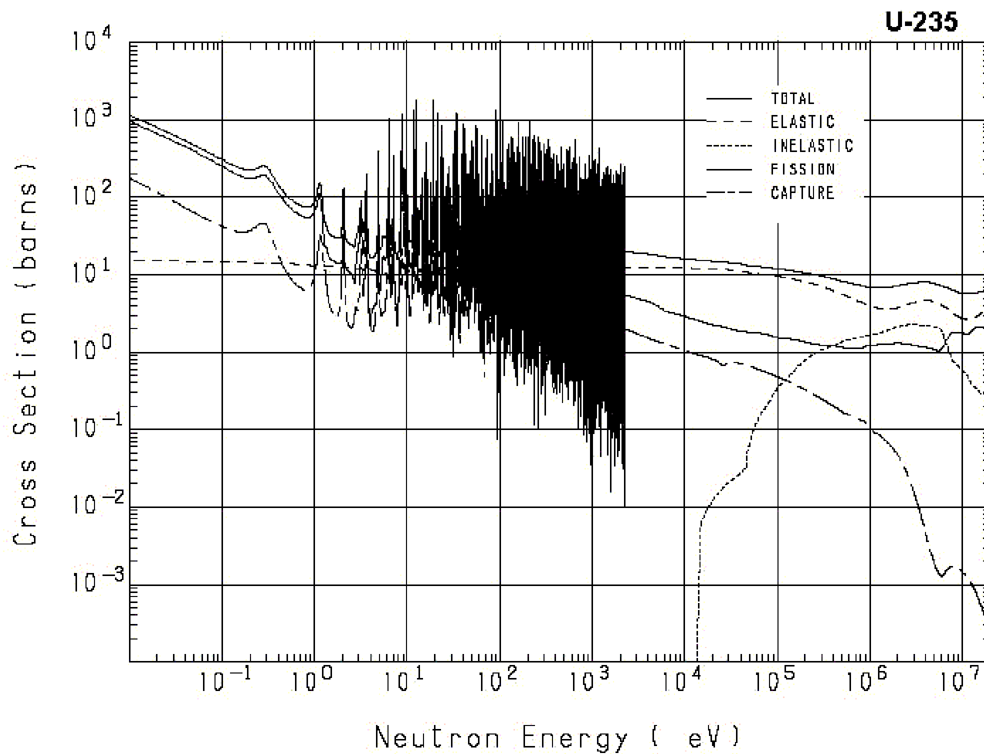


Figure 2-4: Cross Section vs Neutron Incident Energy

2.6 Effective Neutron Multiplication Factor

The effective neutron multiplication factor, k_{eff}, is the rate of neutron production divided by the rate of neutron absorption plus rate of neutron leakage from the system. The value of k determines how a nuclear chain reaction proceeds. (Glasstone and Sesonske, (1969)):

- $k_{eff} < 1$ (subcritical): The system cannot sustain a chain reaction, and any beginning of a chain reaction dies out over time. The rate of production of neutrons would be less than the rate of neutron loss.

- $k_{eff} = 1$ (critical): A steady state is achieved and just as many neutrons are produced as lost.

- $k_{eff} > 1$ (supercritical): More neutrons are being produced than are lost, and so the density (and fission rate) will increase continuously. Nuclear weapons are designed to operate under this state.

The k_{eff} is thus a value that classifies whether a system design is a critical system. MCNP is a software application that can run a series of calculations and then tally the results to determine a statistical value for $k_{eff.}$

2.7 Summary

In summary, neutrons are of major interest when studying or creating nuclear power. Cross sections are dependent on the material in which the neutron is traveling and the neutron's energy. The interaction events each have a cross section associated with them that together can add up to the total cross section.

Chapter 3: Monte Carlo N-Particle

3.1 Introduction

As found in the manual for Monte Carlo N-Particle (MCNP) written by X-5 Monte Carlo Team, (2003): The Monte Carlo N-Particle code is a general purpose program that can be used for modeling and calculating neutron, photon, electron, or coupled neutron/photon/electron transport. MCNP was written and developed by Los Alamos National Laboratory since at least 1957. The program is written in ANSI-Standard Fortran 90 software language. The program uses the Monte Carlo method, which is a numerical analysis technique that uses random sampling to estimate the solution of a physical or mathematical problem.

MCNP uses continuous-energy nuclear and atomic data libraries. The primary source of the nuclear data are evaluations from the Evaluated Nuclear Data File (ENDF) system, Advanced Computational Technology Initiative (ACTI), the Evaluated Nuclear Data Library (ENDL), Evaluated Photon Data Library (EPDL), the Activation Library (ACTL) compilations from Livermore, and evaluations from the Nuclear Physics (T–16) Group at Los Alamos. Evaluated data is processed into a format appropriate for MCNP by codes such as NJOY. The processed nuclear data libraries retain as much detail from the original evaluations as is feasible to faithfully reproduce the evaluator's intent.

3.2 MCNP Input Cards

The MCNP program requires an input file from the user where the user defines the problem geometry, specifies the materials and source, and states the results desired from the calculation. The geometry is determined by identifying cells that are bound by surfaces. Within these surfaces the cell can be either filled with a material or a void.

The input file consists of three major sections: cell cards, surface cards, and data cards. Note that "card" is used to describe a single line of input up to 80 characters and refers back to the usage of punch cards which were created to store data and program

code. However, today the program uses an electronic file, which makes it easier to design the problem. A section in the input file is made up of one or more cards and is delimited by a blank line to distinguish between sections.

The cell cards are used to define the shape and material content of the physical space. This is done by joining the surface cards in the next section of the input file. The data cards hold the information that describes the materials and source, which are both important for criticality calculations. These are only two of the many cards available in MCNP. For a full list, usage of, and formatting requirements for these cards can be found in the MCNP manual.

Appendix A has an example of an input card. It is the input card used for the testing of the optimizations used in this discussion.

3.3 KCode Calculation

MCNP can perform many types of calculations. The KCode calculation is of particular interest for this paper. This calculation determines the average k effective (k_{eff}) value for the given inputs to the program. This helps provide the criticality of the model described in the input file. These calculations can run for a very long time. The test case used in this discussion is a KCode calculation and the base run time is about 15 ½ hours. As you might expect that can cause delay for trying to study various reactor models.

The KCode information is found in the input card. It lists the amount of source particles wanted and how many cycles, or iterations, are wanted to repeat the problem in order to provide the average value of k_{eff} for a given configuration.

3.4 Cross Section Data

The input file holds a list of the material make-up of the different parts of the reactor. This list states the elements and their reference to the data libraries that contain the characteristics of the elements. These characteristics are what determine cross section groupings.

The data libraries containing this information have been determined and gathered through actual experimental processes. The data they store are the probability values for the various types of events that can occur between a neutron and another nucleus and is called a cross section.

At the initialization of the MCNP program it reads in the cross section data that is needed based on what is listed in the materials list. MCNP stores all the data it needs into a data array. At this point MCNP has a smaller amount of information to work with compared to having to store all the cross section data in the full library.

3.5 MCNP Flow

Once MCNP begins the actual calculation processes it begins by generating a source neutron. This neutron is then followed through its "lifetime" until it is either absorbed into a nucleus without a fission event or it leaks out of the system. This process of following the neutron is called the particle's history. See Figure 3-1 for a simplified flow of how the MCNP program works.

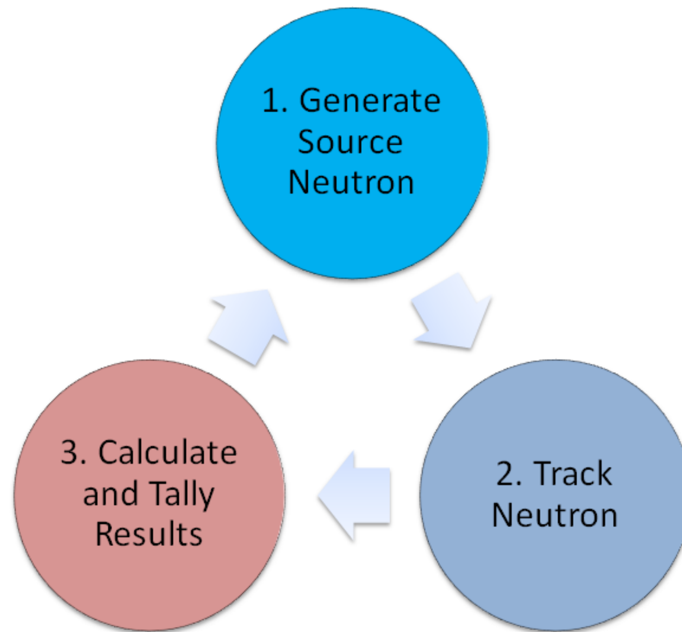


Figure 3-1: Overview MCNP Process Flow

The first thing that is done in the neutron history is the calculation of the cross section data for the given material and energy of the neutron. The program has to access the cross section data it has stored and make the calculation. The program makes the adjustments for energy (erg) and the cell / material (icl) that the neutron is currently in when making calculations with the cross section data.

The neutron is then advanced a random distance. The neutron position is measured to determine if the neutron has moved far enough to have entered into a new cell or material. If it has, then the process starts again of calculating the new cross section data. If, however, the neutron has not reached a boundary, it is now time to determine which interaction event, if any, will occur. The calculated cross section is used to determine which interaction occurs, because as was stated earlier the cross section data is a set of probabilities of which event is most likely to occur.

If an event occurs then the resultant effect is calculated. Scattering events change the direction and energy level of the neutron. Absorption ends the life of the neutron,

however if fission occurs, new neutrons are generated. All but one neutron is stored in a process called banking and then that neutron is followed through its history. After the current neutron finishes its history the program retrieves a banked neutron and starts processing its history and repeats the process stated above. This continues until all banked particles have either been absorbed or leak out of the system. Once the banked particles are used up the program generates another source neutron and begins the history again. This process continues until the numbers of desired source neutrons have been processed.

Each event is recorded and a running tally is kept so that the k_{eff} value can be calculated. Once the source neutron count has been reached and all neutron histories tracked, the cycle is finished and the k_{eff} value is determined. The program will continue until all cycles have been completed.

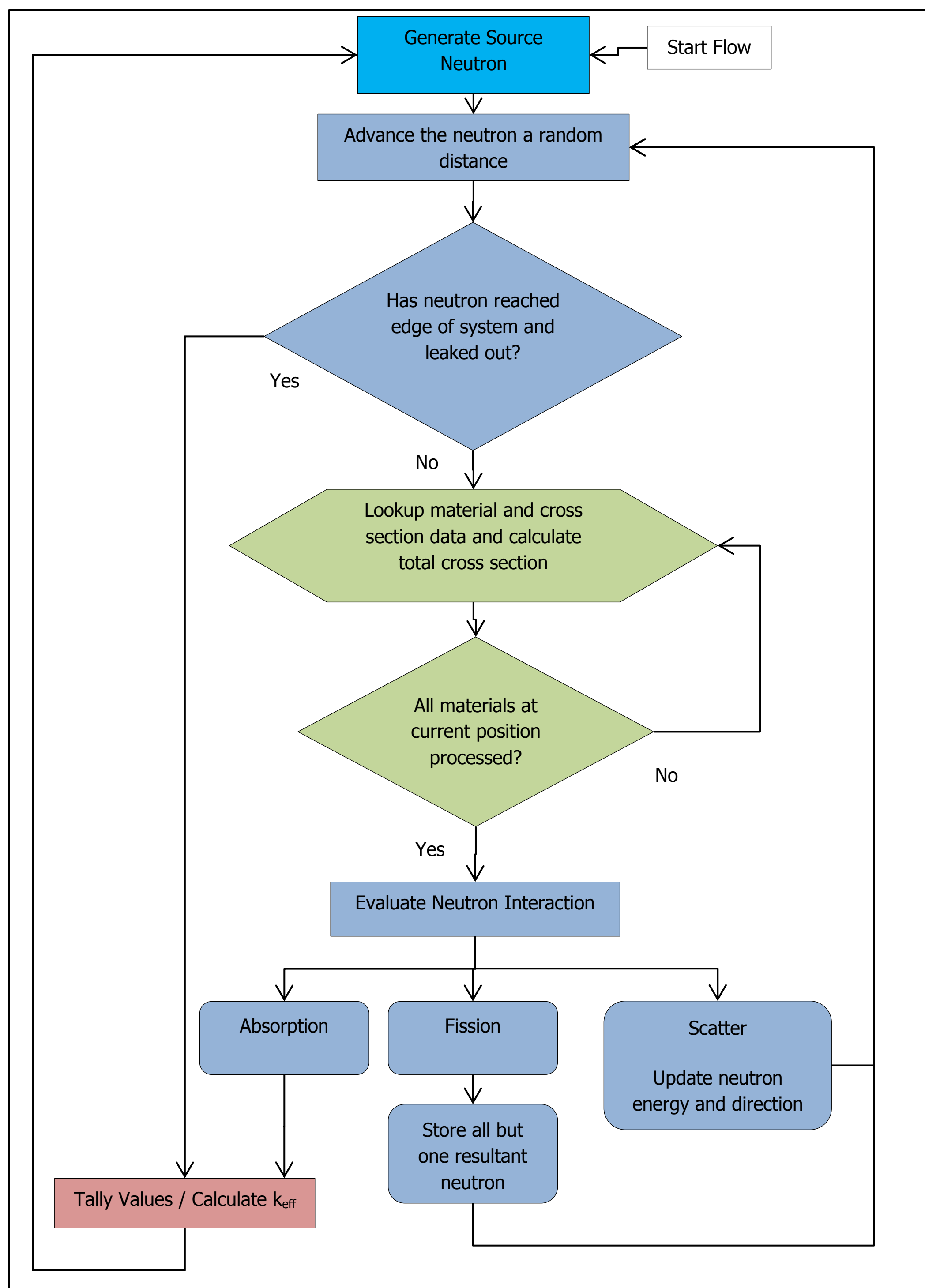


Figure 3-2: Detailed MCNP Flow

The history flow can be shown using high level pseudo code to show the process and how it loops in the program. See Figure 3-3 below.

MCNP Process Pseudo Code

```
1: for each count in desired particle do
2:     generate particle or retrieve stored particle if available
3:     repeat
4:         advance particle random distance in material
5:         lookup material at particle position
6:         for each type of material in current position do
7:             for each interaction event do
8:                 lookup x-section data
9:                 calculate addition to total x-section
10:            end for
11:        end for
12:        randomize interaction
13:        update particle properties (position and energy)
14:        if fission, store all but one particle
15:    until particle is absorbed or leaks
16     tally results for particle
17: end for
```

Figure 3-3: MCNP Process Pseudo Code

Chapter 4: Improvement of Memory Locality Techniques

4.1 Introduction

Various methods of optimization are discussed in this chapter. The methods used in this research code are based on compiler optimization techniques.

4.2 Memory

Memory consists of a hierarchy of locations that all have different speeds at which a program can access them. This hierarchy ranges from the slowest being accessing the data on the hard drive to next being main memory increasing speed by 10k and then the CPU cache-memory which is yet generally 100 times faster than that.

Hennesy and Patterson (2007) describe the most important property that is regularly exploited for memory as the principle of locality: Programs tend to reuse data and instructions they have used recently. A widely held rule of thumb is that a program spends 90% of its execution time in only 10% of the code. An implication of locality is that we can predict with reasonable accuracy what instructions and data a program will use in the near future based on its accesses in the recent past. The principle of locality also applies to data accesses, though not as strongly as to code accesses.

Two different types of locality are defined. Temporal locality states that recently accessed items are likely to be accessed in the near future. Spatial locality says that items whose addresses are near one another tend to be referenced close together in time.

When the processor finds a requested data item in the cache, it is called a cache hit. When the processor does not find a data item it needs in the cache, a cache miss occurs. A fixed-size collection of data containing the requested word, called a block or line run, is retrieved from the main memory and placed into the cache. Temporal locality tells us that we are likely to need this word again in the near future, so it is useful to place it in the cache where it can be accessed quickly. Because of spatial locality, there is a high probability that the other data in the block will be needed soon.

In order to utilize this optimization involving more efficient use of temporal and spatial locality computer code needs to be written in such a way that it stores and organizes the data with these locality references in mind.

Weinberg, (2005) confirmed the importance of memory organization in HPC when he mentioned that some concrete and arbitrary choices have to be made when implementing the formal definitions about how to count locality statistics, and some approximations have to be made to make the data acquisition process tractable for HPC applications.

A big issue to overcome in MCNP is the amount of data that exists in the cross section calculations and the method and speed in which it is accessed. A commonality mentioned in using GPUs to program was memory management. Nelson, (2009) stated that the cross-section lookup routine takes approximately 33% of the computation time. As well, Brown, et al. (2012a) had to pay close attention to where they needed to store the cross section data in order to not slow down their process.

4.3 Compiler Optimizations

The compiler can perform optimizations on the machine level code it creates. Compiler optimization is generally implemented using a sequence of optimizing transformations, which take a program and transform it to produce a semantically equivalent output program that now utilizes fewer resources [e.g., registers, cache lines, data accesses, etc.].

Most compilers have preset groupings of optimizations that can be performed by setting the appropriate flag. They are generally setup in a hierarchical manner with a base group and each group after that adds on new optimizations to those of the previous group.

Programs written in high-level languages, such as Fortran 90, do not efficiently map the access of operands or the utilization of the cache to the underlying computer hardware optimally. The compiler, in addition to translating the written program into

machine language, schedules the compiled instructions in such a way as to maximize the locality of operand references with prolonged temporal usage over as long a period as possible. The primary goal is to achieve minimal movement of the same date (operands and instructions) in and out of the CPU. Wadleigh and Crawford, (2000), and Levesque and Wagenbreth, (2011) discuss techniques used in order to achieve an optimized state. They are Inlining, Register Utilization, Prefetching Operands, Loops (Unrolling, Interchange, and Code Motion), Scalar Optimizations, and Vectorization.

4.4 Inlining

Programs are usually written with functions or subroutines that are used as reusable code. If a function is to be called several times then it usually written in a way that other code can call the function and get the expected outcome every time. Any changes to that function can be made in only one location where the code resides. All calls to this function automatically use the same updated code without ambiguities. This provides a much cleaner and easier code for upkeep.

Functions are typically much simpler in the number of lines of code and the required data arrays as compared by a subroutine (or procedure). When calling a function or a subroutine, the operating system must set aside a stack area and temporary registers sets, along with a pointer back to the program's calling-site. For a function, this overhead can be more costly in setup time than the actual execution time. For this reason, the compiler may copy the body of the function at the point of the function call and remove the call altogether. This inclusion of the code is referred to as function inlining.

4.5 Register Utilization

As described earlier, performance is tied to data access and latency. Registers are placed in the close proximity of the processor's CPU limited in numbers and capacity but has low access latencies, and reside in the close proximity of the processor's CPU. The closest (and smallest) storage areas are registers that are contained in the processor. These registers have little latency and provide additional high speed memory for

multiple, independent instruction streams. This allows the compiler to schedule instructions so that pipelining is more efficient.

4.6 Prefetching Operands

Prefetching upcoming operands, or values, during the processing of current code and data blocks can provide optimization since the latency of fetching operands to the cache is so much longer than fetching them from memory. This means the code needs to be written or the compiler needs to be able to interpret the code in order to guess what will be needed next.

A difficulty can arise in doing this as the operands have to be placed in to the same cache that is being used to hold the data that is currently being processed.

4.7 Loops

Most of the optimizations for loops are related to increasing the amount of data reuse – taking advantage of the spatial and temporal locality of the data in memory.

Loop unrolling is when the value of the loop index is known and constant, the loop may benefit from flattening the loop and getting rid of it entirely. Consider a simple loop that loops four times to add values together. It might be beneficial to just write the four lines of code to add them as opposed to creating the overhead to handle the loop.

Nested loops appear in most significant computational-intensive codes. Since there are many optimizations that can be done to loops, most compilers avoid attempts to optimize nested loops unless told to specifically by the user.

In nested loops the loops can oft times be interchanged to increase the number of unit stride array references. This helps performance due to cache reuse.

Nested loops can also undergo an Unroll and Jam technique. This refers to unrolling multiple loops and jamming them back together in ways to reduce the number of memory operations.

