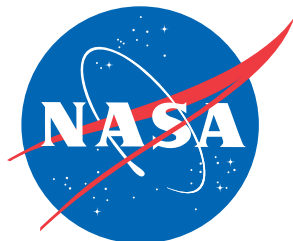NASA-TP-2018-219811

A Single Thread to Fortran Coarray Transition Process for the Control Algorithm in the Space Radiation Code HZETRN

Robert C. Singleterry Jr.
Langley Research Center, Hampton, Virginia

Desh Ranjan and Mohammad Zubair
Old Dominion University, Norfolk, Virginia

March 2018

NASA STI Program . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI program provides access to the NTRS Registered and its public interface, the NASA Technical Reports Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA Programs and include extensive data or theoretical analysis. Includes compila-tions of significant scientific and technical data and information deemed to be of continuing reference value. NASA counter-part of peer-reviewed formal professional papers but has less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

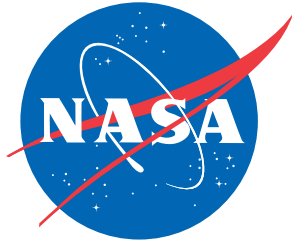- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include organizing and publishing research results, distributing specialized research announcements and feeds, providing information desk and personal search support, and enabling data exchange services.

For more information about the NASA STI program, see the following:

- Access the NASA STI program home page at http://www.sti.nasa.gov

- E-mail your question to help@sti.nasa.gov

- Phone the NASA STI Information Desk at 757-864-9658

- Write to:
  NASA STI Information Desk
  Mail Stop 148
  NASA Langley Research Center
  Hampton, VA 23681-2199

A Single Thread to Fortran Coarray Transition Process for the Control Algorithm in the Space Radiation Code HZETRN

Robert C. Singleterry Jr.
Langley Research Center, Hampton, Virginia

Desh Ranjan and Mohammad Zubair
Old Dominion University, Norfolk, Virginia

March 2018

Contents

List of Tables

List of Figures

List of Fortran Snippets

Acronyms

10GigE	10 Gigabit per second Ethernet
1GigE	1 Gigabit per second Ethernet
AMD	Advanced Micro Devices
AVX	Advanced Vector Extension
CPU	Central Processing Unit
DDR	Double Data Rate
DDTRB	Durability, Damage Tolerance, and Reliability Branch
Exa-	10^{18}
FDR	Fourteen Data Rate
FLOPS	FLOating Point operationS or FLoating point Operations Per Second plus others
FP	Floating Point
GB	GigaBytes
GHz	GigaHertz
GPGPU	General Purpose Graphics Processing Unit
GT	GigaTransfers
HT	AMD's HyperTransport or Intel's Hyperthreading
HZETRN	High Z and Energy TRaNsport
I/O	Input and Output
LaRC	Langley Research Center
MB	MegaBytes
MIC	Many Integrated Cores
MIMD	Multiple Instructions Multiple Data
MP/ACC	Multi-Processing/ACCelerator
MPI	Message Passing Interface
NASA	National Aeronautics and Space Administration
NFS	Network File System
nm	nano-meters
NTFS	New Technology File System
OLTARIS	On-Line Tool for the Assessment of Radiation In Space
PBS	Portable Batch System
Peta-	10^{15}
QPI	Intel's QuickPath Interconnect
SAS	Serial Attached SCSI - Small Computer System Interface
SATA	Serial AT (Advance Technology) Attachment
SGE	Sun Grid Engine
SIMD	Single Instruction Multiple Data
SIREST	Space Ionizing Radiation Effects and Shielding Tool
SSD	Solid State Drive
SSE	Streaming SIMD Extensions
UPC	Universal Parallel C

Abstract

Exa-scale computing is the direction by industry and government are going to generate solutions to problems they deem necessary. Computing hardware is being developed to achieve the transition from Peta-scale to Exa-scale with more CPUs that have more cores per CPU and more accelerators (GPGPUs and MICs) per node. To fully utilize the hardware available now and in the future, algorithms must become multi-threaded. There are a few methods to generate multi-threaded software such as MPI and OpenMP/OpenACC. This paper concentrates on using Coarray Fortran to convert the Fortran 95 based HZETRN code's control algorithm from a single threaded code to a multithreaded code. The resultant Coarray code was 32.5 times faster (with a theoretical speed-up of 74.5 times) than the single threaded version on the hardware tested, as reliable as the Fortran 95 version, and, as it uses native Fortran, was as maintainable as the Fortran 95 version. The Coarray code can be maintained by the same project engineers and scientists who created the original single threaded code. This transition process can be utilized on a C language based code with a compiler that has the UPC extensions to C.

1 Introduction and Background

In the past, project engineers and scientists achieved shorter time-to-solution by buying new and faster hardware and compiling and running their production codes on the new hardware. This new hardware allowed better results by either reducing tolerances in computing a result or allowing more results to be calculated in the same time frame. However, there has been a movement away from faster hardware towards more hardware. This has caused a radical advancement of computer hardware and software in the last few years. For hardware, chip manufacturers have reduced the speed of an individual CPU, or chipset, because of thermal limitations but have made available more cores or computational units per CPU, since the feature size on each chip is shrinking which allows more transistors to be placed on the chip – the so-called Moore's Law.[1] Manufacturers are also introducing new computational accelerators like GPGPUs and MICs. These give the user more options to accelerate their algorithms so that either more calculations can be run, generating more results, or enabling calculations that could not be done in the past because of the execution time needed to complete the calculation.

Giving the user slower but more cores to do the work instead of fewer and faster cores has been called by many experts "the single thread to multithread revolution". The top machines in the world achieved one Peta-FLOPS using the multithreading hardware construct in November 2008[a]. To obtain faster execution speeds, hardware vendors are now adding computational accelerators to their multithreaded machines and this is called a hybrid machine that can operate at 55 Peta-FLOPS[b]. However, most of this computing power comes from the accelerators and not the CPU based cores. Industry and government[2] estimate that Exa-FLOPS (1000 Peta-FLOPS) machines are needed to solve problems they deem necessary. The hybrid machine is positioned to become the Exa-scale machine by 2023.[3] These architectures, to include the hybrid architecture, will then impact smaller machines on the Top-500 List and will quickly impact desktop machines. Then multithreaded programming will be necessary at all levels from high performance computing to desktop analysis.

What has lagged behind this incredible hardware leap is the ability for the software developer to utilize this advanced clustered hardware easily. However, in the past couple of years, that lag is being tackled through improvements in compilers and their ability to automatically generate multithreaded code and accelerator programming paradigms. In the past, developers were limited to the MPI and/or OpenMP/OpenACC paradigms[c]. These paradigms are akin to assembly language inside a high level language (MPI) and suggestions to the compiler to help it multithread the code (OpenMP/OpenACC). However, several new paradigms, Coarrays for Fortran and UPC for C, are native to the languages. UPC will not be discussed further in this

---

[a]See the second ranked machine from Oak Ridge (JAGUAR) at the Top 500 List as of November 2008 website. Note, the top machine, ROADRUNNER from Los Alamos, had achieved one Peta-FLOPS in June 2008, but it is a hybrid machine and utilized accelerators to get to one Peta-FLOPS.

[b]See the second ranked machine from China (Tianhe-2) at the Top 500 List website as of November 2016. To compare, the NASA machine Pleiades, is thirteenth on that Top 500 list at seven Peta-FLOPS and is not a hybrid machine.

[c]MIMD and SIMD architectures: MIMD – a distributed memory system like Pleiades utilizing MPI. SIMD – an array processor, or GPGPU, like a single node on a distributed system or a shared memory architecture utilizing OpenMP/OpenACC. In a cluster environment, both programming paradigms can be used: OpenMP/OpenACC within a node and MPI between nodes.

report as the nature of Fortran Coarrays can be easily translated to UPC. This allows compiler manufacturers to fit the programmed algorithm to existing and new hardware transparently through native Fortran constructs that already exist: Arrays. These tools are relatively new and still being investigated by software developers. NASA has easy access to these new compilers but must change the method and sometimes the algorithm used to access the multithreaded nature of the new hardware using the new software.

For an entry level investigation of Coarrays, this report will try to illuminate two questions:

Q1 do Coarrays increase the efficiency of a code, and

Q2 can project engineers and scientists closest to the code use Coarrays to achieve multithreaded coding without the use of multithreading coding experts?

This paper will use the control algorithm in the space radiation particle transport code HZETRN[4–6] as an example to answer question Q1. The suggestion put forward in question Q2 is that if the answer to question Q1 is true, then the project engineers and scientists who are closest to the project should be and can be doing the coding without the assistance of multithreading coding experts. While this suggestion is subjective and influenced by many factors unique to each coding project, if the answer to question Q1 is true, then Q2 is a question that each project needs to answer.

1.1 Fortran

While Fortran is perceived as a deprecated language by many (the language is no longer offered in most computer science curricula), it is far from dead. In technical computing, Fortran offers explicit data references in the form of native Fortran arrays that speed the execution of programs by having the compiler determine where data exists in memory at the time of compilation instead of at runtime. If space is allocated at runtime, then data is placed there and the starting address of the data is explicitly associated with the data. With C and C++, an indirect pointer to the starting address of the data is associated with the data and not the address itself. C and C++ have no programming constructs like a Fortran array because C and C++ treat an "array" as just a pointer to contiguous memory which must be dereferenced before an array element can be accessed. This is twice the number of memory lookups than using Fortran arrays. However, constructs within C and C++ are now allowing explicit data references for interoperability with Fortran and the speed associated with this data reference methodology.

1.1.1 Fortran Coarrays

While C++ is expanding its capabilities rapidly for multithreaded processing, so is Fortran. Coarrays were added to the Fortran 2003 Standard. Currently, the Fortran 2008 Standard[7] is in force and most compilers conform to that standard. Only Cray (version 8.4.0 and higher), Intel (version 16.0 and higher), and GNU (version 5.2 and higher) Fortran compilers currently implement Coarrays. The 2008 Standard added more capability and clarified some language difficulties associated with Coarrays. The Fortran standard committee is currently working on a minor revision of the 2008 standard to be called Fortran 15[7] and introduces more capability associated with Coarrays.

A Coarray is the same as the native Fortran array construct except that one or more of the array dimensions designates a thread or execution of that particular array element on a separate core. While this is a simple construct and native to Fortran programming, it has hidden complexities. Unlike the MPI multithread programming paradigm, a few simple rules added to a Fortran programmer's knowledge can create native Fortran coding that allows the use of the new hardware and increases the execution speed of the code. With pure MPI calls instead of Coarrays, those coding complexities must be explicitly managed by the programmer with non-native Fortran MPI library subroutines. The resultant pure MPI code is usually faster than Coarray code; however, it is more complex to program and maintain – as illustrated by a relevant analogue from the history of computing: high level versus assembly programmed code.

Assembly code programming produces executables that are very fast since a skilled programmer is directly manipulating the basic objects and processes in the CPU. However, it is complex and very time consuming to produce the coding necessary to perform the desired task and requires a more in-depth understanding of the hardware and software interaction at a low level. If pure speed is all that is relevant, then assembly code is the programming paradigm of choice. Usually technical programming performs the same operations

on large quantities of data. In assembly language, this becomes tedious and error prone. Therefore, higher level programming languages were introduced. The C language macrotizes assembly language tasks to hide the complexity. This can be very useful but still is a very complex programming paradigm; however, it is portable amongst CPU architectures. High level languages like Fortran, Cobol, C++, Pascal, Ada, etc. abstract the CPUs basic objects and processes into appropriate constructs. For Fortran, these constructs are variables and arrays, arithmetic, loops and branches, and input/output. Fortran and the other high level languages will never be as fast as hand-coded assembly language; however, execution speed is not the only metric to optimize in programming. Other metrics include the overall time to solution, reliability and maintainability of the code, and portability which are more readily achieved by high level languages.

Coarrays abstract the basic MPI objects and processes into simple and native Fortran constructs to better balance all of the Fortran programmer's time and resource constraints. Therefore, Coarrays allow non-MPI experts, usually the project engineers and scientists, to create programs that can be used on multithreaded clustered hardware. It is true that expert MPI programmers would create faster, but more complex, programs. The MPI coding would be a "black box" to the project engineers and scientists. However, Coarray Fortran is more reliable and easier to maintain than equivalent MPI coding as Coarrays are extensions of native Fortran constructs and not "black boxes". As an added benefit, since the compiler manufacturers are responsible for the execution of Coarrays on the hardware, future hardware will still be able to utilize Coarrays without re-programming.

Coarrays are not a comprehensive solution for all algorithms. The algorithms must be created to fit within the Coarray construct and yet still produce the wanted results on the hardware available. Fortunately, most technical computing already utilize Fortran arrays. Coarrays are just extensions to arrays. In this paper, Coarrays will be applied to the control algorithm used in the HZETRN space radiation particle transport code. Section 1.2 discusses what parallelism is and some basic metrics. Section 2 will discuss the HZETRN control algorithm in its current form. Section 3 will convert the control algorithm to enable Coarrays. Section 4 will report on actual runtime experiments. Section 5 will conclude if Coarrays are viable for multithreading the HZETRN control algorithm and speculate on how different the resultant code is from the original and if project engineers and scientists can utilize this coding paradigm.

1.2 Parallelism

Parallelism is a multifaceted subject as Section 1 shows. It is clear that computational machines are becoming more complex and software is not abstracting the hardware[d] to the programmer unless experts in the fundamentals of parallelism are involved. The question really is: what is parallel and how is it implemented? This section is not a tutorial on parallelism, but a discussion about it is necessary to see how Coarrays fit into parallelism and how the project engineers and scientists class of programmers fit into utilizing it.

Coarrays usually bundle MPI calls and are suppose to take the complexity out of utilizing MPI. Therefore, the control algorithm being converted from serial to parallel utilizing Coarrays must already have a MIMD, or a distributed memory environment through multiple threads running the same algorithm on multiple data streams, compatible algorithm. The distributed memory environment means that computer nodes are connected by a network infrastructure that link data between the nodes and presents a seamless memory environment (arrays in Fortran) to the programmer. Therefore, data must be spread and collected to a master thread or acquired by a thread through disk access. If a thread needs data from another thread, that must be accommodated also. The concept of data passing by Coarrays is straight forward, since the programmer's mechanism is the well understood Fortran array, which abstracts the MPI calls that do the actual data passing.

The "thread" is the new idea to a serial programmer. The interaction of these threads and their data create new program logic error paths that must be avoided. The structure of Fortran and Coarrays helps with avoiding these error paths, but do not eliminate them. In order to understand what an individual thread must accomplish, the avoidance of these error paths can cause threads to wait for data to arrive from other threads. Of course, while the thread is waiting, it is not computing and therefore, not being "parallel".

---

[d] "abstracting the hardware" is a term of art. Its meaning is to hide the hardware from the software programmer and give the hardware details a simpler interface to the programmer. This is akin to how a high level language abstracts the details of each CPU from the programmer. In Fortran, this allows a programmer to code `x = a + b` and not worry about the details of the CPUs registers, adding hardware, memory hierarchy, and other details.

The "Do Loop" structure in Fortran is used to index through Fortran arrays and is essential in the parallel Coarray structure. However, if part of a computation in a thread is dependent on the outcome of a different thread, that can slow down the overall performance. Also, if one thread has more work to do than another thread, then overall performance can be affected.

A visual mechanism can be used to see the overall performance of a parallel program. For this paper, each thread is depicted as either green for computing or red for idling. All threads are depicted at once. A simple example will be used to show the features of one of these plots. Figure 1a shows the serial version of this example which consists of reading some input, looping over a single index (1-4) without data dependencies, and writing output. Figure 1b shows the parallel version of this plot. The dependent axis is the concurrent thread count. It is assumed each loop index takes y milliseconds to execute, x milliseconds to read the data, and z milliseconds to write the data. It is also assumed that data transfer is very fast and therefore not depicted on the plots. All threads execute 100% of the time (i.e., no red).

Figure 1a shows that the serial code takes $x + 4y + z$ milliseconds to execute while the parallel code in Figure 1b takes $x + y + z$ milliseconds to execute. The gives a speed-up of $(x + 4y + z)/(x + y + z)$. This speed-up seems to be in error. Since there are four threads, intuitively, shouldn't it run four times faster? Assume that if $x = y = z$, then the speed-up is two because there are still serial parts of the code that must be accounted for.

If all parts of the program can run in parallel, then the best a programmer can do is a factor of four speed-up. Amdahl's law[8] gives the actual speed-up of a whole program due to parallelism

$$SU(s) = \frac{1}{(1 - p) + \frac{p}{s}}$$

where SU is the speed-up of the program, s is the speed-up of the part of the program that is serial, and p is the part of the program that is parallel. This law can be very complex in this form if multiple parts of the code are sped-up at different rates (as with the HZETRN example used in this paper). Therefore, this paper will report the theoretical maximum speed-up, TSU, by taking the limit as s goes to infinity of $SU(s)$

$$TSU = \lim_{s \to \infty} SU(s) = \frac{1}{1 - p}.$$

For the $x = y = z$ example above, $s = 4$ and $p = 4/6$ which gives $SU(s) = 2$ and $TSU = 3$. The actual speed-up was shown to be two. Therefore, even though four threads are used concurrently in this example, only a two times speed-up is seen with a theoretical maximum speed-up of three. This is driven by the amount of serial code left after parallelization. This effect will be shown by the HZETRN control algorithm example as appropriate.

2 HZETRN

HZETRN is a particle transport code for space radiation using the linearized Boltzmann transport equation. The code analyzes space radiation particles (projectile particles of the stable nuclei of hydrogen through nickel) as they traverse materials in a space vehicle or habitat (spacecraft). The physics of the code includes interactions such as nuclear stars or the interaction of projectile particles with the nuclei of the materials in the spacecraft to create secondary radiation, the slowing down of charged particles through materials by Coulomb scattering with the materials' electrons and nuclei, and the nuclear scattering of particles that do not create nuclear stars. For this analysis, the computational process flow and associated data structures are critical, whereas the physics of HZETRN are a representation of the computation process and less important in this context.

HZETRN developers used various approximations to achieve the solution algorithm. The eight dimensional phase space represented in the full Boltzmann transport equation (one of particle type, three of space, one of energy, two of angle, and one of time – $\psi_p(x, y, z, E, \theta, \phi, t)$) was reduced to three (one of particle type, one of space and one of energy – $\psi_p(x, E)$) through the use of the straight-ahead, the continuous slowing down, and steady state approximations among others. It is also assumed that the $Z > 2$ particles do not gain total energy through any interaction and do not create larger particle types in any interaction. Numerical marching techniques are then used to solve the problem along a gridded spatial number line in the direction

of the projectile particles. The marching technique is a function of the current location on the number line and the next location on the number line. This can be represented as a Fortran subroutine called MARCH as shown in Fortran Snippet 1.

Fortran Snippet 1: SINGLE MATERIAL MARCH CALL

```fortran
! x is allocated as x(Number_Spatial_Points)
    Call Read_Spatial_Data ( 1, Number_Spatial_Points, x )
!
    Allocate ( psi(Number_Energy_Points, &
                   Number_Particle_Types, &
                   Number_Spatial_Points) )
!
! Set the Boundary Condition stored in F(l,m)
!
BC: Do l = 1, Number_Energy_Points
PA:   Do m = 1, Number_Particle_Types
         psi(l,m,1) = F(l,m)
      End Do PA
   End Do BC
!
! March down the spatial number line for all energies and particle types
!
TR: Do i = 2, Number_Spatial_Points
EN:   Do l = 2, Number_Energy_Points
PB:      Do m = 2, Number_Particle_Types
            Call MARCH ( x(i-1), psi(l-1,m,i-1), x(i), psi(l,m,i) )
         End Do PB
      End Do EN
   End Do TR
```

The boundary condition of interest is `F(l,m)`, `Read_Spatial_Data` is a subroutine that reads the spatial data from a file, `MARCH` is the subroutine that marches the solution to the next phase space point, `psi` holds the phase space solutions, and `x` holds the spatial grid. For executions with a galactic cosmic ray boundary condition, `Number_Energy_Points` is 100 and `Number_Particle_Types` is 59. The value of `Number_Spatial_Points` can vary for each application, but 11 has been used for this analysis and is comparable to values used on the OLTARIS website[9] which implements the HZETRN solution algorithm. This sets the number of calls to MARCH at 57,420 and can be handled by all modern single threaded machines in a few minutes.

In order to evaluate complex spacecraft shielding geometry with HZETRN, ray-tracing procedures are utilized. In a ray-trace procedure, the material types and thicknesses traversed by a large number of rays covering the full 4π steradian are computed and then are coupled to the HZETRN solution to estimate radiation exposure. Two solution methods exist that OLTARIS calls ray-by-ray and interpolation. The ray-by-ray solution method performs an execution of HZETRN on each ray and integrates the response function from all the rays at the source point for the rays. This method's conversion to Coarrays will be explored in the future. This paper just focuses on the interpolation solution method, which creates a database of all combinations of materials for preset thicknesses of those materials. The ray trace is then used to interpolate on this database and the response functions integrated as with the ray-by-ray method. The control algorithm that creates the database is what will be converted to Coarray Fortran.

The current configuration of the interpolation method used for HZETRN is shown graphically in Figure 2. There are three materials with 11 thicknesses per material. The boundary condition is at box (1,1,1) in the figure. The marching technique is broken down into three passes. The first pass, Figure 2a, is an execution of the boundary condition through the first material from box two to box 11. This calculation becomes the boundary condition for the second pass. Figure 2b shows that the second pass is executed 11 times and that takes the second material from box two to 11. This calculation then becomes the boundary condition for the

third pass. Figure 2c shows that the third pass executes 121 times and that takes the third material from box two to 11. This changes Fortran Snippet 1 to Fortran Snippet 2.

```
                    Fortran Snippet 2: THREE MATERIAL DATABASE MARCH CALL

 ! Number_Material_Points is allocated as Number_Materials_Points(3)
 ! x is allocated as x(max(Number_Material_Points),3)
     Call Read_Spatial_Data ( 3, Number_Material_Points, x )
 !
     Allocate ( psi(Number_Energy_Points, &
                    Number_Particle_Types, &
                    Number_Material_Points(1), &
                    Number_Material_Points(2), &
                    Number_Material_Points(3)) )
 !
BC:   Do l = 1, Number_Energy_Points
PA:     Do m = 1, Number_Particle_Types
            psi(l,m,1,1,1) = F(l,m)
          End Do PA
       End Do BC
 !
 ! First Pass
       j = 1
       k = 1
M11: Do i = 2, Number_Material_Points(1)
E1:    Do l = 2, Number_Energy_Points
P1:      Do m = 2, Number_Particle_Types
             Call MARCH ( x(i-1,1), psi(l-1,m-1,i-1,j,k), x(i,1), psi(l,m,i,j,k) )
           End Do P1
         End Do E1
       End Do M11
 !
 ! Second Pass
       k = 1
M12: Do i = 1, Number_Material_Points(1)
M22:   Do j = 2, Number_Material_Points(2)
E2:      Do l = 2, Number_Energy_Points
P2:        Do m = 2, Number_Particle_Types
               Call MARCH ( x(j-1,2), psi(l-1,m-2,i,j-1,k), x(j,2), psi(l,m,i,j,k) )
             End Do P2
           End Do E2
         End Do M22
       End Do M12
 !
 ! Third Pass
M13: Do i = 1, Number_Material_Points(1)
M23:   Do j = 1, Number_Material_Points(2)
M33:     Do k = 2, Number_Material_Points(3)
E3:        Do l = 2, Number_Energy_Points
P3:          Do m = 2, Number_Particle_Types
                 Call MARCH ( x(k-1,3), psi(l-1,m-2,i,j,k-1), x(k,3), psi(l,m,i,j,k) )
               End Do P3
             End Do E3
           End Do M33
```

```
        End Do M23
      End Do M13
```

To establish a baseline for comparison, a default set of values are used: `Number_Material_Points` are 11 each, `Number_Energy_Points` is 100, and `Number_Particle_Types` is 59. These values set the number of calls to MARCH at 7,636,860 which is again not unreasonable for a single thread on modern machines and compilers. It might take several tens of minutes depending on the chip set used. The amount of memory for the `psi` array is 15 MB (Megabytes) for 64-bit reals.

