20th International Conference on Computing in High Energy and Nuclear Physics (CHEP2013) IOP Publishing
Journal of Physics: Conference Series **513** (2014) 022013 doi:10.1088/1742-6596/513/2/022013

Introducing concurrency in the Gaudi data processing framework

Marco Clemencic, Benedikt Hegner, Pere Mato, Danilo Piparo
CERN CH-1211, Switzerland

E-mail: marco.clemencic@cern.ch, benedikt.hegner@cern.ch, pere.mato@cern.ch, danilo.piparo@cern.ch

Abstract. In the past, the increasing demands for HEP processing resources could be fulfilled by the ever increasing clock-frequencies and by distributing the work to more and more physical machines. Limitations in power consumption of both CPUs and entire data centres are bringing an end to this era of easy scalability. To get the most CPU performance per watt, future hardware will be characterised by less and less memory per processor, as well as thinner, more specialized and more numerous cores per die, and rather heterogeneous resources. To fully exploit the potential of the many cores, HEP data processing frameworks need to allow for parallel execution of reconstruction or simulation algorithms on several events simultaneously. We describe our experience in introducing concurrency related capabilities into Gaudi, a generic data processing software framework, which is currently being used by several HEP experiments, including the ATLAS and LHCb experiments at the LHC. After a description of the concurrent framework and the most relevant design choices driving its development, we describe the behaviour of the framework in a more realistic environment, using a subset of the real LHCb reconstruction workflow, and present our strategy and the used tools to validate the physics outcome of the parallel framework against the results of the present, purely sequential LHCb software. We then summarize the measurement of the code performance of the multithreaded application in terms of memory and CPU usage.

1. Introduction

In the past, the increasing demands for HEP processing resources could be fulfilled by the ever increasing clock-frequencies and by distributing the work to more and more physical machines. Limitations in power consumption of both CPUs and entire data centers are bringing an end to this era of easy scalability. To get the most CPU performance per watt, future hardware will be characterised by less and less memory per processor, as well as thinner, more specialized and more numerous cores per die, and rather heterogeneous resources. To fully exploit the potential of the many cores, HEP data processing frameworks need to allow for parallelization at three different levels – parallel execution of algorithms within a single event, parallel processing of multiple events at the same time, and parallelization within the algorithms themselves.

We are trying to address this by extending the successful Gaudi [1] event-processing framework by concurrency. This paper describes its application and performance for a real use-case taken from the LHCb reconstruction, which offers us the possibility to validate the results against a working sequential program. The design of concurrent Gaudi and its rationale have been discussed in [2], but for the understanding of this paper we will mention the most important concurrency-enabling components again. Those are a multi-event store (*Whiteboard*),

the resource protection functionality and the scheduling infrastructure. The Gaudi framework is based on the idea of executing *Algorithms* that act on event data items residing in data store. Algorithms are chained to provide a full data processing application. Evolving Gaudi for concurrency implies to be able to execute these algorithms on different event data at the same time

1.1. Whiteboard

The *Whiteboard* is a multi-event store, which can contain multiple Event Stores, implements the original event store's interface in a thread safe manner. In addition it provides some additional functionalities, for example the bookkeeping of the newly added data objects, which is useful for algorithm scheduling, as mentioned further below.

1.2. Algorithm Pool

The *AlgorithmPool* is a new component providing multiple, interchangeable instances of the same, potentially thread-unsafe, algorithms. The need for this cloning will be very apparent in Section 4 discussing the scaling of the MiniBrunel prototype.

1.3. Forward Scheduler

While multiple scheduling strategies for concurrent execution of these algorithms exist, we follow the idea of forward scheduling. As soon as all its preconditions for execution of a particular algorithm are fulfilled, the algorithm is being executed. These are the actual need for scheduling it (according to *control flow*), the availability of its required data inputs (according to *data flow*), and the availability of the required resources, such as non-reentrant I/O, access to databases, as well as the availability of a free algorithm instance to be executed. A distinct feature of this approach is that the exact set of prerequisites for a certain task to be executed is fully known and thus potential conflicts can be avoided upfront. In particular this addresses the concern of dead-locks[1]. In the present implementation, the handling of algorithm scheduling is reflected in the following very simple state machine, see Figure 1. The transition from *Initial* to *ControlReady* is accomplished by evaluating the control conditions using the return codes of previously run algorithms. The transition from *ControlReady* to *DataReady* makes direct use of the Whiteboard and checks it for the availability of required input data. Finally, the transition from *DataReady* to *Scheduled* happens by claiming the required resources via the AlgorithmPool and then pushing a *tbb::task* into the runtime environment of the used task-management system TBB [3]. After the *tbb::task* finishes, the scheduling infrastructure handles all claimed resources back to the *AlgorithmPool* and sets the algorithm's state to either *Event Accepted* or *Event Rejected*, depending on the filter decision of the algorithm. The event is considered complete when all algorithms that needed to be executed are in either of these two states. On starting a new event the algorithms are put back into the *Initital* state.

