oTN-2008-01 openlab Technical Note

# Multi-threaded Geant4 with Shared Detector

Author: Xin, Dong
Supervisor: Sverre, Jarp

2 August 2008
Version 1
Distribution:: **Public**

**1 Introduction ....... 1**
**2 Parallel computing on many-core machines and relevant technologies ....... 2**
2.1 Methodology for scalable parallel computing on many-core machines ....... 2
2.2 Thread-Local Storage (TLS) ....... 4
2.3 Semi-automatically transforming C++ code using a C++ parser ....... 5
2.4 TOP-C parallelism for Geant4 ....... 5
2.5 Recognizing read-write data and Valgrind/Helgrind ....... 6
**3 Unconditionally thread-safe Geant4 ....... 7**
3.1 How to make variables thread-local ....... 7
3.2 Collecting what should be changed ....... 8
3.3 Automatically making all variables thread-local ....... 9
3.4 Several additional changes ....... 10
**4 Implementation with shared detector data ....... 11**
4.1 Data model for shared detector ....... 11
4.2 Safely sharable data ....... 12
4.3 Sharing part of data members for C++ instances in a given class ....... 13
4.4 Private data initialization ....... 14
**5 Summary and future work ....... 17**
**6 References ....... 18**

## 1 Introduction

Geant4[4] is million-line software toolkit to simulate the interaction of particles with matter. It is widely used in many fields such as high energy, nuclear and accelerator physics, medical science, and space science. To improve the throughput of a simulation, the end user splits the input data (the input events) into multiple subsets, and starts multiple simultaneous processes, with each process simulating a different subset of input events.

To enhance Geant4 applications with parallel computing, a more sophisticated method is ParGeant4[9] based on the free TOP-C[1,2] (Task Oriented Parallel C/C++) distribution. TOP-C adopts a *master/worker architecture*. The master takes responsibility to monitor the state of each worker, and generate and assign a task for any idle worker. Workers wait for tasks from the master, execute the tasks and send the results back to the master. In ParGeant4, a task is a single event (*event-level parallelism*). In this way, ParGeant4 implements both event-level parallelism and load-balancing.

Each event is associated with a unique number which is used as the random generator seed for this event. This number itself comes from a repeatable sequence of pseudo-random random numbers. This ensures that the result of a parallel computation is reproducible. This *reproducibility* is important for determining and maintaining program correctness. If the original sequential implementation uses the same set of seeds for a simulation, then for the same input events, a simulation result returned by a sequential computation will correspond to the simulation result of the parallel computation.

As we move toward modern multi-core machines, we must again implement the same process parallelism, whether by scripts or the use of ParGeant4. It then becomes crucial that memory is sufficient for master and workers. For large experiments in high energy physics such as ATLAS, CMS and ALICE[6], detector data is huge: up to 1 GB. It is important for master and worker processes to reduce image size by sharing data. Since detector data is initialized dynamically in the process heap, it is impractical for processes to share this data simply through the use of virtual memory and the UNIX *mmap* system call.

To share dynamically initialized data, one possible solution is to utilize the *copy-on-write* mechanism associated with the UNIX *fork* system call. The methodology is that all child processes are forked after the main process has initialized most (or all) of its dynamic data. Using the copy-on-write technique, child processes inherit the same resources as their parent including the same content of the page table. A child process does not duplicate the parent's resources, but instead shares physical pages with its parent until one of them tries to write to a page. At that time, a copy of the page is made and assigned to the writing process.

Although copy-on-write works successfully with many other applications, it does not work well with Geant4. The reason is that the size of C++ objects in Geant4 is usually small compared with the page size. (A typical page size is 4 KB.) In addition, few object instances are purely read-only. Whenever even one data member of an instance is changed by a process, the operating system must duplicate the entire page containing that instance for that process. The likely outcome is that that the whole instance is duplicated along with the page on which it resides. In the end, almost all detector data is duplicated for every process as a result.

In this work, our objective is to implement a multi-threaded Geant4 with shared detector data. This implementation allows application developers to launch a single multi-threaded process, instead of multiple processes. A multi-threaded Geant4 that shares detector data (and corresponding object instances) will consume less memory. The goal is achieved in two phases.

1. First, we semi-automatically generate an *unconditionally thread-safe Geant4*. This version is called unconditional because any thread can call any function of Geant4. Each thread has a thread-private copy of the entire data. There is no data sharing in this multi-threaded version. But threads also never conflict or influence each other.
2. Second, we determine the data that can be shared among threads during the parallel computation. Starting from the unconditionally thread-safe Geant4, we modify it step by step to share more and more data. Using this methodology, most detector data can be shared by all threads. This reduces image size tremendously. Finally, we obtain a multi-threaded version of Geant4 that is scalable on multi-core machines.

A secondary goal is to create the multi-threaded version of Geant4 in a semi-automatic manner. This allows us to take new versions of the sequential Geant4 code and automatically transform them into multi-threaded code. We wish to avoid the necessity of maintaining multiple versions of Geant4.

This report is organized as follows. Chapter 2 briefly introduces parallel computing on many-core machines and relevant technologies. Chapter 3 presents the implementation of unconditionally thread-safe Geant4. Chapter 4 shows how to change the first thread-safe Geant4 gradually to share more and more data. Chapter 5 discusses a future implementation.

# 2 Parallel computing on many-core machines and relevant technologies

## 2.1 Methodology for scalable parallel computing on many-core machines

Many-core machines are different from computer clusters in two aspects. On the one hand, from a cluster with hundreds of thousands of processors, we obtain the commensurate quantity of memory and disk space. For many-core machines, it is an incorrect assumption that memory and disk space are always sufficient when thousands of cores allow many processes reside in one machine. On the other hand, on many-core machines, we no longer count on network to organize parallel computing. It helps to achieve as much performance for applications, which we could not expect from clusters. In the HPC world of scientific computation on clusters with a large number of processors, IPC between nodes becomes bottleneck when computing granularity is decreased for load-balancing among all processors. This bottleneck limits the optimal number of processors for effectively speeding up the computing of a given application because the communication complexity is square of the number of processors and communication overload increases rapidly when the number of processors increases.

Consider machines with a few cores, for example, dual, quad and eight cores. They are generally treated as machines with two, four and eight processors. For these machines, process parallelism based on sequential computing still works. We just integrate a proper IPC method with the original sequential algorithm for a given application. In this case, job size granularity is sufficient for load-balancing among a few cores. But the same parallelism does not squeeze more performance from abundance cores for two reasons. First, processes are not lightweight and consume lots of resources, which form a heavy burden for both operating system and hardware when the number of processes is large. Second, job size granularity is too coarse to allow sufficient number of jobs needed to make all cores busy.

