Rethinking particle transport in the many-core era towards GEANT 5

John Apostolakis, René Brun, Federico Carminati, Andrei Gheata
European Organization for Nuclear Research (CERN) - Geneva, Switzerland
Andrei.Gheata@cern.ch

Abstract. Detector simulation is one of the most CPU intensive tasks in modern High Energy Physics. While its importance for the design of the detector and the estimation of the efficiency is ever increasing, the amount of events that can be simulated is often constrained by the available computing resources. Various kind of "fast simulations" have been developed to alleviate this problem, however, while successful, these are mostly "ad hoc" solutions which do not replace completely the need for detailed simulations. One of the common features of both detailed and fast simulation is the inability of the codes to exploit fully the parallelism which is increasingly offered by the new generations of CPUs. In the next years it is reasonable to expect an increase on one side of the needs for detector simulation, and on the other in the parallelism of the hardware, widening the gap between the needs and the available means. In the past years, and indeed since the beginning of simulation programs, several unsuccessful efforts have been made to exploit the "embarrassing parallelism" of simulation programmes. After a careful study of the problem, and based on a long experience in simulation codes, the authors have concluded that an entirely new approach has to be adopted to exploit parallelism. The paper will review the current prototyping work, encompassing both detailed and fast simulation use cases. Performance studies will be presented, together with a roadmap to develop a new full-fledged transport program efficiently exploiting parallelism for the physics and geometry computations, while adapting the steering mechanisms to accommodate detailed and fast simulation in a single framework.

1. Particle transport simulation at the dawn of an era
Monte Carlo codes developed along with computers from their early beginnings. From codes like the MCS/MCN(P) [1] suite and EGS [2] in the 60's to the modern simulation toolkits used by today’s HEP applications, simulation applications have generally followed the computing technology trends at the moment when their development has started. Very few of these codes changed the initial choice along the way even if some are half a century old. While this was true for many other HEP applications, simulation has been the most demanding in terms of CPU cycles and events for physics comparisons with real data. The first complex simulation models had to fit within few tens or hundreds of kilobytes of memory and had to trade precision for speed using algorithms with a low memory footprint. GEANT3 [3] introduced a compact and fixed size memory model with its ZEBRA format, supporting the concept of pointers. As memory was really precious at the time, this was closely managed by the application and always pre-allocated. The limited resources and increasing complexity of the user application required full awareness on the algorithm performance and implementation of low level optimizations. High-precision results could always be achieved by...
sacrificing simulation time. This was the case for codes like FLUKA [4], which aimed for very accurate predictions for many different applications, including high energy experimental physics.

As computers became more powerful, abstraction became a necessary feature for describing and solving more and more complex problems. Computer languages took the challenge of providing users with higher levels of abstraction, while compilers the one of preserving the performance via low level architecture-dependent optimizations. This increased the distance between the programming language semantics and the actual code executed by the processor. As the CPU clock frequency could be scaled up with each new generation and the increase of the number of transistors was exponential, memory containment became less critical and applications became faster. HEP applications could profit from the Object Oriented paradigm allowing for a high level of encapsulation of data and functionality, without having to care about resources as much as before. This led to writing much more complex programs but less efficient in terms of memory contiguity, cache misses and instructions per clock cycle.

Around 2003, the impossibility to continue scaling up the clock frequency had an immediate impact on the need of parallelism. Since then, the computation efficiency still increased according to Moore’s law, but it is now driven by the number of sockets and number of floating point operations per cycle, which led to a general trend of putting more and more computational units in the same box and building processors with higher IPC capabilities. This opened new technological opportunities that providers are now exploring and exploiting with fast changing hardware architectures coming in many hybrid configurations.

Figure 1. Technology trends for the number of transistors, CPU speed, power requirements and ILP. These motivate the increasing number of sockets, materializing in very powerful teraflop devices in many hybrid configurations.

