Efficient parallel tracking 2008 - 2009

Håvard K. F. Bjerke

CERN openlab
30 January 2009

Technical report: Efficient parallel tracking

Håvard K. F. Bjerke

January 30, 2009

Contents

1 Introduction

As a result of increasing density of transistors as well as heat and power constraints, computers increasingly implement parallelism in hardware in order to speed up computation. This has manifested itself in multi-core processors and wider vector instructions. Future computer architectures are expected to increase parallelism in both dimensions. For example, the Intel AVX extensions will have double the SIMD register width of the SSE registers [1]. The forthcoming Intel Larrabee will also have wide SIMD registers, augmented by a large array of x86 CPU cores [2].

In order for High-Energy Physics analytical software to be able to fully utilize the processing power provided by parallel processors, their algorithms must exhibit at least the same degree of parallelism as provided by the processors. Since one can expect a continued increase in parallelism in processors, a proactive choice or design of an algorithm is one that does not exhibit a constant or predetermined number of workloads that can run in parallel. A program that scalably maps itself to both current and future expected parallel processors can be characterized as "forward-scalable." Scalability is further discussed in Section 1.1

This technical report describes a project exploring and developing a forward-scalable approach to high-energy physics particle tracking in the ALICE experiment. The project is a collaboration between members of the Kirchhoff Institute for Physics at the University of Heidelberg, Intel Brühl and CERN openlab. Some of the challenges concerning tracking are presented in Section 1.2

1.1 Scalability

There are many strategies for extracting parallelism from a workload or a set of workloads. Popular taxonomies for parallel computing are, Multiple Program Multiple Data (MPMD), Multiple Program Single Data (MPSD), Single Program Multiple Data (SPMD) and Single Program Single Data (SPSD), as described in Table 1.1.

Name	Synchronization	Parallelism	Example	MP Constraints
MPMD	Asynchronous	Task	Web server	Shared resources
MPSD	Locks	Task	Producer-consumer	Intrinsic, synchronization
SPMD	Barriers, locks	Data	MPI, OpenMP, SIMD	Synchronization
SPSD	Implicit	None	Serial	None

Table 1: Multi-programming paradigms.

The feasibility of a strategy depends on the characteristics of the paral-

lelism exhibited by the algorithm and its data-structures. For example, for an algorithm that exhibits MPMD parallelism, synchronization may be unnecessary and may cause unnecessary overhead. For an SPMD algorithm, however, synchronization is often necessary.

Also, the forward-scalability of a strategy depends on the level of parallelism given by it and its constraints. For example, a web server that can serve only a limited number of clients does not exhibit forward-scalability. Programs that exhibit data level parallelism, however, can often be characterized as forward-scalable, although, in this case, the degree of parallelism in the data can put a constraint on the scalability.

As another example, running multiple SPSD processes in parallel becomes MPMD. Going from SPSD to MPMD proportionally exhausts shared I/O and memory resources. Conversely, parallelizing with an SPMD model alleviates the constraint of shared resources, but adds the constraint of synchronization.

Finally, parallelizing any program usually also adds a new set of constraints that are intrinsic to the problem, which also has to be taken into account.

1.2 Tracking in the ALICE High-Level Trigger (HLT)

When particles collide in the center of the LHC ALICE detector, a set of resulting particles are produced. As these particles pass through detector layers, the HLT delivers electrical signals representing the particles' trajectories, mass and velocity. These signals are imprecise observations, and therefore the data needs to be reconstructed in order to gain more meaningful and precise information, before it is stored.

One of the most important detectors is the Time Projection Chamber (TPC) [4]. It is a cylindrical volume divided into 12 sectors. The barrel is filled with a gas, and as particles pass through the gas it is ionized, which leaves a trace of electrically charged gas particles. An elecrical field pulls the charged gas particles towards a two-dimensional grid, consisting of a number of *pads* or *cells* that detect the impact of the elecrical charge.

The particles' flights through the TPC are represented as discrete *hits* in a three dimensional grid. Based on the timing of the drift and the location of the impacted pads, the three-dimensional cartesian coordinates of the original particle's interaction with the gas can be found, and thus also the trajectory of the particle.

The presence of a particle in a particular space point will usually result in a number of pads being triggered, centered around the impact point. This is illustrated in Fig. 1. In this two-dimensional example pads that are triggered in the same row, centered around the same track, are grouped into *clusters*.

Two of the most significant steps in reconstructing particle tracks are *finding* and *fitting* the tracks. Finding tracks involves finding out which of the hits belong to which tracks of the same particle. Fitting means estimating the real tracks based on the imprecise measurements that are delivered by the TPC.