Something unexpected occurs if a non-expert user of HZETRN wants to increase the fidelity of their answer to say 170 particle types[e], have five or seven materials with 21 points per material[f] and have 100 energy points. This then gives the number of calls to MARCH at 68,331,077,100 for five materials and 30,618,505,180,000 for seven materials. Therefore, the time to execute this model in a single thread could take 0.51 years for five materials and 225.1 years for seven materials if the default database took 30 minutes to generate. Even if the default database took one minute to generate, then these new models would take 6.2 days for five materials and 7.5 years for seven materials. The memory needed for seven materials is 1.2 GB (Gigabytes) for 64-bit reals.

2.1 Conversion of the HZETRN Control Algorithm from Fortran 77 to Modern Fortran

The history of the development of HZETRN is an important aspect to the Coarray investigation. When HZETRN was officially released in 1995, the user interacted with it by editing the source code in Fortran 77 with the model to be analyzed, compiling the new source code, and running the executable to get the wanted results. Only experts could run the code and get any intelligible results out of it; even then, experts made mistakes and those errors had a large chance of being propagated in the code and taint other results.

A few years after the official 1995 release, Singleterry created an input deck driven version of the code for use by non-experts. This fixed the coding in time, or archived it, because it was no longer edited every time a model was run. This also had the added benefit of prescribing the compilation parameters to fit the coding needs. This version was never released as an official version because the resultant executable was very large in order to have predefined arrays to cover all probable inputs. Singleterry then reworked the code to incorporate dynamically allocated arrays and other elements from Fortran 95 as they were introduced into the production Fortran compilers. This reduced the size of the resultant executable and made the code run faster. However, the code was still input deck driven with text based data files and never released as an official version.

A web interface was added to this last version of HZETRN created by Singleterry and the result was called SIREST.[12] The HZETRN code was broken into pieces and made easier to maintain along with the ancillary codes associated with HZETRN. More Fortran 95 elements were added to aid in compilation and runtime. The web interface to SIREST became a maintenance problem very quickly due to the tools used to create the website. A proposal was put forward and accepted by NASA to make HZETRN and all of its ancillary codes into a modern and easier to maintain website called OLTARIS. The OLTARIS website is still operating but is still using the antiquated coding of Fortran 77/95 and the quirks of the compilers at the initial time it was broken up for SIREST.

Many changes have been made to Fortran in the intervening years. The Coarray investigation coincides with an investigation into what a modern Fortran version of HZETRN should be. The physics routines (those routines used by `MARCH` in this paper) are not being modernized at this time. First, deprecated and obsolete Fortran constructs are being eliminated (minus those used in the physics routines). Then, related variables are being grouped and placed into derived data types. Subprograms and their related data are being placed into modules (a crude first attempt at object oriented programming by encapsulating related

---

[e]This reference[10] makes a case that tracking 170 instead of 59 particles types is more accurate.

[f]Five or seven spacecraft materials is not unreasonable and could allow the OLTARIS programmers to enable the more accurate bi-directional neutron transport methodology[11] for the interpolation database. The center material would be tissue (and the source for the ray traces), mirrored by one spacecraft material and one or two shielding materials. With 21 points per material, the interpolation errors would be reduced. The ultimate effect of this new model would be to reduce the uncertainty of the solution and could be used to create a benchmark quality solution.

data and processes). The major modernization is in the input and the output of the code. The current version of HZETRN is based on an old Intel Fortran compiler which contained some proposed but not finalized Fortran 95 constructs. As Fortran 95 was finalized and language elements changed in the newer versions of the compiler, the older Fortran elements were not deprecated, but better, more readable and easier to maintain elements were allowed. These newer and Fortran Standard compliant elements were never implemented in HZETRN and the non-standard coding was left alone. Therefore, the central read and write routines, which had the worst of the non-standard code, were rewritten based on unlimited polymorphic constructs. Again, the physics routines in the code were not modified for this work, still contain deprecated and obsolete constructs, and should be pointing to pre-calculated values instead of recalculating or re-reading those values. The results of this paper for Coarrays will be incorporated into this testbed and published separately. The Fortran Snippets in this paper contain some of the modernized Fortran constructs developed and not the current production version (OLTARIS) constructs.

3 Conversion of the HZETRN Control Algorithm to Coarrays

The Coarray construct in Fortran allows array memory allocation to be more amenable to distributed memory parallel processing. This changes the meaning of the arrows in Figure 2. Instead of executing in a single thread, they can now be executed concurrently or multithreaded. This gives two passes that are multithreaded as the first pass is still a single thread. The control algorithm for `MARCH` as shown in Fortran Snippet 2 has to change. This affects the `psi` array as shown in Fortran Snippet 3.

Fortran Snippet 3: DEFINITION AND ALLOCATION OF THE psi COARRAY

```
Real, Allocatable, Dimension(:,:,:), Codimension[:,:] :: psi
!
Allocate ( psi ( Number_Energy_Points, &
                 Number_Particle_Types, &
                 Number_Material_Points(3))[Number_Material_Points(1),*]
```

The ∗ is necessary in the second dimension of the Coarray as per the standard.

Since each thread is being executed concurrently, each thread needs to have an identity. Therefore, new code must be added to identify threads as shown in Fortran Snippet 4.

Fortran Snippet 4: ULTIMATE THREAD IDENTIFICATION

```
pid = this_image(psi)
p1 = pid(1)
p2 = pid(2)
```

This coding can only be used once `psi` has been allocated. Therefore, if thread identification is needed before the allocation of the `psi` Coarray, then Fortran Snippet 5 is used.

Fortran Snippet 5: INITIAL THREAD IDENTIFICATION

```
p = this_image()
```

Thinking in Parallel: This is where a new type of thinking is necessary for programmers who predominately code single threaded programs. A Coarray program executes by starting all threads at the same time. If there are more threads than cores, then the threads are scheduled by whatever algorithm the operating system has implemented to manage threads. For the Windows and Linux operating systems, the algorithm starts all the threads and lets them compete amongst each other for CPU resources utilization using a scheduler based on the round robin algorithm with qualifiers like priority.

An example of this thinking in parallel can be explored through the simple task of initializing the `psi` array. The Fortran 77 code is shown in Fortran Snippet 6.

```
                Fortran Snippet 6: FORTRAN 77 INITIALIZATION OF PSI

I1: Do i = 1, Number_Material_Points(1)
I2:   Do j = 1, Number_Material_Points(2)
I3:     Do k = 1, Number_Material_Points(3)
IE:       Do l = 1, Number_Energy_Points
IP:         Do m = 1, Number_Particle_Types
              psi(l,m,i,j,k) = 0.0
            End Do IP
          End Do IE
        End Do I3
      End Do I2
    End Do I1
```

Of course, in most Fortran 95 compilers (and now is standard in Fortran 2003), to initialize an array is `psi = 0.0`. This allows the compiler to optimize or parallelize the step without the programmer dictating the algorithm. However, to perform the same task in a Coarray program is shown in Fortran Snippet 7.

```
                Fortran Snippet 7: COARRAY INITIALIZATION OF PSI

    i = p1
    j = p2
I3: Do k = 1, Number_Material_Points(3)
IE:   Do l = 1, Number_Energy_Points
IP:     Do m = 1, Number_Particle_Types
          psi(l,m,k)[i,j] = 0.0
        End Do IP
      End Do IE
    End Do I3
    Sync All
```

This utilizes `Number_Material_Points(1)*Number_Material_Points(2)` threads. The new Fortran command introduced here is `Sync All`. This command makes sure that all threads are finished with their task before execution continues past this point in the code.

It is important to remember that each thread only has direct (local memory) access to certain data. In the case for Fortran Snippet 7, only the portion of the `psi` array that the `(p1,p2)` thread has initialized is local. However, this thread can access the other parts of the `psi` array non-local to it, but an underlying data transfer call (for the Intel compiler, a series of MPI calls) is made and will, of course, take more time than a local memory data access.

Reading a File and Processing the Data: Learning to think in parallel can have a stumbling block if the programmer is not careful and does not understand that all threads execute at once. The first difficulty encountered is reading a file and making that data available on all threads. Only one thread can read a file safely (i.e. without race conditions[g]). Even though a file can be opened `READONLY` for concurrent utilization, records are locked when being read by a thread and can cause an error if two or more threads are trying to read the same record (a race condition). Therefore, an if-block construct should be placed around the `Open/Close` Fortran construct so that all threads do not try and read the file. Also, that data only exists on the thread that read it and must be propagated (or copied) to the other threads to be local to all threads. This can be accomplished for the spatial data with normal Fortran 95 constructs as shown in Fortran Snippet 8.

