An Optimization Approach for Load Balancing in Parallel Link Discovery

Mohamed Ahmed Sherif
University of Leipzig
Augustusplatz 10
04109 Leipzig, Germany
sherif@informatik.uni-leipzig.de

Axel-Cyrille Ngonga Ngomo
Augustusplatz 10
04109 Leipzig, Germany
ngonga@informatik.uni-leipzig.de

ABSTRACT
Many of the available RDF datasets describe millions of resources by using billions of triples. Consequently, millions of links can potentially exist among such datasets. While parallel implementations of link discovery approaches have been developed in the past, load balancing approaches for local implementations of link discovery algorithms have been paid little attention to. In this paper, we thus present a novel load balancing technique for link discovery on parallel hardware based on particle-swarm optimization. We combine this approach with the Orchid algorithm for geo-spatial linking and evaluate it on real and artificial datasets. Our evaluation suggests that while naïve approaches can be super-linear on small data sets, our deterministic particle swarm optimization outperforms both naïve and classical load balancing approaches such as greedy load balancing on large datasets.

1. INTRODUCTION
With the constant growth of Linked Data sources over the last years comes the need to develop highly scalable algorithms for the discovery of links between data sources. While several architectures can be used to this end, previous works suggest that approaches based on local hardware resources suffer less from the data transfer bottleneck [18] and can thus achieve significantly better runtime than parallel approaches which rely on remote hardware (e.g., cloud-based approaches [13]). Moreover, previous works also suggest that load balancing (also called task assignment [20]) plays a key role in getting approaches for Link Discovery (LD) to scale. However, load balancing approaches for local parallel LD algorithms have been paid little attention to so far. In particular, mostly naïve implementations of parallel LD algorithms have been integrated into commonly used LD framework such as SILK [8] and LIMES [16].

The load balancing problem, which is know to be NP-complete [20], can be regarded as follows: Given n tasks $\tau_1, ..., \tau_n$ of known computational complexity (also called cost) $c(\tau_1), ..., c(\tau_n)$ as well as m processors, distribute the tasks τ_i across the m processors as evenly as possible, i.e., in such a way that there is no other distribution which would lead to a smaller discrepancy from a perfectly even distribution of tasks. Consider for example 3 tasks τ_1, τ_2 respectively τ_3 with computation complexities 3, 4 resp. 6. An optimal distribution of these tasks amongst two processors would consist of assigning τ_1 and τ_2 to the one of the processor (total costs: 7) and task τ_3 to the other processor (total costs: 6). No other task distribution leads to a more balanced load of tasks.

SEMANTiCS '15, September 15-17, 2015, Vienna, Austria
© 2015 ACM. ISBN 978-1-4503-3462-4/15/09 ...$15.00.
DOI: http://dx.doi.org/10.1145/2814864.2814872

In this paper, we address the research gap of load balancing for link discovering by first introducing the link discovery as well as the load balancing problems formally. We then introduce a set of heuristics for addressing this problem, including a novel heuristic dubbed *deterministic particle swarm optimization* (DPSO). This novel heuristic employs the basic insights behind particle swarm optimization (PSO) to determine a load balancing for link discovery tasks in a deterministic manner. Our approach is generic and can be combined with any link discovery approach that can divide the LD problem into a set of tasks within only portions of the input datasets are compared, including methods based on blocking (e.g., *Multiblock* [8]) and on space tiling (e.g., [17]). We evaluate our approach on both synthetic and real data.

2. PRELIMINARIES
The formal specification of LD adopted herein is akin to that proposed in [16]: Given two sets S respectively T of source respectively target resources as well as a relation R, our goal is to find the set $M \subseteq S \times T$ of pairs $(s, t) \in S \times T$ such that $R(s, t)$. If R is `owl:sameAs`, then we are faced with a *deduplication task*. Given that the explicit computation of M is usually a demanding endeavor, M is usually approximated by a set $\tilde{M} = \{(s, t, \delta(s, t)) \in S \times T \times \mathbb{R}^+ : \delta(s, t) \leq \theta\}$, where δ is a distance function and $\theta \geq 0$ is a distance threshold. For example, geo-spatial resources s and t are described by using single points or (ordered) sets of points, which we regard as polygons. Most algorithms for LD achieve scalability by first dividing S respectively T into non-empty subsets $S_1 \ldots S_k$ resp. $T_1 \ldots T_l$ such that $\bigcup\limits_{i=1}^{k} S_i = S$ and $\bigcup\limits_{j=1}^{l} T_j = T$. Note that the different subsets of S respectively T can overlap. In a second step, most time-efficient algorithms determine pairs of subsets (S_i, T_j) whose elements are to be compared. All elements (s, t) of all Cartesian products $S_i \times T_j$ are finally compared by means of the measure δ and only those with $\delta(s, t) \leq \theta$ are written into M'.

The idea behind load balancing for LD is to distribute the computation of δ over the Cartesian products $S_i \times T_j$ over n processors. We call running δ through a Cartesian product $S_i \times T_j$ a *task*. The set of all tasks assigned to a single processor is called a *block*. The cost $c(\tau)$ of the task τ is given by $c(S_i \times T_j) = |S_i| \cdot |T_j|$ while the cost of a block B is the sum of the cost of all its elements, i.e.,

$c(B) = \sum_{t \in B} c(\tau)$. Finding an optimal load balancing is known to be NP-hard. Hence, we refrain from trying to find a perfect solution in this paper. Rather, we aim to provide a heuristic that (1) achieves a good assignment of tasks to processors while (2) remaining computationally cheap. We measure the quality of an assignment by measuring the mean squared error (MSE) to a potentially existing perfect solution. Let $B_1, ..., B_m$ be the blocks assigned to our m processors. Then, the MSE is given by

$$\sum_{i=1}^{m} \left| c(B_i) - \sum_{j=1}^{m} \frac{c(B_j)}{m} \right|^2. \qquad (1)$$

It is obvious that there might not be a solution with an MSE of 0. For example, the best possible MSE when distributing the 3 tasks τ_1, τ_2 respectively τ_3 with computation complexities 3, 4 respectively 6 over 2 processors is 0.5.

