LA-UR 96-3619

Conf-970430--2

TITLE: MC++: A PARALLEL, PORTABLE, MONTE CARLO NEUTRON TRANSPORT CODE IN C++

AUTHOR(S): Stephen R. Lee
Julian C. Cummings
Steven D. Nolen

SUBMITTED TO: 30th Annual Simulation Symposium, Atlanta, Georgia, April 7-9, 1997

CONF-970430--2

LA-UR--96-3619

MC++: A Parallel, Portable, Monte Carlo Neutron Transport Code in C++

Stephen R. Lee
Transport Methods Group, MS B226
Applied Theoretical and Computational Physics Division
Los Alamos National Laboratory
Los Alamos, NM 87544
Phone: 505 665 2989
Fax: 505 665 5538
Email: srlee@lanl.gov

Julian C. Cummings
Los Alamos National Laboratory

Steven D. Nolen
Texas A&M University[1]

Abstract

MC++ is an implicit multi-group Monte Carlo neutron transport code written in C++ and based on the Parallel Object-Oriented Methods and Applications (POOMA) class library. MC++ runs in parallel on and is portable to a wide variety of platforms, including MPPs, SMPs, and clusters of UNIX workstations. MC++ is being developed to provide transport capabilities to the Accelerated Strategic Computing Initiative (ASCI). It is also intended to form the basis of the first transport physics framework (TPF), which is a C++ class library containing appropriate abstractions, objects, and methods for the particle transport problem. The transport problem is briefly described, as well as the current status and algorithms in MC++ for solving the transport equation. The alpha version of the POOMA class library is also discussed, along with the implementation of the transport solution algorithms using POOMA. Finally, a simple test problem is defined and performance and physics results from this problem are discussed on a variety of platforms.

Description of the General Problem

We begin with the time-dependent transport equation for multiplying systems (neglecting delayed neutrons)[2] [Lewis & Miller 1984]

$$\begin{aligned}\left[1/v\frac{\partial}{\partial t} + \hat{\Omega}\cdot\vec{\nabla} + \Sigma(\vec{r}, E)\right]\psi(\vec{r}, E, t) &= Q(\vec{r}, \hat{\Omega}, E, t) + \\ &\int dE' \int d\hat{\Omega}'\Sigma_s(\vec{r}, E' \rightarrow E, \hat{\Omega}'\cdot\hat{\Omega})\psi(\vec{r}, \hat{\Omega}', E', t) + \\ &\chi(E)\int dE' \upsilon(E')\sigma_f(\vec{r}, E')\int d\hat{\Omega}'\psi(\vec{r}, \hat{\Omega}', E', t)\end{aligned} \tag{1}$$

Where $\hat{\Omega}'$ represents the direction of travel and $\psi(\vec{r}, \hat{\Omega}, E, t)$ represents the angular flux, such that

1. Currently a graduate research assistant working with Stephen Lee at LANL.
2. Cross section notation has been modified from the reference to be self consistent in this document, which generally uses [Bell & Glasstone 1970] notation.

$$\psi(\vec{r}, \hat{\Omega}, E, t)dVd\Omega dEdt$$

represents the total of the path lengths traveled during dt by all particles in the incremental phase space volume $dVd\Omega dE$. The term Q represents external sources of neutrons, $\Sigma_s(\vec{r}, E' \rightarrow E, \hat{\Omega}' \cdot \hat{\Omega})$ is the scattering cross section, and $\Sigma_f(\vec{r}, E)$ is the fission cross section. The mean number of fission neutrons produced in a fission caused by an incident neutron of energy E is given by $\upsilon(E)$, and $\chi(E)$ is the energy spectrum of fission neutrons such that

$$\upsilon(E)\Sigma_f(\vec{r}, E)\chi(E')dE'd\Omega'$$

is the probable number of fission neutrons produced at $\vec{r}$, with energies within dE' about E', within the cone of angles $d\Omega'$ about $\hat{\Omega}$, per path length traveled by neutrons with energy E.

In many transport calculations, including MC++, time-dependence is treated *implicitly.* Therefore, re-writing (1) in a time-independent form and assuming no external sources,

$$[\hat{\Omega} \cdot \vec{\nabla} + \Sigma(\vec{r}, E)]\psi(\vec{r}, \hat{\Omega}, E) = \int dE' \int d\hat{\Omega}' \Sigma_s(\vec{r}, E' \rightarrow E, \hat{\Omega}' \cdot \hat{\Omega})\psi(\vec{r}, \hat{\Omega}, E) + \chi(E)\int dE' \upsilon(E')\sigma_f(\vec{r}, E')\int d\hat{\Omega}' \psi(\vec{r}, \hat{\Omega}, E) \qquad (2)$$

A system containing fissionable material is said to be *critical* if there is a self-sustaining time-independent chain reaction in the absence of external sources of neutrons. That is, after sufficient time has passed, a time-independent, asymptotic distribution of neutrons will exist such that the rate of fission neutron production is *just equal* to the losses due to absorption and leakage from the system. If such an equilibrium cannot be established, the neutron distribution will either increase or decrease exponentially in time. Systems are said to be *subcritical* if the population decreases, and *supercritical* if it increases.

To treat the criticality problem, MC++ introduces an auxiliary eigenvalue, *k*. The average number of neutrons per fission event, ν, is replaced by ν*/k*, and *k* is then adjusted to obtain a time-independent asymptotic solution to the transport equation [Bell & Glasstone 1970]. Therefore, at $k = k_{eff}$, the effective multiplication factor, this solution is found. This amounts to varying the number of neutrons emitted per fission by *1/k*. Therefore, we cast (2) as an eigenvalue problem and write

$$[\hat{\Omega} \cdot \vec{\nabla} + \sigma(\vec{r}, E)]\psi(\vec{r}, \hat{\Omega}, E) = \int dE' \int d\hat{\Omega}' \Sigma_s(\vec{r}, E' \rightarrow E, \hat{\Omega}' \cdot \hat{\Omega})\psi(\vec{r}, \hat{\Omega}, E) + \chi(E)/k\int dE' \upsilon(E')\Sigma_f(\vec{r}, E')\int d\hat{\Omega}' \psi(\vec{r}, \hat{\Omega}, E) \qquad (3)$$

Any chain reaction can be made critical if the number of neutrons per fission is adjusted between zero and infinity. Clearly the system is just critical if *k=1*. A value of *k<1* implies that the hypothetical number of neutrons per fission required to make the system critical is *larger* then ν, the number physically available. Therefore, such a system is sub-critical. If *k>1*, the number of neutrons required to make the system critical is *fewer* than that available in reality and hence the system is supercritical.

MC++ calculates this effective multiplication factor, k_{eff} (which will be called k from now on).

Description of the Algorithm

All transport codes require some sort of numerical description of the geometry and composition of the system. For MC++, the geometry description is in the form of a three-dimensional cartesian mesh obtained from another simulation code. Each mesh element contains specific information as to its size in each dimension, location of cell vertices, material composition, and density. MC++ then stores this description (which is read from a NetCDF file) along with isotopic information supplied by the user to fully describe the materials and composition of the problem. Equation (3) is then solved in a simple iterative scheme that is widely employed in other transport codes [Briesmeister 1993] and is described briefly here.

The calculation is started by guessing an initial spatial distribution of neutrons. In MC++, this initial guess is a simple scheme that places neutrons in cells containing fissile material in a "round-robin" manner until all particles are exhausted. This, along with an initial guess for the system k-effective (supplied by the user) begins the first iteration in MC++. This is called the "first generation" (or "cycle") of our neutron population. In MC++, k can be thought of as the ratio between the number of neutrons in successive generations, with fission events being regarded as the "birth event" that separates the generations. The mean number of fission neutrons produced in fission events are estimated and these generated neutrons are stored as the *source points* for the *next* cycle. A cycle is therefore defined as the life of all neutrons in the problem from birth (by fission) to death (by escape or capture[1]). Particles in the next cycle are started *isotropically* at the location at which the birth took place.

The user controls the nominal number of particles to track per cycle and the number of cycles during which to accumulate the results. The user can also specify the number of initial cycles to "skip" to allow the neutron population to stabilize before accumulating the results.

MC++ estimates the distance to collision in a given mesh cell by computing

$$cdist = -\ln(\xi)/\Sigma_t(E, \vec{r}) \qquad (4)$$

for each particle in the problem. Here, ξ is a uniformly distributed random number on the interval $[0,1)$, and Σ_t is the total macroscopic cross section in the mesh cell in which the particle resides. MC++ also computes the particle trajectory distance to the nearest boundary (which could be a mesh, problem, or material boundary) [Nolen *et. al.* 1996]. MC++ then selects the smaller of these distances (since this is the closer event) and does the appropriate physical interaction for each particle (collision or boundary crossing event).

In the Monte Carlo estimation of k, let N be the desired number of particles per generation (specified by the user). During the problem one tracks the particles from birth to death through a series

1. Both fission and absorption events are treated as capture events.

of cycles, M. For any given fission event, the number of expected neutrons is given by

$$W \cdot \frac{\nu}{k} \times \frac{\sigma_f}{\sigma_t} \tag{5}$$

in our formulation. In (5), W is the weight of the particle, and is a statistical contrivance which is a measure of an individual particle's relative "importance" in the population. The sum of (5) over all neutron tracks in the *current* cycle gives the total number of particles to be tracked in the *next* cycle (*i.e.*, these points are tallied and stored for the next cycle). When the next cycle begins, all particle weights are normalized by setting them equal to the ratio of N and the number of *actual* source points written out in the *previous* cycle (from (5)). All tallies are therefore also normalized to N, rather than the sum of (5) for each cycle. Note also from (5) the effect of the ν/k term in computing the number of neutrons per fission event, as shown in (3).

