Parallel computing of SNiPER based on Intel TBB

J H Zou[1], T Lin[1], W D Li[1], X T Huang[2], T Li[2], Z Y Deng[1], G F Cao[1], Z Y You[3]

[1] Institute of High Energy Physics, Chinese Academy of Sciences, Beijing, China
[2] Shandong University, Jinan, China
[3] Sun Yat-sen University, Guangzhou, China

E-mail: `zoujh@ihep.ac.cn, lintao@ihep.ac.cn`

Abstract. SNiPER is a general purpose offline software framework for high energy physics experiments. During the development, we pay more attention to the requirements of neutrino and cosmic ray experiments. And now it has been adopted by many experiments. It is necessary for us to implement parallel computing to improve the data processing efficiency. Intel Threading Building Blocks (TBB), as a powerful high level library, emancipates us from trivial and complex details of raw threads. In this proceeding, we implemented an event level parallel computing wrapper for the original serial SNiPER based on Intel TBB. We have taken parallel computing into account at the beginning of SNiPER's design, it is possible to achieve multithreading in a non-invasive way. The SNiPER kernel module and Intel TBB are absolutely de-coupled. Features of multithreading are transparent to most users, except a few conventions such as global variables. This approach will significantly reduce the costs of migration from serial to parallel computing. However, it is more complicated for critical resources handling, such as disk I/O and memory management. There are also some attempts for these challenges.

1. Introduction

Offline software plays an important role in current high energy physics experiments. It helps physicists to achieve results accurately and efficiently. Under the fast development of particle detection technology and information technology, the volume of data collected in HEP becomes larger and larger, the requirement for offline data processing becomes more and more complicated and challenging. However, there are always some generic patterns in the offline software of this domain. Software framework can be independently implemented to present these patterns. It acts as the skeleton of a software system to provide a unified environment for programming and execution. Accordingly, users are able to concentrate on their specific tasks without concern for technical details. There are already many widely used frameworks, such as Gaudi [1], Art [2] and so on.

Now several Non-collider particle physics experiments are under construction in China, such as Jiangmen Underground Neutrino Observatory (JUNO) [3] and Large High Altitude Air Shower Observatory (LHAASO) [4]. All of them are comprised of very large scale detectors, which will produce huge amount of data in the near future. Meanwhile, there are some specific requirements for these experiments. Based on the experience of previous experiments, we implemented an offline software framework from scratch, which is named as Software for Non-collider Physics Experiments (SNiPER) [5]. Although the specific requirements of JUNO and

LHAASO are taken into account, SNiPER is implemented for general purpose that can be used by other experiments.

As a general purpose framework, SNiPER is customizable and extensible for different experiments, while its kernel module and main functionalities are independent of any experiments. For each specific experiment, physicists can implement their own data processing modules with the interfaces provided by SNiPER, as shown in Figure 1. These modules, such as simulation and reconstruction algorithms, can be dynamically loaded and configured during execution. Therefore we can perform various data processing jobs in a unified style.

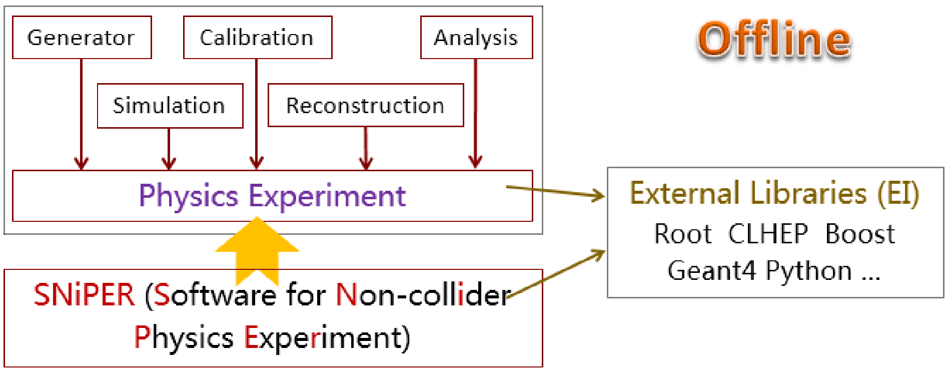

Figure 1. A typical offline software system based on SNiPER

At present, SNiPER has been officially adopted by JUNO, and successfully used in JUNO detector design studies [6, 7]. The practice proves the functionalities of SNiPER. It achieves our original goal. There is a lot of room for improvement, including parallel computing development.