3. LOAD BALANCING ALGORITHMS

The main idea behind load balancing techniques is to utilize parallel processing to distribute the tasks necessary to generate the solution to a problem across several processing units. Throughput maximization, response time minimization and resources overloading avoidance are the main purposes of any load balancing technique. We devised, implemented and evaluated five different load balancing approaches for linking geo-spatial datasets.

In each of the following algorithms, as input, we assume having a set $\mathcal{T}$ of n tasks and a set of m processors. Through each of the algorithms, we try to achieve load balancing among the m processors by creating a list of balanced task blocks $\mathcal{B} = \{B_1, ..., B_m\}$ with size m, where each processor p_i will be assigned its respective block B_i.

In order to ease the explanation of the following load balancing algorithms, we introduce a simple running example where we assume having a set of four tasks $\{\tau^7, \tau^1, \tau^8, \tau^3\}$ where the superscript of the task stands for its computational cost. Moreover, we assume having two processing units p_1 and p_2. The goal of our running example is to find two balanced tasks blocks B_1 and B_2 to be assigned to p_1 respectively p_2. In the following, we present different approaches for load balancing.

3.1 Naïve Load Balancer

The idea behind the naïve load balancer is to divide all tasks between all processors based on their index and regardless of their complexity. Each task with the index i is assigned to the processor with index $((i + 1) \mod m) + 1$. Hence, each of the m processors is assigned at most $\left\lceil \frac{n}{m} \right\rceil$ tasks. Algorithm 1 shows the pseudocode of our implementation of a naïve load balancing approach in which tasks are assigned to processors in the order of the input set. Applying the naïve load balancer to our running example we get $B_1 = \{\tau^7, \tau^8\}$, $B_2 = \{\tau^1, \tau^3\}$ and MSE = 30.25.

3.2 Greedy Load Balancer

The main idea behind the greedy load balancing [5] technique is to sort the input tasks in descending order based on their complexity. Then, starting from the most complex task, the greedy load balancer assigns tasks to processors in order. This approach is basically a heuristic that aims at achieving an even distribution of the total task complexity over all processors. The pseudo code of the greedy load balancer technique in presented in algorithm 2. Back to our running example, the greedy load balancer first sorts the example tasks (line 2) to be $\{\tau^8, \tau^7, \tau^3, \tau^1\}$. Then, in order, the tasks are assigned to the task blocks (line 4) to have $B_1 = \{\tau^8, \tau^3\}$, $B_2 = \{\tau^7, \tau^1\}$ with MSE = 2.25.

Algorithm 1: Naïve Load Balancer

```
input : $\mathcal{T} \leftarrow \{\tau_1, ..., \tau_n\}$ : set of tasks of size $n$
        $m$ : number of processors
output: $\mathcal{B} \leftarrow \{B_1, ..., B_m\}$ : a partition of $\mathcal{T}$ to a list of $m$ blocks of
        tasks
1 $i \leftarrow 1$;
2 foreach task $\tau$ in $\mathcal{T}$ do
3     addTaskToBlock($\tau, B_i$);
4     $i \leftarrow (i \mod m) + 1$;
5 return $\mathcal{B}$;
```

Algorithm 2: Greedy Load Balancer

```
input : $\mathcal{T} \leftarrow \{\tau_1, ..., \tau_n\}$ : set of tasks of size $n$
        $m$ : number of processors
output: $\mathcal{B} \leftarrow \{B_1, ..., B_m\}$ : a partition of $\mathcal{T}$ to a list of $m$ blocks of
        balanced tasks
1 $\mathcal{T} \leftarrow$ descendingSortTasksByComplexity($\mathcal{T}$);
2 $i \leftarrow 1$;
3 foreach task $\tau$ in $\mathcal{T}$ do
4     addTaskToBlock($\tau, B_i$);
5     $i \leftarrow (i \mod m) + 1$;
6 return $\mathcal{B}$;
```

3.3 Pair-Based Load Balancer

The pair-based load balancing [14] is reminiscent of a two-way breadth-first-search. The approach assigns processors tasks in pairs of the form *(most complex, least complex)*. In order to get most homogeneous pairs, the algorithm first sorts all input tasks according to tasks' complexities. Afterwards, from the sorted list of tasks, the pair based algorithm generates the $\lceil \frac{n}{2} \rceil$ pairs of tasks where the pair i is computed by selecting i^{th} and $(n - i + 1)^{th}$ tasks from the sorted list of tasks. The pseudo-code of the pair based technique is shown in Algorithm 3.

The pair-based load balancer starts dealing with our running example tasks by sorting them to be $\{\tau^1, \tau^3, \tau^7, \tau^8\}$ (line 1). Afterwards, the algorithm generates tasks pairs as (first, forth) and (second, third) to have $B_1 = \{\tau^1, \tau^8\}$, $B_2 = \{\tau^3, \tau^7\}$ with MSE = 0.25.

Algorithm 3: Pair Based Load Balancer

```
input : $\mathcal{T} \leftarrow \{\tau_1, ..., \tau_n\}$ : set of tasks of size $n$
        $m$ : number of processors
output: $\mathcal{B} \leftarrow \{B_1, ..., B_m\}$ : a partition of $\mathcal{T}$ to a list of $m$ blocks of
        balanced tasks
1 $\mathcal{T} \leftarrow$ sortTasksByComplexity($\mathcal{T}$);
2 $i \leftarrow 1$;
3 for $i \leq \lceil n/2 \rceil$ do
4     addTaskToBlock($\tau_i, B_i$);
5     addTaskToBlock($\tau_{n-i+1}, B_i$);
6     $i \leftarrow i + 1$;
7 return $\mathcal{B}$;
```

3.4 Particle Swarm Optimization

The particle swarm optimization (PSO) [12][10][11] is a population-based stochastic algorithm. PSO is based on social psychological principles. Unlike evolutionary algorithms, in a typical PSO, there is no selection of individuals, all population members (dubbed as particles) survive from the beginning to the end of a algorithm. At the beginning of PSO, particles are randomly initialized in the problem solution space. Over successive iterations, particles cooperatively interact to improve of the fitness of the optimization problem solutions. PSO is normally used for continuous problems but that it has been extended to deal with discrete problems [22][4] such as the one at hand.