Code Motion is a term that describes the action of the compiler to move (reschedule) instructions in such a way as to make the utilization of registers more efficient. Code Motion can guarantee the availability of operands at the moment an instruction is to be executed or to hoist loop invariant operands and instructions before the beginning of the loop as a means to avoid multiple fetches from memory. In addition, the compiler applies code motion in such a way that the resulting code is still executed correctly. The example below shows that the loading of the scale variable Y takes place inside the Do loop. However, notice that value of Y is never updated during the execution of the loop body. We refer to Y in this situation as a loop 'invariant'. In Fig. 4-1 the Original Loop fetches Y from memory during each loop iteration. In the Optimized Loop the value of Y is stored in register 1 and becomes part of the instruction sequence within the loop:

Original Loop	Optimized Loop
Do i= 1,N	Load y into register 1
X(i) = X(i) * Y	Do i = 1,N
End Do	X(i) = X(i) * register 1
	End Do

Figure 4-1: Loop Code Motion

4.8 Scalar Optimizations

Compilers can often replace an explicit use of an operation with a less expensive iterative operation in a loop. In particular, a multiplication can be replaced with an addition, and an exponential can be replaced with a multiplication.

Below is an example of an exponentiation being replaced by a multiplication:

Original Exponentiation	Compiler Optimized Equivalent
Do i= 1,10	XTemp = X
A(i) = X**i	Do i = 1,10
End Do	A(i) = XTemp
	XTemp = XTemp*X
	End Do

Figure 4-2: Exponentiation Replaced by Multiplication

Similarly, the compiler can replace a multiplication with an addition:

Original Exponentiation	Compiler Optimized Equivalent
Do i= 1,10	XTemp = X
A(i) = X*I	Do i = 1,10
End Do	A(i) = XTemp
	XTemp = XTemp + X
	End Do

Figure 4-3: Multiplication Replaced by Addition

4.9 Vectorization

This optimization is where a program is converted from a scalar implementation, which processes a single pair of operands at one time, to a vector implementation which then processes one operation on multiple pairs of operands at once. The code is written to group and organize the data so that it can take advantage of the vector implementation.

The vector implementation takes advantage temporal and spatial locality as the same operation is used multiple times and does not have to be fetched for each pair of operands.

Brown and Martin (1984) demonstrated in their research the benefits of vectorization in their discussion on vectorized Monte Carlo calculations. They structured the code to extract long vectors of particles that could be executed in a single loop. Their results in all cases, for sufficiently large batch sizes, show speedups of at least 20-40x over non-vectorized calculations.

Chapter 5: Application of Neutron Groupings

5.1 Introduction

The changes in the MCNP code flow made by applying the optimizations mentioned earlier and their various results will be discussed in this section. The main ideas behind the modifications used in this study are to improve of memory locality, both temporal and spatial.

5.2 Source Generation Modifications

The first change made to MCNP was the source generation. As discussed earlier in Chapter 3, MCNP source neutrons are generated in a random pattern. The Monte Carlo method requires a significantly large number of source neutrons to achieve acceptably accurate results.

The code is modified to generate all source neutrons and store them for later use. The original code generates a single neutron that is then tracked through a history of events. After the completion of a neutron's history, the next source neutron is generated and its history tracked. The loop continues until all source neutrons have been tracked. The new code can be seen in Appendix B.2.

This new loop created a structure that the complier can take advantage of. In condensing the loop around a smaller section of the code the compiler used the techniques for inlining on the function calls to generate the neutron. Also, the compiler could use the prefetching optimization to move the data necessary to generate the pseudorandom values for the neutron into cache. This allows the program faster access to the values.

Each neutron is stored after generation so they can be accessed when it is time to run through each of their life cycle histories. This call is made in the code found in Appendix B.3. After all source neutrons have been generated the program can proceed

with the rest of the process. This optimization is a common optimization used in all variants of the code sets developed in this research.

5.3 Material Bins

As discussed earlier cross sections are dependent on the material that the neutron is traveling through. A grouping of neutrons by like material, in which they are traveling through, would allow the reuse of the cross section data for the current material that is already in the cache. Further dividing the material groups into respective energy groups uses the spatial locality aspect of memory optimization. If all the neutrons being processed at that time are of the same material and energy then the cross section data should be in cache and can be accessed faster.

The code modifications that were done to the MCNP code, groups the source particles as they are generated into groups of the same material. This is done during the source particle generation as the source will be located in one location for most calculations. In some instances you have multiple source locations and so the particles are grouped into their respective bins.

Looking at the MCNP input cards you will see that the critical configuration being modeled for the calculation is made up of material cells. All cell regions of the same material are grouped into the same bin.

In the configuration used during the test case for this research it was determined that there were nine groups. These nine groups make up the material cells in the design configuration. The ninth group being the void area where there is no material.

As mentioned this is done at the beginning of each cycle during the source particle generation. However, as the cycle is processed new particles are generated during fission events. As the new particles are generated they are stored or “banked,” as the code states, into the material groups until it is time to process them.

5.4 Energy Bins

Another factor that affects the cross section data needing to be accessed and stored in cache is the energy of the neutron. The MCNP code is modified to group the neutrons into subgroups based on their energy levels. The energy bins handled a range of energy levels.

The spread in energy ranges invoke the principle of spatial locality. The ranges being accessed at the given time provides the program better access to the data as it is small enough to be stored in cache.

The energy bins used are related to the supergroups introduced by Brown and Martin, (1987) that closely match the energy ranges within the cross section data. The subroutine used to determine the energy bin to which the neutron belongs is illustrated in Figure 5-7 showing the use of 5 bins of energy ranges. When 10 bins were used the energy bins classify the neutrons into thermal (< 0.1 eV), epithermal (0.1 eV to 1 eV), and resonance (1 eV to 100keV) ranges. The other bins break up the fast neutrons into smaller groups. This code was used in all code combinations as each test code used energy bins.

Energy Group Determination

```
!code found inbanked_particle_mod.F90 – Appendix B.4
subroutine getErgGrp(tmpErgGrp)
   implicit none
   integer, intent(out) :: tmpErgGrp
     if(erg < 1.0) then
         tmpErgGrp = 1
      else if(erg >= 1.0 .and. erg < 2.0) then
         tmpErgGrp = 2
      else if(erg >= 2.0 .and. erg < 3.0) then
         tmpErgGrp = 3
      else if(erg >= 3.0 .and. erg < 4.0) then
         tmpErgGrp = 4
      else
         tmpErgGrp = 5
      end if
end subroutine getErgGrp
```

Figure 5-1: Energy Group Determination

The groups were arbitrarily decided in an attempt to match how the cross section data might be organized. As the organization of the data was unknown and again not wanting to have to rewrite the data in order to organize and know the exact groupings, the 2 sets of groups were chosen to prove that neutron groups could be used to optimize MCNP.

The ability to go back to the cache data for each neutron in the current group allows for the principle of temporal locality. The program now has a better chance of finding the data it needs for the current neutron in the cache and doesn't spend time retrieving more data from the slower memory.

5.5 New MCNP Flow

MCNP's original code would generate a neutron and then follow it through its life cycle and store any particles created during fission events. Then once the source neutron was absorbed or allowed to leak out of the system the stored particles would be processed until all were done. At that time a new source particle was generated. See figure 2 for the diagram of the flow.

The changes made to the original code resulted in a change to the flow of the MCNP process.

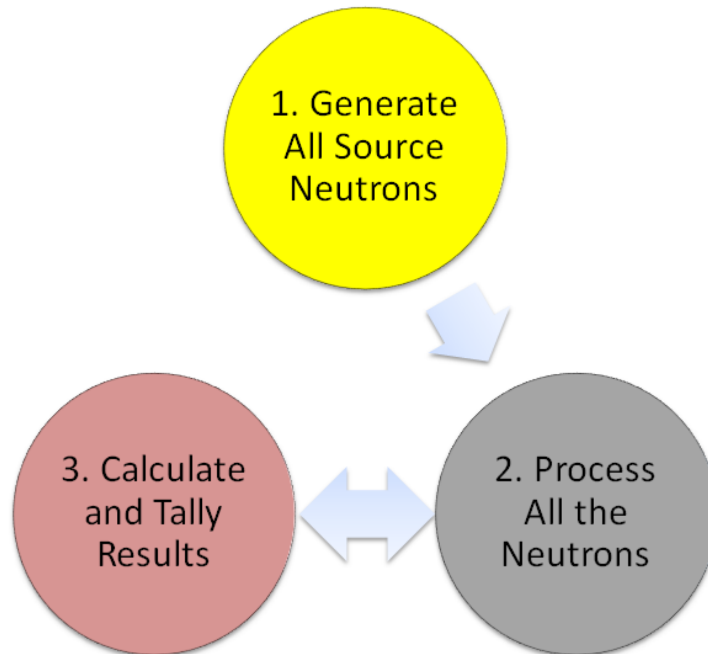


Figure 5-2: New Overview MCNP Process Flow

This new flow differs from the original flow in that all the source particles are generated at once in the first section. Then a neutron is retrieved from the highest energy level in the current group that still has neutrons stored. The neutron life cycle is then processed in the same manner as before with the exception that the neutrons that are generated in fission events are stored back into the same bins that already exist and are processed when they are retrieved after determining the current material and energy group.

The other change you can notice in figure 5-1 is that the flow has a back and forth process between section 2 and 3. This is to represent that after tallying up the results from each neutron the code just goes back to the storage bins and gets the next neutron. It only goes back to section 1 when all neutrons have been processed so it can start the whole process again in the new cycle.

Figure 5-2 displays the more detailed flow with the changes in the flow marked in bold. The flow starts off by generating all source neutrons that the test configuration states. Once each neutron is generated it is stored in its energy group. Then once all neutrons have been stored it proceeds to grab the neutron from the first energy group. The flow then proceeds in the normal method that the original MCNP follows. The next difference occurs when an interaction event occurs. If there is fission all but one neutron is stored in their respective energy groups. After returning from the interaction code the neutron is checked to see if it has the same energy group as the current group being processed. If it does it continues on, if not it is stored and the next neutron in the current energy group is retrieved and the process continues until all neutrons have been absorbed or leaked.

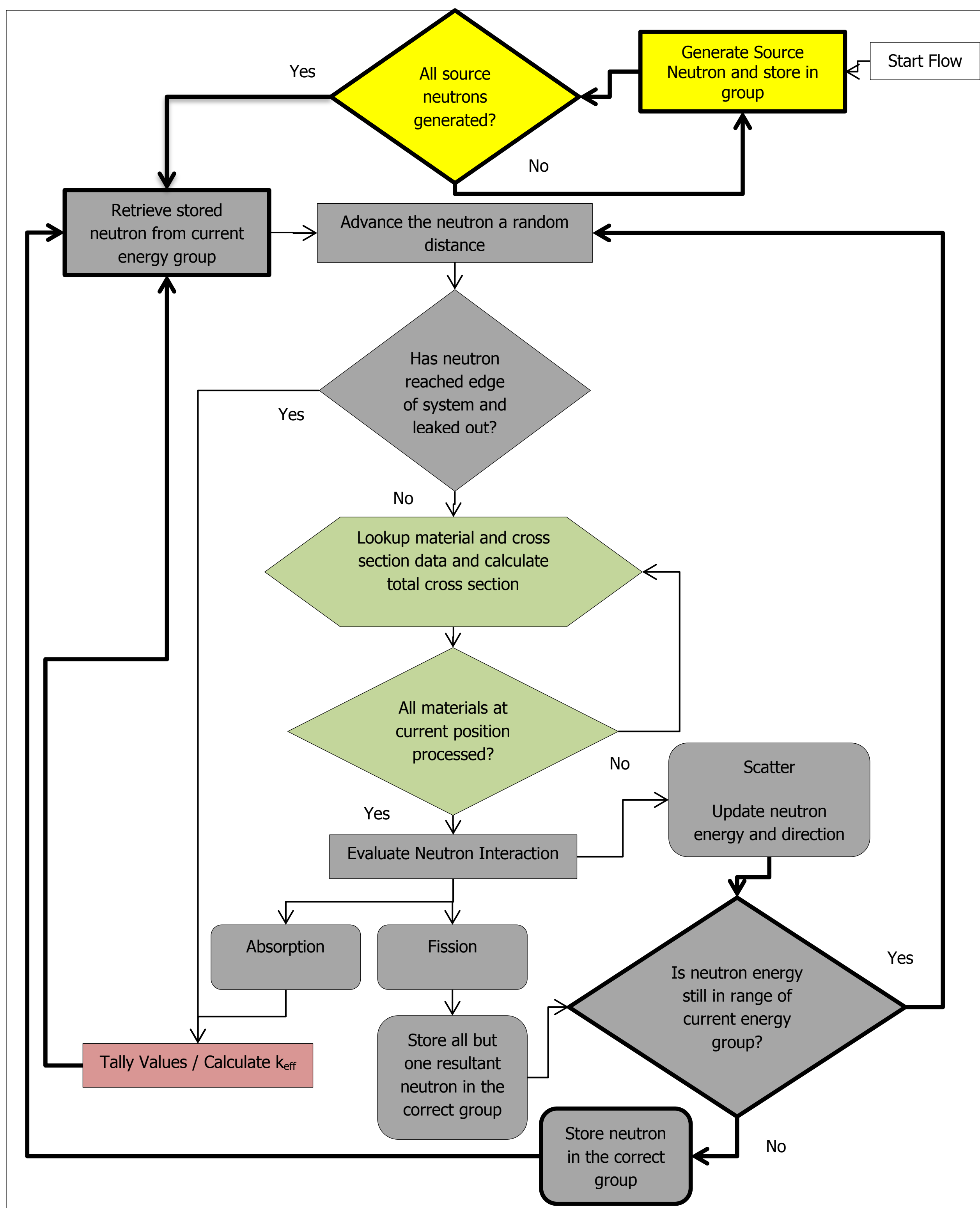


Figure 5-3: New Detailed MCNP Flow

The new flow can also be shown using high level pseudo code to show the process and how it loops in the program. See Figure 5-4 below.

```
New MCNP Process Pseudo Code
1: for each count in desired particle do
2:       generate particle
3:       store particle in material/energy group
4: end for
5: for each material/energy group do
6:       for each particle in group do
7:             repeat
8:                   retrieve stored particle
9:                   advance particle random distance in material
10:                  lookup material at particle position
11:                  for each type of material in current position do
12:                        for each interaction event do
13:                              lookup x-section data
14:                              calculate addition to total x-section
15:                        end for
16:                  end for
17:                  randomize interaction
18:                  update particle properties (position and energy)
19:                  if fission, store all but one particle stored corresponding
                     group
20                   if material/energy has changed, store particle in new group
                     and retrieve next particle
21:            until particle is absorbed or leaks or leaves current processing
               group properties
22             tally results for the particle and calculate k_eff
23:      end for
24: end for
```

Figure 5-4: New MCNP Process Pseudo Code

5.6 Neutron Groups and Cache Effects

The organization of the neutrons into groups allows data in the cache to be reused. When an application looks up data it will first look in the cache. If the data is there, known as cache hit, it is used. If the data is not there, a cache miss, then the application must spend time to go out to other memory locations to read and retrieve the data that is needed.

Figures 5-5, 5-6, and 5-7 show a representation of how the original MCNP code accessed the cross section data in cache. The first neutron accesses the cross section in cache (X-Sec 2) but the next neutron needs different data for its calculations.

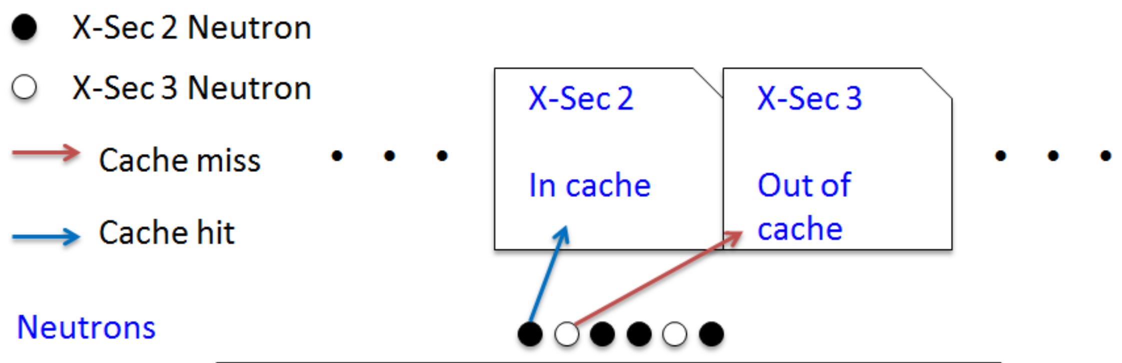


Figure 5-5: MCNP Cache Hit/Miss 1

The cache miss will cause the program to need to read in the new cross section data (X-Sec 3) and swap out X-Sec 2. Overhead is created having to spend time reading in the new cross section and swapping out the old.

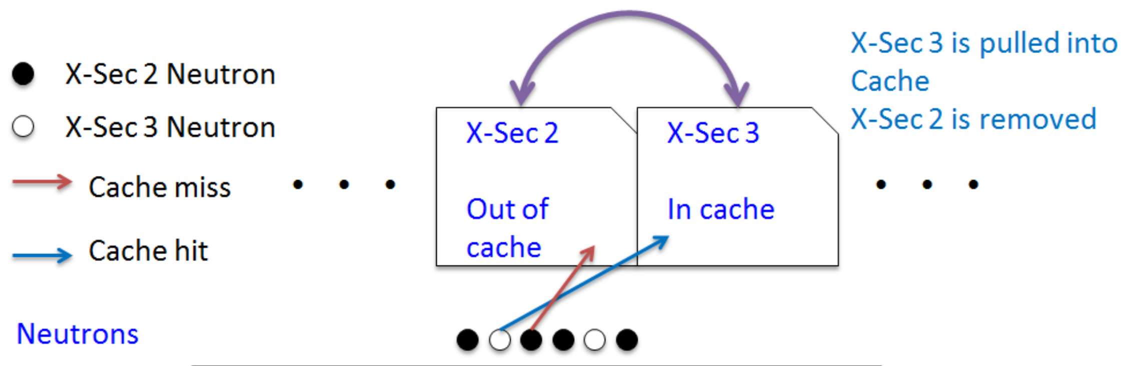


Figure 5-6: MCNP Cache Hit/Miss 2

Since MCNP is not sorted the next neutron could need X-Sec 2 data and again the program has to spend time swapping out the information.

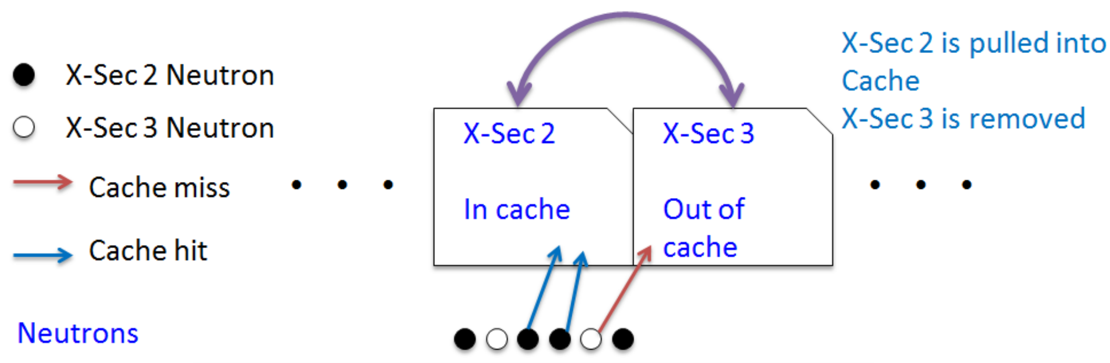


Figure 5-7: MCNP Cache Hit/Miss 3

The time spent can be considerable as each neutron using the Monte Carlo method is going to have random attributes and is very unlikely that it will be similar to the previous neutron or the next. In order to optimize the time spent in memory and help keep each access to the cache as a cache hit, the neutrons are organized in to groups.

The neutron groups force the data of each neutron to fit into a small range and thus the cross section data needed for each neutron can be found more often in the cache and save time from not having to read in the new cross section data for each neutron. This is show in Figures 5-8 and 5-9 where the neutrons in the first group can use the same X-Sec 2 data for their calculations. Then when the first group is processed the program moves on to the second group. The X-Sec data is swapped out but only once and then the second group can reuse that data for each of its neutrons.

