Rule-based reasoning using GPUs

Martin Peters

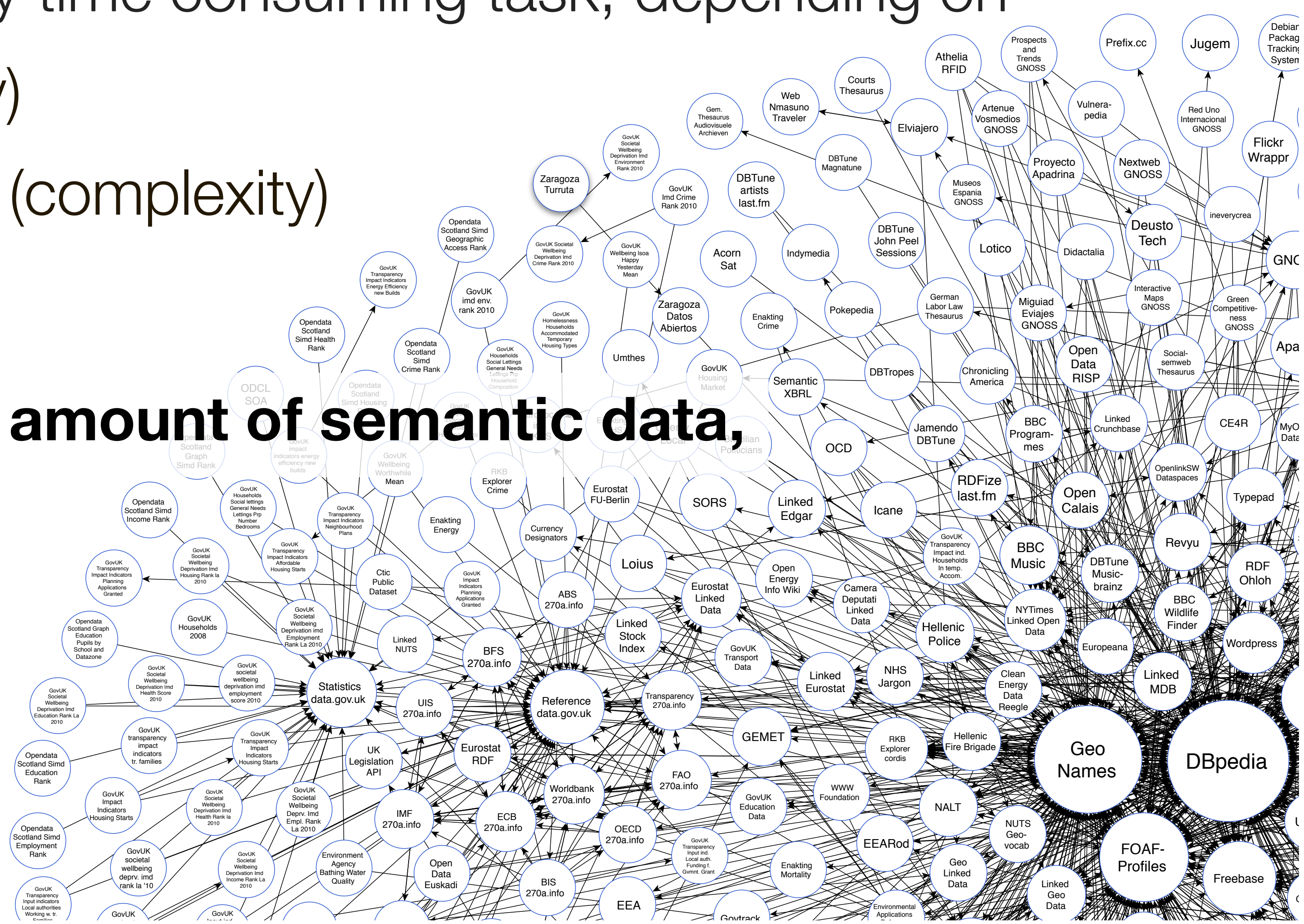
University of Applied Sciences and Arts Dortmund, Germany
Smart Environments Engineering Lab

Motivation

- Reasoning is one key feature when using ontologies
- Reasoning means to create new knowledge by inferring facts that are implicitly given by the existing data

- But: reasoning can still be a very time consuming task, depending on
  - the dataset (size, complexity)
  - the used ontology language (complexity)

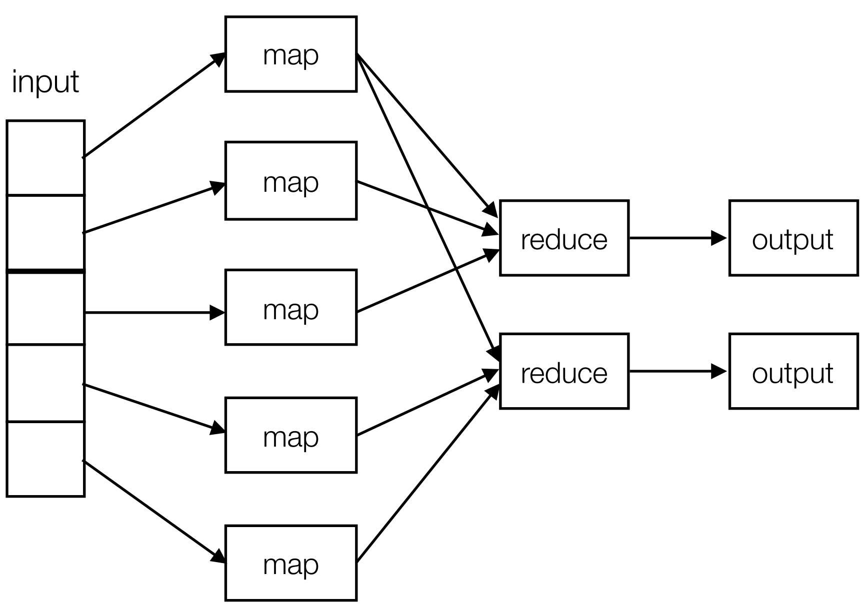
➔ **With respect to the growing amount of semantic data, reasoning needs to scale!**