During the 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 tallies, or estimators, are used in MC++, and are the same as used in MCNP [Briesmeister 1993].

The *collision* estimator for k is given by

$$k_c = \frac{1}{N}\sum_i W_i \left[\frac{\sum_j f_j \nu_j \sigma_{f,j}}{\sum_j f_j \sigma_{f,j}} \right] \tag{6}$$

where i is summed over all collisions where fission is possible and j is summed over all isotopes in the material involved in the i^{th} collision. $\sigma_{t,j}$ is the total microscopic cross section for nuclide j, f_j is the atomic fraction of nuclide j, ν_j is the average number of neutrons[1] produced by the fissioning nuclide for a neutron at a given incident energy, $\sigma_{f,j}$ is the microscopic fission cross section for nuclide j, and W_i is the weight of the particle entering the i^{th}collision.

In this criticality scheme, fission and absorption are both treated as capture events. Capture can be handled in an *analog* manner, in which the particle undergoing capture is removed from the problem, or *implicitly*, in which the weight of the particle is modified by the probability for capture and a roulette game is subsequently played with the modified weight. In either case, the probability for capture is $(\sigma_{a,j} + \sigma_{f,j})/\sigma_{t,j}$, the sum of the absorption and fission cross sections divided by the total cross section for isotope j in which the event occurs.

The estimates for the absorption k differ, however. For analog capture

$$k_a = \frac{1}{N}\sum_i W_i \left[\frac{\nu_j \sigma_{f,j}}{\sigma_{a,j} + \sigma_{f,j}} \right] \tag{7}$$

where i is summed over each analog capture event in the j^{th} isotope.

1. This value can include *prompt* neutrons or *total* neutrons depending on the cross section data used.

For implicit capture,

$$k_a = \frac{1}{N}\sum_i W'_i\left[\frac{\nu_k\sigma_{f,j}}{\sigma_{a,j}+\sigma_{f,j}}\right] \tag{8}$$

where in this case i is summed over all collisions in which a fission is possible, W'_i is the weight adjustment for implicit capture (i.e., the incoming particle weight times the capture probability discussed above), and j is the nuclide involved in the collision.

Finally, the track-length estimator for k is given by

$$k_t = \frac{1}{N}\sum_i W_i\rho d\sum_j f_j\nu_j\sigma_{f,j} \tag{9}$$

where i is summed over all particle trajectories, ρ is the atomic density in the mesh cell that the particle is in at the time of the score, and d is the trajectory distance travelled from the last event.

The estimates for k are accumulated using (6), (9), and (7) or (8) each cycle. The average values and relative errors of these estimators are computed over all active cycles M using standard equations.

These are the main tallies in MC++. However, there are a variety of other, simpler tallies called *event* tallies that simply count the number of times something happens. Because tallies are crucial to Monte Carlo transport calculations, MC++ has an abstract base class, and two derived classes devoted to handling tallies. As a result, implemented tallies in MC++ know how to clear themselves, count themselves, compute their own averages and errors, and even communicate with other nodes in a parallel computation when needed (e.g., for global summation). These classes are not discussed in detail here, but are available elsewhere [Lee *et. al.* 1996a].

Brief Overview of POOMA-alpha

The POOMA Framework is a C++ class library intended to support a wide variety of parallel scientific computing applications [Wilson & Lu 1996]. POOMA stands for parallel object-oriented methods and applications and has been in development since 1994. 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 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 [Appley & Gallaher 1996]. It is an integrated and layered system of classes, in which classes in higher layers utilize the classes from lower layers to build capability. POOMA is built from 5 such class layers, as shown in Figure 1.

