GSoC 2019 Project Proposal

Abstract

Pursuing the goal of running $Allpix^2$ simulation's events -independent by nature- in parallel, have led to the identification of performance bottlenecks that prevent $Allpix^2$ from fully utilizing available CPU cores and from scaling the execution time relative to the number of used cores. In this year GSoC, I propose to continue working on solving these bottlenecks, most importantly the Geant4 dependency by implementing a custom run manager encapsulated within $Allpix^2$ that would fix the scalability issues and allow for running events in parallel.

Contact Details

Implement Event based Seeding and Multi-Threading

Introduction

$Allpix^2$ is an easy to use simulation framework for silicon detectors implemented in modern C++. It allows "simulating the performance of Silicon detectors, starting with the passage of ionizing radiation through the sensor and finishing with the digitization of hits in the readout chip"[1]. The framework is designed in a modular way allowing it to only maintain a lightweight core that can be configured to run different experiments with different algorithms -modules- that are developed independently of the core. This gives both users and developers greater flexibility and ease of use to include/develop modules in the simulation.

Each Simulation is the execution of a configured number of independent events. For each event, included modules in the simulation are executed in the user-specified order and share information with each other using a message based approach that provides more encapsulation and increases the loose coupling between modules.

In last year GSoC[2], a new multi-threaded approach was implemented that would allow for events to run in parallel and $Allpix^2$ execution time was optimized for up to 8 threads. However, multiple bottlenecks have been identified that limited the framework's ability to scale when using more threads. Namely, the dependency of the Geant4 library that has its own model of multi-threading that contradicts with $Allpix^2$ thread management.

The goal of this project is to resolve the main bottleneck for running independent events in parallel by implementing a custom Geant4 run manager that conforms with $Allpix^2$ architecture. This will allow $Allpix^2$ to fully leverage multi-core execution environments that will greatly improve the simulation time with millions of events.

Getting Started with Allpix2

This section describes my first contact with $Allpix^2$ and the lessons learned while working on the evaluation task. The goal of this section is to summarize my interaction with the project and highlight the things that I have learned about even before working on the actual project.

My first step was to build $Allpix^2$ on my local machine which was a smooth process after figuring out the needed Geant4 configuration. Then I started looking at the user manual and some of the examples provided in the project's repository to get comfortable with the program as an end user. Also, most importantly, I was looking for more information to understand the simulations and how they work. I have also found multiple resources online[3][4][5] that helped me understand what is a silicon detector and how it works.

My second step was to start looking at the code base which I have found to be very well documented and structured. I really liked the architecture of the framework because it separates the module algorithms from its core infrastructure allowing the framework to be very generic and agnostic about the simulation algorithms it runs. Also from the module's perspective, there is no need to know any details about the core infrastructure or about other modules; communication is made by messages.

By the time I felt motivated and confident enough to apply for this project, I contacted the project's mentors to discuss the project idea and start working on evaluation tasks as needed which was in two parts; carry out a simulation and describe the outcomes and implement a simple framework in C++ that can run events in parallel. I found that these two tasks helped me understand the framework, the project and, its challenges.

In the first task, I dug deeper into how to use the framework. I played around with different parameters and observed the change in the results. While doing so I got to learn how to use ROOT to analyze the simulation results. I have gone through the basic courses from the ROOT online tutorial website[6]. Also, the mentors were really helpful in answering my questions which helped me carry out this task. My configuration files and results for this task can be found in this shared folder.

In the second task, I got to explore the project idea in a simplified example. The challenge of running events in parallel while keeping the order of the output the same -so that simulations are reproducible. I took advantage of doing this task to try out different approaches about running events in parallel in general that can also be applied to the framework. I used instrumentations and measurements to compare different approaches based on the tradeoff between performance and memory usage. My repository for this task along with a report comparing the different approaches I have tried is available on Github.

Related Work

In this section, I will discuss the current state of the framework and the related work that was carried out in GSoC18. The goal of this section is to summarize the current state, discuss some concerns and try to conclude with some lessons that will be used during this project.

The production version of the framework currently supports a form of multithreading in terms of running modules that are independent of each other in parallel within the same event. So the parallelization happens within each event itself.

GSoC18[7][8] saw the first attempt to run events in parallel. The following issues were identified and dealt with. I will discuss them in the order of importance to this year project.

Geant4 Dependency

Geant4 is used by certain modules in the simulation flow for building the world geometry and depositing charges through the detector. It is useful because it encapsulates a wide range of particles and physical processes that can be used in a simulation.

