LA-UR-97-920

CONF-971005--11

Title: A PORTABLE, PARALLEL, OBJECT-ORIENTED MONTE CARLO NEUTRON TRANSPORT CODE IN C++

Author(s): Stephen R. Lee
Julian C. Cummings
Steven D. Nolen

Submitted to: Joint International Conference on Mathematical Methods & Superconducting for Nuclear Applications, Saratoga Springs, NY, October 5-10, 1997

Los Alamos
NATIONAL LABORATORY

A PORTABLE, PARALLEL, OBJECT-ORIENTED MONTE CARLO NEUTRON TRANSPORT CODE IN C++

Stephen R. Lee
Transport Methods Group
Applied Theoretical and Computational Physics Division
Los Alamos National Laboratory
MS B226
Los Alamos, New Mexico USA 87545
srlee@lanl.gov

Julian C. Cummings
Advanced Computing Laboratory
Communications, Information, and Computing Division
Los Alamos National Laboratory
MS B287
Los Alamos, New Mexico USA 87545
julianc@lanl.gov

Steven D. Nolen
Texas A&M University/Los Alamos National Laboratory
MS B226
Los Alamos, New Mexico USA 87545
sdnolen@lanl.gov

Abstract -- We have developed a multi-group Monte Carlo neutron transport code using C++ and the Parallel Object-Oriented Methods and Applications (POOMA) class library. This transport code, called MC++, currently computes k and α-eigenvalues and is portable to and runs parallel on a wide variety of platforms, including MPPs, clustered SMPs, and individual workstations. It contains appropriate classes and abstractions for particle transport and, through the use of POOMA, for portable parallelism. Current capabilities of MC++ are discussed, along with physics and performance results on a variety of hardware, including all Accelerated Strategic Computing Initiative (ASCI) hardware. Current parallel performance indicates the ability to compute α-eigenvalues in *seconds* to minutes rather than *hours* to days. Future plans and the implementation of a general transport physics framework are also discussed.

I. BRIEF DESCRIPTION OF THE PROBLEM

The development of the multi-group neutron transport code MC++ has centered on criticality problems. Therefore, MC++ currently computes static k and α-eigenvalues on computational meshes. The eigenvalue problems are formulated in the usual way[1,2]; k being calculated as the ratio of the number of neutrons in successive generations, and α being calculated as adding absorption to the problem, iterating until the computed value for k given an α is 1.0.

Rather than analytic surfaces for geometry descriptions (as used by MCNP[3]), MC++ gets its description of geometry through a cartesian computational mesh obtained from another simulation code[4]. This mesh fully describes the problem geometry in question, and each mesh element contains material information, density, and so on to fully specify the problem for transport. MC++ transports particles through this geometry in the usual way, with particles interacting with individual mesh cell boundaries rather than analytic surfaces.

II. COMPUTATIONAL METHODOLOGY

The k-eigenvalue algorithm is described first, since the α-eigenvalue algorithm uses the same mechanism. It should be noted that the k-eigenvalue algorithm is similar to that of MCNP.

A. The k-eigenvalue Algorithm

The k-eigenvalue calculation is started by guessing an initial spatial distribution of neutrons. In MC++, this initial guess is a simple scheme that places neutrons in mesh elements containing fissile material in a round-robin manner until all particles requested by the user are exhausted. This, along with an initial guess for the system *k* (also user supplied) beings the first iteration in MC++. This is called the first *generation* (or "cycle") of our neutron population. The *k*-eigenvalue is nothing more than the ratio of the number of neutrons in successive generations, with fission events being regarded as the birth event that separates these generations. Therefore, during the calculation, the mean number of neutrons produced per fission event are estimated and stored as source points for the *next* cycle. A single cycle, or generation, is therefore defined as the life of all neutrons in the problem from birth (by fission) to death (by escape or capture). Particles in the next cycle are started isotropically at the location at which the birth took place. A single cycle will therefore have a series of "transport loops" in which all particles in the current generation must be disposed of before continuing to the next cycle.

The neutrons are tracked through and interact with the mesh just as they would any geometry, undergoing collisions with isotopes that compose the material within each mesh cell, and undergoing boundary interactions with the mesh itself.

The user controls the nominal number of particles to track per cycle and the number of cycles during which to accumulate results. The user can also specify the number of initial cycles to skip to allow the neutron population to stabilize before accumulating the results. During each cycle, MC++ accumulates information about the likelihood and result of specific events into tallies, the purpose of which is to compute estimates of *k*. Three different such tallies, or estimators, for *k* are used by MC++. These estimators are the collision, absorption, and track-length estimators, and are the same as employed in the transport code MCNP. Fig. 1 shows the flow of the algorithm in MC++.