Particle transport software mixes intensive geometry and physics calculations with the I/O performed by the client user application for storing the hits and digits structures. The classical transport engine behaves like a typical state machine as a particle is traversing the detector geometry.
The traditional transport algorithms are following particles and their produced secondaries until they exit the setup while communicating the state changes to the user code. The immediate implication of this approach is navigating very large data structures and firing most of the code when transporting a single particle. In addition, the widespread use of virtualization down to low-level objects increases by a large factor the instruction cache misses. This approach makes nowadays particle transport programs run inefficiently on the emerging parallel architectures. While event-level parallelism can still solve part of the problem, it cannot make this kind of approach more efficient and has to deal with synchronization overheads for digitization and merging.

The inefficiencies described above translate into very large amounts of time and resources needed for providing the large simulation statistics required by many of today’s HEP experiments. This fact has prompted the community to develop fast simulation alternatives. It has been shown that such approaches can provide considerable speed-ups of $\mathcal{O}(100)$ without reducing much the simulation precision for the interesting signal. This kind of optimizations are very detector and analysis dependent, and they have to be exploited together with detailed particle transport, giving more flexibility to the framework and allowing for important performance gains without losing coherency.

After a thorough investigation, and based on many years of experience, we concluded that an important boost in particle transport efficiency for the new parallel architectures can only be achieved by a deep re-design of the simulation code.

The following sections describe some of the main ideas for improvement together with the prototyping work performed to test their impact on the efficiency of a fine grain simulation application. We have started by re-designing the data structures and propagation algorithms to adapt to the multiple levels of parallelism offered by today’s technology.

2. A route towards a modern particle transport framework

From a structural analysis of the main components of a particle transport framework, we have tried to identify those that will clearly have to be re-designed in a fine grain parallel approach. For instance, a multithread-aware stack will need non-blocking thread-local hooks to easily rebuild the particle “history” in a completely different manner than a standard stack. Also, the changes required for a “vectorised” transport will certainly affect the main event and track loops compared to the standard approach. Not only the calling sequence for the physics processes and geometry navigation will have to be changed, but also the user scoring algorithms.

While it is not yet clear if or how particle vectors could be percolated into the low-level CPU-intensive physics and geometry methods, there is no doubt that without re-organizing the steering code vector-wise we will not be able to profit from the IPC features of modern CPUs. We have good reasons to believe that such code re-organization effort will produce efficiency gains beyond vectorisation by just pushing expensive decisions upstream and moving common computations outside the loops. There will be also “locality” gains in terms of data and instruction caches.

From the previous experience with existing simulation software, it is clear that the fast and detailed simulations should be able to co-exist in a single framework. This can be achieved by adding a virtual layer on top of the different possible track propagators, as presented in figure 2. The simulation framework can provide a standard detailed track propagator and a set of simple fast versions, like a $\eta-\phi$ propagator or a simple helix propagator for event displays. The idea is to allow for tracks to be propagated differently according to user-defined criteria. The interface should allow the inclusion of custom propagators, so that users can implement their own fast approach without having to interfere with the general steering of the framework. We have already a very positive experience with such an approach that has been used for the Virtual Monte Carlo framework [5], at a more global level, to virtualise the full functionality of different particle transport packages.

One of the main goals of a modern particle transport framework should be to profit from parallelism on many kinds of different resources. This raises a legitimate question: what kind of parallel technologies are we aiming for? Should we consider higher-grain task-oriented approach that will profit from the next generation of many-cores? Are we going to bet on the teraflops offered by
the next generation of Tesla GPU’s and write the whole code in CUDA [6]? Is it wise to make now clear technology or language choices for a 3 to 5 years development, while there is no “winning” trend on the market? Or is it possible to write smart adaptive code that will be able to run, may be not optimally but still efficiently, on any client system? It seems quite impossible to answer these questions at the point when technology is taking a turn, so what we can try as a first step is to form an educated opinion of what can be achieved or accessible, based on some simple prototyping.

