Multinode Approach for Map Data Processing

Vít Ptošek, Kateřina Slaninová

IT4Innovations National Supercomputing Center,
VŠB - Technical University of Ostrava,
Czech Republic
(vit.ptosek@vsb.cz, katerina.slaninova@vsb.cz)

Abstract. OpenStreetMap (OSM) is a popular collaborative open source project that offers free editable map across the whole world.
However, this data often needs a further on-purpose processing to become utmost valuable information to work with. That is why the main motivation of this paper is to propose a design for big data processing along with data mining leading to the obtaining of statistics with a focus on detail of a traffic data as a result in order to create graphs representing a road network. To ensure our High Performance Computing (HPC) platform routing algorithms work correctly, it is absolutely essential to prepare OSM data to be useful and applicable for above-mentioned graph, and to store this persistent data in both spatial database and HDF5 format.

Keywords: OpenStreetMap, Road Network Quality, Big Data Parsing, Multinode Processing, ETL, State Machine, Pipeline

1 Introduction

One of the biggest advantages of OSM[3] is such that everyone can contribute[13]. There is no doubt this factor has helped the project to be successful to the possible extent[16]. Nevertheless, this feature tends to produce a lot of misleading if not missing data. Our objective is to present an approach for processing such a big spatial data to be of better quality.

The main motivation of this paper is encouraged by Antarex[1] project's second use case - a self-adaptive server-side navigation system to be used in smart cities. Such a navigation system surely needs to be backed by a reliable graph-like data for a static routing as well as derived information helping with decisions and planning in a dynamic routing. The more data we are able to process and benefit from, the better we can successfully reach this goal and provide such a service. Our interest is to take advantage of large datasets, as we aim to pay attention to detail on complete road network, which inevitably brings big data-related tasks and touches usage of parallel algorithms.

This whole data-driven process is supposed to be easily configurable, reusable and efficient. That has been achieved by two mutually communicating services working as process chains as seen in Figure 1 below.

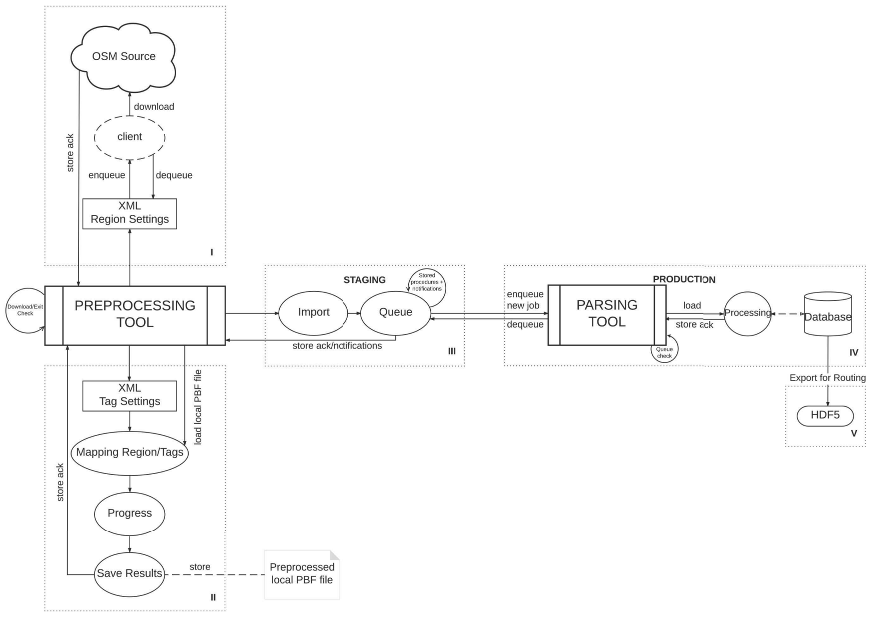


Fig. 1. Processing design scheme

The first one is meant to obtain and decompress input raw data. Then we need to filter out desired preprocessed output which from our point of view makes sense to keep as an input for the very next process called parsing and post-processing. In the following step, a job being part of a pipeline is enqueued and kept track of. At this point, a chosen custom metrics are calculated, verified and stored.

Having defined a criterion for a road map data assessment, we have to complete graph with respect to information we need at the final stage of processing. As a result, the network in question can serve for example the routing purposes and traffic data overview.

