DE GRUYTER OPEN
Acta Univ. Sapientiae, Informatica **9,** 1 (2017) 49–64

DOI: 10.1515/ausi-2017-0004

Parallel k_t jet clustering algorithm

Dedicated to the memory of Antal Iványi

Richárd FORSTER
Eötvös University
email: forceuse@inf.elte.hu

Ágnes FÜLÖP
Eötvös University
email: fulop@caesar.elte.hu

Abstract. The numerical simulation allows to study the high energy particle physics. It plays important of role in the reconstruction and analyze of these experimental and theoretical researches. This requires a computer background with a large capacity. Jet physics is an intensively researched area, where the factorization process can be solved by algorithmic solutions.

We studied parallelization of the k_t cluster algorithms. This method allows to know the development of particles due to the collision of high-energy nucleus-nucleus.

The Alice offline library contains the required modules to simulate the ALICE detector that is a dedicated Pb-Pb detector. Using this simulation we can generate input particles, that we can further analyzed by clustering them, reconstructing their jet structure. The FastJet toolkit is an efficient C++ implementation of the most widely used jet clustering algorithms, among them the k_t clustering. Parallelizing the standard non-optimized version of this algorithm utilizing the available CPU architecture a 1.6 times faster runtime was achieved, paving the way to drastic performance increase using many-core architectures.

Computing Classification System 1998: I.1.4
Mathematics Subject Classification 2010: 58A20
Key words and phrases: jet, cluster algorithm, database of experimenthal particle physics parallel computing, multi-core, C++11

1 Introduction

We studied the structure of the jet in the high energy physics using the many-core architecture of the modern CPUs. We consider the important concept of the particle physics.

The parton model was introduced by Richard Feynman to analyse the high-energy hadron collisions. Any hadron can be considered a composition of a number of point-like constituents.

In the theoretical physics the Quantum Chromodynamics (QCD) is the theory of strong interaction which describe the interaction between quark and gluon [5]. The QCD analogue of electric charge is a property called color. The phenomenon of color confinement, that no particle with colour charge which can be observed on its own, was introduced. So the color neutral particles are examined, then we can measure the bound states, hadrons, which are composed of quark-antiquark pair, meson or three quarks, so called baryon [6, 7].

Parton is useful for interpreting the cascades of radiation produced from QCD processes and interactions in high-energy particle collisions.

Modern CPU architectures are capable of running multiple threads at the same time, allowing the developers to utilize more resources at the same time for the same application, increasing it's performance. With the increase of complexity and performance standard tools are including more tools to ease the development for those architectures. The C++11 standard contains important tools for threading, like conditional variables and mutexes, that gives more control over the parallelized algorithms. Using these tools and applying them in a reasonable way by taking into consideration the hardware and operating system limitations a 1.6 times better performing k_t clustering was achieved without doing any additional optimizations on the code based on the FastJet libraries solution.

2 Jet in high energy physics

In the experiment the conception of jet differs from the picture which was presented by theoretical physics. We observe final state particles moving in one direction, because the information about particle origin was lost during hadronisation phase of collision. Therefore, when speaking about jets then we speak about collimated sprays of particles (Figure 1).

Therefore one should have to map a set of particle to a set of jets by special rules. These rules define a jet algorithm [8]. This method contains some

parameters, which allow us to describe the behaviours more precisely. All together they determine the definition of the jet. We apply these consideration to understand the structure of jet in the high energy experimental and theoretical physics. The results of theoretical QCD are built in the simulation, recombination and analyses of the jet research.

2.1 Jet kinematics

We introduce some quantities which are important to know in the k_t jet algorithm. The interacting partons are not generally in the centre-of-mass of colliding system, because the fraction of the hadron momentum is changing from event to event which is specified by each partons. The jets can be introduced by longitudinally boost-invariant variables because the centre-of-mass system of the partons is boosted along the direction of the colliding hadrons randomized. The mass, transverse momentum, azimutal angle and rapidity are introduced by the next expressions:

mass:	$m = \sqrt{E^2 - p_x^2 - p_y^2 - p_z^2}$
transverse momentum:	$p_T = \sqrt{p_x^2 + p_y^2}$
azimutal angle:	$\Phi = \arctan(p_y/p_x)$
rapidity:	$y = \arctan(p_x/E) = \frac{1}{2}\ln\frac{E+p_z}{E-p_z}$

In the high energy limit, when $|p| >> m$, the directly measured quantities are the following

energy E or the transverse energy: $E_T \sin\Theta \cong p_T$
the azimuth: Φ
the pseudo-rapidity: $\eta = -\ln[\tan(\Theta/2)]$,
where the polar angle is given by $\Theta = \arctan(p_T/p_z)$.