![Diagram of a general particle transport framework](image)

**Figure 2.** Global view of a general particle transport framework, virtualizing the transport model. The framework uses ROOT as infrastructure and service provider, and is able to use existing or new physics provided from many sources. The data structures and flow together with the transport procedure have to be designed from a parallel perspective.

Our current effort is intended as a seed for a many-year collaborative project to develop the next generation simulation software. It is by no means an attempt to replace the existing simulation tools that have to be maintained for many years to come. It is common sense to consider reusing parts of the existing GEANT 4 code for the next step, since this includes many years of developments of well-tuned algorithms. The GEANT 4 project [7] itself introduced parallelism at event level and is considering further optimizing the code for different parallel resources. Our prototyping and benchmarking work is based on the fact that there is much more to be gained beyond event-level parallelism by re-casting completely the particle transport from a fine-grain parallel perspective.

While many of the ideas on possible alternative transport approaches came up after long discussions, it became clear that they have to be expressed in the form of a simple prototype that will allow us to “measure” the progress. In the next section we will describe the current version of this prototype together with some preliminary results and conclusions.

### 3. A playground for new ideas

The central idea that we have tried to prototype in a first phase was based on a simple observation: the particle transport in HEP detector geometries is very much localized, and the classical strategy in propagating the particles ignores the benefits that could come from this locality. The example from figure 3 shows that most of the time particles are tracked in only few percent of the geometry volumes, which is true for most known detectors. A typical example are the showers developed in calorimeters,
which are taking a very large amount of simulation time for transporting very few particle types in a replicated structure of cells. Why not trying to profit from having the corresponding data structures and code cached and reuse them for more than just one particle?

Figure 3. Particle transport in most HEP detector geometries is very much localized. The figure represents the ATLAS volumes sorted by total transport time. 50% of the time is spent in less than 1% of the volumes. The same is observed in many of the geometries available in ROOT format from a database of about 30 real HEP experiments that the prototype can use.

Going from this simple observation to an actual implementation that could profit from it, required a bit more effort. We needed a set of ingredients to build a toy model. The first step was to have realistic HEP experiments geometries and a thread safe modeller to handle geometry queries in parallel. The procedure for making the ROOT [7] geometry thread safe will not be described in detail, but we mention that we achieved this without having to lock memory resources beyond the initialization step. This was quite important for minimizing the overheads coming from competing for memory resources in our prototype. Optimizing the geometry structures for parallelism in general and for GPUs in particular is a very interesting subject currently under investigation.

A second step was to implement a minimal set of toy physics processes providing very basic physics emulation during particle transport. The modelled “processes” are: energy loss, scattering and a generic interaction. We will not discuss their implementation here but we plan porting a more realistic set of processes from GEANT 4 that would reproduce at a level of 50-80% the expected particle distribution and spectra. All physics processes derive from an abstract base class and implement specific stepping actions, taking as input a vector of input particles. The outputs for these methods are also vectors, containing either proposed step lengths or list of surviving tracks. In a multi-thread environment, physics processes use thread-local random seed generators and are pushing the thread identifier to the user scoring methods to allow for a non-blocking bookkeeping for the hit structures.

After putting together the ingredients, we have developed two types of transport: a single threaded one implementing the classical approach, and a vectorised version able to transport together a set of particles until they exit the current geometry volume. By volume we intend the replicated geometry instances of a single volume object (like a calorimeter cell). The state information about which volume instance is containing which particle is fully carried within the particle data structure, so that all particles traversing different replicas of the same volume and “feeling” physics-wise the same material
can be grouped and transported in a common vector. The point in developing the two versions of transport was to ensure that we get the same results with both and to measure the overhead of scattering/gathering particle lists. The particle vectors are not yet propagated to the CPU-intensive navigation methods, since the ROOT geometry modeller is not supporting yet vector queries, so there was no expectation of a speed gain at this level. This procedure was more a consistency test and allowed developing a basic vectorised transport.