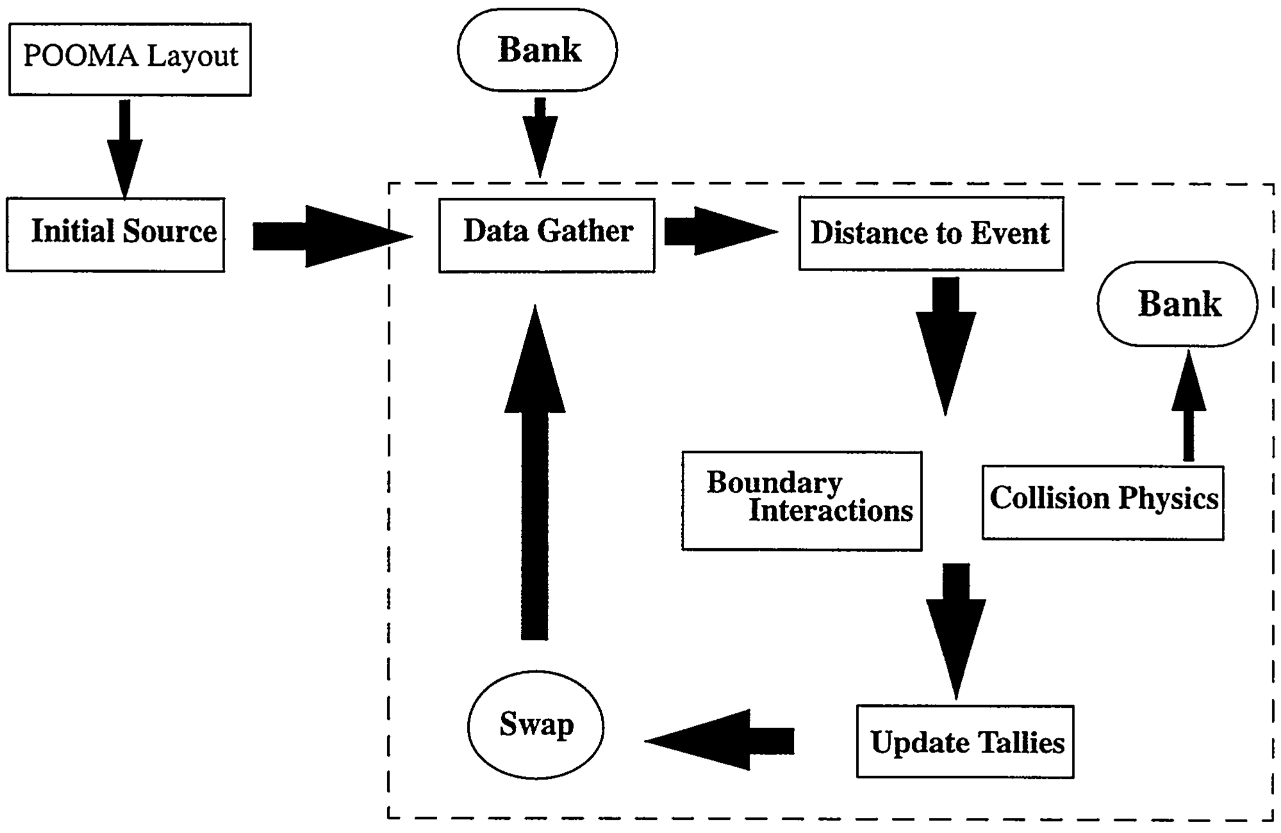

Fig. 1: The MC++ transport algorithm. The dashed box represents work that is done during a given generation of particles, i.e., the transport loops. Shaded objects represent work that is done simultaneously (i.e., in parallel) on all nodes. Note therefore that the unit of work in MC++ is not individual loops over particles , but particle *generations*, that is, the transport loops.

This figure shows the initialization of POOMA (explained later) and the source, and the beginning of the user-requested iteration loops, or individual neutron generations. Each neutron generation "lives" within the dashed box. That is, the previously discussed transport loops occur within the dashed box, with the created fission neutrons being stored on the "bank" (the bank is just a POOMA Particles object). Once a generation has completed (all particles have terminated by absorption or escape), the next iteration cycle begins with the bank particles as shown in the figure.

During the transport loops, k tallies are accumulated at different events as appropriate. For example, when a particle is within a cell containing fissile material, and that particle undergoes a collision interaction, fission neutrons are generated and stored to the bank. The particle then goes on to absorb, scatter, and whatever other interactions are in store for it until it leaves the problem through escape or capture. Once all of the particles in the current generation have been disposed of, global sums of all tallies in the problem are done, and a new iteration cycle begins.