In order to model our problem in terms of the PSO technique, we consider the input tasks $\mathcal{T}$ as the *particles*[1] to be optimized. The aim here is to to balance the size of the blocks (i.e., the total complexity of tasks included in each block) as well as possible. To adapt the idea of the PSO to load balancing, we define the *fitness function* as the task complexity difference between the most overloaded task block and least underloaded task block. Formally, The PSO fitness function is defined as

$$F = c(B^{+}) - c(B^{-}), \tag{2}$$

where $B^{+} = arg \max_{B \in \mathcal{B}} c(B)$ and $B^{-} = arg \min_{B \in \mathcal{B}} c(B)$ are the most and least loaded blocks respectively, and $\mathcal{B}$ is the list of all task blocks.

Initially, the PSO based load balancing approach starts like the naïve approach (see Algorithm 1). All particles are distributed equally into the task blocks regardless of tasks' complexities, i.e., each block now contains at most $\lceil \frac{n}{2} \rceil$ particles. We dubbed the task block list as *Best Known Positions* (BKP). Afterwards, PSO computes the fitness function to the initial BKP and saves it as *Best Known Fitness* (*BKF*). Until a termination criterion is met, in each iteration, PSO performs the *particles migration* process. This process consists of first assigning a random velocity v to each particle p included in a block B_i, where $v \in \mathbb{N}$ and $0 \leq v \leq m$. If $v \neq i$, p is moved to the new block B_v, otherwise p stays in its block B_i. After moving all the particles, the PSO computes the new fitness F. If the new fitness F is less than *BKF*, PSO updates both BKF and BKP.

Note that the termination criteria can be defined independently of the core PSO algorithm. Here, we implemented two termination criteria: (1) *minimum fitness threshold* and (2) *maximum number of iterations*. If the minimum fitness threshold is reached in any iteration the algorithm terminates instantly and the BKP is returned. Otherwise, the BKF is returned after reaching the maximal number of iterations. The pseudo-code of the PSO load balancing technique in presented in algorithm 4.

Back to our running example, assume we set the maximal number of iterations to 1 (**I** = 1). First, the PSO initializes $B_1 = \{\tau^7, \tau^8\}$, $B_2 = \{\tau^1, \tau^3\}$ (lines 3–5) and the best known Fitness $F = 11$ (lines 9). Then, the PSO clones B_1 and B_2 to B_1^* respectively B_2^* (line 12). Assume that PSO generates random velocity $v = 1$ for τ^7 (line 16). Then, τ^7 stays in its current block B_1^*. For τ^8, assume $v = 2$ (which is different than τ^8's block B_1^*), then τ_8 migrates to B_1^* (line 18). For τ^1 and τ^3 assume $v = 1$ which make both τ^1 and τ^3 stays in B_2^*. Consequently, we have $B_1^* = \{\tau^7\}$, $B_2^* = \{\tau^1, \tau^3, \tau^8\}$ with the new fitness function $F^* = 5$ (line 21) and as $F^* < F$ then $\mathcal{B}$ and

Algorithm 4: Particle Swarm Optimization Load Balancer

```
input  : $\mathcal{T} \leftarrow \{\tau_1, ..., \tau_n\}$ : set of tasks of size $n$
         $m$ : number of processors
         **F** : fitness function threshold (zero by default)
         **I** : number of iterations
output: $\mathcal{B} \leftarrow \{B_1, ..., B_m\}$ : the Best Known Particles' positions as
         a list of $m$ blocks of balanced tasks
1  Initialize Particles' Best known Position $\mathcal{B}$
2  $i \leftarrow 1$;
3  foreach task $\tau$ in $\mathcal{T}$ do
4      addTaskToBlock($\tau, B_i$);
5      $i \leftarrow (i \mod m) + 1$;
6  Initialize best known Fitness $F$;
7  $B^{+} \leftarrow$ getMostOverloadedBlock($\mathcal{B}$);
8  $B^{-} \leftarrow$ getLeastUnderloadedBlock($\mathcal{B}$);
9  $F \leftarrow c(B^{+}) - c(B^{-})$;
10 $i \leftarrow 1$;
11 while $i < $ I do
12     $\mathcal{B}^* \leftarrow \mathcal{B}$;
13     Move each task $\tau$ (particle) to new position based on a
       random particle velocity $v$
14     foreach block $B^* \in \mathcal{B}^*$ do
15         foreach particle $\tau \in B^*$ do
16             $v \leftarrow$ generateRandomVelocity($0, m$);
17             if $B_v^* \neq B^*$ then
18                 migrateParticleToBlock($\tau$, $B_v^*$);
19                 If better fitness achieved update result
                   $B^{*+} \leftarrow$ getMostOverloadedBlock($\mathcal{B}^*$);
20                 $B^{*-} \leftarrow$ getLeastUnderloadedBlock($\mathcal{B}^*$);
21                 $F^* \leftarrow c(B^{*+}) - c(B^{*-})$;
22                 if $F^* < F$ then
23                     $F \leftarrow F^*$;
24                     $\mathcal{B} \leftarrow \mathcal{B}^*$;
25                 if $F ==$ F then
26                     return $\mathcal{B}$;
27     $i \leftarrow i + 1$;
28 return $\mathcal{B}$;
```

F are updated by $\mathcal{B}^*$ respectively F^* (lines 22–24). The PSO terminates as it is reached the maximum number of iterations (line 11) and returns $B_1 = \{\tau^7\}$, $B_2 = \{\tau^1, \tau^3, \tau^8\}$ with MSE = 6.25.