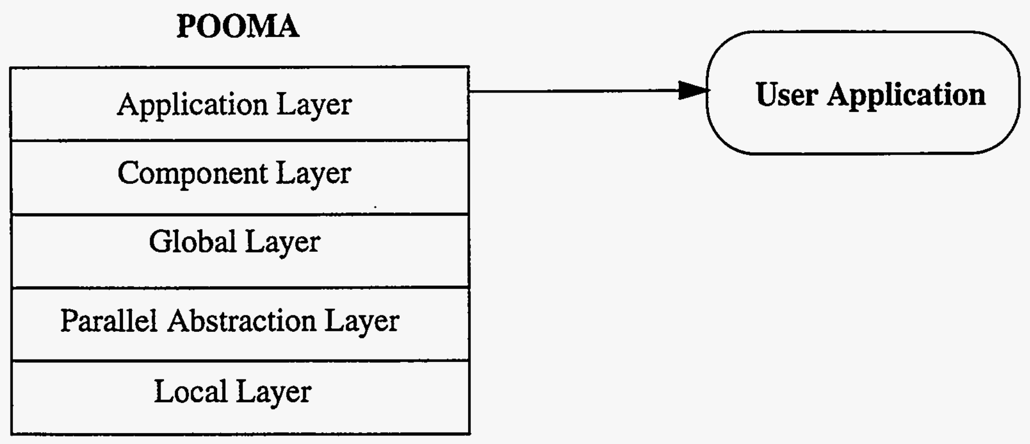

Figure 1: The POOMA Framework hierarchy. The user application can actually access POOMA at any level.

The higher levels in the Framework hierarchy represent abstractions that are directly relevant to the application (such as distributed data structures and standard numerical methods) while classes at the lower level represent parallel abstractions and computational kernels. The typical user application works only with classes in the higher levels although the user can access any level of the Framework,

POOMA 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.

The 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.

For example, the DPField class contains overloaded operators that allow one to update all particles positions in a native data-parallel statement,

x += u*dist; (10)

where *x* represents the x-coordinate of all particles in the problem, *dist* is the distance to move all particles in the problem, and *u* is the u-direction cosine of all particles in the problem. In this way, all particles in the problem have their position updated in parallel on all processors. Because the multiplication operator (*) is overload, POOMA handles the computations for DPFields with no intervention from the user, even though the particles in the problem will reside on different processors.

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.

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 [Wilson & Lu 1996, http://www.acl.lanl.gov/PoomaFramework].

POOMA Implementation of Transport Physics

Naturally, to build physics simulation software using POOMA, one must buy-in to the virtual node construct and other POOMA class implementations. The problem also needs to be castable into data-parallel form.

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 as in (10), 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 [Nolen *et. al.* 1996]. 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 in the transport algorithm, data-parallel updates of particle attributes are done, as shown in (10) for updating the particle positions. However, whenever individual particle interactions must be treated, MC++ loops over all nodes[1] in the problem, and all particles local to each node, and handles 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).

Even given these constraints, the problem has been properly formulated using POOMA. For more details on how the transport algorithm was formulated using POOMA, see [Lee *et. al.* 1996b, Nolen *et. al.* 1996].

A Simple Test Problem

To facilitate testing our implementation of physics in MC++, we used a "collision-only" version, in which there are no problem boundaries and no way for particles to "escape" from the problem. Thus this forms an *infinite medium* problem, in which k is given by the ratio of the number of neutrons *produced* in a single generation to the number *absorbed* in the previous generation [Glasstone & Sesonske1994]. That is

$$k_\infty = \frac{\nu\sigma_f}{\sigma_a + \sigma_f} \tag{11}$$

Using (11), we generated a specific 1-group cross section set with appropriate values for all quantities that would produce a k of 1.0. It is this problem that we used to test MC++.

Preamble

Before discussing physics and timing results, some background information is needed on the platforms tested and on portability experiences with MC++.

Platforms

MC++ was tested on a wide variety of platforms and, where possible, tested on multiple nodes. MCNP version 4B was also run on the same problem on the same platforms to provide a comparison for physics (and to a lesser degree, performance). MCNP is a long-standing, well known, powerful and very well-tested Monte Carlo transport code. For more information on MCNP, see [Briesmeister 1993].

1. In POOMA, the concept of a virtual node is used, which is a useful abstraction to enable load balancing, architecture independence, and so on. In this paper, virtual nodes and nodes are used interchangeably. One can think of a node as a physical processor for the purpose of this discussion, although in reality it is not.

Table 1 is a summary of the platforms used throughout these tests.

Table 1: Tested Platforms

Platform Abbreviation	Platform Name	Description
SGI64	wort.acl.lanl.gov	64-bit SGI R8000
SGI5	kent.acl.lanl.gov, *et. al.*	32-bit SGI cluster with MPI
SGIMP	black.acl.lanl.gov	32-bit multi-headed SGI with MPI
RS6K	cluster.lanl.gov	IBM RS6000 cluster with MPI
T3D	t3d.acl.lanl.gov	Cray T3D with MPI
SGI64_MPI	jasper.acl.lanl.gov	64 bit multi-headed SGI R10000 with MPI
TFLOP	tinyflop.cs.sandia.gov	ASCI Red Intel Teraflop
SUN4SOL2	u2.lanl.gov	Sun Sparc10 with Solaris 2.5

In Table 1, all platforms listed in bold have parallel capability. MC++ was tested in both single and multi-node mode on all of these platforms. Unfortunately, while MCNP will run in parallel using PVM, we were unable to test MCNP in parallel on any platform other than the T3D. This was due to a variety of reasons, including the restriction to PVM (which was not available on all tested platforms) and problems with its implementation on other platforms.