2. The MiniBrunel Prototype

To demonstrate the readiness of the described design for real-use cases, we adopted a pragmatic approach. We selected a subset of the LHCb reconstruction application, in the following called *MiniBrunel*. This MiniBrunel subset covers the raw decoding and local reconstruction of the vertex locator detector. In total, the *MiniBrunel* slice of the LHCb reconstruction consists of 14 algorithms and 24 tools used by them. The data dependencies between the algorithms are shown in Figure 2. The MiniBrunel prototype does not only serve as a test case for the concurrent

[1] In contrast, as soon as a on-demand component is being introduced, like in a backwards-scheduling approach, the risk for dead-locks arises. The required resources of tasks are not known upfront and two tasks may during execution run into resource conflicts. In the best case, this leads to one task busy-waiting for the other. In the

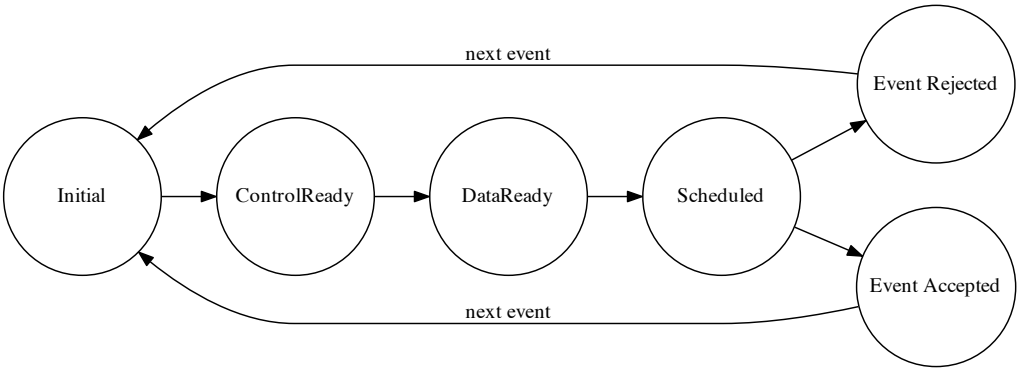

Figure 1. The algorithm state machine using in the forward scheduling implementation.

Figure 2. The algorithms and data objects involved in the MiniBrunel workflow.

Gaudi design, and its scaling behavior. It allows us to estimate the migration costs for the entire application as well. The aim of the chosen design was to preserve the experiments' investment in algorithmic code and to change of it as little as possible.

2.1. Required code changes

In addition to the concurrent framework components of Gaudi, a few parts of Brunel needed to be adjusted as well. First and foremost, the infrastructure to announce the data dependencies had to be put in place for all algorithms listed in Figure 2. This is straight forward, but

worst case to a full dead-lock of the entire application.

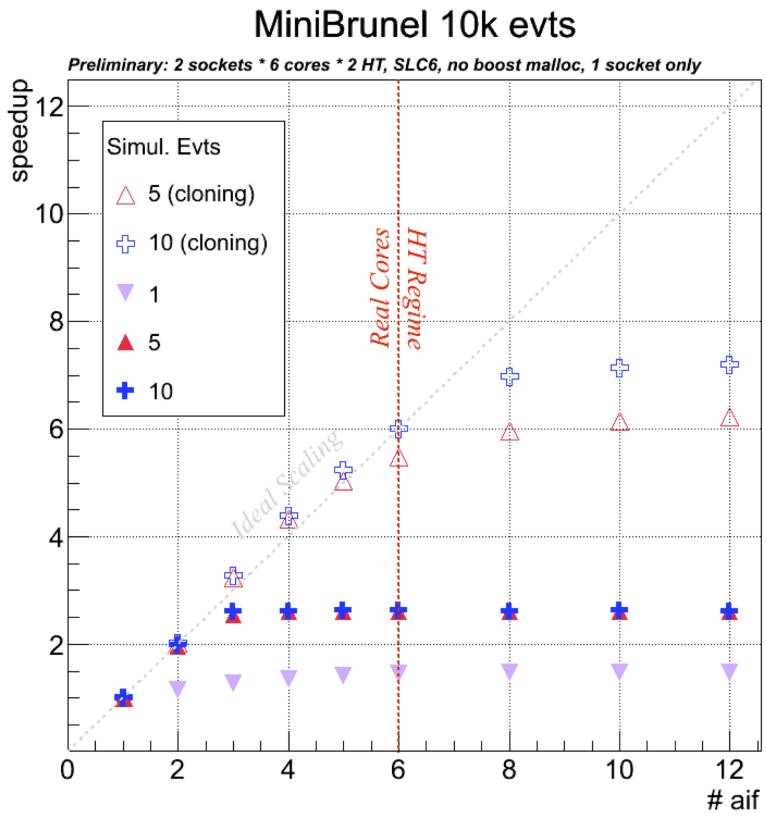

Figure 3. Speedup of MiniBrunel for a different number of algorithms allowed to run in parallel. Shown are the behavior for a different number of simultaneous events and with and without cloning of the three most time-consuming algorithms.