The parallel transport model is represented in figure 4. The basic work unit is represented by a so-called *basket* of particles located in the same logical volume. Baskets have a variable threshold that represents the minimum number of particles for the basket to be actually transported by the system. In other words, in the transport flow, baskets are waiting until they reach this transportability threshold. The vectorised transport procedure mentioned above is executed by a number of worker threads, while a separate thread is acting as dispatcher. The schema will accommodate in future a separate thread to handle the I/O.

![Figure 4](image.jpg)

**Figure 4.** The current schema for the transport prototype. The work is done on baskets containing tracks from a single logical volume, which are handled by a single thread at a time, which transport particles till they reach the volume boundaries or otherwise disappear. Tracks exiting the volume are sent to a dispatching thread in a queue. The dispatcher regroups tracks in new baskets, putting them in the work queue in a continuous flow. The schema is able to prioritize the full transport of a subset of events and consequently flushing out the corresponding user structures (hits and digits). This feature will allow releasing or reusing the memory slots while injecting new events.

The work dispatcher handles general track containers that hold transportable tracks which are not grouped by volume. A given track container is always filled by a single thread at a time with tracks crossing the current volume boundaries into a new volume, so each transport thread holds such a container. A special case is when a track container is filled by the particle generation procedure, which starts the whole transport process. Note that we are injecting more than one event at a time since our goal is to really fill the CPU pipeline and have “good” vector populations. The difference with respect to a classical event generator is that this can be called before the previous events are completely transported to avoid depletion of particle populations.
The dispatcher thread holds a list of baskets, one for each logical volume. Note that the most complex LHC geometries have no more than few thousands such volumes. The dispatcher thread loops over the tracks from the input track container, moving them into the appropriate baskets, which get dispatched into the work queue as soon as they reach the transport threshold. The transport queue content is constantly monitored by the dispatcher which can take appropriate actions to ensure load balancing according to a low watermark. There are few choices for doing that as it will be explained next.

The work queue follows the principle that workers are idle until at least one object is available. The design is such that there are always available baskets in the queue to avoid dead times. Any worker thread will pick a basket at a time and start transporting it, invoking the vectorised physics processes and geometry querying. In future this process will produce hits stored in the global memory space in thread-reserved areas. Currently no hits are produced, so the prototype just simulates transport without I/O. When all tracks from a basket are transported (either entering into the next volumes or killed by physics processes), they end-up in a thread local track collection which is injected in a dedicated separate queue. This queue has the transport threads as providers and only the dispatcher thread as client, taking all available collections in a single iteration. The whole flow ends with a new iteration of the dispatcher over the collected tracks, to dispatch the workload into transportable baskets.

![Figure 5](image_url)

**Figure 5.** Evolution of number of baskets in the transport queue if the system does only garbage collections triggered by a low watermark threshold, without re-injection. The transport in such case suffers from sub-optimal track content in the transported baskets, due to a long depletion regime.

Some of the interesting parameters to monitor in this workflow are: the population of baskets in the main transport queue, the number of workers that are not idle (i.e. waiting for work in the queue) and the average number of tracks in the transported baskets. A typical evolution of these parameters with the dispatching iteration number is presented in figure 5. As shown by the upper plot, the initial injection of events produces a large number of transportable baskets, since all particles are generated in a single volume. As baskets get transported, particles get spread into different volumes of the detector, making harder to reach the transportability threshold. The number of baskets in the work
queue goes down, which would eventually end up in a blocking situation, when all remaining particles are kept in baskets below threshold all over the detector. To avoid this, our first approach was to trigger a garbage collection whenever the queue population goes below a low watermark. This just pushes to the work queue all baskets, including those with a single particle, boosting processing but with lower efficiency. The baskets in the queue get consumed again until the next garbage collection. This procedure can be repeated until all particles get fully transported and it was tested to use efficiently all available CPU cycles. The problem with such approach is that garbage collections become more and more often as particles get farther and farther from the generation point. The situation is somewhat similar to a champagne cascade: pouring champagne in the top glass will fill fast the ones just below, but much slower those at the bottom. The result is a depletion regime for the work queue that requires garbage collections after each iteration of the dispatcher.