2. Parallel computing schema of SNiPER

The data amounts of future experiments will be very huge, the processing of which will take a great number of computing resources. Then we encounter this urgent challenge, how to improve the execution efficiency of SNiPER. Due to technical limits, it is hard to increase CPU clock speed rapidly in recent years. Alternatively increasing cores number became the trend in the CPU industry. Accordingly, it is natural to apply parallel computing from the view of software developers. This became a hot spot in high energy physics field. Many software have supported parallel computing with varying degrees, including the framework GaudiHive [8].

In our long term plan, we will accomplish a mixture of multithreading and MPI programming for SNiPER. Multithreading is used on each multi-core node, meanwhile MPI is used for cross nodes parallelism. The two methods can be used separately or jointly. This paper focuses on our first step of multithreading programming.

There are a lot of existing modules developed by users. We want to find a way to minimize the costs of migration from serial to parallel computing. Our data consist of a large number of events. Every event is independent of each other, or correlated to nearby events only. And the processing procedure is similar for all events. We can use this characteristic to simplify the parallel computing schema of SNiPER. It is the underlying events parallelism from the view of framework. In this schema, there can be more than one parallel flows of execution, but each event is handled in a flow of execution uninterruptedly. As a result, the procedure of each event is almost the same as the serial version. Hence existing modules can be migrated easily. Moreover, this schema is also able to support the MPI implementation of SNiPER in the future.

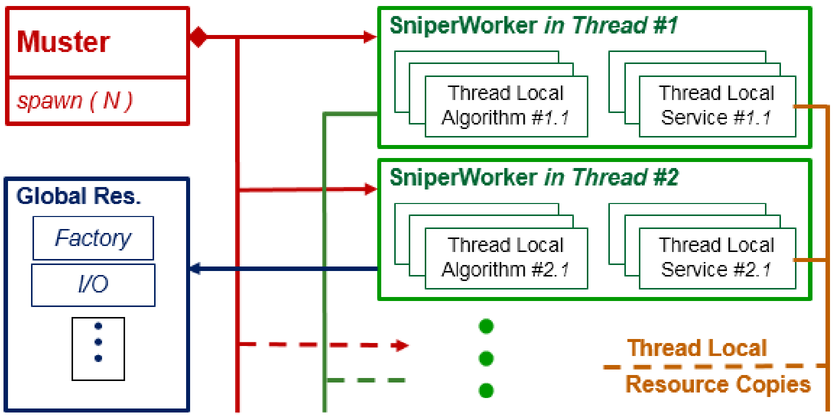
Muster
spawn (N)
SniperWorker in Thread #1
Thread Local Algorithm #1.1
Thread Local Service #1.1
SniperWorker in Thread #2
Thread Local Algorithm #2.1
Thread Local Service #2.1
Global Res.
Factory
I/O
Thread Local Resource Copies

Figure 2. Parallel computing schema of multi-threaded SNiPER

Figure 2 shows the schema of multi-threaded SNiPER. There is a Worker in each thread, which holds a local copy of a SNiPER Task component and its belongings (algorithms and services). All Workers are managed by the global Muster (Multiple SNiPER Task Scheduler). However, sharing resources, such as physical memory and I/O streams, have to be treated carefully.

3. Implementation of SNiPER Muster

Intel TBB [9] is a widely used C++ template library for task parallelism. It provides rich features for general purpose parallelism. Developers are able to think parallelism at the higher level avoiding dealing with the details of raw threads. It is easier for us to reach a better dynamic workload balancing with Intel TBB tasks and scheduler. According to the advantages, we chose to implement SNiPER Muster based on Intel TBB.

It's a coincidence that a class in SNiPER is also named Task (but start with an upper case T). The Task component performs like a lightweight application manager. Data processing procedures are configured and managed in the Task components. In a serial SNiPER application, it handles the event loop continuously. However, we can handle a single event by one invoking each time. There is another important feature in SNiPER, more than one independent Task instances can exist simultaneously.

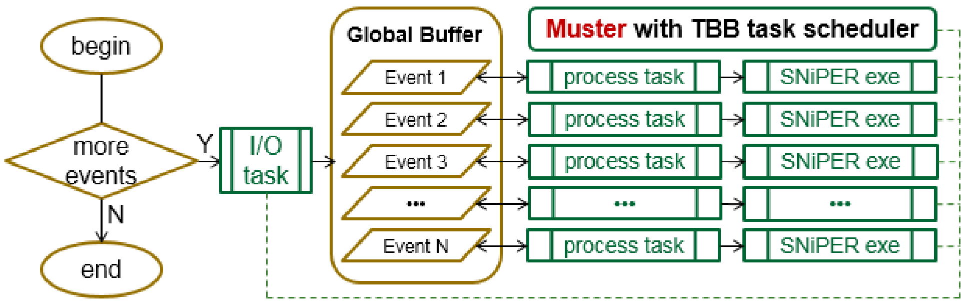

Figure 3. Task scheduling of the SNiPER Muster

In the multi-threaded SNiPER, the traditional event loop is broken up by the Muster. As shown in Figure 3, different events are dispatched to different Workers. The Muster is in charge of the creation and scheduling of Workers, which is mainly based on Intel TBB's scheduler. We can simply map a SNiPER Task component to an Intel TBB task in each Worker. Then Workers can be executed in threads concurrently.