3 Jet algorithm

The origin of the basic publication of the jet finding method is Sterman and Weinberg [4] and there is huge literature which was published about the newer version of these processes [16, 15, 3, 1]. These algorithms can be divided into two major groups: cone algorithms and sequential recombination algorithms.

3.1 Cone algorithms

In the case of cone algorithm we study that particles which are situated inside the conical angular regions, then the sum of the particle's momentum concurs

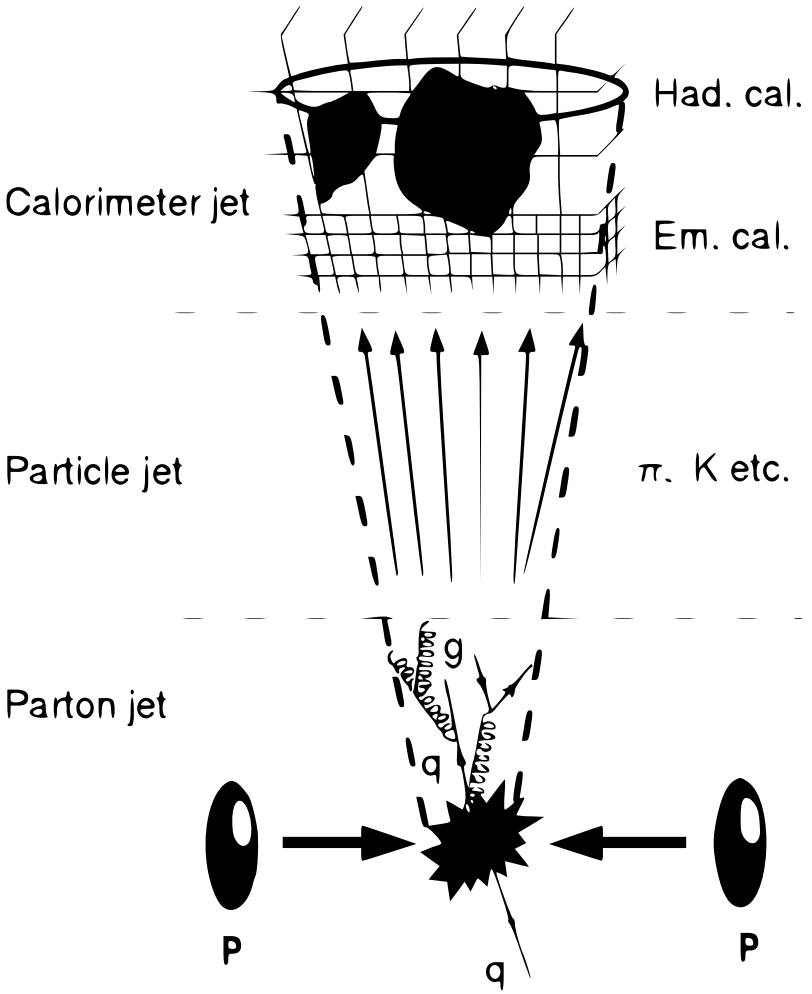

Figure 1: The structure of jet.

with the cone axis. The QCD radiation and hadronisation don't change the direction of a parton's energy flow. The stable cones are close to original partons in the direction and energy. The discrepancy between these cone algorithms are the strategy to find the stable cones and that process which is applied in that cases where the same particle is found in multiple stable cones.

3.2 Sequential recombination algorithms

This type of the algorithm identifies the closest particles in a pair to calculate the distance measure and recombine them. This process is repeat again, until it reaches a stopping criterion. The distance measure depends on the divergence of the perturbative QCD. The differences among the sequential recombination algorithms are the selecting of the distance measure and the stopping parameters.