The HLT tracking software runs on a cluster of SMP machines connected to front-end electronics to the HLT [6]. Data from each sector is transferred to

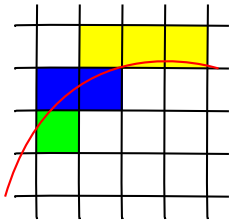

Figure 1: The triggering of pads from a track.

separate sub-clusters in a hierarchical structure optimized for data-flow. Each sector is processed on separate nodes, and, in order to merge tracks that cross between sectors, sectors are combined further up in the hierarchy of nodes.

2 CA track finder

Two classical example algorithms for finding tracks are Hough transform and conformal mapping, which are usually limited to simple event topologies. More recent algorithms are track following with the Kalman filter, neural networks and Cellular Automaton (CA) [7, 8].

A cellular automaton track finder has been developed for the CBM experiment, using SIMD [9].

A traditional CA algorithm is Conway's Game of Life (GOL). It is traditionally implemented as a grid of cells that can be either dead or alive. The state of each cell is modified in step, governed by a set of rules:

- Starvation: A live cell with fewer than two live neighbouring cells dies.
- Overpopulation: A live cell with more than three live neighbouring cells dies.
- Survival: A live cell with two or three live neighbouring cells survives.
- Birth: Any cell with three live neighbouring cells survives.

Some interesting properties of CA are

- *Locality:* The state change of a cell is governed by the state of its neighbours.
- *Parallelism:* The cells' state changes are independent from each other, within one iteration.

In order to use CA for the purpose of finding tracks, the algorithm is modified while keeping intact locality and concurrency.

The CA algorithm holds promises of being efficient both in terms of finding tracks and execution time.

2.1 The AliRoot track finder

A CA track finder for the AliRoot framework is being developed by Sergey Gorbunov and Ivan Kisel. It contains routines for track finding, most notably

- reading and parsing collision events
- CA track finder
- track fitting
- global merging of tracks between sectors
- measuring the efficiency of the algorithm

3 Visualization

Visualizing the process of finding tracks significantly eases the development of the algorithm. By having a view of the TPC and the process of the CA one can immediately see the impact of changes to the algorithm.

Two libraries have been used for visualization in the track finder. VTK has been used to visualize the TPC and its sectors. The visualization allows an interactive view of the CA process. Fig. 2 shows a VTK visualization of a simulated event in a pseudo-sector after the tracks have been found. The individual tracks that the track finder has found can be identified by color.

The CERN Root library was used in order to display histograms of the tracker's performance, both in terms of tracking and timing. It was also used in earlier development stages to display a two-dimensional grid for the purpose of exploring a two-dimensional tracking algorithm.

Initial development of the VTK visualization, including interactive tracking of individual sectors, was done by Intel in Brühl. Functionality for visualizing the TPC barrel and individual sectors was added later, based on this work. A later implementation of the visualization is shown in Fig. 3. It presents two view-ports into the TPC: The left view-port gives a global view of the TPC containing an event, and the right view-port interactively cycles through the sectors of the TPC.

4 Algorithm design & optimization

Our track finding algorithm is built on an implementation of GOL. Originally, Ralf Ratering developed an optimized GOL benchmark, which was later extended to include routines for tracking. This included simple simulation of tracks, visualization, the tracking CA and performance histogramming.

It was later decided that the algorithm should be integrated with the AliRoot framework. This allows us to exploit some of the already existing routines as listed in Section 2.1. Our approach was to replace the existing tracking algorithm with ours, in order to get a more optimized tracker.