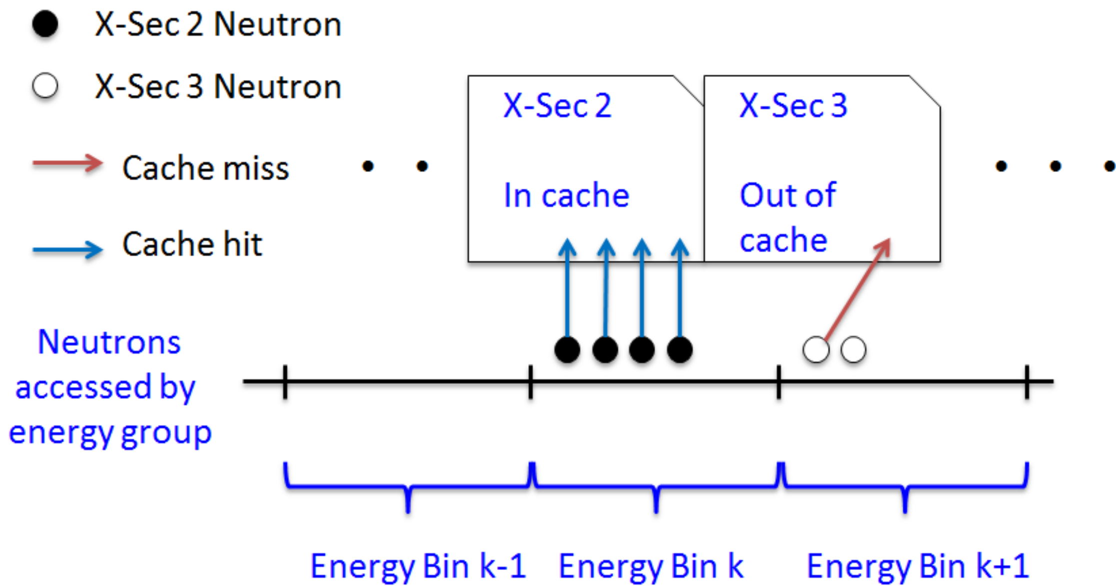


Figure 5-8: Neutron Group Cache Hit/Miss 1

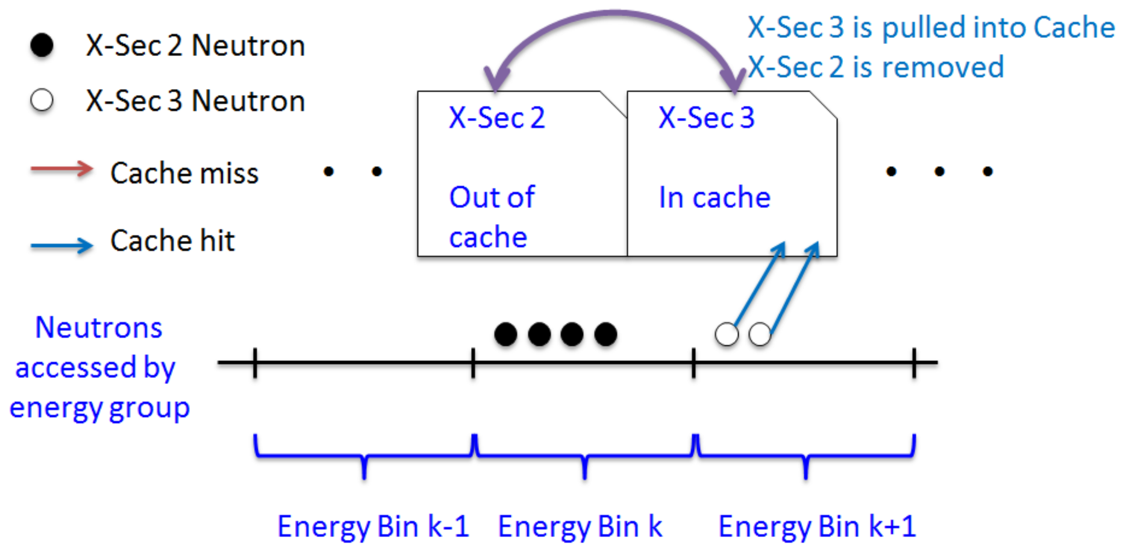


Figure 5-9: Neutron Group Cache Hit/Miss 2

A large number of bins can force more of the neutrons into the cross section table loaded into the cache. As the number of bins decrease, the number of neutrons increases

per bin and can result in a greater chance of a retrieved neutron that requires a cross section table that is not in the cache.

5.7 Code Changes and Results for Energy Groups

Several code changes were done in this thesis to achieve the best speedups with the least amount of changes. Special attention was paid during design to the knowledge, stated in Chapter2, that material and energy affect cross sections. The changes were done in order to maximize the temporal and spatial locality of the data to utilize the data most often in cache to provide the fastest access times. Small examples of the code changes are included in this section but to see where the code is used refer to Appendix B.

The original MCNP code was compiled using the delivered compiler optimization flag of O1. The results from this implementation form the basis for comparisons made to results reported here. The results of Base for the test design took 924.58 minutes. The resultant k_{eff} value was 0.99571. The k_{eff} is consistent with the value that is expected for the test design delivered by the MCNP team to test the installation of the MCNP program.

The new code was tested with eight different code sets each exploring different combinations of groupings for the neutrons. Each test was an attempt to find the most optimal method to arrange the neutrons so that the data for the cross section data needed for the current neutron and the next neutron would be available in the cache.

5.7.1 Store Neutron in Energy Group

The eight code sets are divided into two major grouping types. The first set of tests only takes into account the energy of the neutron and creates the groups based on a range of energy. The next major grouping type was to group the neutrons by the material they were in and then divided those further into energy bins. These bins were handled by data arrays.

In order to store the neutrons in their respective bins their attributes needed to be gathered and stored. MCNP makes use of global common block memory in the Fortran

code to store the attributes (energy, location, etc.) of the current neutron being processed. This works, as the Base code only runs one neutron from source generation to termination. In order to store the neutrons into their groups the use of a Fortran complex type was used. The complex type is equivalent to object oriented classes that exist in other languages such as Java, C#, etc. The complex type used was called a banked_particle and can be seen in part in Figure 5-10. It is just a collection of attributes that define the neutron.

Banked_Particle Complex Type

```
!code found in banked_particle_mod.F90 – Appendix B.4 and Appendix B.7
type banked_particle
   …
   real(dknd), dimension(:,:),pointer ::  bprtc
   real(dknd), dimension(:,:),pointer ::  bpudt
   real(dknd), dimension(:,:),pointer ::  bpptb
   real(dknd), dimension(:,:,:),pointer ::  bpuran_trf
   integer, dimension(:,:),pointer :: bpktc
   real(dknd) :: xxx                  != X-coordinate of the particle position.
   real(dknd) :: yyy                  != Y-coordinate of the particle position.
   real(dknd) :: zzz                  != Z-coordinate of the particle position.
   real(dknd) :: uuu                  != Particle direction cosine with X-axis.
   real(dknd) :: vvv                  != Particle direction cosine with Y-axis.
   real(dknd) :: www                  != Particle direction cosine with Z-axis.
   real(dknd) :: erg                  != Particle energy.
   real(dknd) :: wgt                  != Particle weight.
   …
end type banked_particle
```

Figure 5-10: Banked_Particle Complex Type

The banked_particle has all the attributes that MCNP keeps track of for a neutron. Some of the attributes are stored in arrays. These arrays are of unknown length at compile time. First attempts were made to guess at the size of the arrays to make them static in an attempt to avoid the time it requires for the program to dynamically allocate the arrays. However, the memory space required by the banked_particle was big enough that the static arrays caused the group arrays to be bigger and thus they weren’t able to be

read into cache efficiently. Thus causing the data access reads for the neutrons to take more time. The use of dynamic arrays provided a tradeoff that resulted in smaller overall memory use of the bin arrays.

When a neutron is stored the attributes of the neutron are copied into the banked_particle object. The object is then stored into its respective bin to be retrieved when its bin is processed.

The four combinations of groupings that only used energy bins were stored in five or ten arrays depending on how many groups were used. The groupings were done with arrays of the structure shown in Figure 5-11.

Energy Group Arrays

```
!code found in banked_particle_mod.F90 – Appendix B.4
…
type bpErgArrayType
   integer :: bpErgTotal = 0
   type(banked_particle), dimension(:),pointer :: bpArray
 end type bpErgArrayType

 type(bpErgArrayType), allocatable :: bpErgArray(:)

…
subroutine bpInit
   implicit none
   allocate(bpErgArray(5))
   bpErgArray(1)%bpErgTotal = 0
   allocate(bpErgArray(1)%bpArray(10000))
   bpErgArray(2)%bpErgTotal = 0
   allocate(bpErgArray(2)%bpArray(10000))
   bpErgArray(3)%bpErgTotal = 0
   allocate(bpErgArray(3)%bpArray(5000))
   bpErgArray(4)%bpErgTotal = 0
   allocate(bpErgArray(4)%bpArray(5000))
   bpErgArray(5)%bpErgTotal = 0
   allocate(bpErgArray(5)%bpArray(5000))
end subroutine bpInit
…
```

Figure 5-11: Energy Group Arrays

The neutrons were stored in the arrays with code shown in Figure 5-12. For storage the code first determine the energy bin, add 1 to the bin total and then add the neutron to the array bin.

```
if(erg < 1.0) then
    tmpErgGrp = 1
else if(erg >= 1.0 .and. erg < 2.0) then
    tmpErgGrp = 2
else if(erg >= 2.0 .and. erg < 3.0) then
    tmpErgGrp = 3
else if(erg >= 3.0 .and. erg < 4.0) then
    tmpErgGrp = 4
else
    tmpErgGrp = 5
end if

bpErgArray(tmpErgGrp)%bpErgTotal = bpErgArray(tmpErgGrp)%bpErgTotal + 1
bpErgArray(tmpErgGrp)%bpArray(bpErgArray(tmpErgGrp)%bpErgTotal) = curbp
```

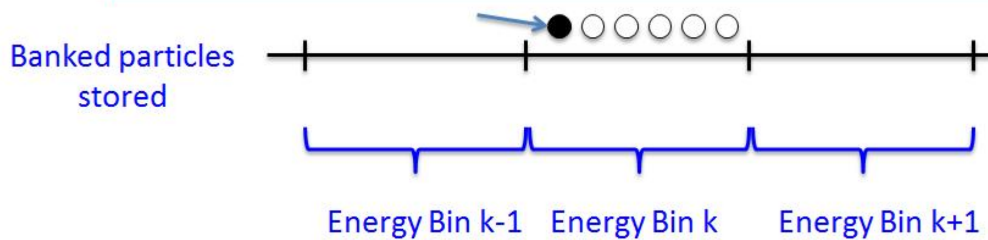


Figure 5-12: Neutron Storage in Groups

To retrieve the neutron, when needed, MCNP checks to see if the current energy group is empty and if so, it moves to the next energy group down and retrieves the neutron. Figure 5-13 demonstrates the retrieval.

```
if(bpErgArray(curErgGrp)%bpErgTotal  <= 0) then
    if(bpErgArray(5)%bpErgTotal > 0) then
        curErgGrp = 5
    else if(bpErgArray(4)%bpErgTotal > 0) then
        curErgGrp = 4
    else if(bpErgArray(3)%bpErgTotal > 0) then
        curErgGrp = 3
    else if(bpErgArray(2)%bpErgTotal > 0) then
        curErgGrp = 2
    else
        curErgGrp = 1
    end if
end if
curbp = bpErgArray(curErgGrp)%bpArray(bpErgArray(curErgGrp)%bpErgTotal)
bpErgArray(curErgGrp)%bpErgTotal = bpErgArray(curErgGrp)%bpErgTotal - 1
```

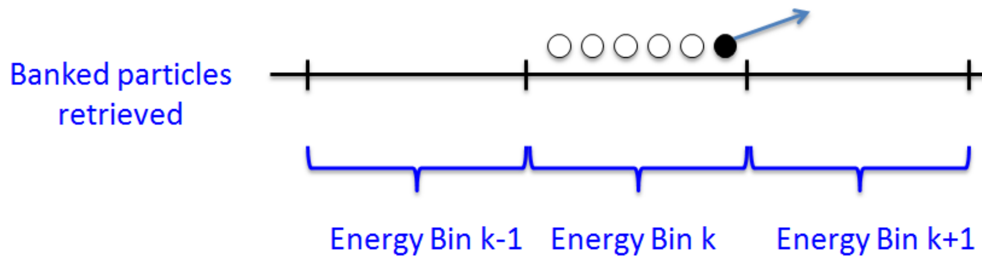


Figure 5-13: Neutron Retrieval from Groups

The code that used Energy Bins (5 Bins) had a resultant run time of 887.44 minutes on the O1 compiler level. The k_{eff} achieved was 0.99575. This was a 4% speed up from the Base code. The same code using Energy Bins (10 Bins) resulted in a run time of 895.30 minutes on the O1 compiler level. The k_{eff} achieved was 0.99577. The speed up was only 3.2% from the Base code. The difference can be attributed to the fact that in the 10 bin code there is more calls to the storing and retrieving of the neutron in the fast energy ranges as the neutron loses energy and doesn't fit the current energy group.

The Energy Bin codes were then run using the O2 compiler level to determine if the compiler could improve on the code by applying its optimizations where it could. The Energy Bin (5 Bins) code had a resultant run time of 774.53 and k_{eff} of 0.99583. This provided a speed up of 16.2% over the Base code. The Energy Bin (10 Bins) code resulted in a run time of 799.28 and k_{eff} of 0.99583, providing a speed up of 13.6 over

Base. The same conclusions on run times difference in O2 are the same as stated previously for O1.

5.7.2 Arrays of Memory Pointers

Brown and Martin, (1987) discussed that more speed up could be achieved with the use of arrays that held pointers to give a location in memory where the neutron needing to be stored or retrieved was. This the allowed manipulation of the pointers in the array groups without having to actually move the neutron data around from group to group.

This use of pointers was applied to the Energy Bin code. The storage code of the neutron changed while the retrieval remained the same. An array was used to store all the neutrons in memory and the pointers in the energy groups held the address of where the neutron was stored in memory in the array. The code in Figure 5-14 shows how the storage was accomplished.

Store neutron in memory

```
srcMasterTot = srcMasterTot+1
bpMasterArray(srcMasterTot) = curbp

tmpTotal = bpErgArray(tmpErgGrp)%bpErgTotal + 1
bpErgArray(tmpErgGrp)%bpErgTotal = tmpTotal
bpErgArray(tmpErgGrp)%bpArray(tmpTotal)%p_bp => bpMasterArray(srcMasterTot)
```

Designate memory location for pointer

Figure 5-14: Neutron Storage with Pointers

The results for the O1 compiler level of the Energy Bins w/ Pointers (5 Bins) showed a run time of 882.15 minutes and k_{eff} of 0.99575, achieving a speed up of 4.6% over the Base code. The Energy Bins w/ Pointers (10 Bins) resulted in a run time of 879.27 minutes and k_{eff} of 0.99574, giving it a speed up of 4.9% over Base. The increase

in speed up over the Energy Bin code without pointers shows that the program was able to spend less time in accessing and moving the neutron pointer when it didn't have to move the actual neutron data within the groups.

The O2 compiler level results for the Energy Bins w/ Pointers, however, didn't achieve faster results. The Energy Bins w/ Pointers (5 Bins) had a run time of 776.19 and k_{eff} of 0.99583. The speed up was 16.1% over Base. The Energy Bins w/ Pointers (10 Bins) had a run time of 769.89 minutes and k_{eff} of 0.99583. The speed up was 16.7% over Base and achieved the fastest run time of all the code sets.

The application of pointers to the code did not have as much an increase in speed as expected. With no need to move the whole neutron data around in the various energy group arrays the results in theory and according to Brown and Martin should have had a much higher impact. One reason the impact was not as high is because the code in this research was not written for a vector machine like the one that Brown and Martin used. Vector machines have specialized hardware that can perform Gather/Scatter functions.

Gather/Scatter occurs when data is not arranged in memory in a sequential, single stride fashion (for example using pointers to access and update operands in memory). On a vector machine special hardware is provided to access (Gather) data for a vector operation and write back (Scatter) data to their corresponding memory locations. On a typical dual core machine such hardware is not available; however, it is possible to use one of the cores to dedicate it to gather operands before the data is needed and scatter operands after the data has been updated and written back to memory. Although to implement that on a dual core machine the MCNP code would have to be rewritten to dedicate one of the cores to do the Gather/Scatter of the operands. See Figure 5-15 that illustrates how the pointers in the array can perform Gather/Scatter functions.

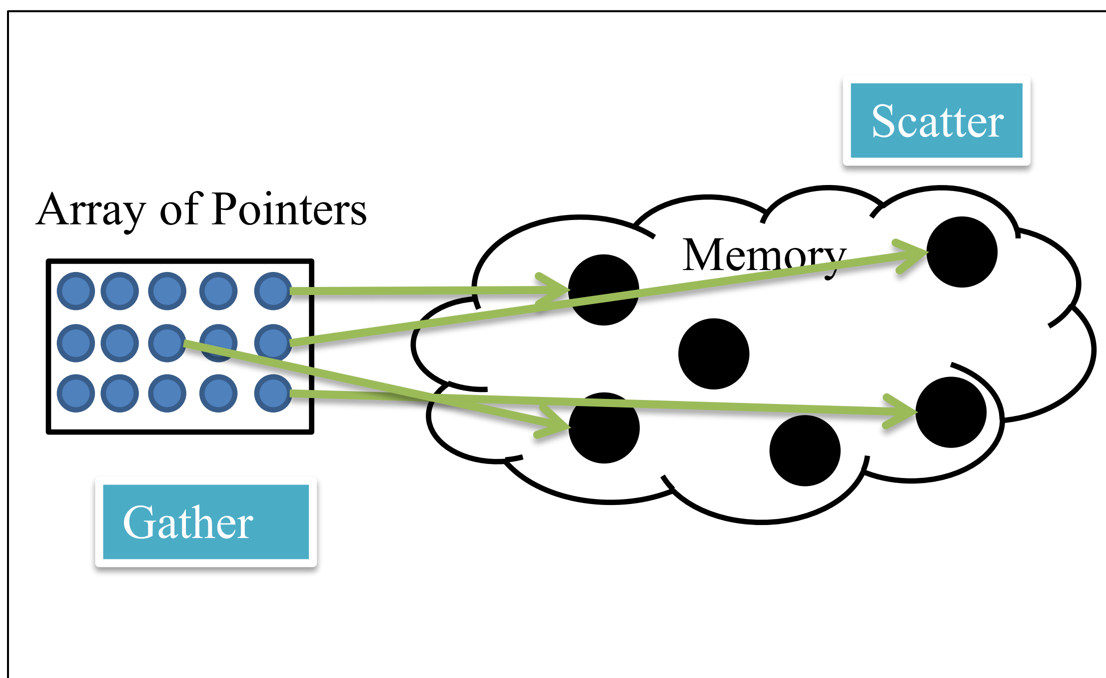


Figure 5-15: Array of Pointers Showing Gather/Scatter

Another reason why this did not happen is due to the fact that the pointers were only implemented in the new code sections that were added. In order to fully implement the pointers the common memory block where MCNP stores the current neutron attributes would have to be modified to include the pointers. This would require a rewrite of the majority of the MCNP code in order to utilize the new pointers in the common block. As it was the purpose of this research to avoid a rewrite of MCNP it was not fully implemented, but was used in a partial state that did provide speed improvement.

5.8 Code Changes and Results for Material/Energy Groups

The second major grouping of tests involved the organizing of the neutrons into the current material they are in and then into energy groups within those material groups. This takes into account that both material and energy affect the cross section. The advantage being that the cross section data needed for the neutrons would be even more

specific and reused. This organization would provide better temporal and spatial locality of the data in memory.

The banked_particle object is the same as used in the other tests to describe the neutrons and their attributes. The energy group ranges were kept the same. The material bins are determined by the surface cards in the MCNP input card. Figure 5-16 illustrates what the design configuration looks like for this research.

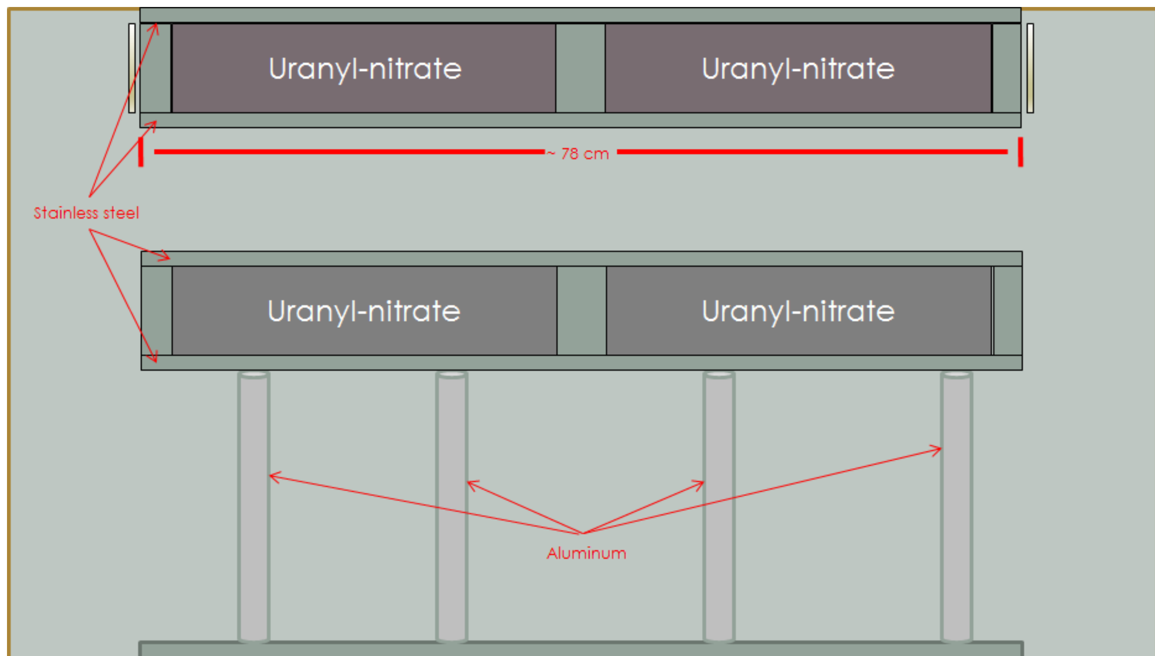


Figure 5-16: Geometry of Criticality Design used in Research

Figure 5-17 shows how it is written in the cell card. The first two columns are the important columns for looking up the material bins. The first column is the cell index; the second is the material index.

```
Material Cells
!found in cell card of input card – Appendix A
…
1   10  -7.8894  2 -1 4 -5          $ Top tank (#1) wall thickness
2   20  -7.883   -4 3 -1            $ Bottom plate of top tank
3   20  -7.883   5 -6 -1            $ Top plate of top tank
4   70  -7.8297  -7 -5 4            $ Top tank support post
5   40  -1.5542   4 -2 -5 7         $ Tank #1 Uranyl-nitrate Solution
6   30  -7.8932   -11 -8 9 12       $ Bottom tank (#2) wall thick
7   20  -7.883   11 -10 -8          $ Top plate of bottom tank
8   20  -7.883   -12 13 -8          $ Bottom plate of bottom tank
9   70  -7.8297  -7 12 -11          $ Bottom tank support post
10  50  -1.5551   -11 -9 12 7       $ Tank #2 Uranyl-nitrate Solution
11  60  -2.69    -13 55 -15 19      $ Bottom Support leg
12  60  -2.69    -13 56 -16 19      $ Bottom Support leg
13  60  -2.69    -13 57 -17 19      $ Bottom Support leg
14  60  -2.69    -13 58 -18 19      $ Bottom Support leg
15  60  -2.69    -19 20 -21 22 -23 24 $ Aluminum Support Plate
16  60  -2.69    -25 26 -35 36 30 -31 $ Top X-axis Plate - A
17  60  -2.69    -25 26 33 -34 30 -31 $ Top X-axis plate - B
18  60  -2.69    -25 26 -27 28 29 -30 $ Top Y-axis plate - C
19  60  -2.69    -25 26 -27 28 31 -32 $ Top Y-axis plate - D
20  60  -2.69    -26 37 -27 38 40 -41 $ Top X-axis plate - E
21  60  -2.69    -26 37 28 -39 40 -41 $ Top X-axis plate - F
22  80  -7.8849   10 -42 -43        $ SS - Bottom
23  0      28 -27 40 -41 20 -6 #1 #2 #3 #4 #5 #6 #7 #8 #9 #10
```

Figure 5-17: Material Cells

The code to handle the lookup of the material group in the program was an array containing the material group number by giving the cell number. This can be seen in Figure 5-18. The lookup was created to combine each cell index found in configuration design into groups of the same material. For example cell index 2 and 3 are the bottom and top plates of the configuration design and are both made of the same stainless steel material and therefore are given the same group, in this case group 2.

```
Material Group Lookup
!code found in banked_particle_mod.F90 – Appendix B.7
…
   allocate(IclGrpLookup(25))
   IclGrpLookup(1) = 1
   IclGrpLookup(2) = 2
   IclGrpLookup(3) = 2
   IclGrpLookup(4) = 7
   IclGrpLookup(5) = 4
   IclGrpLookup(6) = 3
   IclGrpLookup(7) = 2
   IclGrpLookup(8) = 2
   IclGrpLookup(9) = 7
   IclGrpLookup(10) = 5
   IclGrpLookup(11) = 6
   IclGrpLookup(12) = 6
   IclGrpLookup(13) = 6
   IclGrpLookup(14) = 6
   IclGrpLookup(15) = 6
   IclGrpLookup(16) = 6
   IclGrpLookup(17) = 6
   IclGrpLookup(18) = 6
   IclGrpLookup(19) = 6
   IclGrpLookup(20) = 6
   IclGrpLookup(21) = 6
   IclGrpLookup(22) = 8
   IclGrpLookup(23) = 9
   IclGrpLookup(24) = 9
   IclGrpLookup(25) = 9

!Found in bankit.F90 – Appendix B.9
…
tmpIclGrp = IclGrpLookup(icl)
```

Figure 5-18: Material Group Lookup

The groupings were done with arrays of the structure shown in Figure 5-19. The arrays represented the material and each contained 5 or 10 sub arrays that represented the 2 sets of energy groups being used.

Material/Energy Group Arrays

```
!found in banked_particle_mod.F90 – Appendix B.7
…
 type bpErgArrayType
  integer :: bpErgTotal = 0
  type(banked_particle), dimension(:),pointer :: bpArray
 end type bpErgArrayType
 type bpIclArrayType
  integer :: bpIclTotal = 0
  type(bpErgArrayType), dimension(:),pointer :: bpErgArray
 end type bpIclArrayType
 type(bpIclArrayType), allocatable :: bpIclArray(:)
…
  allocate(bpIclArray(iclGrpCnt))
  do i = 1, iclGrpCnt, 1
    allocate(bpIclArray(i)%bpErgArray(ergGrpCnt))
    bpIclArray(i)%bpIclTotal = 0

    bpIclArray(i)%bpErgArray(1)%bpErgTotal = 0
    allocate(bpIclArray(i)%bpErgArray(1)%bpArray(7000))

    bpIclArray(i)%bpErgArray(2)%bpErgTotal = 0
    allocate(bpIclArray(i)%bpErgArray(2)%bpArray(7000))

    bpIclArray(i)%bpErgArray(3)%bpErgTotal = 0
    allocate(bpIclArray(i)%bpErgArray(3)%bpArray(5000))

    bpIclArray(i)%bpErgArray(4)%bpErgTotal = 0
    allocate(bpIclArray(i)%bpErgArray(4)%bpArray(5000))

    bpIclArray(i)%bpErgArray(5)%bpErgTotal = 0
    allocate(bpIclArray(i)%bpErgArray(5)%bpArray(5000))

    if (ergGrpCnt == 10) then
    bpIclArray(i)%bpErgArray(6)%bpErgTotal = 0
    allocate(bpIclArray(i)%bpErgArray(6)%bpArray(5000))

    bpIclArray(i)%bpErgArray(7)%bpErgTotal = 0
    allocate(bpIclArray(i)%bpErgArray(7)%bpArray(5000))

    bpIclArray(i)%bpErgArray(8)%bpErgTotal = 0
    allocate(bpIclArray(i)%bpErgArray(8)%bpArray(5000))

    bpIclArray(i)%bpErgArray(9)%bpErgTotal = 0
    allocate(bpIclArray(i)%bpErgArray(9)%bpArray(5000))

    bpIclArray(i)%bpErgArray(10)%bpErgTotal = 0
    allocate(bpIclArray(i)%bpErgArray(10)%bpArray(5000))
    end if
  enddo
```

Figure 5-19: Material/Energy Group Arrays

The lookup provided the 9 material bins needed to group the neutrons together in the correct material. The storage and retrieval of the neutrons had to be modified to allow for the material bins and energy bins. This can be seen in Figure 5-20.

Material/Energy Group Neutron Storage and Retrieval

```
!code found in banked_particl_mod.F90 – Appendix B.7
…
!storage
   call getErgGrp(tmpErgGrp)
   tmpIclGrp = IclGrpLookup(icl)
   tmpIclTot = bpIclArray(tmpIclGrp)%bpIclTotal + 1
   tmpErgTot = bpIclArray(tmpIclGrp)%bpErgArray(tmpErgGrp)%bpErgTotal + 1
   bpIclArray(tmpIclGrp)%bpIclTotal = tmpIclTot
   bpIclArray(tmpIclGrp)%bpErgArray(tmpErgGrp)%bpErgTotal = tmpErgTot

   bpIclArray(tmpIclGrp)%bpErgArray(tmpErgGrp)%bpArray(tmpErgTot) = curbp
!new pointer code
!     bpIclArray(tmpIclGrp)%bpErgArray(tmpErgGrp)%bpArray(tmpErgTot)%p_bp =>
bpMasterArray(srcMasterTot)
!new pointer code
…
!retrieval
   if(bpIclArray(curIclGrp)%bpErgArray(curErgGrp)%bpErgTotal  <= 0) then
   do i = ergGrpCnt, 1, -1
     if(bpIclArray(curIclGrp)%bpErgArray(i)%bpErgTotal > 0) then
       curErgGrp = i
       goto 120
     else
       do j = iclGrpCnt, 1, -1
         if(bpIclArray(j)%bpIclTotal > 0) then
           curIclGrp = j
           do k = ergGrpCnt, 1, -1
             if(bpIclArray(curIclGrp)%bpErgArray(k)%bpErgTotal > 0) then
               curErgGrp = k
               goto 120
             end if
           enddo
         end if
       enddo
     end if
   enddo
   end if
120 continue
   tmpIclTot = bpIclArray(curIclGrp)%bpIclTotal
   tmpErgTot = bpIclArray(curIclGrp)%bpErgArray(curErgGrp)%bpErgTotal

   curbp = bpIclArray(curIclGrp)%bpErgArray(curErgGrp)%bpArray(tmpErgTot)
!new pointer code
!    curbp = bpIclArray(curIclGrp)%bpErgArray(curErgGrp)%bpArray(tmpErgTot)%p_bp
!new pointer code
   bpIclArray(curIclGrp)%bpErgArray(curErgGrp)%bpErgTotal = tmpErgTot - 1
   bpIclArray(curIclGrp)%bpIclTotal = tmpIclTot – 1
```

Figure 5-20: Material/Energy Group Neutron Storage and Retrieval

The addition of the Material groups takes more code to look up neutrons in the groups. This extra overhead, as will be discussed, causes the Material Bin code sets to not achieve competitive speedups.

The Material/Energy Bins (5 Bins) code used the 9 material groups and 5 energy groups. The material groups were those mentioned above and the energy groups where the same as the ones used in the Energy Bins code. The results from the O1 compiled code was a run time of 945.72 minutes and k_{eff} of 0.99588. This is a 2.2% decrease in speed from Base. It also is a decrease from the Energy Bins (5 Bins) code. The reason that was determined to be the cause of the slowdown was the size of the array structure that made up the groups. Since the array structure was large and 3 dimensional, it wasn't able to fit into the cache entirely. So the program would have to load up sections of the array and see if it contained the neutron to process and if not then it would have to go back to the memory and pull the next section of the array and look again. This shifting in and out the data from the cache caused the slowness.

The Material/Energy Bins (5 Bins) compiled at the O2 level gave a run time of 841.29 minutes and k_{eff} of 0.99593. This gave a speed up of 9% faster than Base code. However, it is still slower than just using energy bins.

The code was then changed to Material/Energy Bins (10 bins) and compiled at the O1 optimization level. The result was a run time of 952.56 minutes and k_{eff} of 0.99593. This was another decrease in speed of 3.0% from Base. At the O2 level the same code gave a run time of 842.42 minutes and k_{eff} of 0.99588. Speed up of 8.9% over Base. These times also are a resultant of the struggle with the large array structure.

In order to cut down on the size of the array structure and moving the data around in the array and thus cut down the memory access time the array of pointers was applied to these sets of code.

First the Material/Energy Bins w/ Pointers (5 Bins) was compiled at the O1 level. The results were a run time of 938.82 minutes and k_{eff} of 0.99593. This was still a decrease of 1.6% from Base but an increase of the same code without pointers. At the O2

level of compiler optimization the resultant run time was 840.08 minutes and k_{eff} of 0.99593. The speed up achieved was 9.2% over Base. Still the array size is an issue.

Next the Material/Energy Bins w/ Pointers (10 Bins) was compiled at the O1 level. The run time was 939.31 minutes and k_{eff} 0.99585. The speed decreased by 1.6% from Base. However, at the O2 level of compiler optimization the run time was 836.28 minutes and k_{eff} of 0.99578. The speed up was 9.6% over Base. The 10 Bins w/ Pointers at O2 achieved the best result for the Material/Energy Bins code set.

5.9 Side by Side Comparison of Code Set Results

The following table provides a quick side by side comparison.

Table 5-1: Run Time Comparisons

Code	Compiler O1 (mins) ± 5 mins	O1 Speedup from Base O1 ± 0.5%	Compiler O2 (mins) ± 5 mins	O2 Speedup from Base O1 ± 0.5%
Base	924.58 (15.4 hours)		830.05 (13.8 hours)	10.2%
Energy Bins (5 Bins)	887.44 (14.8 hours)	4%	774.53 (12.9 hours)	16.2%
Energy Bins (10 Bins)	895.30 (14.9 hours)	3.2%	799.28 (13.3 hours)	13.6%
Energy Bins w/ Pointers (5 Bins)	882.15 (14.7 hours)	4.6%	776.19 (12.9 hours)	16.1%
Energy Bins w/ Pointers (10 Bins)	879.27 (14.65 hours)	4.9%	769.89 (12.8 hours)	16.7%
Material/Energy Bins (5 Bins)	945.72 (15.8 hours)	-2.2%	841.29 (14.0 hours)	9%
Material/Energy Bins (10 Bins)	952.56 (15.9 hours)	-3.0%	842.42 (14.0 hours)	8.9%
Material/Energy Bins w/ Pointers (5 Bins)	938.82 (15.6 hours)	-1.6%	840.08 (14.0 hours)	9.2%
Material/Energy Bins w/ Pointers (10 Bins)	939.31 (15.7 hours)	-1.6%	836.28 (13.9 hours)	9.6%

We compared the accuracy of the calculated k_{eff}. Table 5-2 compares k_{eff} calculated by the various code modifications and the Base code. As can be seen in this table the O2 compiler introduced a higher error when compared to the Base code. However if the O2 results are compared with the Base O2 results there isn't much error introduced by the new code sets.

The small differences in k_{eff} can be explained by the fact that in changing the code to group and order the neutrons, we have changed the behavior of the MCNP code and its use of random numbers. MCNP uses a sequence of random numbers throughout the history of the cycle and neutron history. Therefore when the new code sets change the order of neutrons, the neutrons use different random numbers they would have used in the original order. This may not appear to be a problem but the neutron history is based on several factors and one of them is the random number. Thus a neutron using a different random number can easily take a different path in history and result in a slightly different statistical behavior.

As the k_{eff} can be affected by many factors any slight changes to the code will cause a variation. In fact the k_{eff} can be different from computer to computer. Part of the install process of the MCNP program has you run test programs to verify that your installed program results in a k_{eff} value that is statistically close to the expected accepted value. In the case of the different code sets researched here, all are close to the accepted value and thus provides a measure of confidence that the new code sets have achieved an acceptable k_{eff} value.

Table 5-2: k_{eff} Comparisons

Code	Compiler O1 k_{eff} $\sigma = 0.00004$	Reactivity ρ / Reactor Period τ	Compiler O2 k_{eff} $\sigma = 0.00004$	Reactivity ρ / Reactor Period τ
Base	0.99571	$-0.00431	0.99582	$-0.00420
		-31.35s		-31.85s
Energy Bins (5 Bins)	0.99575	$-0.00427	0.99583	$-0.00419
		-31.53s		-31.89s
Energy Bins (10 Bins)	0.99577	$-0.00425	0.99583	$-0.00419
		-31.61s		-31.89s
Energy Bins w/ Pointers (5 Bins)	0.99575	$-0.00427	0.99583	$-0.00419
		-31.53s		-31.89s
Energy Bins w/ Pointers (10 Bins)	0.99574	$-0.00428	0.99583	$-0.00419
		-31.48s		-31.89s
Material/Energy Bins (5 Bins)	0.99588	$-0.00414	0.99593	$-0.00409
		-32.13s		-32.37s
Material/Energy Bins (10 Bins)	0.99583	$-0.00419	0.99588	$-0.00414
		-31.89s		-32.13s
Material/Energy Bins w/ Pointers (5 Bins)	0.99588	$-0.00414	0.99593	$-0.00409
		-32.13s		-32.37s
Material/Energy Bins w/ Pointers (10 Bins)	0.99585	$-0.00417	0.99578	$-0.00424
		-31.98s		-31.66s

Comparing the reactor period τ between the code sets, the differences in the outcome of the various code sets are within about ½ second of the Base code set and show that the new code modifications deliver acceptable results.

5.10 Complex Configuration Designs

The results as discussed are for the design slab tank experiment. As shown, the material code sets actually produced a slight slowdown of speed. Therefore, the groupings to choose would be the energy groups for this design. This design when compared to a full reactor design is considered a simple design. As material is still a factor in cross section data, the concern arises that if a more complex design is tested with thousands of materials will the energy groupings achieve the same speed up. Also as the reactor fuel is depleted during different stages of burnup the material composition in the reactor changes and the different materials present in the reactor again increases.

Regardless of configuration and material composition energy of the neutron still affects the cross section data needed in the calculation. Therefore, the groupings by energy will still have an effect on the application run time for the more complex designs. The effective speed up may not be as significant as that found in this less complex design but a speed up is expected. The act of creating the groups and sorting them provides order from the pseudo-random neutron sample provided using the Monte Carlo method. This order improves the spatial and temporal locality of the cross section data and will increase the speed up.

Chapter 6: Conclusion

The objective of this thesis was to demonstrate that a more optimal sequential MCNP performance can be obtained with minimal programming effort by organizing the neutrons into groups. The unmodified MCNP process took 15 ½ hours to perform the criticality calculations for the two slab tank experiment. The improved memory locality, by using 10 groups of energy bins with pointer arrays, was able to achieve a 16.7% ± 0.5% speedup and results in 2 ½ hours being removed from the run time taken to perform the criticality calculations.

Research discovered a slowdown may occur if one does not take care in structuring the group sizes. Some groups may be unable to fit in memory and have to push out the cross section data from cache. This is inherent in the use of the Material/Energy Group code sets discussed in that the group array structure was a large three dimensional array (material x energy x neutron). This array structure was not able to fit in the cache during stored neutron data retrieval and required extra time to read from the slower memory sections when data was swapped out of cache to provide room.

It was shown that the time savings of the MCNP code can be achieved without completely rewriting the code. As stated in the thesis research, previous investigators have rewritten the code to take advantage of new computer hardware, like GPUs, in order to achieve speed ups. This however, according to the investigators, takes significant effort and the end results are often a completely new program.