Finally, the object labeled "Swap" in Fig. 1 refers to a POOMA member function that keeps particles and mesh data local on individual nodes. This is explained in more detail below.

B. The α-eigenvalue Algorithm

As mentioned earlier, the α calculation is done via a series of k calculations. Therefore, this calculation uses the same mechanism as depicted in Fig. 1. The algorithm hinges on selecting a value for α and converging the calculation, based on this α, to $k = 1.00$. To achieve this, MC++ computes k using a series of settle cycles, so named to allow the initial neutron population to stabilize, or settle, before beginning the α calculation. If after the settle cycles the system is supercritical (i.e., $k > 1.0$), the α calculation begins. An initial value for α is selected using

$$\alpha' = \alpha + (\alpha - 1) \times (\alpha + k/t_g) \qquad (1)$$

where t_g is the fission generation time defined as the time required to generate fission neutrons in a given generation, k is the current value for the k-eigenvalue (which can come from any of the afore mentioned tallies), and α is the current value of the α-eigenvalue. This is zero initially. Because the calculation relies on converging k to 1.0, some number of *inner* k-effective cycles are run per α-cycle in order to ensure that the computed k is itself reasonably converged. Typically this is 2-4 inner cycles per α-cycle, depending on the problem. During the α-cycles, computed values for $ln(\alpha)$ and $ln(k)$ are stored. After each set of inner k-cycles, a new value for α is chosen using Eq. 1 prior to the next α-cycle. Once the α-cycles are complete, a linear regression scheme is used to extrapolate the y-intercept of the line of best fit through these values. This intercept should be the point at which $ln(k)$=0, i.e., $k = 1.0$. This y-intercept is then exponentiated and is reported as the computed value for α. An estimate of the error is also computed[5]. This error is the calculated from the estimated standard deviation in the fit of the data-points to the line. A different method for estimating the error in α is under development.

It is worth noting that the α algorithm described here was added to MC++ and tested, serially and in parallel, on a variety of platforms, including ASCI Red and Blue options, in less than one day. This owes not just to the overall design of MC++, but to the power of object-oriented methods, the POOMA class library, and the C++ language itself. MC++ has been written to ensure portable parallelism, but also to ensure easy maintenance and addition of physics capabilities.

III. IMPLEMENTATION OF NEUTRON TRANSPORT USING POOMA AND OBJECT-ORIENTED METHODS

A. POOMA

POOMA stands for parallel object-oriented methods and applications, and is a class library intended to support a wide variety of parallel scientific computing applications[6]. If one examines code development in general and physics software in particular over the last several years, one often finds that the physics is imbedded in what was (at the time) the latest architecture, software environment, or parallel paradigm. POOMA was developed in an effort to retain key physics investments in a changing environment. Therefore, one of the key elements of POOMA is an *architecture abstraction.* That is, POOMA was developed to provide the same interface for an end user (a transport methods developer in this case) to different computational platforms. This allows the methods developer to focus on the computational physics algorithm and let POOMA handle the communications, domain decomposition, and other parallel-architecture concerns on different platforms. The programming paradigm in POOMA is the data-parallel model, which allows for a clean abstraction with some loss of generality to some physics problems that are not inherently data-parallel (such as Monte Carlo neutron transport).

A *framework* can be thought of as something that captures reusable software design and supports common capabilities within a specific problem domain[7]. It is an integrated and layered system, in which classes in higher layers utilize the classes from lower layers to build capability. POOMA is built from 5 such class layers and provides the user with data-parallel representations for a variety of data types. These data types, called global data types (GDT), include matrices, fields, and most importantly for MC++, *particles.* In an application code, the user typically calculates only with the GDT objects. Class member functions for GDTs in POOMA have been designed to seem similar to familiar procedural, data-parallel language syntax where possible. However, it does not prevent users from using inheritance and polymorphism to create new classes that map directly into problem domains of interest. This, combined with the parallel abstraction that POOMA provides, is what first interested us in the framework technology.

B. Particles Classes

In POOMA, particles are free to move about a given domain while interacting with a fixed grid. Naturally it is important to maintain particle locality within a given region on a local processor, otherwise the simulation will be dominated by interprocessor communication as each particle will potentially fetch field data across nodes. The particle classes provide a data-parallel expression syntax while handling the processor communication within the framework.

