Многопоточные приложения в Geant4

В Geant4, начиная с десятой версии, была реализована поддержка многопоточных приложений в рамках ОС UNIX.

Современные операционные системы уже достаточно долго поддерживают многозадачность, причем каждая из этих задач вполне может выполняться в несколько потоков. Однако такой подход к программированию существенно усложняет разработку. Не будем основательно углубляться в особенности многопоточного программирования и сконцентрируемся лишь на тех аспектах, которые встречаются в Geant4. До десятой версии в Geant4 все происходило последовательно: сначала инициировалось ядро, потом устанавливались настройки модели, такие как геометрия, процессы. Затем инициализировались действия, описывались первичные частицы и присваивались все опциональные объекты. По завершению настройки программа запускалась на расчет и каждое событие моделировалось тоже последовательно.

Отличие работы в многопоточном режиме можно объяснить следующим образом. Точно также, как и ранее инициализируется ядро, описывается геометрия и физические процессы. Все это происходит на потоке, называемом главным или мастером, и никак не отличается от не многопоточного приложения. Затем начинается инициализация объектов классов действий и здесь начинаются отличия. Метод Build() класса G4VUserActionInitiliztion позволяет создавать объекты для каждого рабочего потока – потока, работа которого начинается по команде «мастер потока». Одновременно может существовать более одного потока, причем на время существования рабочих потоков мастер поток переводится в режим ожидания, а все созданные им потоки работают независимо. Когда последний рабочий поток закончит свои операции, управление программой вернется на «мастер – поток». Это означает, что во время расчета цикл обработки будет осуществляться не на одном потоке,

а на потоках, количество которых указано в объекте класса G4RunManager. По умолчанию их число равно 2.

Сразу стоит отметить, что для изменения этого значения следует

```
void SetNumberOfThreads( G4int n );
```

вызывать метод

Не плохим решением, чтобы не определять сколько потоков доступно на той или иной расчетной машине, будет использование данного метода вместе со статическим методом на определение доступного числа ядер:

вместе эта конструкция будет выглядеть следующим образом:

```
G4int G4GetNumberOfCores();
runManager->SetNumberOfThreads(G4Threading::G4GetNumberOfCores());
```

Возвращаясь к методу Build() класса G4VUserActionInitialization, при его вызове происходит генерация независимых объектов для каждого рабочего потока.

Таким образом, одновременно существует n копий генерации первичных частиц, запусков, событий и т. п. где n количество потоков.

Теперь рассмотрим, что происходит во время запуска. Если на запуск было послано 1000 событий, это совершенно не означает, что они будут равномерно распределены между потоками. На самом деле будет создана очередь событий, из которой потоки будут выхватывать следующее событие, после чего моделировать его в своей замкнутой и независимой системе. Что же касается геометрии и физических процессов, то, как известно, их нельзя изменить во время цикла обработки событий, и они доступны только для чтения рабочим потоком. Подводя итоги, можно сказать, что осуществляется максимальная производительность, однако в финале получится, что один из потоков (допустим их 2) обработал 535 событий, а другой 465.

Из - за этого возникает необходимость постобработки расчетных данных или сшивки. В оригинальном Geant4 существует класс G4SensitiveDetector или метод Merge() класса G4RunManager, однако и то и другое требует

дополнительных условностей в коде. Информацию об их использовании можно попытаться найти на официальном сайте, однако в рамках данной главы мы воспользуемся стандартным инструментом многопоточного программирования - мьютексом.

Мьютекс представляет собой одноместный семафор. Задача мьютекса — защита объекта от доступа к нему других потоков, отличных от того, который завладел мьютексом. В каждый конкретный момент только один поток может владеть объектом, защищённым мьютексом. Если другому потоку будет нужен доступ к переменной, защищённой мьютексом, то этот поток блокируется до тех пор, пока мьютекс не будет освобождён. В наших задачах мы будем использовать мьютекс самым простым образом.

Рассмотрим пример:

Пусть результат вычислений хранится в переменной result в классе RunAction наследующем G4UserRunAction, и пусть имеется n объектов класса RunAction, каждый из который существует на своем рабочем потоке. Тогда необходимо сшить результаты в переменную f_result на мастер потоке.

Начнем с создания объекта, целью которого будет хранение суммарных данных и непосредственно сшивка. Тогда шаблон класса данного объекта будет выглядеть следующим образом:

```
class Scoring{
private:
   G4int f_result;
public:
   void setF_result(G4int f_result) {
      Scoring::f_result = f_result;
   }
```

Такая схема была бы верной в не многопоточном случае, однако, в данной задаче доступ на запись к переменной f_result будут пытаться получить все рабочие потоки, следовательно, метод setF_result следует защитить мьютексом.

Для этого в Geant4 уже предоставленна достаточно удобная оболочка для pthread.h.

```
G4MUTEX_INITIALIZER;
```

Сделаем мьютекс частью класса Scoring и инициализируем его.

```
class Scoring{
   G4int f_result;
   G4Mutex aMutex=G4MUTEX_INITIALIZER;
public:
   void setF_result(G4int f_result) {
      G4AutoLock l(&aMutex);
      Scoring::f_result += f_result;
   }
};
```

В свою очередь, для переключения мьютекса в закрытое состояние при вызове любым потоком метода setF_result будет служить класс G4AutoLock (на самом деле класс называется G4ImpMutexAutoLock), при вызове конструктора которого мьютекс переводится в состояние «занято», а при вызове деструктора (что происходит во время уничтожения локальной переменной l по завершению работы функции setF_result()) мьютекс переводится в состояние «открыто».

Такая структура класса Scoring защит переменную f_result от одновременного обращения двух рабочих потоков и обеспечит верность финального результата.

Что же касается объектов RunAction, то их вид будет достаточно тривиальным. Каждому из них следует передать указатель на объект класса Scoring, который будет существовать где-то под управлением мастер потока.

```
class RunAction: public G4UserRunAction{
   Scoring* scoring;
   G4int result;
   explicit RunAction(Scoring *m_scoring):scoring(m_scoring){}

public:
   void EndOfRunAction(const G4Run *aRun) override {
      scoring->setF_result(result);
   }
```

Такая структура объектов RunAction обеспечит автоматическую сшивку данных в финале запуска.

Объект же класса Scoring можно создать как поле потомка G4VUserActionInitialization, так как объект этого класса находится под управлением мастер потока, например,

```
class Action: public G4VUserActionInitialization{
public:
   Action::Action(){
      scoring = new Scoring();
   }

   ~Action() override{
      delete scoring;
   }

   void Build() const override{
      SetUserAction(new PrimaryPat);
      SetUserAction(new RunAction(scoring));
   }
```

Аналогично данному способу можно решать большинство многопоточных задач, возникающих в Geant4.

Однако в данном примере не было указано, как извлекать данные из объекта Scoring. Для решения этой задачи существует много путей, например, если математическая модель на Geant4 является частью более крупного приложения, то результат может напрямую передаваться в другую часть программы. Либо работу с объектом Scoring можно настроить с помощью пользовательских команд, о которых пойдет речь в следующей главе.