3.5 DPSO Load Balancer

The PSO load balancer (see section 3.4) has a main drawback of being an *indeterminism* approach. This drawback is inherited from the fact that the PSO is a *heuristic* algorithm that depends up on a random selection of velocity for moving particles. In order to overcome this drawback, we propose the *Deterministic PSO* (DPSO).

The DPSO starts in the same way as the PSO by partitioning all the n tasks to m task blocks, where m equals the number of processors. In this stage, each block contains at most $\lceil n/m \rceil$ tasks regardless of tasks' complexities. Until a termination criterion is met, in each iteration the DPSO:

1. Finds the most overloaded block $B^{+} = arg \max_{B \in \mathcal{B}} c(B)$ and the

[1]In the rest of the paper we will use the terms *tasks* and *particles* interchangeably.

least underloaded block $B^- = arg \min_{B \in \mathcal{B}} c(B)$, where $\mathcal{B}$ is the list of all task blocks.

2. Sort tasks within B^+ based in their complexities.

3. As far as a better balancing between B^+ to B^- is met, DPSO perform *task migration*, where DPSO moves task per task in order from B^+ to B^-.

4. Compute *fitness function* as $c(B^+) - c(B^-)$.

Here, We implement two termination criteria akin with the ones defined previously for PSO: (1) *minimum fitness threshold* (**F**) and (2) *maximum number of iterations* (**I**). The pseudo code of the DPSO load balancing algorithm in presented in algorithm algorithm 5. Note that the termination criteria can be defined independently of the core DPSO algorithm. For instance, fitness function convergence could be considered as the termination criterion.

The deterministic nature of DPSO comes from the fact that (1) DPSO only moves tasks from most overloaded block B^+ to the lest underloaded block B^-, i.e. no random particles migration as in PSO, (2) DPSO sorts B^+ tasks before it starts the task migration process. Sorting insures migration of smaller tasks first in which away an optimal load balancing between most and least loaded blocks is achieved in each iteration.

Assume we apply the DPSO for our running example for one iteration ($\mathbf{I} = 1$). First, the DPSO initializes $B_1 = \{\tau^7, \tau^8\}$, $B_2 = \{\tau^1, \tau^3\}$ (lines 2–5). Then, DPSO sorts tasks within the most overloaded block B^+ which is B_1 (line 8) to be $\{\tau^7, \tau^8\}$ (line 11). Consequently, the DPSO migrates τ^7 from $B^+ = B^1$ to $B^- = B^1$ (line 13). Finally, as the DPSO finds that $c(B^+) < c(B^-)$ it breaks (line 14) and returns the result $B_1 = \{\tau^8\}$, $B_2 = \{\tau^1, \tau^3, \tau^7\}$ with MSE = 2.25.

Algorithm 5: DPSO Load Balancer

```
input : $\mathcal{T} \leftarrow \{\tau_0, ..., \tau_n\}$ : set of tasks of size $n$
        $m$ : number of processors
        $\mathbf{F}$ : fitness function threshold (zero by default)
        $\mathbf{I}$ : number of iterations
output: $\mathcal{B} \leftarrow \{B_0, ..., B_m\}$ : the Best Known Particles' positions as
        a list of $m$ blocks of balanced tasks
1  Initialize Particles' Best known Position $\mathcal{B}$
2  $i \leftarrow 1$;
3  foreach task $\tau$ in $\mathcal{T}$ do
4      addTaskToBlock($\tau, B_i$);
5      $i \leftarrow (i \mod m) + 1$;
6  $i \leftarrow 1$;
7  while $i < \mathbf{I}$ do
8      $B^+ \leftarrow$ getMostOverloadedBlock($\mathcal{B}$);
9      $B^- \leftarrow$ getLeastUnderloadedBlock($\mathcal{B}$);
10     Balance $B^+$ and $B^+$ by migrating particles (tasks) from sorted
       $B^+$ to $B^-$
11     $B^+ \leftarrow$ sortTasksByComplexity($B^+$);
12     foreach particle $\tau \in B^+$ do
13         migrateParticleToBlock($\tau, B^-$);
14         if $c(B^+) < c(B^-)$ then
15             break;
16         Compute fitness function $F$ as the complexity difference
           between most and least loaded tasks
17         $F \leftarrow c(B^+) - c(B^-)$;
18         if $F == \mathbf{F}$ then
19             return $\mathcal{B}$;
20     $i \leftarrow i + 1$;
21 return $\mathcal{B}$;
```

4. EVALUATION

The aim of our evaluation was to quantify how well DPSO outperforms traditional load balancing approaches (i.e. naïve, greedy and pair-based). To this end, we measured the runtime for each of the five load balancing algorithms for both synthetic and real data. In the following, we begin by presenting the algorithm and data that we used. Thereafter, we present our results on different datasets.

4.1 Experimental Setup

For our experiments, the parallel task generation was based on the Orchid approach [17]. The idea behind Orchid is to improve the runtime of algorithms for measuring geo-spatial distances by adapting an approach akin to divide-and-conquer. Orchid assumes that it is given a distance measure δ. Thus all pairs in the mapping M that it returns must abide by $\delta(s, t) \leq \theta$. Overall, Orchid begins by partitioning the surface of the planet. Then, the approach defines a *task* as comparing the points in a given partition with only the points in partitions that abide by the distance threshold θ underlying the computation. A task is the comparison of all points in two partitions.

We performed controlled experiments on five synthetic geographic datasets[2] and three real datasets. The synthetic datasets were created by randomly generating a number of polygons ranging between 1 and 5 million polygons in steps of 1 million. We varied the synthetic dataset polygons' sizes from one to ten points. The variation of sizes of polygons was based on a *Gaussian* random distribution. Also, the (latitude, longitude) coordinates of each point are generated akin with the *Gaussian* distribution.