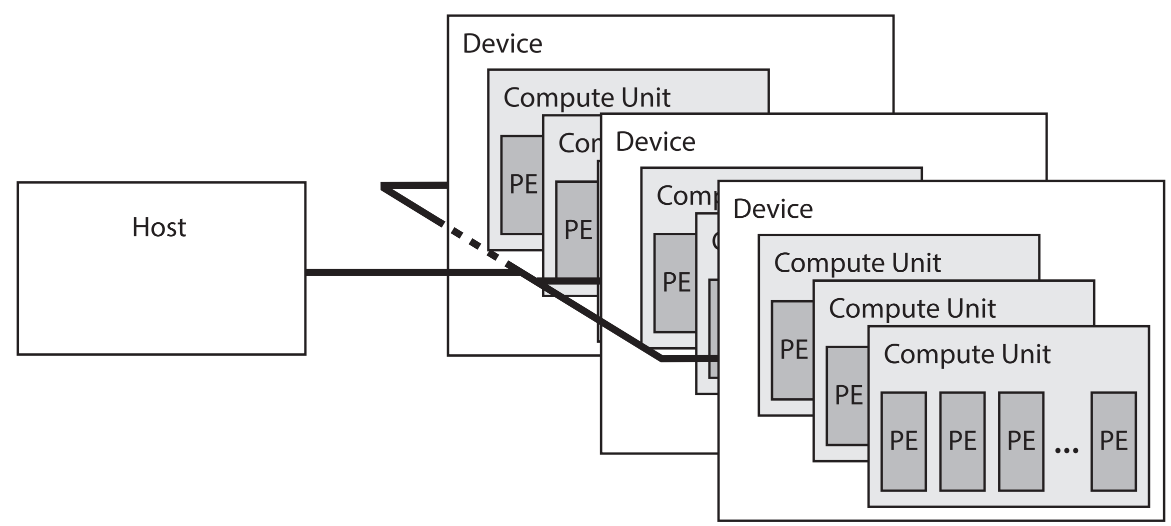
Linking Open Data cloud diagram 2014, by Max Schmachtenberg, Christian Bizer, Anja Jentzsch and Richard Cyganiak. http://lod-cloud.net/

How can the reasoning process be scaled?

1. Optimizing reasoning algorithms (each for a specific ontology language)
2. Scaling the hardware, e.g. in terms of a higher clock frequency
3. Scaling by parallelizing the reasoning process



Cluster / MapReduce



Multicore CPU/GPU

Large scale parallel reasoning

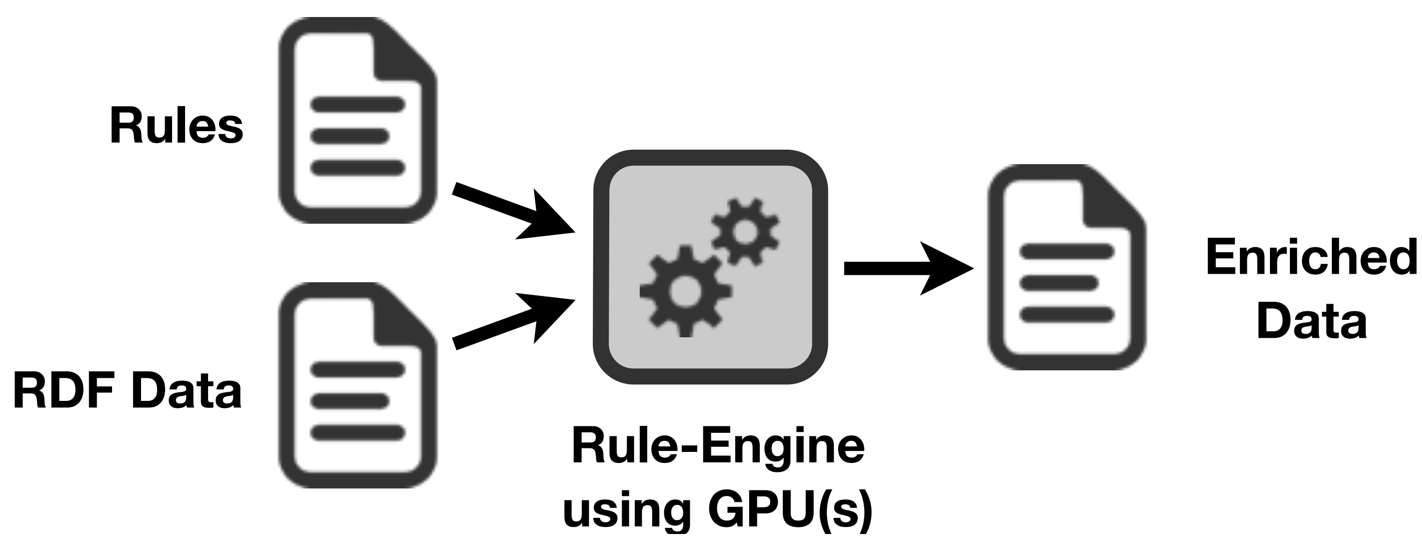
- Cluster-based approaches
  - Oren et al. (2009): divide-conquer-swap strategy
  - Maier et al. (2010): MapReduce for EL++
  - Liu et al. (2012): MapReduce for Fuzzy pD*
  - WebPie (Urbani et al. 2009/2010/2012): MapReduce for RDFS and pD*

- Approaches using a single machine
  - Kazakov et al. (2011): Classification of EL ontologies
  - Ren et al. (2011): ABox reasoning of EL ontologies
  - Urbani et al: (2013): DynamiTE: Parallel Materialization of Dynamic RDF Data
  - Heino et al. (2012): RDFS reasoning on massively parallel hardware

Large scale parallel reasoning: summary

- Current large scale reasoner:
  - usually rely on a specific ontology language and area of application
  - rarely make use of the highly parallel (and thus powerful) architecture of GPUs
  - provide a weak support for reasoning on single machines

- How to create a rule-based reasoner that makes use of single computing node and is able to perform large scale reasoning?



Implementing a rule-engine

What is the RETE algorithm?

