When to Reach for the Cloud: Using Parallel Hardware for Link Discovery

Axel-Cyrille Ngonga Ngomo, Lars Kolb, Norman Heino, Michael Hartung, Sören Auer, and Erhard Rahm

Department of Computer Science, University of Leipzig
04109 Leipzig, Germany
{ngonga,kolb,heino,hartung,auer,rahm}@informatik.uni-leipzig.de

Abstract. With the ever-growing amount of RDF data available across the Web, the discovery of links between datasets and deduplication of resources within knowledge bases have become tasks of crucial importance. Over the last years, several link discovery approaches have been developed to tackle the runtime and complexity problems that are intrinsic to link discovery. Yet, so far, little attention has been paid to the management of hardware resources for the execution of link discovery tasks. This paper addresses this research gap by investigating the efficient use of hardware resources for link discovery. We implement the $\mathcal{HR}^3$ approach for three different parallel processing paradigms including the use of GPUs and MapReduce platforms. We also perform a thorough performance comparison for these implementations. Our results show that certain tasks that appear to require cloud computing techniques can actually be accomplished using standard parallel hardware. Moreover, our evaluation provides break-even points that can serve as guidelines for deciding on when to use which hardware for link discovery.

Keywords: Link discovery, MapReduce, GPU

1 Introduction

Link Discovery (LD) is of central importance for realizing the fourth Linked Data principle [1]. With the growth of the Web of Data, the complexity of LD problems has grown considerably. For example, linking places from *LinkedGeoData* and *DBpedia* requires the comparison of hundreds of thousands of instances. Over the last years, several time-efficient algorithms such as *LIMES* [19], *MultiBlock* [9] and $\mathcal{HR}^3$ [18] have been developed to address the problem of the a-priori quadratic runtime of LD approaches. In general, these algorithms aim at minimizing the number of unnecessary similarity computations to carry out. While these approaches have been shown to outperform naïve LD implementations by several orders of magnitude, the sheer size of the number of links can still lead to unpractical runtimes. Thus, cloud implementations of some of these algorithms (e.g., *LIMESMR* [7] and Silk MapReduce[1]) have been recently developed. The speed-up of these implementations is, however, limited

[1] https://www.assembla.com/spaces/silk/wiki/Silk_MapReduce

by a considerable input-output overhead that can lead to worse runtimes than on single machines. Interestingly, the use of standard parallel hardware has recently been shown to have the potential to outperform cloud computing techniques [6].

The multiplicity of available hardware solutions for carrying out LD led us to ask the following fundamental question: *When should which type of hardware be used to optimize the runtime of LD processes?* Providing an answer to this question promises to enable the development of highly flexible and scalable LD frameworks that can adapt to the available hardware environment. It will allow to decide intelligently upon when to reach for remote computing services such as cloud computing services in contrast to using local resources such as graphics processing units (GPUs) or multi-processor and multi-core technology. To answer our research question, we compare the runtimes of several implementations of $\mathcal{HR}^3$ for several datasets and find break-even points for different hardware. We chose the $\mathcal{HR}^3$ algorithm because it is the first algorithm with a guaranteed reduction ratio [18]. Thus, it promises to generate less overhead than other LD algorithms for comparable problems. Moreover, this algorithm can be used in manifold scenarios including LD, finding geographically related data (radial search) as well as search space reduction for other LD algorithms. The main contributions of this work are:

- We present the first implementation of a LD approach for GPUs. It relies on the GPU for fast parallel indexing and on the CPU for the computation of distances.
- We show how load-balancing for Map-Reduce can be carried out for LD approaches in affine spaces.
- We obtain guidelines for the use of different parallel hardware for LD by the means of a comparative evaluation of different implementations on real-world datasets from the Linked Open Data Cloud.

The remainder of the paper is organized as follows: We begin by giving a brief overview of $\mathcal{HR}^3$ and other paradigms used in this work. In Section 3, we then show how $\mathcal{HR}^3$ must be altered to run on GPUs. Section 4 focuses on the Map-Reduce implementation of $\mathcal{HR}^3$ as well as the corresponding load balancing approach. Section 5 presents a comparison of the runtimes of the different implementations of $\mathcal{HR}^3$ and derives break-even points for the different types of hardware[2]. The subsequent section gives an overview of related work. Finally, Section 7 summarizes our findings and presents future work.

2 Preliminaries

The specification of *link discovery* adopted herein is tantamount to the definition proposed in [18]. Given a formal relation[3] R and two (not necessarily disjoint) sets of instances S and T, the goal of link discovery is to find the set $M = \{(s,t) \in S \times T : R(s,t)\}$. Given that the explicit computation of R is usually a very complex endeavor, most frameworks reduce the computation of M to that of the computation of an approximation $\tilde{M} = \{(s,t) : \delta(s,t) \leq \theta\}$, where δ is a (complex) distance function and θ is a

[2] Details to the experiments and code are available at `http://limes.sf.net`.

[3] For example, `http://dbpedia.org/property/near`

distance threshold. Note that when $S = T$ and $R =$ `owl:sameAs`, the link discovery task becomes a *deduplication* task. Naïve approaches to computing $\tilde{M}$ have a quadratic time complexity, which is impracticable on large datasets. Consequently, a large number of approaches has been developed to reduce this time complexity (see [18] for an overview). Most of these approaches achieve this goal by optimizing their reduction ratio. In newer literature, the $\mathcal{HR}^3$ algorithm [17] has been shown to be the first algorithm which guarantees that it can achieve any possible reduction ratio.