We used three publicly available datasets for our experiments as real datasets. The first dataset is the *Nuts*[3]. We chose this dataset because it contains fine-granular descriptions of 1,461 geo-spatial resources located in Europe. For example, Norway is described by 1981 points. The second dataset, *DBpedia*[4], contains all the 731,922 entries from DBpedia that possess geometry entries. We chose DBpedia because it is commonly used in the Semantic Web community. Finally, the third dataset was *LinkedGeoData*, contains all 3,836,119 geo-spatial objects from `http://linkgeodata.org` that are instances of the class `Way`.[5]. Further details to the datasets can be found in [17].

All experiments were carried out on a 64-core server running *OpenJDK* 64-Bit Server 1.6.0_27 on *Ubuntu* 12.04.2 LTS. The processors were 8 quad-core *Intel(R) Core(TM) i7-3770 CPU @ 3.40 GHz* with 8192 KB cache. Unless stated otherwise, each experiment was

[2]All synthetic dataset are available at `https://github.com/AKSW/LIMES/tree/master/evaluationsResults/lb4ld`

[3]Version 0.91 available at `http://nuts.geovocab.org/data/` is used in this work

[4]We used version 3.8 as available at `http://dbpedia.org/Datasets`.

[5]We used the RelevantWays dataset (version of April 26th, 2011) of *LinkedGeoData* as available at `http://linkedgeodata.org/Datasets`.

assigned 20 GB of memory. Because of the random nature of the PSO approach we ran it 5 times in each experiment and provide the mean of the five runs' results. The approaches presented herein were implemented in the LIMES framework.[6] All results are available at the project web site.[7]

4.2 ORCHID vs. Parallel ORCHID

We began by evaluating the *speedup* gained by using parallel implementations of ORCHID algorithm. To this end, we first ran experiments on the three real datasets (Nuts, DBpedia and LinkedGeoData). First, we computed the runtime of the normal (i.e., non-parallel) implementation of ORCHID [17]. Then, we evaluated the parallel implementations of ORCHID using the aforementioned five load balancing approaches. To evaluate the *speedup* gained from increasing the number of parallel processing units, we reran each of the parallel experiments with 2, 4 and 8 threads. Figure 1 shows the runtime results along with the mean squared error (MSE) results of the experiments.

Our results show that the parallel ORCHID implementations using both PSO and DPSO outperform the normal ORCHID on the three real datasets. Particularity, when dealing with small dataset like *NUTS* (see Figure 1 (a)), PSO and DPSO achieve up to three times faster than the non-parallel version of ORCHID. When dealing with larger dataset such as *LinkedGeoData* (see Figure 1 (e)), PSO and DPSO are capable of achieving up to ten times faster than the non-parallel version of ORCHID. This fact shows that our load balancing heuristics deployed in PSO and DPSO are capable of achieving *superlinear* performance [1, 2] when ran on two processors. This is simply due to the processor cache being significantly faster than RAM and thus allowing faster access to data and therewith also smaller runtimes. On the other side, greedy and pair-based load balancing fail to achieve even the run time of the normal ORCHID. This fact is due to the significant amount of time required by greedy and pair-based load balancing algorithms for sorting tasks prior to assigning them to processors.

4.3 Parallel Load balancing Algorithms Evaluation

We performed this set of experiments with two goals in mind: First, we wanted to measure the run time taken by each algorithm when applied to different datasets. Our second aim was to qualify the quality of the data distribution carried out by each of the implemented algorithm using MSE. To this end, we ran two sets of experiments. In the first set of experiments, we used the aforementioned three datasets of *Nuts*, *DBpedia* and *LinkedGeoData*. The result of this set of experiments are presented in Figure 1. In the second set of experiments, we ran our five load balancing algorithms against a set of five synthetic randomly generated datasets (see subsection 4.1 for details). The results are presented in Figure 2.

Our results suggest that DPSO and PSO outperform the naïve approach in most cases. This can be seen most clearly in Figure 2 (note the log scale). DPSO is to be preferred over PSO as it is deterministic and is thus the default implementation of load balancing currently implemented in LIMES. Still, the improvements suggest that preserving the integrity of the hypercubes generated by ORCHID still leads to a high difference in load across the processors as shown by our MSE results. An interesting research avenue would thus be to study approaches which do not preserve this integrity while guaranteeing result completeness. This will be the core of our future work.

[6]`http://limes.sf.net`
[7]`https://github.com/AKSW/LIMES/tree/master/evaluationsResults/lb4ld`

5. RELATED WORK

Load balancing techniques have been applied in a large number of disciplines that deal with big data. For handling massive graphs such as the ones generated by social networks, [21] introduces two message reduction techniques for distributed graph computation load balancing. For dealing with federated queries, [3] proposes an RDF query routing index that permits a better load balancing for distributed query processing. [14] proposes two approaches for load balancing for the complex problem of entity resolution (ER), which utilize a preprocessing *MapReduce* job to analyze the data distribution. [13] demonstrates a tool called *Dedoop* for MapReduce-based ER of large datasets that implements similar load balancing techniques. A comparative study for different load balancing algorithms for MapReduce environment is presented in [7].

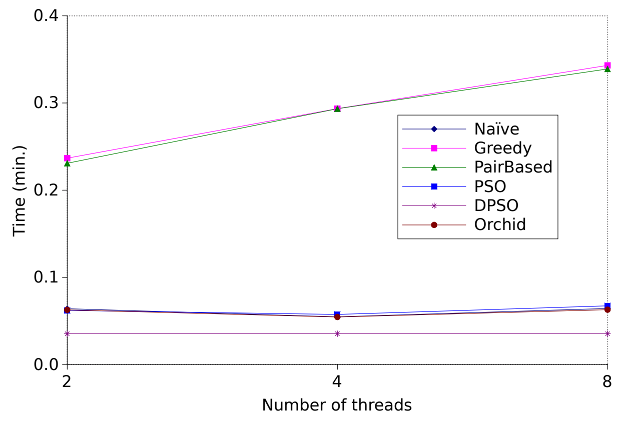
Finding an optimal load balancing is known to be NP-complete. Thus, [15] provides two heuristics for distributed grid load balancing, one is based on *ant-colony optimization* and the other is based on *particle-swarm optimization*. Yet, another heuristic (dubbed as BPSO) is proposed by [9]. BPSO is a modified *binary particle-swarm optimization* for network reconfiguration load balancing. [19] proposes an artificial bee-colony-based load balancing algorithm for cloud computing.