The particle classes consist of a double-precision particle field, or DPField, class, and a class that represents a *distribution* of particles (called Particles). DPFields represent physical attributes of a particle, such as its position, direction cosines, weight, and so on. A Particles object contains a set of DPFields that completely describe all of the particle attributes. While both of these objects point to the same data, through the class member functions each of these objects operate on the attributes of the particles in different ways. Generally speaking, the DPField class allows one to operate on individual attributes of the particle (there are many examples of this in MC++), whereas the Particles class operates across particle attributes.

The Particles class contains even higher-level member functions that allow scatter/gather operations, interpolation functions, and so on. Among these, a *swap* function is provided, which in combination with the problem domain-decomposition (also provided by POOMA), provides load-balancing capabilities as particles move in the simulation. This function is responsible for ensuring that particles are located on the same processor as local mesh data, and is invoked in MC++ after particle positions are updated. The current domain decomposition algorithm is a simple spatial decomposition scheme. Once particle positions are updated, the swap function is called as shown in Fig. 1.

Through the specifications of the DPFields, particles are constructed within POOMA. Using specifications of the problem domain, which in this case is a computational mesh provided by another code, the Particles object and data layout are constructed. Once complete, the problem is fully specified within the POOMA framework, and one can then take advantage of the functions POOMA offers.

There are many other details about POOMA that are not discussed here. For more information about POOMA, see Refs. 6, 8.

C. POOMA Implementation of Transport Physics

Naturally to use the POOMA framework, one must be able and willing to cast the problem into data-parallel form and to utilize POOMA class implementations to access data and perform the physics. For Monte Carlo transport, the alpha version of POOMA is a bit cumbersome to use. This is magnified by the nature of the problem being solved which is *not* inherently data-parallel. At any time in the simulation, individual neutrons in the distribution can undergo different interactions with their surroundings. This presents a problem, as it becomes difficult to write nice tidy data-parallel statements all the time, which is one of the nice features of POOMA.

Considering the transport problem to be solved, one is lead to a set of particle attributes that are required to simulate the criticality problem. As mentioned before, to create particles in POOMA all one needs to do is describe their attributes using DPFields. Once the DPFields have been specified, one then creates the Particles *object* (class instantiation) based on the layout of the problem. In our case, the layout is defined by the mesh information. The details on how this is done are important, but far to detailed to describe here[4,9,10,11]. The Particles object is created by specifying the layout, number of DPFields, and other information.

During tracking, particle attributes are retrieved from the Particles object in a straight-forward way. Occasionally during the transport algorithm, data-parallel updates of particle attributes are done, for example, updating particle positions

$$x \mathrel{+}= u * dist; \qquad (2)$$

where x is the x-coordinate of the particle, u is the x-direction cosine, and *dist* is the distance to move the particle. Because the multiplication operator (*) is overloaded, POOMA handles the computations for DPFields with no intervention from the user, even though the particles in the problem will reside on different processors.

However, whenever individual particle interactions must be treated, MC++ loops over all nodes in the problem, and all particles local to each node, and handle the interactions. This operation, while serial on individual nodes, is parallel across nodes. Due to the non-data-parallel nature of the problem being solved, this happens often (e.g., some particles undergo collision events, other particles pass through a given cell without a collision and therefore cross a cell boundary into the next cell). This is shown in Fig. 1, and is one of the reasons the unit of parallelization spans individual functions.

D. The Tally Classes

In addition to using the POOMA class library, MC++ has class definitions of it's own to support abstractions appropriate to the problem being solved. One such set of classes are those that are used to implement Monte Carlo tallies. These classes are briefly described here, but additional information is available elsewhere[10].

In a Monte Carlo calculation, there are generally two types of tallies. Those that count events, and those that keep track of some other quantities. The main difference is that event counters can be represented as integers, while other types of counters are represented by floating point numbers. MC++ implements tallies using these simple ideas. The abstract base class AbstractTally is a *templated* class containing common functions and data for all types of tallies. These include functions that will allocate memory for the tally (if needed), initialize the tally, reset the tally, increment the tally, sum the tally at appropriate intervals, do multi-processor gathers of tallies, broadcast summed results to other processors, and so on. Because AbstractTally is an *abstract* class, it *cannot* be instantiated. It can only be used to derive other kinds of tallies. In addition, AbstractTally is a templated class, meaning that at compile time the compiler will decide what type (int, float, *etc.*) a given derived tally is and call the correct functions throughout the code. In MC++, there are currently two derived types of tallies, both of which are concrete (meaning they can be instantiated and used in the code). These are the SimpleTally and Tally classes.

The SimpleTally class is used for integer tallies, and is very simple. It overloads the ++ operator to allow all such tallies to be incremented in a straightforward way, e.g., *++number_collisions;*. It also provides complete definitions for functions to compute a running sum of the tally and to return the total value of the tally at any time (all of the correct type of course, thanks to the templated base class). Naturally, since SimpleTally (and Tally described below) inherit from AbstractTally, member functions of AbstractTally are available to derived objects.

