Parallelization of Geant4 Using TOP-C and Marshalgen

Gene Cooperman Viet Ha Nguyen Igor Malioutov
College of Computer and Information Science
Northeastern University
Boston, MA 02115
Email: {gene,vietha,imltv}@ccs.neu.edu

Abstract

Geant4 is a very large, highly accurate toolkit for Monte Carlo simulation of particle-matter interaction. It has been applied to high-energy physics, cosmic ray modeling, radiation shields, radiation therapy, mine detection, and other areas. Geant4 is being used to help design some high energy physics experiments (notably CMS and Atlas) to be run on the future large hadron collider: the largest particle collider in the world. The parallelization, ParGeant4, represents a challenge due to the unique characteristics of Geant4: (i) complex object-oriented design; (ii) intrinsic use of templates and abstract classes to be instantiated later by the end user; (iii) large program with many developers; and (iv) frequent releases. The key issue for parallelization is not just how to parallelize "correctly" but also how to parallelize "with minimum effort". In addition, the parallelization should make as few assumptions about the source code as possible, due to the frequent release schedule of Geant4. We use TOP-C (Task Oriented Parallel C/C++) for parallelization and Marshalgen for marshaling/serialization. In some examples on a cluster of 100 nodes yielded a speedup of up to 94.4. The code's portability, scalability and performance are also discussed.

1 Introduction

Geant4 is a toolkit for particle-matter simulation using modern object-oriented design principles [9]. It contains about a million lines of C++ code. The development of Geant4 is coordinated at CERN and includes participation from more than 100 scientists and from more than ten national high energy physics laboratories in Europe, Russia, Japan, Canada and the United States. The design of Geant4 dates from the mid-90s. After the first production release, the collaboration was formalized in January, 1999 as the Geant4 Collaboration.

Geant4 has components to model the geometry, the materials involved, the fundamental particles of interest, the generation of primary particles for new events, the tracking of particles through materials and external electromagnetic fields, the physics processes governing particle interactions, the response of sensitive detector components, the generation of event data, the storage of events and tracks, the visualization of the detector and particle trajectories, and the capture for subsequent analysis of simulation data at different levels of detail and refinement [1, 2].

1.1 Why parallelizing Geant4 is important

Geant4 has a community of approximately one hundred developers from high energy physics research centers around the world including CERN (European Organization for Nuclear Research), KEK (High-Energy Accelerator Research Organization of Japan), and SLAC (Stanford Linear Accelerator Center). Geant4 also supports thousands of scientific users around the world. This community also holds an annual Geant4 workshop. A large simulation application using Geant4 runs over weeks, months, or years.

Therefore, it is vital to parallelize Geant4 to reduce the running time of the software and thus shorten the cycle of experiments. There are several examples of the successful use of ParGeant4 (this parallelization of Geant4). One example is an ESA study [11] using a modified ParGeant4 to parallelize MULASSIS [13], an application to study radiation shielding. Another example is its use (jointly with A.S. Howard) in a Geant4 simulation for an experiment to detect Dark Matter in the universe [10, 17]. Additionally, the parallelization of Geant4 has been demonstrated to run on the Computational Grid [6].

1.2 Why parallelizing Geant4 is difficult

Parallelizing Geant4 is difficult due to the characteristics associated with a large software package. There are anecdotes of at least two previous attempts to parallelize Geant4, using MPI. Both attempts had failed.

Parallelizing involves two tasks: (i) parallelizing the computation-intensive run loop inside the Geant4 library, and (ii) adapting the application data structures to the requirements of the parallel framework. These two tasks correspond to:

1. parallelizing control structures (implemented by TOP-C [5] in this paper).

2. marshaling objects (implemented by Marshalgen [7, 8] in this paper).

Included in the parallelization of control structures are the following three issues: identifying which parts of code can be executed in parallel, identifying shared data and task-local data, and resolving the dependencies among parallel tasks. *Marshaling* is the conversion of data structures and C++ objects in memory to buffers, and vice versa. This is essential for exchanging data via the network in distributed-memory parallel computing. Marshaling has also been popularized in Java under the name "serialization".

The difficulty of parallelizing Geant4 does not lie in the *ability* to do the above two tasks, but in the *human effort* to do them. The issues may be summarized as follows.