Creating groups based on the neutron energy increases the efficiency of the reuse of data located in the cache. The cross section data in the cache used by a neutron in a group is already loaded and can be used by the next neutron in the same group due to the similar energy levels of the neutron. The reuse of data is an efficient use of the temporal locality principle.

Several sets of code modifications were tested to determine the makeup of the neutron groups to coincide with the energy ranges found in the ENDF/B VI cross section

data. This allowed the cross section data in the cache to be reused allowing the program to take advantage of the temporal and spatial locality of the data to increase its efficiency and not spend more time than necessary outside the cache retrieving information for its calculations.

The modifications to the MCNP code consisted of rearranging the main functional loop of tracking the history of the neutron, from source generation through its lifecycle, in the program to break it out into two separate loops: generation of source neutrons and tracking the history of each neutron. All the neutrons were generated from the source and stored into groups based on their energy levels. Then each neutron history was tracked as in the normal MCNP flow until it was absorbed, leaked out of the system, or moved into another energy range encompassed by another energy group.

As stated, this research showed that a speedup is achieved for the two slab tank design by grouping neutrons by their energy levels to optimize the reuse of the cross section data in the cache. It gives results that are promising and with thorough verification and validation the additions can be used for further study in all system designs. For example, further study can be done to determine the speedup results that can be achieved using more complex designs like a full reactor model. Organization of neutron into groups offers potential MCNP improvements and valuable time savings for future engineers.

References

Brown, F., Liu, T., Ding, A., Jib, W., Xu, X., and Carothers, C., 2012a. A Monte Carlo Neutron Transport Code For Eigenvalue Calculations on a Dual-GPU System and CUDA Environment. *PHYSOR 2012 – Advances in Reactor Physics.*

Brown, F., Ding, A., Liu, T., Liang, C., Ji, Shepard, M., W., and Xu, X., 2012b. Evaluation of Speedup of Monte Carlo Calculations of Two Simple Reactor Physics Problems Coded For the GPU/CUDA Environment (2012) *PHYSOR 2012 – Advances in Reactor Physics.*

Brown, F., and Martin, W., 1984. Monte Carlo Methods for Radiation Transport Analysis on Vector Computers.

Brown, F., and Martin, W., 1987. Status of Vectorized Monte Carlo for Particle Transport Analysis, International Journal of High Performance Computing Applications 2(1): 11-32

Carstens, N., 2004. "Speedup of MCNP(X) Parallel KCODE Execution Via Communication Algorithm Development and Beowulf Cluster Optimization" Massachusetts Institute of Technology Master Of Science in Nuclear Engineering Candidate thesis.

Glasstone, S., and Sesonske, A., 1969. Nuclear Reactor Engineering 3rd Edition, Van Nostrand Reinhold Company. ISBN 0-442-02725-7

Gong, C., Liu, J., Yang, B., Deng, L., Li, G., Li, X., Hu, Q., and Gong, Z., 2011. Accelerating MCNP-based Monte Carlo Simulations for Neutron Transport on GPU, *National University of Defense Technology 410073, China, Institute of Applied Physics and Computational Mathematics, 100088, China.*

Hadjidoukas, P., Bousis, C., and Emfietzoglou, D., 2010. Parallelization of a Monte Carlo particle transport simulation code, Computer Physics Communications, *www.elsevier.com/locate/cpc.*

Hennessy J., and Patterson, D., 2007. Computer Architecture – A Quantitative Approach 4th Edition, Morgan Kaufmann Publishers. ISBN 0-123-70490-1

Levesque, J., and Wagenbreth, G., 2011. High Performance Computing: Programming and Applications, CRC Press. ISBN 1-420-07705-8

Martin, W., 2007. "Advances in Monte Carlo Methods for Global Reactor Analysis," Invited lecture at the *M&C 2007 International Conference*, Monterey, CA, USA, April 15-19.

McKinney, G., and West, T., 1993. Multiprocessing MCNP on an IBM RS600 Cluster LANL LA-UR-93-463.

Nelson, A., 2009. Monte Carlo Neutron Transport on Graphics Processing Units Using CUDA, Master's Thesis Pennsylvania State University.

Rinard, P., 1991. Neutron interactions with matter. *Passive Nondestructive Assay of Nuclear Material, Los Alamos Technical Report NUREG/CR-5550, LA-UR-90*732: 357-377.

Siegel, A., Smith, K., Felker, K., Romano, P., Forget, B., and Beckman, P., 2013. Improved Cache Performance in Monte Carlo Transport Calculations Using Energy Banding, Computer Physics Communications, http://dx.doi.org/10.1016/j.cpc.2013.10.008

Smith, K., 2003. "Reactor Core Methods," Invited lecture at the *M&C 2003 International Conference*, Gatlinburg, TN, USA, April 6-10.

Wadleigh, K., and Crawford, I., 2000. Software Optimization for High Performance Computing: Creating Faster Applications, Prentice Hall. ISBN 0-130-17008-9

Weinberg, J., 2005. Quantifying Locality in the Memory Access Patterns of HPC Applications University of California Master of Science in Computer Science Candidate thesis.

X-5 Monte Carlo Team, (2003) MCNP – A General Monte Carlo N-Particle Transport Code, Version 5, Volume I: Overview and Theory.

Appendix A

MCNP Input Card

```
Slab Tank Experiment
c Cell Cards
1   10  -7.8894  2 -1 4 -5          $ Top tank (#1) wall thickness
2   20  -7.883   -4 3 -1            $ Bottom plate of top tank
3   20  -7.883   5 -6 -1            $ Top plate of top tank
4   70  -7.8297  -7 -5 4            $ Top tank support post
5   40  -1.5542   4 -2 -5 7         $ Tank #1 Uranyl-nitrate Solution
6   30  -7.8932   -11 -8 9 12       $ Bottom tank (#2) wall thick
7   20  -7.883   11 -10 -8          $ Top plate of bottom tank
8   20  -7.883   -12 13 -8          $ Bottom plate of bottom tank
9   70  -7.8297  -7 12 -11          $ Bottom tank support post
10  50  -1.5551   -11 -9 12 7       $ Tank #2 Uranyl-nitrate Solution
11  60  -2.69    -13 55 -15 19      $ Bottom Support leg
12  60  -2.69    -13 56 -16 19      $ Bottom Support leg
13  60  -2.69    -13 57 -17 19      $ Bottom Support leg
14  60  -2.69    -13 58 -18 19      $ Bottom Support leg
15  60  -2.69    -19 20 -21 22 -23 24 $ Aluminum Support Plate
16  60  -2.69    -25 26 -35 36 30 -31 $ Top X-axis Plate - A
17  60  -2.69    -25 26 33 -34 30 -31 $ Top X-axis plate - B
18  60  -2.69    -25 26 -27 28 29 -30 $ Top Y-axis plate - C
19  60  -2.69    -25 26 -27 28 31 -32 $ Top Y-axis plate - D
20  60  -2.69    -26 37 -27 38 40 -41 $ Top X-axis plate - E
21  60  -2.69    -26 37 28 -39 40 -41 $ Top X-axis plate - F
22  80  -7.8849   10 -42 -43        $ SS - Bottom
23  0     28 -27 40 -41 20 -6 #1 #2 #3 #4 #5 #6 #7 #8 #9 #10
        #11 #12 #13 #14 #15 #16 #17 #18 #19 #20 #21 #22
24  0          -28:27:-40:41:-20:6  $ External Void

c Surface Cards
1   cz 37.90696
2   cz 35.81654
3   pz 4.63296
4   pz 5.60095
5   pz 14.50162
6   pz 15.46301
7   cz 1.26873
8   cz 37.95903
9   cz 35.78733
10  pz -5.27050
```

```
11  pz -6.24434
12  pz -15.15593
13  pz -16.11249
15  c/z 26.4922 26.4922 2.54
16  c/z 26.4922 -26.4922 2.54
17  c/z -26.4922  26.4922 2.54
18  c/z -26.4922 -26.4922 2.54
19  pz -51.67249
20  pz -52.94249
21  px 39.37
22  px -39.37
23  py 39.37
24  py -39.37
25  pz 12.92301
26  pz 5.30301
27  px 58.42
28  px -58.42
29  py -39.2176
30  py -38.5826
31  py  38.5826
32  py  39.2176
33  px  38.5826
34  px  39.2176
35  px -38.5826
36  px -39.2176
37  pz  2.76301
38  px 48.26
39  px -48.26
40  py -58.42
41  py 58.42
42  pz -4.63296
43  cz 38.09619
55  c/z 26.4922 26.4922 1.905
56  c/z 26.4922 -26.4922 1.905
57  c/z -26.4922  26.4922 1.905
58  c/z -26.4922 -26.4922 1.905

mode n
imp:n 1 22r 0
```

```
c    Hoop#1
m10   6000.70c  1.9778-4
     25055.70c  1.6518-3   15031.70c  4.7551-5
     16032.70c  7.4074-6
     14028.70c  1.0453-3   14029.70c  5.2930-5   14030.70c  3.5135-5
     24050.70c  7.2653-4   24052.70c  1.4010-2   24053.70c  1.5887-3
     24054.70c  3.9545-4
     28058.70c  4.5245-3   28060.70c  1.7435-3   28061.70c  7.5767-5
     28062.70c  2.4152-4   28064.70c  6.1544-5
     42092.70c  1.9108-5   42094.70c  1.1910-5   42095.70c  2.0499-5
     42096.70c  2.1477-5   42098.70c  1.2297-5   42098.70c  3.1070-5
     42100.70c  1.2400-5
     29063.70c  1.6027-4   29065.70c  7.1501-5
      7014.70c  2.0352-4
     26054.70c  3.4635-3   26056.70c  5.4772-2   26057.70c  1.3138-3
     26058.70c  1.6720-4
c    Tank Plates
m20   6000.70c  1.9762-4
     25055.70c  1.6504-3   15031.70c  4.7512-5
     16032.70c  7.4013-6
     14028.70c  1.0445-3   14029.70c  5.2888-5   14030.70c  3.5108-5
     24050.70c  7.2596-4   24052.70c  1.3999-2   24053.70c  1.5874-3
     24054.70c  3.9514-4
     28058.70c  4.5209-3   28060.70c  1.7421-3   28061.70c  7.5705-5
     28062.70c  2.4133-4   28064.70c  6.1494-5
     42092.70c  1.9092-5   42094.70c  1.1900-5   42095.70c  2.0481-5
     42096.70c  2.1459-5   42098.70c  1.2286-5   42098.70c  3.1043-5
     42100.70c  1.2389-5
     29063.70c  1.6014-4   29065.70c  7.1446-5
      7014.70c  2.0336-4
     26054.70c  3.4607-3   26056.70c  5.4727-2   26057.70c  1.3127-3
     26058.70c  1.6707-4
c    Hoop#2
m30   6000.70c  1.9788-4
     25055.70c  1.6526-3   15031.70c  4.7574-5
     16032.70c  7.4109-6
     14028.70c  1.0458-3   14029.70c  5.2953-5   14030.70c  3.5151-5
     24050.70c  7.2688-4   24052.70c  1.4017-2   24053.70c  1.5894-3
     24054.70c  3.9564-4
```

```
    28058.70c  4.5267-3   28060.70c  1.7443-3   28061.70c  7.5803-5
    28062.70c  2.4164-4   28064.70c  6.1573-5
    42092.70c  1.9117-5   42094.70c  1.1916-5   42095.70c  2.0508-5
    42096.70c  2.1487-5   42098.70c  1.2302-5   42098.70c  3.1084-5
    42100.70c  1.2405-5
    29063.70c  1.6035-4   29065.70c  7.1535-5
     7014.70c  2.0362-4
    26054.70c  3.4652-3   26056.70c  5.4798-2   26057.70c  1.3144-3
    26058.70c  1.6729-4
c    Tank#1 Solution
m40  92234.70c  8.7477-6   92235.70c  9.6338-4   92236.70c  2.8747-6
    92238.70c  5.9027-5    8016.70c  3.7675-2    7014.70c  2.2608-3
     1001.70c  5.7842-2
mt40 lwtr.01t
c    Tank#2 Solution
m50  92234.70c  8.7673-6   92235.70c  9.6404-4   92236.70c  2.8787-6
    92238.70c  5.9069-5    8016.70c  3.7695-2    7014.70c  2.2622-3
     1001.70c  5.7871-2
mt50   lwtr.01t
c    Al Support Plates
m60  14028.70c  3.1918-4   14029.70c  1.6161-5   14030.70c  1.0728-5
    26054.70c  5.8882-6   26056.70c  9.3114-5   26057.70c  2.2334-6
    26058.70c  2.8426-7
    29063.70c  4.4070-5   29065.70c  1.9661-5
    25055.70c  2.2115-5
    12024.70c  5.2648-4   12025.70c  6.6651-5   12026.70c  7.3383-5
    24050.70c  2.7074-6   24052.70c  5.2209-5   24053.70c  5.9201-6
    24054.70c  1.4736-6
    29063.70c  2.1414-5  29065.70c  9.5533-6
    22046.70c  2.0300-6  22047.70c  1.8524-6   22048.70c  1.8727-5
    22049.70c  1.3956-6  22050.70c  1.3703-6
    13027.70c  5.8433-2
c    Center Support Bar
m70   6000.70c  1.9628-4
    25055.70c  1.6393-3   15031.70c  4.7191-5
    16032.70c  7.3513-6
    14028.70c  1.0374-3   14029.70c  5.2528-5   14030.70c  3.4869-5
    24050.70c  7.2105-4   24052.70c  1.3905-2   24053.70c  1.5767-3
    24054.70c  3.9247-4
```

```
     28058.70c  4.4903-3   28060.70c  1.7303-3   28061.70c  7.5193-5
     28062.70c  2.3970-4   28064.70c  6.1078-5
     42092.70c  1.8963-5   42094.70c  1.1820-5   42095.70c  2.0343-5
     42096.70c  2.1314-5   42098.70c  1.2203-5   42098.70c  3.0833-5
     42100.70c  1.2305-5
     29063.70c  1.5906-4   29065.70c  7.0961-5
      7014.70c  2.0198-4
     26054.70c  3.4373-3   26056.70c  5.4357-2   26057.70c  1.3038-3
     26058.70c  1.6594-4
c    SS
m80   6000.70c  1.9767-4
     25055.70c  1.6508-3   15031.70c  4.7524-5
     16032.70c  7.4031-6
     14028.70c  1.0448-3   14029.70c  5.2902-5   14030.70c  3.5117-5
     24050.70c  7.2614-4   24052.70c  1.4003-2   24053.70c  1.5878-3
     24054.70c  3.9524-4
     28058.70c  4.5219-3   28060.70c  1.7425-3   28061.70c  7.5723-5
     28062.70c  2.4138-4   28064.70c  6.1509-5
     42092.70c  1.9096-5   42094.70c  1.1903-5   42095.70c  2.0486-5
     42096.70c  2.1464-5   42098.70c  1.2289-5   42098.70c  3.1050-5
     42100.70c  1.2392-5
     29063.70c  1.6018-4   29065.70c  7.1461-5
      7014.70c  2.0340-4
     26054.70c  3.4616-3   26056.70c  5.4740-2   26057.70c  1.3130-3
     26058.70c  1.6711-4
c
kcode  10000  1.0  100  50100
kopts  blocksize=10  kinetics=yes  precursor=yes
ksrc 15. 0. 9. -15. 0. 9. 15. 0. -9. -15. 0. -9
prdmp j 10000 -1
```

Appendix B

Code

B.1 Introduction

Included here is the code that was written to make the optimizations. Two new files were added to the MCNP code base called banked_particle_mod.F90 and newSource.F90. Three files were modified to include references to the new files; they are trnspt.F90, hstory.F90 and bankit.F90. Code, except where it has been modified and is my orginal code, has been removed from the three files in accordance with RSICC Licensing and intellectual propery rights. Also to stay in accordance to the license some modifications cannot be included as they would publicize the original code.

B.2 trnspt.F90

```
subroutine trnspt
 /* Code removed to protect Copyright information */
! Optimization Modification Begins
    srcbpmax = nsa
    srcbpproc = 0
    srcbptot = 0
    binCnt = 5
  srcbpcur = nsa
20 continue

     do i = 1,srcbpcur
      ! Run the next history.
       /* Code removed to protect Copyright information */

      call newSource

    enddo

    if(bpErgArray(curErgGrp)%bpErgTotal  <= 0) then
        if(bpErgArray(5)%bpErgTotal > 0) then
            curErgGrp = 5
        else if(bpErgArray(4)%bpErgTotal > 0) then
            curErgGrp = 4
        else if(bpErgArray(3)%bpErgTotal > 0) then
            curErgGrp = 3
        else if(bpErgArray(2)%bpErgTotal > 0) then
            curErgGrp = 2
        else
```

```
        curErgGrp = 1
      end if
    end if

    do while( srcbptot > 0 )

      call bankit(101)
      call hstory

    enddo

    if(.not. time_to_stop()) go to 20
/* Code removed to protect Copyright information */
end subroutine trnspt
```

B.3 newSource.F90

```
subroutine newSource
  ! Description:
  ! Get New particles from source

  ! Modules:
/* Code removed to protect Copyright information */

  ! Start a particle from the source.
20 continue

/* Code removed to protect Copyright information */
  ! Set particle random number
  call RN_init_particle( int(npstc,i8knd) )

  call startp
  !$ call sm_loff(jlock,1)

    npa=1
    call bankit(1)

  return
end subroutine newSource
```

