Parallelization of particle transport using Intel® TBB

R. Brun[1], F. Carminati[1], A.Gheata[1], S. Wenzel[1], J. Apostolakis[1], E. Ovcharenko[2], S. Belogurov[2]
1 – European Organization for Nuclear Research (CERN), Geneva, Switzerland
2 – Institute for Theoretical and Experimental Physics (ITEP), Moscow, Russia

Introduction

Numerous factors like hardware evolution, rising requirements to the speed of simulations and a 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. One particular effort towards this goal 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. [1]
The work presented here is dedicated to moving the existing prototype to the task-based schema by using Intel® Threading Building Blocks (Intel® TBB). One of the reasons for choosing Intel® TBB is the ability to provide as many dispatchers as needed by the data workflow which follows naturally in the task-based implementation. This feature is intended to overcome the potential bottleneck on many-core architectures and result in better scalability comparing to classic thread-based programming. Also Intel® TBB principle of scheduling tasks which is designed to reuse cache fits well to the basket-based propagation where locality of data and instructions is needed.

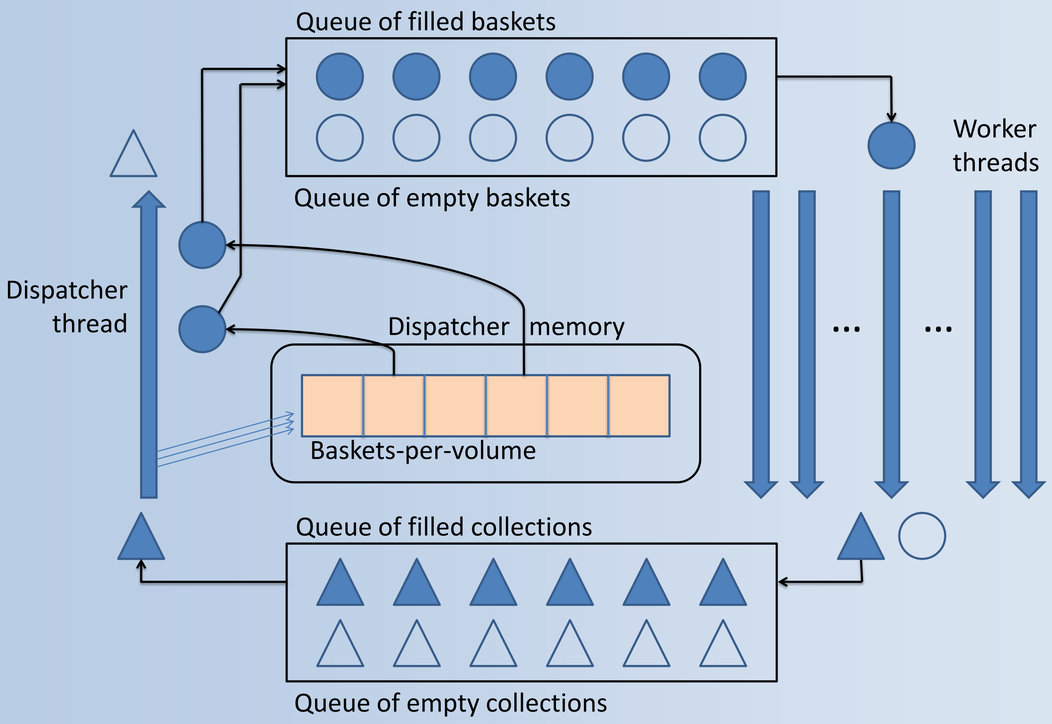
Primary problem

The basic entity for parallelization is a *basket* – a group of tracks coming from different events such that several baskets are propagated independently. One thread is working at one basket at a time. There can be different criteria of grouping tracks into basket. So far only a geometry criterion is implemented – tracks in one basket are in one logical volume.