1. Geant4 is a moving target. New versions of Geant4 routines are released every six months.

2. Geant4 is called from an end-user's main routine, and many Geant4 routines (e.g. abstract classes, templates, hook functions, etc.) are instantiated or shadowed by end-user routines. Any parallelization strategy must take into account such future end-user code.

3. Geant4 is large. Most users download the Geant4 libraries directly. Any parallelization must be compatible with the Geant4 binary libraries that are semi-yearly updated.

4. Geant4 data structures are complex. They involve compound, and even recursive data structures. Hand-coding the marshaling routines would be error-prone.

5. Geant4 makes references to a user-defined data structure. The user extends Geant4's G4Hit data structure in an application-dependent manner. A collection of G4Hits is generated on the slave, and must then be passed to the master. The end user must be given simple instructions for creating marshaled versions of his or her G4Hits class.

1.3 The ParGeant4 Approach

To maintain compatibility across Geant4 versions, a new ParRunManager class is distributed in source form. Thus end users first install the Geant4 binary libraries. Then, following the instructions of the example/extended/parallel/ directory of the Geant4 distribution, they download TOP-C (Task Oriented Parallel C/C++), and use it to compile the ParGeant4 code to create one additional class, ParRunManager. The end user must then download Marshalgen, and add annotations (comments) to the Hit class used in their Geant4 simulation. Those annotations are used by Marshalgen to create a MarshaledHit class. They then modify their "main" routine to create a ParRunManager, and then link with the Geant4 libraries, and begin their simulation. See the ParGeant4 web page [15] for further details.

1.4 Novel Features

To minimize the human effort, we use two automatic tools: *TOP-C* for parallelizing control structures, and *Marshalgen* for marshaling. Using these software packages, we were able to parallelize the Geant4 library with about 200 lines of additional code. Based on this, any user-customized Geant4 application can be parallelized with a few additional lines of simple annotations in the declaration files (e.g, .h files) for the customized data structures.

1.5 Background

ParGeant4 Work on ParGeant4 [15] was begun in 1999 [3, 4], initially using track-level parallelism (a task simulates a particle track). This was later changed to event-level parallelism, and all modifications to the original source code of Geant4 were eliminated. The issue of marshaling Geant4 data structures later motivated the development of Marshalgen, beginning in 2002 [7, 8]. ParGeant4 also uses the TOP-C (Task Oriented Parallel C/C++) package [5] for easy parallelization.

Marshalgen and other Marshaling Packages Previous well-known marshaling systems include rpcgen [18], Corba IDL [14], and Java serialization [16] as part of the Java RMI (Remote Method Invocation) facility.

A common problem with most of the above packages is the difficulty of recursively marshaling pointers to other objects, and deciding which pointers not to marshal. This issue was addressed in version 1 of Marshalgen [7]. Version 2 of Marshalgen [8] enhanced its capabilities to easily handle object-oriented issues (templates, polymorphism, inheritance and private data members), as needed in Geant4.

1.6 Outline of the Paper

Section 2 describes the Geant4 software package. The methods for parallelizing Geant4 are explained in Section 3. The methods for marshaling are described in Section 4. The performance results are given in section 6.

2 Overview of Geant4

As a toolkit, Geant4 includes only libraries, similarly to many other scientific subroutine libraries, such as LinPACK [12]. It is up to the application writer to write a `main` routine. Inside the `main` routine, the application writer will set up the necessary parameters for the simulation such as: the types of particle to simulate; materials; geometry; electromagnetic fields; etc. The application writer then makes a function call to the method of Geant4 that starts the simulation.

Unlike LinPACK, the Geant4 routines call application-defined routines to set up the geometries, determine what experimental data to store when a particle strikes a detector (G4Hit), etc.

2.1 Structure of a Geant4 Simulation

Before discussing any further about the simulation done by Geant4, we examine how particle-matter interaction is modeled within Geant4. An experiment (also denoted as a *run*) is a sequence of *events*. An event is the entrance of an external particle into the region being modeled. The particles that trigger events are called the *primary particles*. The primary particles come, for example, from an accelerator or from the cosmic rays that enter the region by incident.

Physical processes are continuous in time and space. However, the simulation is computed in a discrete manner. A *step* is the smallest discrete unit in space for doing a computation. All physical characteristics are assumed to be constant during a step. Based on the status of a particle at the end of a step, one or more *physical processes* are called to calculate the new physical characteristics of the particle at the next step (location, momentum, energy, etc.).