The study [1] introduces *superlinear* performance analyses of real-time parallel computation. The study shows that parallel computers with *n* processors can solve a computational problem more than *n* times faster than a sequential one. In another work [2], the *superlinear* performance concluded to be also possible for parallel evolutionary algorithms both theoretically and in practice.

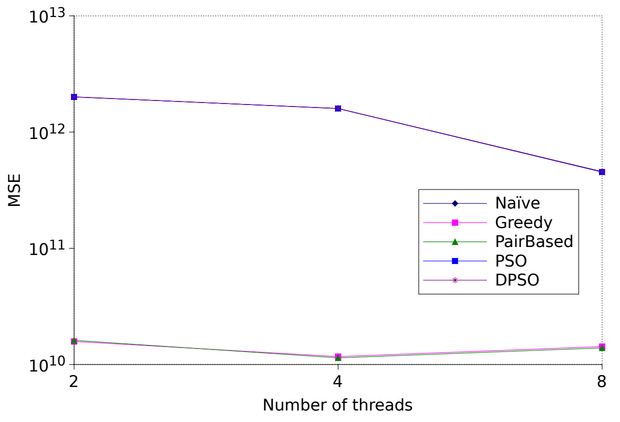
6. CONCLUSION AND FUTURE WORK

In this paper, we presented and evaluated load balancing techniques for link discovery. In particular, in the PSO approach, we applied *particle-swarm optimization* to optimize the distribution of tasks of different sizes over a given number of processors. While the PSO outperforms classical load balancing algorithms, it has the main drawback of being indeterministic in nature. Therefore, we proposed the DPSO, where we altered the task selection of PSO for ensuring deterministic load balancing of tasks. We combined the aforementioned approaches with the ORCHID algorithm. All the implemented load balancing approaches are evaluated on real and artificial datasets. Our evaluation suggests that while naïve approaches can be super-linear on small data sets, our deterministic particle swarm optimization outperforms both naïve and classical load balancing approach such as greedy load balancing on large datasets as well as a datasets which originate from highly skewed distributions.

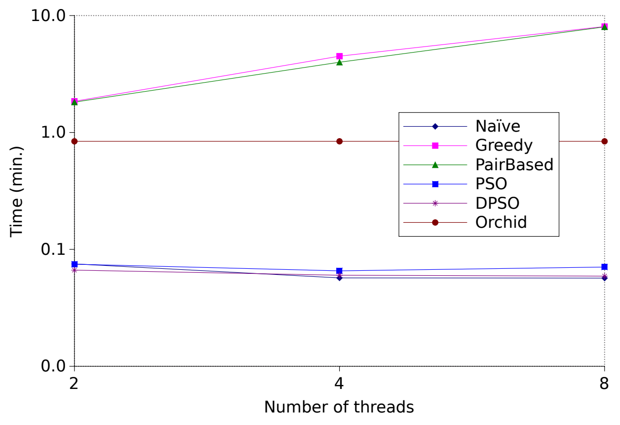
Although we achieve reasonable results in terms of scalability, we plan to further improve the time efficiency of our approaches by enabling the splitting of one task over more than one processor. As an extension of DPSO, we plan to implement a caching technique, which enables DPSO to be used on larger datasets that can not be fitted in memory [6]. While DPSO was evaluated in combination with ORCHID in this paper, we will study the combination of our approach with other space tiling and/or blocking algorithms for generating parallel tasks.