The rest of the paper is organized as follows. The relevant work in Section 2. Exploited data is described in Section 3 for better understanding of design clarified in Section 4 and ETL[1] logic in Sections 4.1, 4.2 and 4.3. Results and comparisons are explained in Section 5. Conclusions and future work are to be found in Section 6.

[1] Extract, transform, load process

2 Related Work

Some research has been done in field of assessing OSM data quality to show how much trusted such can be[12]. There are couple of methods applicable. Some of them use ground-truth data, determining semantic data quality by co-occuring tags[10]. Another interesing approach rests on machine learning. In that case, unlike ground-truth data, geometrical (lenght, linearity) and topological (connectivity) characteristics based on segment collocation[14] are highlighted.

Either one proves that there really is inconsistency of tags. Also incorrect or even missing keys (described further in Section 3.1) are to be found in original OSM datasets. Looking into our samples, we can confirm that way elements may have little or no information related to road network as well as they might be suspiciously wrong. In spite of this fact, we have decided that there is strong need to complete and normalize this data. Once this is done and validated, we try to calculate the rest with maximal possible accuracy and minimal risk. That being said we came with domain-specific rule sets to be applied. Those are combined with our beforehand approved results that turned out to be correct and reliable. We use semantics to detect related tags and its alternatives if such exist. Geometrical characteristics like number of nodes can give us an idea of road classification. Containing any value we normalize it (unit, type, access, formatting, language), in the opposite we flag them as needing calculations and do so by implication depending on known facts and our previous observations.

Another topic related to this paper introduces an idea of incremental processing chain[11] using VGI[2] data and Osmosis[4]. The described work flow finishes initial processing of whole world OSM data (by then of size as Europe in present) in 8 hours, whereas the update takes around 3 hours daily due to search operations. In our case a custom versioning was designed to avoid these computations. Computing itself was desired to focus on high performance workload to be able to complete several times a day for each country alone.

Finally, a similarly targeted publication discusses routing with OSM data and dealing with its preprocessing [15] for shortest path both on-line and off-line using different routing profiles. Albeit this paper offers a valuable contribution, our intention is to only concentrate on server-side routing using computing power to bring some extra navigation use-cases, like emergency or public service routing. To speed routing on the data side, we can generate graph specialized for given task to minimize unnecessary nodes to visit. This means we can process routing graphs for bicycle and car navigation at the same time, each for different kind of work, and decrease routing cost regardless algorithm for finding optimal path used.

[2] Volunteered Geographic Information

3 Data

The data to be processed are obtained from Geofabrik[2] download server and contains OSM dataset under Open Database License 1.0. The raw data is organised by regions and refreshed on daily basis.

In spite of the fact that everybody can contribute and propagate map data to the source being used, there is always room for inaccurate and flawed or obsolete information we cannot afford to use. Another thing is duplicity or unnecessary data. There have been some limitations experienced with downloading also.

Due to big amount and quality of above mentioned data, a reduction based on custom settings and sanity check rules has to be made and applied. Detection of missing or broken data is of high priority as some of them can be (re)calculated for use of ours. On the other hand, the rest may be omitted or skipped and discarded.

3.1 Input

Input data file is compressed in Protocolbuffer Binary Format (PBF)[7], which is Google's data interchange format. This data consists of following:

- **Elements** - the basic components
  - **Nodes** - point on surface defined by latitude and longitude
  - **Ways** - ordered list between nodes that defines polyline (roads in this case)
  - **Relations** - data structure documenting relation between elements
- **Tags** - key-value pair belonging to an data element representing its information

3.2 Output

There are several outputs on the way to the final one. Both preprocessed and parsed file are stored in binary formats for which PBF and HDF5[9] have been found suitable. The final results are also stored in spatial database[6] as graph of complete coverage of custom roads, segments and additional metrics and information.

We are only interested in roads that we calculated as accessible by car. For such roads we gather information like whether it is one way, toll way or bridge, what classification it belongs to, what is its maximal allowed speed, etc. For every road we have 22 tags we are able to parse or calculate values for. We add 3 more custom tags for our use only that give us extra information for routing.

4 Design

The whole process scheme is divided into several parts (see Figure 2) and each one of them solves separate set of tasks based on its order, type and logic. Two main parts of a single whole are Preprocessing Tool (Section 4.1) and Parser (Section 4.2). These are further consisting of sub-processes and sub-sections. Our idea is to run described architecture as a remote service under various parameters due to scalability as described in Section 5.1.

The solution as described had been implemented from scratch for both Windows stack and Linux cluster deployment. At this point of a time it is optimized on several levels and able to run on both platforms with the similar if not same features like shut-down signal handling and shell commands execution, just to name a few. The runtime environment comparison is to be seen in Section 5.2.

To fully exploit multi-node architecture, besides of a multi-thread, concurrent and parallel processes, certain changes have been made to provide with optional dynamic load balancing between instances reflecting the cluster PBS[5] job scripts where possible. Also some parts, like database import explained in Section 4.3, can be run in the background.

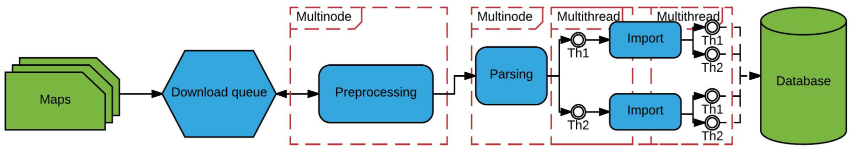


Fig. 2. Multinode and parallelism role in scheme

4.1 Preprocessing

The very beginning of the data processing starts with loading settings for maps and tags we are interested in. The behaviour of a run depends on switch options. Preprocessing tool manages the queue and pipeline as a control element and it is designed as a remote data-pump checking a state of a job. This means that every map we want from download queue goes through an implemented chain starting here and only picks the data we need to be saved. It looks after communication, importing borders and new jobs to start on as well as can be seen in Figure 1.

State Machine Preprocessing tool is realized as a state machine (Figure 3) which allows to run in loops based on a phase it is in. The benefit of this is that we don't need to restart process because of modifications made from the outside as the whole run can be forced remotely on the fly due to the updated pipeline it keeps track of.

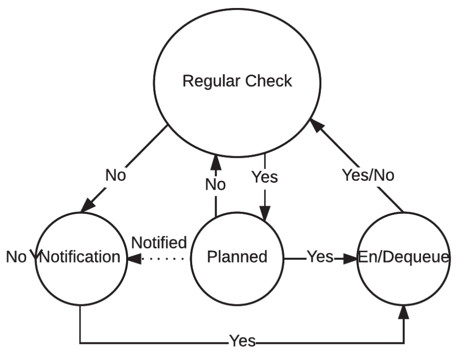


Fig. 3. Processing represented as a state machine

Pipeline Every area being processed has its own separate pipeline in a queue (see Table 1) saying what process is responsible for a certain planned task and which one is following. There is a given order that has to be followed, although different maps can be processed at the same time, only time-shifted.

The steps of a pipeline are matter of customization and serve to speed up the process as the first map can be parsed meanwhile the second one is being preprocessed for another parser.

source	target	task	info	country	done
stored_function	preprocessing	regular_check	<switch parameters>		t
preprocessing	preprocessing	downloading_map	<map PBF file>-<size>MB	<name>	t
preprocessing	preprocessing	extracting	<map PBF file>	<name>	t
preprocessing	parsing	parse_pbf	<storage path>	<name>	t
preprocessing	preprocessing	import	<boundary file>>><db>	<name>	t
parsing	osm2pgsql	export_country	async	<name>	t
parsing	stored_procedure	boundary_cut	staging-asynchronous	<name>	t
parsing	stored_procedure	deploy_to_production	production-synchronous	<name>	t
parsing	stored_procedure	reprocess_changes	production-synchronous	<name>	t
preprocessing	log	exit	<planned/unplanned>		t
preprocessing	log	exit	<cluster node>|<reason>		t
parsing	log	exit	<cluster node>|<reason>		t

Table 1. Example of a pipeline queue

Planning and Reprocessing Since the pipeline queue is available for use, it can be taken advantage of in the planning-ahead manner. It also serves its purpose when it comes to abortion of a task, because the process(es) in charge of a cancelled one can easily catch up again and continue from when it was.

Instead of going through the complete pipeline from the very beginning, starting over and redoing what has been done already, a reprocessing point is detected for a new start as illustrated in Figure 4.

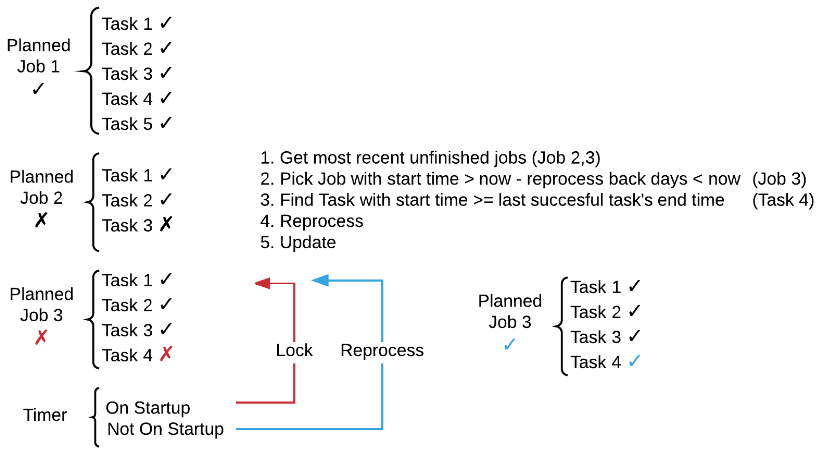


Fig. 4. Logic of pipeline reprocessing when enabled

4.2 Parsing

Parsing instance always follows a leading Preprocessing one and does what it is requested for and lets the Preprocessor know about the progress of a given task. It reminds of master-slave approach based on ETL.

The objective of parsing is to take the data that hasn't been ruled out (and thus found needful) and to only keep nodes already belonging to the preprocessed roads, which also helps with a compression. The ways are supposed to represent road network for routing. The better information we have about the roads, the more valuable the graph for navigation and visualization would be.

Some data can be extracted from datasets and cleaned afterwards, but most of them can be missing or misleading, so Parser uses our pre-defined set of rules to (re)calculate metrics and gather basic and additional information of a road and then proceeds to managed indexing of way and node elements of a currently processed country into the road network.

Given as example, that we know some road being currently processed is situated somewhere in the Czech Republic and only comes with tags *lanes:3, oneway:yes*. We can deduct missing classification tag which (now) would be *highway:motorway*. Since we also know there is no restriction, we can lookup an upper speed limit of a mentioned country for a specific type of highway, and thus can also add *maxspeed:130* and most likely *toll:no*, otherwise the toll would be stated.

Parser can run under several settings and in one of two checking modes

1. Continual - on standby waiting for notification scheduling task
2. Periodical - self-scheduling future tasks and waiting for their time to come

Queueing As mentioned in Section 4.1, the continuous process leans on a pipeline queue. That brings enqueuing and dequeuing logic into a question. It is also a special case of a log.

In case of more than one instances, it is extremely important to ensure that concurrent ones are not stealing jobs from each other. Not only it would be ineffective to work on the same thing several times, but it could lead to a damaged untrustworthy data. The same stands for the real opposite where some task would not be picked by neither one of a running workers at all. For these cases it is necessary to use locks, double-checking, and sometimes timers in a custom specially designed structures for handling this.

Queueing, forcing start and stop as well as reprocessing and reloading parameters and settings can be done remotely via database commands and notifications.

Import Importing processed data is multi-thread asynchronous awaitable process that carries out preparing and moving data into a staging spatial database. Once this is done upon request from pipeline, everything is ready for the next step completely being executed in a database which is a phase of preparing data for a production.

4.3 Database Layer

Stating an obvious, a fair amount of work is done within a database layer because of geometry data type values stored in spatial structures. In our case, it also brings advantages that stored procedures give to us. That is why it is so important to pay attention to detail, especially when talking about indexing, prepared statements, query optimization; last but not least, significant communications happen here.

Spatial Data It is not only crucial to have information about nodes and ways as they go and are related to each other, but also to be able to represent them in a graphical way - to visualise. For querying geometry data it is unnecessary to support spatial database so we can link GPS point to a specific road, tell whether some bridge overlaps any highway and many more.

Automatisation and Asynchronous Awaitable Functions In regards to database stored procedures and spatial approach, the process running in the DBMS combines both of them. When some country is successfully stored in preliminary tables, there always are some roads crossing the neighbour country border (Figure 6). The correct way is to only store the part belonging to a specific country before generating the final graph for navigation routing, so those lines are cut on a border line having the same node ID on both sides down to logically linking them back together on a precisely measured spot.

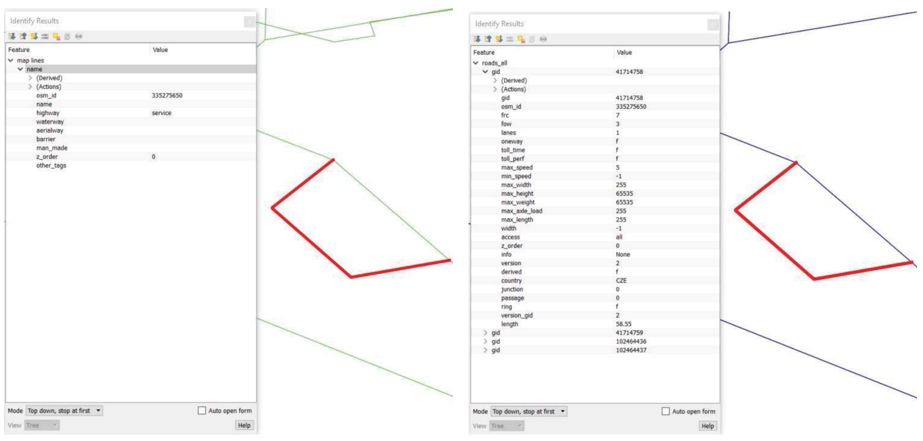
This is however quite costly and there is no need for any application calling this function to be blocked by waiting for a specific result. One way how this is handled is that the stored functions call each other in a specified order. The better one is that the first one called is run asynchronously in the pipeline and the Parser gets back to it once the database gives the result. Meanwhile, it takes another tasks form the queue available. We can manage to split whatever area we desire in that manner and apply to divide-and-conquer technique.

Audit and Versioning Every task is logged on a specific level, but there is one special case when we want to log custom changes so we can take them into account for the next-time processing. The bright side of this takes place in incremental reprocessing, history view, no redundancy and taking no risk in overwriting changes with someone else's as we prefer ours as long as they are up to date.

When some change is propagated into the downloaded dataset we use, its element version changes and so it is important to us to keep our changesets and versions too. Every country table with the road network has its very own audit table rolled out dynamically and thanks to that we can maintain the most recent and actual versions and also recover from our backups.

5 Results

Data Processing Tool is used at least once a month automatically to mainly catch up with major updates, data enrichment (Figure 5) and graph readjustment (Figure 6). In spite of that, it can be run whenever needed.



Feature	Value
map lines	
name	
(Derived)	
(Actions)	
osm_id	335275650
name	
highway	service
waterway	
aerialway	
barrier	
man_made	
z_order	0
other_tags	

Feature	Value
roads_all	
gid	41714758
(Derived)	
(Actions)	
gid	41714758
osm_id	335275650
frc	7
fow	3
lanes	1
oneway	f
toll_time	f
toll_perf	f
max_speed	5
min_speed	-1
max_width	255
max_height	65535
max_weight	65535
max_axle_load	255
max_length	255
width	-1
access	all
z_order	0
info	None
version	2
derived	f
country	CZE
junction	0
passage	0
ring	f
version_gid	2
length	58.55
gid	41714759
gid	102464436
gid	102464437

(a) OSM Original with poor tagging (b) OSM Processed with extra information

Fig. 5. OSM road prior and after processing

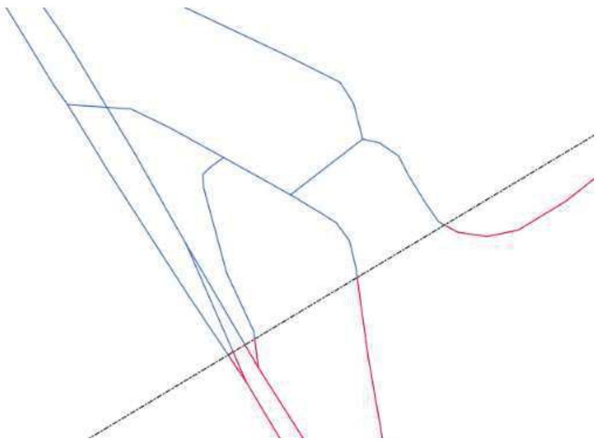


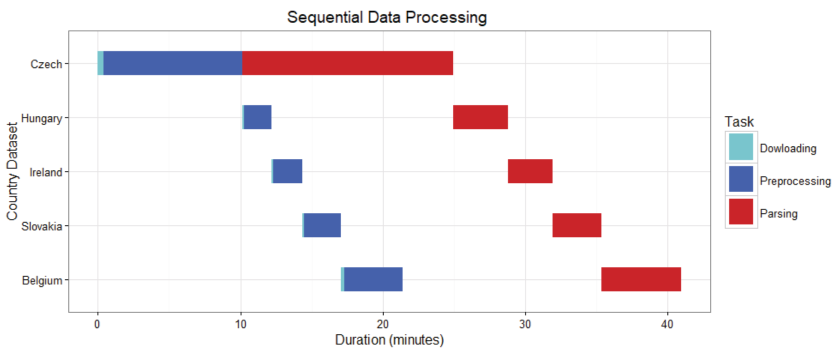
Fig. 6. Graph split on border

5.1 Single-node versus Multi-node

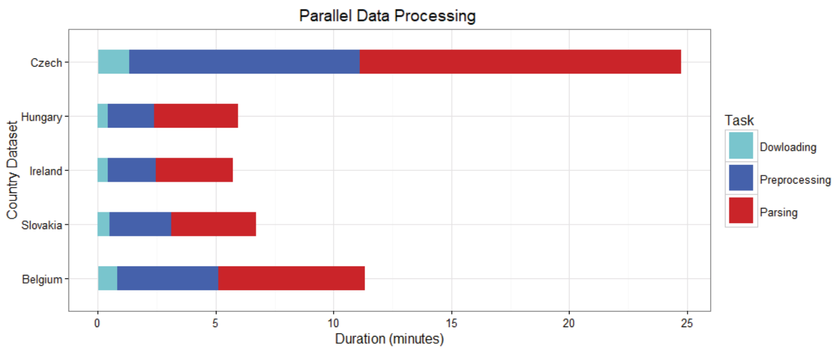
It is only needed to have one instance of each part for complete processing in a single mode, but our observations based on measurements showed us to launch three Parsers for one Preprocessor when considering more than one instance. It is very important to take into account the computational efficiency in case of processing difference as well as time shifting - especially in tasks involving big data as large maps are.

To keep all the parts fully utilized, a lightweight resolver has been implemented so the download queue is split between all the active Preprocessors from the map list. Every one of them then tells the first free Parser what to do regardless the node they run on as the pipeline is shared. This helped us to accelerated by ≈ 55 % as seen in the Figure 7.

If the processing is found to be short on resources available, it is possible to add up workers on the fly to finish sooner scaling both up and out without interrupting the ongoing job.



(a) Sequential pipeline



(b) Parallel pipeline

Fig. 7. Pipeline speed-up

5.2 Runtime Environment Comparison

There are measured results representing duration of complete process imaged in Figure 8. This process consists of scheduled sub-processes and tasks shown previously in Table 1. In this case a map of France has been chosen for benchmarking as it is the largest available European dataset which makes it suitable because bigger data to process helps us express diversity of taken execution time better as the difference rise along. The binary file sizes of used map varying during the processing lifetime along with count of nodes and vertices are exposed in the Table 2.

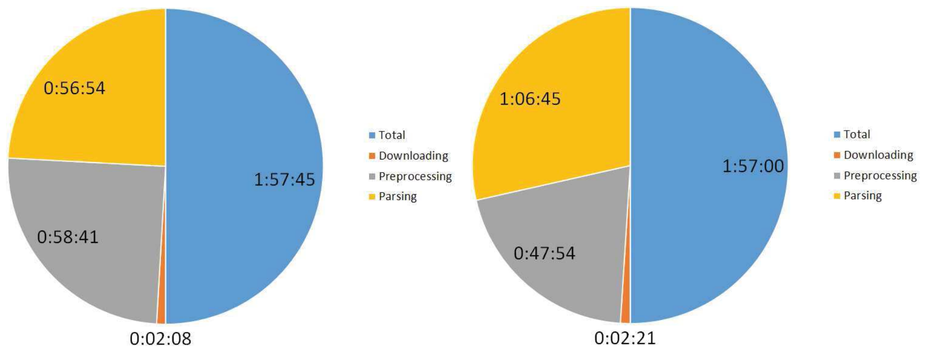
By complete process we mean both preprocessing and parsing part, as seen in Figure 1. In both cases, the given testing scenario used the same pipeline without the interruption. Although the total duration almost equals, the time ratio of these parts differs obviously. The first part is CPU-bound whilst the latter is memory-bound. We also need to take into account an IO overhead and network speed causing some divergences especially when talking about virtualization.