---

[g]Parallel programming jargon for an explicit coding error type. See `https://en.wikipedia.org/wiki/Race_condition` for a detailed explanation of this term.

```
┌──────────── Fortran Snippet 8: Acquire Spatial Data on a Single Thread ────────────┐
│ Real, Dimension(:)    :: Number_Material_points_SD                                  │
│ Real, Dimension(:,:) :: x_SD                                                        │
│ If ( p .eq. 1 ) Then                                                                │
│   Open ( unit=10, file='spatial.dat', status='old' )                               │
│   Allocate ( Number_Material_Points_SD(3) )                                         │
│   Read ( unit=10, fmt=* ) (Number_Material_Points_SD(i), i=1,3)                     │
│   MaxX = maxval ( Number_Material_Points_SD )                                       │
│   Allocate ( x_SD(MaxX,3) )                                                         │
│   Do i = 1, 3                                                                       │
│     Read ( unit=10, fmt=* ) (x_SD(j,i),j=1,Number_Material_Points_SD(i))           │
│   End Do                                                                            │
│   Close ( unit=10)                                                                  │
│ End If                                                                              │
│ Sync All                                                                            │
└────────────────────────────────────────────────────────────────────────────────────┘
```

The suffix _SD designates a "Single Dimension" or no Coarray Dimension. The data only exist on thread 1 and must be copied to the other threads for utilization there. This can be accomplished in a Fortran 2008 manner as shown in Fortran Snippet 9.

```
┌──────────── Fortran Snippet 9: Copy Spatial Data to Other Threads, Fortran 2008 ────────────┐
│ Real, Dimension(:),   Codimension[:] :: Number_Material_Points                              │
│ Real, Dimension(:,:), Codimension[:] :: x                                                   │
│ Allocate ( Number_Material_Points(3)[*] )                                                   │
│ Allocate ( x(MaxX,3)[*] )                                                                    │
│ If ( p .eq. 1 ) Then                                                                         │
│   Number_Material_Points(:) = Number_Material_Points_SD(:)                                   │
│   x(:,:)                     = x_SD(:,:)                                                      │
│   Deallocate ( Number_Materials_SD, x_SD )                                                   │
│ End If                                                                                        │
│ Sync All                                                                                      │
│ If ( p .ne. 1 ) Then                                                                          │
│   Number_Material_Points(:) = Number_Material_Points(:)[1]                                    │
│   x(:,:)                     = x(:,:)[1]                                                       │
│ End If                                                                                         │
│ Sync All                                                                                       │
└────────────────────────────────────────────────────────────────────────────────────────────┘
```

If the 2018 Fortran standard is used, then Fortran Snippet 10 is used.

```
┌──────────── Fortran Snippet 10: Copy Spatial Data to Other Threads, Fortran 2018 ────────────┐
│ Real, Dimension(:),   Codimension[:] :: Number_Material_Points                               │
│ Real, Dimension(:,:), Codimension[:] :: x                                                    │
│ Allocate ( Number_Material_Points(3)[*] )                                                    │
│ Allocate ( x(MaxX,3)[*] )                                                                     │
│ If ( p .eq. 1 ) Then                                                                          │
│   Number_Material_Points(:) = Number_Material_Points_SD(:)                                    │
│   x(:,:)                     = x_SD(:,:)                                                       │
│   Deallocate ( Number_Materials_SD, x_SD )                                                    │
│ End If                                                                                         │
│ Call Co_Broadcast ( Number_Material_Points, 1 )                                               │
│ Call Co_Broadcast ( x, 1 )                                                                     │
└────────────────────────────────────────────────────────────────────────────────────────────┘
```

This type of two step procedure must be performed for all the data to be read. The beauty of Coarrays is that while thread 1 is reading the spatial data, thread 2 is concurrently reading the energy data, thread 3 is concurrently reading the particle type data, and so on. Therefore, all the data can be read in and copied around concurrently.

First Pass: After all the input data is read and propagated to all the threads, the three passes of the `MARCH` subroutine can begin. The coding for the first pass must change as in Fortran Snippet 11 (but it is still a single thread).

```
Fortran Snippet 11: FIRST PASS FOR THE COARRAY MARCH SUBROUTINE

Real :: Temp
!
! First Pass
!
     If ( p1 .eq. 1  .and.  p2 .eq. 1 ) Then
M11:   Do i = 2, Number_Material_Points(1)
E1:      Do l = 2, Number_Energy_Points
P1:        Do m = 2, Number_Particle_Types
             Call MARCH ( x(i-1,1), psi(l-1,m-1,1)[i-1,1], x(i,1), Temp )
             psi(l,m,1)[i,1] = Temp
           End Do P1
         End Do E1
       End Do M11
     End If
!
     Sync All
```

The Coarray dimension, `[p1,p2]`, is not required for the `x` array as it has been previously copied to all threads. Since the return value from the physics subroutine `MARCH` is a `Real` value and not a Coarray, the `Temp` variable is needed to store the value in the Coarray properly.

Second Pass: For the second pass, the Coarray coding needs to change as shown in Fortran Snippet 12.

```
Fortran Snippet 12: SECOND PASS FOR THE COARRAY MARCH SUBROUTINE

!
! Second Pass
!
     i = p1
     If ( p2 .eq. 1 ) Then
M22:   Do j = 2, Number_Material_Points(2)
E2:      Do l = 2, Number_Energy_Points
P2:        Do m = 2, Number_Particle_Types
             Call MARCH ( x(j-1,2), psi(l-1,m-2,1)[i,j-1], x(j,2), Temp )
             psi(l,m,1)[i,j] = Temp
           End Do P2
         End Do E2
       End Do M22
     End If
!
     Sync All
```

This utilizes `Number_Material_Points(1)` threads. Because of the Coarray structure, data is being passed from thread to thread as different cores are calculating different parts of the Coarray.

Third Pass: For the third pass, the Coarray coding is shown in Fortran Snipett 13.

```
Fortran Snippet 13: THIRD PASS FOR THE COARRAY MARCH SUBROUTINE
!
! Third Pass
!
      i = p1
      j = p2
M33: Do k = 2, Number_Material_Points(3)
E3:    Do l = 2, Number_Energy_Points
P3:      Do m = 2, Number_Particle_Types
             Call MARCH ( x(k-1,3), psi(l-1,m-1,k-1)[i,j], x(k,3), Temp )
             psi(l,m,k)[i,j] = Temp
           End Do P3
         End Do E3
      End Do M33
!
      Sync All
```

This utilizes `Number_Material_Points(1)` times `Number_Material_Points(2)` threads.

Advantages of Coarrays in the Conversion: There are some advantages to this method of programming in parallel. First, the actual coding does not change significantly from the serial version. An MPI-based version of the HZETRN control algorithm would be complex to generate and not easy to maintain by the project programmers (and is why an MPI version was not generated for this analysis). The Coarray coding on the other hand is very much like the original Fortran 77 coding with a small amount of new thinking and syntax and one new Fortran statement. As shown with the use of the `Temp` variable, Coarray elements are not identical to Fortran 77 array elements and some simple accommodations must be made. Second, the amount of data that must be stored by each thread is much smaller than for the single threaded version of the code. The single threaded version needs the entire `psi` array to reside on or be accessible to a single thread tied to a single core. The manner in which memory is managed and utilized in Intel (QPI) and AMD (HT) CPUs mandates that some of the data needs to be transferred from other cores on the same chip which is not really local to the core executing the code and wanting the data. Also, the smaller the data set, the more effective on-core and on-chip data cache utilization becomes. These memory access issues are important for the overall throughput of the code but have not been investigated in this work.

Output of the Data: To output the `psi` array in the single threaded code, a simple `Write` procedure is necessary as sown in Fortran Snippet 14.

```
Fortran Snippet 14: SINGLE THREADED CODING NECESSARY TO OUTPUT psi
Open ( unit=10, file='Flux.dat', status='replace' )
Write ( unit=10, fmt=* ) Number_Energy_Points, &
                         Number_Particle_Types, &
                         Number_Material_Points(1), &
                         Number_Material_Points(2), &
                         Number_Material_Points(3)
Write ( unit=10, fmt=* ) psi
Close ( unit=10 )
```

This writes `psi` in normal Fortran order `(l,m,i,j,k)` where `l` is indexed first, `m` second, `i` third, etc.

For the Coarray version, the data exists in different threads. Those threads could all be on one CPU or node or on many nodes connected with a networking infrastructure. There are many methods to gather the

data to one thread and write the data to disk. To try and write the data in the same order as is written in Fortran Snippet 14, the data transfer would be very inefficient. Therefore, the data is written efficiently for Coarrays as shown in Fortran Snippet 15.

```
        Fortran Snippet 15: COARRAY OR MULTITHREADED CODING NECESSARY TO OUTPUT psi

COA:   If ( p .eq. 1 ) Then
          Open ( unit=10, file='Flux_coa.dat', status='replace' )
          Write ( unit=10, fmt=* ) Number_Energy_Points, &
                                   Number_Particle_Types, &
                                   Number_Material_Points(1), &
                                   Number_Material_Points(2), &
                                   Number_Material_Points(3)
COA1:   Do i = 1, Number_Material_Points(1)
COA2:      Do j = 1, Number_Material_Points(2)
              Write ( unit=10, fmt=* ) psi(:,:,:)[i,j]
           End Do COA2
        End Do COA1
      End If COA
```

No `Sync All` is needed because the threads are accessed one at a time by the `COA1` and `COA2` loops. To read the data from the file, the Coarray structure in a single thread is structured like Fortran Snippet 16.

```
        Fortran Snippet 16: READING COARRAY WRITTEN psi IN TO A SINGLE ARRAY

Allocate ( psi_coa(1:Number_Energy_Points, &
                   1:Number_Particle_Types, &
                   1:Number_Material_Points(3), &
                   1:Number_Material_Points(2), &
                   1:Number_Material_Points(1)) )
Open ( unit=10, file='Flux_coa.dat', status='old' )
Read ( unit=10, fmt=* ) psi_coa
Close ( unit=10 )
```

This can then be compared with `psi` from Fortran Snippet 14 for each index.

The important parts of the HZETRN interpolation method control algorithm have been converted from a single thread version to a multithreaded, Coarray version. The resultant Coarray coding is very understandable by Fortran 77 programmers and mirrors the single threaded code.

Amdahl's Law: The control algorithm as stated in this paper is complex. There are three passes with some being serial, some being parallel, and have differing speed-up rates. Some of the threads could be idle while some are still calculating. A calculation of $SU(s)$ would be very complex due to the multiple speed-ups available to the code in different places. Therefore, the theoretical speed-up, TSU, will be calculated and reported with p being the ratio of the calculation serial execution time (the rows labeled Calculation in the tables) over the total serial execution time (the rows labeled Total in the tables) for each case.

Figure 3 shows a possible timing versus thread count diagram for the HZETRN control algorithm. Block A represents the initial setup. Block B represents the reading and sharing of the data where the green color represents computation and the red color represents idle time. Block C represents initialization of the `psi` array for each pass. Block D represents the first pass which is serial. Block E represents the second pass. Block F represents the third pass. Finally, Block G represents the output of the results. The complexity is evident in this figure and why $SU(s)$ is very complex to compute.

4   Runtime Comparisons

Since the physics of the code has been stripped out of this work to focus on the Coarray aspects, some data generation had to be accomplished in order to make sure that the Coarray version gave the same results as the single threaded version. To this end, the `MARCH` subroutine was simulated as shown in Fortran Snippet 17.

```
             Fortran Snippet 17: MARCH Subroutine Structure for Testing

Module MARCH_Mod
! No explicit typing
    Implicit None
! Make all names private
    Private
! Make these Subroutine names Public
    Public :: MARCH
Contains
    Subroutine MARCH ( psi_next, X_next, psi, X )
        Use Target_Material_Mod, only : TM
        Implicit None
        Real, Intent(in)  :: psi, X, X_next
        Real, intent(out) :: psi_next
        Integer           :: i, j
FK1:    Do j = 1, 5000
          psi_next = psi
FK2:      Do i = 1, TM%Num_Isotopes
            psi_next = psi_next + ( TM%A_Target_Material_p(i) * &
                                    TM%Z_Target_Material_p(i) * X )
          End Do FK2
        End Do FK1
        Return
    End Subroutine Propagate
End Module Propagate_Mod
```

The `TM` defined type structure contains the atomic number (`Z`) and atomic weight (`A`) of the target material for each pass. This algorithm creates a unique set of numbers that can be compared between versions.

The next issue that this algorithm addresses is the nature of the Coarray execution: the time to execute the underlying MPI subroutine calls used by the Coarray runtime executable. This is called Coarray overhead or the tasks needed to be performed simply because Coarrays are used. The actual routine called in HZETRN simulated by this `MARCH` routine calls numerous functions and performs numerous calculations. This takes time to perform. This is simulated with the loop labeled `FK1` in Fortran Snippet 17. With this loop, the Coarray overhead is then not the only execution item being timed. No study was performed to actually determine that 5000 loops was sufficient to cover the actual execution time for the real routine. However, 5000 gives wall-clock times that can be compared without Coarray overhead being the primary execution time consumer.