3.2.1 The clustering algorithms

The k_t algorithm uses the final state particles in a shower which are collinear, it means that they have small transverse momentum between their constituent particles [10]. All sequential clustering algorithms have similar method. We define two distance variables.

The first of them is the one between two particles i and j, where $d_{ij} = \min\{p_{ti}^a, p_{tj}^a\} \cdot \frac{R_{ij}^2}{R}$ and a is an exponent which means the kind of the particular clustering algorithm. The value R_{ij} is determined by this expression $R_{ij}^2 = (\eta_i - \eta_j)^2 + (\Phi_i - \Phi_j)^2$ is a distance between the two particles i and j in the $(\eta - \Phi)$ space and R is the radius parameter which specifies the final size of the jet.

The second distance variable is $d_{iB} = p_{ti}^a$ this means the distance between the beam axis and the measured particle in the momentum space.

In the sequential clustering algorithms that process plays important role to find the minimum of the entire set $\{d_{ij}, d_{iB}\}$. If d_{ij} is the minimum value of the distance then particles i and j are unified into one particle (ij) and it is calculated sum of four-vectors after which i and j are removed from the list of particles. If d_{iB} is the minimum, then object i is becomed part of a final jet and removed from the list of particles.

This process is repeated until there are particles in a jet, where the distance between the axes R_{ij} greater than R, this process is an inclusive clustering. Otherwise the all amount of jets have been found, this is exclusive clustering.

k_t algorithm The a value corresponding to the k_t algorithms is 2.

The first step of the method is creating a list of the distance in the momentum-space and the distance from the beams. This algorithm involves a distance value d_{ij} between all pairs of particles i and j

$$d_{ij} = d_{ji} = min(p_{ti}^2, p_{t,j}^2)\frac{\Delta R_{ij}^2}{R^2}, \tag{1}$$

where p_{ti} means the transverse momentum of particle i with respect to the beam direction z and the expression $\Delta R_{ij}^2 = (y_i - y_j)^2 + (\Phi_i - \Phi_j)^2$, where $y_i = \frac{1}{2}\ln\frac{E_i+p_{zi}}{E_i-p_{zi}}$ denotes the i's rapidity and Φ_i constituted the azimuth. The jet radius R is a parameter of the algorithm. This method contains a distance measure between each of particles i and the beam:

$$d_{iB} = p_{ti}^2.$$

The first kind of this method was the exclusive variation of the k_t algorithm [13], where the consideration of smallest d_{ij} and d_{iB} were introduced.

If it is a d_{ij}, we can replace i and j together with a single object which has a momentum $p_i + p_j$. This object is a pseudojet, this is neither a particle, nor a full jet.

If the smallest distance is a d_{iB}, then we take away i from the list of particles and pseudojets than we add it to the beam jet. This method is repeated until the smallest d_{ij} or d_{iB} reaches the threshold d_{cut}. All particles and pseudojets are processed.

In the case of the inclusive variation of the k_t algorithm [14] the distances d_{ij} and d_{iB} are the same as in the exclusive method.

The difference is between them when we determine the smallest value d_{iB}, then the object i is removed from the list of particle and pseudojet set and it is added to the list of final inclusive jets. There is no threshold d_{cut} and the clustering process is kept until particle or pseudojets remain.

Because the distance measures are the same in both of the inclusive and exclusive algorithms, the clustering sequence is same in these processes.

We consider the longitudinally-invariant k_t algorithm for hadron collisions [13, 14]. It was the first jet algorithm to be implemented in FastJet [9].

Longitudinally invariant k_t jet algorithm This jet method applies the inclusive version [14]. The steps of algorithm are written as following:

1. For each pair of particles i, j determine the k_t distance to use equation (1). with $\Delta R_{ij}^2 = (y_i - y_j)^2 + (\Phi_i - \Phi_j)^2$, where p_i, y_i and Φ_i are transverse

momentum (with respect to the beam direction), rapidity and azimuth of particle i. R is a jet-radius parameter which is taken of order 1. For each parton i we calculate the beam distance $d_{iB} = p_{ti}^2$.

2. We find the minimum d_{min} of all the d_{ij}, d_{iB}. If d_{min} is a d_{ij} merge particles i and j into a single particle, then we sum their four-momenta. If it is a d_{iB} then declare particle i which is a final jet and remove it from the list.

3. Repeat from step 1 until no particle are left.

The exclusive version of the longitudinally invariant k_t jet algorithm [14] is similar, except two cases:

i. a d_{iB} is the smallest value, that particle becomes part of the beam jet.

ii. The clustering is stopped when all d_{ij} and d_{iB} are larger than the value d_{cut}.

In the next section we study the parallelization of the cluster k_t algorithm. We apply the simulation of the CERN Alice experiment offline method.

4 FastJet clustering

To do the jet clustering on the detected points, the FastJet [9] package is used. It is implemented in C++ and provides many different jet finding algorithms and analysis tools. The user can select from the widely used sequential recombination algorithms, that are implemented efficiently, while it also supports plugins for other solutions. The initial motivation to use this toolkit is the inclusion of it in the Alice Offline Framework, so after the simulation of the detector, the further process of the resulting particles can generate the jet structures.

4.1 AliRoot

AliRoot is the Off-line framework for simulation, reconstruction and analysis of the ALICE experiment at CERN. The framework and all applications are developed based on the ROOT system. Mostly it is based on Object Oriented programming paradigm, but as it was developed since 1998 it has some legacy code and other libraries, that were developed based on different principles.

The simulation part of the tool covers all processes of primary collisions and it generates the newly created particles, follows through their transportation in the detector, calculates the hits in each component. The result can be either stored in so called summable digits or digits derived from the summable ones and it can also create raw data.

For the work presented in this paper the system's TPC detected points were used. The Time Projection Chamber (TPC) detector is the main tracking component of ALICE. Particles passing through this detector ionizes the gas molecules inside the TPC and these ionization points are registered. The TPC detector consists of two cylindrical volumes sitting along the beam. These volumes are split into 18 trapezoidal readout sectors. The detector measures track positions on 159 rows [2].

4.2 Parallelizing the clustering

The goal is to not rewrite the structure of the toolkit, only include the necessary parts to utilize the parallel processing capabilities of the CPU. The used N2 clustering - which requires $O(N^2)$ operations - is considerably slower than the more optimized versions of the tile based clustering methods, but applying SIMD techniques can show rapidly how much we can gain on this field from utilizing multiple cores [17]. Even if parallelism is used, the current algorithm can lead to very inefficient solutions if not done right. It is necessary to check each element N times, which naively may result in the generation of new threads for every single particle. Even if the maximum number of threads supported by the hardware is taken into account, the overhead of creating just a few threads in each iteration results in significant bottlenecks. Thus, for such applications a generally good idea is to implement a *Thread Pool* that will create the maximum number of threads only once and keep them alive until there is any future work to do. In this initial work parallelization is done on the distance update of the newly created jets after each recombination step. This requires the new jet to being compared with other existing jets and find the nearest neighbor and to set it's distance. In case a closer jet is found the process becomes sequential on the the assignment of the new closest neighbor (Subsection 3.2.1).

4.3 Implementation

An important part of the implementation is the presence of the *Thread Pool*. The threading uses the elements of the C++11 standard, namely the pool depends on *conditional variables*, *mutexes* and *locks*. It provides two queues for storing the incoming callable functions and the input data. The required argument of the function implemented for the parallel computation is a user defined *JetData* typed parameter. This will contain the necessary values for the computation of clusters. After the pool's creation, the initialized threads

are waiting on a conditional variable until some work is presented to the FIFO. To push work into the queues and to retrieve them from there, a *unique lock* is ensuring, that only one thread can reach the container at any given time. After a job is pushed in or popped out by a worker, the lock gets released and some other threads can reach the additional if any tasks. The workers are notified through the conditional variable if they have any processing to do. After pushing in the jobs, the pool will know how much (*numberOfTasks*) work there is to do. This information is used for waiting until all the tasks are finished. This is followed by an atomic counter *done*, that is increased each time a task finishes it's work. The main thread of the application will wait on a conditional variable (*exitCondition*) of the pool, dedicated to signal the conclusion of all the processing, that have been assigned before the last *waitKernel* call. After all the work is done and the waiting is over the counters are reset to 0. At the program's end, all containers are destroyed and the threads are joined.

The function responsible for the processing of the work needs to be able to access the internal functions required for the jet clustering, so it was implemented in the *ClusterSequence class* as an additional member function. It requires a template parameter, that will tell what is the jet definition used by FastJet, when the clustering itself was instantiated.

The clusterization is done through the ClusterSequence's *_simple_N2_cluster* function, as such, this is the only routine from FastJet, that is changed. These modifications apply the currently available parallelism, creating the thread pool, preparing the work for the threads and waiting for them to be finished before moving onto the recombination step of the next jet. For the input data of the workers an evenly chunked subset of the jets are used. Because the jets are not changed, except the current one, it will not create additional race condition among the threads. The presented implementation has one part only where concurrency applies, when the new nearest neighbor is set. To prevent issues from this a *unique lock* protects the assignment of the new element.

4.3.1 Algorithm

The algorithm of the thread pool's enqueue is presented in Figure 2. The input parameter *fn* is a function pointer to the implemented worker callable *InitJetBPool* and data is a *JetData* type pointer to the input of the actual task.

The function handling the wait for finishing all running tasks is shown in Figure 3.

How the worker threads are retrieving their task and processing it is shown

```
1: procedure ENQUEUE(fn, data)
2:     lock queue_mutex
3:     tasks.push_back(fn)
4:     datas.push_back(data)
5:     ++numberOfTasks
6:     unlock queue_mutex
7: end procedure
```

Figure 2: The Enqueue function of the Thread Pool

```
1: procedure WAITALL
2:     lock wait_mutex
3:     if done != numberOfTasks then
4:         exitCondition.wait(wait_mutex)
5:         done ← 0
6:         numberOfTasks ← 0
7:     else
8:         done ← 0
9:         numberOfTasks ← 0
10:    end if
11: end procedure
```

Figure 3: The *WaitAll* function of the Thread Pool

in Figure 4. As this is the most time consuming kernel, it incorporates the shared memory to compute the triplets as fast as possible.

The routine responsible for the nearest neighbor check is described in Figure 5. The required parameters are the last jet (jetA), the new jet after jet-jet recombination (jetB), the table containing the distance between two jets (diJ), the pointer to the first element of the list containing the jets (head) and the pointer to the last element (tail). The *NN* member of jetI is the nearest neighbor of jetI, while *NN_dist* is the distance between the two. The initial NN_dist is set to R^2, where in this case R, the jet-radius parameter is set to 0.2. The parallel version of *InitJetB* also two more parameters to know which interval a specific thread needs to work on.

```
1: procedure WORKER
2:     lock queue_mutex
3:     task ← tasks.front()
4:     data ← datas.front()
5:     tasks.pop_front()
6:     datas.pop_front()
7:     unlock queue_mutex
8:     TASK(data)
9:     ++done
10:    if done == numberOfTasks then
11:        exitCondition.notify_one()
12:    end if
13: end procedure
```

Figure 4: The worker retrieving a task with it's argument and processing it

```
1: procedure InitJetB(jetA, jetB, diJ, head, tail)
2:     for jetI in head..tail do
3:         if jetI→NN == jetA or jetI→NN == jetB then
4:             find nearest neighbor for jetI
5:         end if
6:         if jetB != NULL and jetI != jetB then
7:             if distance(jetI, jetB) < jetI→NN_dist then
8:                 jetI→NN_dist = distance(jetI, jetB)
9:                 jetI→NN = jetB
10:                Update diJ accordingly
11:            end if
12:            if distance(jetI, jetB) < jetB→NN_dist then
13:                jetB→NN_dist = distance(jetI, jetB)
14:                jetB→NN = jetI
15:            end if
16:        end if
17:        if jetI→NN == tail then
18:            jetI→NN = jetA
19:        end if
20:    end for
21: end procedure
```

Figure 5: Nearest neighbor calculation after a recombination step

The part dispatching the parallel work is described in Figure 6. The required parameter is *maxThread* that tells how many concurrent threads can run on the given processor.

```
1:  procedure DispatchToPool(maxThread)
2:      JetData data[maxThread]
3:      for i in 0..maxThread do
4:          data[i].begin ← beginning of the i^th chunk of the jet list
5:          data[i].end ← end of the i^th chunk of the jet list
6:          data[i].jetA ← jetA
7:          data[i].jetB ← jetB
8:          data[i].diJ ← diJ
9:          data[i].head ← head
10:         data[i].tail ← tail
11:         ThreadPool.enqueue(InitJetBPool, data[i])
12:     end for
13:     ThreadPool.WaitAll()
14: end procedure
```

