LA-UR- 96-4808

CONF-970342--1

Title:

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

Author(s):

Stephen R. Lee
Julian C. Cummings
Steven D. Nolen

Submitted to:

8th SIAM Conference on Parallel Processing for Scientific Computing, March 14-17, 1997
Minneapolis, MN

Los Alamos
NATIONAL LABORATORY

MC++: Parallel, Portable, Monte Carlo Neutron Transport in C++[*]

Stephen R. Lee[†] Julian C. Cummings[‡] Steven D. Nolen[**]

Abstract

We have developed an implicit Monte Carlo neutron transport code in C++ using 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, clustered SMPs, and individual workstations. It contains appropriate classes and abstractions for particle transport and parallelism. Current capabilities of MC++ are discussed, along with future plans and physics and performance results on many different platforms.

1 Description of the General Problem

We begin with a time-implicit formulation of the sourceless transport equation, given by

$$[\hat{\Omega} \bullet \vec{\nabla} + \Sigma_t(\vec{r}, E)]\psi(\vec{r}, \hat{\Omega}, E) = \int dE' \int d\Omega' \Sigma_s(\vec{r}, E' \to E, \hat{\Omega}' \bullet \hat{\Omega})\psi(\vec{r}, \hat{\Omega}', E') + \chi(E)\int dE' \upsilon(E')\Sigma_f(\vec{r}, E')\int d\Omega' \psi(\vec{r}, \hat{\Omega}', E') \quad (1)$$

where $\hat{\Omega}$ represents the direction of travel and $\psi(\vec{r}, \hat{\Omega}, E)$ is the angular flux. Σ_s is the scattering cross section, Σ_f is the fission cross section, and Σ_t is the total cross section. The mean number of fission neutrons produced by an incident neutron of energy E is given by $\upsilon(E)$, and $\chi(E)$ is the energy spectrum of fission neutrons.

A system containing fissile 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 population will either increase or decrease exponentially in time. Systems are said to be *subcritical* if the population decreases, and *supercritical* if the population increases.

To treat this criticality problem, MC++ implements an auxiliary eigenvalue, *k*, and the average number of fission neutrons per event, υ, is replaced by υ/k. Using the eigenvalue and re-writing Eq. (1), we have

* This work performed under the auspices of the Department of Energy at Los Alamos National Laboratory.

† Transport Method Group, Applied Theoretical and Computational Physics Division, Los Alamos National Laboratory, P.O. Box 1663, MS B226, Los Alamos, NM 87545 (srlee@lanl.gov).

‡ Advanced Computing Laboratory, Computing, Information, and Communications Division, Los Alamos National Laboratory, P.O. Box 1663, MS B287, Los Alamos, NM 87545 (julianc@lanl.gov).

** Department of Nuclear Engineering, Texas A\&M University, College Station, Texas, 77841 (nolen@trinity.tamu.edu).

$$[\hat{\Omega} \bullet \vec{\nabla} + \Sigma_t(\vec{r}, E)]\psi(\vec{r}, \hat{\Omega}, E) = \int dE' \int d\Omega' \Sigma_s(\vec{r}, E' \rightarrow E, \hat{\Omega}' \bullet \hat{\Omega})\psi(\vec{r}, \hat{\Omega}', E') +$$
$$\frac{\chi(E)}{k} \int dE' \upsilon(E') \Sigma_f(\vec{r}, E') \int d\Omega' \psi(\vec{r}, \hat{\Omega}', E') \quad (2)$$

In this formulation, k can be adjusted to obtain a time-independent solution to Eq. (2). It is this *effective multiplication factor, k,* that is computed by MC++.

2 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++ stores this description along with isotopic information supplied by the user to fully describe the materials and composition of the problem. Eq. (2) is then solved in a simple iterative scheme that is widely employed in other transport codes [1] 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*). Particles in the next cycle are started *isotropically* at the location at which the birth took place.

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 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. 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 same as employed in the transport code MCNP [1].

3 POOMA

MC++ is written in C++ and uses the POOMA class library. 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 [2]. 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 [3]. It is an integrated and layered system, in which

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

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.

3.1 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.

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 [2], [4].

3.2 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 [5], [6], [7], [8]. 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 (e.g., updating particle positions). 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).

4 Physics and Performance Results
In this section, physics and performance results from some simple test problems are shown.

4.1 Preamble
Before discussing the results, some background information is needed on the test problems run and the platforms used for testing.

4.1.1 Platforms. MC++ was tested on a wide variety of platforms and, where possible, tested on multiple nodes of these platforms. Table 1 contains a list of tested platforms.

Table 1: Tested Platforms

Platform Abbreviation	Description
SGI64_MPI	64-bit SGI R10000 (multiple heads)
SGI5	32-bit SGI Cluster
RS6K	IBM RS6000 Cluster
T3D	Cray T3D
TFLOP	ASCI Red Intel Teraflop
SUN4SOL2	Sun Sparc10, Solaris 2.5
SGIMP	32-bit SGI R8000 (multiple heads)

Because the MC++ application was written for a specific program (the Accelerated Strategic Computing Initiative, or ASCI), many of the later tests were run only on hardware relevant to the ASCI program. This hardware includes the SGI64_MPI and TFLOP. Preliminary tests of MC++ were made on all of the hardware in Table 1.

Note also from the table that platforms shown in bold have parallel capability. The abbreviations in Table 1 are used throughout this paper.

4.1.2 Test Problems. Two test problems were used to exercise this version of MC++. These problems consist of an infinite medium problem constructed such that *k* is exactly 1, and a simple *double density godiva* problem, which is a well known transport benchmark problem with well understood criticality characteristics [9].

The Godiva 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. A variety of tests were performed on meshes of different resolutions and differing characteristics. Results are shown for a 32x32x32 mesh (32,768 cells), a 128x128x128 mesh (2,097,152 cells), and a 256x256x256 mesh (16,777,216 cells).

4.1.3 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 POOMA, the concept of a virtual node is used, which is a useful abstraction that leads to architecture independence, load balancing, and so on. In this paper, virtual nodes are used interchangeably with physical nodes.

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 code was compiled and run in parallel without incident on all platforms shown in Table 1. 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).

4.2 Physics Results

Figure 1 shows representative results for the infinite medium problem. Note the excellent convergence to the expected result of k = 1.0000. The same convergence was noted on all platforms, in both serial and parallel, in Table 1. The problem was run with 40,000 particles per cycle for a total of 10 cycles..