4.1   Timing Issues

At NASA LaRC, there are two cluster machines used for this analysis: the K and HZE clusters. The K cluster is a LaRC resource that is not a hybrid machine. Computing power comes from CPUs only and is a smaller mimic of the Pleiades cluster, the NASA (Agency) engineering and scientific computing resource. The HZE cluster is a branch resource available to one of the authors, Singleterry. Computing power comes from CPUs only. Also used for the timing comparisons is a workstation class Windows laptop, a workstation class Windows desktop, and a Linux based workstation. Hardware, compiler, and operating system details are shown in Table 1.

The timing data gathered on each execution was the time to read (and copy to all threads for the Coarray version) all the input, the three pass database generation calculation, the output of the `psi` array, and then the total time for execution. This was performed for the single thread and the Coarray versions of the code. The timings were generated ten times on each system, averaged, and a variance (standard deviation in this case) calculated to look for calculation stability. After each execution of the single thread and the Coarray version, the output flux files were compared and validated.

4.2 Compiler Issues

Compilers are complex and the manufactures can change option defaults, even between minor versions. Therefore, using the same minor version for each machine is critical. All simulations were compiled with the Intel Fortran Compiler Version 16.0. In order to better understand how to compile a Coarray containing code, the compiler and its options must be explained and are complex.

There are three license versions of the Intel compiler but only two types of licenses matter for Coarray compilation. The first license type is found in the Composer and Professional Edition versions. They allow for single node execution of Coarrays using a runtime version of the Intel-MPI libraries. The Composer version was used on the the Linux and Laptop machines. The second license type is found in the Cluster Edition version. This allows for multiple node or distributed execution of a Coarray program again, using the Intel-MPI libraries. This version was used on the HZE and K clusters along with the Desktop workstation even though the Desktop was compiled to be run in the non-distributed or shared mode.

The Intel compiler has some details that must be addressed to get the best performance from a particular machine and consistent performance from all machines. The details of the HZE and K Clusters, the Linux, Desktop, and Laptop workstations are shown in Table 1. First, the two versions of the compiler noted in the table are deemed sub-minor versions and the defaults between the two versions did not change. Second, the chip sets are noted in the table and have real relevance on the floating point performance of that machine and how the Intel compiler generates chip specific instructions. As is shown for the FP Acceleration Item in Table 1, different FP acceleration hardware is used that can directly affect execution times, and if compiled improperly, can throw runtime exceptions that terminate the execution or can just slow down the execution.

To make the compiler generate the instructions necessary to fully utilize the FP hardware, a delicate issue must be examined: the Intel compiler generates faster code on Intel chip sets using the compiler defaults as is shown in Table 2. There are many compiler options available to make the compiler generate the proper FP instructions. The first problem is that for this major version of the compiler, the Windows and Linux options are not the same. For the Linux version of the compiler, the only option that works across AMD and Intel chip sets is the `-mtune` option although as shown in Table 2, the `-mtune` option is not needed for Intel chip sets. The other code generation options throw illegal instruction runtime exceptions on the AMD chip set and terminate the execution. The values used for `-mtune` are listed in Table 1. The use of `corei7-avx` is counterintuitive since these are Xeon or Opteron chip sets and not Core-i7 chip sets, even though all have the `AVX` vector processor. The use of `slm` for the Xeon based Linux workstation chip set is taken verbatim from the Intel Compiler documentation. The Windows version of the compiler does not have the `-mtune` option. Since the Windows machines are Intel chip sets and they are single node executions, the `/QxHost` option, or compile to the FP parameters that exist on the host, was used to get the proper FP acceleration.

Using these compiler options allows the AMD and Intel chip sets to generate the appropriate and consistent FP instructions for the chip sets. This then allows for a fair comparison amongst different manufactures and versions of chip sets.

4.3 Execution Issues

How the Coarray version of the code is compiled and executed is important in understanding the results. When the code is executed under Linux or Windows, the number of threads started is designated by either the compiler command with `-coarray-num-images=$value` (Linux) or `/Qcoarray-num-images=%value%` (Windows) or by setting this environment variable `setenv FOR_COARRAY_NUM_IMAGES=$value` (cshell in Linux) or `set FOR_COARRAY_NUM_IMAGES=%value%` (Windows). Since the default values of the looping parameters in Fortran Snippet 2 were used, the `value` in all of these cases is set at 121 (or Number_Materials(1) times Number_Materials(2), which is 11 times 11 or 121). When the Coarray code is executed, 121 threads are

created. The HZE Cluster machine uses SGE to dispatch batch jobs. The K Cluster uses PBS to dispatch batch jobs. The other machines execute the command from the command line (`cshell` in Linux and `cmd` in Windows).

The Fortran Snippets above implicitly assume each thread executes on one core. However, that may not always be the case. The Workstations (Linux, Desktop, and Laptop in this paper) all have many fewer cores than the default case needs. If there are not enough cores available, then there are two strategies that can be implemented. The first is to just allow all the threads to execute on the cores available. However, this can cause problems by spending more time thread context switching than computing and thrashing[h]. The second is to rewrite the `Codimension` indexes and loops to only execute one thread per core. However, this slightly complicates the resultant coding by introducing another loop over the number of cores and adjusting the other loops and `Codimension` indexes to compensate. For the purposes of this paper, less complex coding was used and the effects of more threads per core, and the context switching involved, was investigated.

The K Cluster is configured to assign whole nodes, with increments of 16 cores per node, to a job. Therefore, if 64 cores are wanted for a job, the job asks for four nodes at 16 cores per node. If the job asks for a number of cores not divisible by 16, then more cores are given to the next 16 increment and the extra cores on the last node are idle. No node can execute more threads than 16 and all threads belong to the same job.

The HZE Cluster is set to assign cores to jobs. Each node has the capability to run 32 threads with one thread per core. Therefore, if 64 cores are asked for in a job needing 121 threads, then SGE could give the job one core on 64 nodes or any other combination of cores and nodes. A problem occurs with this method when two (or more) threads are assigned to one core. Now, more than 32 threads are running per node and then the other job's threads are impacted and slowed along with the Coarray job's threads. Care must be taken in order to not impact other jobs with this method of assigning threads to cores.

4.4 Raw Timing Data

Because of the three pass execution of the Coarray code for the interpolation version of HZETRN, it is not straightforward to compare different simulations on different machines or even the same machine with different number of threads per core (as explained in Section 4.3). However, the raw timing simulations will be discussed here and then normalized, compared, and discussed in Section 4.5.

HZE Cluster: Table 3 shows the execution times for the HZE Cluster with 121 cores at one thread per core. The Coarray calculation has a 4.0 times speed-up over the single threaded calculation. For the single threaded version, the variance of the calculation was very small in comparison to the value, assuming that the means form a normal distribution. This implies that the variation between simulations was small and the single threaded calculation is stable. The Coarray calculation has a much larger variance but was still small in comparison to its value. The Coarray calculation was still stable, but was also a function of the traffic from itself and others on the 10GigE Network Infrastructure; however, the execution of the reported instance in Table 3 ran on four nodes with no other jobs running on those nodes. The variance for the input and output for both the Coarray and single threaded executions shows that the time to perform input and output was dependent on the traffic from itself and others on the 1GigE NFS mounted disk array.

K Cluster: Table 4 shows the execution times for the K cluster with 121 cores at one thread per core. The Coarray calculation has a 32.5 times speed-up over the single threaded calculation. For the single thread, the variance was very small and the calculation was stable. The same can be said for the Coarray variance; however, the Infiniband Network Infrastructure was not being shared with other jobs between nodes. The variances on the input and output were small; however, while the file system is a parallel file system, it's speed from run to run was dependent on the load of the file system from other jobs. Intel's HT, or two threads concurrent per core, was turned off.

---

[h]Thrashing is where the virtual memory system of the OS is spending more time paging than computing. Paging is where the OS brings a memory page into or out of the CPUs cache. See `https://en.wikipedia.org/wiki/Thrashing_%28computer_science%29` for more details.

Linux Workstation: Table 5 shows the execution times for the Linux workstation with 3.78125 threads per core. The Coarray calculation has a 3.4 times speed-up over the single threaded calculation. For the single thread, the variance was small in comparison to its value. This means that the calculation was stable. The same can be said for the Coarray variance. The variances for the input and output were also small. It cannot be guaranteed that these were the only executing user processes on the machine, but the machine appeared to be empty during the execution of these tests. Intel's HT, or two threads concurrent per core, was turned off.

Desktop Workstation: Table 6 shows the execution times for the Desktop workstation with 10.083 threads per core. The Coarray calculation has a 3.8 times speed-up over the single threaded calculation. For the single thread, the variance was small in comparison to its value. This means that the calculation was stable. The same can be said for the Coarray variance. The variances for the input and output were also small. It can be guaranteed that these were the only executing user processes on the machine. Intel's HT, or two threads concurrent per core, was turned off.

Laptop Workstation: Table 7 shows the execution times for the Laptop workstation with 30.25 threads per core. The Coarray calculation shows a 1.2 times slow down over the single threaded calculation (see Section 4.5 for possible reasons for the slowdown). The variances for the single thread and the Coarray were small in comparison to their values and show that the calculation is stable. The variances for the input and output were also small. It can be guaranteed that these were the only executing user processes on the machine. Intel's HT, or two threads concurrent per core, was turned off.

Intel Hyperthreading (HT): To better understand Intel's HT in this simulation, it was turned on for the Desktop and Laptop and the simulations run again. Therefore, for the Desktop workstation there were 5.04167 threads per apparent core, and for the Laptop there were 15.125 threads per apparent core. Tables 8 and 9 show the results for these simulations. If Intel's HT really gives two times more core capability, then the speed-ups should also be doubled. When Intel's HT was turned off for the Desktop run, the speed-up was 3.8; therefore, when Intel's HT was turned on, the expected speed-up should be 7.6. With Intel's HT turned on for the Desktop, the speed-up was 4.8 and only a 1.3 times core increase instead of two. The Intel's HT differences for the Laptop run was a speed-up of 0.85, or a slow down; therefore, when Intel's HT was turned on, the expected speed-up should be 1.7. With Intel's HT turned on, the speed-up was 0.92 and only a 1.1 times core increase instead of two. Intel's HT does not increase any speed-ups by a factor of two. While a small speed-up was seen, in more intensive computational cases, Intel's HT could slow down the code execution and therefore should be turned off.

K Cluster with higher thread per core count: Tables 10 through 11 show the results for reduced core counts of 64 and 16 cores for the K Cluster still executing 121 threads. A near-linear relationship exists between the core count and the execution time as shown in Figure 4, as would be expected.

HZE Cluster with higher thread per core count: Tables 12 through 13 show the results for reduced core counts of 64 and 32 cores for the HZE Cluster still executing 121 threads. All threads ran on nodes that were running no other job threads. A linear relationship is not shown in Figure 5. More research is required to determine why this relationship is apparent for this hardware and software set up. There are some things that could qualitatively explain the non-linearity like cache utilization (see Table 1 for a list of possible issues pertaining to hardware that must be addressed); however, to determine the exact cause would take a detailed analysis of the interaction between the code and the hardware and was not performed for this work.