A primary particle is modeled as moving in steps according to one or more physical processes (e.g. electric, magnetic, and/or gravitational fields). At each step, a primary particle has some probability of decaying into one or more *secondary particles* (through radioactive decay, interaction with matter, etc.). These secondary particles may in turn generate other secondary particles at a later step. Therefore, a single primary particle with a high enough energy level may trigger a cascade of interactions and eventually generate a large number of secondary particles.

A *track* represents the path traversed by a particle (either primary or secondary) from the time that it is generated until it decomposes or hits a *detector*.

The outcome of the experiment, the *collection of hits*, is collected by numerous *detectors* located inside the collider. A *hit* is generated when a particle strikes a detector. Depending on the type of the detector, the new hit may contain various information, such as particle type, energy level, momentum, etc. The hit is stored in a hit collection associated with the current event.

To write a simulation for such an experiment, the application writer needs to specify the characteristics of the collider: geometries; materials; electromagnetic fields; the type and initial momentum and other characteristics of the primary particles; the number of primary particles to simulate; the types and the geometries of the detectors; data to be stored in a hit; etc. Given these parameters, the Geant4 library will perform the simulation and return the result, which is the information about the hits.

Optionally, Geant4 can also be used to drive a separate analysis package. The analysis package will select data to store in an output file (a histogram). Other parts of the analysis package are later called to further process and visualize the data.

2.2 Code Design

Hereafter, we will distinguish between *"the Geant4 library"* and *"a Geant4 application"*. A Geant4 application additionally includes the user-defined "`main`" routine and any user-defined data structures and functions. This is linked to the Geant4 library, which contains the simulation routines.

The routines for a simulation are contained inside the Geant4 library. However, application writers will usually define additional internal data structures. For example, the application writers may define the classes representing hits, which contain additional information about the hits they would like to collect. Moreover, the application writers may add their own hook functions, which are triggered during various phases of the simulation to collect additional types of information.

We are particularly interested in the Geant4 classes for `Event` and for `Hits`. Events and hits are the input and the output of parallel computing tasks and therefore need to be exchanged over the network. The class hierarchies in the `Event` and `Hits` categories determines how complex the marshaling task will be when we want to send an event or a hit collection over the network.

3 Parallelization

3.1 Tool Used: TOP-C

The TOP-C constructs of TOP-C allow the end user to parallelize a sequential program while modifying relatively few lines of original source code. This is useful for very large programs that are likely to pass through frequent version changes.

TOP-C uses a master-slave topology. This topology may map onto a distributed architecture, shared memory architecture, or some other architectures. A single TOP-C application may use a distributed memory model (message-based for clusters, Globus protocols for Grid) or a shared memory model (thread-based for a multiprocessor computer), simply by linking with the appropriate TOP-C library.

A *task* is a fragment of code that takes a *task input* and produces a *task output*. Each task is given to a slave process for execution. In the context of ParGeant4, the task input is an event (generation of a primary particle), and the task output is a collection of hits.

In TOP-C, the user describes a task using callback functions. These callback functions are passed to TOP-C as function pointers. In a more object-oriented style, these callback functions would be thought of as abstract methods of an abstract class (or Java interface). In other words, the user presents a task to TOP-C via "an interface" (callback functions). TOP-C treats the callback functions as opaque routines and call them whenever it needs to access the input, output, or the shared data of the tasks. Consequently, there are four callback functions:

1. GenerateTaskInput() $\rightarrow$ taskOutput;
2. DoTask(taskInput) $\rightarrow$ taskOutput;
3. CheckTaskResult(taskInput, taskOutput) $\rightarrow$ TOPCaction; and
4. if (TOPCaction == `UPDATE_SHARED_DATA`) UpdateSharedData(taskInput, taskOutput).

In ParGeant4, TOPCaction is always `NO_ACTION`. UpdateSharedData and actions other than `NO_ACTION` are included in the TOP-C model in order to handle non-trivial parallelism.

For more details on TOP-C, see [5, 6]. An example of how Geant4 is parallelized using TOP-C is presented in Section 3.2.