The Tally class is used for all other types of tallies. In addition to defining similar functions as SimpleTally, it also defines functions to compute averages and statistical errors on tally data. For example, if one had a tally called "sflux", to return the average value and statistical error in this tally to the user at any time, all one would have to do is *cout << "Surface flux = " << sflux.average() << " +/- " << sflux.error() <<endl;* The class member functions take care of gathering tally data from all processors (if a parallel calculation), computing the sum, re-broadcasting the values, and computing the average value and the statistical error. The class user need only call these functions for *any* given tally where the physics dictates.

The power of such objects should be clear. Once one has provided definitions for the behavior of all tallies through the class structure, one can instantiate (or create) *any* new tallies *immediately* at *any* time, and all tallies share the *same* functionality, the *same* interfaces, and so on. For example, all current and future tallies can handle parallel or serial computations with no special consideration or intervention from the user of the tally objects. This highlights one of the benefits of object-oriented programming, the separation of the *interface* to some abstract object from the *implementation* of that object under differing circumstances.

E. The Cross Section Classes

Similar to the Monte Carlo tally classes are a set of much more complex classes to handle the cross section data, and the interface to that data, in MC++. This set of classes makes use of the standard template library (STL), and consists of a complex series of base and inherited classes, each working in concert to provide the class user with the same interface and functionality regardless of the cross section set, or type, used. That is, the cross section objects exhibit *polymorphism.* This is very handy, as one can retrieve any cross section value needed in the same way, e.g., *isotope->total_xsec(argument)* where the argument *type* and possibly the number of arguments tells the compiler which overloaded member function to use to return the appropriate value (i.e., multigroup, continuous). In this example, the proper cross section is returned for *isotope.* Since most of the cross section information is stored as STL Maps[12], STL interface functions are used by the cross section class member functions to return the appropriate value for the cross section.

This set of abstractions leads to a powerful capability, in which one can seamlessly read in, manipulate, and use cross sections of widely varying types.

IV. PHYSICS AND PERFORMANCE RESULTS

In this section, physics and performance results for a series of test problems on a variety of platforms is discussed.

A. Platforms

MC++ was developed to be highly portable, and to run in parallel on platforms with parallel capability once compiled there. Table I shows a list of all platforms MC++ has been run on to date.

TABLE I
Computational Platforms

Platform Abbreviation	Description
SGI64_MPI	64-bit SGI R10000 with multiple heads
SGI5	32-bit SGI Cluster
RS6K	IBM RS6000 Cluster
SGIMP	32-bit SGI R8000 with multiple heads
T3D	Cray T3D
TFLOP	ASCI Red Intel TeraFLOPS
ORIGIN	ASCI Mountain Blue SGI Origin 2000
SP2	ASCI Pacific Blue IBM SP2
SUN4SOL2	Sun Sparc10, Solaris 2.5
SGIO2	SGI O2

Because the MC++ application has been developed for a specific program (the Accelerated Strategic Computing Initiative, or ASCI), most of the results shown here are only for ASCI-relevant hardware. This hardware includes the SGI64_MPI, TFLOP, ORIGIN, and SP2.

Note also from the table that all platforms shown in bold have parallel capability. The platform abbreviations shown are used throughout this paper.

B. Test problems

Several test problems were used to fully test the physics in MC++. Here, we focus on the double density godiva series of tests. These problems consists of a bare Uranium sphere of radius 6.993555 cm. As previously mentioned, the geometry is described using a rectangular mesh generated by another code[4]. A variety of tests were performed on meshes of different resolutions and differing characteristics. Tested mesh sizes were 32x32x32 (32,768 cells), 64x64x64 (262,144 cells), 128x128x128 (2,097,152 cells), and 256x256x256 (16,777,216 cells).

C. Portable Parallelism

For the most part, MC++ has been developed and debugged on a single platform (SUN4SOL2). Once it was mature enough to run test problems, it was simply compiled on all platforms of interest and run there. However, not only did the code run on these different platforms, but it did so in *parallel*, with no additional work or special considerations on any of the platforms in question. This highlights one of the benefits of the POOMA framework, *portable parallelism*. Through POOMA's architecture and communications abstractions, MC++ is not only portable to different platforms, but also runs in parallel on these platforms, allowing us to do most development locally in a robust computing environment rather than on somewhat experimental architectures with poor development environments.