4.5 Normalized Data, Comparison, and Discussion

To compare the Coarray data for each of the machine and core combinations described above, it is important to determine machine differences. The single thread simulations enable a normalization factor to be determined based on the machine and its particular equipment. Table 14 shows the normalization factors for all the times and machines listed in Tables 3 through 13 for the single threaded simulations. While the input

and output speed-ups are listed, their values are not relevant to any normalization since the actual amount of time spent in those tasks was small and is also a function of what the machine was doing when the code was executing. For example, the Laptop, while running the timed code, was also performing numerous tasks that support the Windows operating system. Disk I/O is part of those tasks and would lengthen the I/O time of the timed code. This is shown in the Linux machine's Input and Output times. While the Linux machine is the slowest computational machine, its input I/O is the fastest and its output I/O is about average. These I/O times are closely tied to how the disk drives are attached to the computer, the kind of disk drive hardware and firmware, and the non-code related disk I/O being performed. The one machine where this was not a problem is the K Cluster where the disk I/O is performed on a disk farm that is executing a parallel I/O system (Lustre) and can handle multiple users without single user degradation. Therefore, the disk I/O speed-ups are similar for Input and Output and show a four to five times speed-up. Of course, the fastest calculation is performed by the latest chip set which is on the Laptop workstation and shows an almost three times speed-up over the oldest chip set on the Linux workstation.

Table 15 shows each machine and core combination Coarray speed-ups normalized to the slowest chip set (the Linux workstation). Therefore, these data compare the Coarray simulations to one another on an equal per instruction execution basis. It is clear that a faster networking interface gives faster results. The K Cluster results with the Infiniband networking interface can be 38 times faster than the slowest machine (the Laptop without HT). The 10GigE networking interface used on the HZE Cluster gives, at most, a 5-10 times speed-up over the slowest machine.

Figure 6 shows the normalized execution times graphically and the machine with the largest slope is the K Cluster. Even though the K Cluster does not have the fastest chip set, it has the fastest network and disk I/O. It is unclear from this study how far the slope would continue before the Coarray overhead, context switching, and the other problems with more than one thread per core would start to overtake the computational gains. Because of the limits on number of cores per job and other administrative limits, a study like this would have to be planned closely with the administrators and users and therefore was not conducted. It is clear that context switching becomes a problem for the lower core count machines. Even though the Laptop Workstation has the fastest chip set, the Coarray code is actually slower than the single threaded version.

Table 16 shows in a single table the calculation program section speed-up of the Coarray code over the single threaded code reported in Tables 3 through 13 with Amdahl's law theoretical speed-ups. The K Cluster with one thread per core is the fastest with a 32.5 times speed-up over the single threaded version. Amdahl's Law theoretical speed-up of 74.5 times is not achieved because of the nature of the control algorithm with parallel sections of the code interrupted by serial sections of the code (see Figure 3). The 10GigE networking infrastructure does not compare to the Infiniband networking infrastructure as the HZE Cluster is at most ten times faster than the single threaded version. The Laptop workstation, while having the fastest chip set, because of its limited number of cores, is actually slower than the single threaded version on the same machine. It is obvious that all of the elements of the machine are important for the speed-up of the Coarray code over the single threaded code; however, a fast core which is executing optimized code, a large number of cores, and a fast networking infrastructure are the top contributors to any speed-up. From the HZE speed-ups at one and two threads per core, the microarchitecture of the chip set (cache, data transfer speeds, etc.) may also be important as shown in Figure 5; however, this analysis has not been performed.

5   Conclusions

The initial attempt at creating a multithreaded program using Coarrays for the control algorithm in the HZETRN program that generates an interpolation database in a space radiation environment simulation generates a speed-up of 32.5 times over the single threaded version on the K Cluster with an Amdahl's Law based theoretical speed-up of 74.5 times. If the simulation is run with fewer cores than threads, then the Coarray execution time can become longer than the single thread version as was shown on the Laptop workstation. This work shows that the number of threads per core is the important parameter for execution time. Networking infrastructure and code optimized to the chip set also contribute heavily.

For the scenario with seven materials and 21 points per material, a total of 85,766,121 cores would be needed if the algorithm developed here is used for a single thread per core. If the K Cluster had this many

cores, then instead of taking 225.1 years to execute if the default database took 30 minutes to generate, then it would take 6.9 years to execute. If the database only took ten seconds to generate (see Table 4), then the execution would go from 7.5 years to 14.1 days. In single thread mode, the one minute generation time seemed improbable; however, if this large K Cluster is used, then a one minute database generate time is very plausible. Therefore, with Coarrays and an appropriate sized cluster, larger problems can be solved or more problems can be solved in the same time. However, real core counts are important. Even the largest cluster in the Top 500 list today only has 10,649,600 cores but it is in China and not accessible to the authors. The NASA Pleiades cluster has 185,344 Xeon cores and is not a hybrid cluster. To run this scenario on Pleiades, each core would have to run 462 or 463 threads. Context switching would be a problem and the loop structure would need to be rewritten to handle multiple threads per core.

Because of the parameters used and the computers involved, 121 threads was the maximum number of threads programmed for the examples in this work. Two more Do Loops in Fortran Snippets 11 through 13 (indexes in the `psi` array) could be made into Coarrays; however, this would then intrude into the manner in which the physics and transport algorithms are programmed and was one of the restrictions imposed on this effort. There are loops and other algorithms within the physics that would benefit from Coarrays or other threading techniques. HZETRN is rich with chances to take advantage of parallel programming. There are many other processes and algorithms within the OLTARIS system, and not just HZETRN, that could be sped up with Coarrays.

Coarrays are the easiest and arguably most productive method for engineering and scientific programmers to enter into the world of parallel programming in comparison to the MPI programming paradigm. This paper has shown that with a simple tool and a Fortran 77/95 programmer's skill set, multithreaded code using Coarrays is not hard to generate with the current programmers that exist within the project. The resultant code is just as reliable and as easy to maintain as the Fortran 77/95 code that exists today. Therefore, an effort should be made by NASA to make HZETRN and the other codes within the OLTARIS system encompass Coarrays as the starting foray into scalable parallel processing. This paper can also be used by other engineering and science groups to investigate Coarrays in their analyses.

References

[1] Link: Moore's Law

[2] Department of Energy, Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee, "The opportunities and Challenges of Exascale Computing," U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Fall 2010.

[3] Link: HPC Wire: Paul Messina Shares Deep Dive Into US Exascale Roadmap, June, 2016.

[4] Wilson, J.W., Badavi, F.F., Cucinotta, F.A., Shinn, J.L., Badhwar, G.D., Silberberg, R., Tsao, C.H., Townsend, L.W., Tripathi, R.K.; "HZETRN: Description of a free-space ion and nucleon transport and shielding computer program," NASA Technical Paper 3495, May 1995.

[5] Wilson, J.W., Tripathi, R.K., Mertens, C.J., Blattnig, S.R., Clowdsley, M.S., Cucinotta, F.A., Tweed, J., Heinbockel, J.H., Walker, S.A., Nealy, J.E.; "Verification and Validation: High Charge and Energy (HZE) Transport Codes and Future Development," NASA Technical Paper 2005-213784, July 2005.

[6] Slaba, T.C., Blattnig, S.R., Badavi, F.F.; "Faster and More Accurate Transport Procedures for HZETRN," NASA Technical Paper 2010-216213, March 2010.

[7] Link: The Fortran Standards Committee J3.

[8] Rogers, D.P., "Improvements in multiprocessor design," Assoc. Computing Machinery, Vol. 13 (3), pp. 225-231, June 1985.

[9] Singleterry Jr., R.C., Blattnig, S.R., Clowdsley, M.S., Qualls, G.D., Sandridge, C.A., Simonsen, L.C., Norbury, J.W., Slaba, T.C., Walker, S.A., Badavi, F.F., Spangler, J.L., Aumann, A.R., Zapp, E.N., Rutledge, R.D., Lee, K.T., Norman, R.B., "OLTARIS: On-Line Tool for the Assessment of Radiation In Space", NASA Technical Paper 2010-216722, July 2010.

[10] Cucinotta, F.A., Premkumar, B.S., Xiaodong, H., Myung-Hee, Y.K., Cleghorn, T.F., Wilson, J.W., Tripathi, R.K, Zeitlin, C.J., "Physics of the Isotopic Dependence of Galactic Cosmic Ray Fluence Behind Shielding," NASA Technical paper 2003-210792, February 2003.

[11] Slaba, T.C., Blattnig, S.R., "Coupled Neutron Transport for HZETRN," NASA Technical Paper 2009-215941, October 2009.

[12] Singleterry, R.C., Wilson, J.W., Shinn, J.L., Tripathi, R.K., Thibeault, S.A., Noor, A.K., Cucinotta, F.A., Badavi, F.F., Chang, C.K., Qualls, G.D., Clowdsley, M.S., Kim, M.H., Heinbockel, J.H., Norbury, J., Blattning, S.R., Miller, J., Zeitlin, C., Heilbronn, L.H., "Creation and utilization of a World Wide Web based space radiation effects code: SIREST," Physica medica, Vol. 17 Sup. 1, pp. 90-93, 2001.

Acknowledgment
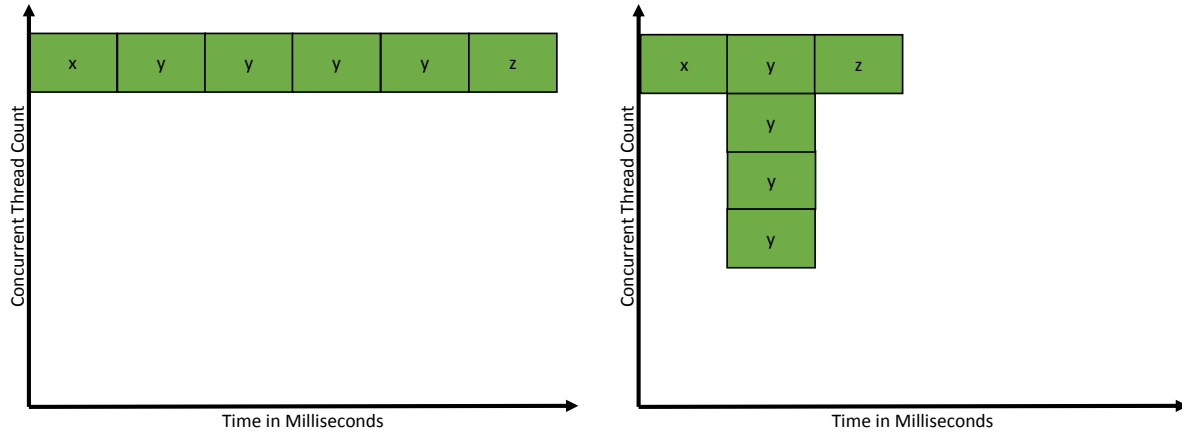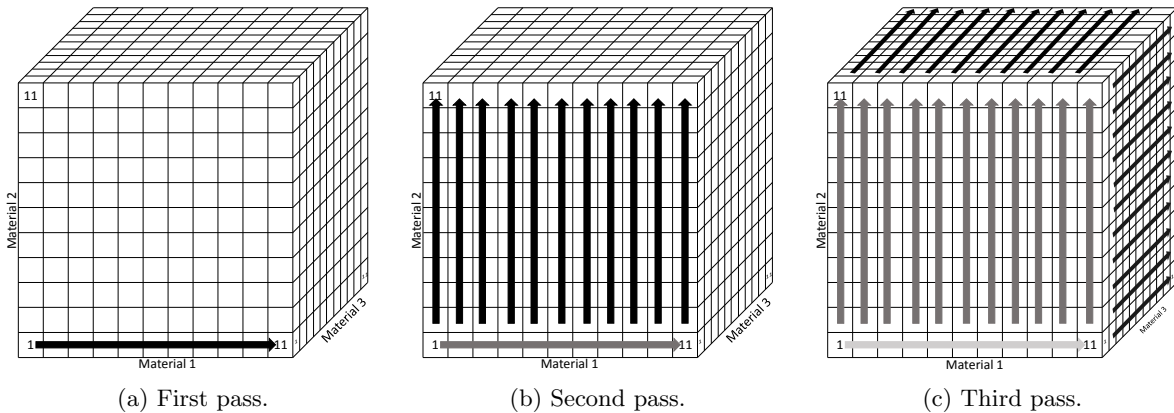
(a) Serial execution of the code.   (b) Parallel execution of the code.