The way Geant4 public API is structured is more oriented on the approach of using it as a framework where it takes control of executing the event loop and user supplies callbacks/actions that are executed when appropriate by Geant4 run manager. This contradicts with $Allpix^2$ architecture. And as a result, the multithreading capabilities -already built in Geant4- can't be used by the framework.

The current use model within $Allpix^2$ is to turn off the multithreading of Geant4 and for each event call the library once -in contrary of calling the library once and tell it to execute a specific number of events in parallel. To overcome this issue -when going parallel, all Geant4 modules are executed sequentially on the main thread before queueing the event for parallel execution.

To conclude this is the main bottleneck that needs to be addressed during the project. To do so the following issues need to be further investigated:

1. Investigate how Geant4 event loop works.
2. Identify the main steps carried by Geant4 run manager that our custom implementation needs to do.

Seed Distribution

To allow simulations to be reproducible, the order of seeding the random number generator for each event must be kept the same with the non-multithreaded run. To overcome this issue, the new class *Event* was equipped with its own random number generator.

Since the random number generator of type Mersenne Twister[11] used in the framework is a heavy object and keeping a big number of this object as the number of events would be practically impossible, the choice was made to maintain only a specific number of *Event* objects equal to the number of threads used multiplied by 4 and this is enforced by polling the thread pool internal queue size every 50ms.

My concern about this approach is that it introduces an overhead by setting a limit to the buffer of available *Event* objects that can be used by thread workers. While yes this can be needed for example if we can't keep simulation results in memory and we have to write them in patches but I didn't see enough measurements carried out to support this decision or the heuristic regarding the buffer size limit.

To conclude, the following issues need to be further investigated:

1. Explore the idea of moving the random number generators from *Event* class to be allocated per thread using the C++11 *thread_local* storage specifier.
    - This keeps the number of random number generators created to the minimum.
2. Carry out memory profiling of the framework to understand the memory usage and accordingly update any heuristic about it.

Reader & Writer Modules

In case a module is reading or writing to a file, and in order to keep the simulation reproducible, modules that require disk access were made to run in a sequential order to guarantee that they produce the same results.

The problem with this approach is that it creates a global bottleneck for all events running in parallel. Since all events have to wait for the correct event to execute its modules-of-interest first. Furthermore, the proposed solution was not benchmarked and therefore we can't tell how good is it since the reported performance improvements in GSoC18 report didn't use any writer/reader modules in the configuration used.

To conclude, the following issues would need to be further investigated:

1. Measure the cost of this approach with respect to execution time and memory usage.
2. Investigate the idea of buffering file writes to decrease the need for waiting for the correct event to execute first.
3. Investigate other possible solution based on the outcome of 1 & 2.

ROOT Dependency

One of the bottlenecks reported from GSoC18 is the dependency on ROOT *TObject* and *TRef* classes that are used to relate objects created during the simulation with each others allowing for further data analysis after executing the simulation.

The reported problem was that these objects don't scale well when multiple threads try to manipulate them. Although a patch[12] was made to address this specific issue, it caused some other problems within the framework that was not further investigated.

To conclude, the following issues would need to be further investigated:

1. Reproduce and investigate the crash caused by ROOT patch.
2. Investigate the idea of removing the ROOT object inheritance and constructing them just before writing simulation results.

Deliverables

This section defines the set of deliverables that I will be working during the project to complete. These will be the main outcomes that I will focus on completing to consider this project as a success.

1. Runtime and Memory Benchmark of $Allpix^2$.
2. Implementation of a custom Geant4 run manager within $Allpix^2$.
3. Benchmark $Allpix^2$ multithreaded execution.
4. Writing tests to cover the new use model.
5. Writing and updating documentation.

Additionally, I would think of other deliverables as optional which means that they will be a secondary goal and have lower priority depending on the time.

1. Performance optimizations for seed distribution.
2. Performance optimizations for Reader/Writer modules.
3. Investigating ROOT dependency.

Analysis

In this section, I will go into the details of my analysis of Geant4. Such details will be useful in defining what do I really need to implement in the custom run manager to make it work in a multi-threaded environment.

The first part of my analysis is to examine how the framework uses Geant4 to identify the main classes and methods involved. The modules that currently use Geant4 are *GeometryBuilderGeant4* and *DepositionGeant4*.

The main Class involved is *G4RunManager* which exists as a singleton within the entire application that get created in *GeometryBuilderGeant4* then passed to *DepositionGeant4* for later usage. Having a closer look I can identify the main method used to be *BeamOn*. Trying to run these modules in parallel caused an exception to be thrown and the following error message to be printed on standard error: “Illegal application state - BeamOn() ignored.”

