CONF-971218--

LA-UR- 97-2399

Approved for public release; distribution is unlimited.

Title: MC++ AND A TRANSPORT PHYSICS FRAMEWORK

Author(s): STEPHEN R. LEE
JULIAN C. CUMMINGS
STEVEN D. NOLEN
NOEL D. KEEN

Submitted to: INTERNATIONAL CONFERENCE ON COMPUTATIONAL PHYSICS, MARINA DEL REY, CA, DECEMBER 15-19, 1997

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

MASTER

Los Alamos
NATIONAL LABORATORY

MC++ and a Transport Physics Framework

Stephen R. Lee
Julian C. Cummings
Los Alamos National Laboratory
Los Alamos, New Mexico USA 87545

Steven D. Nolen
Texas A&M University
College Station, Texas USA 77804

Noel D. Keen
University of New Mexico
Albuquerque, New Mexico USA 87131

1 Motivation

The Department of Energy has launched the Accelerated Strategic Computing Initiative (ASCI) to address a pressing need for more comprehensive computer simulation capabilities in the area of nuclear weapons safety and reliability. In light of the decision by the U. S. Government to abandon underground nuclear testing, the Science-Based Stockpile Stewardship (SBSS) program is focused on using computer modeling to assure the continued safety and effectiveness of the nuclear stockpile. Doing this will require major advances in computing speed and the complexity of computational models. Thus, it is anticipated that more modern software development approaches will be needed in order to accelerate prototyping of new computer applications, facilitate the integration of different physics models, manage the complexity of simulation codes, and port the simulation codes to new computational platforms.

We believe that the utilization of object-oriented design and programming techniques can help in this regard. Object-oriented programming (OOP) has become a popular model in the general software community for several reasons. It can help manage a software application by encouraging code designers to break the application down into constituent *objects* and their interactions. This leads to a natural decomposition of a software project into manageable pieces. Often, these pieces may have multiple uses within a given project or may be reused in subsequent projects, thus shortening the time for application development. Finally, object-oriented programming provides for very explicit interfaces between relatively self-contained objects, which typically makes adding new code or extending an existing application more straightforward.

These benefits transfer very well to the realm of high-performance physics modeling and rapid application development required for ASCI. Many people will be lending their expertise to ASCI application development, and OOP will help them work together more effectively. The simulation codes produced will need to be flexible and agile, quickly adapted to the newest high-performance computing platforms as they become available. Moreover, these codes must be written in a high-level, easily understood fashion, so that the critical physics knowledge embedded in the models is not lost as codes are passed along from one researcher to the next. OOP can help address these important aspects of ASCI code development.

MC++ is a specific ASCI-relevant application project which demonstrates the effectiveness of the object-oriented approach. It is a Monte Carlo neutron transport code written in C++. It is designed to be simple yet flexible, with the ability to quickly introduce new numerical algorithms or representations of the physics into the code. MC++ is easily ported to various types of Unix workstations and parallel computers such as the three new ASCI platforms, largely because it makes extensive use of classes from the Parallel Object-Oriented Methods and Applications (POOMA) C++ class library. The MC++ code has been successfully benchmarked using some simple physics test problems, has been shown to provide comparable serial performance and a parallel efficiency superior to that of a well-known Monte Carlo neutronics package written in Fortran, and was the first ASCI-relevant application to run in parallel on all three ASCI computing platforms.

2 Algorithm

MC++ currently computes the k and α eigenvalues of the neutron transport equation. These eigenvalues are used to assess criticality of a system containing fissile material. Such a system is said to be *critical* if there is a self-sustaining, time-independent chain reaction in the absence of an external source of neutrons. In a critical system, the rate of neutron production via fission is just equal to the rate of loss due to absorption and leakage from the system. If there is no such equilibrium, then the neutron population will either increase (a *supercritical* system) or decrease (a *subcritical* system) exponentially in time. Each of the aforementioned eigenvalues treats this criticality problem in a slightly different way, but each provides valuable information on the criticality of the system in question.

MC++ is currently written to perform its calculations on a three-dimensional Cartesian mesh. Information on the dimensions of the mesh, the types of materials contained in the system, and material densities in each mesh cell are obtained from another simulation code and read in to MC++. It stores this information along with a database of isotopic information supplied by the user to describe the materials in the problem. Then, the Monte Carlo calculation is begun by loading neutrons into cells containing fissile material in a "round-robin" fashion. The neutrons are tracked through the system, undergoing collisions with isotopes that compose the materials in each mesh cell and boundary interactions with the mesh itself. Collisions that result in a fission event produce new neutrons which are stored as source points for the next *generation* or *cycle*. All particles are tracked until they trigger a fission event, are absorbed, or escape from the problem. Once all particles in the current generation have completed their tracks, the next generation is begun. During each cycle, MC++ accumulates *tallies* of specific events and uses this information to compute estimates of k or α.

3 Implementation

The MC++ code is largely written in an object-oriented, data-parallel style, and it makes extensive use of classes from the POOMA framework. POOMA is a C++

class library designed to encapsulate the details of parallel programming and provide a set of high-level, physics-based data structures and algorithms with which to build scientific applications. In object-oriented programming, a general structure or outline for a set of codes intended to address a certain class of problems is known as a *framework*. The basic aim of the POOMA class library is to be an object-oriented framework for scientific computing on parallel architectures. Using this framework, we have built a simple and efficient Monte Carlo neutronics package.