3.2 Implementation Details: Parallelizing events using TOP-C.

In the Geant4 library, the simulation process is controlled by the class `G4RunManager`. We implement the parallel simulation process by extending the class `G4RunManager` to a new class `ParRunManager`. The sequential computation logic in the method `G4RunManager::DoEventLoop` is overridden by the parallel version in `ParRunManager::DoEventLoop`. Each iteration of this main loop corresponds to the simulation of an event.

To perform a parallel computation, the application writer need only construct and invoke in the `main` routine the class `ParRunManager` instead of the class `G4RunManager`. Then, the parallelization of the simulation process is thus transparent to the application writer. Note that the class `ParRunManager` can be provided independently from the Geant4 library. This approach allows the application writers to parallelize a Geant4 application while using the unmodified binary distribution of the Geant4 library.

In order to ensure that the code `ParRunManager` closely mirrors the "for loop" structure of `G4RunManager`, we use TOP-C's *raw constructs*. In effect, instead of using the TOP-C callback function `GenerateTaskInput`, the application writer arranges to submit new tasks by directly calling the TOP-C function `raw_submit_task_input(TOPC=MSG(...))` from inside the original loop of `G4RunManager`.

Object marshaling/unmarshaling is handled orthogonally to this parallelization. By design, TOP-C requires no knowledge about the data structures of the application. It sees all data as a buffer of specified size in memory. It assumes that it is the user's responsibility to convert objects to buffers and vice versa.

3.3 Issues in Parallelization of Geant4

3.3.1 Random number generation

To simulate stochastic physical models, Geant4 uses a random number generator. The state (or the seed) of the random generator before an event depends on the number of times the random generator was called during the computation of the previous event. This number of calling times, unfortunately, is hard to predict because the simulation computation is a randomized computation.

The above phenomenon of random generator seed creates a "virtual" dependency among events. The computation of the i^{th} event (where i is an arbitrary natural number) depends on the random seed resulted from the computation of the $(i-1)^{th}$ event. This causes a problem in the parallelization of Geant4: the results of events computed in parallel may differ from those computed sequentially.

In order to be able to compute two events (for example, the $(i-1)^{th}$ and i^{th} events) in parallel, and still produce output identical to that of the sequential version, one must be able to predict the random seed after the $(i-1)^{th}$ event without waiting for the computation of $(i-1)^{th}$ event to complete. This prediction is possible in theory. However, there is no general algorithm for such prediction in practice. In addition, some random generators may be provided as a "black-box" in the format of an external binary library, making the access to the logic of the random generator impossible.

Nevertheless, a simple strategy yields high quality statistical results for the parallelized version. In addition to the random number generator of the sequential application, an independent random number generator is included in the parallel version. We use the "parallel random number generator" to set the random seed prior to each event to a random number. The random seed is generated on the master and passed to the slave as part of the task input for that event. On the slave, this provides a seed for the standard "sequential random number generator". The seeds are generated by using an independent random generator from the one used by the sequential Geant4 application. This approach of parallelization produces outputs which are not identical to the output of the sequential version. However, from the statistical point of view, the data produced by the sequential version and the randomized seeds of the parallel version should have identical statistical characteristics.

The parallel use of random seeds has the further advantage that the parallel code can produce identical results regardless of whether there is one slave process or more. This is the case because the events of ParGeant4 are produced according to a predetermined sequence depending only on the seed for the "sequential random generator" on the master. Given fixed initial seeds for the sequential and parallel random number generators, these two seeds will uniquely determine the remainder of the parallel computation, independently of the number of slave processes and which process executes on which slave.

3.3.2 Event-Level Parallelism vs. Track-Level Parallelism

In some applications, such as air showers, a single event (e.g. cosmic ray) may give rise to very many secondary tracks. In such a case, a Geant4 simulation will often simulate only a single event. In such a case, track-level parallelism is preferred to event-level parallelism.

In such a situation, it is possible to get around this problem. One can simulate the single event until multiple secondary particles are produced. The secondary particles can then be considered as multiple events, and fed back into Geant4, to simulate using event-level parallelism. This has not been implemented.

3.3.3 Distributed Memory vs. Shared Memory Parallelism

Geant4 is parallelized using the distributed memory model. Each task is executed on a separate node with its own memory. Data is exchanged via network messages.

Shared memory parallelization is attractive for those simulations with short events and large hit data data structures (large ratio of communication to computation). TOP-C (version 2.5) provides an option for aggregation of multiple tasks into a single message. This can be used to alleviate the startup overhead of sending a network message.