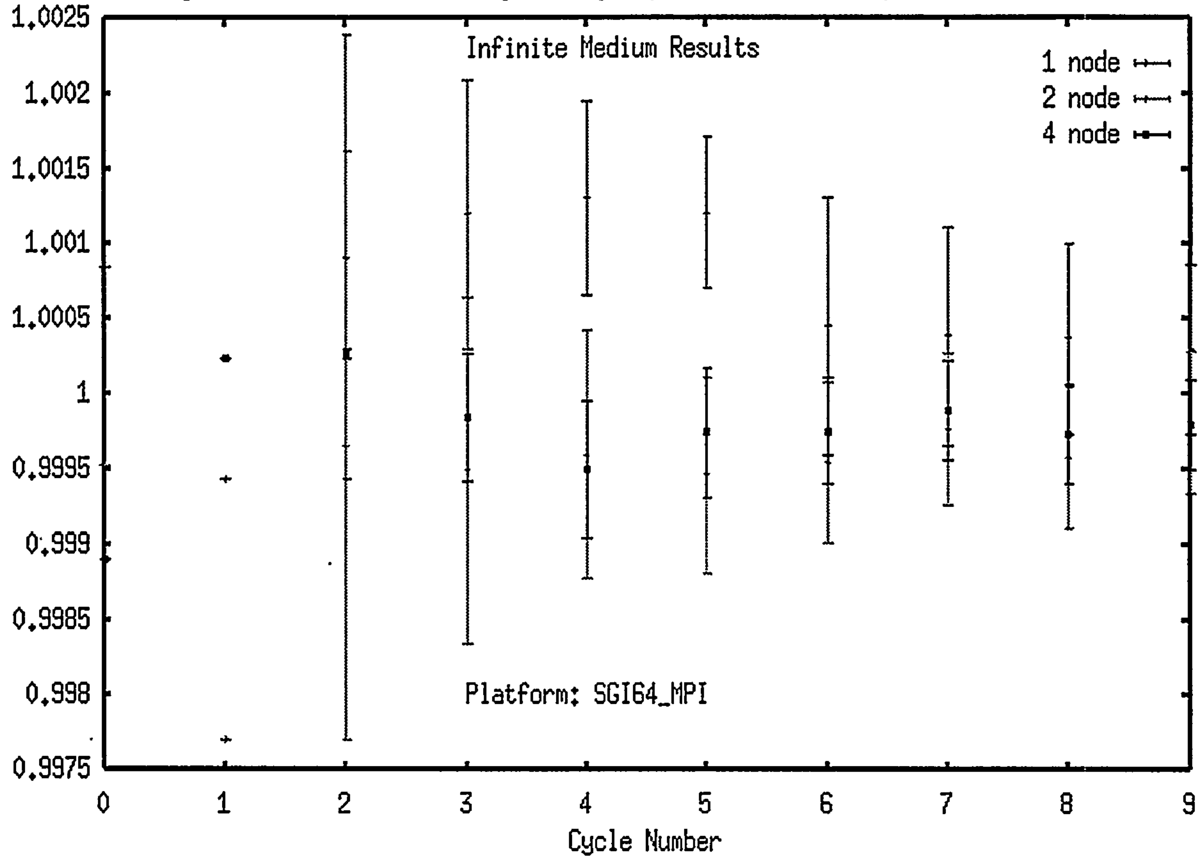

Fig 1. *Collision estimator for k-effective v.s. cycle number. This excellent convergence to k = 1.0000 was seen on all platforms in Table 1.*

Figure 2 shows the representative results of the MC++ calculation on all three meshes, along with the MCNP result on the actual spherical geometry. Because MCNP does not support the concept of a computational mesh, these results are shown for an actual sphere of the proper radius. The meshes that MC++ computes on are approximations to this sphere, the higher resolution meshes being the better approximations.

These problems were all run with 40,000 particles per cycle for a total of 30 cycles. The first 5 of these cycles were skipped to allow the neutron distribution to stabilize before accumulating the tallies.