- pattern-matching algorithm
- introduced by Charles Forgy in 1982
- widely used in many rule-engines as well as for semantic reasoner

- Basic steps:

1. create the RETE network
2. repeat until no new triples are derived
    - perform the alpha-matching
    - perform the beta-matching
    - fire rules

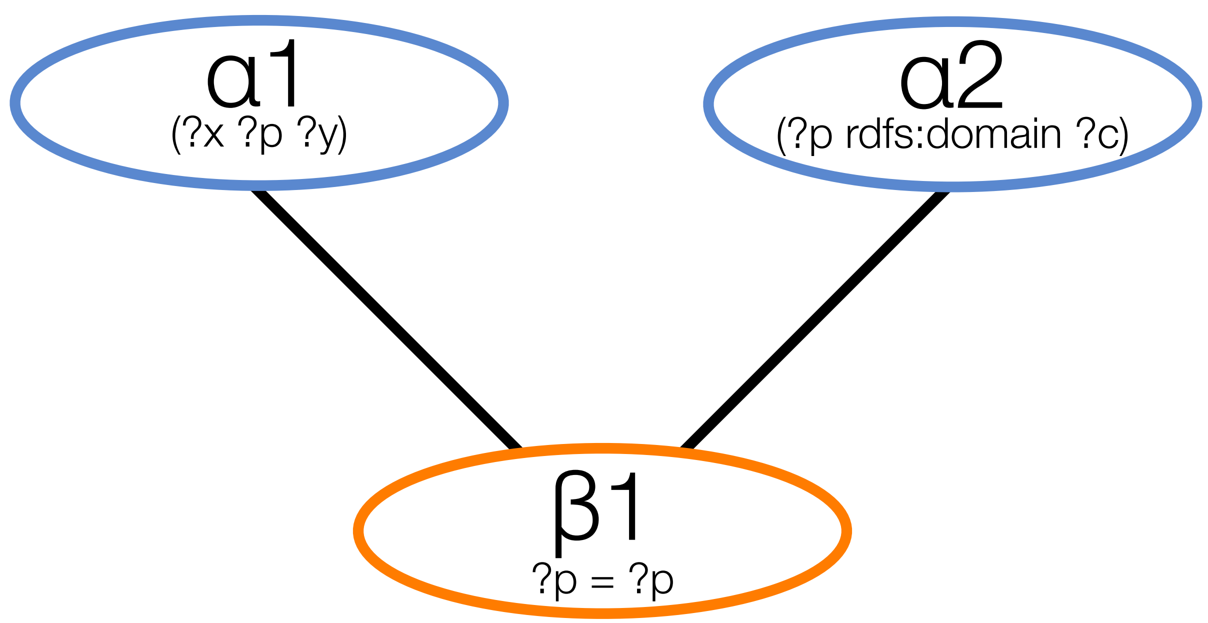
The RETE network

R1: (?x ?p ?y) → (?p rdf:type rdf:Property)

R2: (?x ?p ?y) (?p rdfs:domain ?c) → (?x rdf:type ?c)

α1 α2

β1



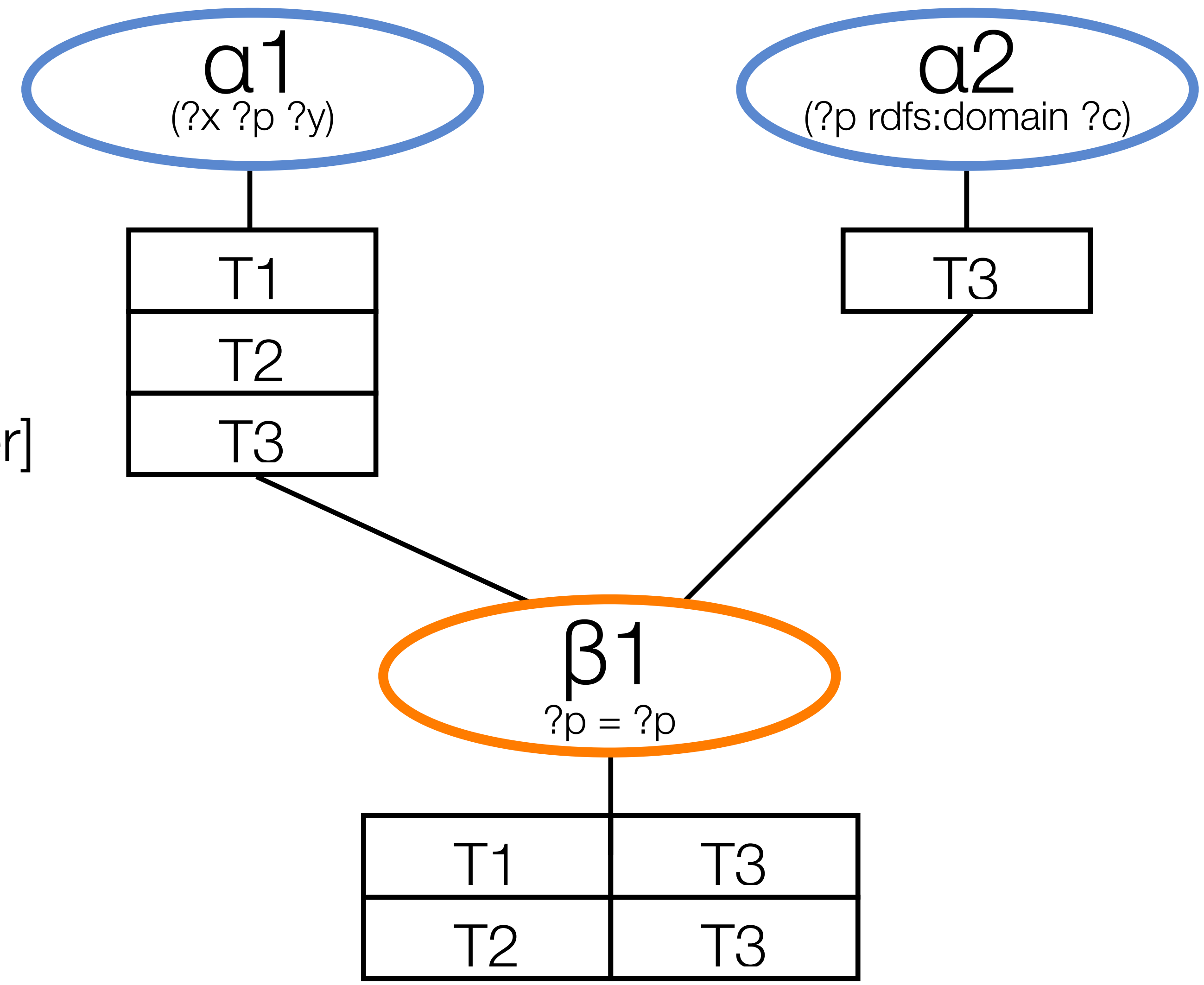
alpha- and beta matching

R1: (?x ?p ?y) → (?p rdf:type rdf:Property)

R2: (?x ?p ?y) (?p rdfs:domain ?c) → (?x rdf:type ?c)

α1 α2

β1



T1: [Bob uni:publishes Paper1]

T2: [Alice uni:publishes Paper2]

T3: [uni:publishes rdfs:domain Researcher]

T4: [uni:publishes rdf:type rdf:Property]

[uni:publishes rdf:type rdf:Property]

T5: [rdfs:domain rdf:type rdf:Property]

T6: [Bob rdf:type Researcher]

T7: [Alice rdf:type Researcher]

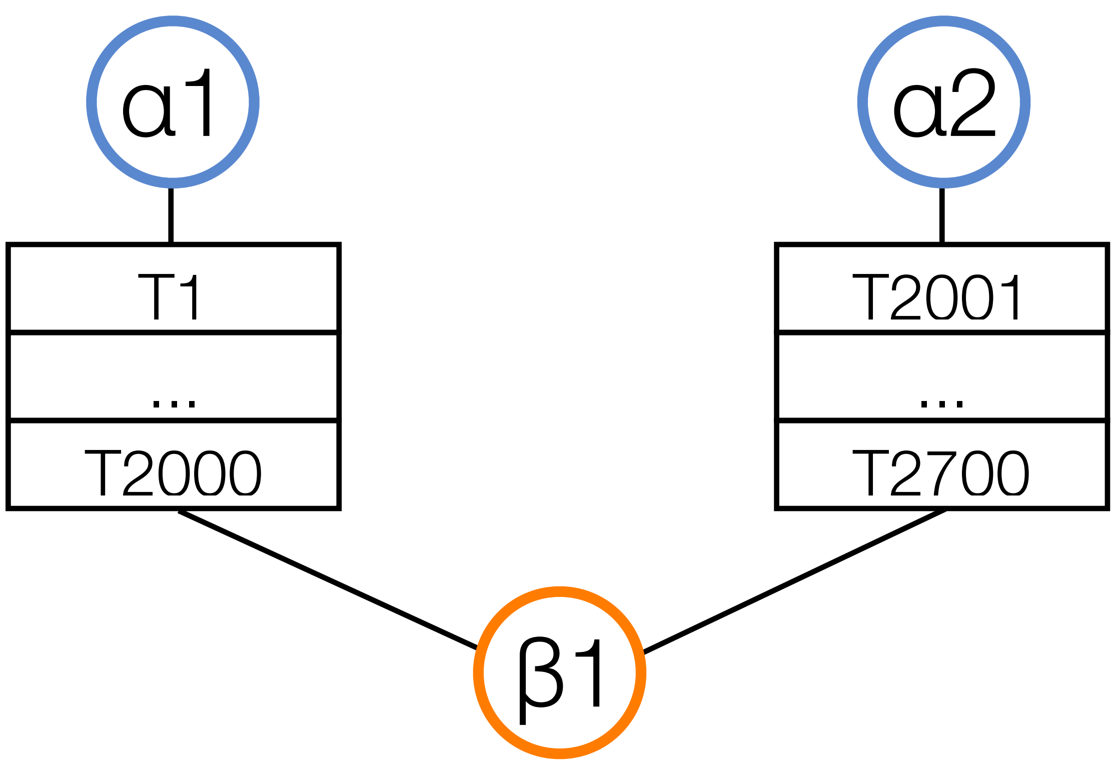
Parallelization

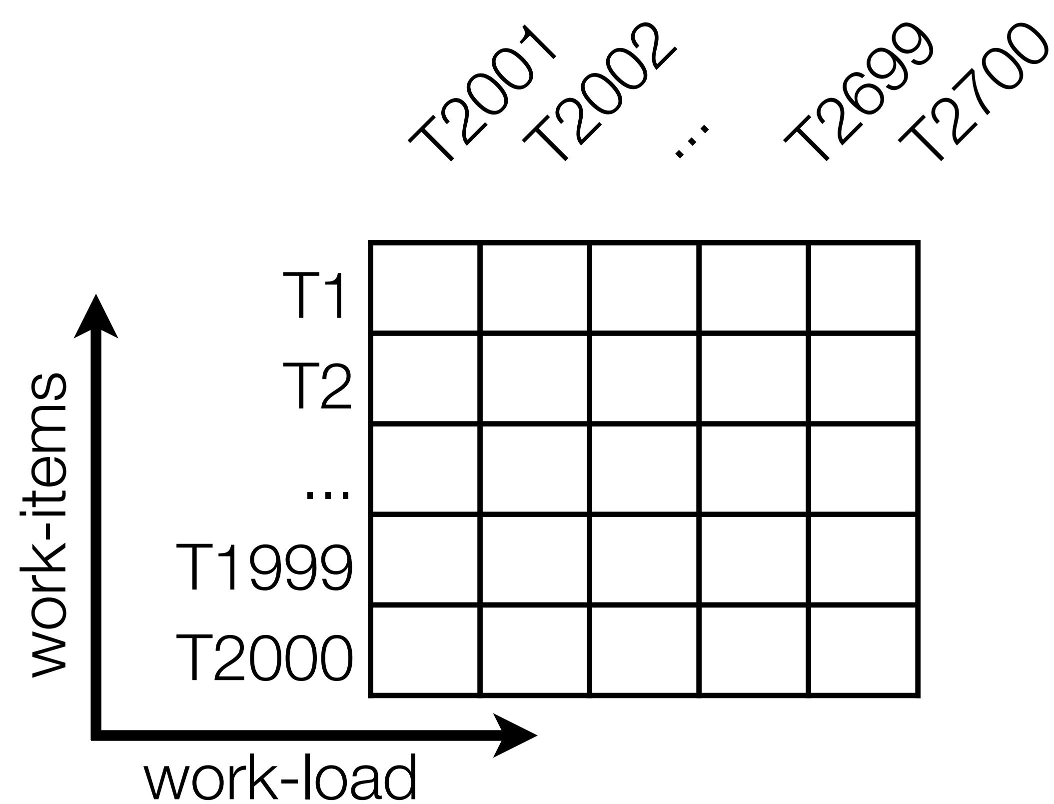
- Basically two approaches, to parallelize the RETE algorithm
    - data partitioning → huge amount of synchronization necessary
    - rule partitioning → parallelization depends on the number of rules