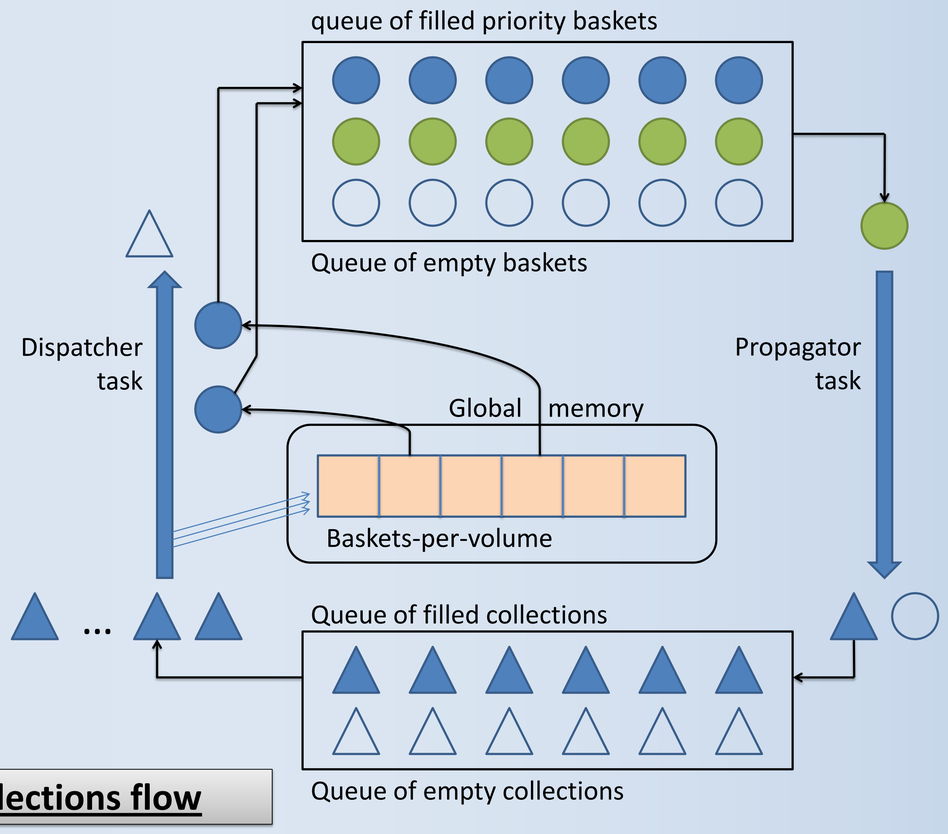
There are two main levels of parallelism the prototype aims for:
1) Thread parallelism. Basket management and mapping baskets to threads is to be handled by the scheduler.
2) Vectorization. Tracks in one basket are packed into a vector which is transported using vectorized navigator and vectorized physics.

Baskets have to be reasonably populated to exploit vectorization. The value of 20 tracks per basket is used in the tests presented below. This fixed size of a basket is chosen very preliminary. Further developments may have more flexible policy which will result in a dynamic size of a basket from some range. Another term used in this work is a *track collection* or simply *collection* – a group of particles which reached the boundaries of the current volume during propagation. A collection is a result of transporting tracks of one basket.

Original model based on classic thread-based programming



New model based on task-based programming



Application of Intel® TBB

1) Manually-written concurrent queues featuring locks are implemented using tbb::concurrent_bounded_queue
2) Manually-written per-thread data containers are implemented using tbb::enumerable_thread_specific
3) No direct thread handling – tasks are created and spawned to tbb::task_scheduler which cares about mapping them to threads
4) Some operations using locks are reimplemented using tbb::atomic

Major changes in the baskets and collections flow

- Prioritization of events is implemented using double-ended feeder queue. Baskets that correspond to priority events are pushed to the front which allows worker threads pop baskets from the front without knowing whether the basket is prioritized or not.
- Only one thread works with the “global” container of tracks so it is safe without locks.

- A propagation task has a priority flag as a parameter. It pops either from a priority feeder queue or from a normal feeder queue depending on the flag value. A dispatcher task has the number of collections to pop as a parameter.
- To allow several dispatcher tasks to work with the global container of tracks, classic locks are used to put some operations into critical sections.