B.4 Energy Group banked_particle_mod.F90

```
module banked_particle_mod
    use mcnp_global
    use mcnp_debug
```

```
  implicit none
type banked_particle
  integer ::  abprtc = 0
  integer ::  abpudt = 0
  integer ::  abpptb = 0
  integer ::  abpuran_trf = 0
  integer ::  abpktc = 0
  real(dknd), dimension(:,:),pointer ::  bprtc
  real(dknd), dimension(:,:),pointer ::  bpudt
  real(dknd), dimension(:,:),pointer ::  bpptb
  real(dknd), dimension(:,:,:),pointer ::  bpuran_trf
  integer, dimension(:,:),pointer :: bpktc
  real(dknd) :: xxx                 != X-coordinate of the particle position.
  real(dknd) :: yyy                 != Y-coordinate of the particle position.
  real(dknd) :: zzz                 != Z-coordinate of the particle position.
  real(dknd) :: uuu                 != Particle direction cosine with X-axis.
  real(dknd) :: vvv                 != Particle direction cosine with Y-axis.
  real(dknd) :: www                 != Particle direction cosine with Z-axis.
  real(dknd) :: erg                 != Particle energy.
  real(dknd) :: wgt                 != Particle weight.
  real(dknd) :: tme                 != Time at the particle position.
  real(dknd) :: vel                 != Speed of the particle.
  real(dknd) :: dls                 != Distance to next boundary.
  real(dknd) :: dxl                 != Distance to nearest DXTRAN sphere.
  real(dknd) :: dtc                 != Distance to time cutoff.
  real(dknd) :: elc(mipt)           != Energy cutoffs in the current cell.
  real(dknd) :: fiml(mipt)          != Importance of the current cell.
  real(dknd) :: fismg               != Multigroup importance.
  real(dknd) :: wtfasv              != Accumulated weight of adjoint particle.
  real(dknd) :: rnk                 != RNR at point where new track was created.
  real(dknd) :: spare(mspare)       != Spare banked array for user modifications.
  real(dknd) :: totmp               != Total cross section for previous track.
  real(dknd) :: ralfp(2)            != Eigenvalue by 2nd order perturb method.
  real(dknd) :: zpblcm

  integer :: npa              != Number of tracks in the same bank location.
  integer :: icl              != Program number of the current cell.
  integer :: jsu              != Program number of the current surface.
  integer :: ipt              != Type of particle.
  integer :: iex              != Index of the current cross section table.
  integer :: node             != Number of nodes in track from source to here.
  integer :: idx              != Number of the current DXTRAN sphere.
  integer :: ncp              != Count of collisions per track.
  integer :: jgp              != Neutron: particle energy group number.
                              != Electron/photon: generation class for F6:p tally.
```

```
    integer :: lev                 != Level of the current particle.
    integer :: iii                 != First lattice index of particle location.
    integer :: jjj                 != Second lattice index of particle location.
    integer :: kkk                 != Third lattice index of particle location.
    integer :: iap                 != Program number of the next cell.
    integer :: iexp                != IEX from previous collision.
    integer :: mtp                 != Reaction MT from previous collision.
    integer :: nmco                != Stores value of NMC as it is updated.
    integer :: i_positron          != Flag for positron (when ipt==electron).
    integer :: node_above          != Current PHTVR node of the particle
    integer :: branch              != PHTVR tree branch of the particle
    integer :: scoring_particle    != Flag to indicate that the particle
                                   != does not contribute to regular tallies.
    integer :: progenitor_id       != progenitor id for adjoint weighting in kcode.
    integer :: delayed_group       != delayed neutron group
    integer :: mpblcm
  end type banked_particle
  type bpErgArrayType
    integer :: bpErgTotal = 0
    type(banked_particle), dimension(:),pointer :: bpArray
  end type bpErgArrayType

  type(bpErgArrayType), allocatable :: bpErgArray(:)

  integer :: curErgGrp = 0
  integer :: binCnt = 5
  integer :: srcbptot
  integer :: srcbpmax
  integer :: srcbpcur
  integer :: srcbpproc

contains
subroutine bpInit
    implicit none
    allocate(bpErgArray(5))
    bpErgArray(1)%bpErgTotal = 0
    allocate(bpErgArray(1)%bpArray(10000))
    bpErgArray(2)%bpErgTotal = 0
    allocate(bpErgArray(2)%bpArray(10000))
    bpErgArray(3)%bpErgTotal = 0
    allocate(bpErgArray(3)%bpArray(5000))
    bpErgArray(4)%bpErgTotal = 0
    allocate(bpErgArray(4)%bpArray(5000))
    bpErgArray(5)%bpErgTotal = 0
    allocate(bpErgArray(5)%bpArray(5000))
```

```
end subroutine bpInit

subroutine getErgGrp(tmpErgGrp)
   implicit none

   integer, intent(out) :: tmpErgGrp

   if (ergGrpCnt == 10) then
      if(erg < 0.0000001) then !thermal
         tmpErgGrp = 1
      else if(erg >= 0.0000001 .and. erg < 0.000001) then !epithermal
         tmpErgGrp = 2
      else if(erg >= 0.000001 .and. erg < 0.1) then !resonance
         tmpErgGrp = 3
      else if(erg >= 0.1 .and. erg < 1.0) then !fast
         tmpErgGrp = 4
      else if(erg >= 1.0 .and. erg < 1.5) then
         tmpErgGrp = 5
      else if(erg >= 1.5 .and. erg < 2.0) then
         tmpErgGrp = 6
      else if(erg >= 2.0 .and. erg < 2.5) then
         tmpErgGrp = 7
      else if(erg >= 2.5 .and. erg < 3.0) then
         tmpErgGrp = 8
      else if(erg >= 3.0 .and. erg < 3.5) then
         tmpErgGrp = 9
      else
         tmpErgGrp = 10
      end if
   else
      if(erg < 1.0) then
         tmpErgGrp = 1
      else if(erg >= 1.0 .and. erg < 2.0) then
         tmpErgGrp = 2
      else if(erg >= 2.0 .and. erg < 3.0) then
         tmpErgGrp = 3
      else if(erg >= 3.0 .and. erg < 4.0) then
         tmpErgGrp = 4
      else
         tmpErgGrp = 5
      end if
   end if
end subroutine getErgGrp
end module banked_particle_mod
```

B.5 Energy Group hstory.F90

```
/* Code removed to protect Copyright information */
/*Return from code that describes collision effects*/
! Optimization Modification Begins
call getErgGrp(tmpErgGrp)
  if(tmpErgGrp == curErgGrp) then
    go to 30
  else
    npa = 1
    call bankit(1)
    go to 290
  endif
! Optimization Modification Ends
  !
  ! ******************  Process terminated particles.  ********************
  !

/* Code removed to protect Copyright information */
end subroutine hstory
```

B.6 Energy Group bankit.F90

```
  subroutine bank_particle
       ! Description:
/* Code removed to protect Copyright information */
    integer :: tmpErgGrp = 5
    type (banked_particle) :: curbp
/* Code removed to protect Copyright information */
/* Assign curbp attributes from variables recorded in MCNP */
    srcbptot = srcbptot+1
    nbnk = srcbptot
!new pointer code
!    srcMasterTot = srcMasterTot+1
!    bpMasterArray(srcMasterTot) = curbp
!new pointer code
    call getErgGrp(tmpErgGrp)
    tmpTotal = bpErgArray(tmpErgGrp)%bpErgTotal + 1
    bpErgArray(tmpErgGrp)%bpErgTotal = tmpTotal
    bpErgArray(tmpErgGrp)%bpArray(tmpTotal) = curbp
!new pointer code
!    bpErgArray(tmpErgGrp)%bpArray(tmpTotal)%p_bp =>
bpMasterArray(srcMasterTot)
!new pointer code
```

```
    return
end subroutine bank_particle
  ! --------------------------------------------------------------------
  subroutine unbank_particle
    ! Description
    type (banked_particle) :: curbp
    if(bpErgArray(curErgGrp)%bpErgTotal  <= 0) then
      do i = ergGrpCnt, 1, -1
        if(bpErgArray(i)%bpErgTotal > 0) then
          curErgGrp = i
          goto 120
        end if
      end do
    end if
120 continue

  tmpTotal = bpErgArray(curErgGrp)%bpErgTotal
  curbp = bpErgArray(curErgGrp)%bpArray(tmpTotal)
!new pointer code
!    curbp = bpErgArray(curErgGrp)%bpArray(tmpTotal)%p_bp
!new pointer code
  bpErgArray(curErgGrp)%bpErgTotal = tmpTotal – 1

130 continue
    ! Retrieve the particle.
    /* Code removed to protect Copyright information */
   /* Assign MCNP Global variables from curbp attributes */

    srcbptot = srcbptot-1
    nbnk = srcbptot
    if(curbp%abpptb == 1) then
        deallocate(curbp%bpptb)
    end if

    if(curbp%abpudt == 1) then
        deallocate(curbp%bpudt)
    end if

    if(curbp%abprtc == 1) then
        deallocate(curbp%bprtc)
    end if

    if(curbp%abpktc == 1) then
        deallocate(curbp%bpktc)
    end if
```

```
    if(curbp%abpuran_trf == 1) then
       deallocate(curbp%bpuran_trf)
    end if
    curbp%abprtc = 0
    curbp%abpudt = 0
    curbp%abpptb = 0
    curbp%abpuran_trf = 0
    curbp%abpktc = 0

    return
  end subroutine unbank_particle
```

B.7 Material/Energy Group banked_particle_mod.F90

```
module banked_particle_mod

    use mcnp_global
    use mcnp_debug
    implicit none

  type banked_particle
    integer ::  abprtc = 0
    integer ::  abpudt = 0
    integer ::  abpptb = 0
    integer ::  abpuran_trf = 0
    integer ::  abpktc = 0
    real(dknd), dimension(:,:),pointer ::  bprtc
    real(dknd), dimension(:,:),pointer ::  bpudt
    real(dknd), dimension(:,:),pointer ::  bpptb
    real(dknd), dimension(:,:,:),pointer ::  bpuran_trf
    integer, dimension(:,:),pointer :: bpktc
    real(dknd) :: xxx                    != X-coordinate of the particle position.
    real(dknd) :: yyy                    != Y-coordinate of the particle position.
    real(dknd) :: zzz                    != Z-coordinate of the particle position.
    real(dknd) :: uuu                    != Particle direction cosine with X-axis.
    real(dknd) :: vvv                    != Particle direction cosine with Y-axis.
    real(dknd) :: www                    != Particle direction cosine with Z-axis.
    real(dknd) :: erg                    != Particle energy.
    real(dknd) :: wgt                    != Particle weight.
    real(dknd) :: tme                    != Time at the particle position.
    real(dknd) :: vel                    != Speed of the particle.
    real(dknd) :: dls                    != Distance to next boundary.
    real(dknd) :: dxl                    != Distance to nearest DXTRAN sphere.
```

```
  real(dknd) :: dtc                 != Distance to time cutoff.
  real(dknd) :: elc(mipt)           != Energy cutoffs in the current cell.
  real(dknd) :: fiml(mipt)          != Importance of the current cell.
  real(dknd) :: fismg               != Multigroup importance.
  real(dknd) :: wtfasv              != Accumulated weight of adjoint particle.
  real(dknd) :: rnk                 != RNR at point where new track was created.
  real(dknd) :: spare(mspare)       != Spare banked array for user modifications.
  real(dknd) :: totmp               != Total cross section for previous track.
  real(dknd) :: ralfp(2)            != Eigenvalue by 2nd order perturb method.
  real(dknd) :: zpblcm

  integer :: npa                    != Number of tracks in the same bank location.
  integer :: icl                    != Program number of the current cell.
  integer :: jsu                    != Program number of the current surface.
  integer :: ipt                    != Type of particle.
  integer :: iex                    != Index of the current cross section table.
  integer :: node                   != Number of nodes in track from source to here.
  integer :: idx                    != Number of the current DXTRAN sphere.
  integer :: ncp                    != Count of collisions per track.
  integer :: jgp                    != Neutron: particle energy group number.
                                    != Electron/photon: generation class for F6:p tally.
  integer :: lev                    != Level of the current particle.
  integer :: iii                    != First lattice index of particle location.
  integer :: jjj                    != Second lattice index of particle location.
  integer :: kkk                    != Third lattice index of particle location.
  integer :: iap                    != Program number of the next cell.
  integer :: iexp                   != IEX from previous collision.
  integer :: mtp                    != Reaction MT from previous collision.
  integer :: nmco                   != Stores value of NMC as it is updated.
  integer :: i_positron             != Flag for positron (when ipt==electron).
  integer :: node_above             != Current PHTVR node of the particle
  integer :: branch                 != PHTVR tree branch of the particle
  integer :: scoring_particle       != Flag to indicate that the particle
                                    != does not contribute to regular tallies.
  integer :: progenitor_id          != progenitor id for adjoint weighting in kcode.
  integer :: delayed_group          != delayed neutron group
  integer :: mpblcm
end type banked_particle

type bpErgArrayType
  integer :: bpErgTotal = 0
  type(banked_particle), dimension(:),pointer :: bpArray
end type bpErgArrayType

!new pointer
```

```
!  type bpArrayType
!    type(banked_particle),pointer :: p_bp
!  end type bpArrayType
!
!  type bpErgArrayType
!    integer :: bpErgTotal = 0
!    type(bpArrayType), dimension(:),pointer :: bpArray
!  end type bpErgArrayType
!new pointer

  type bpIclArrayType
    integer :: bpIclTotal = 0
    type(bpErgArrayType), dimension(:),pointer :: bpErgArray
  end type bpIclArrayType

  type(bpIclArrayType), allocatable :: bpIclArray(:)
  !type(bpErgArrayType), allocatable :: bpErgArray(:)

!new pointer
!  type(banked_particle), allocatable, target :: bpMasterArray(:)
!new pointer

  integer, allocatable :: IclGrpLookup(:)

  integer :: curErgGrp = 10
  integer :: ergGrpCnt = 10
  integer :: iclGrpCnt = 9
  integer :: curIclGrp = 9

!  type(banked_particle),allocatable :: arraySrcbp(:)
  integer :: srcbptot
  integer :: srcbpmax
  integer :: srcbpcur
  integer :: srcbpproc
  integer :: particleMaxSet = 0
  integer :: srcMasterTot
contains

subroutine bpInit
    implicit none
    integer :: i,j = 0

    allocate(IclGrpLookup(25))
    IclGrpLookup(1) = 1
    IclGrpLookup(2) = 2
```

```
IclGrpLookup(3) = 2
IclGrpLookup(4) = 7
IclGrpLookup(5) = 4
IclGrpLookup(6) = 3
IclGrpLookup(7) = 2
IclGrpLookup(8) = 2
IclGrpLookup(9) = 7
IclGrpLookup(10) = 5
IclGrpLookup(11) = 6
IclGrpLookup(12) = 6
IclGrpLookup(13) = 6
IclGrpLookup(14) = 6
IclGrpLookup(15) = 6
IclGrpLookup(16) = 6
IclGrpLookup(17) = 6
IclGrpLookup(18) = 6
IclGrpLookup(19) = 6
IclGrpLookup(20) = 6
IclGrpLookup(21) = 6
IclGrpLookup(22) = 8
IclGrpLookup(23) = 9
IclGrpLookup(24) = 9
IclGrpLookup(25) = 9

allocate(bpIclArray(iclGrpCnt))

do i = 1, iclGrpCnt, 1
   allocate(bpIclArray(i)%bpErgArray(ergGrpCnt))
   bpIclArray(i)%bpIclTotal = 0

   bpIclArray(i)%bpErgArray(1)%bpErgTotal = 0
   allocate(bpIclArray(i)%bpErgArray(1)%bpArray(7000))

   bpIclArray(i)%bpErgArray(2)%bpErgTotal = 0
   allocate(bpIclArray(i)%bpErgArray(2)%bpArray(7000))

   bpIclArray(i)%bpErgArray(3)%bpErgTotal = 0
   allocate(bpIclArray(i)%bpErgArray(3)%bpArray(5000))

   bpIclArray(i)%bpErgArray(4)%bpErgTotal = 0
   allocate(bpIclArray(i)%bpErgArray(4)%bpArray(5000))

   bpIclArray(i)%bpErgArray(5)%bpErgTotal = 0
   allocate(bpIclArray(i)%bpErgArray(5)%bpArray(5000))
```

```
      if (ergGrpCnt == 10) then
      bpIclArray(i)%bpErgArray(6)%bpErgTotal = 0
      allocate(bpIclArray(i)%bpErgArray(6)%bpArray(5000))

      bpIclArray(i)%bpErgArray(7)%bpErgTotal = 0
      allocate(bpIclArray(i)%bpErgArray(7)%bpArray(5000))

      bpIclArray(i)%bpErgArray(8)%bpErgTotal = 0
      allocate(bpIclArray(i)%bpErgArray(8)%bpArray(5000))

      bpIclArray(i)%bpErgArray(9)%bpErgTotal = 0
      allocate(bpIclArray(i)%bpErgArray(9)%bpArray(5000))

      bpIclArray(i)%bpErgArray(10)%bpErgTotal = 0
      allocate(bpIclArray(i)%bpErgArray(10)%bpArray(5000))
      end if
   enddo
!   allocate(bpMasterArray(100000))
   return
end subroutine bpInit

subroutine getErgGrp(tmpErgGrp)
   implicit none

   integer, intent(out) :: tmpErgGrp

   if (ergGrpCnt == 10) then
      if(erg < 0.0000001) then !thermal
         tmpErgGrp = 1
      else if(erg >= 0.0000001 .and. erg < 0.000001) then !epithermal
         tmpErgGrp = 2
      else if(erg >= 0.000001 .and. erg < 0.1) then !resonance
         tmpErgGrp = 3
      else if(erg >= 0.1 .and. erg < 1.0) then
         tmpErgGrp = 4
      else if(erg >= 1.0 .and. erg < 1.5) then
         tmpErgGrp = 5
      else if(erg >= 1.5 .and. erg < 2.0) then
         tmpErgGrp = 6
      else if(erg >= 2.0 .and. erg < 2.5) then
         tmpErgGrp = 7
      else if(erg >= 2.5 .and. erg < 3.0) then
         tmpErgGrp = 8
      else if(erg >= 3.0 .and. erg < 3.5) then
         tmpErgGrp = 9
```

```
        else
          tmpErgGrp = 10
        end if
    else
        if(erg < 0.0000001) then !thermal
          tmpErgGrp = 1
        else if(erg >= 0.0000001 .and. erg < 0.000001) then !epithermal
          tmpErgGrp = 2
        else if(erg >= 0.000001 .and. erg < 0.1) then !resonance
          tmpErgGrp = 3
        else if(erg >= 0.1 .and. erg < 3.0) then!fast
          tmpErgGrp = 4
        else
          tmpErgGrp = 5
        end if
    end if
end subroutine getErgGrp
```

B.8 Material/Energy Group hstory.F90

```
/* Code removed to protect Copyright information */
/*Code that handles the path of the neutron and determines if it goes to a new cell*/
!if new cell lookup group to store neutron in
    if(IclGrpLookup(Icl) == curIclGrp ) then
      go to 50
    else
      npa = 1
      call bankit(1)
      go to 290
    endif

/*Return from code that describes collision effects*/
! Optimization Modification Begins
call getErgGrp(tmpErgGrp)

  if(tmpErgGrp == curErgGrp) then
    go to 30
  else
    npa = 1
    call bankit(1)
    go to 290
  endif
! Optimization Modification Ends
  !
  ! *****************  Process terminated particles.  ******************
```

```
  !

/* Code removed to protect Copyright information */
end subroutine hstory
```

B.9 Material/Energy Group bankit.F90

```
  subroutine bank_particle
       ! Description:
/* Code removed to protect Copyright information */
    integer :: tmpErgGrp = 5
    type (banked_particle) :: curbp
/* Code removed to protect Copyright information */
/* Assign curbp attributes from variables recorded in MCNP */
    srcbptot = srcbptot+1
    nbnk = srcbptot
!new pointer code
!    srcMasterTot = srcMasterTot+1
!    bpMasterArray(srcMasterTot) = curbp
!new pointer code
    call getErgGrp(tmpErgGrp)

    tmpIclGrp = IclGrpLookup(icl)
    tmpIclTot = bpIclArray(tmpIclGrp)%bpIclTotal + 1
    tmpErgTot = bpIclArray(tmpIclGrp)%bpErgArray(tmpErgGrp)%bpErgTotal + 1
    bpIclArray(tmpIclGrp)%bpIclTotal = tmpIclTot
    bpIclArray(tmpIclGrp)%bpErgArray(tmpErgGrp)%bpErgTotal = tmpErgTot

    bpIclArray(tmpIclGrp)%bpErgArray(tmpErgGrp)%bpArray(tmpErgTot) = curbp
!new pointer code
!     bpIclArray(tmpIclGrp)%bpErgArray(tmpErgGrp)%bpArray(tmpErgTot)%p_bp =>
bpMasterArray(srcMasterTot)
!new pointer code

    return
end subroutine bank_particle
  ! ----------------------------------------------------------------------
  subroutine unbank_particle
    ! Description
    type (banked_particle) :: curbp
    if(bpIclArray(curIclGrp)%bpErgArray(curErgGrp)%bpErgTotal  <= 0) then
    do i = ergGrpCnt, 1, -1
      if(bpIclArray(curIclGrp)%bpErgArray(i)%bpErgTotal > 0) then
        curErgGrp = i
        goto 120
```

```
     else
       do j = iclGrpCnt, 1, -1
         if(bpIclArray(j)%bpIclTotal > 0) then
           curIclGrp = j
           do k = ergGrpCnt, 1, -1
             if(bpIclArray(curIclGrp)%bpErgArray(k)%bpErgTotal > 0) then
               curErgGrp = k
               goto 120
             end if
           enddo
         end if
       enddo
     end if
   enddo
   end if
120 continue
   tmpIclTot = bpIclArray(curIclGrp)%bpIclTotal
   tmpErgTot = bpIclArray(curIclGrp)%bpErgArray(curErgGrp)%bpErgTotal

   curbp = bpIclArray(curIclGrp)%bpErgArray(curErgGrp)%bpArray(tmpErgTot)
!new pointer code
!    curbp =
bpIclArray(curIclGrp)%bpErgArray(curErgGrp)%bpArray(tmpErgTot)%p_bp
!new pointer code
   bpIclArray(curIclGrp)%bpErgArray(curErgGrp)%bpErgTotal = tmpErgTot - 1
   bpIclArray(curIclGrp)%bpIclTotal = tmpIclTot – 1

130 continue
   ! Retrieve the particle.
   /* Code removed to protect Copyright information */
  /* Assign MCNP Global variables from curbp attributes */

   srcbptot = srcbptot-1
   nbnk = srcbptot
   if(curbp%abpptb == 1) then
      deallocate(curbp%bpptb)
   end if

   if(curbp%abpudt == 1) then
      deallocate(curbp%bpudt)
   end if

   if(curbp%abprtc == 1) then
      deallocate(curbp%bprtc)
   end if
```

```
  if(curbp%abpktc == 1) then
    deallocate(curbp%bpktc)
  end if

  if(curbp%abpuran_trf == 1) then
    deallocate(curbp%bpuran_trf)
  end if
  curbp%abprtc = 0
  curbp%abpudt = 0
  curbp%abpptb = 0
  curbp%abpuran_trf = 0
  curbp%abpktc = 0

  return
end subroutine unbank_particle
```

Appendix C

Thesis Defense Presentation

IMPROVED MCNP MEMORY LOCALITY BY NEUTRON GROUPING

AARON BLY
BLY[illegible]@VANDALS.UIDAHO.EDU / AARONDBLY@GMAIL.COM
MASTER OF SCIENCE IN NUCLEAR ENGINEERING
2014

OUTLINE

- Objective
- Neutron Interaction with Matter
- MCNP
- Optimization
- New MCNP Process
- Results

2

OBJECTIVE

- Implement optimizations to the Monte Carlo N-Particle (MCNP) program to achieve a speedup in the program run time.
- Apply the modifications chosen that require minimal changes to MCNP, thus avoiding a rewrite of the program.