With the exception of some tuning of our sourcing algorithm and some problems with NetCDF and parallel I/O on the T3D (the meshes are provided to MC++ via NetCDF portable binary files), the code was compiled and run in parallel without incident on all platforms shown in Table I. As the code grew in complexity, and we added additional physics features, we continued to enjoy the abstractions offered by POOMA. This greatly facilitated getting MC++ up and working across all platforms in a short period of time (it was developed in about 5 months).

D. Physics Results, k

All *k*-effective problems were run with 30 cycles, 20 of which were used to compute averages and errors. Fig. 2 shows a representative result from the TFLOP platform. The convergence of *k* on all platforms was the same, converging to the expected value of 1.4. In addition, as the mesh resolution increased, the convergence improved. Shown in Fig. 2 is a calculation on the 2 million mesh cell godiva problem compared to an MCNP calculation on an analytic spherical surface of the proper radius.

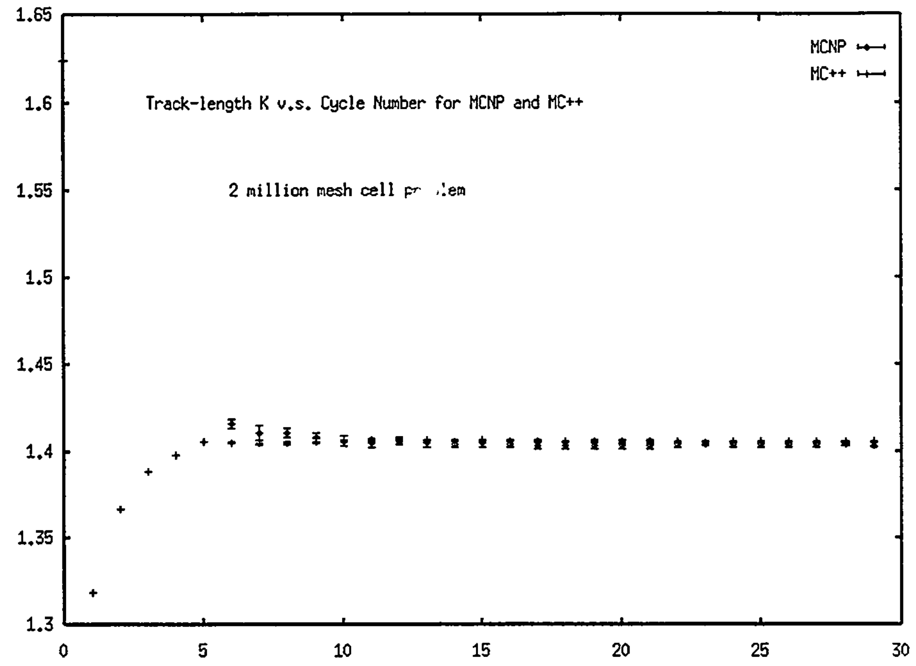

Fig. 2: Track-length *k*-effective v.s. cycle number compared to MCNP on the 2 million mesh ceıı problem. Here, MC++ was run on TFLOP using 32 processors. MCNP was run serially on the SUN4SOL2 platform.

E. Physics Results, α

All α-eigenvalue problems were run with 18 α-cycles, 2 inner *k*-cycles per α-cycle, 12 α-cycles used for regression analysis, and 2 settle cycles prior to the start of the α-calculation. On all platforms in question, MC++ computed values for the a-eigenvalue clustered around the expected value of 1.22 gen/sh (where 1 sh is 10^{-8} sec). A typical value for α on the 64x64x64 mesh (as an example) was 1.224 gen/sh +/- 0.084. As with the *k*-calculations, the higher the resolution the mesh, the better the results, as expected.

F. Timing Results

Even without any special performance tuning on any of the platforms in question, parallel performance of MC++ is quite reasonable.

Fig. 3 shows the parallel performance of MC++ running the k-effective calculation on a variety of platforms and the 32x32x32 mesh. This plot is representative of the performance noted on all parallel platforms in Table I.

Fig. 4 shows parallel performance of MC++ running on the α-eigenvalue calculation on ASCI hardware. Note that as the amount of work to do increases (either through the addition of particles or mesh cells), the parallel efficiency increases.

Note from all of these figures that the parallel performance is good, and good speedups are noted, even without special tuning or other platform-specific considerations. We just compiled and ran MC++ and produced these results.

Note in particular that the timings shown in Fig. 4 indicate that the computation of the α-eigenvalue is now possible in a manner of *seconds*, which is orders of magnitude faster than previously available with other codes available at Los Alamos.