Faced with a many core or massive core future, our effort concentrates on task-oriented thread parallelism. For any application, tasks are identified by studying the original sequential algorithm and are flexible units for computing. It can be one step or several steps in a loop. Or it is associated to a bunch of processing to a block of data. Task is always generated by master thread dynamically. The only requirement is that the generated task should be independent on all tasks that are being processed by worker threads. Task size granularity is generally small enough to make the number of tasks sufficient for load-balancing. All threads should be independent on each other and there is no data race among all threads in the computing of tasks. So we do not need any thread synchronization mechanism. This feature is critical for parallel computing on many core machines. Thread synchronization results in sequential computing for critical section. It hurts performance when the number of threads is large.

Although some applications do not satisfy conditions mentioned above, we obtain task-oriented thread parallelism in many other applications. Whether we use thread model or process model does not influence task-oriented parallelism. The parallelism for process model is still applicable to thread model. Therefore, given an application, the parallel algorithm for clusters can be ported to many core machines using thread model. TOP-C is very helpful for developers to move from the computing on clusters to the computing on many core machines using thread model. The discussion in section 2.4 can be considered as an example of TOP-C parallelism for both process model and thread model.

To move from process parallelism to thread parallelism, one problem is that any global variable changed in the course of processing one task may be changed by all threads, which is not allowed in our model. The solution to this problem is to make all global variables thread-private by thread-local storage (TLS) technology. This technology is introduced in section 2.2. Based on this technology, all global variables are transformed into thread-private variables. Therefore, any memory unit touched by each thread is proprietary to this thread. If the programming pattern of a C++/C application is good, this transformation can be achieved semi-automatically. In section 2.3 and section 3, this method is explained implicitly by its use in Geant4.

Consider the parallel computing of applications. There are three cases: computation intensive applications, memory access intensive applications and message passing intensive applications. If an application is message passing intensive, then thread parallelism is expected to achieve more performance on many-core machines because it does not need any IPC mechanism. Inter-thread communication can be implemented by newly introduced global variables. If an application is computation intensive only, then process parallelism on clusters, process parallelism on many-core machines and thread parallelism on many-core machines obtain similar performance. The most difficult case is memory access intensive applications. For many-core machines, each thread compete memory bandwidth with others to access memory.

It is important for memory access intensive applications to reduce total image size no matter what model they adopts. For thread model, we expect to share read-only data in the course of processing one task. These data include read-only global data that resides in data segment. There is only one copy for such kind of data and they are shared naturally by all threads. For read-only data allocated dynamically, to share only the pointer to this data is not enough. We expect to share the data itself especially for some applications where dynamically allocated data is far more than those in data segment. To this end, three things need to be considered.

First, we must answer precisely what read-only variables are. Consider a read-only variable which does not change in the course of processing one task. There is no write access happens to this variable when any task is processed, which is not observable. We figure out what data is read-write and can not be shared safely because changes to a variable are observable by helgrind--one valgrind option. Two threads are started deliberately to execute subroutines of processing one task. If a variable is read-write, it must be changed by two threads. Because no thread holds any lock, helgrind thinks that it is not a correct pattern and will issue an error message. In section 2.5 and section 4.2, we show how helgrind works for this purpose.

Second, no compiler provides any mechanism to put dynamically allocated data in TLS or somewhere outside Heap. Following our methodology, we require read-only data has one copy shared by all threads while read-write data has one copy per thread. Therefore, read-write data members must be separated from read-only data members as a new instance if they exist in one instance. For a set of shared instances of a given class type, a TLS pointer is used as the base address to access those read-write instances. This TLS is defined as a static member of the given class. Otherwise, we can not claim it as a TLS variable. Accordingly, all read-write instances are organized as an array. In addition, we associate to each instance an ID, which is used as the index to this array. Inside a class method, when a read-write data member is accessed, the program follows the TLS pointer and instances ID to access thread-private units. This is implemented simply by a macro definition if each data member has a globally unique name.

Third, data initialization should be changed after we separate thread-private data members from shared data members. A shared instance has one copy and there is only one thread (usually master thread) to take the responsibility of constructing it. Other threads must have different behaviour from this thread when they initialize data. Therefore, all threads are no longer equal when they initialize data. The class whose instances are shared should implement more methods to support worker thread initialization. A more detailed description about the initialization for thread-private data refers to the example in section 4.4.

## 2.2 Thread-Local Storage (TLS)

*Thread-Local Storage* (TLS) is the critical technology for this work. If a variable is declared as *thread-local*, the compiler will reserve space for this variable in the TLS of each newly created thread. As a result, different threads may refer to the same thread-local variable by name, and yet access a thread-local copy of that variable in its own TLS. The TLS syntax is simple. The magic keyword is "*__thread*". This keyword was introduced in the ISO C++ 98 standard, and is supported by recent versions of GNU g++. Below is an example to explain the usage of TLS.

```
#include <stdio.h>
#include <pthread.h>
__thread int gvar = 0;
void *increase(void *)
{
  gvar++;
  printf("Value in child thread: \%d\n", gvar);
}

int main(int argc, char* argv[])
{
  pthread_t tid;
  printf("Value in main thread: %d\n", gvar);
  pthread_create( \&tid, NULL, increase, NULL );
  pthread_join(tid, NULL);
  printf("Value in main thread: %d\n", gvar);
  return 0;
}
```

In this example, both the main thread and a second, dynamically created thread access the variable *gvar*. Because this variable is declared as thread-local, the two threads actually access their thread-local memory in TLS. There is no data race when they access this variable simultaneously. So the output of this program appears as follows:

```
Value in main thread: 0
Value in child thread: 1
Value in main thread: 0
```

In addition to the term *thread-local*, we will often refer to *thread-private* data. A thread may create a new object, which is stored on the common heap shared by all threads. If the pointer to the new object is stored in the thread's thread-local data, then no other thread can access this new object on the heap. In this situation, we refer to both the new object and the thread-local pointer to the new object as being *thread-private*. However, only the pointer is *thread-local*, since only the pointer is stored in TLS.

## 2.3 Semi-automatically transforming C++ code using a C++ parser

To obtain the initial, unconditionally thread-safe Geant4, a semi-automatic method is adopted to transform the original Geant4 source code to be thread-safe. This method has been applied to Geant4 9.0 and Geant4 9.1.p01. For each of them, it requires less than one day to build the corresponding thread-safe version. This method includes two steps.