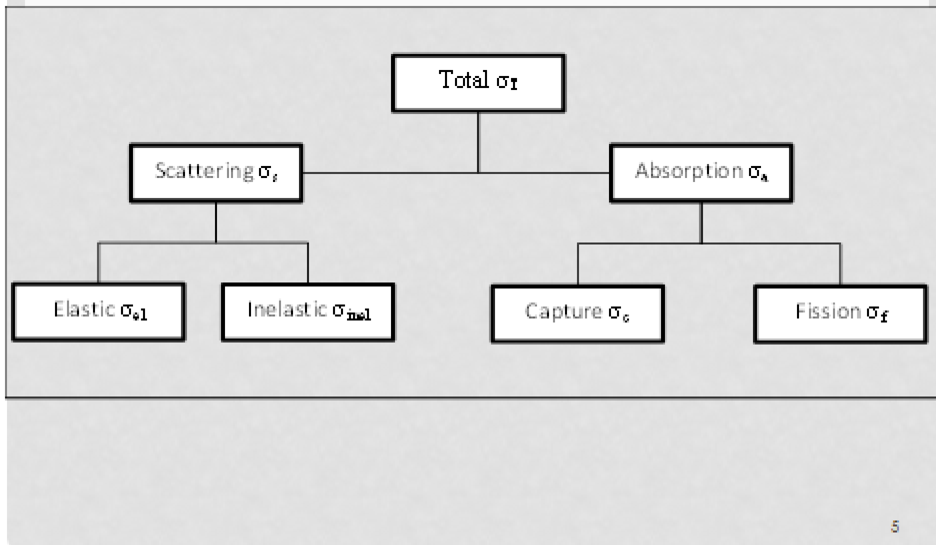
3

CROSS SECTION

- Microscopic cross section is the probability that a nuclear reaction will occur and convention has adopted σ as its representation.
- Cross sections are dependent on various parameters:
  - Energy of the free neutron
  - Material in which the neutron is traveling
  - Energy of the material nuclides
  - The relative angle between neutron and target material nuclide.

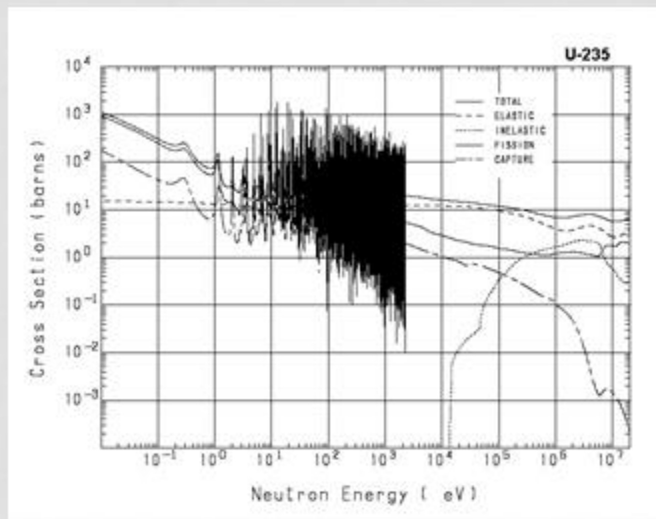
4

TYPES OF CROSS SECTIONS



5

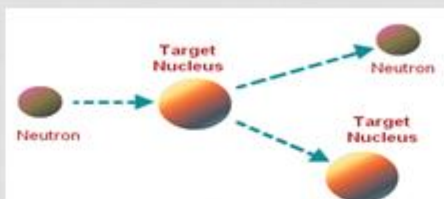
CROSS SECTION VS NEUTRON ENERGY



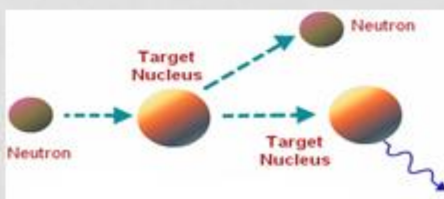
6

SCATTERING - ELASTIC AND INELASTIC

Elastic



Inelastic

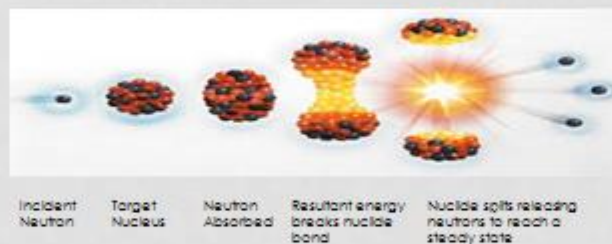


7

ABSORPTION - CAPTURE AND FISSION

Capture

Fission



8

EFFECTIVE NEUTRON MULTIPLICATION FACTOR, K_{EFF}

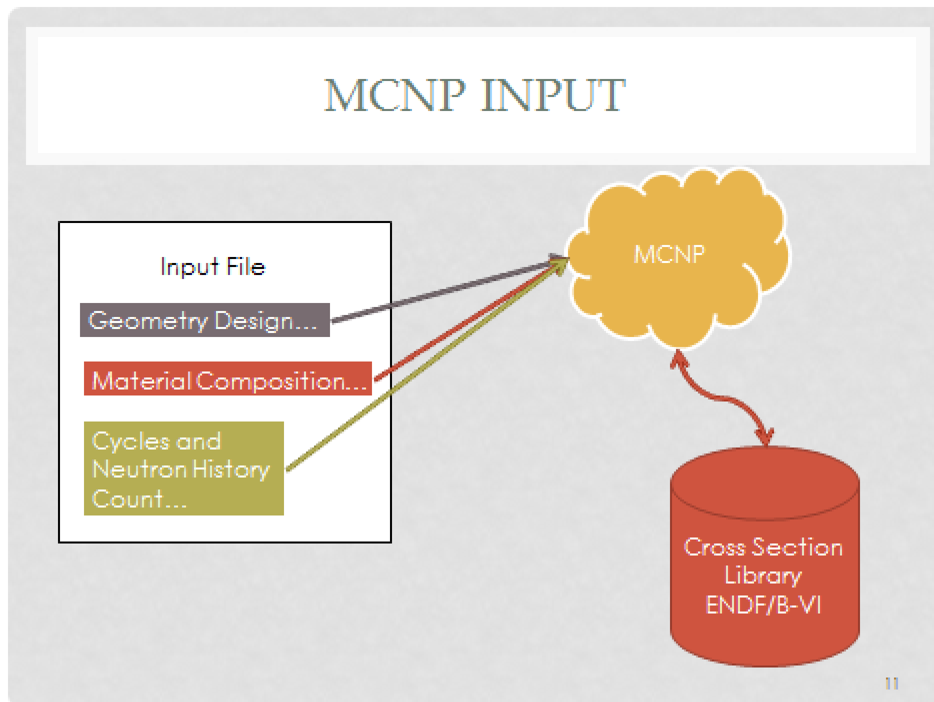
- The effective neutron multiplication factor, k_{eff}, is the rate of neutron production divided by the rate of neutron absorption plus rate of neutron leakage from the system.
- Geometry of the configuration will affect the k_{eff}
  - For example, If the configuration is small then the neutrons could leak out before fission can occur.
  - Measured as B_g, called Geometric Buckling.
- The value of k determines how a nuclear chain reaction proceeds:
  - $k_{eff} < 1$ (subcritical)
  - $k_{eff} = 1$ (critical)
  - $k_{eff} > 1$ (supercritical)

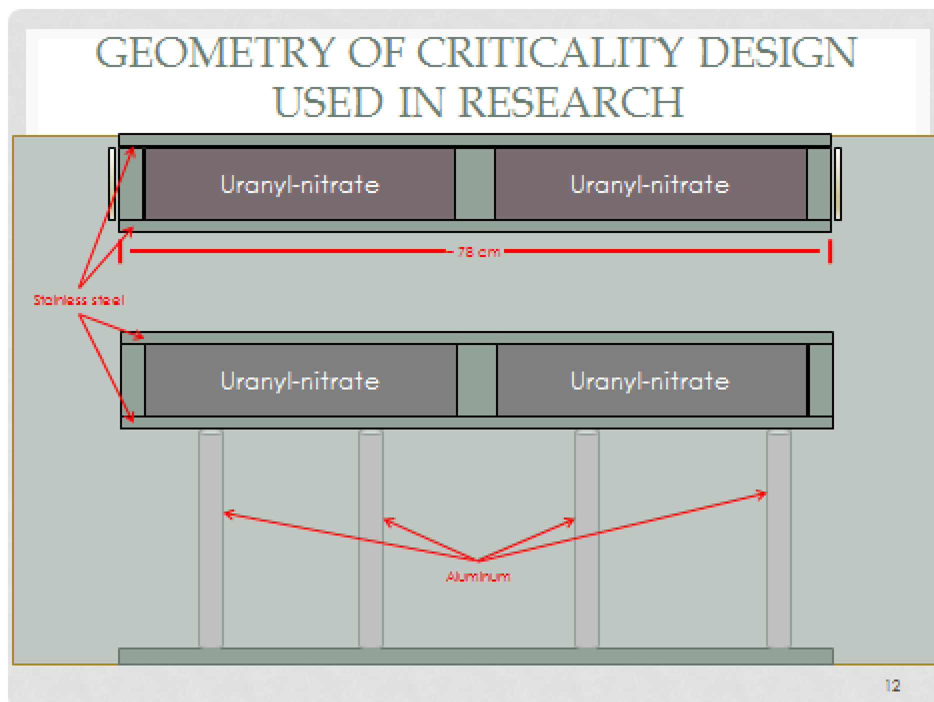
9

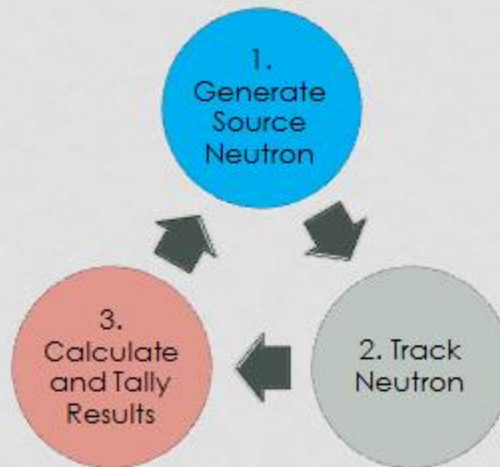
MONTE CARLO N-PARTICLE (MCNP)

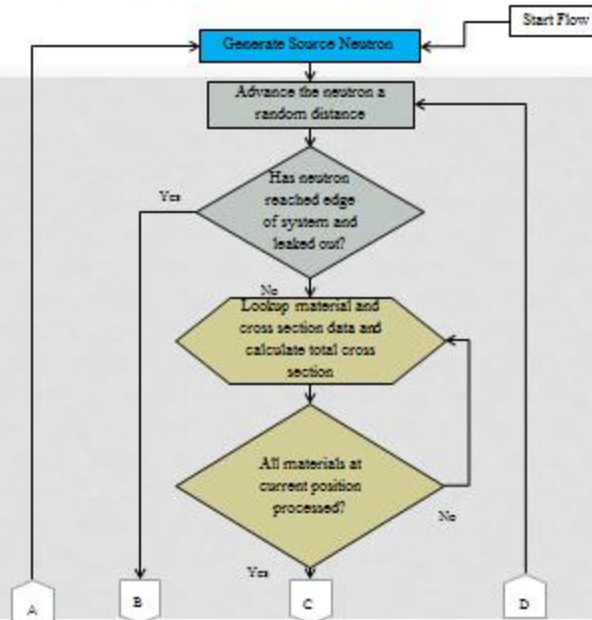
- Software used for modeling and calculating neutron, photon, electron, or coupled neutron/photon/electron transport.
- Employs the Monte Carlo method.
  - The Monte Carlo method is a numerical technique that uses psuedo-random numbers and probability to solve problems.
  - Psuedo- random numbers are an attempt by a computer to produce truly random sequences of numbers.
- Is used for many calculations, in particular k_{eff} calculations to determine criticality for a given design.

10

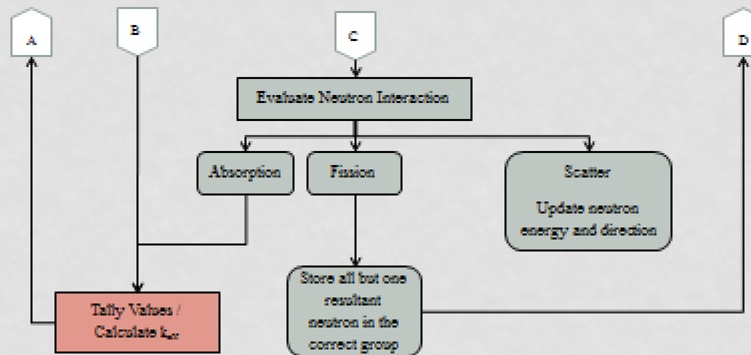

MCNP INPUT
Input File
Geometry Design...
Material Composition...
Cycles and
Neutron History
Count...
MCNP
Cross Section
Library
ENDF/B-VI
11


GEOMETRY OF CRITICALITY DESIGN
USED IN RESEARCH
Uranyl-nitrate
Uranyl-nitrate
78 cm
Stainless steel
Uranyl-nitrate
Uranyl-nitrate
Aluminum
12


MCNP PROCESS FLOW - OVERVIEW
1. Generate Source Neutron
2. Track Neutron
3. Calculate and Tally Results
13


MCNP FLOW DIAGRAM
Start Flow
Generate Source Neutron
Advance the neutron a random distance
Has neutron reached edge of system and leaked out?
Yes
No
Lookup material and cross section data and calculate total cross section
All materials at current position processed?
No
Yes
A
B
C
D
14

MCNP FLOW DIAGRAM (CONTINUED)



15

ISSUES CAUSED BY PSEUDO-RANDOMNESS

- Due to the pseudo-randomness of the neutrons presented by the Monte Carlo Method, MCNP has to spend time reading in the cross section data for each neutron.

16

VECTORIZATION

- Brown and Martin (1984)
- Reorganize MCNP into a 'vectorized' execution model where 'vectors' (arrays) of similar particle event interactions are grouped for future processing.
  - Why is this advantageous?
    - Increased performance by organizing MCNP to introduce Spatial and Temporal Locality of data and instructions.
  - What are the drawbacks?
    - Have to rewrite the code to make use of the vector hardware.
    - Rewrite MCNP which has approximately 334 files with about 86415 lines of code

17

OPTIMIZATION

- Group and order neutrons.
  - Re-reading the cross section table into the local cache memory is less efficient than accessing the cross section table once and using it many times.
  - Take advantage of Temporal and Spatial locality of data.
- Function Inlining
  - Reduces overhead

18

OPTIMIZATION

- Change Compiler Optimization Level
  - Each level has automatic optimizations the compiler can try to apply.

- Optimization Level 1
  - fmerge-constants
  - fdce
  - More…

- Optimization Level 2
  - Same optimizations of level 1
  - finline-small-functions
  - falign-loops
  - More…

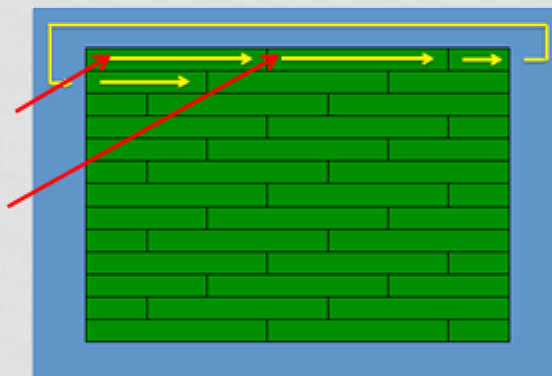
19

SPATIAL LOCALITY

- **Spatial Locality**: The concept that the likelihood of referencing a resource is higher if a resource near it was just referenced.

Stride 1 is accessed first

Then stride 2.



20

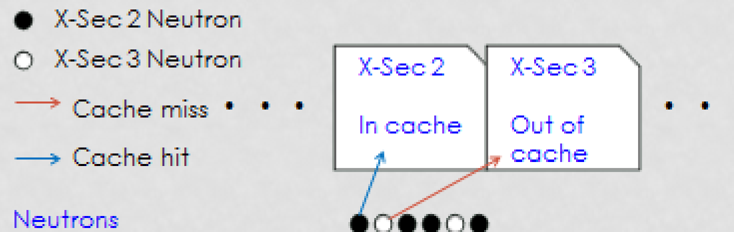
TEMPORAL LOCALITY

- **Temporal Locality**: The concept that a resource that is referenced at one point in time will be referenced again sometime in the near future.

Original Loop	Optimized Loop
Do i= 1,N	Load y into register 1
X(i) = X(i) * Y	Do i = 1,N
End Do	X(i) = X(i) * register 1
	End Do

21

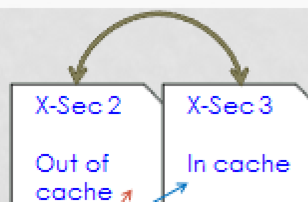
CROSS SECTION DATA CACHE HIT/MISS



22

CROSS SECTION DATA CACHE HIT/MISS

- ● X-Sec 2 Neutron
- ○ X-Sec 3 Neutron
- → Cache miss
- → Cache hit



X-Sec 3 is pulled into Cache
X-Sec 2 is removed

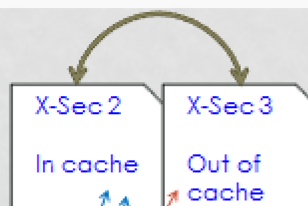
Neutrons

- Time is spent swapping X-Sec data in and out of the cache

23

CROSS SECTION DATA CACHE HIT/MISS

- ● X-Sec 2 Neutron
- ○ X-Sec 3 Neutron
- → Cache miss
- → Cache hit



X-Sec 2 is pulled into Cache
X-Sec 3 is removed

Neutrons

- Time is spent swapping X-Sec data in and out of the cache

24

NEUTRON GROUPS

- Neutrons are grouped by attributes that affect cross sections.
- Groups chosen:
  - Material
  - Energy

25

ENERGY GROUPS

- Energy Groups used: 5 and 10.

```
subroutine bpInit
    implicit none

    allocate(bpErgArray(5))

    bpErgArray(1)%bpErgTotal = 0
    allocate(bpErgArray(1)%bpArray(10000))

    bpErgArray(2)%bpErgTotal = 0
    allocate(bpErgArray(2)%bpArray(10000))

    bpErgArray(3)%bpErgTotal = 0
    allocate(bpErgArray(3)%bpArray(5000))

    bpErgArray(4)%bpErgTotal = 0
    allocate(bpErgArray(4)%bpArray(5000))

    bpErgArray(5)%bpErgTotal = 0
    allocate(bpErgArray(5)%bpArray(5000))
end subroutine bpInit
```

Arrays used to track groups

```
subroutine bpInit
    implicit none

    allocate(bpErgArray(ergGrpCnt))

    bpErgArray(1)%bpErgTotal = 0
    allocate(bpErgArray(1)%bpArray(20000))

    bpErgArray(2)%bpErgTotal = 0
    allocate(bpErgArray(2)%bpArray(20000))

    bpErgArray(3)%bpErgTotal = 0
    allocate(bpErgArray(3)%bpArray(20000))

    bpErgArray(4)%bpErgTotal = 0
    allocate(bpErgArray(4)%bpArray(20000))

    bpErgArray(5)%bpErgTotal = 0
    allocate(bpErgArray(5)%bpArray(20000))

    bpErgArray(6)%bpErgTotal = 0
    allocate(bpErgArray(6)%bpArray(5000))

    bpErgArray(7)%bpErgTotal = 0
    allocate(bpErgArray(7)%bpArray(5000))

    bpErgArray(8)%bpErgTotal = 0
    allocate(bpErgArray(8)%bpArray(5000))

    bpErgArray(9)%bpErgTotal = 0
    allocate(bpErgArray(9)%bpArray(5000))

    bpErgArray(10)%bpErgTotal = 0
    allocate(bpErgArray(10)%bpArray(5000))

end subroutine bpInit
```

26

NEUTRON GROUP CHOICE

- Groups are created to implement the "Supergroups" that Brown and Martin (1987) discuss in vectorizing Monte Carlo.
- Groups were decided in an attempt to match how the Cross Section data is organized in MCNP to lessen the cache misses.
- Martin and Brown were able to implement their supergroups with the knowledge of exactly how the data was organized.