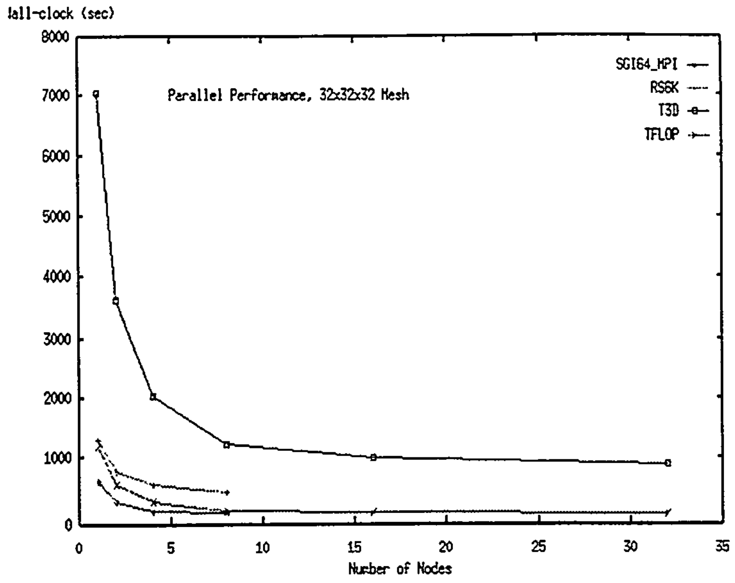

Fig. 3: *k*-eigenvalue performance on the 32x32x32 mesh for a variety of platforms.