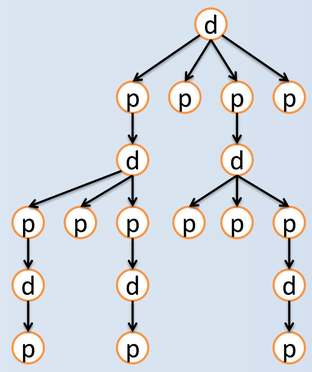
Plans:

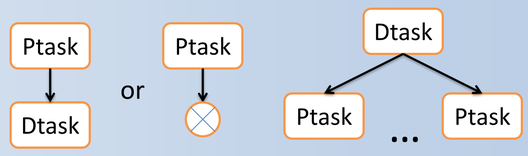
1) Follow the development of the other components and make corresponding more realistic tests;
2) Study the influence of cache misses to the propagation performance;
3) Apply and benchmark advanced features of Intel® TBB: priority of tasks, affinity of tasks;
4) Test the prototypes on more cores, particularly on Intel® MIC, tacking into account the architecture features.

Prototype algorithm

Moving to a task-based schema may result in some additional free parameters of the policy that need to be optimized for better performance. Current policy requires a threshold (`I`) 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. In the prototype there is some number (`II`) of events to be transported. At one moment there is only a fixed number (`III`) of event slots in the memory. The average track number (`IV`) for each event is an input parameter. Initially all slots are filled with events – a big number of tracks is injected at once into one collection. A dispatcher task is started. It pops that collection and distributes all the tracks into several baskets, pushing filled basket into the feeder queue and spawning propagation tasks for each pushed basket. A basket is considered to be filled if it has a given number (`V`) of tracks. While propagating the feeder queue gets consumed while the particles get sparse in the detector. When it gets to the minimum threshold (`VI`) a given number (`VII`) of older events are being prioritized. This policy is applied to keep the data structures in memory to a constant level and refresh the feeder queue regularly with efficient baskets. When an event in some slot is fully transported, a new event is injected into that slot. A number of threads to be used is limited by an explicit call to tbb::task_scheduler_init with a specified number of threads (`VIII`) as an argument.

The prototype uses a dynamic mechanism to spawn “dispatcher” and “propagator” tasks according to the basket flow and processing needs. In Intel® TBB there are different techniques for allocating, spawning and waiting for tasks. The prototype uses “scheduler bypass” and “continuation passing” templates featuring tbb::empty_task’s as successors.

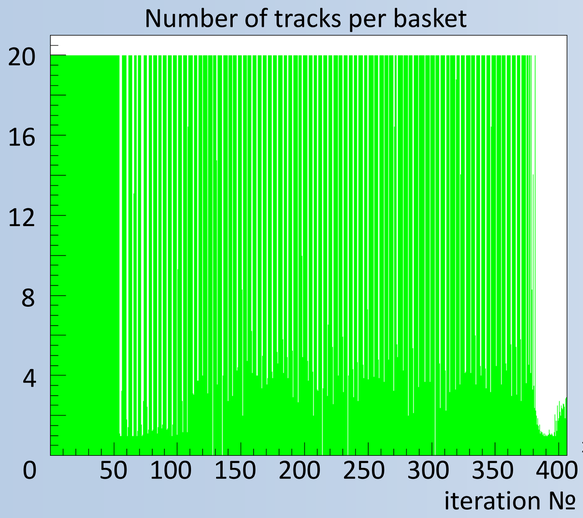




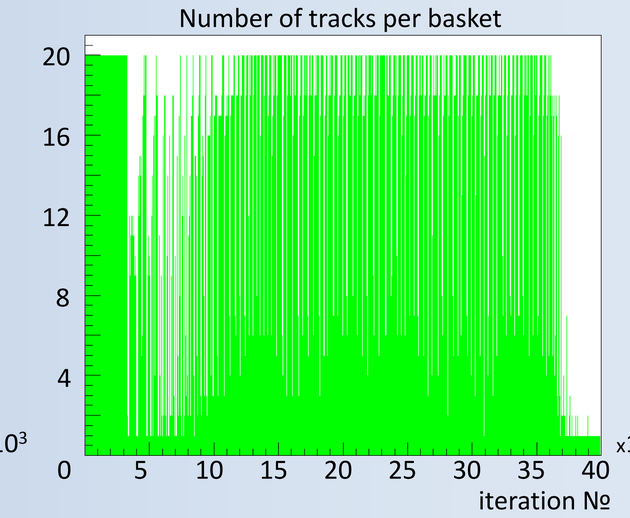
Preliminary results

High identity of results. About 10 times less iterations done at approximately same time.