Studying the Geant4 codebase[9] this method can be summarized in the following pseudocode:

```
BeamOn()
   ConfirmBeamOnCondition()
       if (state == BAD) {
                  cerr << "Illegal application state - BeamOn()
ignored.";
           return false
       }
    ConstructScoringWorld()
    RunInitialization()
    DoEventLoop()
        InitializeEventLoop()
        ProcessOneEvent()
            GenerateEvent()
                return new G4Event
            eventManager->ProcessOneEvent()
            AnalyzeEvent()
                G4VPersistencyManager->store()
                G4Run->RecordEvent()
            UpdateScoring()
        TerminateOneEvent()
             StackPreviousEvent()
        TerminateEventLoop()
   RunTermination()
```

The following are the main classes involved and their responsibilities:

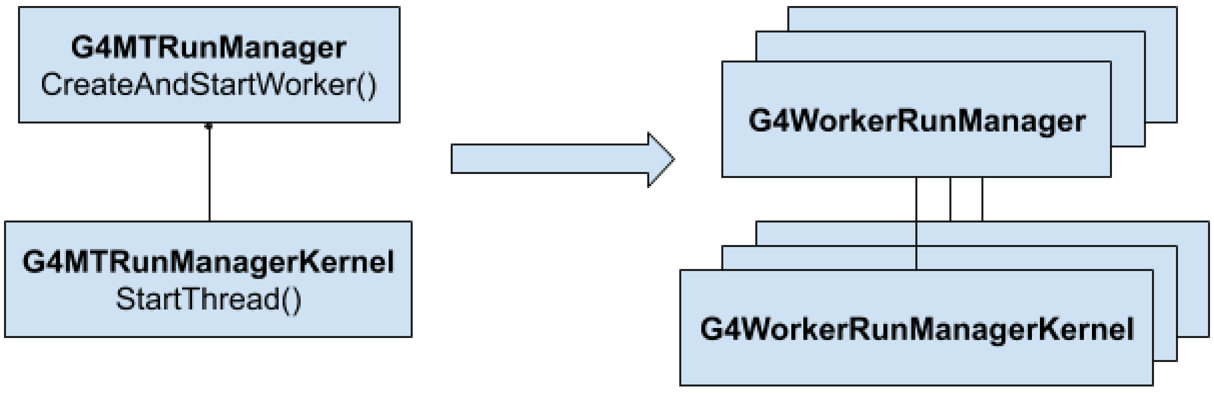
- *G4RunManager*: Singleton that manages the event loop, and all other objects exposing the BeamOn API to clients.
- *G4RunManagerKernel*: Singleton that does the actual job under the hood. Can be used by a custom implementation if and only if it is correctly initialized.
- *G4StateManager*: Maintain the state of the *G4RunManager*. For example: processing an event, writing output, idle, etc...

A good point to start is to look at the prepared set of examples by the project’s mentors[10]. There is another run manager in Geant4 that allows for running events in parallel but under this manager’s control. This class *G4MTRunManager* is a child class of *G4RunManager* that overrides some of its methods to implement a multithreaded flow. Then comparing the two classes will show the needed steps to use the underlying kernel into the custom implementation of the run manager.

Further investigation of *G4MTRunManager* shows that this is used with the association to another manager; *G4WorkerRunManager*. The first is used by the main thread to spawn the required threads while the latter is created for each thread to manage the thread's execution.

Accordingly, the *G4MTRunManager* has an instance of the class G4MTRunManagerKernel which is a child class of *G4RunManagerKernel* and contains the threads' start method *StartThread*. For each thread, the *G4WorkerRunManager* instance with the associated *G4WorkerRunManagerKernel* is used to execute the same logic that is executed in the normal flow.

The following diagram summarizes how Geant4 operates in the multithreaded mode. The worker classes are defined as thread local and therefore once the threads are created each thread will have its own copy of the worker manager and kernel and carry on the work accordingly.



Implementation

To identify the steps needed to implement the custom run manager, we can make a similar approach based on the implementation of *G4MTRunManager*. In this repository, I have started with a clone of both *G4MTRunManager* and *G4WorkerRunManager*. Then I modified their implementation as follows:

1. Remove all parts dealing with thread management.
2. Make the worker manager instantiated per thread on the module's run method.
3. Initialize the main run manager on the main thread.

So far I consider this as a work in progress and I may require some time to finalize the implementation plan and identify the exact needed steps to ensure correct functionality of Gean4 modules. I would expect that this effort can be done -in the worst case- during the community bonding period to have the implementation plan ready when I start the coding period.

Benefits to the Community

The goal of this project is to release a new version of *Allpix*^{2} full equipped with the required infrastructure to utilize multi-core execution environments and dramatically reduce the simulation time needed for millions and even tens of millions of events. This would allow scientists to carry out complex simulations that would otherwise take much longer to execute. And the cost of errors, misconfiguration, changing parameters or even trying to reproduce the results would be greatly reduced.

Timeline

In this section, I provide a basic timeline for the project with the estimated duration for working on the previously mentioned deliverables. Since the GSoC timeline is divided into 3 phases each with around 3 weeks of work period followed by an evaluation period, my planning will be around the same idea dividing the work into 3 phases as follows:

Duration	Tasks
Community Bonding Period	● Familiarize with the community, agree on communication channels and communication frequency. ● Further study code base and Geant4 ● Carry out Performance and Memory Benchmarks ● Finalize the implementation plan for the run manager
Phase 1 27/05 - 23/06	Implement Custom Geant4 run manager
27/05 - 09/06 (2 weeks)	● Implement the first version of Geant4 run manager
10/06 - 16/06 (1 week)	● Testing to verify simulation correctness
17/06 - 23/06 (1 week)	● Fix bugs & update implementation
First Evaluation 24/06 - 28/06	
Phase 2 01/07 - 21/07	Performance Benchmarks and optimizations
01/07 - 07/07 (1 week)	● Carry out performance and memory benchmarks
08/07 - 21/07 (2 weeks)	● Analyze performance reports ● Refactor as necessary
Second Evaluation 22/07 - 26/07	
Phase 3 29/07 - 18/08	Performance optimizations and improve test coverage

29/07 - 04/08 (1 week)	● Write functional tests ● Write performance tests
05/08 - 11/08 (1 week)	● Analyze performance reports ● Fix bugs & continue performance optimization
12/08 - 18/08 (1 week)	● Prepare final benchmarks ● Write tests ● Fix Bugs
Final Evaluation 19/08 - 26/08	

The evaluation periods will be dedicated to writing the necessary documentation for both project documentation and GSoC evaluation.

Availability

Additional Information

Contribution

The following is the list of pull requests I have submitted and been merged into the project's master branch by the the time of submitting this proposal:

1. Add log message when successfully applying options from the command line.
2. Allow Parsing of Scientific Notations.

Motivation

I am motivated to participate in this project because I want to:

1. Improve my proficiency in C++. I am looking to work with C++11/14 and improve my knowledge about the standard libraries which is currently lacking.
2. Build my set of expertise in computing performance. I am interested in problems in the fields of high-performance computing and performance optimization. I am looking for challenges to further improve my skills in this area.
3. Contribute to both the scientific and open source communities. While I have previously contributed some patches to some open source projects -eg: 0AD and jitsi- I didn't participate in a big open source project before. Also, I am looking forward to making a contribution to a big organization like CERN. This will be something I am proud to have achieved in my lifetime.
4. Explore different applications of computing. I love how computing nowadays is helping people in all domains to solve their problems. And of all domains, I am more passionate about applications in the scientific fields. I have heard before on multiple occasions about Monte Carlo simulations but didn't have the chance to understand it, maybe it's time now to do so.

About

References

1. Allpix User Manual. https://project-allpix-squared.web.cern.ch/project-allpix-squared/usermanual/allpix-manual.pdf
2. GSoC18 project submission. https://github.com/allpix-squared/allpix-squared/pull/1
3. Silicon Single-electron Transfer and Detection Device -YouTube. https://www.youtube.com/watch?v=UguFUpcV5Ew
4. Silicon Tracker. http://www.ams02.org/what-is-ams/tecnology/tracker/
5. The Silicon Pixel Detector. http://aliceinfo.cern.ch/Public/Objects/Chapter2/DetectorComponents/silicon_pixel_detector.htm
6. ROOT Courses. https://root.cern.ch/courses
7. GSoC18 Submission. https://gitlab.cern.ch/allpix-squared/allpix-squared/merge_requests/159
8. GSoC18 Presentation. https://indico.cern.ch/event/748000/contributions/3093860/attachments/1699348/2736175/CERN_presentation_-_Allpix_Squared.pdf
9. Geant4 Doxygen Reference. http://www.apc.univ-paris7.fr/~franco/g4doxy4.10/html/index.html
10. MultiThreading Examples with Geant4. https://github.com/simonspa/geant4-multithreading
11. Mersenne Twister C++ Reference. http://www.cplusplus.com/reference/random/mt19937_64/
12. ROOT patch. https://github.com/root-project/root/pull/2381