ENERGY RANGES FOR THE GROUPS

Code sets with 5 groups:

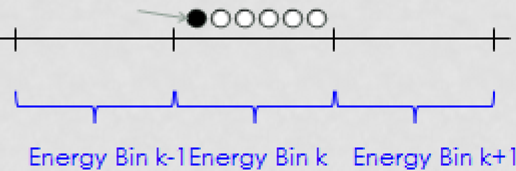
- < 1 MeV
- 1 MeV to 2 MeV
- 2 MeV to 3 MeV
- 3 MeV to 4 MeV
- >= 4 MeV

Code sets with 10 groups:

- < 1 eV (Thermal)
- 1 eV to 10 eV (Epithermal)
- 10 eV to 100 keV (Resonance)
- 100 keV to 1 MeV (Fast)
- 1 MeV to 1.5 MeV
- 1.5 MeV to 2 MeV
- 2 MeV to 2.5 MeV
- 2.5 MeV to 3 MeV
- 3 MeV to 3.5 MeV
- >= 3.5 MeV

CODE FOR STORING NEUTRONS INTO THE ENERGY STORAGE ARRAYS

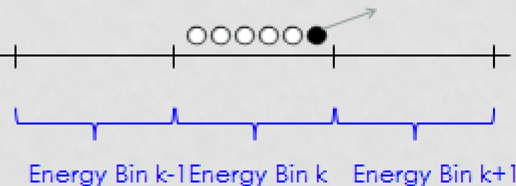
```
if(erg < 1.0) then
    tmpErgGrp = 1
else if(erg >= 1.0 .and. erg < 2.0) then
    tmpErgGrp = 2
else if(erg >= 2.0 .and. erg < 3.0) then
    tmpErgGrp = 3
else if(erg >= 3.0 .and. erg < 4.0) then
    tmpErgGrp = 4
else
    tmpErgGrp = 5
end if

bpErgArray(tmpErgGrp)%bpErgTotal = bpErgArray(tmpErgGrp)%bpErgTotal + 1
bpErgArray(tmpErgGrp)%bpArray(bpErgArray(tmpErgGrp)%bpErgTotal) = curbp
```



29

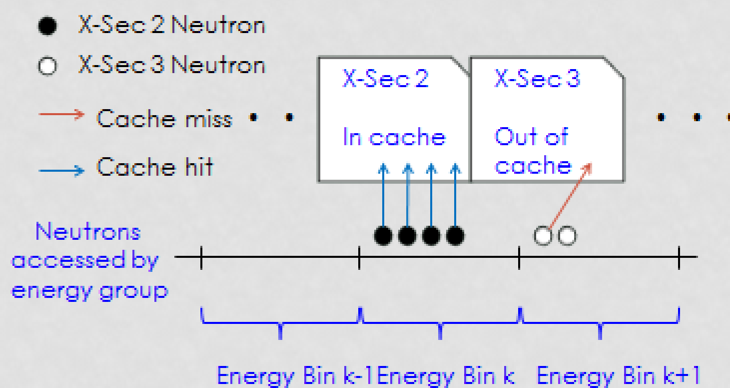
CODE FOR RETRIEVING NEUTRONS FROM THE ENERGY STORAGE ARRAYS

```
if(bpErgArray(curErgGrp)%bpErgTotal  <= 0) then
    if(bpErgArray(5)%bpErgTotal > 0) then
        curErgGrp = 5
    else if(bpErgArray(4)%bpErgTotal > 0) then
        curErgGrp = 4
    else if(bpErgArray(3)%bpErgTotal > 0) then
        curErgGrp = 3
    else if(bpErgArray(2)%bpErgTotal > 0) then
        curErgGrp = 2
    else
        curErgGrp = 1
    end if
end if
curbp = bpErgArray(curErgGrp)%bpArray(bpErgArray(curErgGrp)%bpErgTotal)
bpErgArray(curErgGrp)%bpErgTotal = bpErgArray(curErgGrp)%bpErgTotal - 1
```



30

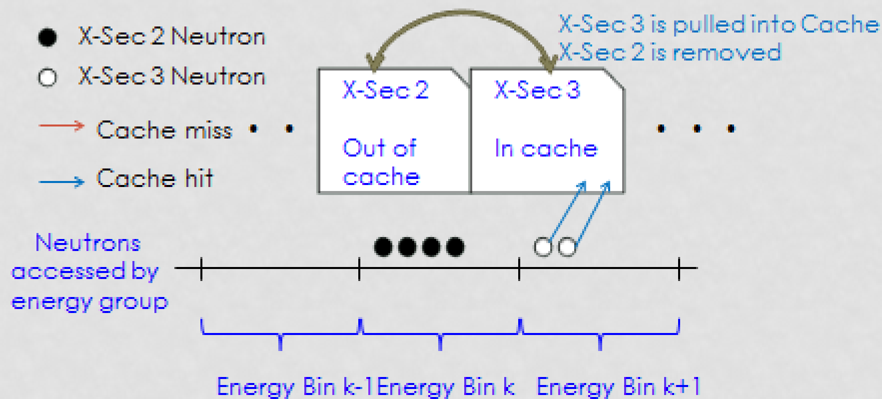
NEW CROSS SECTION DATA CACHE



- Energy groups contain similar neutrons and force the same cache reuse utilizing temporal locality.

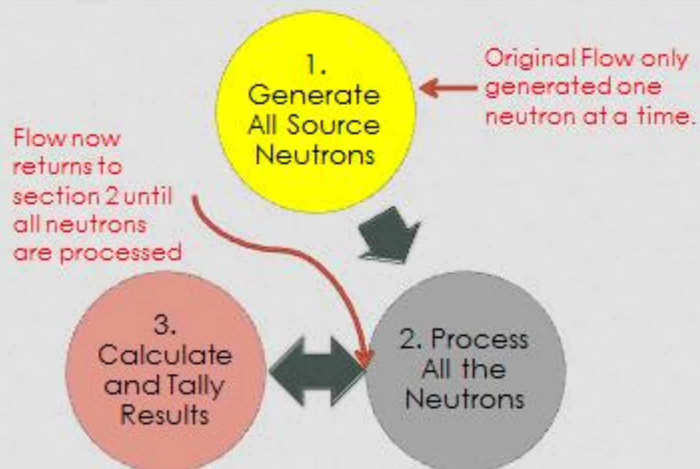
31

NEW CROSS SECTION DATA CACHE

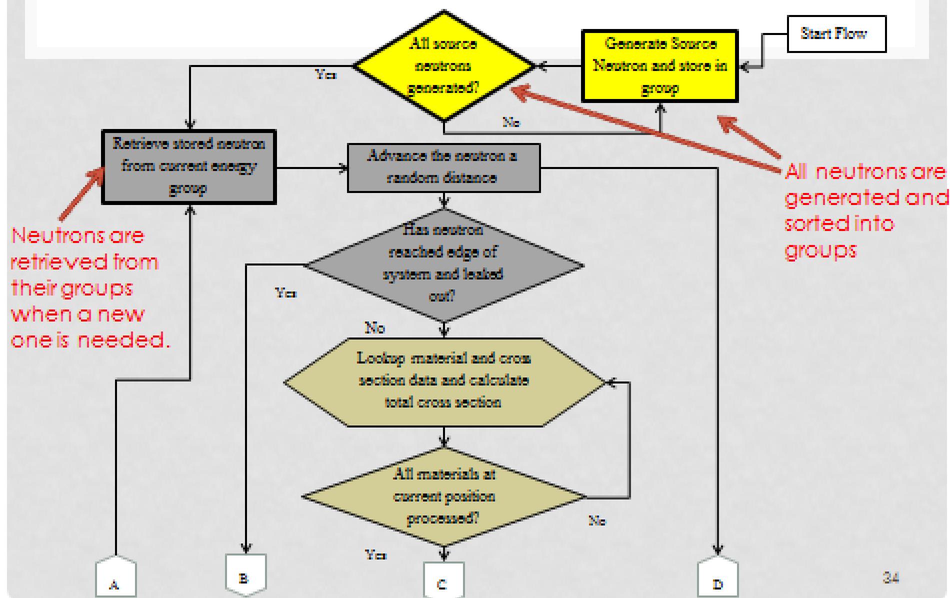


- Groups are structured to avoid X-Sec data swaps in and out of cache except when changing groups and then only causing a shift to the next set of data utilizing spatial locality.

32

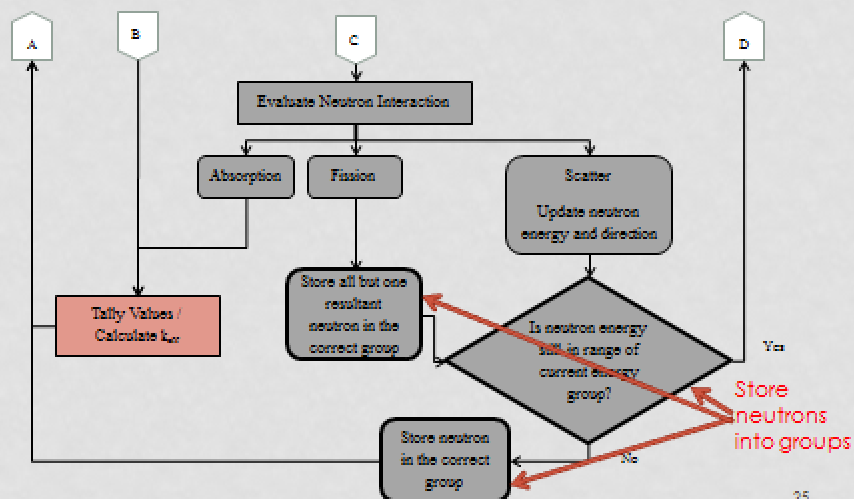


MODIFIED PROCESS FLOW - OVERVIEW
1. Generate All Source Neutrons
Original Flow only generated one neutron at a time.
Flow now returns to section 2 until all neutrons are processed
3. Calculate and Tally Results
2. Process All the Neutrons
33



NEW MCNP FLOW DIAGRAM
Start Flow
Generate Source Neutron and store in group
All source neutrons generated?
Yes
No
Retrieve stored neutron from current energy group
Advance the neutron a random distance
All neutrons are generated and sorted into groups
Neutrons are retrieved from their groups when a new one is needed.
Has neutron reached edge of system and leaked out?
Yes
No
Lookup material and cross section data and calculate total cross section
All materials at current position processed?
No
Yes
A
B
C
D
34

NEW MCNP FLOW DIAGRAM (CONTINUED)



MATERIAL GROUPS

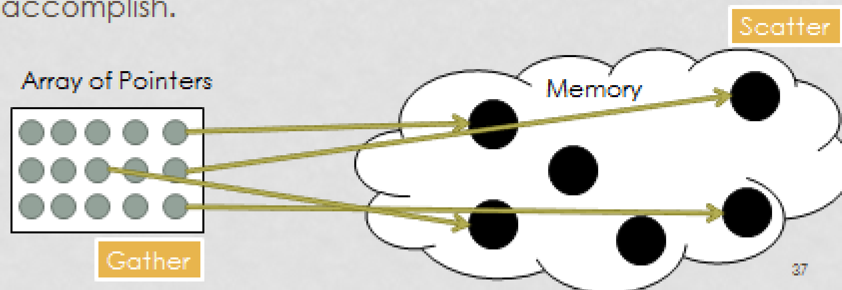
- Material Groups based on the geometry and material found in design.
- Cells containing same material were grouped together.
- Each group contained the same energy groups as previously shown.

```
allocate(IclGrpLookup(25))

IclGrpLookup(1) = 1
IclGrpLookup(2) = 2
IclGrpLookup(3) = 2
IclGrpLookup(4) = 7
IclGrpLookup(5) = 4
IclGrpLookup(6) = 3
IclGrpLookup(7) = 2
IclGrpLookup(8) = 2
IclGrpLookup(9) = 7
IclGrpLookup(10) = 5
IclGrpLookup(11) = 6
IclGrpLookup(12) = 6
IclGrpLookup(13) = 6
IclGrpLookup(14) = 6
IclGrpLookup(15) = 6
IclGrpLookup(16) = 6
IclGrpLookup(17) = 6
IclGrpLookup(18) = 6
IclGrpLookup(19) = 6
IclGrpLookup(20) = 6
IclGrpLookup(21) = 6
IclGrpLookup(22) = 8
IclGrpLookup(23) = 9
IclGrpLookup(24) = 9
IclGrpLookup(25) = 9

allocate(bpIclArray(iclGrpCnt))
```

36

APPLICATION OF MEMORY POINTER ARRAYS

- Pointers were used in order to reduce the need to move the neutron data from one group to another.
- The Pointers point to a section of memory where the neutron data was located. Then if the neutron needed to move to another group the pointer was moved instead.
- In this fashion the Pointers can perform Gather/Scatter that vector machines have specific hardware to accomplish.



MEMORY POINTER ARRAYS

- Code example of using pointers

Store neutron in memory

```
srcMasterTot = srcMasterTot+1
bpMasterArray(srcMasterTot) = curbp

tmpTotal = bpErgArray(tmpErgGrp)%bpErgTotal + 1
bpErgArray(tmpErgGrp)%bpErgTotal = tmpTotal
bpErgArray(tmpErgGrp)%bpArray(tmpTotal)%p_bp => bpMasterArray(srcMasterTot)
```

Designate memory location for pointer

NEW PROCESS RESULTS*

Code	Compiler O1 (mins) ± 5 mins	O1 Speedup from Base O1 ± 0.5%	Compiler O2 (mins) ± 5 mins	O2 Speedup from Base O1 ± 0.5%
Base	924.58 (15.4 hours)		830.05 (13.8 hours)	10.2%
Energy Bins (5 Bins)	887.44 (14.8 hours)	4%	774.53 (12.9 hours)	16.2%
Energy Bins (10 Bins)	895.30 (14.9 hours)	3.2%	799.28 (13.3 hours)	13.6%
Energy Bins w/ Pointers (5 Bins)	882.15 (14.7 hours)	4.6%	776.19 (12.9 hours)	16.1%
Energy Bins w/ Pointers (10 Bins)	879.27 (14.65 hours)	4.9%	769.89 (12.8 hours)	16.7%

*Results from running on Laptop:
- Intel(R) Core(TM)2 Duo CPU T9550 @ 2.66GHz (2 CPUs), ~2.7GHz
- 4 GBs RAM
- Windows 7 64-bit
- cygwin

39

NEW PROCESS RESULTS (CONTINUED)*

Code	Compiler O1 (mins) ± 5 mins	O1 Speedup from Base O1 ± 0.5%	Compiler O2 (mins) ± 5 mins	O2 Speedup from Base O1 ± 0.5%
Material/Energy Bins (5 Bins)	945.72 (15.8 hours)	-2.2%	841.29 (14.0 hours)	9%
Material/Energy Bins (10 Bins)	952.56 (15.9 hours)	-3.0%	842.42 (14.0 hours)	8.9%
Material/Energy Bins w/ Pointers (5 Bins)	938.82 (15.6 hours)	-1.6%	840.08 (14.0 hours)	9.2%
Material/Energy Bins w/ Pointers (10 Bins)	939.31 (15.7 hours)	-1.6%	836.28 (13.9 hours)	9.6%

*Results from running on Laptop:
- Intel(R) Core(TM)2 Duo CPU T9550 @ 2.66GHz (2 CPUs), ~2.7GHz
- 4 GBs RAM
- Windows 7 64-bit
- cygwin

40

K_{EFF} COMPARISON

Code	Compiler O1 k_{eff} $\sigma = 0.00004$	Reactivity ρ / Reactor Period τ	Compiler O2 k_{eff} $\sigma = 0.00004$	Reactivity ρ / Reactor Period τ
Base	0.99571	$-0.00431	0.99582	$-0.00420
		-31.35s		-31.85s
Energy Bins (5 Bins)	0.99575	$-0.00427	0.99583	$-0.00419
		-31.53s		-31.89s
Energy Bins (10 Bins)	0.99577	$-0.00425	0.99583	$-0.00419
		-31.61s		-31.89s
Energy Bins w/ Pointers (5 Bins)	0.99575	$-0.00427	0.99583	$-0.00419
		-31.53s		-31.89s
Energy Bins w/ Pointers (10 Bins)	0.99574	$-0.00428	0.99583	$-0.00419
		-31.48s		-31.89s

41

K_{EFF} COMPARISON (CONTINUED)

Code	Compiler O1 k_{eff} $\sigma = 0.00004$	Reactivity ρ / Reactor Period τ	Compiler O2 k_{eff} $\sigma = 0.00004$	Reactivity ρ / Reactor Period τ
Material/Energy Bins (5 Bins)	0.99588	$-0.00414	0.99593	$-0.00409
		-32.13s		-32.37s
Material/Energy Bins (10 Bins)	0.99583	$-0.00419	0.99588	$-0.00414
		-31.89s		-32.13s
Material/Energy Bins w/ Pointers (5 Bins)	0.99588	$-0.00414	0.99593	$-0.00409
		-32.13s		-32.37s
Material/Energy Bins w/ Pointers (10 Bins)	0.99585	$-0.00417	0.99578	$-0.00424
		-31.98s		-31.66s

42

EXPLANATION OF RESULTS

- Grouping by energy shows improvement over the Base code.
- Addition of pointers allowed the Energy w/ 10 Bins code set to perform the best by cutting down the time of moving the neutron data between groups.
- K_{eff} results for the code sets are within acceptable statistical variances.
  - Confidence is high that no errors were introduced to the calculations

43

OUTCOME IN MATERIAL/ENERGY CODE SETS

- Data may not be grouped by material and energy in a way that allows easy access.
  - This can be seen when attempt to organize just by material when no speed up was seen.
  - MCNP seems to have the cross-sections arranged by energy regardless of material.
- Arrays holding the elements become complicated and possibly too big for the memory.
  - This can cause the program to have to swap out the cross section data in memory to look up the particle from the arrays and then swap the data back in, resulting in cache thrashing.

44

CONCLUSION

- It was shown, that by creating groups based on the neutron energy, one can increase the efficiency of the reuse of data located in the cache.
- Best result for time in the groups used was the Energy Bins w/ Pointers (10 Bins) at a 16.7% ± 0.5% speed up.
  - A savings of 2 ½ hours for the configuration tested.

45

CONCLUSION

- The changes made were non-intrusive and did not require a complete rewrite of the MCNP code.
- There is no change in the cross section data.
- The program takes advantage of the temporal and spatial locality of the particles to increase its efficiency.
  - As long as the groups' structure can be kept small in size in order to restrict the neutrons to utilize the current data in cache.

46

KEY REFERENCES

- Brown, F., and Martin, W., 1984. Monte Carlo Methods for Radiation Transport Analysis on Vector Computers.
- Brown, F., and Martin, W., 1987. Status of Vectorized Monte Carlo for Particle Transport Analysis, International Journal of High Performance Computing Applications 2(1): 11-32

47