Parallelization of particle transport using Intel® TBB

J Apostolakis[1], S Belogurov[2], R Brun[1], F Carminati[1], A Gheata[1], E Ovcharenko[2] and S Wenzel[1]

[1] European Organization for Nuclear Research (CERN), Geneva, Switzerland

[2] Institute for Theoretical and Experimental Physics (ITEP), Moscow, Russia

E-mail: E.Ovcharenko@gsi.de

Abstract. One of the current challenges in HEP computing is the development of particle propagation algorithms capable of efficiently use all performance aspects of modern computing devices. The Geant-Vector project at CERN has recently introduced an approach in this direction. This paper describes the implementation of a similar workflow using the Intel® Threading Building Blocks (Intel® TBB) library. This approach is intended to overcome the potential bottleneck of having a single dispatcher on many-core architectures and to result in better scalability compared to the initial pthreads-based version.

1. Introduction

Numerous factors like hardware evolution, increasing requirements on the speed of simulations and the race for cheaper computations stimulate the development of efficient particle transport codes that make use of all performance dimensions on contemporary and future computing architectures. As relevant examples we would like to mention recent developments of GEANT4 where event level parallelism is implemented [1] and the development of high energy electromagnetic particle transportation package using GPGPU [2].

One particular effort towards finer grain parallelism in particle transport was initiated by the Geant-Vector prototype project at CERN, and a first prototype exploiting thread parallelism on a track level was presented at CHEP2012 [3, 4]. The current status of the vectorized particle transport is discussed in [5].

The basic work unit for track level parallelization is a *basket* – a group of tracks coming from one or more events. A worker thread is processing one basket at a time in a concurrent environment where several baskets are processed independently. There can be different criteria of grouping tracks into baskets to optimize code locality and vectorization. For the moment we are using only a geometry criterion: a basket will only collect tracks from a single logical volume. The result of transporting the particles from one basket is a collection of particles that have either reached the boundaries of the current volume, interacted or fallen below the transportation energy threshold. The output track collections are being processed by a single data dispatching thread that regroups all tracks into new baskets, sending them to the main work queue for further propagation. The main goal of dispatching policy is to provide for transport baskets with reasonable track content to be supplied to vectorized geometry navigator and physics.

The work presented here is dedicated to exploring a task-based schema using Intel® TBB as a replacement for the current pthreads-based implementation. One of the reasons for choosing a task-based scheduling library such as Intel® TBB is the possibility to scale up the number of dispatchers

(which now become tasks) with the available resources, as needed by the data workflow. This feature is intended to overcome the potential bottleneck of having a single dispatcher on many-core architectures and should result in better scalability comparing to classic thread-based programming. Also Intel® TBB's principle of scheduling tasks, which is designed to maximize cache reusage, fits well with the basket-oriented propagation where consecutive tasks may have same data or instructions to process. There are several other positive experiences in using Intel® TBB in HEP such as the concurrent version of the Gaudi framework [6], which exploits different levels of parallelism such as event level, algorithm level and intra-algorithm level.

2. Prototype algorithm

The data-flow algorithm of the prototype is the same for the pthreads-based and Intel® TBB implementations. The goal is to propagate the tracks of several events throughout the detector. The prototype takes as input a given *number of events to be transported*. At any moment there are only a fixed *number of event slots* in the memory in order to use a limited amount of memory. The *average number of tracks* for each event is also an input parameter. Initially all slots are filled with events – all generated tracks are injected at once into a single track collection. The dispatcher then takes this collection and its tracks are "basketized" according to a "transportability" threshold for the basket population. The filled baskets are injected into the work queue and the propagation starts.

The propagation procedure takes a basket and transports its tracks within one volume from the current position to the volume boundaries or the place of occurrence of a physics interaction using a very simple toy physics model. The result of propagation is a *collection* – a group of tracks on the boundaries ready to leave the current volume. The goal of the dispatcher is to reschedule the tracks from one or several collections into the corresponding baskets. When the basket gets filled to its transportability threshold it is automatically sent to the propagation. The dispatcher also controls the work queue content, taking several actions once the workload drops below a low watermark.

3. Implementation based on Intel® TBB tasks

In the pthreads-based implementation a fixed number of threads is used – one dispatcher and one or more workers. Figure 1 shows the schematic model of the original prototype. The threads are initialized and started explicitly only once at the beginning, reused during the transport and explicitly joined at the end.
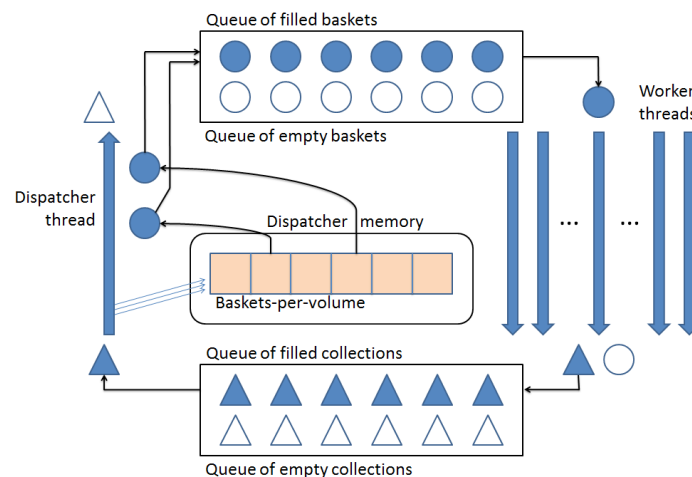

Figure 1. Schematic model of the thread-based prototype.