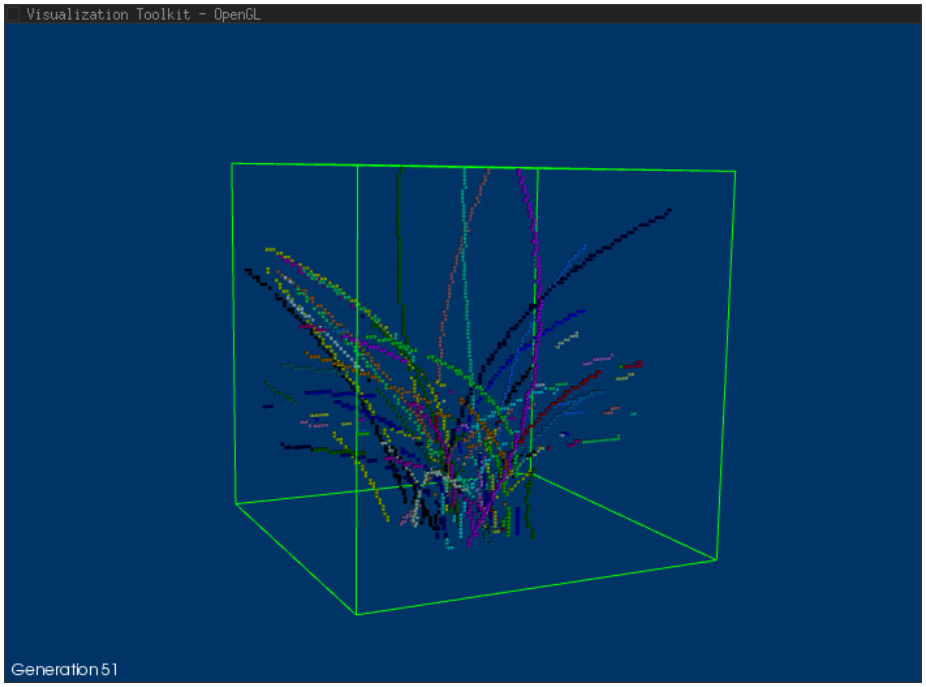

Figure 2: VTK visualization

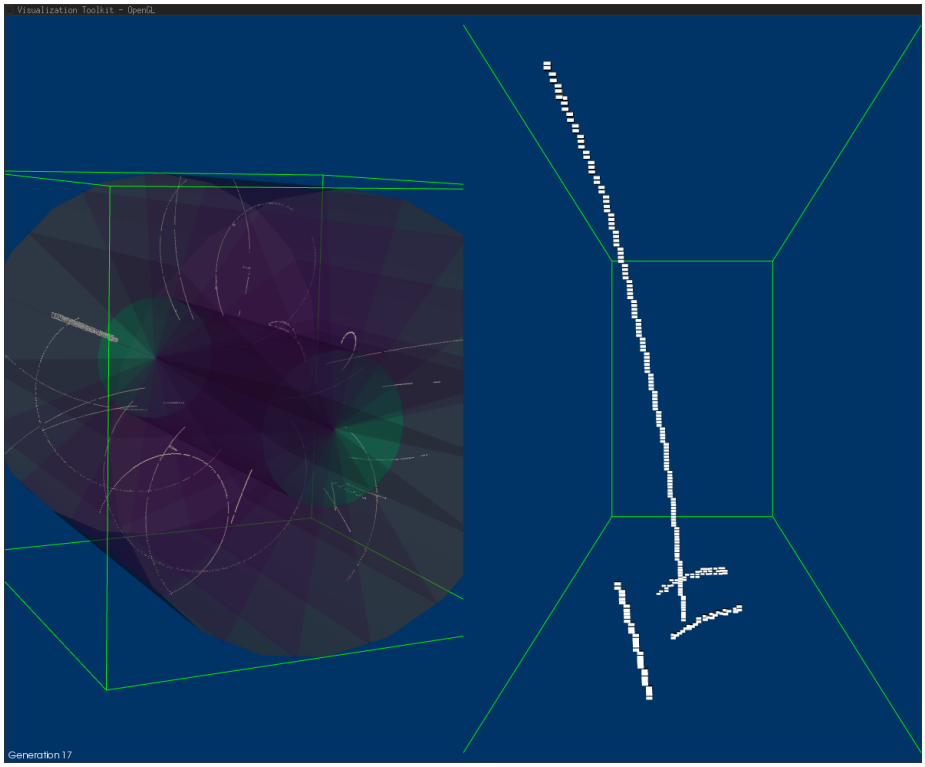

Figure 3: Dual view-port visualization

In our internal representation of the event data, each sector of the TPC is transformed into a uniform 3-dimensional grid. Each cell in the grid represents the presence or absence of a hit covering that cell. In other words, each hit is digitized, or binned, from the original continuous coordinates and dimensions of the hit to a discrete grid.

The rules that govern the survival of a cell modified in this way, in which a *cluster* is a collection of cells that make out a hit:

1. Starvation: A cluster with no immediate neighbouring live cells above dies.
2. Overpopulation I: A cluster with a number of immediate neighbouring clusters above or below over a given threshold dies.
3. Overpopulation II: A cluster with a number of immediate neighbouring cells above or below over a given threshold dies.
4. Survival: All other live cells survive.
5. Birth: No new live cells are created.