Shared-memory parallelization is difficult due to the design of Geant4. Inside the Geant4 library, there is a global navigation module that keeps track of which secondary particle has been generated and has been simulated. For efficiency reason, this module is designed as a singleton (one instance for one memory space). If two tasks shares the same memory space, they will share the same global navigation module. The navigation module has its own internal states, so the sharing between parallel tasks will incur race conditions.

In order to do shared memory parallelization, the design of the Geant4 library must change the global navigation module from a singleton to a multiple-instantiable object, or introduce a concurrency mechanism (such as locks) on the module. Both approaches will result in an efficiency sacrifice for all Geant4 sequential applications. The benefit of being able to do shared memory parallel computing may not be large enough for such a sacrifice.

4 Marshaling and Unmarshaling of Complex Objects

4.1 Issue: Why we need Automatic Object Marshaling

The distributed memory model ofparallelism is chosen for Geant4, as explained in Section 3.3.3. The current version of Marshalgen primarily supports a homogeneous architecture, corresponding to most computing clusters. Marshalgen could easily be extended to support heterogeneous, at the cost of additional overhead.

One issue in doing distributed parallel computing is the ability to send data over the network. One should be able to *marshal* objects to a buffer and then *unmarshal* the buffer to reconstruct the same objects at a remote machine. The terminology "marshal" and "unmarshal" are also known as "serialize"and "deserialize" in Java. In C++, there is no such serialization mechanism. Therefore, in MPI or TOP-C, the user has to write marshaling code manually.

With a complex class hierarchy as in Geant4, writing marshaling code for all classes manually would be tedious. Moreover, since application writers usually define their own data structures and have the Geant4 library use them instead of the default data structures, writing marshaling/unmarshaling code for those customized data structures is an important issue. The application writers have to make sure that their marshaling/unmarshaling code of the customized classes is compatible with the marshaling/unmarshaling code of related classes in the Geant4 library. They may have to update their marshaling/unmarshaling code manually whenever the Geant4 library is upgraded. These tasks require huge effort in writing, testing and maintaining the marshaling/unmarshaling routines and the parallelized application.

Thus, the complex class hierarchy in such a large software package as Geant4 necessitates an automatic tool for generating marshaling/unmarshaling code. The next section is the discussion of such a tool: *Marshalgen*.

4.2 Tool Used : Marshalgen

Marshalgen is a semi-automatic tool for generating marshaling/unmarshaling code for C++ classes. The application writers use annotations to specify which data to be marshaled and how the data should be marshaled. These annotations should be next to the declaration of the data structures in .h files. Then, the application programmers run the annotated .h files through the Marshalgen preprocessor to generate marshaling/unmarshaling classes and methods, along with their declarations.

Marshalgen allows the application writers to generate marshaling/unmarshaling code without having to access the original source code (.cpp or .cc files). The annotations are added as comments, leaving the original program unchanged. Therefore, there is no need to access the original code to recompile the program. Only the declarations (.h files) are needed. This is useful for generating marshaling code for data structures from external libraries. This is because most libraries (include the Geant4 library) are usually distributed in pre-compiled binary code together with .h files.

The marshaling/unmarshaling code for a class `Foo` is generated as a class `MarshaledFoo`. The marshaling buffer for an object of type Foo is constructed by calling the constructor of the class MarshaledFoo with the target object as an argument. The class MarshaledFoo then provides necessary methods to access the constructed marshaling buffer. Unmarshaling is done in the following manner: the constructor of the class MarshalFoo is first called with the target buffer as an argument, then an object of type Foo is constructed from the buffer by making a call to the method MarshaledFoo::unmarshal.

Some of the possible Marshalgen annotations are listed in Table 1. For more details on Marshalgen, see [7, 8].