Portable Parallelism

For the most part, MC++ has been developed and debugged on a single platform (Sun workstation running Solaris 2.5). 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 virtual node model and architecture and communications abstractions, MC++ is not only portable to different platforms, but also runs in parallel on these platforms. This has allowed 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 on the T3D, the code was compiled and run in parallel without incident on all platforms. As the code grew in complexity, and we added additional physics and features, we continued to enjoy the abstractions offered by POOMA. POOMA greatly facilitated getting MC++ up and working in parallel across all of these platforms in a short period of time (MC++ has been developed in about 5 months).

Physics Results

The problem as defined previously was run with 40,000 nominal source particles per cycle for a total of 10 cycles. 9 of these cycles were used to compute the average quantities and relative errors in the estimators.

Table 2 shows the final k-effective estimators for MC++ and MCNP for the problem in question. The track-length and collision estimators are shown here. Because we are using implicit capture, and due to the nature of the problem, the absorption estimator is identical to the collision estimator, and is not shown.

Table 2: MC++ and MCNP Physics Results

Code	Nodes	Collision K	Relative Error	Track-length K	Relative Error
mcnp	1	0.9996	0.0005	0.9993	0.0013
MC++	1	1.0003	0.0006	1.0016	0.0010
MC++	1	1.0003	0.0006	1.0016	0.0010
MC++	2	0.9998	0.0005	1.0028	0.0015
MC++	4	0.9998	0.0003	1.0007	0.0012
MC++	8	0.9999	0.0005	1.0010	0.0016
MC++	16	0.9999	0.0005	1.0010	0.0016

MC++ same-node results on all platforms are identical, so only the node number, rather than the platform, are shown in table 2 (*e.g.*, an 8 node run on RS6K yields identical results to an 8 node run on the T3D).

Note that good agreement with MCNP is seen. The fact that the results are different (although statistically identical) using differing number of nodes is due to the different random number sequence used on each node. Repeated runs using the same number of nodes produces the exact same results, of course.

Figures 2 and 3 show representative plots of both the collision k estimator and the track-length k estimator for this problem. Note the convergence of the solution to the correct answer, and note the overlapping MCNP and MC++ results. Similar results were seen for all estimators on all platforms tested.

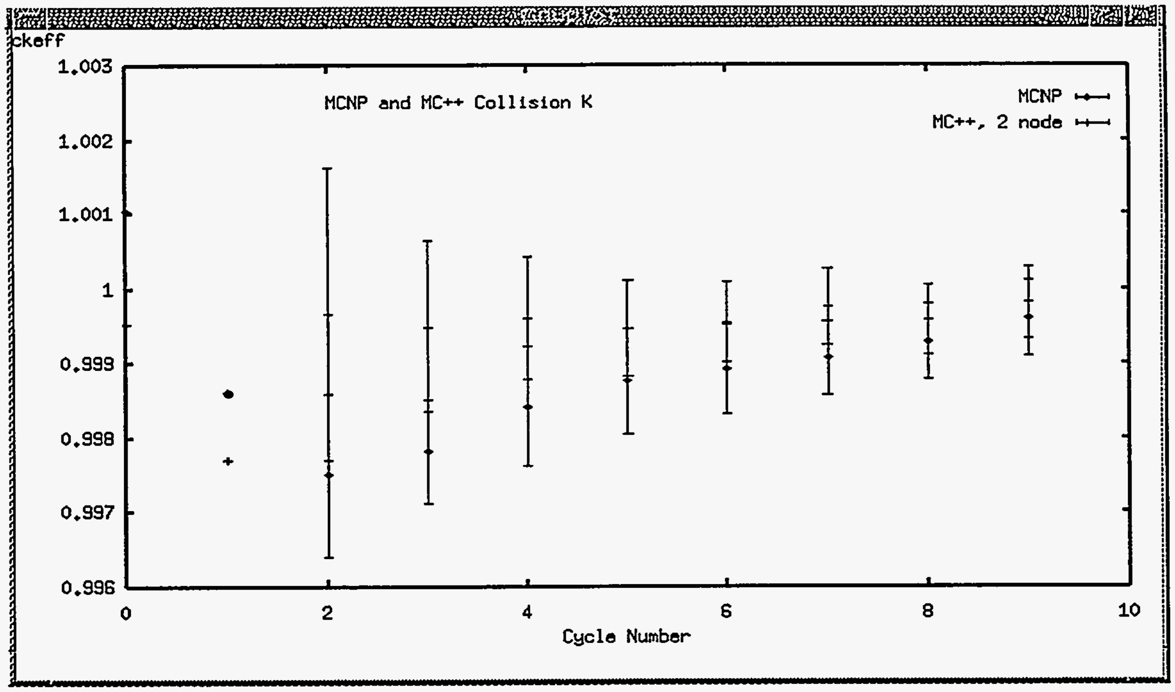

Figure 2: MCNP and MC++ collision k-effective for 2 nodes. Note that the error bars overlap in the final result.