$\mathcal{HR}^3$ builds upon the *HYPPO* algorithm presented in [16]. The rationale of $\mathcal{HR}^3$ is to maximize the reduction ratio of the computation of $\tilde{M}$ in affine spaces with Minkowski measures. To achieve this goal, $\mathcal{HR}^3$ computes an approximation of $\tilde{M}$ within a discretization of the space $\Omega = S \cup T$. Each point $\omega = (\omega_1, \ldots, \omega_n) \in \Omega$ is mapped to discrete coordinates $(\lfloor \omega_1/\Delta \rfloor, \ldots, \lfloor \omega_n/\Delta \rfloor)$, where $\Delta = \theta/\alpha$ and $\alpha \in \mathbb{N}\backslash\{0\}$ is called the granularity parameter. An example of such a discretization is shown in Figure 1: The point B with coordinates $(12.3436, 51.3339)$ is mapped to the discrete coordinates $(2468, 10226)$. The set of all points with the same discrete coordinates forms a hypercube (short: cube) of width α in the space Ω. The cube that contains ω is called $C(\omega)$. We call the vector $(c_1, \ldots, c_n) = (\lfloor \omega_1/\Delta \rfloor, \ldots, \lfloor \omega_n/\Delta \rfloor) \in \mathbb{N}^n$ the coordinates of $C(\omega)$.

Given the distance threshold θ and the granularity parameter α, $\mathcal{HR}^3$ computes the set of candidates $t \in T$ for each $s \in S$ by using the index function given in Eq. 1.

$$index(C, C') = \begin{cases} 0, \text{ if } \exists i : |c_i - c'_i| \leq 1 \text{ with } i \in \{1, ..., n\}, \\ \sum_{i=1}^{n} (|c_i - c'_i| - 1)^p \text{ else.} \end{cases} \tag{1}$$

where $C = C(s)$ and $C' = C(t)$ are hypercubes and p is the order of the Minkowski measure used in the space Ω.

Now, all source instances s are only compared with the target instances t such that $index(C(s), C(t)) \leq \alpha^p$. In our example, this is equivalent to computing the distance between B and all points contained in the gray-shadowed area on the map. Overall, $\mathcal{HR}^3$ achieve a reduction ratio of ≈ 0.82 on the data in Figure 1 as it only performs 10 comparisons instead of 55.

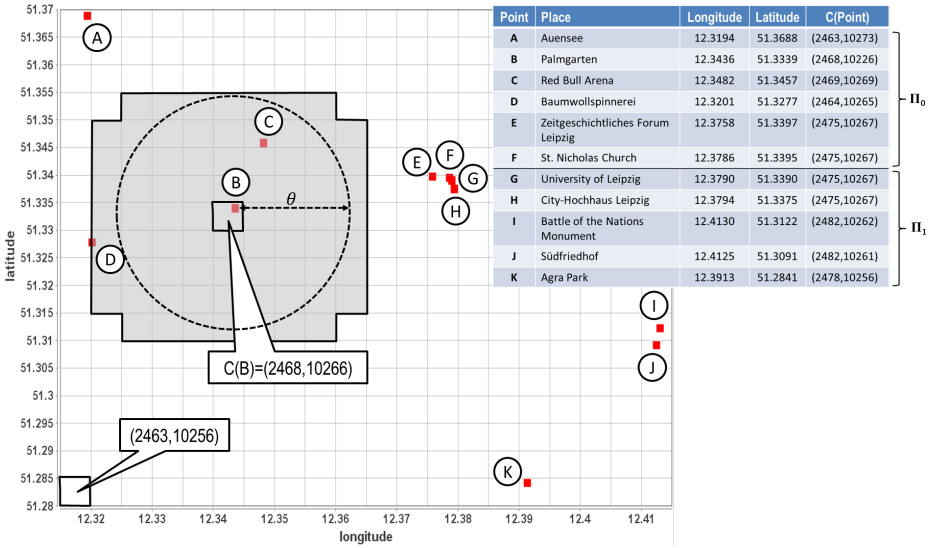
3 Link Discovery on GPUs

3.1 General-Purpose Computing on GPUs

GPUs were originally developed for processing image data. Yet, they have been employed for general-purpose computing tasks in recent years. Compared to CPUs the architecture of GPU hardware exhibits a large number of simpler compute cores and is thus referred to as *massively parallel*. A single compute core typically contains several arithmetic and logic units (ALU) that execute the same instruction on multiple data streams (SIMD).

Parallel code on GPUs is written as *compute kernels*, the submission of which is orchestrated by a host program executed on the CPU. Several frameworks exist for performing general purpose computing on GPUs. In this work we use *OpenCL*[4], a vendor-

[4] `http://www.khronos.org/opencl/`



Point	Place	Longitude	Latitude	C(Point)	
A	Auensee	12.3194	51.3688	(2463,10273)	Π_0
B	Palmgarten	12.3436	51.3339	(2468,10226)	
C	Red Bull Arena	12.3482	51.3457	(2469,10269)	
D	Baumwollspinnerei	12.3201	51.3277	(2464,10265)	
E	Zeitgeschichtliches Forum Leipzig	12.3758	51.3397	(2475,10267)	
F	St. Nicholas Church	12.3786	51.3395	(2475,10267)	
G	University of Leipzig	12.3790	51.3390	(2475,10267)	Π_1
H	City-Hochhaus Leipzig	12.3794	51.3375	(2475,10267)	
I	Battle of the Nations Monument	12.4130	51.3122	(2482,10262)	
J	Südfriedhof	12.4125	51.3091	(2482,10261)	
K	Agra Park	12.3913	51.2841	(2478,10256)	

Fig. 1. Example dataset containing 11 places from Leipzig. To identify all points with a maximum Euclidean distance $\theta = 0.02$, the space is virtually tiled into hypercubes with an edge length of $\Delta = \theta/4$. A cube is identified by its coordinates $(c_1, ..., c_n)$. The gray-shadowed cells indicate the cubes whose points are compared with B, i.e., $\{C' \mid index(C(B), C') \leq \alpha^p\}$.

agnostic industry standard. The memory model as exposed to OpenCL kernels is depicted in Figure 2: An instance of a compute kernel running on a device is called a *work item* or simply thread[5]. Work items are combined into *work groups*. All items within the same group have access to low-latency local memory and the ability to synchronize load/store operations using barriers. Thus, the actual number of kernel instances running in parallel is often limited by register and local memory usage. Each work item is assigned a globally (among all work items) and locally (within a work group) unique identifier, which also imposes a scheduling order. Typically those identifiers are used to compute local and global memory offsets for loading and storing data items that a given thread works on. Data transfer between host program and compute device is done via global device memory to which all work items have access, albeit with higher latency.