- targeting modern GPUs as parallel hardware, both approaches don‘t work out
    - global synchronization of data can be very expensive
    - a problem needs to be break down into a high amount (e.g. millions) of small problems that can be computed independently, thus, a high amount of parallel threads should be executed

Parallelization: targeting massively parallel Hardware

- alpha-matching
  - for each input triple one thread is created that checks, if that triple matches to one or more alpha-nodes (n triples → n threads)
- beta-matching
  - one thread for each match of one of the parent-nodes, that iterates through all matches of the second parent-node and checks for a match

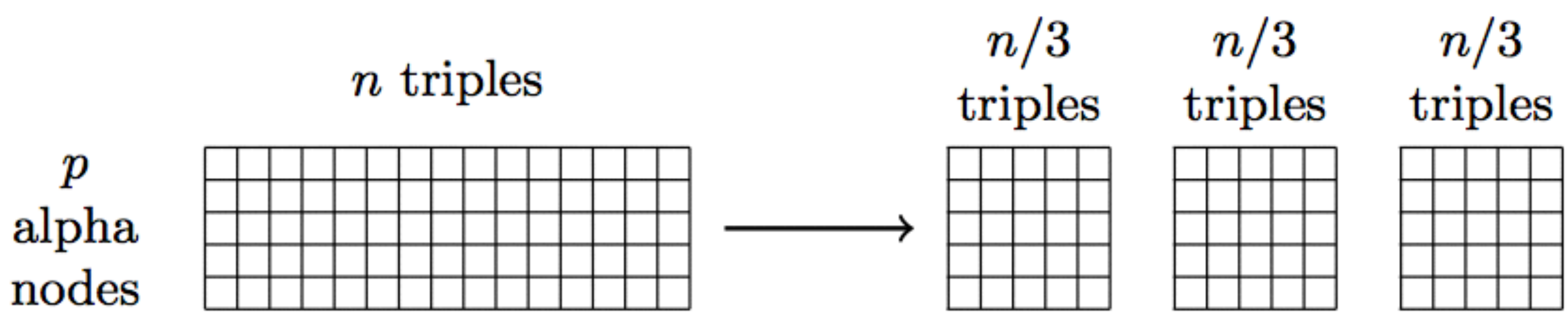




How to handle large datasets?

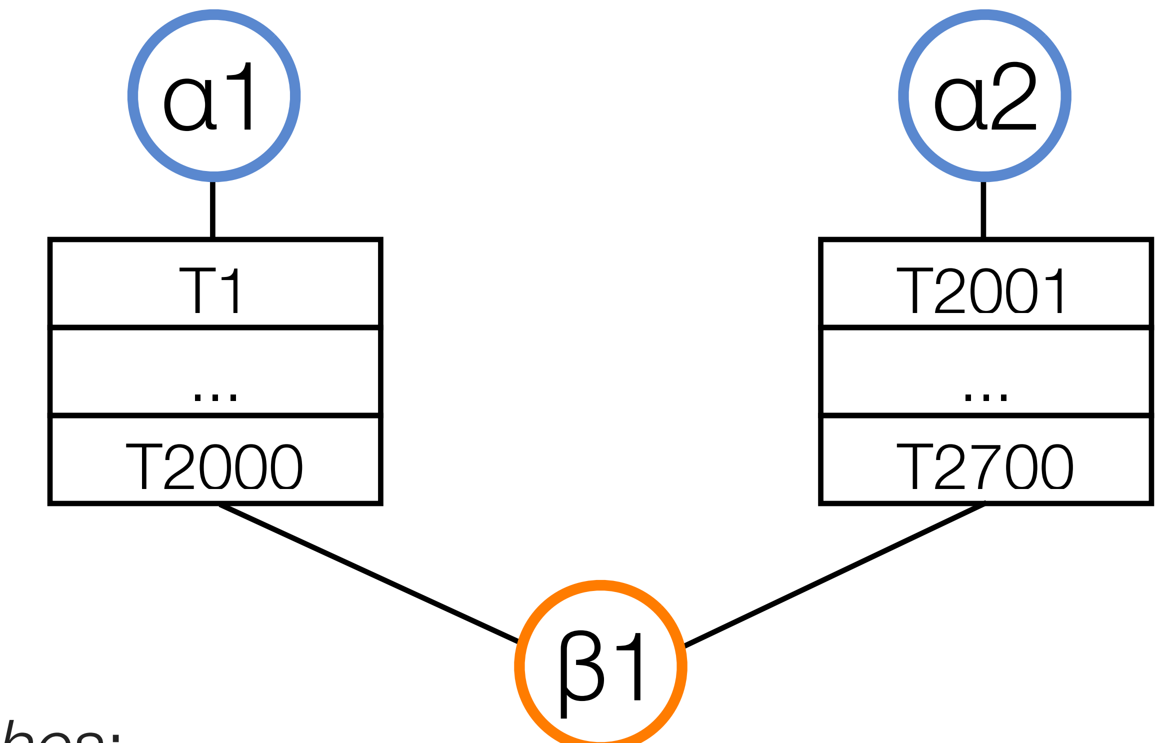
alpha-matching:

- the workload can easily be partitioned into smaller chunks that can be processed independently
- the chunk size can be chosen with respect to the target device



beta-matching

- for beta-matching the workload cannot simply be divided
  - the triple-references in the working-memories need to be resolved
  - thus, all triples need to be available during that step
  - this would limit the size of processable data to the amount of triples, that fit into the memory of the GPU