![Number of baskets in the transport queue](image1)

![Number of active workers](image2)

![Number of particles/basket](image3)

**Figure 6.** Introduction of a double-ended work queue in the prototype allowed evacuating ranges of events with priority. This new schema allows for almost perfect concurrency while keeping the track content of baskets at efficient levels.

To avoid this depletion regime we could just keep injecting generated events, however this may lead to memory inflation. So we have tested the following scenario: what if instead of doing garbage collections we just transport some events with higher priority, so that in future we will be able to inject more events without jeopardizing the memory? Using our analogy, to allow for a continuous flow, we would need to pour more champagne at the top, but we cannot do that unless we are able to collect the one from the bottom. It was not clear if it was even possible to prioritize events in an efficient way, since they get all mixed up in the different baskets. After several tries, the winning solution was to transform the work queue in a priority queue, by using double-ended queue mechanism. The normal flow of baskets in the queue is that they get inserted at the top and extracted by workers from the bottom. We introduced the concept of priority baskets, which get injected directly at the bottom of the
queue so they get transported first. To allow transporting a given event range, we added an array of priority baskets and the mechanism to fill them with selected tracks at the dispatching level.

The behaviour of the system changes in the following way: once the low watermark is reached, we start prioritizing events in a given event number range. These will be transported with priority but in a non-blocking manner, since the work queue still contains non-prioritized baskets that can balance the workload. After the selected events get fully transported, the same number of generated events can be injected in the system. After the selected events get fully transported, the system changes in the following way: once the low watermark is reached, we start prioritizing events in a given event number range. These will be transported with priority but in a non-blocking manner, since the work queue still contains non-prioritized baskets that can balance the workload. After the selected events get fully transported, the same number of generated events can be injected in the system. At this point of the development we just start prioritizing the next range of events, but in future this phase will trigger the user digitization procedure and the I/O to disk. A given event can be digitized by the user code only after being fully transported. This step allows releasing the user hit structures which can have an important size depending on the geometry and application. The current implementation is sub-optimal due to missing re-injection of events, but the exercise so far has shown that the event evacuation phase is much shorter than the total simulation time as shown by the trends in figure 6, which is very good news.

The current implementation is sub-optimal due to missing re-injection of events, but the exercise so far has shown that the event evacuation phase is much shorter than the total simulation time as shown by the trends in figure 6, which is very good news.

Figure 7. Performance of the prototype is being continuously monitored. Benchmarking the different steps in this development are very important for understanding the impact of different design or implementation changes to the performance of the prototype.

We have continuously evaluated the prototype in terms of concurrency. The current version shows a good scalability of the speed-up with respect to the number of threads, as shown in figure 7. We have so far understood that one of the main sequential bottlenecks is due to usage of queues, which are however the best choice in terms of work balancing. More detailed performance studies are ongoing.

4. Future developments

The following steps in our investigation are to include digitization and I/O in the general flow, to measure the actual throughput that could be achieved. We intend to implement this in a completely non-blocking manner, which would allow us to reproduce the whole simulation chain in a multi-thread environment, without major sequential overheads. It is obvious that we will need more realistic physics processes to reproduce more accurately the actual particle distributions and tune the parameters of our transport model.
After having a complete particle transport flow with the new schema, we will focus on different aspects of the framework, like the interface allowing switching the transport model. Note that a fast simulation model can be looked at as a special geometry volume working with dedicated track baskets and specialized physics processes, which can be naturally integrated with our current transport approach.