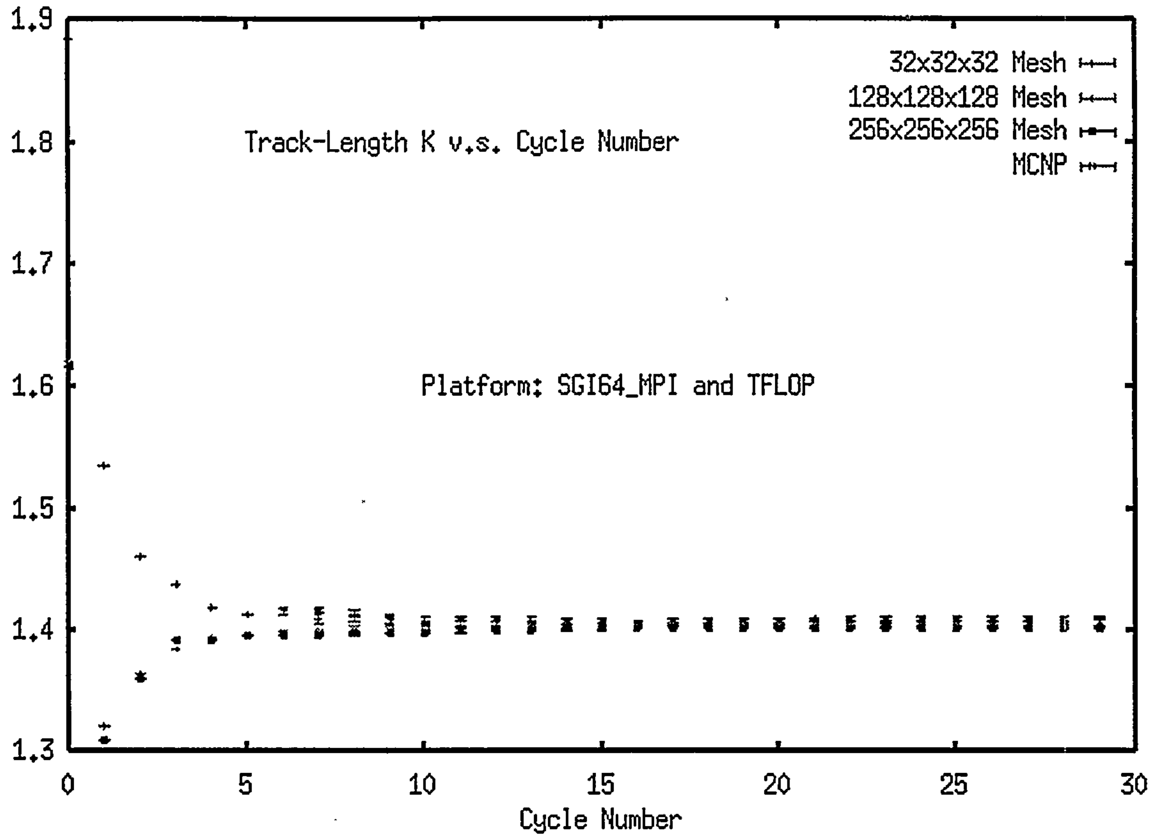

Fig 2. *K-Effective estimator for different mesh resolutions compared to the MCNP result. Note the excellent convergence to the expected result of k ~ 1.4.*

MCNP is a well-known and widely used particle transport code that has been around for decades. Version 4B was used to benchmark the MC++ physics results.

Note the good agreement with MCNP shown in the figure. The *k* for this configuration should be approximately 1.4.

Similar results for the Godiva problems were seen on all tested platforms.

4.3 Performance Results

Figure 3 shows parallel performance on all platforms for MC++ running the infinite medium problem. Note the reasonable speedups as the number of processors applied to the problem increases.

Figure 4 shows the parallel performance for the 16 million mesh cell godiva problem on ASCI-relevant hardware. Similar speedups were noted for the smaller meshes as well.

As can be seen in Figure 3 and 4, 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.