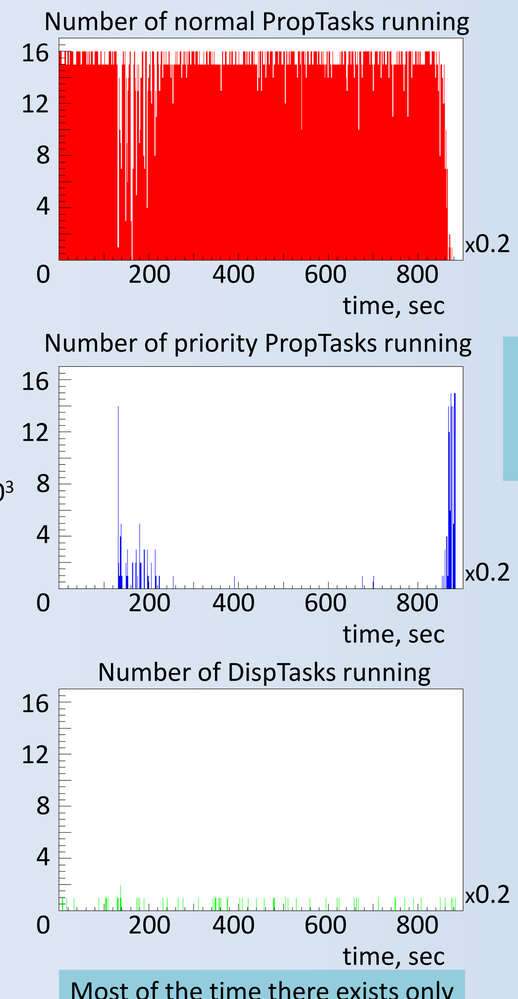
Original – pthreads



New – Intel® TBB

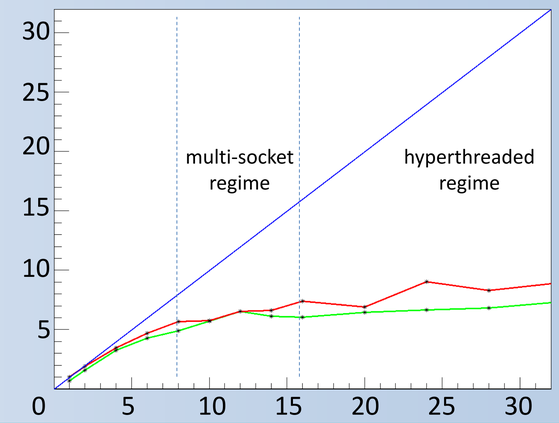


Most of the time spent on propagation



Most of the time there exists only one dispatcher, extra are spawned when needed

Scalability (normalized)
2 sockets – each Intel® Xeon E5-2690 (8 cores + 8 HT)
red – pthreads, green – Intel® TBB



Poor speedup both in pthreads and Intel® TBB models. Expected some improvement after implementing more realistic propagation. A detailed “locks and waits” analysis will be performed to understand communication bottlenecks.

Conclusion:

A task-based approach was applied to the particle propagation prototype using Intel® Threading Building Blocks template library. The results of the new prototype are close to expectations, however there are some features that need to be further understood, including unexpected increase of cache misses and comparatively low scalability.

Acknowledgements:

This work is supported by SC ROSATOM and Helmholtz Association (grant IK-RU-002) via FAIR-Russia Research Center.

References:

[1] Rethinking particle transport in the many-core era towards GEANT. Apostolakis, John; Brun, Rene; Carminati, Federico; Gheata, Andrei. J. Phys.: Conf. Ser. 396 (2012) 022014