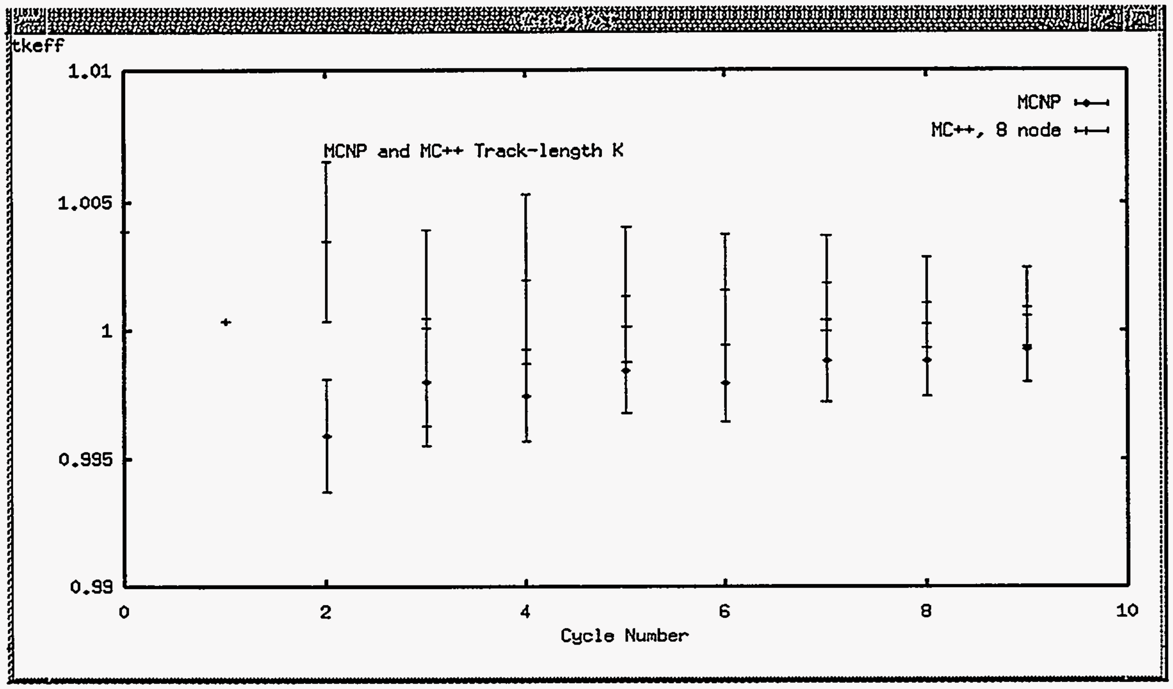

Figure 3: Track-length K for MCNP and MC++, 8 nodes.

Performance Results

Both a comparison with MCNP and parallel performance are described in this section. It should be noted that while these comparisons are interesting, they are here to illustrate that MC++ is not unreasonably slow, and is even competitive, with its Fortran counterpart. However, MCNP has physics capabilities far beyond that of MC++, which was written for a specific target application.

No timing studies were done beyond 16 nodes due to the size of the test problem. Because it is a small problem, further increasing the number of nodes degraded performance due to the lack of work for each node to perform.

MCNP Comparison

In Table 3, timing results are shown for single and multi-node runs of MC++, and single node runs for MCNP. In addition, multi-node runs for MCNP are shown on the T3D.

Table 3: MC++ and MCNP Timings

Code	Platform	Nodes	CPU (sec)	Wall[a] (sec)	MC++/MCNP[b] (wall)	Parallel Speedup[c]
mcnp[d]	SGI64	1	115.5	179.0	-	-
MC++[e]	SGI64	1	100.9	102.2	0.57	-
mcnp	SGI64_MPI	1	34.6	34	-	-
MC++	SGI64_MPI	1	50.9	51.8	1.50	-
MC++	SGI64_MPI	2	31.2	31.5	-	1.6
MC++	SGI64_MPI	4	18.4	19.2	-	2.7
mcnp	RS6K	1	53.3	116	-	-
MC++	RS6K	1	119.6	127.4	1.09	-
MC++	RS6K	2	73.0	85.4	-	1.5
MC++	RS6K	4	41.8	46.6	-	2.7
MC++	RS6K	8	46.5	51.6	-	2.5
mcnp	SGIMP	1	120.4	121	-	-
MC++	SGIMP	1	106.8	107.4	0.89	-
MC++	SGIMP	2	92.04	92.6	-	1.2
MC++	SGIMP	4	50.9	51.4	-	2.1
mcnp	SGI5 (kent)	1	293.9	324	-	-
MC++	SGI5_MPI	1	207.3	217.5	1.5	-
MC++	SGI5_MPI	2	133.9	277.0	-	0.8