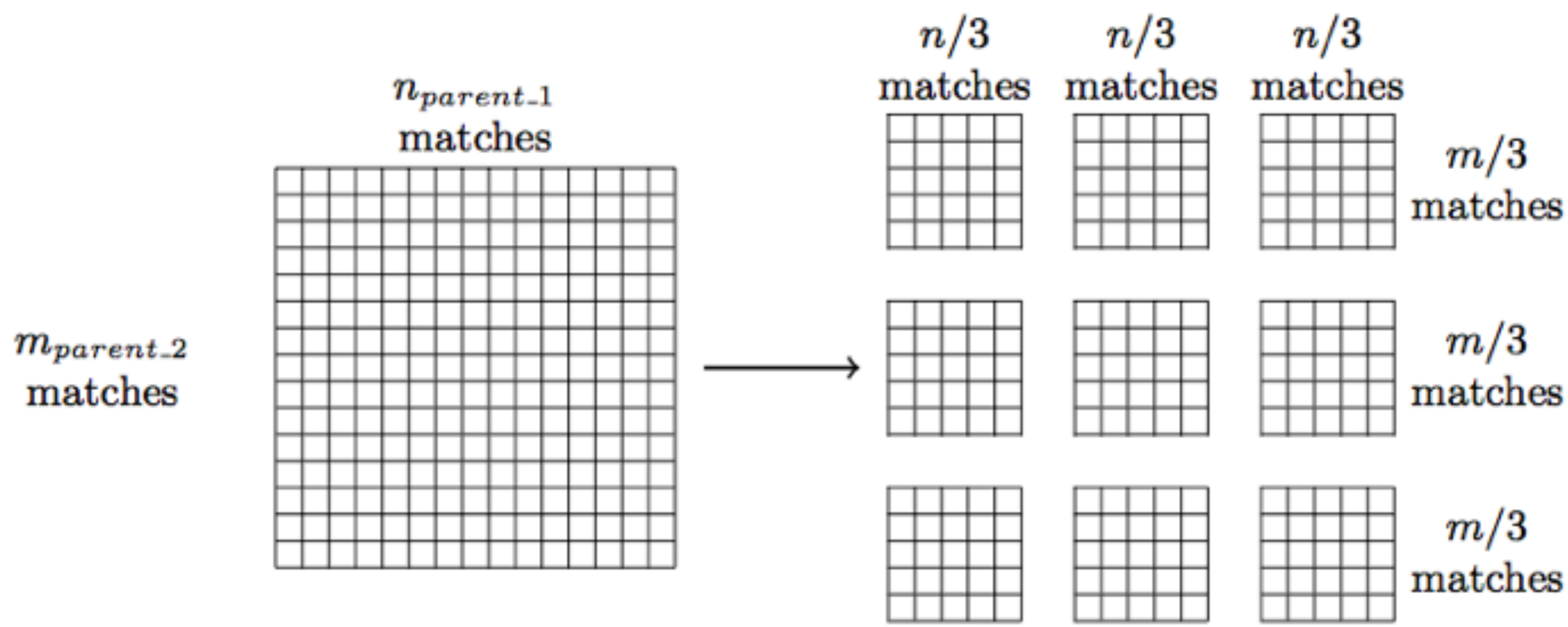


- to overcome this issue, we use *triple-matches*:

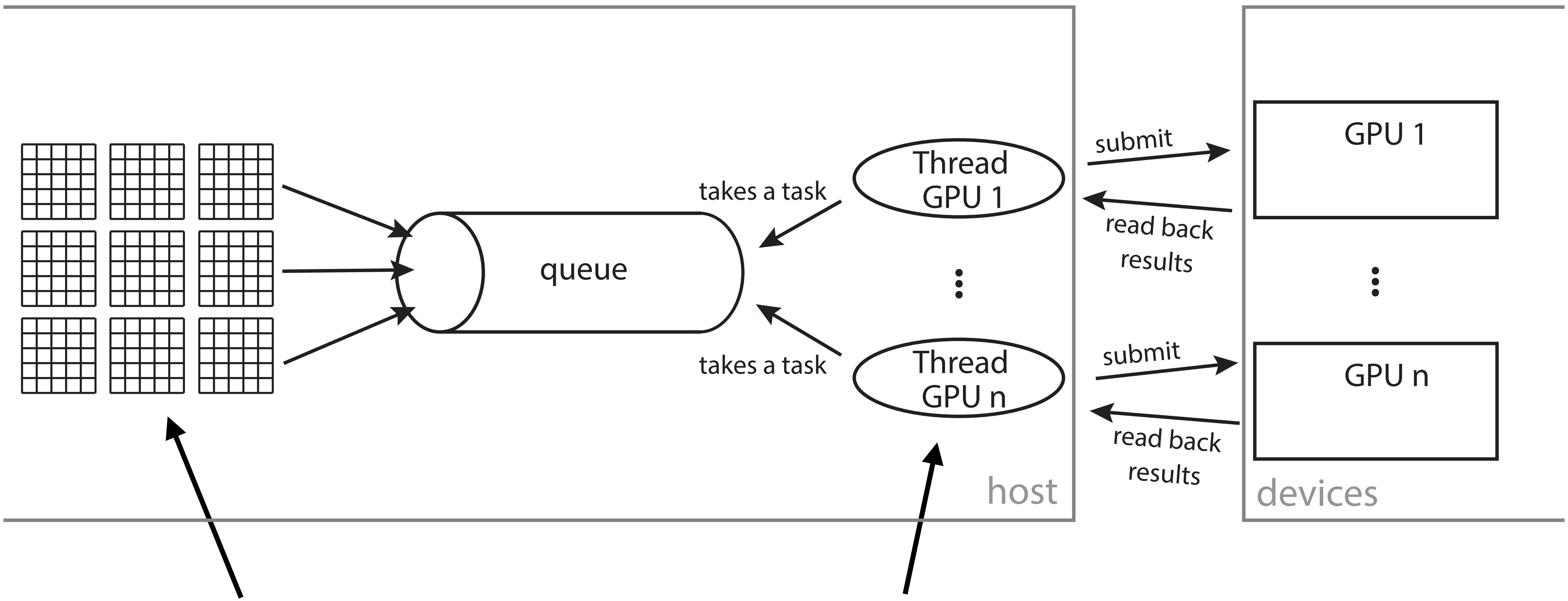
*A **triple-match m = (s,p,o,r)** is a quadruple with s=subject, p=predicate, o=object of a triple and r=triple reference (unique number, that is used for identification in the internal triple store).*

beta-matching II

- working-memories need to be transformed to triple-matches which then are transferred to the GPU for beta-matching
  - the working-memories can be divided to smaller chunks
  - the transformation of the chunks can be done in parallel, too (using multithreading)