The most heavily used POOMA feature in MC++ is its particle simulation capabilities. POOMA provides a `ParticleBase` class with a minimal description of a particle population. The user (in this case, the MC++ application developer) then derives a class from `ParticleBase` and adds members to describe the particle characteristics, such as velocity or mass, needed for this simulation model. This method allows for an extremely flexible particle description. Nevertheless, the `ParticleBase` class provides the derived class with many powerful features, including automatic decomposition of the particle data across processors, the ability to add or delete particles from the particle population at will, and a convenient *array syntax* for performing data-parallel operations involving particle attributes.

MC++ also makes use of the `Field` and `Mesh` types in POOMA. It stores the mesh description inside a POOMA `Cartesian` object, which can represent a non-uniform Cartesian mesh. The `Cartesian` class provides a translation from "index space" to the physical domain of the simulation and simplifies the calculation of the distance to the nearest mesh cell boundary performed in MC++. POOMA `Field` objects are used to hold information on the type and density of each material in each mesh cell. The data is automatically distributed across processors, so that only a portion of the data of each `Field` is owned by each processor. This allows MC++ to perform simulations on very large meshes with a relatively small number of processors. POOMA has a data layout option which will maintain data for each particle on the same processor that owns data for that portion of the field near the particle's current location. Maintaining this *data locality* allows for much more efficient computations and lookups of field data. Although MC++ mainly uses the POOMA `Field` as a storage class, it has many other powerful features, such as built-in boundary conditions, array syntax for writing expressions involving `Fields`, and optimized differential operators like `div()` and `grad()` for `Fields` residing on a specific type of POOMA `Mesh`.

A few other general features of POOMA that add value to the MC++ application are worthy of mention. POOMA manages virtually all of the parallelism and other "computer science" issues that arise in getting this neutronics code to run efficiently on various architectures. The only explicit message passing in MC++ is done in the `Tally` classes to gather event tallies across all the processors. (The `Tally` classes were built specifically for Monte Carlo simulation and were not provided by POOMA.) Everything else, including the maintenance of particle data locality and coordination of operations on the distributed particle data, is done transparently by POOMA. This allows the vast majority of MC++ to be written using high-level objects in an easily understood syntax that elevates the physics content while hiding the computer science details. Furthermore, the extensive use of C++ templates and the *expression template* technique in POOMA helps the compiler to produce highly optimized code and provides a high-level application development syntax with little or no sacrifice in application performance.

4 Code Results

The transport eigenvalue estimates produced by MC++ have been benchmarked for accuracy against the MCNP code for a set of simple test problems. MCNP is a code written in Fortran 77 that uses PVM for message passing on parallel computers and employs an analytic description of materials in the simulated system rather than a mesh description. MC++ provides eigenvalues within statistical error of those provided by MCNP for the standard "godiva" problem of a bare uranium sphere. The code performance of MC++ has also been compared with MCNP on a variety of computing platforms, including several types of Unix workstations and the Cray T3D. In order to make the comparison with MCNP fair, mesh cell boundary crossing events were turned off in MC++ (collisions only) and an "infinite medium" test problem was chosen (no material interfaces). Code performance was roughly comparable on the various serial platforms. MC++ exhibited vastly superior parallel efficiency on multiple processors of the T3D, the only modern parallel computing platform on which MCNP was readily available. MC++ has been rapidly ported to many different types of parallel computers, including the Intel Teraflops machine (ASCI Red), an SGI Origin 2000 cluster (ASCI Blue Mountain), and the IBM SP2 (ASCI Blue Pacific). It shows relatively good parallel efficiency (within 70-80% of linear speedup) on all platforms, with no special performance tuning on any platform.

5 Transport Physics Framework

One area of continuing development in the MC++ project is the generalization of the MC++ code into a Transport Physics Framework (TPF). Just as the POOMA framework has been quite useful in accelerating code development and promoting code reuse across applications, a TPF could aid in the development of new transport physics codes and facilitate testing of novel techniques. What is needed is a general outline of the process of transport simulation and a set of tools that could be directly employed or readily modified for a specific simulation need.

For example, MC++ currently uses a set of cross-section classes to encapsulate the type of cross section employed during transport, and to make the reading of different cross-section libraries, the storage of the data, and the retrieval of required data during transport simple and general. Another example is the set of various particle sourcing methods that are currently under development, once again making extensive use of templates. Once complete, the addition of a new sourcing method for particle energy, direction, or position will be completely trivial. It is extremely useful to have these and other such methods readily available in the form of interchangeable modules or classes with a common interface. Such a "plug and play" capability will greatly simplify the task of evaluating the effectiveness and utility of the various algorithms. This same sort of generality would be beneficial in other areas of MC++, such as the modeling of neutron capture events and the representation of material isotopic information. Reformulating MC++ in terms of a TPF will provide the ASCI project and other researchers with a much more flexible tool for the exploration of alternative techniques and models in neutron transport studies.