Figure 1: Time comparisons of a serial executable and a parallel executable where x is the time for input, y is the time per loop, and z is the time for output.

(a) First pass.　　　　　　　　(b) Second pass.　　　　　　　　(c) Third pass.

Figure 2: HZETRN marching algorithm that produces the database needed by the interpolation method. Each arrow can represent a serial execution order or a thread with the same shade of gray representing concurrent threads.
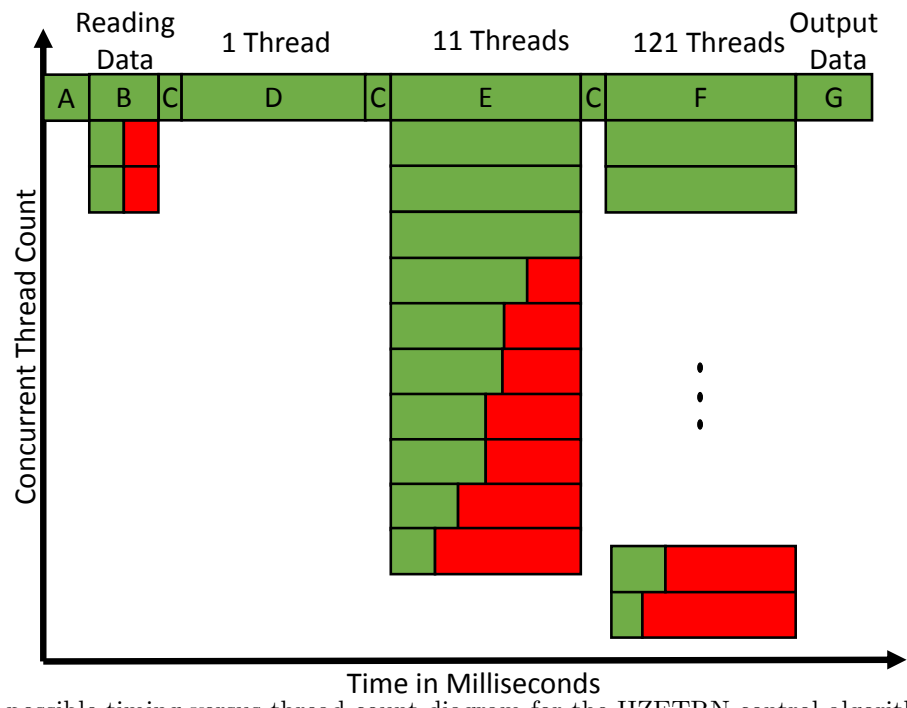
Figure 3: A possible timing versus thread count diagram for the HZETRN control algorithm. Block A represents the initial setup. Block B represents the reading and sharing of the data where the green color represents computation and the red color represents idle time. Block C represents initialization of the `psi` array for each pass. Block D represents the first pass which is serial. Block E represents the second pass. Block F represents the third pass. Finally, Block G represents the output of the results.

Figure 4: K Cluster core versus Calculation times.

Figure 5: HZE Cluster core versus Calculation times.

Figure 6: Normalized Calculation times for all machines.

Table 1: Machine hardware, software, and compiler parameters.

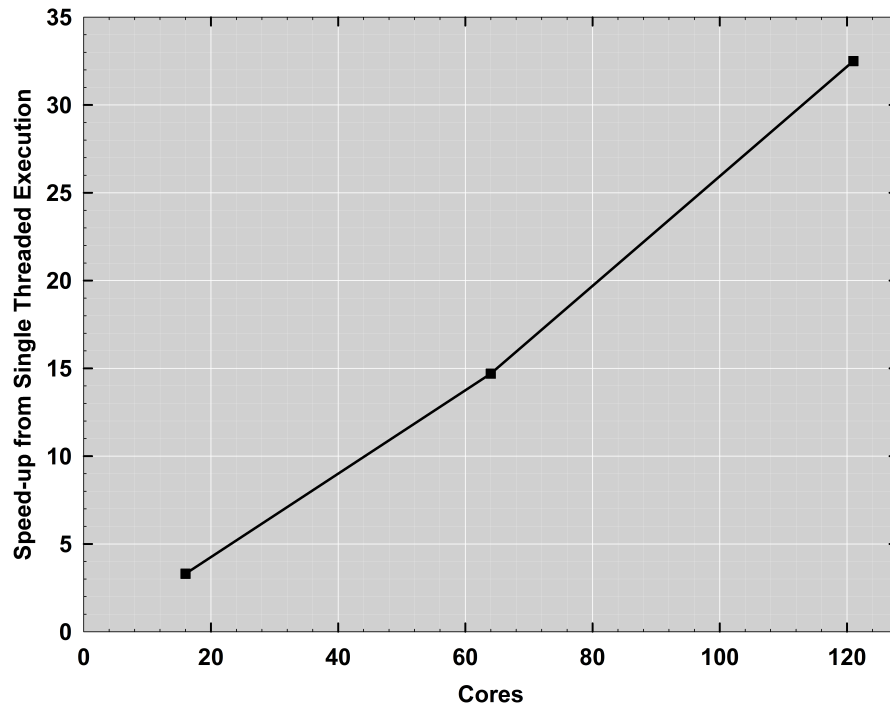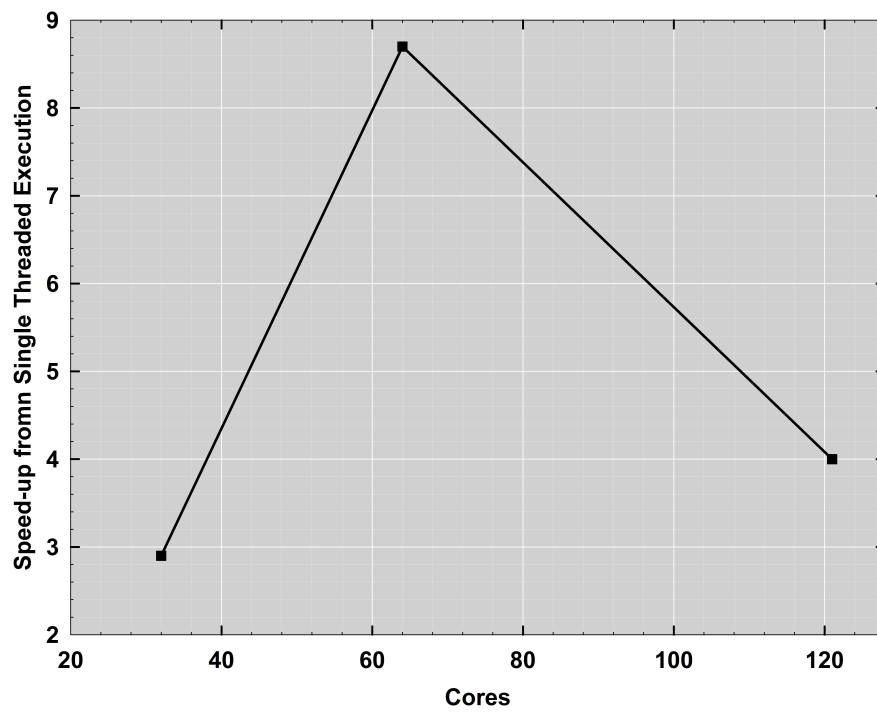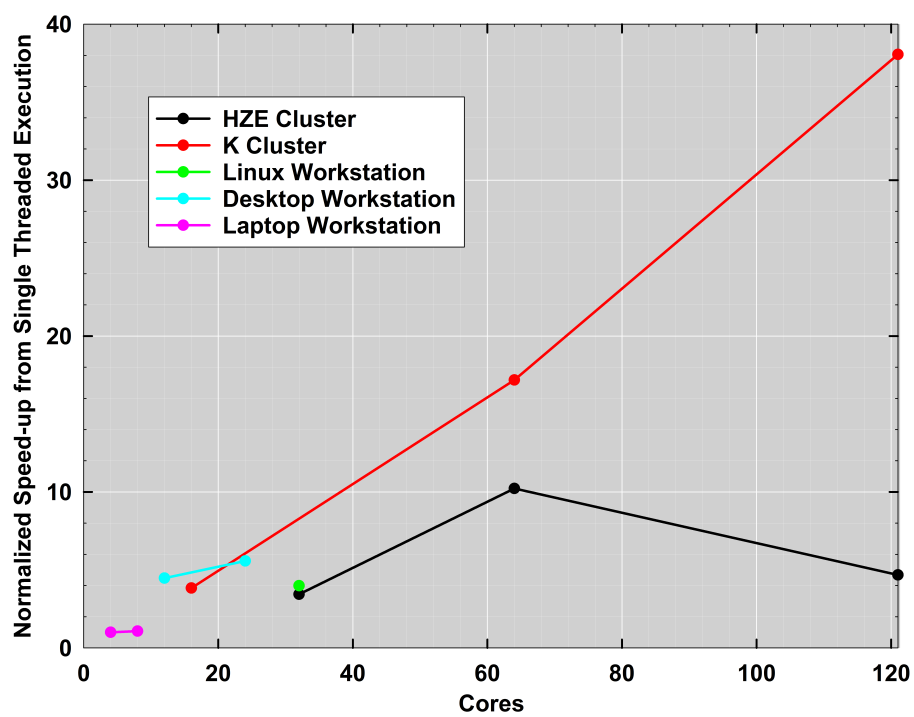| Item	HZE Cluster	K Cluster	Linux	Desktop	Laptop
Machine Hardware					
Chip Set	AMD Opteron 6220	Intel Xeon E5-2670	Intel Xeon E7-4830	Intel Xeon E5-2640	Intel Core-i7 4900MQ
Chip Set Code Name	Bulldozer Iterlagos	Sandy Bridge-EP	Westmere-EX	Ivy Bridge	Haswell
CPUs per Node	4	2	4	2	1
Cores per CPU	8	8	8	6	4
Base Freq (GHz)	3.0	2.6	2.13	2.5	2.8
Turbo Freq (GHz)	3.6	3.3	2.4	3.0	3.8
Cache (MB)	16	20	24	15	8
Feature Size (nm)	32	32	32	32	22
QPI or HT (GT/s)	6.4	8.0	6.4	7.2	5.6
DDR3 Mem (GT/s)	1.6	1.6	1.066	1.333	1.6
FP Acceleration	AVX	AVX	SSE 4.2	AVX	AVX 2.0
Disk Infrastructure	SATA	SAS	SAS	SAS/SSD	SATA/SSD
Network Infrastructure	10GigE	4X FDR Infiniband (56 Gbps)	local	local	local
Machine Software					
Operating System	CentOS 6.5	CentOS 6.6	OpenSUSE 13.1	Windows 7	Windows 7
File System	1GigE NFS Disk Array	4X FDR Infiniband Lustre 2.5	1GigE NFS Disk Array	local NTFS	local NTFS
Queuing Software	SGE	PBS	Command Line	Command Line	Command Line
Compiler Options					
Compiler Version	16.0.0.109	16.0.0.109	16.0.0.109	16.0.1.146	16.0.1.146
-mtune=value	corei7-avx	corei7-avx	slm	−	−
/Qx%value%	−	−	−	Host	Host

Table 2: Calculation average execution times for the single threaded non-accelerated simulations on the HZE and K Clusters compared to the accelerated simulations.