Test environment: architecture



chunks are prepared and submitted to the queue using java multithreading

one thread for each GPU (each thread has exclusive rights to a GPU)

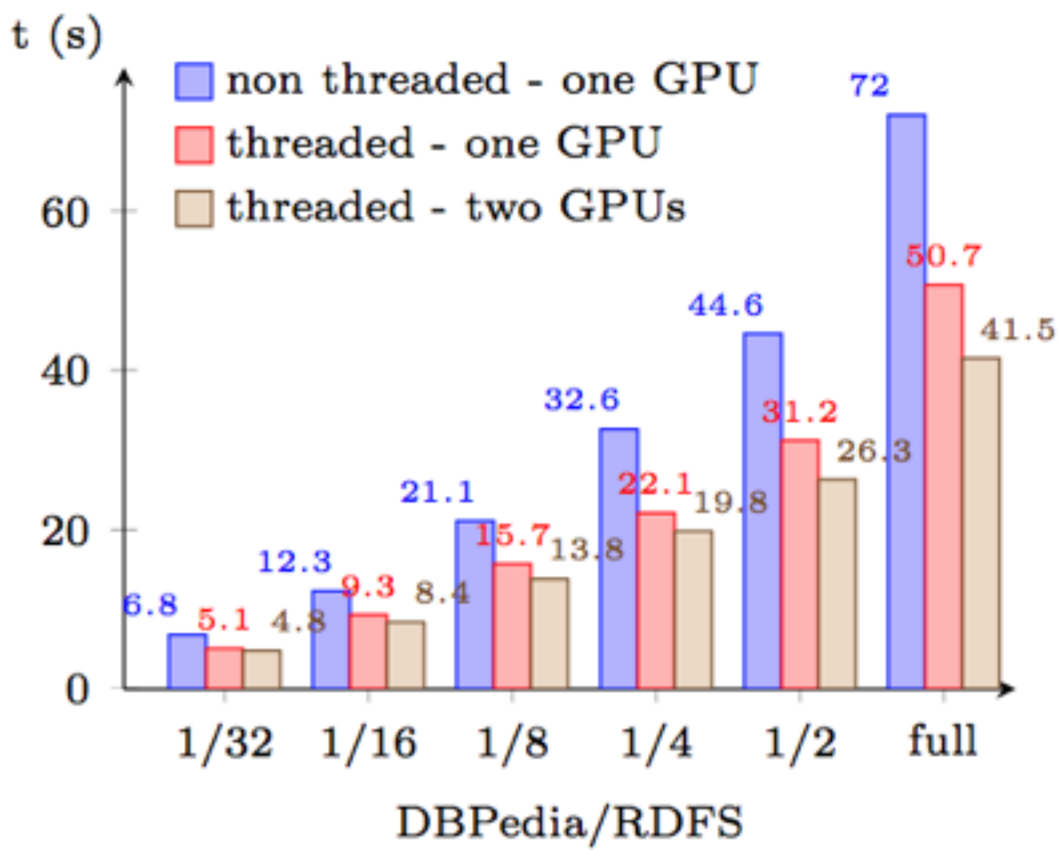
Test environment: setup

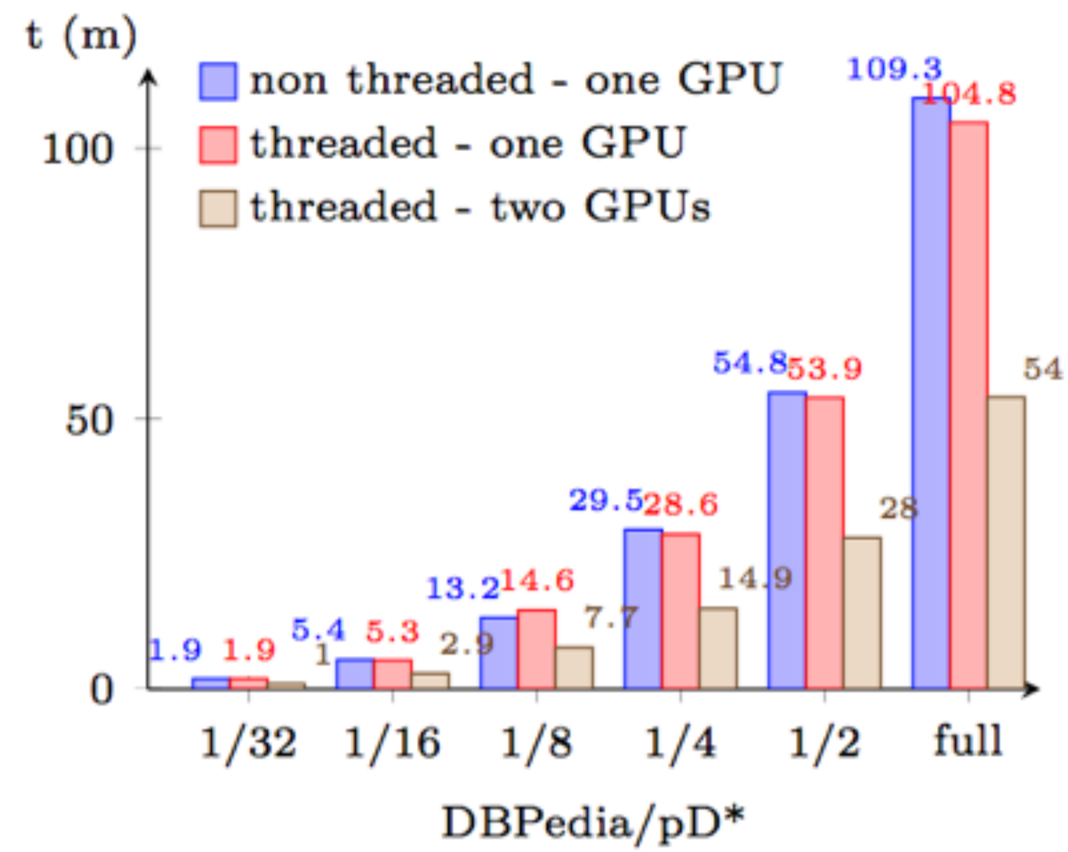
- Datasets:
  - Lehigh University Benchmark (LUBM): LUBM125 to LUBM8000
  - DBpedia scaled to full, 1/2nd, 1/4th, 1/8th, 1/16th, and 1/32nd