When a Worker is invoked, it grabs and locks an event from the memory until it completes. During the execution of a Worker, the event is handled by a thread local SNiPER Task component. This feature ensures that a Worker looks the same as a serial SNiPER application. Existing algorithms and services can be migrated almost transparently. The only restrictions are global variables and sharing resources. Once a module is migrated, it can be used in both serial and multi-threaded context.

There is only serial computing in the original SNiPER. But we have taken into account the requirements of SNiPER in a Worker at the beginning. So it's possible for us to implement Muster as a non-invasive wrapper of the SNiPER kernel module. By this way, users are able to switch between serial and parallel computing seamlessly.

4. I/O and memory management

In Figure 3, there is also an underlying I/O task Worker and a global buffer. The I/O task Worker represents the integration of I/O streams and Intel TBB. It is invoked according to the utilization ratio of the global buffer. The global buffer represents the memory management for all events. Besides that, there is a buffer locally managed by the SNiPER Task component in each Worker. We name it as local buffer to distinguish from the global buffer.

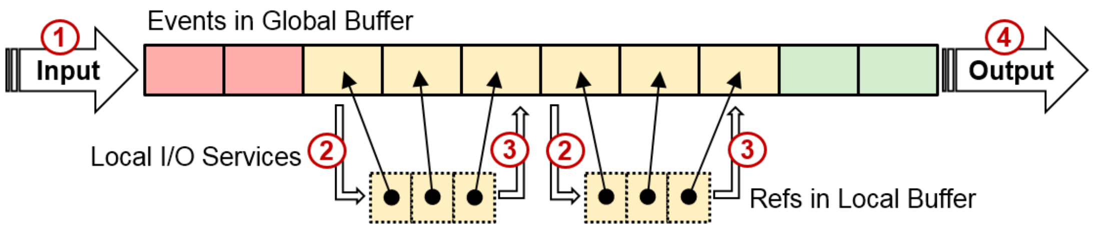

Figure 4. The data flow of multi-threaded SNiPER

A status code is assigned to each event. The event status can be READY, OCCUPIED or COMPLETE. As shown in Figure 4, the event data flow and status transition are expressed in 4 steps:

(1) When the utilization ratio of global buffer is low, the I/O task Worker is invoked for reading. New events are read from input stream, filled into the front of global buffer, and then marked as READY.

(2) A Worker requests a READY event from global buffer. A reference to the event is filled in the local buffer. Then the event is marked as OCCUPIED. In fact, a Worker can request a number of events each time for optimization.

(3) When an event is processed by the Worker, the reference is removed from the local buffer. And the event is marked as COMPLETE.

(4) When the utilization ratio of global buffer is high, the I/O task Worker is invoked for writing. COMPLETE events at the end of global buffer are sent to output stream and then removed.

From the previous description, we can notice that step 2 and 3 can be executed concurrently by Workers. Meanwhile, there are some advantages for us to simplify the programming:

- The I/O streams are associated to the global buffer, but not Workers. So that the I/O streams are invoked serially. We can avoid the complexity of stream synchronization.
- The SNiPER Task component in each Worker is executed serially, except that it has a customized I/O service. Instead of operating streams, it grubs event by reference from the global buffer. This operation is fast, and the lock can be very thin.
- The right order of events is kept in the global buffer. There is no need for sorting when we write events out.

The JUNO I/O service is still using ROOT 5. Since we applied a mechanism of data lazy loading, we have to add very fat locks in the I/O stream module. Otherwise there are potential conflicts when Workers access events data simultaneously. In order to make this prototype execute properly for JUNO, there are still a lot of things to do. ROOT 6 has made great progress for multithreading [10]. Now we are working on the migration from ROOT 5 to ROOT 6, which should be helpful to solve that problem.

