Geant4 MT transformation

A mini-guide to develop code that works with a G4 multi-threaded application for non-experts.

Introduction

This is not a guide to write multi-threaded code, but gives a prescription to be followed by Geant4 developers. Geant4 multi-threaded is a transformation of Geant4 code that allows to run multi-threaded applications that support event-level parallelism.
The design of multi-threaded Geant4 is such that the changes needed to adapt our source code to be thread-safe are relatively simple. So simple that tools were developed to automatically transform the code. However new developments, should be ported to Geant4-MT under the responsibility of the the developer. If you follow strictly these guidelines you should be able to transform your code in few simple steps.
You should be aware of two important points:

1. We do not discuss the case of split classes (i.e. how to make part of data members available to all threads, while keeping a part of them thread-private). Geometry module and electromagnetic tables are the only places where this happens.
2. If you follow the following instructions your code will work in a multi-threaded application. A small CPU penalty may appear. However your first goal should be to make the code work in multi-thread. If you think that performances are an important aspect of the code, it is better to consult with some of the MT experts in the collaboration to discuss alternative strategies.

Geant4 classes in MT applications

Geant4 MT uses an event level parallelism. This means that more than one G4Event is processed at the same time. To minimize access to shared resources and reduce memory footprint the read-only parts of the most memory consuming classes are shared (geometry and electromagnetic tables), while all other class instances are thread private.
You can imagine a Geant4 MT application as a set of clones of the same application sharing part of the memory.
This approach largely simplifies the changes that are needed to the code. In particular it turns out that the only modifications of the code that are needed are the declaration of **global** and **static** variables.
The problem with this type of variables is that they exists only once in memory and thus are shared among all threads. This can be a problem if the variable holds a state that is not invariant between events (it should be noted, however, that if the variable is also const there is no need to transform it).
The change that is needed is to add a special C++ keyword that instructs the compiler to give each thread a private copy of that variable. In Geant4 the file tls.hh defines **G4ThreadLocal** keyword to be an alias to the correct os-dependent keyword (for example for linux G4ThreadLocal is equivalent to __thread).
To apply the changes to the code you need to change all definitions and, in some cases, the use of static and global variables (note that global variables should never be used in Geant4, only in very special cases should be used). Consider the following example:

Example 1:
G4mysourcefile.cc contains at global scope:

```
G4double aGlobalVariable; //This does need a change
const G4double pi = 3.1415; // This does not changes
```

G4myheaderfile.hh contains:

```
class G4class {
 private:
   G4double aVariable; // This does not need changes
   static G4double anotherVariable; // This does need changes
   static G4anotherclass* aPointer; // This does need changes
};
```

How to transform the code in case of a simple data type

If your variable is of a simple data type (G4int, G4double, G4bool, or any pointer) it is very straigthforward to transform the code: add the keyword G4ThreadLocal after the static keyword. Following the Example 1, change the code to:

Example 1:
G4mysourcefile.cc contains at global scope:

```
G4double G4ThreadLocal aGlobalVariable;
const G4double pi = 3.1415;
```

G4myheaderfile.hh contains:

```
class G4class {
 private:
     G4double aVariable;
     static G4ThreadLocal anotherVariable;
     static G4ThreadLocal G4anotherclass* aPointer;
};
```

Note that pointers to any data type can be safely treated in this way.
Special attention may be required if the static variable is defined inside a method implementation or any other function: see Example 2.

Example 2:

```
void
G4class::aMethod() {
  // ...
  static const G4double pi = 3.1415;
  static G4double aVariable = foo();
  //...
 return;
}
```

Transform the code to:

```
void
G4class::aMethod() {
  // ...
  //Note that this variable does not need to be
  //transformed since it is a const read-only
  //and can be safely shared among threads.
  static const G4double pi = 3.1415;
  static G4ThreadLocal G4double aVariable = foo();
  //...
 return;
}
```

The compiler may complain about the modified line depending on how function foo() is defined. In such a case some additional lines have to be added:

```
void
G4class::aMethod() {
  // ...
  //Note that this variable does not need to
  //be transformed since it is a const read-only
  //and can be safely shared among threads.
  static const G4double pi = 3.1415;
  static G4ThreadLocal G4double aVariable = 0;
  static G4ThreadLocal G4bool aVariableIsInitialized = false;
  if ( aVariableIsInitialized == false ) {
     aVariableIsInitialized = true;
           aVariable = foo();
  }
  //...
 return;
}
```

How to transform the code in case the variable is not of a simple type

The compiler can handle thread local storage (i.e. G4ThreadLocal) only for simple types (G4double, double, G4bool) and pointers. For complex data types a modified version of the change is needed.
Consider the following code to be transformed:

Example 3:
G4myClass.hh:

```
class G4myData {
    void doSomeWork();
};

class G4myClass {
 private:
      static G4myData myDatum;
 public:
      void aMethod();
};
```

G4myClass.cc:

```
void
G4myClass::aMethod() {
   // ...
   myDatum.doSomeWork();
}
```

Since myDatum is not of a simple type the following more complex transformation is needed:

G4myClass.hh

```
class G4myClass {
  private:
      static G4ThreadLocal G4myData* myDatum_G4MT_TLS_;
  public:
      void aMethod();
};
```

G4myClass.cc:

```
G4ThreadLocal G4myClass::myDatum_G4MT_TLS_ = 0;

void
G4myClass::aMethod() {
  if ( ! myDatum_GMT_TLS_ ) myDatum_GMT_TLS_ = new G4myData;
  G4myData& myDatum = *myDatum_GMT_TLS_;
  //...
  myDatum.doSomeWork();
}
```

The static data member is transformed into a pointer that is initialized to the correct value when needed.
For Geant4 MT we have a convention for the transformed variable name: use the same name of the original variable followed by _G4MT_TLS_.
Please use this convention! In this way it is very simple to “grep” our entire source code in search of these cases if needed.
You may wonder why in this example we add the line:

```
G4myData& myDatum = *myDatum_GMT_TLS_;
```

instead of using simply:

```
myDatum_GMT_TLS_->doSomeWork();
```

This choice is to avoid to change all the code that you have already written. This method has an additional advantage: changing the name of the variable will help you to debug. The compiler will give an error (myDatum undefined) in all places where you use myDatum and you did not add the two lines at the beginning of the method. Again we require that you follow a convention: **always add the two lines at the very beginning of all the needed methods.**

Conclusion

If you follow the simple prescriptions of this document you will be able to change your code for Geant4 multi-threaded applications. If you are not familiar with multi-threaded programming we strongly encourage to follow **exactly** these instructions and consult with experts in case of problems.
If you need to review your code or develop new one and you are using the “static” keyword it is a good opportunity to ask yourself if you can declare the variable const. Do you want to define a const object that will be only accessed “read-only” once initialized?
If so add the “const” specification and you do not need to change anything for multi-threading (the variable “pi” in examples 1 and 2).