Figure 6: Dispatching work to the thread pool and synchronization at the end

4.4 Results

The complexity of the used algorithm (Subsection 4.3.1) is $O(N^2)$, which requires a high amount of computation to be done. In such case the usage of parallel computing can reduce the overall runtime. In this subsection the resulting performance increase is discussed and how the implemented thread pool helps keeping the thread creation overhead down. All tests were running on the same environment detailed in Table 1. The performance evaluation and comparison was done using the raw data from an event simulated with the AliRoot (Subsection 4.1) framework's PbPbbench test application. The detected points were directly sent to the modified FastJet routine. The generated event used for the tests contained 140535 elements.

The system used for development and testing is described in Table 1.

While applying parallelism (Subsection 4.2) on a given problem can greatly increase the performance, it also generates some overhead by creating additional threads. Comparing a parallel implementation (Subsection 4.3) with constant thread creation and another with the thread pool enabled, the threading performance of the two is far from each other. Also while using multiple threads the operating system needs to do context switching to let a specific

work be done. Figure 7 shows the runtime of the two different approach.

CPU	#Thread	OS	Compiler
Intel Core i7 4710HQ	8	Windows 10 Pro	Visual C++ 2013

Table 1: The test system

The result shows, that the naive threading based implementation took $271,31$ seconds to finish, while with the thread pool the runtime was only $207,9$ seconds, leading to a 1.3 times better performance.

Figure 8 shows the runtimes of the parallel thread pool based implementation with the original sequential clustering.

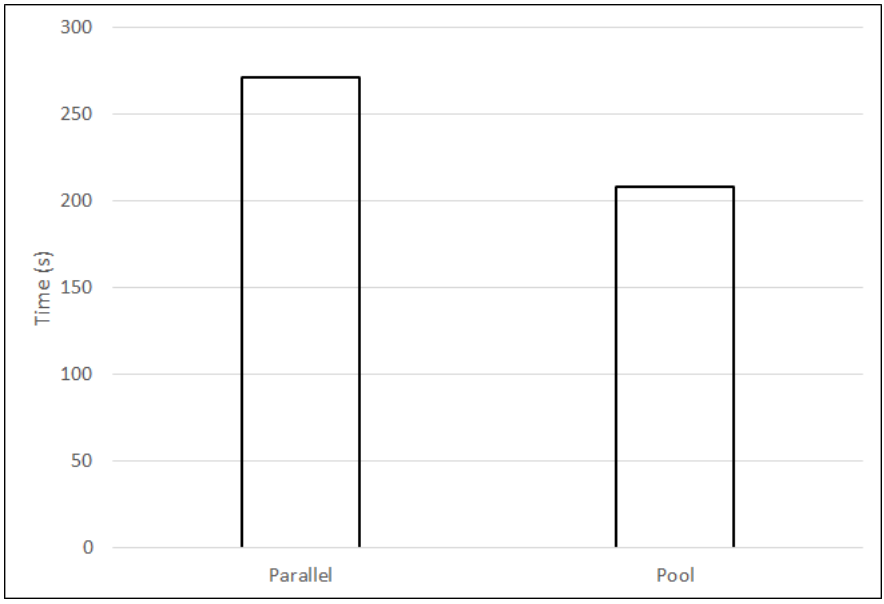

Figure 7: Runtime using naive threading or a thread pool

The parallel implementation taking $207,9$ seconds is 1.67 times faster compared to the original sequential solution's $347,18$ second long run.

Figure 9 shows the full time of the recombination loop. Comparing the parallel implementation with the performance of the optimized tile based clustering, the difference is still big.