First, we collect all occurrences of "static" and global variables. For each global variable, we also collect all occurrences of external references to it. It is achieved by patching some code in the C++ parser. The patched code just prints out the declaration with its location for each occurrence of "static" variables, global variables and their external references. The information for one declaration looks like:

```
G4StateManager.hh: 135 : static G4StateManager* theStateManager;
```

Second, we developed a program to change declarations collected in the first step. The above example is the simplest case. It becomes:

```
static __thread G4StateManager* theStateManager;
```

In most cases, the program transforms those declarations correctly and automatically. In a few special cases, we have to change source code manually. For the second step, another C++ parser (not the GNU C++ parser, g++) is employed to read declaration words. This parser need not to be a powerful one. Currently, we use the open source C++ parser Elsa[3]. For more detailed information about this parser, refer to its webpage.

There are about 10,000 lines of code that need to be changed. It is time-consuming and painful to change them for a human being. Furthermore, to maintain such huge software by hand is error-prone. This semi-automatic method is very helpful here.

## 2.4 TOP-C parallelism for Geant4

TOP-C is free open source software to support parallel computing. The goal of the TOP-C philosophy is to parallelize sequential code while making minimal changes to the original sequential code. The resulting parallel algorithm is derived by identifying tasks to be computed in parallel from the original sequential algorithm. In addition, the parallel code works for both distributed-memory machines and shared-memory machines. For shared-memory machines, TOP-C adopts the POSIX multi-threading model.

ParGeant4 for distributed-memory machines takes advantage of TOP-C parallelism, which is internally implemented on top of MPI[5,7,8]. It achieves parallel computation without any change to the Geant4 source code, by re-implementing certain virtual methods. ParGeant4 application developers need only to use a parallel version of run manager class. This class implements a parallelized DoEventLoop method, which is illustrated in Figure 1.

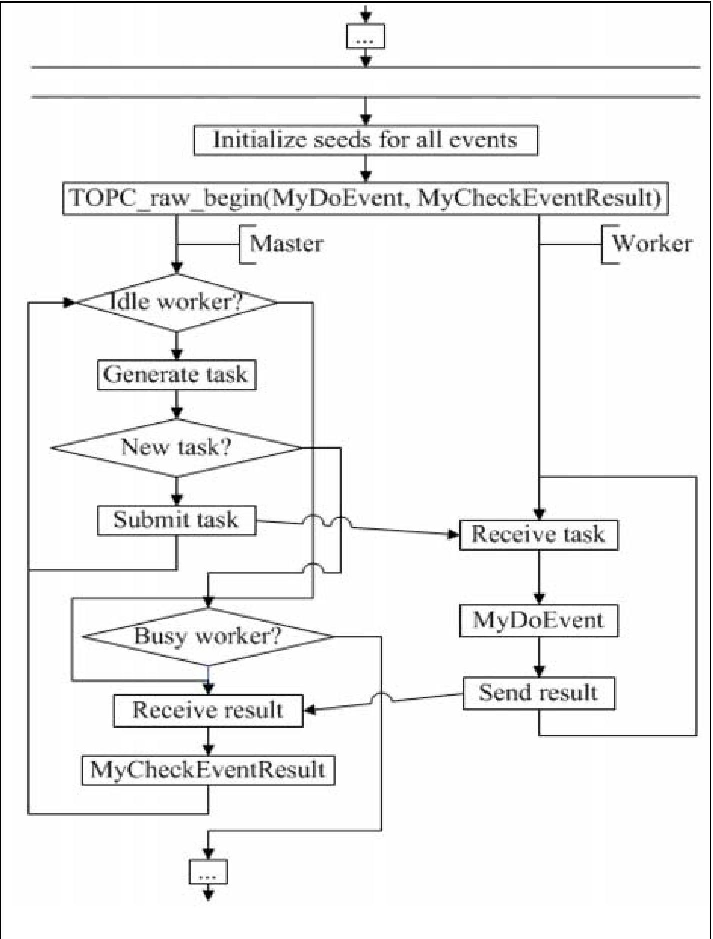

Fig. 1. TOP-C Parallel DoEventLoop.

In the figure above, master and workers first register two callback functions. One callback function is *myCheckEventResult*, which is used by master; another callback function is *MyDoEvent*, which is used by workers. The master submits tasks to workers until there are no more tasks or until no worker is idle. Then, it waits for any result from any of the active workers. After a result is received and handled, the master loops to submit tasks. Idle workers wait for any task from the master, and then call *MyDoEvent* function and send the result for this task back to master. Here, the parallel computation happens when all workers process the assigned event simultaneously.

Multi-threaded Geant4 adopts a methodology similar to that of ParGeant4. However, it requires the unconditionally thread-safe Geant4 as a starting point.

## 2.5 Recognizing read-write data and Valgrind/Helgrind

From unconditionally thread-safe Geant4 to the current implementation, we must consider the methodology to share detector data in order to reduce the image size. The first question is what data can be shared safely. From what we have said in last section, parallel computation occurs only in the *MyDoEvent* method of the parallel run manager class. This method encapsulates functions for task message unpacking, event generating, event processing and results message packing. TOP-C ensures that task message unpacking and results message packing are thread-safe. Event generating (*GenerateEvent*) is naturally thread-safe because it just creates some instances and initializes them for the current event. Therefore, event processing (*ProcessOneEvent*) is the only source of data races. Furthermore, the data that is not changed by the method *ProcessOneEvent* can be safely shared by all threads. However, in the case of Geant4, few instances are purely read-only. In most cases, only some data members of a given instance are read-only while others are read-write. When this instance is shared by all threads, each thread should still hold its own thread-private copy of read-write data members. So, all accesses to read-write data members need to be changed.

To figure out what data is changed by *ProcessOneEvent* method and hence cannot be shared safely, one could expand this method to expose all variable accesses. However, it is not feasible currently because of complicated inheritance relationships and usage of virtual function in Geant4. Virtual functions are dynamically linked to Geant4 applications and can only be checked at runtime.

Another way is to utilize valgrind[10] with the option --*tool=helgrind*. The original objective of this tool is to check whether a computation by many threads employs a correct pattern. It can be used to check data races -- accessing memory without adequate locking. If two threads pass through and change the same variable without adequate locking, this tool will issue an error message. In our usage of this tool, we deliberately start two threads to pass through some functions used by the ProcessOneEvent method. This causes each read-write data is accessed by two threads without any lock. Helgrind complains about all accesses of this kind. From the Geant4 source code, we can find the name of the read-write variable corresponding to a particular error message. For more detailed information on valgrind, refer to its webpage.

## 3 Unconditionally thread-safe Geant4

### 3.1 How to make variables thread-local

Local variables are already thread-local and hence thread-safe, since they are stored on the stack of the thread that invokes them. However, global variables (including those declared in namespace) and "static" variables are generally not thread-safe. When two threads access the same global or "static" variable, they influence each other by changing the value of the variable. One exception is static *const* variables. They can be regarded as constants and are not used as l-values.

If we put __*thread* in front of each variable declaration, all variables become thread-safe, i.e., each thread has its own copy of global or "static" variables. In this sense, multi-threading is the same as the multi-processing and threads do not influence each other. In ParGeant4, TOP-C and MPI have been introduced to support the master/worker model based on process-parallelism. If Geant4 is thread-safe, it is trivial to move from the process parallelism to the thread parallelism following the TOP-C philosophy. So our change to Geant4 is to make all "static" variables, global variables and their external reference declarations have a "__thread" prefix or infix keyword.

The difficulty is that we cannot place "__thread" in front of all types of dynamic variables. Because the C++ compiler handles "__thread" at compile time only, it is difficult for the compiler to determine when dynamic initialization for a variable should happen after a new thread is created. As a result, most TLS implementations do not support thread-local variables of class type, since this would requires dynamic initialization of the variable whenever a new thread is created. (Whenever a global or static variable of class type is declared, an implicit call to its constructor must be executed prior to executing the "main" or "startup" function.) So currently, "__thread" can only be used for a *plain old datatype* (POD) variable. For the same reason, C++ does not support using the "__thread" keyword with dynamic initializers in general, even for POD variables. For example, consider the declaration below

```
static __thread int variable1 = variable2;
```

It is not valid for the GNU g++ compiler.

To solve this problem, we can change any type into a pointer type pointing to the original type. For example, we change

```
std::vector<...> variable;
```

to

```
std::vector<...> *variable_NEW_PTR_;
```

For this kind of change, three extra considerations need to be handled carefully. First, the pointer variable must be initialized before its first use. Second, this pointer should be initialized only once. Third, all references to the original name "variable" should be changed to the expression "*(*variable_NEW_PTR_)*".

As an example, consider this scenario: we'd like to declare a TLS variable that is initialized by another variable. We already mentioned that the declaration below is not correct.

```
static __thread int variable1 = variable2;
```

However, it can be implemented by the following code:

```
static __thread int *variable1_NEW_PTR_ = 0;
if (!variable1_NEW_PTR_)
{
  variable1_NEW_PTR_ = new int;
  *variable1_NEW_PTR_ = variable2;
}
int &variable1 = *variable1_NEW_PTR_;
```

Here, *_NEW_PTR_* can be any string that never previously occurred in Geant4.

The above code works not only for POD variables, but it works for variables of any type including class type. One just changes "new int" to "new class-name". If there is never an assignment to *variable1*, we can optionally delete the statement "**variable1_NEW_PTR_ = variable2;*". For global variables and "static" class members, the above code must be added to the beginning of all relevant functions. Fortunately, the compiler is sensitive to type changes, since C/C++ is a language of strongly typed. If we make any mistake and forget to change any relevant function, the compiler will complain and let us know.

## 3.2 Collecting what should be changed

We need to collect 4 kinds of declarations.

1. *"static" declarations:* Some static declarations define class members. Some static declarations define variables used in a function or a method. Generally, functions and methods can be considered as constants. However, functions or methods containing declarations of static variables have state, which consists of the values of all static variables. In addition, some other static variables are declared and valid in one particular file. They are part of the state for this compilation unit.

2. *global declarations:* Most of them initialize "static" members defined in a given class. They are part of the state of the class. The rest are a few truly global variables. They can be used by other files through external reference declarations. They are part of state of the resulting binary.

3. *"extern" declarations:* When we change a global variable to be a thread-local variable, we should also change the corresponding external reference declarations to be external thread-local reference declarations (using *__thread*), in order to introduce thread-local variables from other files.

4. *"static const" declarations:* They are relevant to singleton classes. If a class is a singleton, then

there is only one instance for this class. Therefore we regard the address of this instance as a constant. However, when we make each thread hold one instance of this singleton class, the pointer to hold the address for instances of this class should be a variable and is no longer constant. We also push this kind of variable into TLS rather than treat it as a constant.

A natural way to collect the above information is to patch some code in the C++ parser to print out the information. In our case, we change the GNU (g++ version 4.2.2) C++ parser source code. It is *./gcc/cp/parser.c* in the gcc source directory. We just patch some output statements there. For "static", "extern" and "static const" cases, there are corresponding keywords in this file. For "global" case, there is no corresponding keyword. A global variable is defined in a file and outside of any function but just like an ordinary one. We create a new function *cp_parser_declaration_seq_opt_toplevel* in the file *parser.c* and replace the function *cp_parser_declaration_seq_opt* with this new function in the g++ functions *cp_parser_translation_unit* and *cp_parser_namespace_body*. For the GNU C++ parser, the translation unit and namespace body are the top level containers: the only place where some global variables may reside.

After the GNU C++ parser file has been changed, we re-compile gcc-4.2.2. We then use the patched compiler to build Geant4 once more. This time all declarations concerned and their locations are collected as part of the build process.

## 3.3 Automatically making all variables thread-local

A tool has been developed to change the Geant4 source file automatically following the information collected in the above step. This tool utilizes an open source C++ parser, Elsa. For each declaration that should be changed, this tool will open the corresponding source file, read the declaration from the source file and write it into a temporary file. Then it uses Elsa to parse it and change the source file accordingly. What kind of change should be made depends on the pattern of the declaration. Whether the type is POD, and whether there is an assignment, influences the way it is changed. In the case of an assignment, the syntax of this assignment also influences how it is changed. For a small number of array assignments, the change must be hard-coded in the Geant4 source code.

The change to a declaration for a non-class member is completely different from the change to a declaration for a class member. The change for a class member declaration is generally more complicated. For example, we have two declarations below. The second declaration is dynamically initialized using the first declaration.

```
G4double    G4VEnergyLoss::dRoverRange    = 20*perCent;
G4double    G4VEnergyLoss::c1lim = dRoverRange;
```

Because TLS can not be used for a dynamically initialized variable, we have to expand *dRoverRange* as follows when we declare the second class data member as a TLS variable.

```
__thread G4double    G4VEnergyLoss::c1lim =(20*perCent);
```

Another case happens when we add "__thread" for some non-POD class members. We can not initialize them immediately in their declarations. We postpone the initialization for a non-POD class member to the time when any class method is called. One more issue for this case is that sometimes we need to change the methods *operator*= and *operator*<<. They have a parameter whose type is the class itself. These two methods may use this parameter to access "static" class members. Under such a circumstance, the tool still works well if we merely change member names for this class. However, we must hard-code this change in the Geant4 source code.

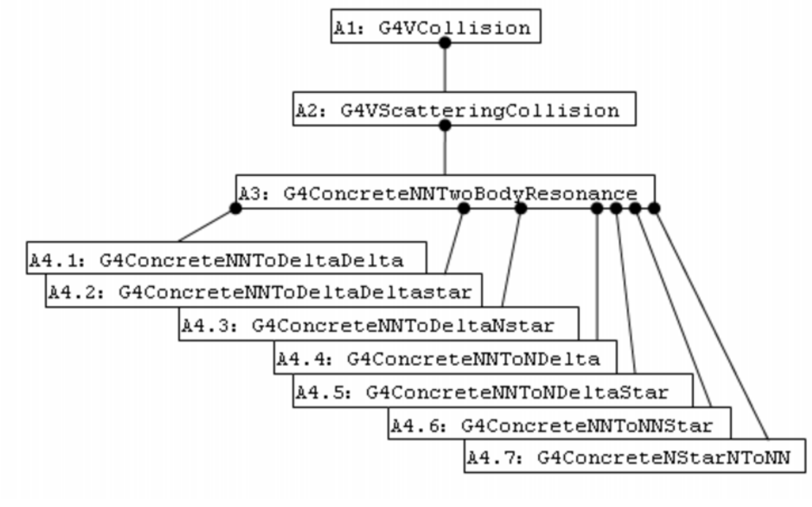
A1: G4VCollision
A2: G4VScatteringCollision
A3: G4ConcreteNNTwoBodyResonance
A4.1: G4ConcreteNNToDeltaDelta
A4.2: G4ConcreteNNToDeltaDeltastar
A4.3: G4ConcreteNNToDeltaNstar
A4.4: G4ConcreteNNToNDelta
A4.5: G4ConcreteNNToNDeltaStar
A4.6: G4ConcreteNNToNNStar
A4.7: G4ConcreteNStarNToNN

Fig. 2. Nontrivial TLS class members.

A more complicated case is to change a hierarchy classes in Figure 2. For each A4.* class, the declaration

```
G4XDeltaDeltaTable *G4ConcreteNNToDeltaDelta::theSigmaTable;
```

should be changed to

```
__thread G4XDeltaDeltaTable *G4ConcreteNNToDeltaDelta::theSigmaTable_NEW_PTR_ = 0;
```

However, the declared class member "*theSigmaTable*" is used by the constructor of the parent class A3. We must ensure that it is initialized before the constructor of the parent class is called.

The solution here is:

1. Add an empty constructor in all ancestor classes A1, A2 and A3.
2. Add a method that simulates the constructor for each A1, A2 and A3.
3. In each A4.* class, force the constructor to use the empty constructor of the parent class A3 added in the first step.
4. Change the constructor for each A4.* class, to implement the method described as below.

First allocate space for the pointer *theSigmaTable_NEW_PTR_* if it is 0. Then define a reference variable as "*G4XDeltaDeltaTable &theSigmaTable = *theSigmaTable_NEW_PTR_;*". Finally call the method added in step 2.

For this case, the whole change is also hard-coded in the Geant4 source code.

## 3.4 Several additional changes

Consider the names below:

```
G4ProductionCuts.cc: gammaDef, electDef, positDef
G4EnergyLossTables.cc : lastParticle
```

They belong to the fourth kind of declaration discussed in section 3.2. So we must replace "const" with "__thread". Furthermore, for some local variables declared as "static const" type, it is not trivial to determine whether a previous constant has been used as an l-value. Hence these static local variables can no longer be considered as constants. This case is handled carefully.

More extra changes result from our change to the names of "static" class members. Although any reference to a "static" class member directly hurts encapsulation, it is more convenient for Geant4 developers to use them in some special situations. In addition to making local change to the implementation of a given class, we change global references to "static" class members by hard-coding it in the Geant4 source code, similarly to the case discussed in section 3.1. The compiler complains about name changes and tells us what should be changed.

To generate this unconditionally thread-safe system, all relevant libraries should also be transformed to be unconditionally thread-safe. Geant4 uses CLHEP in addition to C++ standard library. The current usage of CLHEP in Geant4 is through the static interface of the class *HepRandom.* We merely apply the same methodology to CLHEP and make it unconditionally thread-safe on the condition that the GNU implementation of the C/C++ standard library is thread-safe. CLHEP also provides non-static interface by defining different generator objects by means of the various engines available. If the future version of Geant4 replaces the static interface with this non-static interface, then it does not matter whether CLHEP is unconditionally thread-safe or not.

## 4 Implementation with shared detector data

### 4.1 Data model for shared detector

To transform from the original Geant4 to the unconditionally thread-safe Geant4 implemented in section 3, we push part of the data segment into TLS. Only constants reside in the data segment. Following this methodology, each thread still initializes and holds its own data. So, all threads are independent. They completely simulate process-parallelism in a thread-parallel context. Since multiple independent processes are a form of "embarrassingly trivial parallelism", we typically obtain the expected linear speedup for Geant4 as more CPU cores and threads are provided. Hence, the software is *scalable*.

However, this scalability does not hold if the memory is insufficient on multi-core machines. This can easily occur, since the image size is proportional to the number of workers. This is especially true, for example, for fullCMS, which is a simplified version of CMS experiments. For the current implementation, we manage to move some data from TLS (in the unconditional thread-safe Geant4) back to the data segment and therefore share it, as illustrated in Figure 3.

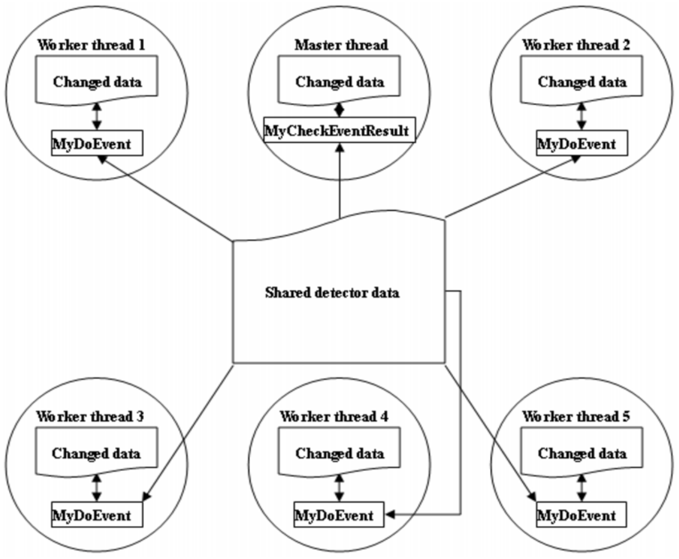

Fig. 3. Data model with shared detector data.

The above paradigm is attained in three steps:

1. Figure out what data can be shared.
2. Provide a solution on how to share some of data members for a set of a given C++ instance of a particular class.
3. Determine the behavior of master and workers in data initialization.

## 4.2 Safely sharable data

The tool *helgrind* is utilized to recognize read-write data, which is complementary to safely sharable data. This tool works directly with binary code. It sets up the state for each memory unit and monitors status transitions. For this reason, it is impractical to check how much data is updated by multi-threaded ProcessOneEvent, for example in fullCMS, whose image size is about 150M. Hence, we use helgrind with Geant4 unit tests for some of its modules.

For the geometry module, test programs for navigator and navigation can be use to check what data is changed. Here, we use the original sequential Geant4 (not the unconditionally thread-safe version) to deliberately produce data races. The programs are modified to create multiple threads that all run the same test case. The multi-threaded test programs look as follows.

```
G4VPhysicalVolume *myTopNode;
int sleepTime = 10;
void *worker(void *waitTime_ptr)
{
  testG4Navigator1(myTopNode);
  testG4Navigator2(myTopNode);
  //sleep a while, so valgrind can analyze it
  sleep(sleepTime);
}

int main()
{
  myTopNode=BuildGeometry();
  CloseGeometry(false);
  …
 pthread_create( &tid1, NULL, worker, NULL);
```

```
  pthread_create( &tid2, NULL, worker, NULL);
  …
}
```

One compiles them as ordinary Geant4 applications and then starts executable files using valgrind as follows.

```
valgrind --tool=helgrind --log-file=output a.out
```

As the last step, one analyzes the output from helgrind. For example, we see the following complaints.

```
==538== Possible data race during write of size 4 at 0x56360A0
==538==    at 0x42B944: G4PVReplica::SetCopyNo(int) (G4PVReplica.cc:180)
…
==538==   Old state: owned exclusively by thread #2
==538==   New state: shared-modified by threads #2, #3
==538==   Reason:    this thread, #3, holds no locks at all
```

In the above message, helgrind says that the second thread (worker thread 1) changes data member *fcopyNo* (integer number) for a *G4PVReplica* instance. Because we do not use any lock, helgrind sets up the state of *fcopyNo* to be "owned exclusively by worker thread 1". Just like the second thread, the third thread (worker thread 2) also changes the same variable. Then helgrind understands that *fcopyNo* is shared-modified by two threads. However, helgrind thinks that it is an error for two threads to modify the same variable without any lock.

After all complaints from helgrind have been analyzed, we discover those data members that are changed by the navigator (and therefore also by ProcessOneEvent) for three kinds of instances: physical volumes, logical volumes and solids, which are expected to be shared. Threads only share read-only data members and keep read-write data as thread-private.

## 4.3 Sharing part of data members for C++ instances in a given class

We use class G4PVReplica as an example to explain how threads share some data members for a set of C++ instances of a given class. Suppose we have three replicas. As described in Figure 4, they reside in the common process heap. All threads share these replicas while *fcopyNo* is kept thread-private.

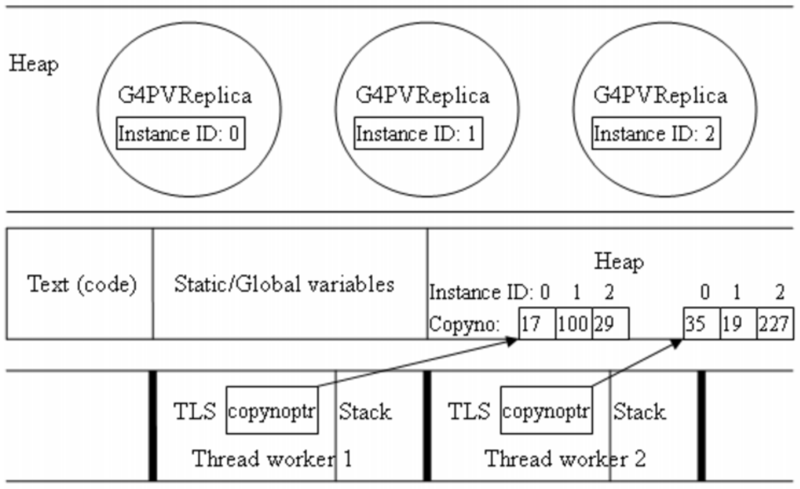

Fig. 4. Sharing replicas while *fcopyNo* is still thread private.

To share replicas, first we add one integer class member to G4PVReplica. This new member is used as an ID for each G4PVReplica instance. Second, we delete class member *fcopyNo* from G4PVReplica. Third we use a pointer of integer which points to an array of integers (fcopyNos). This pointer is declared as "static" thread local member of class G4PVReplica. This array is indexed by replica IDs. When a thread accesses *fcopyNo* for a particular instance, it follows this pointer and the instance ID to access thread-private memory regions.

In summary, we must separate safely sharable part from thread-private portion for any class whose instances are expected to be shared. In addition, we need an object counter for this class to assign an ID to each instance. For this purpose, we must change all constructors of this class to touch this object counter in order for each newly created instance to get an ID.

## 4.4 Private data initialization

After data members of shared instances of a given class are separated into a shared part and a thread-private part, the initialization must be changed accordingly. For the master thread, initialization is re-implemented by changing only constructor(s) for this class in order to allocate space for private-data. However, worker threads must skip some routines -- especially those methods implemented by the detector construction class. It should not generate more thread-private copies of those instances of the detector. But it must initialize its own thread-private part of the data. So workers should coordinate with the master to finish initialization correctly. Here, we assume that the workers do not start to initialize data until the master has finished the data initialization.

To explain how instances, object counter, master thread and worker threads interact with each other, we introduce some names below.

SI: a representation of shared instances.
SC: the class of SI. (After thread-private data members has been removed and data member instance ID has been added)
TPC: a new class to contain all thread-private data members
ID: new class member added to SC as instance ID.
OC: object counter for SC.

OC holds a TLS pointer–*offsetP*–which points to an array of TPC type pointers. The pointer *offsetP*, whose initial value is 0, is allocated dynamically. OC implements the method *newInstance*. This method returns the minimal unused instance ID. In order for worker threads to share *offsetP* with the master thread and then set the value of workers' thread-private data based on the master thread-private data, another pointer–*shadowP*–is defined as a static member (not thread-local) of OC. The value of *shadowP* always equals to *offsetP*.

In the master thread, when a new SI is created, the constructor must finish some additional actions before instance initialization. First, it calls *OC::newInstance* to get an instance ID and use the returned value to set SI.ID. Second, it allocates memory of type TPC and set *OC::offsetP[ID]* to the address of the newly allocated memory. Third, this constructor initializes the new instance in a standard way. Furthermore, we must re-define the access to original data members to be access to thread-private data. The definition using C preprocessor macros works perfectly if each thread-private member has a globally unique name. If so, we just use "*#define CMname (OC::offsetP[ID]->CMname)*", where *CMname* is the corresponding class member name.

For worker threads, all SIs are shared and should not be constructed once again. Any worker thread just sets up for each SI its thread-private data. The programming pattern used by Geant4 team enables worker threads to access SIs. Because the number of SIs is huge, it is worthwhile to share these instances. For exactly the same reason, Geant4 usually uses a storage class to register all instances. We can enumerate each element in this storage class to handle all SIs uniformly.

First, worker threads allocate space for their own *offsetP*. Then for each SI, they allocate memory of TPC type and set *OC::offsetP[ID]* to the address of new created TPC instance. Finally, they initialize thread-private data using instructions similar to SC constructor(s). If all constructors are idempotent, it does not matter if we execute the entire SC constructor(s).

Sometimes, there are different subclasses for SC. Under such circumstance, worker threads dynamically cast SC to the exact subtype and handle the initialization particular to this subtype. Some typical examples in Geant4 are listed below.

Figure 5 is the data model for shared logical volumes. For each G4LogicalVolume instance, both master thread and worker threads have their own thread-private data.

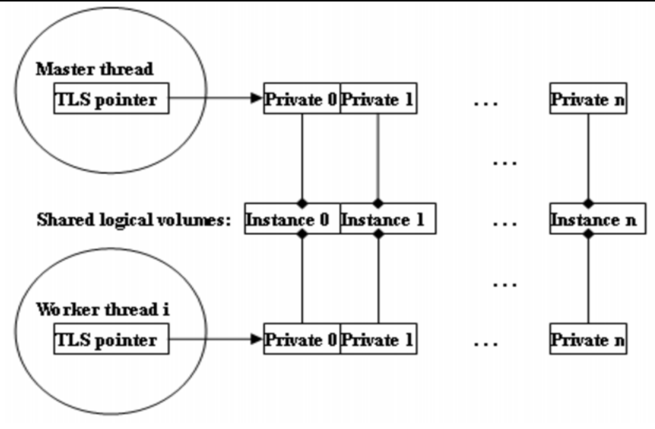

Fig. 5. Data model for shared logical volumes.

Figure 6 is the data model for shared physical volumes. There are three subtypes. For a G4VPhysicalVolume instance, master thread and worker threads share the whole instance even the thread-private data, which does not change. G4PVReplica is a subclass of G4VPhysicalVolume. For a G4PVReplica instance, both master thread and worker threads have two examples of thread-private data: the thread-private data as a G4VPhysicalVolume instance; and the thread-private data as a G4PVReplica instance. G4PVParameterised is a subclass of G4PVReplica. Just like the class G4PVReplica, both master thread and worker threads hold two examples of thread-private data for each G4PVParameterised instance.

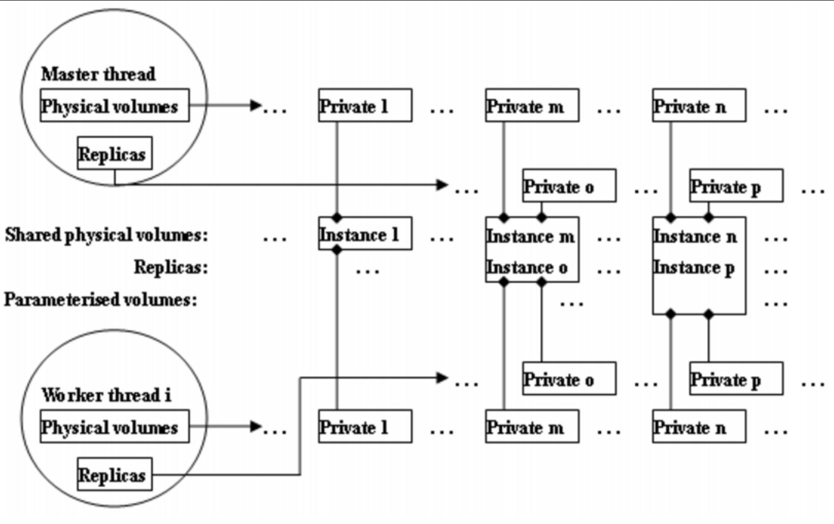

Fig. 6. Data model for shared physical volumes.

Consider a thread-private data member of pointer type. In our methodology, it will be moved to TPC. If only a pointer is changed, there is nothing special thing to do. However, if the instance pointed by this pointer is changed (mutated), then in each worker thread, we must give this private pointer a clone of the instance held by the master thread.

For a G4PVParameterised instance, the solid pointer in the corresponding logical volume belongs to this case. Figure 7 illustrates the data model to handle this solid pointer. In this figure, logical volume q is used by a G4PVParameterised instance, whose ID is 0. The solid pointer of logical volume q is thread-private. Furthermore, for each worker thread, a cloned solid is associated to the solid pointer of logical volume q.

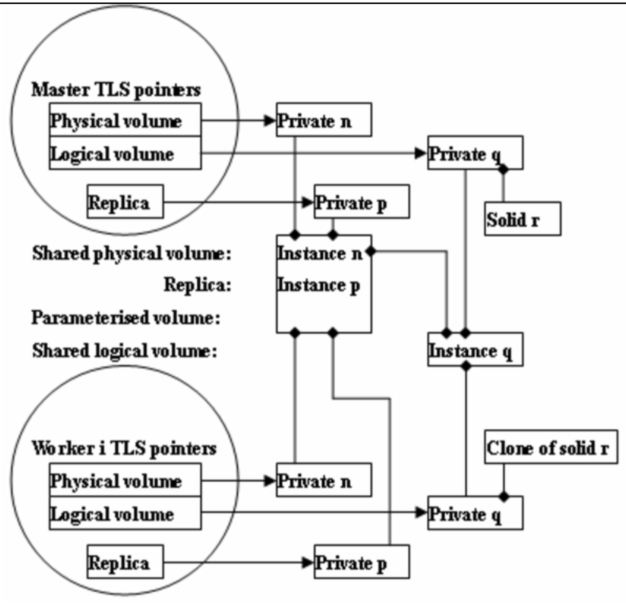

Fig. 7. Data model for a cloned solid.

