Parallelizing LIMES for Large-Scale Link Discovery

Stanley Hillner
itemis AG
Ludwig-Erhard-Strasse 51
04103 Leipzig, Germany
stanley.hillner@itemis.de

Axel-Cyrille Ngonga Ngomo
AKSW Research Group
University of Leipzig
Johannisgasse 26/5-22
04103 Leipzig, Germany
ngonga@informatik.uni-leipzig.de

ABSTRACT

The Linked Open Data cloud consists of more than 26 billion triples, of which less than 3% are links between knowledge bases. However, such links play a central role in key tasks such as cross-ontology question answering, large-scale inferencing and link-based traversal query execution models. The mere size of the Linked Data Cloud makes manual linking impossible. Consequently, Link Discovery Frameworks have been developed over the last years with the aim of providing means to detect links between knowledge bases automatically. Yet, even the current runtime-optimized frameworks for linking lead to unacceptable runtimes when presented with very large datasets. This paper addresses the time complexity of Link Discovery on very large datasets by presenting and evaluating the parallelization of the time-optimized LIMES framework by means of the MapReduce paradigm.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

Keywords

Link Discovery, Linked Data Web, Parallel Computing

1. INTRODUCTION

The Linked Data Web has evolved from 12 knowledge bases in May 2007 to 203 knowledge bases in September 2010, i.e., in less than four years [7]. Currently, the Linked Open Data cloud (LOD) consists of more than 26 billion triples, of which less than 3% are links between knowledge bases. Yet, links between knowledge bases play a central role in tasks such as cross-ontology question answering [10] and large-scale inferences [11].

Due to the mere size of the LOD cloud, the manual linking of entities is a time-consuming process that would result in a minute subset of all possible links. For examples, manually checking the films in DBpedia [2] for duplicates would require approximately 2.4 billion comparisons to complete. Assuming that a human could compare 1 pair of entities per second, he would still need more than 70 years to carry out such a task. Link Discovery Frameworks have been developed over the last years to mitigate this problem. Yet, even runtime-optimized approaches such as SILK [16] or LIMES [12] still require several million comparisons to complete the task above.

Two main categories of remedies can be envisaged to mitigate the time complexity of Link Discovery: developing highly efficient algorithms and distributing the computation effort across several machines. While the first remedy is implemented in [12], this paper focuses on a minimally invasive extension of this given algorithm from a single-core version to the parallelized framework for Link Discovery dubbed LIMES M/R. For this purpose we first study the requirements to an effective parallelization of the framework at hand before presenting our approach to parallelizing LIMES. We then evaluate the runtime of our approach against the single-core version of LIMES and the parallelized version of SILK in ten settings and show that LIMES M/R achieves a speedup of up to 11 and that we therewith outperform the parallelized version of SILK significantly while achieving exactly the same results as LIMES.

The remainder of this paper is structured as follows: In Section 2 we give a brief overview of related work. Section 3 presents parallelization paradigms and models out of which we chose the most suitable for implementing a parallel version of the LIMES approach. In the subsequent section, