Evolution of the STAR Framework OO model for the Multi-Core era

Valeri Fine[1]
Brookhaven National Laboratory
PO Box 5000, NY 11973, USA
E-mail: fine@bnl.gov

Jérôme Lauret
Brookhaven National Laboratory
E-mail: jlauret@bnl.gov

Victor Perevoztchikov
Brookhaven National Laboratory
E-mail: perev@bnl.gov

With the era of multi-core CPUs, software parallelism is becoming both affordable as well as a practical need. Especially interesting is to re-evaluate the adaptability of the high energy and nuclear physics sophisticated, but time-consuming, event reconstruction frameworks to the reality of the multi-threaded environment. The STAR offline OO ROOT-based framework implements a well-known "standard model" composed of chained modules, where input for each module is the output of the other modules. At its basic principle, modules do not communicate with each other directly and act as consumers and providers of data structures. They use the framework via the special “query” / “publish” API to query the presence of the input data and publish the output results the modules produce. As a proof of principle, we will show that by complementing the base framework with the ability to start several modules in parallel and synchronize the global data access between the "consumer" modules and "producer", one can transparently enhance the existent packages to leverage the multi-core hardware capability. Such conceptual design modification should allow re-using the existing offline software, initial built with single thread architecture and embarrassingly parallel processing in mind, in the multi-threaded environment if needed. However, we realize that the "query"/"publish" paradigm is not sufficient to run the multi-threaded application effectively. It should be complemented with an API to “register” the module output to notify the framework members about an output dataset “to be produced soon”. With such addition, the receiving module thread can be automatically suspended if the data it has requested is not ready yet. We will explain how the STAR Offline Framework was modified to test the present approach by building the sophisticated interactive real time applications.

XII Advanced Computing and Analysis Techniques in Physics Research
Erice, Italy
3-7 November 2008

[1] Speaker

1. Introduction

The Solenoidal Tracker At RHIC (STAR) is a large acceptance collider detector which started data taking at the Brookhaven National Laboratory in summer 2000.

At the time, STAR developed a single all-purpose yet modular ROOT-package-based Object-Oriented framework for simulation, reconstruction and user analysis in offline production also providing the capability to perform interactive physics analysis or online monitoring [1]. It implements a well-known architecture model composed of chained modules, where input for each module is the output of the other modules. Such scheme did fit well the single thread batch enviroment and every unit of work was organized in a tree-like sequence. In recent times, the evolution of the hardware and CPU architecture has made multi-core and multi-CPU node not only affordable but a response to a growing computing power demand facing limitations in power consumption, cooling or even organizational aspects such as floor space. Under such new reality, it is especially interesting to re-evaluate the adaptability of the STAR framework and similar high energy and nuclear physics sophisticated, but time-consuming, event reconstruction applications to the reality of the multi-threaded environment.

Base principles for a proven and production-mode framework build over a development cycle of the order of ten years implies that any appealing solution requiring changes of the core code and algorithms by STAR scientist are not acceptable. In other words, any solution for such mature experiment like STAR must be backward compatible and "transparent" to minimize development cost, further time investment and possible disruptions in productivity.

2. STAR Framework OO Model

The STAR framework was designed to support chronologically chained components, which can themselves be composite sub-chains, with as components "makers" (data object owners) managing named "datasets" they have created and are responsible for [2].

"Makers" and "datasets" inherit from the **`TDataSet`** class which supports their organization into hierarchical structures for management.

`TDataSet` also centralizes almost all system tasks:

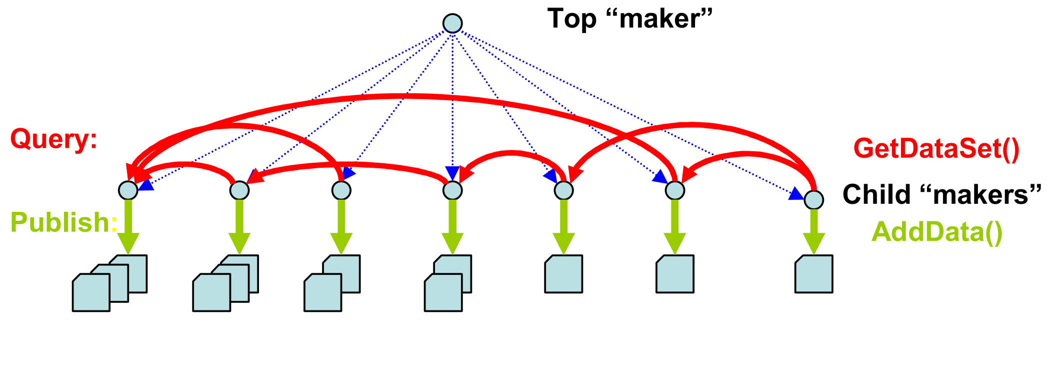
- data set navigation,
- I/O, database access,
- inter-component communication.

The main features of the STAR framework design:

- its "makers" and "datasets" share a common object model
- they derived from one and the same base TDataSet class
- the entire STAR reconstruction chain is a single instance of the StChain class (subclass of TDataSet)

At its basic principle, modules do not communicate with each other directly and act as consumers and providers of data structures. They use the framework via the "query" represented by method **`StMaker::GetDataSet`** / "publish" represented by method

`StMaker::AddData()` API to query the presence of the input data and publish the output results the modules produce (Figure 1).



Each maker can query the data published by the other makers.

Figure 1. STAR framework "Query"/"Publish" modules communications[2]

3. The "transparent" steps towards multi-core era

"By design" each framework module can work independently of any other modules as soon as its query and demands for data-sets are satisfied. This suggested one could "transparently" enhance the existent packages to leverage the multi-core hardware capabilities. To do that it should be sufficient to

- complement the TDataSet-base framework with the ability to start several modules in parallel
- provide some facility to synchronize the global data access between the "consumer" modules and "producer"

3.1. Evolution – main directions

The average "reconstruction chain" comprises of four main steps namely, "Init", "Clear", "Make" and "Finish" corresponding to respectively a main initialization step, an event loop (a method which will be called for each "event"), and a garbage collection or cleanup needed to be done upon completion of all event processing. Since the first and last steps of the chain are invoked once during a regular production session and the "Event Loop" is called as many times as we have events, one should expect a significant performance gain mainly from the parallelization of the "Event Loop", leaving fore the time being the first and the last steps unchanged and singlethreaded (or implementing parralelization outside the scope of a "chain" parralelization).

To go further we are concentrating our effort to implement and test the crucial components such as "transparent"

- parallelization, synchronization

- registration

3.1.1. "Transparent" parallelization and synchronization

To introduce the parallelization and synchronization of the **`Make()`** methods called from for the different "makers" of the STAR production "chain" and this in a "transparent" manner, we found it is enough to

- re-implement the methods **`Init()`**, **`Make()`**, **`Finish()`** and **`Clear()`** in STAR's StMaker base class
- add the extra thread-related private data-members to the base classes. Especially, an additional data-member for the thread ID is needed to distinguish the object owned by the different threads;
- make methods "atomic";
- allow generating or using as many threads as many "makers" are present on the "next level of hierarchy" of the **`StChain`** class instance instead of the calling the methods in loop.

Even though the average STAR chain consists of hundreds [2] of different "makers", the proposed solution does not lead to an application with hundreds of concurrent threads. Within the STAR framework, the change of the hierarchical level does not change the ability to query the data and does not require changing any low-level implementations either (Figure 2).

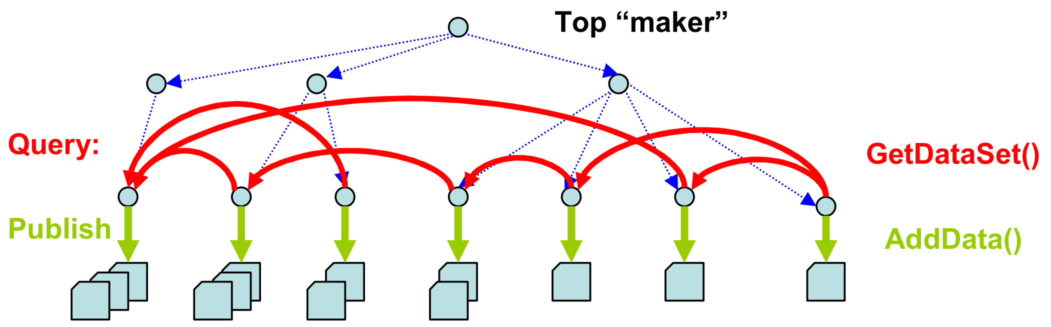


Figure 2. Adding the extra level of the "maker" hierarchy to minimize the number of the threads

To get the optimal number of the threads (Figure 2 shows three threads) and avoid the potential of having as many (resource consuming) threads as Makers on a given chain, it is enough to reduce the number of "makers" on the "next level of hierarchy" via an intermediate "maker" level. Such additional layer does not require changing any user "maker" hence "transparent".

3.1.2. "Transparent" registrationions

It is noteworthy to realize that a "query"/"publish" paradigm is not sufficient to run the multi-threaded application effectively. Such model should be complemented with an API to "register" the module's output data sets, allowing for automatically suspending or resuming other modules waiting for the registered "datasets" to be produced. As illustration of this principle, let us study Figure 3 which shows a simple two "makers" chain communication in a single thread model. "Maker 1" produces a data "maker "2" needs to complete its job. Since they are executed in sequence, there are no synchronization issues and all provider/consumer relations are implicitly self-consistent.

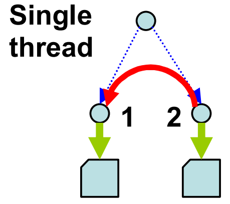


Figure 3. The "maker" commucation within single threaded chain

Figure 4 shows that the same "chain", if parallelized, may produce the wrong result since "maker 2" can query "maker 1" for its consuming data before "maker 1" had the time to provide it (generate it). Whenever "maker 1" finally publishes its result, "maker 2" has already provided the wrong result that is, its own provider's data set not based on nor built from the expected output and information from the first "maker".

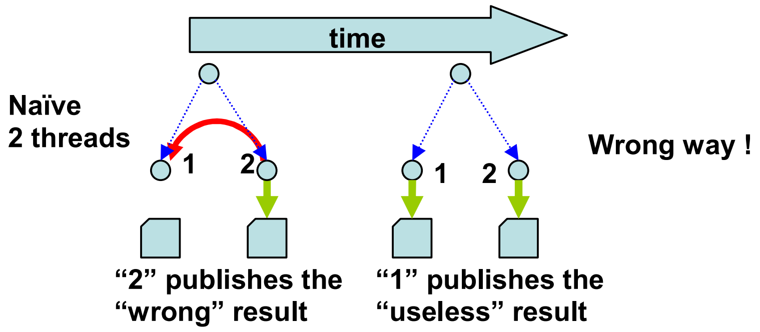

time
Naïve 2 threads
1
2
1
2
Wrong way !
"2" publishes the "wrong" result
"1" publishes the "useless" result

Figure 4. Use case to show the mutithreaded chain can produce the wrong result

Figure 5 illustrates that the introduction the additional facility and API to register the data set within the `Init()` method resolves the ambiguity. This is because the thread the "maker 2" belongs to, can be automatically suspended and then resumed as soon as the "maker 1" produced the "registered" data.

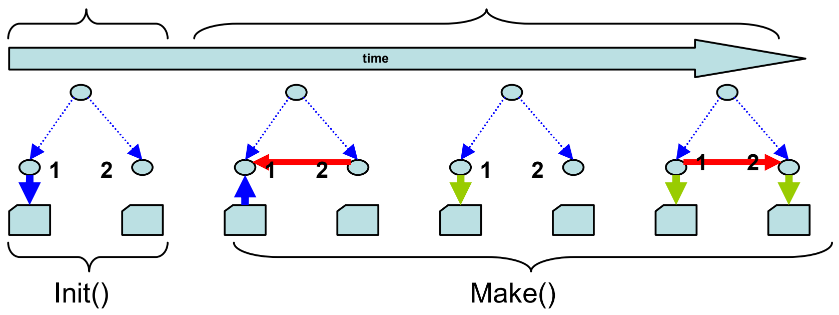


Figure 5. An additional "regsistration" step is introduced to resolve the ambiguity

4. Proof of principle implementation

The first version of the "parallel" chain concept was used in the context of the STAR "Online Event Display". Its multi-threaded version allows the STAR shift crew to investigate the detector response interactively without data taking interruption. It has been deployed during the RHIC running year 8 and proved the approach can be implemented with a reasonable amount of time investment [3] . The enhanced version of our beta-application example will be used during Run 9. It contains all essential components one can find within the STAR full-fledged production "chains" and essentially

- A top level "maker" - "StSteeringModule"
- Input/Output "maker" – "StDataReadModule" and a set of main "maker" producing data
- ‘StDetectorGeomModule" and a visualization module "StDisplayModule"

The thread relation between those modules is represented in Figure 6.

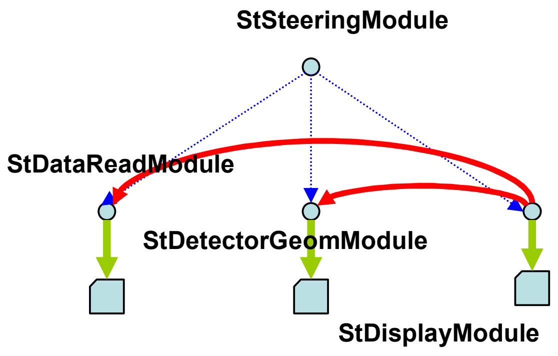


Figure 6. First working version of the parallel chain

5. Conclusion

The analysis showed that the existing design of STAR single-threaded framework could evolve transparently to meet the Multi-Core Era requirement with little effort and no global re-design. While is clear one should anticipate many problems arising from the onset of the multi-ore era and much additional effort (at algorithmic level) may be needed to fully harvest the power of many core architectures, our approach is a first step toward a "soft" evolution of an existing framework which has proven its worth and stability through years of usage. In this paper, we concentrated our effort to identify, implement and test a crucial design component targeted to achieve "transparent" parallelization, synchronization and registration of data in a provider / consumer design model. Furthermore, and since it relies on the ROOT base class TDataSet, our approach could be made general for ROOT users.

Our proof of principle implementation in an "EventDisplay" context allowed testing the main design solution aimed to prepare STAR to enhance its production and analysis software frameworks with the ability to take some advantage of modern hardware. More work will certainly be needed to bring together the existing reconstruction framework and algorithms and the full potential power of the multi-core hardware for the offline "event reconstruction" and raise the level of usability of interactive data-analysis.

6. Acknowledgements

This work was supported by the Office of Nuclear Physics and High Energy Physics within the U.S. Department of Energy Office of Science, and the U.S. National Science Foundation.

References

[1] V. Fine at el, *Steps Towards C++/OO Offline Software in STAR*, in proceedings of *CHEP'98 conference*, Chicago, 1998

[2] V. Fine at el, *OO Model of STAR detector for simulation, visualization and reconstruction*, in proceedings of *CHEP'2000 conference*, Padova, 2000

[3] V. Fine, at al, *The Object Model to Construct the Mixed Open Inventor/ ROOT 3D scenes*, in proceedings of *ACAT 2007 Workshop, Amsterdam,2007* PoS(ACAT)023