- Workstation with Ubuntu 12.04:
  - 2.0 GHz Intel Xeon processor with 6 cores
  - 64 GB memory
  - two AMD 7970 gaming graphic cards with 3GB of memory each

Parallelization: using chunks and multiple GPUs

- evaluation of the impact of:
  - using multithreading to prepare workload-chunks
  - use of a second GPU

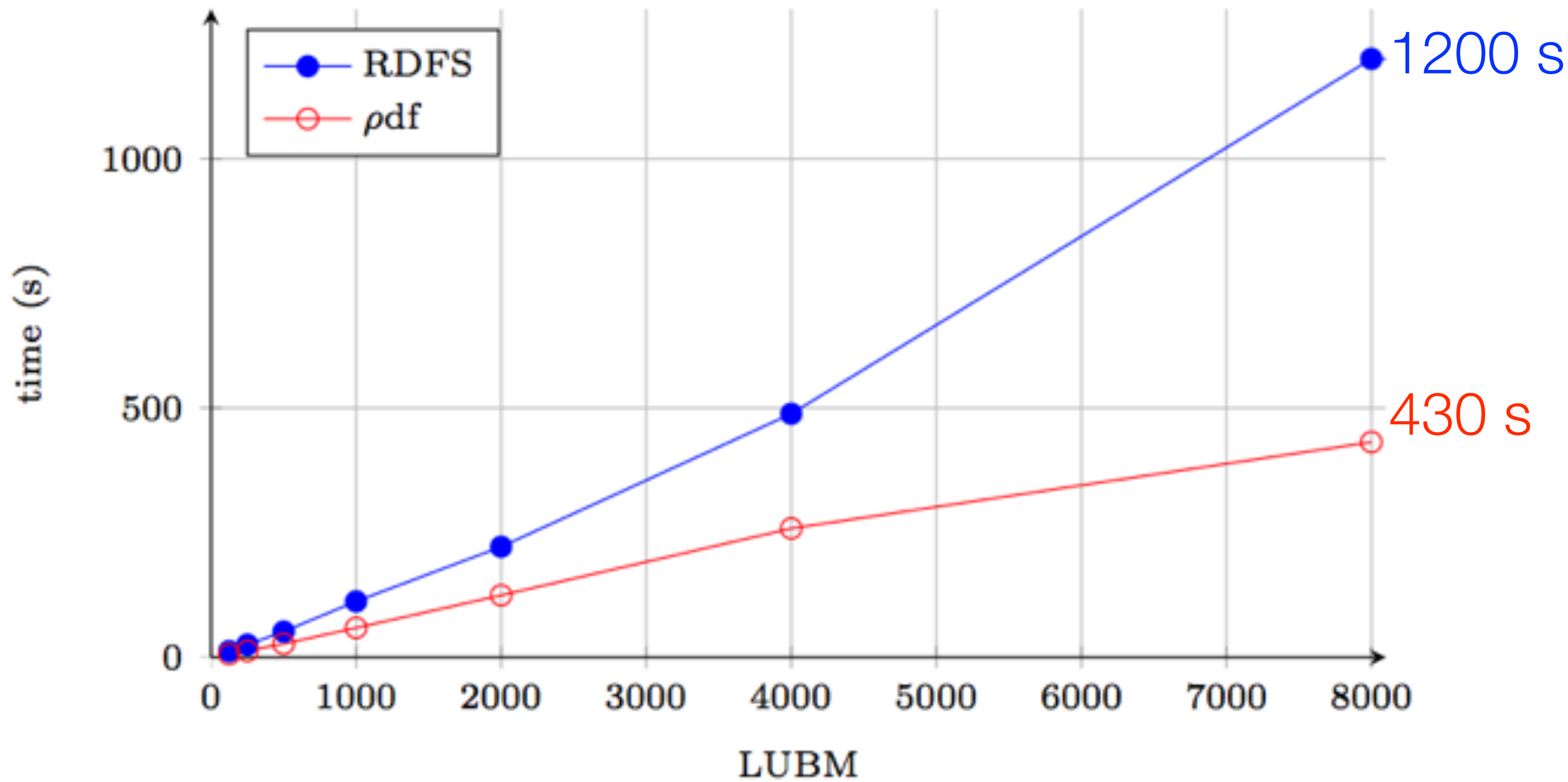




- RDFS reasoning benefits primarily from the thread-level parallelization
- pD* reasoning benefits significantly from the second GPU

Scaleability

- used Hardware: cloud-server with two Tesla M2090 GPUs and 192GB memory
- applying ρdf and RDFS to LUMB datasets from 17.6M to more than 1.1B triples



- max throughput:
  - ~ 2.7M triples/sec. for ρdf (WebPIE reported 2.1M triples/sec on 64 computing nodes)
  - ~ 1.4M triples/sec. for RDFS

Conclusion and future work

- we parallelized the RETE-algorithm for semantic reasoning in a way that
  - the preparation of the workload can be performed in parallel using multithreading
  - the matching process can be performed on the GPU
  - the workload can be distributed to multiple GPUs
- we showed that **large scale** and **rule-based** reasoning (to a limited size) is possible on a **single computing node**
- the new limitation we reached is the main-memory of the computing node itself

- future work will include:
  - investigation of concepts to reduce the memory usage
  - distributing the workload not only to multiple GPUs, but also to multiple hosts equipped with GPUs

Fachhochschule
Dortmund
University of Applied Sciences and Arts

Thank you for your attention!

contact:

Martin Peters
martin.peters@fh-dortmund.de