Machine	Accelerated (seconds)	non-Accelerated (seconds)	Percent Slowdown
HZE Cluster	277.90574	331.46362	16.1580
K Cluster	207.97232	207.95678	−0.0075

Table 3: Average execution times and standard deviations for the HZE Cluster. Non-Coarray is a single thread and Coarray is 121 threads on 121 cores (4 nodes).

| Program Section | Non-Coarray | | Coarray | | Coarray Speed-up |
	Time (seconds)	Standard Deviation	Time (seconds)	Standard Deviation	
Input	0.0439200	0.0753504	0.7382200	0.2296453	0.0594945
Calculation	277.9057400	0.3669780	69.4306800	7.6773789	4.0026360
Output	5.2014100	0.2455158	5.6235200	0.4993794	0.9249385
Total	283.1510700	0.4871622	75.7924200	7.9775391	3.7358758

Table 4: Average execution times and standard deviations for the K Cluster. Non-Coarray is a single thread and Coarray is 121 threads on 121 cores (8 nodes).

| Program Section | Non-Coarray | | Coarray | | Coarray Speed-up |
	Time (seconds)	Standard Deviation	Time (seconds)	Standard Deviation	
Input	0.0076800	0.0003868	0.5254800	0.0094424	0.0146152
Calculation	207.9723200	0.1688097	6.3996300	0.0509266	32.4975538
Output	2.8202700	0.1317372	2.8363200	0.0877392	0.9943413
Total	210.8002800	0.1795862	9.7614300	0.1180925	21.5952253

Table 5: Average execution times and standard deviations for the Linux workstation. Non-Coarray is a single thread and Coarray is 121 threads on 32 cores.

| Program Section | Non-Coarray | | Coarray | | Coarray Speed-up |
	Time (seconds)	Standard Deviation	Time (seconds)	Standard Deviation	
Input	0.0027800	0.0000600	9.4729500	0.1190157	0.0002935
Calculation	303.6618400	4.8841879	89.1517700	1.0243597	3.4061224
Output	7.5039400	0.2435973	11.3279600	0.2960703	0.6624264
Total	311.1685600	4.9242029	109.9526800	1.0772603	2.8300225

Table 6: Average execution times and standard deviations for the Desktop workstation. Non-Coarray is a single thread and Coarray is 121 threads on 12 cores.

| Program Section | Non-Coarray | | Coarray | | Coarray Speed-up |
	Time (seconds)	Standard Deviation	Time (seconds)	Standard Deviation	
Input	0.0069000	0.0079931	2.3275000	0.0925724	0.0029646
Calculation	273.5768000	2.3148607	71.6447000	7.3960588	3.8185211
Output	14.3850000	2.3149841	15.0668000	2.4217578	0.9547482
Total	287.9687000	3.5948185	89.0390000	7.5220555	3.2341861

Table 7: Average execution times and standard deviations for the Laptop workstation. Non-Coarray is a single thread and Coarray is 121 threads on four cores.

Program Section	Non-Coarray		Coarray		Coarray Speed-up
	Time (seconds)	Standard Deviation	Time (seconds)	Standard Deviation	
Input	0.0067000	0.0098392	3.9608000	0.0388608	0.0016916
Calculation	103.5555000	0.0735816	121.4335000	3.2176690	0.8527754
Output	7.0956000	1.1046798	7.6294000	1.2062805	0.9300338
Total	110.6578000	1.0679085	133.0237000	3.5879657	0.8318653

Table 8: Average execution times and standard deviations for the Desktop workstation with Hyperthreading (2 threads per core). Non-Coarray is a single thread and Coarray is 121 threads on 24 apparent cores.

Program Section	Non-Coarray		Coarray		Coarray Speed-up
	Time (seconds)	Standard Deviation	Time (seconds)	Standard Deviation	
Input	0.0040000	0.0048990	1.8492000	0.0210609	0.0021631
Calculation	273.9357000	0.1278515	57.5424000	0.0945476	4.7605887
Output	13.2697000	2.1694012	16.2726000	2.2196714	0.8154628
Total	287.2094000	2.1456527	75.6642000	2.2029522	3.7958427

Table 9: Average execution times and standard deviations for the Laptop workstation with Hyperthreading (2 threads per core). Non-Coarray is a single thread and Coarray is 121 threads on eight apparent cores.

Program Section	Non-Coarray		Coarray		Coarray Speed-up
	Time (seconds)	Standard Deviation	Time (seconds)	Standard Deviation	
Input	0.0022000	0.0044000	3.3166000	0.0760068	0.0006633
Calculation	103.8067000	0.1238023	112.6399000	0.6198891	0.9215802
Output	7.4051000	1.0881400	9.7905000	1.1631966	0.7563557
Total	111.2140000	1.0375296	125.7470000	1.3398445	0.8844267

Table 10: Average execution times and standard deviations for the K Cluster. Non-Coarray is a single thread and Coarray is 121 threads on 64 cores (4 nodes).

Program Section	Non-Coarray		Coarray		Coarray Speed-up
	Time (seconds)	Standard Deviation	Time (seconds)	Standard Deviation	
Input	0.0098200	0.0016696	0.8496200	0.0135030	0.0115581
Calculation	207.9615600	0.1358343	14.1771500	0.1303325	14.6687846
Output	2.9539100	0.1333285	3.0344500	0.1018819	0.9734581
Total	210.9252900	0.2053382	18.0612300	0.1673638	11.6783458

Table 11: Average execution times and standard deviations for the K Cluster. Non-Coarray is a single thread and Coarray is 121 threads on 16 cores (1 node).

Program Section	Non-Coarray		Coarray		Coarray Speed-up
	Time (seconds)	Standard Deviation	Time (seconds)	Standard Deviation	
Input	0.0070400	0.0006264	2.6200900	0.0673951	0.0026869
Calculation	207.9733000	0.1291744	63.4550100	0.2197354	3.2774922
Output	2.9104900	0.1052632	8.8242800	0.0895348	0.3298275
Total	210.8908300	0.1918424	74.8993800	0.2408672	2.8156552

Table 12: Average execution times and standard deviations for the HZE Cluster. Non-Coarray is a single thread and Coarray is 121 threads on 64 cores (2 nodes).

Program Section	Non-Coarray		Coarray		Coarray Speed-up
	Time (seconds)	Standard Deviation	Time (seconds)	Standard Deviation	
Input	0.0420300	0.0737965	0.4927000	0.0120503	0.0853055
Calculation	275.4182100	0.0576928	31.7623000	0.2378710	8.6712300
Output	5.3531000	0.0500943	5.5411300	0.0599317	0.9660665
Total	280.8133400	0.1031759	37.7961300	0.2319002	7.4296850

Table 13: Average execution times and standard deviations for the HZE Cluster. Non-Coarray is a single thread and Coarray is 121 threads on 32 cores (1 node).

Program Section	Non-Coarray		Coarray		Coarray Speed-up
	Time (seconds)	Standard Deviation	Time (seconds)	Standard Deviation	
Input	0.0327100	0.0430953	0.4274000	0.0108316	0.0765325
Calculation	277.4659400	0.2433768	94.3851500	0.5422694	2.9397203
Output	5.0739500	0.0425017	4.9953700	0.0267222	1.0157306
Total	282.5726000	0.2241162	99.8079200	0.5623280	2.8311641

Table 14: Normalization factor for single threaded simulations on different machines.

Name	Program Section	Normalization
HZE Cluster	Input	1.
	Calculation	1.09552
	Output	2.67414
	Total	1.10191
K Cluster	Input	4.15587
	Calculation	1.46013
	Output	4.77647
	Total	1.47563
Linux Workstation	Input	12.22841
	Calculation	1.
	Output	1.84268
	Total	1.
Desktop Workstation	Input	6.23761
	Calculation	1.10924
	Output	1.
	Total	1.08199
Laptop Workstation	Input	7.63933
	Calculation	2.92881
	Output	1.90713
	Total	2.80494

Table 15: Speed-up for the Coarray code on different machines with different core counts per run normalized to the slowest chip set.

Name	Number of Nodes	Number of Cores	Program Section	Speed-up
HZE Cluster	4	121	Calculation	4.67582
			Total	4.46767
	2	64	Calculation	10.22109
			Total	8.95899
	1	32	Calculation	3.43958
			Total	3.39267
K Cluster	8	121	Calculation	38.06123
			Total	25.90375
	4	64	Calculation	17.18101
			Total	14.00002
	1	16	Calculation	3.83859
			Total	3.37596
Linux Workstation	1	32	Calculation	3.98932
			Total	3.39349
Desktop Workstation	1	24 (HT)	Calculation	5.57205
			Total	4.55763
	1	12	Calculation	4.47527
			Total	3.87301
Laptop Workstation	1	8 (HT)	Calculation	1.07807
			Total	1.05787
	1	4	Calculation	1.
			Total	1.

Table 16: Speed-up of the Coarray Calculation section over the single threaded Calculation section with Amdahl's Law Theoretical Speed-up, TSU.

Name	Number of Nodes	Number of Cores	Speed-up	TSU
HZE Cluster	4	121	4.00264	
	2	64	8.67123	53.788
	1	32	2.93972	
K Cluster	8	121	32.49755	
	4	64	14.66878	72.665
	1	16	3.27749	
Linux Workstation	1	32	3.40612	41.452
Desktop Workstation	1	24 (HT)	4.76059	
	1	12	3.81852	20.823
Laptop Workstation	1	8 (HT)	0.92158	
	1	4	0.85278	15.297

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE (DD-MM-YYYY)	2. REPORT TYPE	3. DATES COVERED (From - To)
01-03-2018	Technical Publication	

4. TITLE AND SUBTITLE

A Single Thread to Fortran Coarray Transition Process for the Space Radiation Code HZETRN

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Singleterry, Robert C., Jr.; Ranjan, Desh; Zubair, Mohammad

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER
651549.02.07.01

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

NASA Langley Research Center
Hampton, Virginia 23681-2199

8. PERFORMING ORGANIZATION REPORT NUMBER

L-20908

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration Washington, DC 20546-0001

10. SPONSOR/MONITOR'S ACRONYM(S)
NASA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
NASA-TP-2018-219811

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified-
Subject Category 93
Availability: NASA STI Program (757) 864-9658

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Exa-scale computing is the direction by industry and government are going to generate solutions to problems they deem necessary. Computing hardware is being developed to achieve the transition from Peta-scale to Exa-scale with more CPUs that have more cores per CPU and more accelerators (GPGPUs and MICs) per node. To fully utilize the hardware available now and in the future, algorithms must become multi-threaded. There are a few methods to generate multi-threaded software such as MPI and OpenMP/OpenACC. This paper concentrates on using Coarray Fortran to convert the Fortran 95 based HZETRN code's control algorithm from a single threaded code to a multithreaded code. The resultant Coarray code was 32.5 times faster (with a theoretical speed-up of 74.5 times) than the single threaded version on the hardware tested, as reliable as the Fortran 95 version, and, as it uses native Fortran, was as maintainable as the Fortran 95 version. The Coarray code can be maintained by the same project engineers and scientists who created the original single threaded code. This transition process can be utilized on a C language based code with a compiler that has the UPC extensions to C.

15. SUBJECT TERMS

Co-Arrays; Modern Fortran; Space Radiation

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			STI Help Desk(email help@sti.nasa.gov)
U	U	U	UU	37	19b. TELEPHONE NUMBER (Include area code) (757) 864-9658

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18