The runtime of the parallel $O(N^2)$ algorithm is $234,52$ seconds, the tile

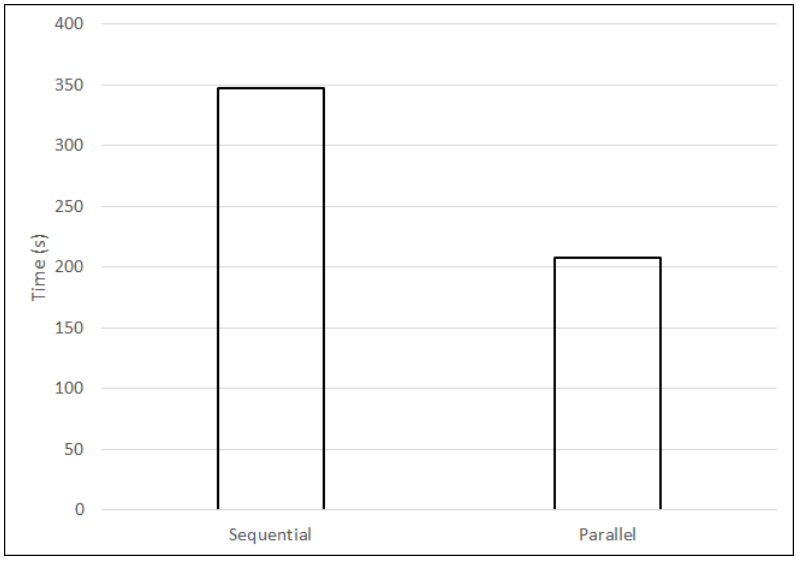

Figure 8: Runtime of the parallel $O(N^2)$ algorithm and the sequential clustering

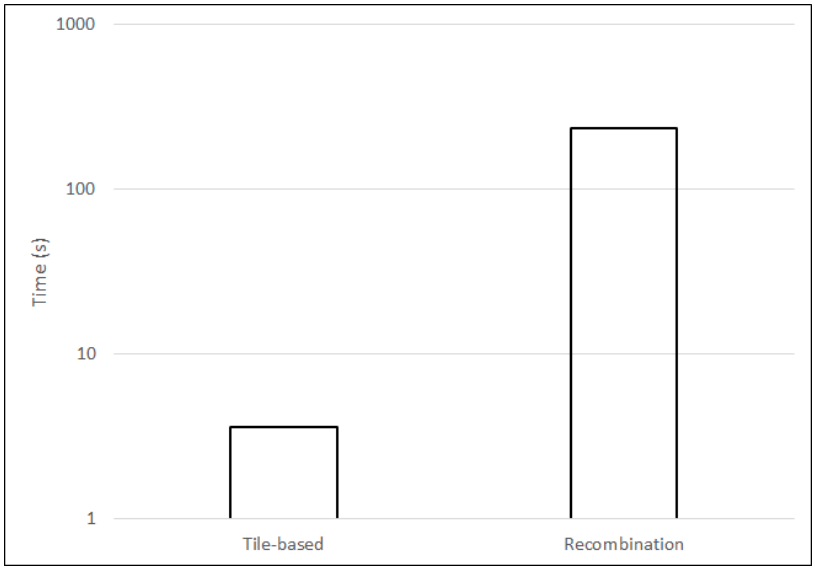

Figure 9: Runtime of the parallel $O(N^2)$ algorithm and tile based clustering

based clustering takes only $3,58$ seconds. This shows the optimized tile based clustering of the FastJet toolkit is 65.5 times faster compared to the proposed parallel method. As more work is still needs to be done on the parallelization this number can be greatly reduced giving the possibility to the parallelized algorithm to be even faster compared to the tile based solution.

5 Summary

Applying parallelization is the mean to utilize all the available processor resources independent from the given algorithm. Even if the complexity is $O(N^2)$, the performance increase is valuable. It was shown, by using a naive parallelization approach the resulting algorithm might perform better in comparison to it's sequential implementation, but the generated overhead will neglect it's positive effect. Thus by applying a thread pool on the overall system and generating the worker threads only once at the start of the application the overall runtime can be decreased considerably. Comparing the proposed parallel method to an already optimized, yet not multi-threaded, tile based clustering method also implemented in the FastJet toolkit (Section 4), the proposed algorithm still shows much slower runtime because of it's $O(N^2)$ nature.