5. Performance measurement

Due to the event I/O issue, this performance measurement uses a dummy algorithm without I/O operations. Following are the details of the testing node:

- Intel(R) Core(TM) i7-6700HQ CPU, 4 cores, Hyper-Threading is off
- 16 GiB Memory
- Ubuntu 16.04 with GCC 5.4
- Intel(R) TBB 4.4

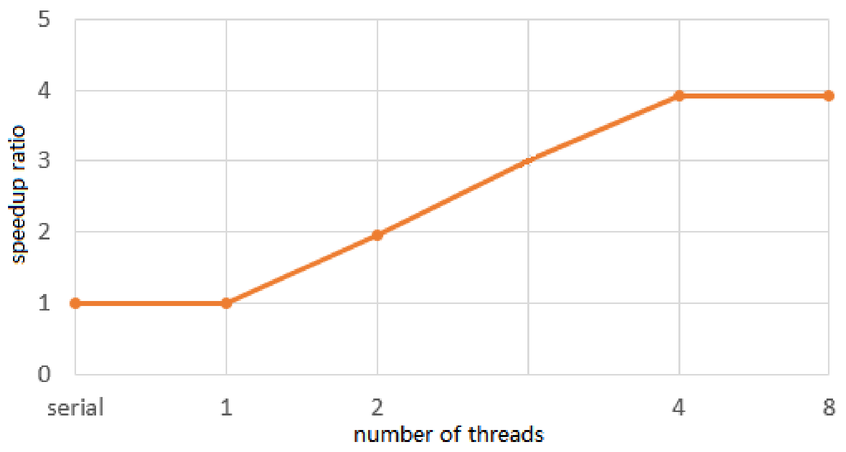

Figure 5. Speedup ratio vs. number of threads

20,000 events are executed three times for each case. The average time is used to calculate the speedup ratio. The result is shown in Figure 5. Since there is no effect from I/O operation, the speedup is nearly linear until all of the 4 cores are busy, just as we expected.

6. Conclusions

We have developed the general purpose framework, SNiPER. It has been adopted by JUNO, LHAASO and several other experiments. Its performance in serial data processing is well demonstrated in practice. Its parallel computing functionalities are under development and being optimized. In this paper, we present the result of multithreading programming based on Intel TBB. A non-invasive wrapper, Muster, is implemented as a SNiPER Task scheduler. We can handle events concurrently with Muster and Workers in threads. Muster is almost transparent to ordinary users. This will minimize the costs of migration from serial to parallel computing. We also present a prototype for I/O and memory management. However, critical resource handling is still a challenge. There is still a lot of room for improvement.

Acknowledgements
This work is supported by Joint Large-Scale Scientific Facility Funds of the NSFC and CAS (Grant No. U1532258), the Youth Innovation Promotion Association of Chinese Academy of Sciences (Grant No. 2017021), the National Natural Science Foundation of China (Grant No. 11605221) and the Strategic Priority Research Program of the Chinese Academy of Sciences (Grant No. XDA10010900).

References

[1] Barrand G *et al.* 2001 GAUDI - A software architecture and framework for building HEP data processing applications *Comput. Phys. Commun.* **140** 45
[2] C Green *et al.* 2012 *J. Phys.: Conf. Ser.* **396** 022020
[3] An F *et al.* (JUNO) 2016 *J. Phys.* **G43** 030401
[4] Cao Zhen *et al.* 2010 A future project at tibet: the large high altitude air shower observatory (LHAASO) *Chinese Phys. C* **34** 249
[5] Zou J H, Huang X T, Li W D, Lin T, Li T, Zhang K, Deng Z Y and Cao G F 2015 *J. Phys.: Conf. Ser.* **664** 072053. See also "SNiPER" [software], version 1.0, Available from `https://github.com/SNiPER-Framework/sniper/releases/tag/v1.0` [accessed 2018-03-20]
[6] Lin T, Zou J, Li W, Deng Z, Fang X, Cao G, Huang X and You Z (JUNO) 2017 *22nd International Conference on Computing in High Energy and Nuclear Physics (CHEP 2016) San Francisco, CA, October 14-16, 2016* arXiv:1702.05275 [physics.ins-det]
[7] Huang X T, Li T, Zou J H, Lin T, Li W D, Deng Z Y and Cao G F 2016 *PoS (ICHEP2016)* 1051 URL `https://pos.sissa.it/282/1051`
[8] M. Clemencic *et al.* 2014 *J. Phys.: Conf. Ser.* **513** 022013
[9] James Reinders. *Intel Threading Building Blocks.* O'Reilly Media, Sebastopol, CA, 2007.
[10] B Bellenot *et al.* 2015 *J. Phys.: Conf. Ser.* **664** 062006