Threads on modern GPUs do not run in isolation. They are scheduled in groups of 64 or 32 work items depending on the hardware vendor. All threads within such a group execute the same instruction in lock-step. Any code path deviations due to control flow statements need to be executed by all items, throwing away unnecessary results (predication). It is therefore essential that each work item in such a group performs the same amount of work. The OpenCL framework does not expose the size of such groups to the API user. An upper bound is given by the work group size, which is always an integer multiple of the schedule group size.

[5] We use the therms work item and thread interchangeably in this work.

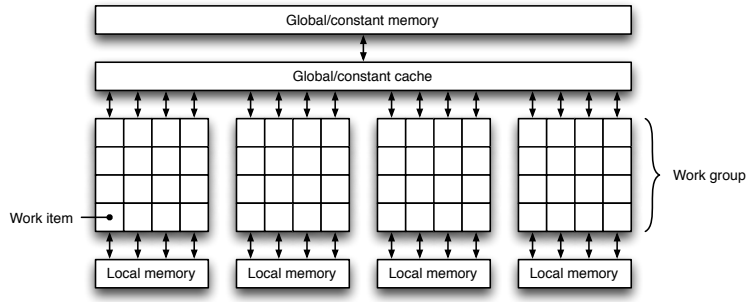


Fig. 2. OpenCL memory model

3.2 GPU-based $\mathcal{HR}^3$ Implementation

For GPU-based computation all data must be copied to the device via the PCIe bus. We therefore only perform expensive computations on the device that benefit from the massive parallelism. In the case of $\mathcal{HR}^3$ this is the computation of the index function that determines which hypercubes a given cubes needs to be compared with. Since GPUs work best with regular memory access patterns a few preparation steps are needed. These are performed serially on the host. First, we discretize the input space $\Omega = S \cup T$, resulting in a set of hypercubes. All hypercubes are then sorted component-wise. The number of hypercubes determines the global work size. That is, each thread is assigned a hypercube (called *pivot cube*) determined by its global id. The work to be done by each thread is then to compute all those hypercubes that abide by the bound on indexes set by $\mathcal{HR}^3$.

A naïve implementation would have each thread compare its pivot cube to all other cubes, resulting in an amount of work quadratic in the number of hypercubes. A better approach is to minimize the amount of cube comparisons while maintaining an even work distribution among threads within the same group. Since hypercubes are globally sorted and fetched by work items in increasing schedule order, the ordering is maintained also locally. That is, let $g = k + 1$ be the local work group size. The work item with the least local id per group is assigned the smallest pivot cube C^0 while the last work item having the highest local id operates on the largest cube C^k as its pivot. Both work items therefore can determine a lower and upper bound for the whole group as follows. The first item computes the cube $C^{0-\alpha} = (c_1^0 - \alpha, \ldots, c_n^0 - \alpha)$ and the last item computes the cube $C^{k+\alpha} = (c_1^k + \alpha, \ldots, c_n^k + \alpha)$, where c_i^0 and c_i^k are the coordinates of the respective pivot cubes. Thread 0 then determines i_l, the index of the largest cube not greater then $C^{0-\alpha}$ while thread k computes i_u, the index of the smallest cube that is greater than $C^{k+\alpha}$. After a barrier synchronization that ensures all work items in a group can read the values stored by threads 0 and k, all work items compare their pivot cube to cubes at indices $i_l, \ldots (i_u - 1)$ in global device memory. Since all work items access the same memory locations fetches can be efficiently served from global memory cache.

In OpenCL kernels dynamic memory management is not available. That is, all buffers used during a computation must be allocated in advance by the host program.

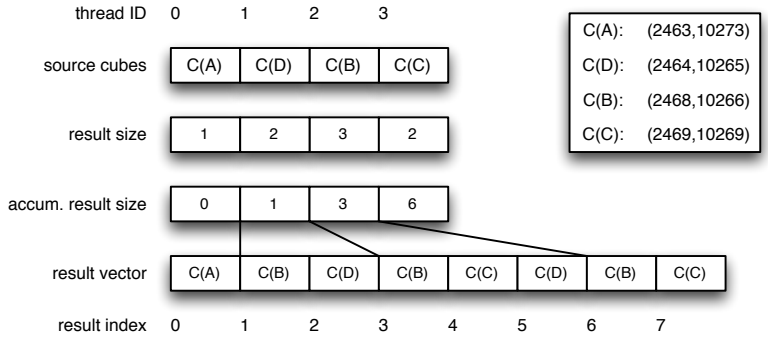


Fig. 3. Result index computation for $\mathcal{HR}^3$ on GPU hardware

In particular, the size of the result buffer must be known before submitting a kernel to a device. We therefore cannot simply write the resulting cubes to an output vector. Instead, we compute results in two passes. During the first pass each thread writes the number of results it needs to produce to an output vector. A prefix sum over this vector yields at each index the accumulated number of results of threads with a lower id. This value can be used as an index into the final output vector at which each thread can start writing its results.