The rationale behind these rules are

1. Gives a reference to the endpoint of a track
2. A case with a likelihood of two or more tracks crossing each other. Such a cluster ambiguates the finding process.
3. A case with a likelihood of two or more tracks crossing each other or resulting from particles travelling perpendicular to the normal track direction. This ambiguates the finding process. The tracks should be identified from segments travelling along the tracking direction.
4. The remaining clusters represent track segments with less ambiguity and without intersection.
5. No benefit of creating new live cells is known.

4.1 Parallelism constraints of Cellular Automaton

GOL is in its entirety a sequential algorithm. After a number of iterations, the state of a cell depends on the previous state of any other cell. Per iteration, however, the state of each cell only depends on its own and adjacent cells' previous state. This makes GOL suitable for SPMD parallelization: In a single iteration, each cell's state can be calculated in parallel, but the state has to be synchronized before the next iteration. If memory is shared, e.g. OpenMP, synchronization means waiting for all threads to finish their calculation. If memory is distributed, e.g. MPI, this means communicating the result of all adjacent cells to all logically adjacent nodes.

```
__m128i sum = _mm_setzero_si128();
int offset_x = -1;
for(int offset_y = -2; offset_y <= 2; offset_y++){

for(int offset_z = -2; offset_z <= 2; offset_z++){
int offset = coord2ext_offset_rel(offset_x, offset_y, offset_z);
unsigned char *ptr = sse_local_grid + ext_vector_offset + offset ;
__m128i neighbor = _mm_loadu_si128((const __m128i *)ptr);
sum = _mm_add_epi8(sum, neighbor);
}

}
```

Figure 4: Neighbor counting using SSE intrinsics.

Traditionally GOL is run a number of iterations to reach a final state. The cost of synchronization, however, is so large that it outweighs the benefits of running multiple iterations, thus in our algorithm only one iteration is run.

4.2 Vectorization

A cluster and the relation between clusters are represented by a few different data structures. The most important data structure is a discrete three-dimensional grid, which at each cell indicates the presence or absence of the center of a cluster. The dimensions of the grid corresponds roughly to the dimensions of a sector of the TPC, which means that a cell corresponds roughly to a pad in the TPC.

In order to keep a low profile in memory and cache and to coincide with the SSE EPI8 format, each cell is a byte value. Using the EPI8 format gives the highest possible vectorized throughput. The same format is used to count the number of neighbors (maximum 50) and give the relative offsets to the (post-CA) surviving neighbors.

The EPI8 SSE format has a throughput of 16 8-bit integer operations per instruction. For example, in our CA, each neighbor of 16 adjacent cells is counted in one instruction. However, each cell is not live (a cluster present), so the effective speedup, on average, is lower than 16, and, since vectors containing zero live cells are ignored, higher than 1.

Fig. 4 shows how the neighbors of 16 cells can be counted in parallel. Note that this code is not fully optimized. It can be further optimized by using aligned loads and shifting the contents of the registers.

One drawback with SIMDized counting of neighbors is that the search has to be exhaustive–there is no control flow in SIMD. In a SISD stream, the search for neighbors can be stopped as soon as enough neighbors to make the decision to kill the cell, have been found.

The pad occupancy in heavy ion collision events ranges from 15 per cent in the outer part of the volume of the TPC to 40 per cent in the inner part of the volume [4]. We can expect that the average cluster occupancy in the grid will be lower, since the grid only records the centers of the clusters. However, our CA algorithm does not take into account vectors that do not occupy any clusters, and thus the average occupancy of the processed vectors is higher than the average occupancy of the grid.

In combination with the CA run, a linked list of the surviving neighbouring clusters is created. These linked list are already representations of candidate track segments.

4.3 Parallelization

The process of reading out the track segments is a matter of following the lists of linked clusters. Following a linked list can be expressed as a recursive function, f. Following a set of linked lists can be expressed as a for-loop, looping over f:

```
for each list
f(list)
```

Our CA overpopulation and starvation rules guarantee that no two track endings directly or indirectly link to the same cluster, i.e. no cluster can belong to more than one list. This makes the process of collecting the track segments trivially parallel. Each loop in the above example can therefore run concurrently instead of consecutively, without conflict.

The degree of parallelism that can be achieved with this method is equal to the number of found track segments and proportional to the number of tracks. With heavy-ion collisions usually resulting in thousands of tracks, the degree of parallelism should be sufficient for many-core processing.

Intel Threading Building Blocks (TBB) is a C++ library for developing multi-threaded applications [3]. TBB facilitates the parallelization of a for-loop with the "parallel_for" template, as shown if Fig. 5. In this example, the number of track endings found is 32, which is represented by the string of blue squares. In the serial version, the for-loop loops over $\#tracks/4$ vectors of tracks in sequence. For the parallel_for implementation, a grain size is chosen so that it contains a big enough working set to justify a thread dispatch. In this example a size of $\#tracks/4$ is chosen. The number of tasks that can run in parallel in this example becomes $\#tracks/grain\_size = 4$.

5 Results

The track finder can benefit from both SIMD and multi-threading concurrency in multi-core processors. Given the cell-level data parallelism of the CA algorithm, the degree of parallelism in the algorithm far exceeds the available concurrency in hardware.

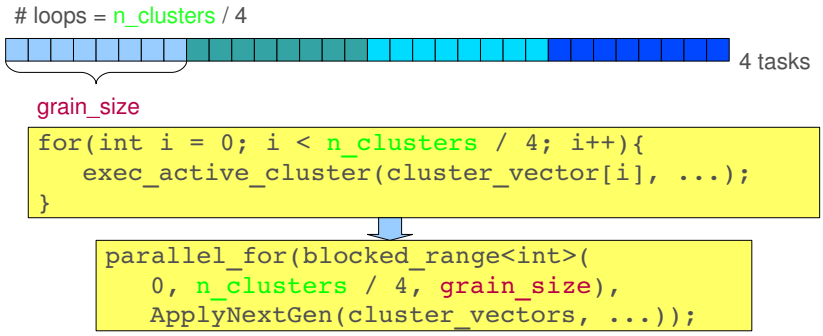

Figure 5: Parallelization with TBB

The performance measurement routines of the AliRoot framework show a tracking efficiency for proton-proton events of between 88 % and 100 % for reference tracks. Efficiency can be higher if the algorithm is properly integrated with the merging framework. The performance measurement routines are not yet able to calculate the efficiency of Pb-Pb events.

The visualization gives an interactive step-by-step view of each iteration of the CA and the resulting found tracks, distinguished by color. It is useful for displaying and debugging proton-proton events. However, for displaying Pb-Pb events, the VTK display is slow and consumes a lot of memory.

6 Conclusion

It is feasible to exploit concurrency in hardware, in a CA track finder. Using a data-parallel algorithm allows the track finder to scale across the SIMD and multi-core axes of concurrency in modern processors.

Taking the SPMD approach, using threading and SSE, also allows better streamlining of data to reduce the contention of shared resources. Given the constraints of the HLT cluster, the higher-level model is a hybrid between MPMD and SPMD: Individual sectors are processed at different nodes, thus only the tracking within one sector is SPMD. This data-flow may hinder the exploitation of parallelism in that it excludes the possibility of a sector-level parallelism model.

7 Future work

A more efficient event display is needed for Pb-Pb events. In order to achieve a full display of Pb-Pb events, the VTK display must be used in a more efficient manner. The Root event visualization [5] should also be evaluated for this purpose.

Also, there are many opportunities left for optimizing the speed of the algorithm. For example, there are several data-structures that can be better optimized for cache utilization, and the SSE specific code can also be better optimized.

Finally, the goal of this work is scalable tracking for today's and future multi-core architectures. This work shows that it is possible to exploit an increase in hardware parallelism, but the code itself does not necessarily map onto other or future architectures. Less architecture dependent programming frameworks, such as OpenCL and Intel Ct, may better support future changes in architecture.

References

[1] *Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency*, Intel Whitepaper (2008).

[2] L. Seiler et al, *Larrabee: A Many-Core x86 Architecture for Visual Computing*, ACM Transactions on Graphics, Vol. 27, No. 3, Article 18 (2008).

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

[4] K. Aamodt et al, *The ALICE experiment at the CERN LHC*, JINST (2008).

[5] M. Tadel, *Raw-data display and visual reconstruction validation in ALICE* (http://indico.cern.ch/contributionDisplay.py?contribId=442&sessionId=23&confId=3580)

[6] R. Bramm et al, *High-level trigger system for the LHC ALICE experiment*, Nuclear Instruments & Methods in Physics Research (2003).

[7] A. Glazov et al, *Filtering tracks in discrete detectors using a cellular automaton* Nuclear Instruments & Methods in Physics Research (1993).

[8] I. Kisel, *Reconstruction of tracks in high energy physics experiments*

[9] S. Gorbunov et al, *Fast SIMDized Kalman filter based track fit*, Comp. Phys. Comm. 178 (2008) 374383.