Code	Platform	Nodes	CPU (sec)	Wall[a] (sec)	MC++/MCNP[b] (wall)	Parallel Speedup[c]
mcnp	SUN4SOL2	1	105	264	-	-
MC++	SUN4SOL2	1	193.1	195.4	0.74	-
mcnp	T3D	1	157.8	248	-	-
mcnp	T3D	2	154.8	242	-	1.0
mcnp	T3D	4	219.0	322	-	0.8
mcnp	T3D	8	82.2	145	-	1.7
mcnp	T3D	16	57.0	110	-	2.2
MC++	T3D	1	225	277.5	1.12	-
MC++	T3D	2	106.8	159.1	0.66	1.7
MC++	T3D	4	82.2	104.0	0.32	2.7
MC++	T3D	8	46	59.8	0.41	4.6
MC++	T3D	16	0[f]	40.4	0.37	6.9
mcnp[g]	TFLOP	-	-	-	-	-
MC++	TFLOP[h]	1	107	107	-	-
MC++	TFLOP	2	69	69	-	1.6
MC++	TFLOP	4	42	42	-	2.6
MC++	TFLOP	8	33	33	-	3.2
MC++	TFLOP	16	32	32	-	3.3

a. Wall-clock times were taken on non-dedicated resources.
b. Ratio of MCNP wall-clock to MC++ wall clock (same number of nodes only).
c. Ratio of single:multiple node wall clock times for MC++.
d. Times reported for MCNP are those from the "time" command.
e. Times reported for MC++ are from POOMA timers used within the code.
f. On the T3D, the only CPU time available was that reported by the operating system. When the CPU time consumed is less than around 20-30 seconds, the T3D OS often reports 0.
g. MCNP is not available on this platform.
h. TFLOP is a *dedicated* resource, so the CPU and wall-clock times are identical.

From this table, *single* node runs of MC++ are on average 1.2 times faster than MCNP (i.e., it is essentially the same speed). This is better than expected considering the lack of sophistication in some C++ compilers, and the table look-up performance of C++ compared to Fortran (MCNP is a Fortran 77 code). The parallel runs of MC++ are all faster than their serial MCNP counterparts. Note also that the 16 node T3D MC++ run is 6 times faster than the single node MCNP run, and 3 times faster than the 16 node MCNP run. Overall, parallel MC++ is on average around 5 times faster than serial MCNP . MCNP's maximum parallel speed up is 2.2, and MC++'s is 6.9. As the work load increases, so will the efficiency of parallel MC++. For example, early tests of the boundary-crossing algorithms combined with collision physics show factors of 20-30 improvement over MCNP.

The single-node performance of MC++ is reasonable, but could be improved. The multi-node efficiency of MC++ is good. Therefore, while the one-node POOMA efficiency is not too bad, the multi-node speedups are reasonable given the size of this problem (not large) and the complex mapping of the transport calculation into the POOMA Framework. We anticipate additional improvements in single-node performance with the next POOMA release and careful performance tuning of MC++.

Parallel Performance
Figure 8 shows parallel performance on all platforms for MC++.