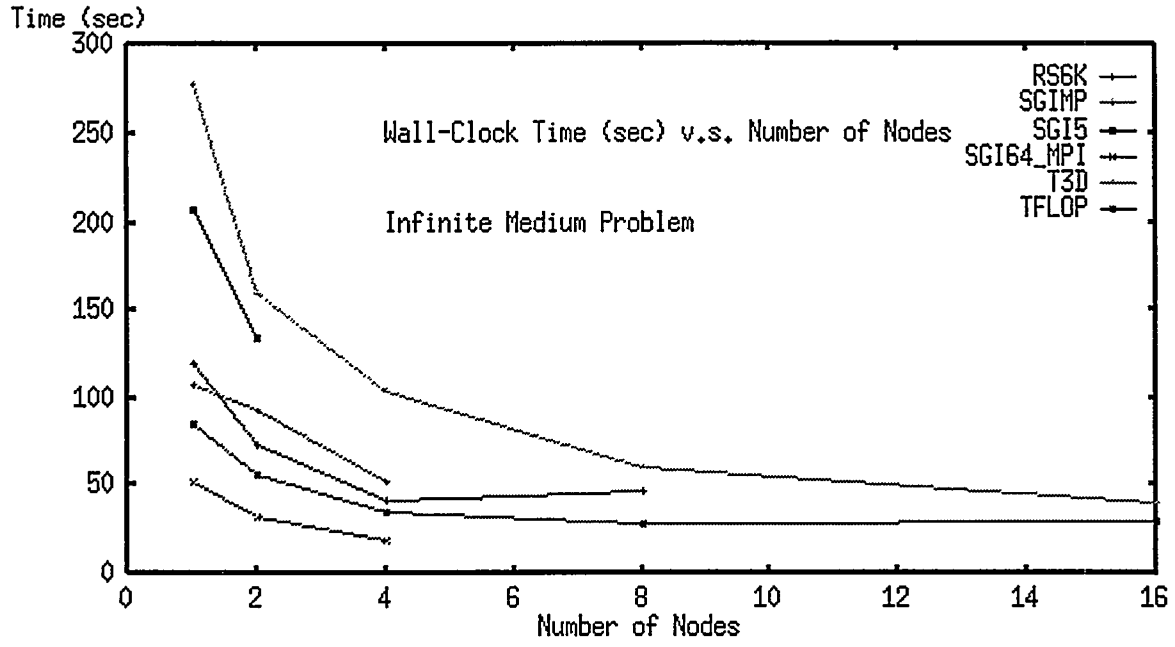

Fig 3. *Parallel performance for the infinite medium problem.*

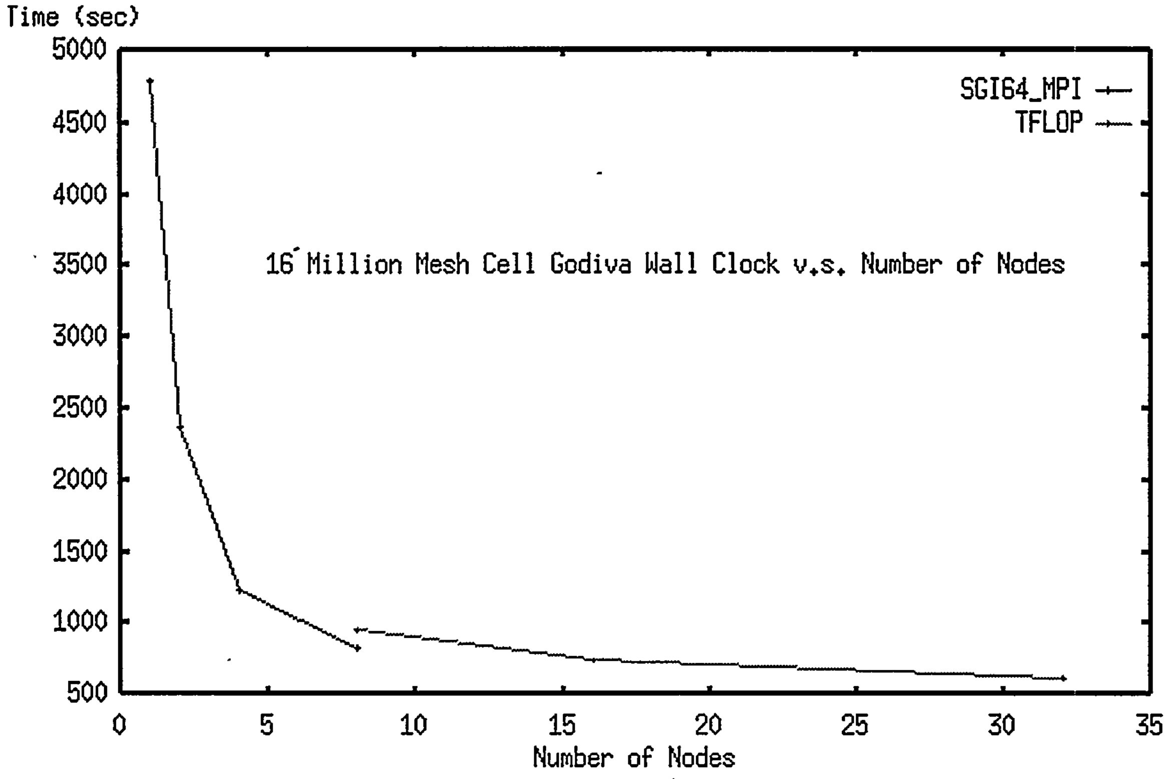

Fig 4. *Parallel performance for the 16 million mesh cell Godiva problem. TFLOP results on fewer than 8 nodes not possible due to memory constraints.*

5 Future Work

Beyond this 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. 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.

6 Conclusions

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

[2] G. Wilson, P. Lu ed. Parallel Programming using C++. MIT Press, 1996. Ch. 14.

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

[4] See http://www.acl.lanl.gov/PoomaFramework.

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

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

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

[8] See http://www-xdiv.lanl.gov/XTM/srlee/PROJECTS/MC++.

[9] G. Bell, S. Glasstone, Nuclear Reactor Theory. Krieger Publishing, 1970. pp 243-245.