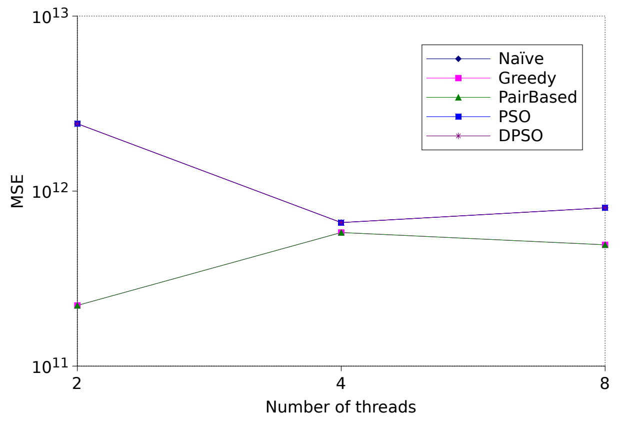
(a) Nuts runtime



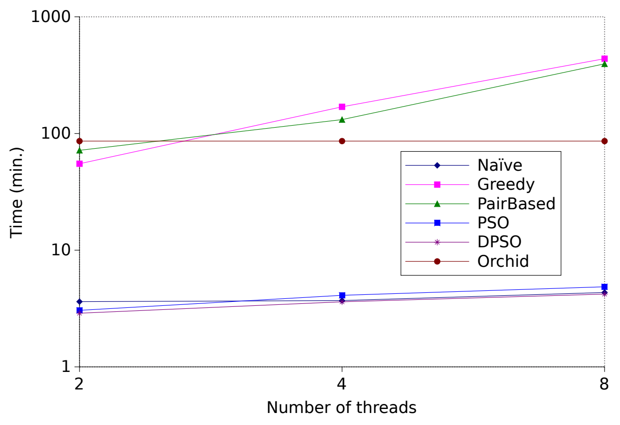
(b) Nuts MSE



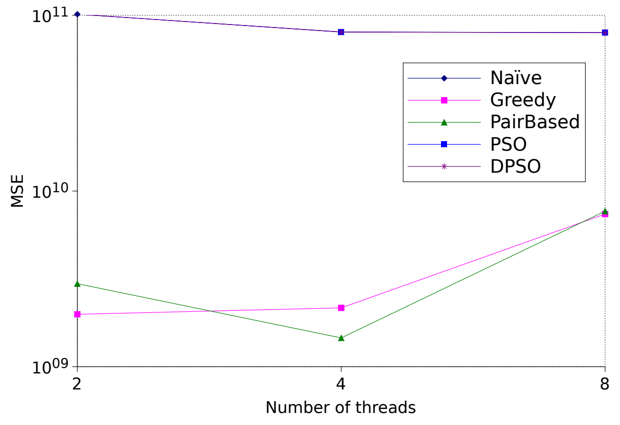
(c) DBpedia runtime



(d) DBpedia MSE

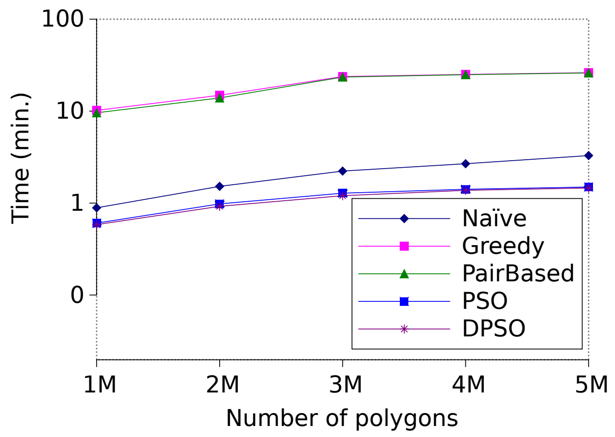


(e) LinkedGeoData runtime

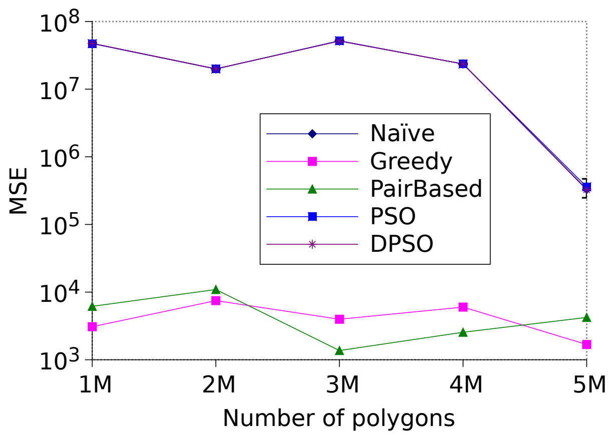


(f) LinkedGeoData MSE

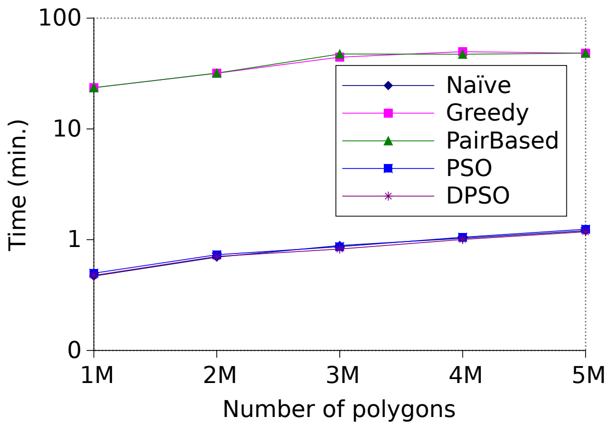
Figure 1: Runtime and MSE generated when applying ORCHID [17] vs. parallel implementations of ORCHID using naïve, greedy, pair based, PSO and DPSO load balancing algorithms against the three real datasets of *Nuts*, *DBpedia* and *LinkedGeoData* using $2, 4$ **and** 8 **threads**