As an example consider Figure 3. It shows four threads $(0 \ldots 3)$, each of which loads a single cube from the sorted *source cubes vector*. The index from which each threads loads its cube is given by its id[6]. In this example we assume a granularity factor of $\alpha = 4$. For thread 1 the smallest cube its pivot cube needs to be compared with is $C(D) = (2464, 10265)$ while the largest is $C(B) = (2468, 10266)$. It therefore writes 2 into an output vector, again using its thread id as an index. Thread 0 as well as 2 and 3 do the same, which results in the *result size vector* as depicted in Figure 3. In order to determine the final indexes each thread can use for storing its results in the result vector, an exclusive prefix sum is computed over the result size vector. This operation computes at each index i the sum of the elements at indexes $0 \ldots (i-1)$, resulting in the *accumulated result size vector*. A result vector of the appropriate size is allocated and in a second kernel run each thread can now write the cube coordinates starting at the index computed in the previous step. Indexing results are then copied back to the host where comparison of the actual input points is carried out. Since this operation is dominated by the construction of the result it cannot be significantly improved on parallel hardware.

4 MapReduce-based Link Discovery

In this section we present an implementation of $\mathcal{HR}^3$ with MapReduce (MR), a programming model designed for parallelizing data-intensive computing in cluster envi-

[6] For means of readability we show only one id per thread that serves as both its local and global id.

Algorithm 1: Basic $\mathcal{HR}^3$ - Map

```
1  map(k_in=unused, v_in = ω)
2      Δ ← θ/α;
3      cid_1 ← getCubeId(C(ω));
4      RC ← getRelatedCubes(C(ω), Δ);
5      foreach C' ∈ RC do
6          cid_2 ← getCubeId(C');
7          if cid_1 ≤ cid_2 then
8              output(cid_1.cid_2.0,
9                (ω, 0));
10         else
11             output(cid_2.cid_1.1,
12               (ω, 1));
   // part = hash(cid_1,cid_2) mod r
   // sort component-wise by entire key
   // group by cid_1, cid_2
```

Algorithm 2: Basic $\mathcal{HR}^3$ - Reduce

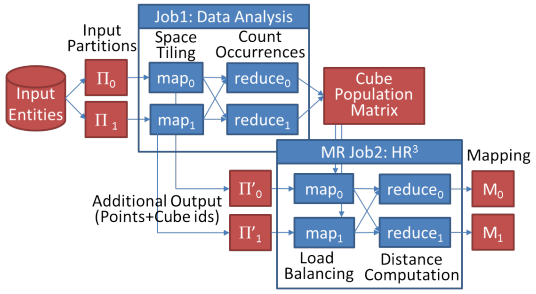
```
1  reduce(k_tmp=cid_1, cid_2,
2    v_tmp=list< ω, flag) >)
3      buf ← {};
4      if cid_1 = cid_2 then
5          foreach (ω, flag) ∈ v_tmp do
6              foreach ω' ∈ buf do
7                  compare(ω, ω');
8              buf ← buf ∪ {ω};
9      else
10         foreach (ω, flag) ∈ v_tmp do
11             if flag=0 then
12                 buf ← buf ∪ {ω};
13             else
14                 foreach ω' ∈ buf do
15                     compare(ω, ω');
```

ronments [2]. MR implementations like *Apache Hadoop* rely on a distributed file system (DFS) that can be accessed by all nodes. Data is represented by key-value pairs and a computation is expressed employing two user-defined functions, `map` and `reduce`, which are processed by a fixed number of map (m) and reduce tasks (r). For each intermediate key-value pair produced in the map phase, a target reduce task is determined by applying a partitioning function that operates on the pair's key. The reduce tasks first sort incoming pairs by their intermediate keys. The sorted pairs are then grouped and the reduce function is invoked on all adjacent pairs of the same group.

We describe a straightforward realization of $\mathcal{HR}^3$ as well as an advanced approach that considers skew handling to guarantee load balancing and to avoid unnecessary data replication. In favor of readability, we consider a single dataset only.

4.1 $\mathcal{HR}^3$ with MapReduce

$\mathcal{HR}^3$ can be implemented with a single MR job. The main idea is to compare the points of two related cubes within a single reduce call. We call two cubes C, C' *related* iff $index(C, C') \leq \alpha^p$. For each input point ω, the map function determines the surrounding cube $C(\omega)$ and the set of related cubes RC, which might contain points within the maximum distance. For each cube $C' \in RC$, map outputs a $(\mathsf{cid}_1 \odot \mathsf{cid}_2 \odot \mathsf{flag}, (\mathsf{p}, \mathsf{flag}))$ pair with a composite key and the point itself as value. The first two components of the key identify the two involved cubes using textual cube ids: $\mathsf{cid}_1 = \min\{C(\omega).id, C'.id\}$ and $\mathsf{cid}_2 = \max\{C(\omega).id, C'.id\}$. The flag indicates whether ω belongs to the first or to the second cube. The repartitioning of the output key-value pairs is done by applying a hash function on the first two key components. This assigns all points of $C(\omega) \cup C'$ to the same reduce task. All key-value pairs are sorted by their complete keys. Finally, the reduce function is invoked on all values whose first two key components are equal. In reduce, the actual distance computation takes place. Due to the sorting, it is ensured that all points of the cube with the smaller cube id are processed first allowing for an efficient



(a) $\mathcal{HR}^3$ with Load Balancing for MR

	Cube index	Partition Π_0	Partition Π_1
C_0	(2463,10273)	1	0
C_1	(2468,10226)	1	0
C_2	(2469,10269)	1	0
C_3	(2464,10265)	1	0
C_4	(2475,10267)	2	2
C_5	(2482,10262)	0	1
C_6	(2482,10261)	0	1
C_7	(2478,10256)	0	1

Cubes

(b) CPM

Fig. 4. Overview of the MR-based $\mathcal{HR}^3$ implementation with load balancing (left) and the cube population matrix for the example dataset with $m = 2$ (right)

comparison of points of different cubes. The pseudo-code of the $\mathcal{HR}^3$ implementation is shown in Algorithms 1 and 2.