The TBB-based implementation presented here uses tasks and the Intel® TBB task scheduler. Figure 2 shows the schematic model of the developed prototype. In this schema there is no need to handle threads manually because the Intel® TBB task scheduler takes care of the mapping of tasks to threads internally.
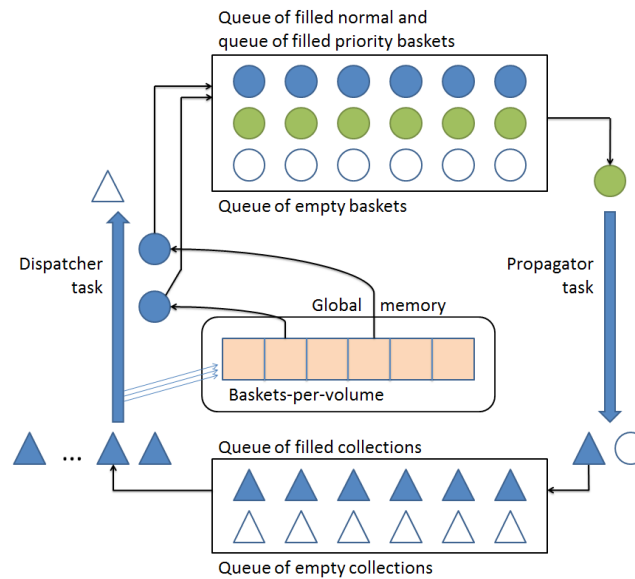
Figure 2. Schematic model of the task-based prototype.

There are two types of tasks – "propagation" and "dispatching" tasks. A propagation task propagates all tracks in one basket through the geometry of a single logical volume. It pops a non-empty basket from the work queue in the beginning and pushes an empty basket in the end. All surviving tracks are the output of this single volume transport stage and they end up in a single track collection. This filled collection is pushed into an output queue. A dispatcher task is started whenever the total number of "output" tracks exceeds a given *threshold*. It may happen however that a propagation task has produced no collection. In this case no child task will be created. Figure 3 shows possible branching of tasks.
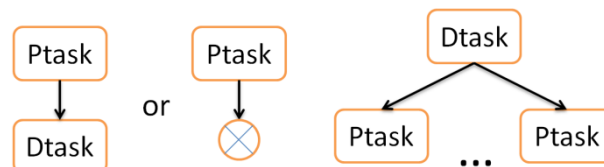

Figure 3. Task dependencies. Ptask – propagation task, Dtask – dispatching task.

A dispatching task now pops one or several collections from the output queue and distributes all the tracks into new baskets, pushing the transportable ones into the work queue and spawning propagation tasks for each pushed basket. The number of collections to pop is passed to the task as a parameter. A basket is considered to be transportable and ready for propagation if it has a given *number of tracks*. The value of 20 tracks per basket is used in the tests presented below. Having a fixed basket size is a very preliminary approach. Further developments will have more flexible policy that will result in a dynamic basket size.

During transport the baskets in the work queue get consumed while the particles are propagated in the detector. When the size of the work queue gets to the *minimum threshold* a given *number of older events* are being prioritized in order to free memory slots. A prioritized event is an event that has to be finished as soon as possible.

In the pthreads-based code, the prioritization of events is implemented using a double-ended work queue. Prioritized baskets are extracted according to the policy "last in first out", while normal ones according to the standard FIFO policy, making the prioritizing process transparent to the workers. In the present task-based implementation a propagation task has a priority flag as a parameter. It pops either from a priority work queue or from a normal work queue depending on the flag value. This is done for several reasons. The first is that tbb::concurrent_bounded_queue that is used to keep the

baskets cannot be double-ended. The second reason is that the present implementation allows the task to know whether the popped basket contains tracks of prioritized events or not. In the future this may allow using Intel® TBB task priority techniques.

In Intel® TBB there are different techniques for allocating, spawning and waiting for tasks. The prototype spawns dynamically "dispatching" and "propagation" tasks according to the basket flow and processing needs. The "scheduler bypass" and "continuation passing" templates featuring tbb::empty_task's as successors are used. "Scheduling bypass" technique allows to skip the Intel® TBB task scheduler and send a task directly to execution on the same thread. Future policies may use it to run the propagation task of the next basket in the same volume on the same thread.

Other cases of using Intel® TBB features not related to tasks are listed in table 1.

Table 1. Application of certain Intel® TBB features.