State of Dataset File	PBF Size (∼GB)	Edges/Vertices (∼mil))
Raw downloaded	3.65	5.4/379.2
Parsed roads	2.83	5.4/50.2
Extracted and processed graph	17.29	5.2/50.2
Indexed final graph	2.03	5.0/55.9

Table 2. Map set of France during processing

These results have been acquired on Salomon[8] supercomputer running single computing node using 24 CPU cores and 128 GB of RAM with disabled Hyper-threading. Data has been stored on a hard drive in the Lustre file system.

The aim of this experiment was to prove that the whole process is feasible to run regardless to operation system. The tested tool provides repeatedly exactly the same usable results in very comparable processing times and manner.



(a) Windows Server 2016 - KVM (b) Linux CentOS 6.8 - Mono

Fig. 8. Task execution time comparison - same dataset under different environments

These results were also correct in both cases and observations. This means that it is possible to combine different parts on various platforms and run in parallel.

6 Conclusion and Future Work

As the volume growth of big data for smart cities rises along with its need, the processing time increases also. The way of acquiring applicable information in a feasible time must often be changed. This paper proposes our design for such data processing resulting in an extensive road network routing graph.

The described big data processing itself is in a state that completes a whole map of Europe nowadays. The time of total processing depends on many factors, but under certain settings and adjustments we are confident of a job to be done in two hours when countries processed simultaneously, with a full result.

We have successfully tested and proved our concept of navigation using this data for the road network and found it working well. Not only it guarantees the parsed road network is suitable for routing, it also shows that values provided by us, like maximal allowed speed computed[3] where not listed by OSM, can be used for prioritizing some paths over another.

An advantage we take of chosen approach is that we can export to a different file formats and vice versa. This is especially useful when it comes to querying, visualising or editing, which is more comfortable via spatial database. On the other hand routing algorithms run faster with use of HDF5 files.

As we can exploit described auditing and map editing[4] into routing systems for our own benefit, an automatic propagation to OSM database would be possible. This can also serve purposes for historical developmnet of road system.

[3] for example based on road classification and number of lanes
[4] that is forcing our changes locally; correcting road information from OSM dataset

Acknowledgement

This work has been partially funded by ANTAREX, a project supported by the EU H2020 FET-HPC program under grant 671623, by The Ministry of Education, Youth and Sports of the Czech Republic from the National Programme of Sustainability (NPU II) project 'IT4Innovations excellence in science - LQ1602'.

References

1. AutoTuning and Adaptivity appRoach for Energy efficient eXascale HPC systems. `http://www.antarex-project.eu`
2. Geofabrik. `https://www.geofabrik.de`
3. OpenStreetMap. `https://www.openstreetmap.org`
4. Osmosis. `https://wiki.openstreetmap.org/wiki/Osmosis`
5. Portable Batch System. `https://www.nas.nasa.gov/hecc/support/kb/portable-batch-system-(pbs)-overview_126.html`
6. PostgreSQL. `https://www.postgresql.org`
7. Protocol Buffers. `https://github.com/google/protobuf/`
8. Supercomputer Salomon Hardware Overview. `https://docs.it4i.cz/salomon/hardware-overview/`
9. The HDF Group. `https://www.hdfgroup.org`
10. Davidovic, N., Mooney, P.: Patterns of tagging in openstreetmap data in urban areas. In: Proceedings of GISRUK (2016)
11. Goetz, M., Lauer, J., Auer, M.: An algorithm based methodology for the creation of a regularly updated global online map derived from volunteered geographic information. In: Proceedings of the Fourth International Conference on Advanced Geographic Information Systems, Applications, and Services, Valencia, Spain. vol. 30, pp. 50–58 (2012)
12. Haklay, M.: How good is volunteered geographical information? a comparative study of openstreetmap and ordnance survey datasets. Environment and planning B: Planning and design 37(4), 682–703 (2010)
13. Haklay, M., Weber, P.: Openstreetmap: User-generated street maps. IEEE Pervasive Computing 7(4), 12–18 (2008)
14. Jilani, M., Corcoran, P., Bertolotto, M.: Automated highway tag assessment of openstreetmap road networks. In: Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems. pp. 449–452. ACM (2014)
15. Luxen, D., Vetter, C.: Real-time routing with openstreetmap data. In: Proceedings of the 19th ACM SIGSPATIAL international conference on advances in geographic information systems. pp. 513–516. ACM (2011)
16. Neis, P., Zielstra, D.: Recent developments and future trends in volunteered geographic information research: The case of openstreetmap. Future Internet 6(1), 76–106 (2014)