The described approach has two major drawbacks. First, a map task operates only on a fraction of the input data without global knowledge about the overall data distribution. Thus, each point is replicated and repartitioned $|RC|$ times, independently of whether there are points in the related cubes or not. Second, this approach is vulnerable to data skew, i.e., due to the inherent quadratic time complexity varying cube sizes can lead to severe load imbalances of the reduce tasks. Depending on the problem size and the granularity of the space tiling, the scalability of the described approach might be limited to a few nodes only. We provide an advanced approach that addresses these drawbacks in the next section.

4.2 $\mathcal{HR}^3$ with Load Balancing

The advanced approach borrows ideas from the load balancing approaches for Entity Resolution presented in [12]. An overview is shown in 4(a). The overall idea is to schedule a light-weight analysis MR job that linearly scans the input data in parallel and collects global data statistics. The second MR job utilizes these statistics for a data-driven redistribution of points ensuring evenly loaded reduce tasks.

Data Analysis Job. The first job calculates the cube index of each point in the map phase and sums up the number of points per (non-empty) cube in reduce. The output is a cube population matrix (CPM) of size $c \times m$ that specifies the number of points of c cubes across m input partitions. For our running example, an analysis job with $m = 2$ map tasks would read data from two input partitions Π_0 and Π_1 (cf. table in Figure 1) and produce the CPM shown in 4(b).

Distance Computation Job. The second MR job is based on the same number of map tasks and the same partitioning of the input data. At initialization, each map task reads the CPM. Similar to the basic approach, the reduce function processes pairs of related cubes. Because the CPM allows for an easy identification of empty cubes, the number of intermediate key-value pairs can be reduced significantly. As an example, for point

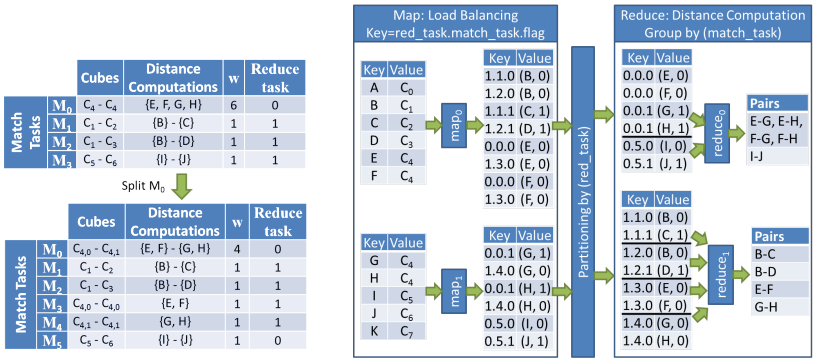


Fig. 5. Match task creation and reduce task assignment with/without splitting of large tasks (left). Example data flow for second MR job (right)

B of the running example, the map function of the basic approach would output 77 key-value pairs. With the knowledge encoded in the CPM, this can be reduced to two pairs only, i.e., for computing B's distances to the points C and D, respectively.

Before processing the first input point, each map tasks constructs a list of so-called match tasks. A match task is a triple (C_i, C_j, w), where C_i, C_j are two related cubes and $w = |C_i| \cdot |C_j|$ ($w = |C_i| \cdot (|C_i| - 1)/2$ for $i = j$) is the corresponding workload. The overall workload W is the sum of the workload of all match tasks. To determine each match task's target reduce task, the list is sorted in descending order of the workload. In this order, match tasks are assigned to the r reduce tasks following a greedy heuristic, i.e., the current match task is assigned to the reduce task with the currently lowest overall workload. The resulting match tasks are shown on the top of 5(a). Obviously, the reduce tasks are still unevenly loaded, because a major part of the overall workload is made up by the match task $C_4 - C_4$. To address this, for each large match task $M = (C_i, C_j, w)$ with $w > W/r$, both cubes are split according to their input partitioning into m subcubes. Consequently, M is split into a set of smaller subtasks, each comprising a pair of split subcubes before the sorting and reduce task assignment takes place. The bottom of 5(a) illustrates the splitting of the large match task $(C_4 - C_4)$. Because its workload $w = 6$ exceeds the average reduce task workload of $9/2 = 4.5$, C_4 is split into two subcubes $C_{4,0}$ (containing E, F) and $C_{4,1}$ (containing G, H). This results in three subtasks $(C_{4,0}, C_{4,0}, 1)$, $(C_{4,1}, C_{4,1}, 1)$, and $(C_{4,0}, C_{4,1}, 4)$ that recompose the original match task. Thus, both reduce tasks compute approximately the same number of distances indicating a good load balancing for the example.

After the initial match task creation, map task i builds an index that maps a cube to a set of corresponding match tasks. Thereby, only cubes of whom the input partition i actually contains points, need to be considered. For each input point ω and each match task of the cube $C(\omega)$, the map function outputs a $(\mathsf{red\_task} \odot \mathsf{match\_task} \odot \mathsf{flag}, (\omega, \mathsf{flag}))$ pair. Again, the flag indicates to which of the match task's (possibly split) cubes ω belongs to. The partitioning is only based on the reduce task index. The sorting is per-

Dataset	Source	Size	Features
DS_1	DBPedia	25,781	min/medium/max elevation
DS_2	DBPedia	475,000	latitude, longitude
DS_3	Linked Geo Data	500,000	latitude, longitude
DS_4	Linked Geo Data	6,000,000	latitude, longitude

Fig. 6. Datasets used for evaluation

formed on the entire key, whereas the grouping is done by match task index. 5(b) illustrates the dataflow for the running example. Note, that due to the enumeration of the match tasks and the sorting behavior, it is ensured that the largest match tasks are processed first. This makes it unlikely that larger delays occur at the end of the computation when most nodes are already idle.

5 Evaluation

The aim of our evaluation was to discover break-even points for the use of parallel processor, GPU and cloud implementations of LD algorithms. For this purpose, we compared the runtimes of the implementations of $\mathcal{HR}^3$ presented in the previous sections on four data sets within two series of experiments. The goal of the first series of experiment was to compare the performance of the approaches for link discovery problems of common size. Thereafter, we carried out a scalability evaluation on a large dataset to detect break-even points of the implementations. In the following, we present the datasets we used as well as the results achieved by the different implementations.