To avoid the need for end-user programming for worker thread initialization, we require that any Geant4 class which has the usage mentioned above must support a clone method. For example, G4VSolid subclasses that are allowed to be used by any G4PVParameterised instance should implement a clone method.

## 5 Summary and future work

To implement multi-threading for Geant4 applications, we make Geant4 thread-safe with no change to the original application logic and design. First, Geant4 is semi-automatically transformed to be absolutely thread-safe. Then we manage to move some data back to data segment and share it. If all inward and outward pointers are set up correctly between shared data and private data, it is still thread-safe. In order to share data in multi-threaded Geant4 applications, we provide a methodology to share some data members for a set of C++ instances of a given class. This methodology has been used to share three kinds of geometry instances: physical volumes, logical volumes, and solids. Moreover, materials are shared completely.

For fullCMS, the image size for single thread is 30MB for text, 70MB for geometry and 50MB for physics. In geometry, about 50MB data belongs to voxelization, i.e., tracking optimization of geometry. Taking advantage of this work, almost all geometry data can be shared by threads except those thread-private data. However, the size of thread-private data is very small. Therefore, the memory consumption for each worker thread is expected to be 50MB, which is the size of physics. On a 32-bit machine with 4GB memory, we are able to start 50-70 worker threads given sufficient cores. Currently, Geant4 team is implementing clone method for solid subclasses that are used by G4PVParameterised instances. We will see the effect of shared geometry on image size reduction for fullCMS in the near future.

The above result is better than the result of process-parallel Geant4 using the copy-on-write technique. In process-parallel Geant4 case, each process has 80MB shared data which are dirty. Following the same methodology, we continue to reduce the image size by sharing more instances in multi-threaded Geant4 applications. In future implementation, instances from region to particles are shared among all threads, as described in Figure 8.

When all threads share particle tables, the process manager for each particle should be thread-private. Because the type of particle table is Map, any access to it should hold a thread lock. In this aspect, ions influence the way to share particles greatly. Some ions are initialized by run manager. Therefore, worker threads must treat this kind of ions as the ordinary particles and initialize process manager for them when they initialize thread-private data. Other ions are inserted into particle table on the fly. We must consider two cases when one ion of this kind is inserted into particle table. If this ion is not in particle table, it is inserted into particle table in a standard way. If this ion has been in particle table, the current thread only initializes thread-private process manager for this ion and for itself.

If we are able to share particles successfully, we can go one step further to share physics tables and physics vectors. To this end, we require that each process is able to clone itself while the clone shares physics table with the original process. All these efforts toward greater data sharing make multi-threaded Geant4 more scalable on multi-core machines.

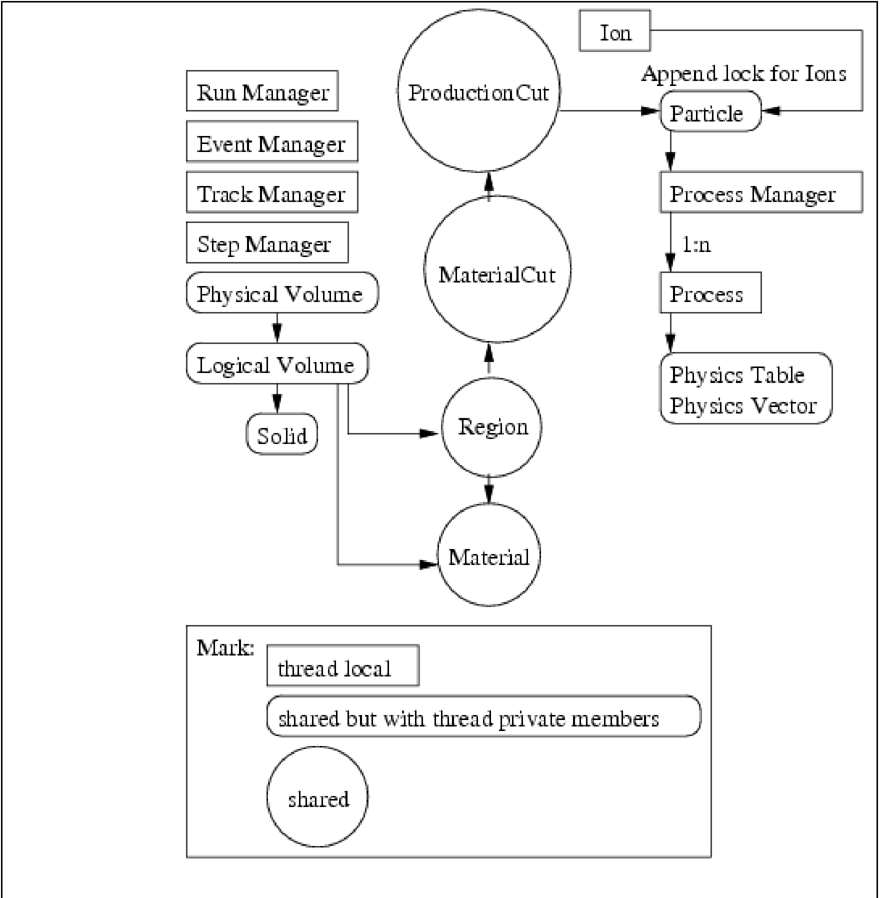

Fig. 8. Data model for future implementation.

## 6 References

[1] G. Cooperman. Practical task-oriented parallelism for Gaussian elimination in distributed memory. *Linear Algebra and its Applications*, 275.276:107.120, 1998.
[2] G. Cooperman. TOP-C: A library that links with your existing sequential code (after small modifications) in order to parallelize it, 2003. http://www.ccs.neu.edu/home/gene/topc.html.
[3] Elsa: An elkhound-based C++ parser. http://www.cs.berkeley.edu/~smcpeak/elkhound/sources/elsa/.
[4] Geant4. http://geant4.web.cern.ch/geant4/.
[5] R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca, and A. Lumsdaine. Open MPI: A high-performance, heterogeneous MPI. In *Proceedings, Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, Barcelona, Spain, September 2006.
[6] High energy physics. http://geant4.web.cern.ch/geant4/applications/hepapp.shtml.
[7] MPI for many important platforms. http://www-unix.mcs.anl.gov/mpi/mpich2.
[8] A high performance message passing library. http://www.open-mpi.org.
[9] Pargeant4 home page. http://www.ccs.neu.edu/home/gene/pargeant4.html.
[10] Instrumentation framework for building dynamic analysis tools.http://valgrind.org/.