nevertheless a task to be carried out. More intrusive changes were connected to the common use of tools across algorithms, which requires sufficient thread-safety of these components, and the need for supporting multiple events at the same time. In particular, the caches of the raw data conversion infrastructure were enabled to deal with multiple events at a time. Details of the chosen approaches can be found in these proceedings [4].

3. Cross-Validation of Results

The reconstruction results of the concurrent MiniBrunel were thoroughly compared with the results of the original code base, using different running scenarios. We used the same set of quantities as used for LHCb release cross-validation. The results were found to be identical.

4. Scaling behavior

Figure 3 shows the measured speedup of MiniBrunel depending on the number of algorithms that are allowed to run in parallel for various scenarios and number of events processed simultaneously. The benchmark machine had 6 physical cores as indicated by the vertical red line. Beyond the line, the application takes advantage of the hyper-threading capabilities of the processor as well. The measured speedup when processing only one event at a time is in the order of 30 % as could be calculated a-priory by analyzing the critical path in the algorithms' dependencies. For this reason, a real gain of parallelization can only be achieved if one moves to processing multiple events at the same time. Again, the figure shows the limited scaling – only a speedup

Table 1. The memory consumption of MiniBrunel using the original components and the upgraded version for a different number of events running concurrently.

Serial version	concurrent 1 event	concurrent 2 events	concurrent 10 events
478 MB	480 MB	485 MB	514 MB

of approximately 2.5 is achieved. This is due to the fact that the slower running algorithms are a resource bottleneck as they, for thread safety, can only be used for one event at a time. The solution to this problem is the *cloning* of algorithms and providing interchangeable, independent instances which can be used concurrently. Already by cloning the three most time consuming algorithms a perfect scaling on the physical cores of the test machine can be achieved. The hyper-threaded cores show the typical behavior of weaker scaling. However, it has to be stated that there are limitations to the cloning approach. For example, cloning algorithms that require thread-unsafe resources has no benefit, if the resources aren't made multi-thread aware at at the same time. This can either happen by addressing thread-safety of these resources or making them subject of cloning themselves. As previously stated, the number of algorithms that need to be cloned for performance reasons is only a small fraction of the total number of algorithms. It seems feasible to solve such issues on a case by case basis.

4.1. Memory consumption

One of the main reasons for parallelizing applications internally rather than executing multiple independent applications, is the overall memory consumption. The developed prototype gives a first hint on whether the problem of memory consumption is indeed properly addressed. Table 1 shows the memory consumption for the original sequential application and a different number of events being processed concurrently. For the MiniBrunel case, the additional memory required for one event is in the order of 3.5 MB, which is less than 1% of the overall memory consumption.

5. Conclusions

The Gaudi event processing framework has been extended to support concurrency at multiple levels. The concurrent framework extensions were used to migrate a slice of the LHCb reconstruction, namely the local reconstruction of the vertex locator, to a fully parallel-aware setup. The results of the parallel execution were successfully validated against the original sequential application. It is important to note that the thread-awareness was introduced without prior knowledge of the actual components and their implementation details. Furthermore, the changes were not connected to the actual physics functionality of the algorithms concerned. This is a strong indication that the migration of similar applications can be mostly carried out without the help of the individual algorithm developers. Only this allows to think about a migration effort of existing HEP applications towards full parallelization. That such an effort is indeed worthwhile can be seen in the scaling behavior of the MiniBrunel prototype. While keeping the memory consumption very close to the sequential application, the application speed scales linearly on the machine used for the multicore-performance measurement.

Acknowledgments

The authors would like to thank the LHCb core software group for their support and assistance.

References

[1] G. Barrand, I. Belyaev, P. Binko, M. Cattaneo, R. Chytracek, et al. GAUDI - A software architecture and framework for building HEP data processing applications. *Comput.Phys.Commun.*, 140:45–55, 2001.

[2] B. Hegner, P. Mato, and D. Piparo. Evolving LHC data processing frameworks for efficient exploitation of new CPU architectures. In *Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC), 2012 IEEE*, pages 2003–2007, 2012.

[3] J. Reinders. *Intel Threading Building Blocks*. O'Reilly Media, 2007.

[4] M. Clemencic, B. Hegner, P. Mato, and D. Piparo. Preparing HEP software for Concurrency. In *these proceedings*.