5.1 Experimental Setup

We utilized the four datasets of different sizes shown in Figure 6. The small dataset DS_1 contains place instances having three elevation features. The medium-sized datasets DS_2 and DS_3 contain instances with geographic coordinates. For the scalability experiment we used the large dataset DS_3 and varied its size up to $6 \cdot 10^6$. Throughout all experiments we considered the Euclidean distance. Given the spectrum of implementations at hand, we ran our experiments on three different platforms. The *CPU experiments* (Java, $Java_2$, $Java_4$, $Java_8$ for 1, 2, 4 and 8 cores) were carried out on a 32-core server running JDK 1.7 on Linux 10.04. The processors were 8 quad core AMD Opteron 6128 clocked at 2.0 GHz. The *GPU experiments* (GPU) were performed on an average consumer workstation. The GPU was a AMD Radeon 7870 GPU with 20 compute units, each of which has the ability to schedule up to 64 parallel hardware threads. The host program was executed on a Linux workstation running Ubuntu 12.10 and AMD APP SDK 2.8. The machine had an Intel Core i7 3770 CPU and 8 GB of RAM. All C++ code was compiled with gcc 4.7.2. Given that C++ and Java are optimized differently, we also ran the Java code on this machine and computed a runtime ratio that allowed our results to remain compatible. The *MapReduce experiments* (basic: MR, load balanced: MR_l)

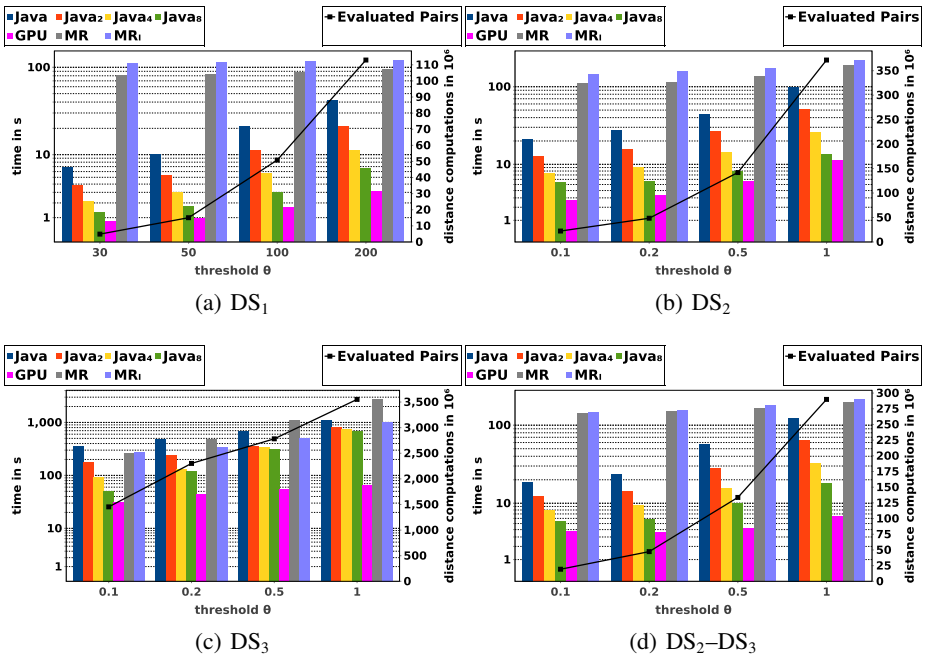


Fig. 7. Comparison of runtimes for Experiment 1

were performed with the *Dedoop prototype* [11] on Amazon EC2 in EU-west location. For the first experiment we used 10 nodes of type c1.medium (2 virtual cores, 1.7 GB memory). For the large data set we employed 20 nodes of type c1.xlarge (8 virtual cores, 7 GB memory).

5.2 Performance Comparison

The results of our performance comparison are shown in Figure 7. While the parallel implementation of $\mathcal{HR}^3$ on CPUs scales linearly for uniformly distributed data, the considerable skew in the DS_3 data led to the 8-core version being only 1.6 times faster than the mono-core implementation with a threshold of 1°. This impressively demonstrates the need for load balancing in all parallel link discovery tasks on skewed data. This need is further justified by the results achieved by MR and MR_l on DS_3. Here, MR_l clearly outperforms MR and is up to 2.7 times faster. Still, the most important result of this series of experiments becomes evident after taking a look at the GPU and Java runtimes on the workstation.

Most importantly, the massively parallel implementation outperforms all other implementations significantly. Especially, the GPU implementation outperforms the MR and MR_l by one to two orders of magnitude. Even the $Java_8$ implementation is outperformed by up to one order of magnitude. The performance boost of the GPU is partly due to the different hardware used in the experiments. To measure the effect of the

hardware, we ran the server Java program also on the workstation. A comparison of the runtimes achieved during this rerun shows that the workstation is between 2.16 and 7.36 times faster than the server. Still, our results suggests that our massively parallel implementation can make an effective use of the underlying architecture to outperform all other implementations in the indexing phase. The added efficient implementation of float operations for the distance computation in C++ leads to an overall superior performance of the GPU. Here, the results can be regarded as conclusive with respect to MR and MR_l and clearly suggest the use of local parallelism when dealing with small to average-sized link discovery problems.

The key observation that leads to conclusive results when comparing GPU and CPU results is that the generation of the cube index required between 29.3% (DS_1, $\theta = 50m$) and 74.5% (DS_3, $\theta = 1°$) of the total runtime of the algorithm during the deduplication tasks. Consequently, while running a parallel implementation on the CPU is advisable for small datasets with small thresholds for which the index computation makes up a small percentage of the total computation, running the approach on medium-sized datasets or with larger thresholds should be carried out on the GPU. This conclusion is yet only valid as long as the index fits into the memory of the GPU, which is in most cases 4 to 8 times smaller than the main memory of workstations. Medium-sized link discovery tasks that do not fit in the GPU memory should indeed be carried out on the CPUs. Our experiments suggest a break-even point between CPU and GPU for result set sizes around 10^8 pairs for 2-dimensional data. For higher-dimensional data where the index computation is more expensive, the break-even point is reached even for problems smaller than DS_1.

5.3 Scalability: Data Size

The strengths of the cloud are revealed in the second series of experiments we performed (see Figure 8). While the DFS and data transfer overhead dominates the total runtime of the LD tasks on the small datasets, running the scalability experiments on 20 nodes reveals that for tasks which generate more than 12 billion pairs as output, MR_l outperforms our local Java implementation. Moreover, we ran further experiments with more than 20 nodes on the 6 million data items. Due to its good scalability, the cloud implementation achieves the runtime of the GPU or performs even better for more nodes, e.g., for 30 (50) nodes MR_l requires approx. 32min (23min). It is important to remember here that the GPU implementation runs the comparisons in the CPU(s). Thus, the above suggested break-even point will clearly be reached for even smaller dataset sizes with more complex similarity measures such as the Levenshtein distance or the trigram similarity. Overall, our results hint towards the use of local massively parallel hardware being sufficient for a large number of link discovery tasks that seemed to require cloud infrastructures. Especially, numeric datasets can be easily processed locally as they require less memory than datasets in which strings play the central role. Still, for LD tasks whose intermediate results go beyond 10^{10} pairs, the use of the cloud still remains the most practicable solution. The clue for deciding which approach to use lies in having an accurate approximation function for the size of the intermediate results. $\mathcal{HR}^3$ provides such a function and can ensure that it can achieve an approximation below or equal to

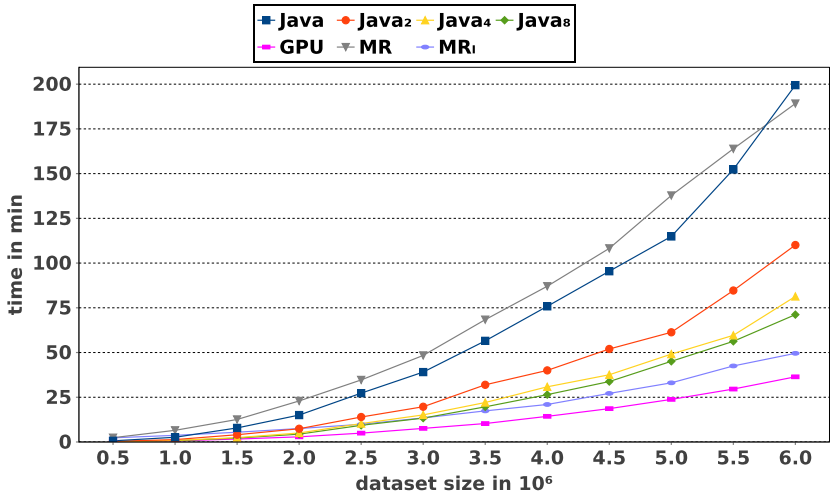


Fig. 8. Comparison of runtimes on DS4

any possible error margin. Providing such guarantees for other algorithms would thus allow deciding effectively and conclusively when to reach for the cloud.

6 Related Work

Link discovery has become an important area of research over the last few years. Herein, we present a brief overview of existing approaches.[7] Overall, the two main problems time complexity and generation of link specifications have been at the core of the research on LD.

With regard to *time complexity*, time-efficient string comparison algorithms such as *PPJoin+* [26], *EDJoin* [25] that were developed for deduplication were integrated into several link discovery frameworks such as *LIMES* [18]. Moreover, dedicated time-efficient approaches were developed for LD. For example in [19], an approach based on the Cauchy-Schwarz inequality is presented. The approaches HYPPO [16] and $\mathcal{HR}^3$ [17] rely on space tiling in spaces with measures that can be split into independent measures across the dimensions of the problem at hand. Especially, $\mathcal{HR}^3$ was shown to be the first approach that can achieve a relative reduction ratio r' less or equal to any given relative reduction ratio $r > 1$. Standard blocking approaches were implemented in the first versions of SILK and later replaced with *MultiBlock* [9], a lossless multi-dimensional blocking technique. *KnoFuss* [22] also implements blocking techniques to achieve acceptable runtimes. Further LD frameworks have been participated in the ontology alignment evaluation initiative [4].

With regard to the *generation of link specifications*, some unsupervised techniques were newly developed (see, e.g., [22]), but most of the approaches developed so far

[7] See [17,10] for more extensive presentations of the state of the art.

abide by the paradigm of supervised machine learning. For example, the approach presented in [8] relies on large amounts of training data to detect accurate link specification using genetic programming. *RAVEN* [20] is (to the best of our knowledge) the first active learning technique for LD. The approach was implemented for linear or Boolean classifiers and shown to require a small number of queries to achieve high accuracy. Later, approaches combining active learning and genetic programming for LD were developed [10,21].

The entity resolution (ER) problem (see [14,3] for surveys) shares many similarities with link discovery. The MR programming model has been successfully applied for both ER and LD. [23] proposes a MR implementation of the *PPJoin+* algorithm for large datasets. A first application for MR-based duplicate detection was presented in [24]. In addition, [7] as well as *Silk MapReduce*[8] implement MR approaches for LD. Several MR implementations for blocking-based ER approaches have been investigated so far. An MR implementation of the popular sorted neighborhood strategy is presented in [13]. Load balancing for clustering-based similarity computation with MR was considered in [12]. The ER framework *Dedoop* [11] allows to specify advanced ER strategies that are transformed to executable MR workflows and submitted to Hadoop clusters.