Default Annotations	Explanations
//MSH: primitive	For int, etc.; built-in marshaling
//MSH: primitive_ptr	For int *, etc.; built-in marshal.
//MSH: predefined	For prev. annotated class
//MSH: predefined_ptr	Ptr. to prev. annotated class
//MSH: array	Array of above type of elt.
//MSH: ptr_as_array	Ptr. to such an array

Table 1. Optional Annotations: one of five default cases, determined by parsing data types

4.3 An example of Marshalgen annotations

Figure 1 shows an example of using Marshalgen annotations to generate marshaling routine for the class Foo.

```
//MSH_BEGIN
class Foo
{
public:
 int count;
 double *HC; /* MSH: ptr_as_array
  [elementType: double]
  [elementCount: { $ELE_COUNT = $THIS->count; }]
  [elementGet: {$ELEMENT= $THIS->HC[$ELE_INDEX];}]
  [elementSet: {$THIS->HC[$ELE_INDEX]= $ELEMENT;}]
 */
}
//MSH_END
```

Figure 1. Marshalgen annotations for a simple class

The declaration of the struct or class to be marshaled must be surrounded by //MSH_BEGIN and //MSH_END. The annotation right after the declaration of a data field describes how Marshalgen should marshal the data field. All the annotations are in the format of */* MSH: annotation_type [options] */.*

In the example Figure 1, the annotation /* MSH: ptr_as_array ... */ tells Marshalgen that the data field double *HC is a pointer to an array, and the information necessary to marshal the data field is specified in the following options.

The option [elementType: double] tells Marshalgen that each element of the array has the double type.

The option [elementCount: { $ELE_COUNT = $THIS->count; }] specifies how Marshalgen can obtain the size of the array in the number of elements (presumably the programmer maintains the correct size of the array HC in the data field count). The C++ code inside {...} is the code to be executed to obtain the array size. The array size should be assigned to $ELE_COUNT, a variable used by Marshalgen. The variable $THIS refers to the object to be marshaled. We would have liked to use "this" instead of "$THIS". However, since the marshaling/unmarshaling routines are put in a separate class, "this" would not refer to the object to be marshaled but to the instance of the class containing the marshaling/unmarshaling routines.

Similarly, the C++ code inside {...} of the options elementGet and elementSet specify how Marshalgen can get access to the marshaled object. In the option elementGet, $ELEMENT refers to the object to be marshaled. In the option elementSet, $ELEMENT refers to the object that has been reconstructed. $ELEMENT is automatically assigned by Marshalgen. $ELE_INDEX refers to the index of the object in the array. The variable $ELE_INDEX is automatically assigned by Marshalgen.

The options elementGet and elementSet are needed because sometimes the data field is not directly accessible from outside the class: the data field may be declared as private or protected. Since we are not allowed to change the original code of the class, the marshaling/unmarshaling routines have to reside outside the class. As a consequence, the user has to provide Marshalgen with necessary code to access the data fields declared as private or protected.

An example of the annotations used for marshalling a collection of Geant4 Hits follows in Figure 2. Each Geant4 application that declares a new type of hit also provides annotations.

5 Ease of Parallelization: the experience of DMX

DMX (Dark Matter Experiment) [10, 17] is contained in the Geant4 distribution under examples/advanced/underground_physics. It is a larger application, containing 9,000 lines of code, in addition to the Geant4 toolkit.

In the parallelization, we wrote a new, derived class, ParRunManager that is used for all Geant4 applications, not just DMX. That file contains about 200 lines of code, of which 137 are actual code, and the rest is comments. Of the 137 lines of code, 61 lines were copied verbatim from the original model in RunManager. This leaves 76 lines of new code to implement the parallelization. This derived class,

```
//MSH_BEGIN
class G4HCofThisEvent
{ ...
  // data members
private:
  G4std::vector<G4VHitsCollection*> *HC;
                                  /* MSH: ptr_as_array
    [elementType: G4VHitsCollection*]
    [elementCount: { $ELE_COUNT =
              $THIS->GetNumberOfCollections(); }]
    [elementGet: { $ELEMENT =
                     $THIS->GetHC($ELE_INDEX); }]
    [elementSet: { $THIS->
      AddHitsCollection($ELE_INDEX, $ELEMENT); }]
  */
  // member methods
public:
  inline G4VHitsCollection* GetHC(G4int i)
                                  { return (*HC)[i]; }
  inline G4int GetNumberOfCollections()  { ... }
  void AddHitsCollection(G4int HCID,
                         G4VHitsCollection * aHC);
  ...
}
//MSH_END
```

Figure 2. `G4HCofThisEvent.hh` **(annotation for a Geant4 class in slanted characters)**

ParRunManager, can be reused verbatim in any Geant4 application.