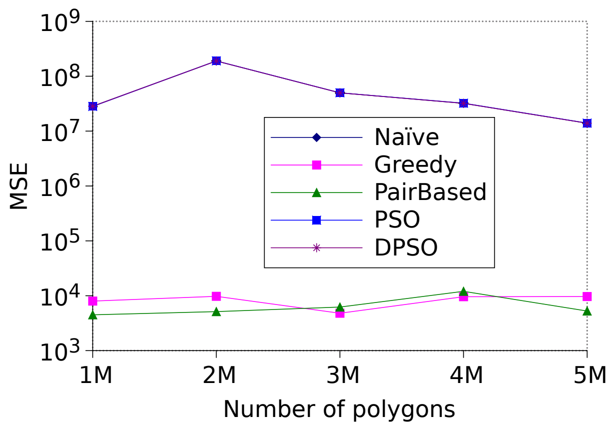
(a) 2 threads runtime



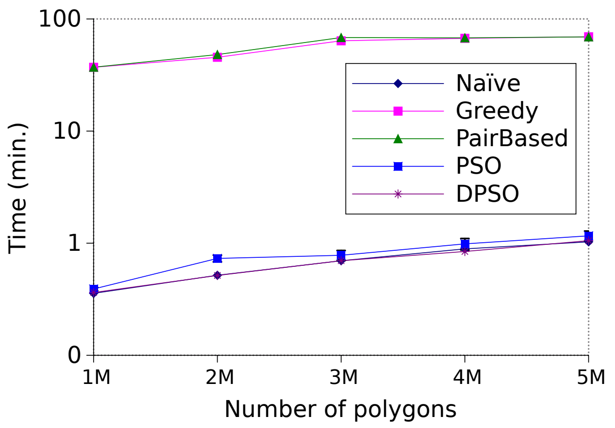
(b) 2 threads MSE



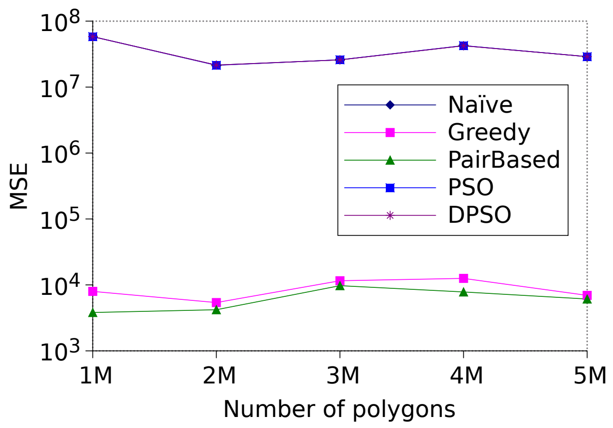
(c) 4 threads runtime



(d) 4 threads MSE



(e) 8 threads runtime



(f) 8 threads MSE

Figure 2: Runtime and MSE generated when applying parallel implementations of ORCHID using naïve, greedy, pair based, PSO and DPSO load balancing algorithms against the five synthetic datasets of sizes $1, 2, 3, 4$ and 5 million polygons using $2, 4$ and 8 threads

Acknowledgments

Parts of this work were supported by the *GeoKnow* project (GA 318159) and the BMWi project *SAKE*.

7. REFERENCES

[1] Selim G Akl. Superlinear performance in real-time parallel computation. *The Journal of Supercomputing*, 29(1):89–111, 2004.

[2] Enrique Alba. Parallel evolutionary algorithms can achieve super-linear performance. *Information Processing Letters*, 82(1):7–13, 2002.

[3] Liaquat Ali, Thomas Janson, and Christian Schindelhauer. Towards load balancing and parallelizing of rdf query processing in p2p based distributed rdf data stores. In *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, pages 307–311. IEEE, 2014.

[4] Qing Cai, Maoguo Gong, Lijia Ma, Shasha Ruan, Fuyan Yuan, and Licheng Jiao. Greedy discrete particle swarm optimization for large-scale social network clustering. *Information Sciences*, 2014.

[5] Ioannis Caragiannis, Michele Flammini, Christos Kaklamanis, Panagiotis Kanellopoulos, and Luca Moscardelli. Tight bounds for selfish and greedy load balancing. *Algorithmica*, 61(3):606–637, 2011.

[6] Mofeed M. Hassan, Renè Speck, and Axel-Cyrille Ngonga Ngomo. Using caching for local link discovery on large data sets. In *Engineering the Web in the Big Data Era*, volume 9114 of *Lecture Notes in Computer Science*, pages 344–354. Springer International Publishing, 2015.

[7] Hesham A Hefny, Mohamed Helmy Khafagy, and Ahmed M Wahdan. Comparative study load balance algorithms for map reduce environment. *International Journal of Computer Applications*, 106(18):41–50, 2014.

[8] Robert Isele, Anja Jentzsch, and Christian Bizer. Efficient Multidimensional Blocking for Link Discovery without losing Recall. In *WebDB*, 2011.

[9] Xiaoling Jin, Jianguo Zhao, Ying Sun, Kejun Li, and Boqin Zhang. Distribution network reconfiguration for load balancing using binary particle swarm optimization. In *Power System Technology, 2004. PowerCon 2004. 2004 International Conference on*, volume 1, pages 507–510. IEEE, 2004.

[10] A Kaveh. Particle swarm optimization. In *Advances in Metaheuristic Algorithms for Optimal Design of Structures*, pages 9–40. Springer, 2014.

[11] James Kennedy. Particle swarm optimization. In *Encyclopedia of Machine Learning*, pages 760–766. Springer, 2010.

[12] Serkan Kiranyaz, Turker Ince, and Moncef Gabbouj. Particle swarm optimization. In *Multidimensional Particle Swarm Optimization for Machine Learning and Pattern Recognition*, pages 45–82. Springer, 2014.

[13] Lars Kolb and Erhard Rahm. Parallel entity resolution with dedoop. *Datenbank-Spektrum*, 13(1):23–32, 2013.

[14] Lars Kolb, Andreas Thor, and Erhard Rahm. Load balancing for mapreduce-based entity resolution. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 618–629. IEEE, 2012.

[15] Simone A Ludwig and Azin Moallem. Swarm intelligence approaches for grid load balancing. *Journal of Grid Computing*, 9(3):279–301, 2011.

[16] Axel-Cyrille Ngonga Ngomo. On link discovery using a hybrid approach. *J. Data Semantics*, 1(4):203–217, 2012.

[17] Axel-Cyrille Ngonga Ngomo. Orchid - reduction-ratio-optimal computation of geo-spatial distances for link discovery. In *Proceedings of ISWC 2013*, 2013.

[18] Axel-Cyrille Ngonga Ngomo, Lars Kolb, Norman Heino, Michael Hartung, Sören Auer, and Erhard Rahm. When to reach for the cloud: Using parallel hardware for link discovery. In *Proceedings of ESCW*, 2013.

[19] Jeng-Shyang Pan, Haibin Wang, Hongnan Zhao, and Linlin Tang. Interaction artificial bee colony based load balance method in cloud computing. In *Genetic and Evolutionary Computing*, pages 49–57. Springer, 2015.

[20] Ayed Salman, Imtiaz Ahmad, and Sabah Al-Madani. Particle swarm optimization for task assignment problem. *Microprocessors and Microsystems*, 26(8):363–371, 2002.

[21] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. Effective techniques for message reduction and load balancing in distributed graph computation. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, pages 1307–1317, Republic and Canton of Geneva, Switzerland, 2015. International World Wide Web Conferences Steering Committee.

[22] Wen-liang Zhong, Jun Zhang, and Wei-neng Chen. A novel discrete particle swarm optimization to solve traveling salesman problem. In *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*, pages 3283–3287. IEEE, 2007.