Manually-written in original prototype	Intel® TBB feature in present prototype
Concurrent queues featuring locks	tbb::concurrent_bounded_queue
Per-thread data containers	tbb::enumerable_thread_specific
Operations in critical sections	tbb::atomic

When an event in some slot is fully transported, a new event is injected into that slot. The number of threads to be used is limited by an explicit call to tbb::task_scheduler_init with a specified *number of threads* as an argument.

Moving to a task-based schema results in some additional free "policy" parameters that need to be further optimized for better performance. For example, current policy requires a *threshold* for the number of tracks waiting to start a dispatching task. The higher the threshold the less dispatching tasks will be spawned. At the same time each task will pop more collections.

4. Preliminary results

The dispatcher thread in the pthreads implementation is mostly idle because of a comparatively low number of workers. The advantage of using Intel® TBB can be seen in the situation when the dispatcher is overloaded which is possible when the number of workers is high. In this case a dynamic number of dispatchers is needed. One possible way to test with a large number of threads that is currently being investigated is to run the prototype on the co-processor Intel® Xeon Phi.

The results obtained with the present prototype correspond to expectations. From the diagrams showing the number of active tasks in time it follows that there are mostly propagation tasks active and rarely one dispatching task. Also sometimes it happens that there is more than one dispatcher at the same time. This happens when the amount of work in the propagation task is small compared to the amount of work in the dispatching task.

We observe a large similarity between the performance characteristics of both prototypes. The time spent on propagation is also approximately the same. Due to the fact that the dispatcher task does not process strictly one collection, which is the case for the pthreads-based code, the number of dispatching iterations is about ten times less.

Currently we see limited speedup both in pthreads and Intel® TBB models. Diagrams of speedup are showed on the figure 4. A detailed "locks and waits" analysis will be performed to understand communication bottlenecks.

5. Plans
- A new implementation of the pthreads-based vectorized particle transport is to be released soon [7]. It is planned to reproduce functionality of this version of the prototype with vectorized navigator and vectorized physics in the Intel® TBB task-based approach;
- Investigate the origin of the limited scalability;

- Study the influence of cache misses on the propagation performance;
- Test the prototypes on more cores, particularly on Intel® MIC, taking into account the architecture features;
- Apply and benchmark advanced features of Intel® TBB: priority of tasks, affinity of tasks;
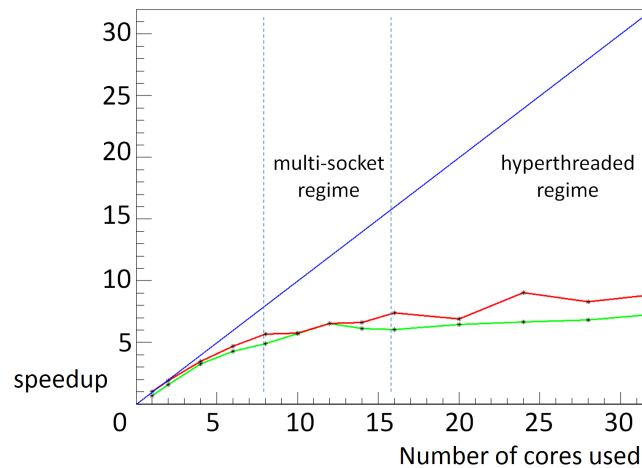- Test new dispatching and scheduling policies.

Figure 4. Scalability of original pthreads model (red) and of the Intel® TBB model (green).

6. Conclusion

A task-based approach was applied to the Geant-Vector particle propagation prototype using the Intel® TBB template library. The overall results of this prototype are close to the original one based on pthreads. The number of dispatching iterations is reduced by the factor of ten as expected. The next version of the TBB-based prototype should include the new vectorized navigator and address the problem of relatively low scalability we have observed in the current code.

Acknowledgments

7. References
[1]   Cosmo G, et al 2013 Geant4 – towards major release 10, these proceedings
[2]   Canal P, Elvira V D, Jun S Y, Kowalkowski J and Paterno M 2013 High Energy Electromagnetic Particle Transportation on the GPU, these proceedings
[3]   Apostolakis J, Brun R, Carminati F and Gheata A 2012 Rethinking particle transport in the many-core era towards GEANT, *J. Phys.: Conf. Ser.* 396 (2012) 022014
[4]   Apostolakis J, Brun R, Carminati F, Gheata A and Wenzel S 2013 A concurrent vector-based steering framework for particle transport, *15th International Workshop on Advanced Computing and Analysis Techniques in Physics (ACAT)*, In course of publication.
[5]   Apostolakis J, Brun R, Carminati F, Gheata A and Wenzel S 2013 The path toward HEP high performance computing, these proceedings
[6]   Hegner B, Mato Vila P and Piparo D 2013 Introducing Concurrency in the Gaudi data processing framework, these proceedings
[7]   Apostolakis J, Brun R, Carminati F, Gheata A and Wenzel S 2013 Vectorizing the detector geometry to optimize particle transport, these proceedings