Load balancing and skew handling are well-known problems for parallel data processing but have only recently gained attention for MapReduce. *SkewTune* [15] is a generic load balancing approach that is invoked for a MapReduce job as soon as the first map (reduce) process becomes idle and no more map (reduce) tasks are pending. Then, the remaining keys (keygroups) of running tasks are tried to redistribute so that the capacity of the idle nodes is utilized. The approach in [5] is similar to our previous load balancing work [12] as it also relies on cardinality estimates determined during the map phase of the computation.

7 Conclusion and Future Work

In this paper, we presented a comparison of the runtimes of various implementations of the same link discovery approach on different types of parallel hardware. In particular, we compare parallel CPU, GPU and MR implementations of the $\mathcal{HR}^3$ algorithm. Our results show that the CPU implementation is most viable for two-dimensional problems whose result set size is in the order of 10^8. For higher-dimensional problems, massively parallel hardware preforms best even for problem with results set sizes in the order of 10^6. Cloud implementations become particularly viable as soon as the result set sizes reach the order of 10^{10}. Our results demonstrate that efficient resource management for link discovery demands the development of accurate approaches for determining the size of the intermediate results of link discovery frameworks. $\mathcal{HR}^3$ provides such a function. Thus, in future work, we will aim at developing such approximations for string-based algorithms. Moreover, we will apply the results presented herein to develop link discovery approaches that can make flexible use of the hardware landscape in which they are embedded.

[8] `https://www.assembla.com/spaces/silk/wiki/Silk_MapReduce`

References

1. Auer, S., Lehmann, J., Ngonga Ngomo, A.C.: Introduction to Linked Data and Its Lifecycle on the Web. In: Reasoning Web. pp. 1–75 (2011)
2. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Commun. ACM 51(1), 107–113 (2008)
3. Elmagarmid, A.K., Ipeirotis, P.G., Verykios, V.S.: Duplicate record detection: A survey. IEEE Trans. Knowl. Data Eng. 19(1), 1–16 (2007)
4. Euzenat, J., Ferrara, A., van Hage, W.R., et al.: Results of the Ontology Alignment Evaluation Initiative 2011. In: OM (2011)
5. Gufler, B., Augsten, N., Reiser, A., Kemper, A.: Load Balancing in MapReduce Based on Scalable Cardinality Estimates. In: ICDE. pp. 522–533 (2012)
6. Heino, N., Pan, J.Z.: RDFS Reasoning on Massively Parallel Hardware. In: ISWC. pp. 133–148 (2012)
7. Hillner, S., Ngonga Ngomo, A.C.: Parallelizing LIMES for large-scale link discovery. In: I-SEMANTICS. pp. 9–16 (2011)
8. Isele, R., Bizer, C.: Learning Linkage Rules using Genetic Programming. In: OM (2011)
9. Isele, R., Jentzsch, A., Bizer, C.: Efficient Multidimensional Blocking for Link Discovery without losing Recall. In: WebDB (2011)
10. Isele, R., Jentzsch, A., Bizer, C.: Active Learning of Expressive Linkage Rules for the Web of Data. In: ICWE. pp. 411–418 (2012)
11. Kolb, L., Thor, A., Rahm, E.: Dedoop: Efficient Deduplication with Hadoop. PVLDB 5(12), 1878–1881 (2012)
12. Kolb, L., Thor, A., Rahm, E.: Load Balancing for MapReduce-based Entity Resolution. In: ICDE. pp. 618–629 (2012)
13. Kolb, L., Thor, A., Rahm, E.: Multi-pass Sorted Neighborhood blocking with MapReduce. Computer Science - R&D 27(1), 45–63 (2012)
14. Köpcke, H., Rahm, E.: Frameworks for entity matching: A comparison. Data Knowl. Eng. 69(2), 197–210 (2010)
15. Kwon, Y., Balazinska, M., Howe, B., Rolia, J.A.: SkewTune: Mitigating Skew in MapReduce Applications. In: SIGMOD Conference. pp. 25–36 (2012)
16. Ngonga Ngomo, A.C.: A Time-Efficient Hybrid Approach to Link Discovery. In: OM (2011)
17. Ngonga Ngomo, A.C.: Link Discovery with Guaranteed Reduction Ratio in Affine Spaces with Minkowski Measures. In: ISWC. pp. 378–393 (2012)
18. Ngonga Ngomo, A.C.: On Link Discovery using a Hybrid Approach. Journal on Data Semantics 1, 203 – 217 (2012)
19. Ngonga Ngomo, A.C., Auer, S.: LIMES - A Time-Efficient Approach for Large-Scale Link Discovery on the Web of Data. In: IJCAI. pp. 2312–2317 (2011)
20. Ngonga Ngomo, A.C., Lehmann, J., Auer, S., Höffner, K.: RAVEN – Active Learning of Link Specifications. In: OM (2011)
21. Ngonga Ngomo, A.C., Lyko, K.: EAGLE: Efficient Active Learning of Link Specifications Using Genetic Programming. In: ESWC. pp. 149–163 (2012)
22. Nikolov, A., D'Aquin, M., Motta, E.: Unsupervised Learning of Data Linking Configuration. In: ESWC (2012)
23. Vernica, R., Carey, M.J., Li, C.: Efficient parallel set-similarity joins using mapreduce. In: SIGMOD Conference. pp. 495–506 (2010)
24. Wang, C., Wang, J., Lin, X., et al.: MapDupReducer: Detecting Near Duplicates over Massive Datasets. In: SIGMOD Conference. pp. 1119–1122 (2010)
25. Xiao, C., Wang, W., Lin, X.: Ed-Join: an efficient algorithm for similarity joins with edit distance constraints. PVLDB 1(1), 933–944 (2008)
26. Xiao, C., Wang, W., Lin, X., Yu, J.X.: Efficient similarity joins for near duplicate detection. In: WWW. pp. 131–140 (2008)