So far we did not prototype any transport optimisations on GPU’s. It is known that the architecture and memory usage patterns for such resources can lead to important speed-ups in case of well-contained CPU intensive tasks. A common experience is that the need for frequent data communication with the CPU memory introduces very large serial overheads which can be overkill for parallelism. Since the particle transport is a state machinery which needs to push the state and extract results from any computational level, an efficient GPU-based transport model is hard to design and implement. This would require hybrid architectures that could work as data flow machines for a large number of computation units. Many providers are designing such new architectures and probably make them soon available on the market. It became clear that independent of the technological aspects, the memory model and data flow for the transport application will have to be adapted for more contiguity while identifying the right segmentation of work that could be dispatched to such resources.

5. Overview and conclusions

Particle transport simulation has a history that starts together with the first computers. It has recently become a limiting factor for the possibility to perform detailed systematic studies in many of the modern HEP experiments. While trying to improve this situation by doing minimal changes on the existing frameworks to allow for event-level parallelism, we have a window of opportunity to adapt the simulation software to the new emerging architectures. This effort is obviously substantial and requires a deep review of the transportation strategy from a parallel perspective, but it is a necessary step to take to stay efficient in a parallel world.

We have started to evaluate the parallelization and vectorisation opportunities in a multi-thread environment, based on a simple prototype. This was intended not only as a proof of principle, but to actually see how efficient we can implement a new approach for parallel particle transport that may potentially profit from data and code locality. The results are so far very promising but are exposing the new dimensions of the problem. Our prototype mostly focused on the steering level of the transport application, which is of course important and which allows factorizing the problem, but this is certainly not enough. Other approaches are trying to port very specific CPU intensive algorithms to low level computation units. While the current architectures does not allow yet to efficiently use on GPUs large scale applications optimized for CPU cores, it is clear that we have to identify or create computational kernels within the simulation framework which could benefit from such resources. This involves in many cases compacting the data structures and rethinking the data flow for minimizing the need to transfer the state parameters between computation blocks. Putting together the two approaches efficiently will be one of the major challenges to come.

The aim of this paper was to present our first steps in the development of the new generation simulation programme GEANT5. The ideas presented here are the outcome of many discussions based on a long experience with particle transport simulations. They are based on the analysis of the existing simulation software with respect to the technology trends and aim to lay out a roadmap for the coming 3 to 5 years. While preserving the large existing database of knowledge and simulation algorithms, it is clear that the transport software has to be re-factored to be able to achieve better efficiency on tomorrow’s hardware and more integration between fast and full simulation. We hope that this project will progressively attract collaborators from the HEP community who will join us in prototyping and testing the new ideas that will be the foundation for the next generation GEANT simulation software.

References
[1] R R Johnston 1963 A General Monte Carlo Neutronics Code *Los Alamos Scientific Laboratory*
Report LAMS–2856
E D Cashwell J R Neergaard W M Taylor and G D Turner 1972 MCN: A Neutron Monte Carlo Code Los Alamos Scientific Laboratory Report LA–4751

[2] R L Ford and W R Nelson 1978 The EGS code system – Version 3 Stanford Linear Accelerator Center Report SLAC-210

[3] Application Software Group CN Div 1993 GEANT Detector Description and Simulation Tool (Version 3.21) CERN Program Library W5013

[4] A Fassò A Ferrari J Ranft and P R Sala 2005 FLUKA: a multi-particle transport code CERN-2005-10, INFN/TC_05/11, SLAC-R-773

[5] I Hrivnacova D Adamova V Berejnoi R Brun F Carminati A Fasso E Futo A Gheata I Gonzalez Caballero A Morsch 2003 The Virtual Monte Carlo arXiv:cs/0306005

[6] J Nickolls I Buck M Garland K Skadron 2008 Scalable Parallel Programming with CUDA Queue - GPU Computing Volume 6 Issue 2 (13pp)

[7] S Agostinelli et al. 2003 Geant4 - A Simulation Toolkit Nuclear Instruments and Methods A 506 (53pp)