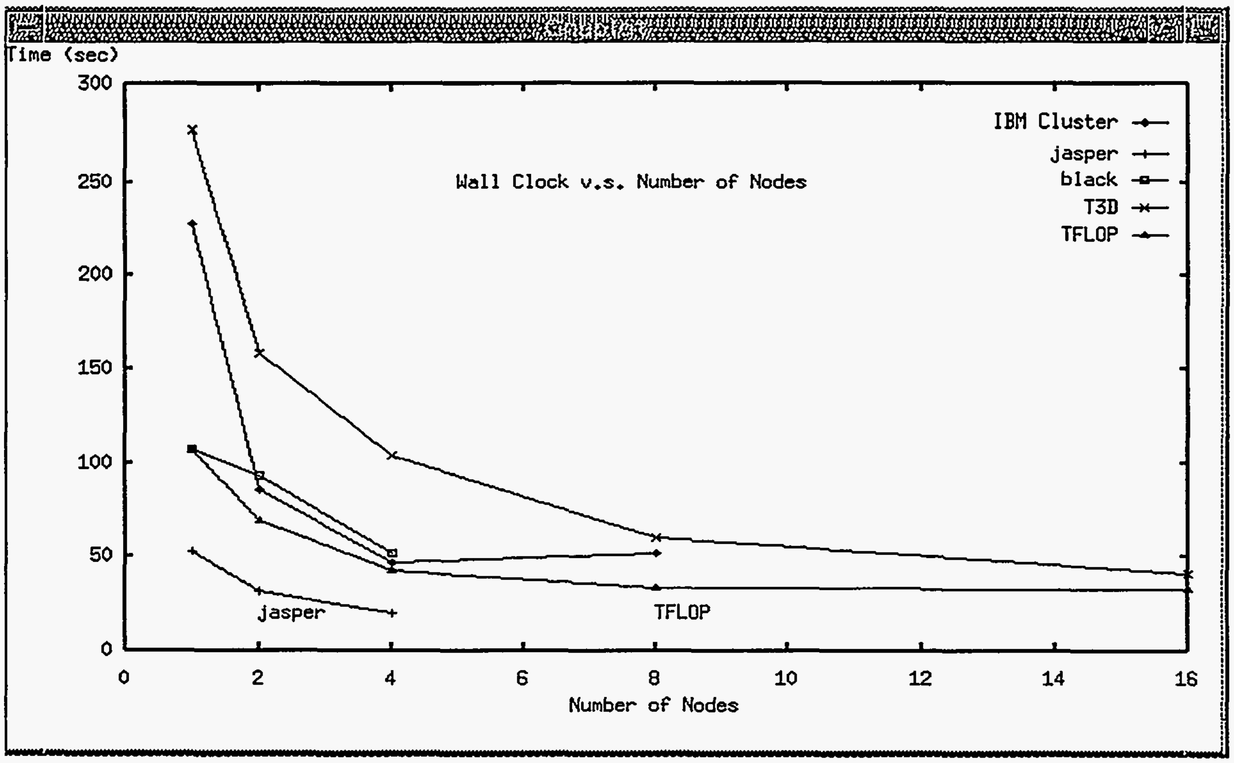

Figure 8: Parallel speedups on each platform for MC++.

As can be seen in Figure 8 and Table 3, the parallel performance of MC++ is quite reasonable, particularly *without any special work* on each platform to tune for parallel performance. We just compiled and ran.

Future Work
From the perspective of MC++ itself, there are several things that need to be done. First, the boundary crossing algorithm needs to be fully implemented and tested (which is underway). Also, careful performance tuning should be done. Beyond this, 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 [Booth 1996], 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. Additional development is required, however, and to this end, some of the methods within MC++ will be re-designed using full object-oriented methods, much as the tally classes were done. Further abstractions will then be made to different mesh types, particle types, problem regimes, and so on. This will provide a TPF which would not only be used for ASCI applications, but also for the development of new methods to be tested and applied within the ASCI program.

Conclusions

Initially, we have concentrated on getting the physics right in MC++ and being portable, rather than on performance. The timing studies presented were our first look for show stoppers, and nothing more. The fact that we did not tune MC++ for performance, yet were able to achieve these results on this many platforms in such a short period of time is a significant accomplishment. POOMA-alpha provides an appropriate architecture-independent abstraction for our problem. 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

[Appley & Gallaher 1996] G. Appley, M. Gallaher, A Framework for Manufacturing-Process Simulation Software. Object Magazine, May 1996. (page 33)

[Bell & Glasstone 1970] G. Bell, S. Glasstone, **Nuclear Reactor Theory**. 1970, Litton Educational Publishing, Inc.

[Booth 1996] T. Booth, A Weight (Charge) Conserving Importance-Weighted Comb for Monte Carlo. Proceedings of the 1996 American Nuclear Society topical meeting, "Radiation Protection and Shielding" Falmouth, Massachusetts, April 21-25, 1996. Volume 2, pg. 770.

[Briesmeister 1993] J.F. Briesmiester, ed., MCNP -- A General Monte Carlo N-particle Transport Code, Version 4A. Los Alamos National Laboratory. LA-12625-M.

[Lee *et. al.* 1996a] S.R. Lee, J.C. Cummings, S.D. Nolen, Some C++ Classes for Monte Carlo Tallies. *X Division Research Notes*. XTM-RN(U)96-004.

[Lee et. al., 1996b] S.R. Lee, J.C. Cummins, S.D. Nolen, Building a Transport Code using POOMA and Object-Oriented Methods. *X Division Research Notes*. XTM-RN(U)96-003.

[Lewis & Miller 1984] E.E. Lewis, W.F. Miller, **Computational Methods of Neutron Transport**. 1984, John Wiley & Sons.

[Glasstone & Sesonske1994] S. Glasstone, A. Sesonske, **Nuclear Reactor Engineering**. 1994 Chapman & Hall.

[Nolen *et. al.* 1996] S.D. Nolen, S.R. Lee, J.C. Cummings, Adding Mesh Tracking Capability to MC++. *X Division Research Notes*. XTM-RN(U)96-019.

[Wilson & Lu 1996] G. Wilson, P. Lu ed. **Parallel Programming using C++**. 1996, MIT Press. (Chapter 14)