For the marshaling, we only needed to add application-specific annotations to existing ".h" files. The actual marshaling code is automatically generated from these annotations. We required 98 lines of annotation to marshal six classes: DMXPmtHit (2 fields, 8 lines), DXMScintHit (5 fields, 20 lines), G4HCofThisEvent (1 field, 17 lines), G4String (1 field, 17 lines), G4VHitsCollection (3 fields, 10 lines), and G4THitsCollection (2 fields, 26 lines).

The first two classes are required for DMX, a particular application of Geant4. The marshalling of the next two classes can be used verbatim in any Geant4 application. The final two classes can be reused almost verbatim, except that one case dispatches according to whether a hit is of type DMXPmtHit or DMXScintHit. In total, 15 fields are annotated.

Of the 98 lines, some of the fields requiring additional lines of annotation are the private fields. In this case, the annotation must specify the accessor and modifier methods to use for marshaling. In another case, one has G4VHitsCollection (an abstract class) as a parent class of G4THitsCollection (a template class). The template G4THitsCollection<T> can be instantiated with DMXPmtHit or DMXScintHit for T. The annotation must specify at compile time how to marshal both template instantiations. From this, Marshalgen must construct a template marshaling class, MarshalG4THitsCollection<T>. In order for this marshaling class to correctly function, we must annotate how to detect at run-time which template instantiation is present, and to marshal accordingly.

6 Performance

The performance at run-time will be almost the same as using MPI. This is because the tools (TOP-C, Marshalgen) used in parallelizing Geant4 incur very little overhead at run-time.

1. TOP-C is essentially an MPI library with a higher-level abstraction. The extra overhead incurred by TOP-C should be very small.

2. Marshalgen acts as a source-to-source preprocessor. The overhead incurred at run-time will be only: (i) constructing of marshaling buffer for objects by the marshaling code, and (ii) parsing buffers to reconstruct objects by the unmarshaling code.

We measured the above overhead at run-time for marshaling and unmarshaling process. It took 0.118 s to marshal an object of class `G4HCofThisEvent` to a buffer of 2,297,504 bytes on a SunBlade 100 machine. This is approximately 18 MB/second. Approximately the same rate is achieved by the unmarshaling process.

The communication overhead, although almost the same as MPI, does incur some cost. Therefore, the speedup depends on the granularity of each particular Geant4 application (see Figure 3). With 50 processes, the master is observed to use approximately 30% of the CPU time. With 100 processes, the master is observed to use approximately 50% of the CPU time.

Application name	# of CPUs	Running time (s)	Speedup	# tasks aggreg.
DMX	50	1106	49.3	1
DMX	100	578	94.4	1
Example N02	1	15,213	1	1
Example N02	50	555	27.4	1
Example N02	50	374	39.0	10
Example N02	50	441	34.5	50
Example N02	100	464	32.8	1
Example N02	100	304	48.6	10
Example N02	100	257	59.3	50

Figure 3. The speedup of parallelized Geant4 applications. (DMX = Dark Matter Experiment)

Figure 3 shows the speedup obtained when we ran the sequential and parallel versions of two Geant4 applications: DMX [10] and N02. DMX can be found in the `examples/advanced/underground_physicsdirectory` of the Geant4 distribution. N02 can be found in examples/novice/N02. The simulations used Geant4 version 4.6.2 and gcc-2.95. They were run on dual processor Pentium machines with 1 GB of RAM running at 1.3 GHz on the lxplus cluster at CERN. The machines were running RedHat 7.3. The machines were lightly loaded. The running time in each case excludes approximately two minutes that Geant4 uses to load its libraries and initialize its data structures on all master and slave nodes.

Number of aggregated tasks	Running time (s) for 50 CPUs	Running time (s) for 100 CPUs
1	555	464
2	410	322
5	385	239
10	374	304
25	393	238
50	454	303

Figure 4. Number of aggregated tasks and running time for ParN02 for 50,000 events

7 Acknowledgement

We gratefully acknowledge the frequent conversations with John Apostolakis and Gabriele Cosmo about the design and use of Geant4. We also acknowledge Stephen Reucroft and John Swain for having introduced us to the issue of parallelization of Geant4. Finally, we acknowledge CERN for the use of their facilities for benchmarking.

Acknowledgement

This work was partially supported by the National Science Foundation under Grant ACIR-0342555.

References

[1] S. Agostinelli et al. Geant4: A simulation toolkit. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 506(3):250–303, 2003. (over 100 authors, incl. G. Cooperman).

[2] J. Allison et al. Geant4 developments and applications. *IEEE Transactions on Nuclear Science*, pages 270–278, 2006. (73 authors, incl. G. Cooperman).

[3] G. Alverson, L. Anchordoqui, G. Cooperman, V. Grinberg, T. McCauley, S. Reucroft, and J. Swain. Scalable parallel implementation of geant4 using commodity hardware and task oriented parallel c. In *Proceedings of the International Conference on Computing in High Energy and Nuclear Physics (CHEP2000), Padova, Italy*, 2000. `http://chep2000.pd.infn.it/short_p/spa_e041.pdf`.

[4] G. Alverson, L. Anchordoqui, G. Cooperman, V. Grinberg, T. McCauley, S. Reucroft, and J. Swain. Using TOP-C for commodity parallel computing in cosmic ray physics simulations. *Nuclear Physics B (Proc. Suppl.)*, 97:193–195, 2001.

[5] G. Cooperman. TOP-C: A Task-Oriented Parallel C interface. In 5^{th} *International Symposium on High Performance Distributed Computing (HPDC-5)*, pages 141–150. IEEE Press, 1996. software at `http://www.ccs.neu.edu/home/gene/topc.html`.

[6] G. Cooperman, H. Casanova, J. Hayes, and T. Witzel. Using TOP-C and AMPIC to port large parallel applications to the Computational Grid. *Future Generation Computer Systems (FGCS)*, 19:587–596, 2003. (also appeared in Proc. of 2^{nd} IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2002)).

[7] G. Cooperman, N. Ke, and H. Wu. Marshalgen: A package for semi-automatic marshalling of objects. In *Proc. of The 2003 International Conference on Internet Computing (IC'03)*, pages 555–560. CSREA Press, 2003. software at `http://www.ccs.neu.edu/home/gene/marshalgen.html`.

[8] G. Cooperman and V. Nguyen. Marshalgen: Marshaling objects in the presence of polymorphism. In *The 2004 International Conference on Internet Computing (IC'04)*, pages 17–23, 2004. software at `http://www.ccs.neu.edu/home/gene/marshalgen.html`.

[9] Geant4 webpage. `http://wwwinfo.cern.ch/asd/geant4/geant4.html`.

[10] A. Howard. Simulating a dark matter prototype detector and understanding characterisation response to alphas, gammas and neutrons. In *Geant4 2003 Workshop, TRIUMF, Vancouver*, 2003. `http://www.triumf.ca/geant4-03/talks/05-Friday-PM-1/02-A.Howard/`.

[11] F. Lei. Radiation transport simulation specification. Technical report, European Space Agency and QinetiQ, January 2002. ESA Technical Note SGD-RTS-QIN-SSTN-003-1.3; at `http://reat.space.qinetiq.com/spacegrid/SGD-RTS-QIN-TN-003-1.0.pdf`, also see `http://reat.space.qinetiq.com/spacegrid`.

[12] LinPack webpage. `http://www.netlib.org/linpack/`.

[13] Geant4 MULASSIS Tool: Multi-LAyered Shielding SImulation Software. `http://reat.space.qinetiq.com/mulassis/mulassis.htm`.

[14] Object Management Group. The Common Object Request Broker: Architecture and specification, 1999. Minor revision 2.3.1, OMG TC Document formal/99-10-07.

[15] ParGeant4 webpage. `http://www.ccs.neu.edu/home/gene/pargeant4.html`.

[16] D. Reilly. Introduction to remote method invocation, Oct. 1998. `http://www.davidreilly.com/jcb/articles/javarmi/javarmi.html`.

[17] P. Smith, N. Smith, J. Lewin, G. Homer, G. Alner, G. Arnison, J. Quenby, T. Sumner, A. Bewick, T. Ali, B. Ahmed, A. Howard, D. Davidge, M. Joshi, W. Jones, G. Davies, I. Liubarsky, R. Smith, N. Spooner, J. Roberts, D. Tovey, M. Lehner, J. McMillan, C. Peak, V. Kudryavtsev, and J. Barton. Dark matter experiments at the UK Boulby Mine. *Physics Reports*, 307:275–282, 1998.

[18] Sun Microsystems, Inc. ONC+ developer's guide, Nov. 1995.