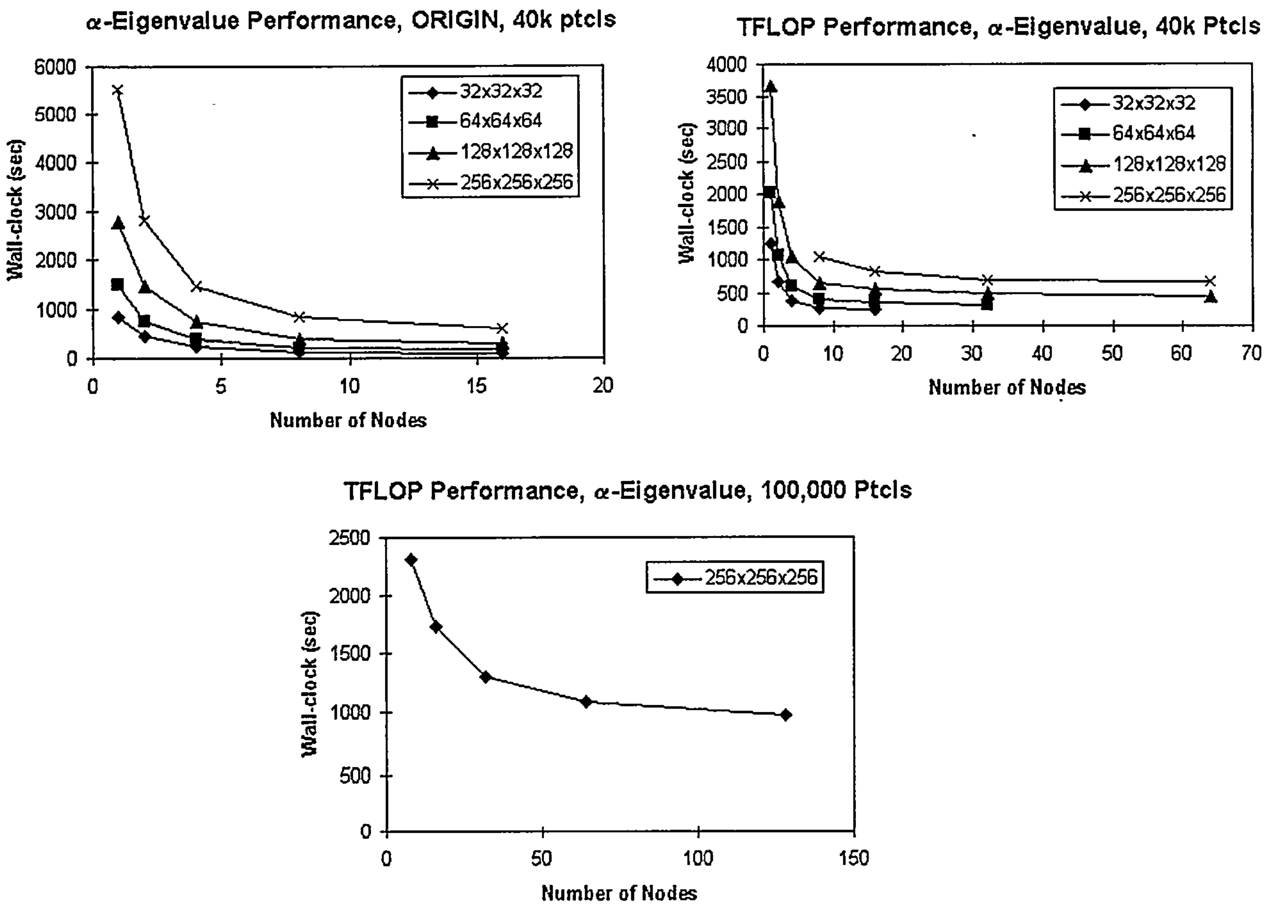

Fig. 4: α-eigenvalue performance on ASCI hardware. Note as the work to do increases, so does the parallel efficiency. Performance results for the large problem on TFLOP for fewer than 8 nodes not possible due to memory constraints.

V. CONCLUSIONS AND FUTURE WORK

Beyond careful performance tuning, which has not been done, there are some new techniques and new physics that need to be added. MC++ can serve not only as a computational physics tool, but also as a platform on which to try some new methods for Monte Carlo transport. These methods include the implementation of an *importance combing* technique, in which particle tracks and weights are manipulated in different ways to enhance convergence, or even the investigation of the application of genetic algorithms to further enhance convergence.

However, this work was not intended to just provide computational physics support for ASCI. It was also intended to help a group of people involved in simulating transport phenomenon with legacy Fortran code to learn a new paradigm, and enable the migration of capabilities encapsulated in these codes to different computing platforms. MC++ is the beginning of a transport physics framework (TPF), which is a class library containing proper abstractions for transport physics, just as POOMA is a class library containing proper abstractions for portable parallelism. This TPF will include proper abstractions for events, energy deposition, variance reduction techniques, spatial differencing, synthetic accelerations, sources, and so on. It will encapsulate transport physics, and will include Monte Carlo as well as other methods to solve the transport equation under different circumstances. MC++ is the first step in this direction. Further abstractions will be made to different mesh types, particle types, problem regimes, and so on.

MC++ has been developed over a period of about 5 months. In that time, we have developed a code that is capable of computing static k-eigenvalues and α-eigenvalues on large problems in parallel on all relevant computing platforms of the day. This portable parallelism proved to be quite valuable as the code was developed and debugged on local workstations with robust programming environments and rich development tools, then re-compiled and run in parallel on the more exotic hardware without incident. The fact that we have not yet performance-tuned MC++, yet were able to achieve these parallel speedups in a short period of time is a significant accomplishment. Although not inherently data-parallel, we have shown that the Monte Carlo problem is castable into data-parallel form, and that our implementation of transport physics in C++, using object-oriented methods and POOMA, can produce a code that is reasonably fast and efficient in a short period of time. This is critical from an ASCI perspective, as platforms and computing environments will rapidly change. It will be crucial to be able to respond to these changes, yet maintain a physics capability while always developing new capabilities and new methods. MC++ is a big step in this direction.

REFERENCES

1. Lewis, E.E., Miller, W.F., Computational Methods of Neutron Transport, John Wiley & Sons Inc., 1984.
2. Glasstone, S., Sesonske, A., Nuclear Reactor Engineering, Chapman & Hall, 1994.
3. J.F. Briesmiester, ed., "MCNP -- A General Monte Carlo N-particle Transport Code, Version 4A", Los Alamos National Laboratory Report, LA-12625-M.
4. Nolen, S.D., Lee, S.R., Cummings, J.C., "Adding Mesh Tracking Capability to MC++", X Division Research Note, Los Alamos National Laboratory, XTM-RN(U)96-019, 1996.
5. Parratt, L.G., Probability and Experimental Errors in Science, John Wiley & Sons Inc., 1961.
6. Wilson, G., Lu, P., ed., Parallel Programming Using C++, MIT Press, 1996.
7. Appley, G., Gallaher, M., "A Framework for Manufacturing-Process Simulation Software", Object Magazine, May 1996, pg. 33.
8. See http://www.acl.lanl.gov/PoomaFramework.
9. Lee, S.R., Cummings, J.C., Nolen, S.D., "Building a Transport Code using POOMA and Object-Oriented Methods", X Division Research Note, Los Alamos National Laboratory, XTM-RN(U)96-003, 1996.
10. Lee, S.R., Cummings, J.C., Nolen, S.D., "Some C++ Classes for Monte Carlo Tallies", X Division Research Note, Los Alamos National Laboratory, XTM-RN(U)96-004, 1996.
11. See http://www-xdiv.lanl.gov/XTM/srlee/PROJECTS/MC++.
12. Musser, D., Saini, A., STL Tutorial and Reference Guide, Addisson-Wesley Publishing Company, 1996.