6 Future work

To further increase the performance of the algorithm and even if not to make an $O(N^2)$ clustering faster than a tile based one, additional techniques should be explored for further optimizations and performance gain. Modern CPUs have some form of vectorization support for multiple generations now that can further speed up the evaluation of the different algorithms. This requires the data to be restructured to conform the requirements of the vectorization.

By using many-core architectures, like GPUs, it is possible to achieve full parallelization [11, 12], meaning to do all available computation in parallel. The downfall of this approach might come from the increasing amount of required memory. To fully parallelize an $O(N^2)$ algorithm, it will take N^2 memory too, which leads to impossible requirements fairly soon. It needs to be explored where the balance is and where the limit should be drawn between runtime and resource requirement to be able to run the application much faster, without needing insane amount of memory.

Furthermore it is important to not just increase the performance of the selected algorithm, but to apply the techniques and conclusions shown in this paper on other already optimized routines, to see how the overall jet clustering can benefit from parallelization.

References

[1] A. Ali, G. Kramer, Jets and QCD: A historical review of the discovery of the quark and gluon jets and its impact on QCD, *Eur. Phys. J.* H**36** (2011) 245–326. arXiv:1012.2288 [hep-ph]. ⇒51

[2] D. Rohr, S. Gorbunov, A. Szostak, M. Kretz, T. Kollegger, T. Breitner, T. Alt, ALICE HLT TPC tracking of Pb-Pb events on GPUs, *Journal of Physics: Conference Series* 396 (2012), doi:10.1088/1742-6596/396/1/012044 ⇒56

[3] G. P. Salam, Towards jetography *Eur. Phys. J.* C**67** (2010) 637-686 arXiv:0906.1833 [hep-ph]. ⇒51

[4] G. Sterman and S. Weinberg, Jets from quantum chromodynamics, *Phys. Rev. Lett.* **39** (1977) 1436. ⇒51

[5] M. E. Peskin, D. V. Schroeder, *Quantum Field Theory*, Westview Press, 1995. ⇒50

[6] T. Muta, *Foundation of Quantum Chrodinamics*, World Scientific Press 1986. ⇒50

[7] M.G. Bowler, *Femptophysics*, Pergamon Press 1990. ⇒50

[8] S. Salur, Full jet reconstruction in heavy ion collisions, *Nuclear Physics A* 830 (1-4) (2009)139c–146c. ⇒50

[9] M. Cacciari, G. P. Salam, G. Soyez, FastJet user manual, *Eur. Phys. J. C* **72** (2012) 1896 arXiv:1111.6097v1. ⇒54, 55

[10] R. Atkin, Review of jet reconstruction algorithms, *Journ. of Phys.: Conf. Ser.* **645**(2015) 012008. ⇒53

[11] R. Forster, A. Fülöp, Yang-Mills lattice on CUDA, *Acta Univ. Sapientiae, Informatica*, **5**, 2 (2013) 184–211. ⇒63

[12] R. Forster, A. Fülöp, Jet browser model accelerated by GPUs, *Acta Univ. Sapientiae Informatica* **8** 2 (2016) 171–185. ⇒63

[13] S. Carani, Yu.L Dokshitzer, M.H. Seymour, B.R. Webher, Longitudinally-invariant $k_\perp$-clustering algorithms for hadron-hadron collisions, *Nuclear Physics B* **406** (1993) 187–224. ⇒54

[14] S.D. Ellis, D. E. Soper, Successive combination jet algorithm for hadron collisions,*Phys. Rev. D* **48** 7 (1993) 3160. ⇒54, 55

[15] S. D. Ellis, J. Huston, K. Hatakeyama, P. Loch, M. Tonnesmann, Jets in Hadron-Hadron Collisions *Prog. Part. Nucl. Phys.* **60** (2008) 484 arXiv:0712.2447 [hep-ph]. ⇒51

[16] S. Moretti, L. Lönnblad and T. Sjöstrand, New and Old Jet Clustering Algorithms for Electron-Positron Events *JHEP* **9808** (1998) 001 arXiv:hep-ph/9804296. ⇒51

[17] Technology Insight: *Intel Next Generation Microarchitecture Code Name Haswell*, IDF2012. ⇒56

Received: April 3, 2017 • Revised: June 30, 2017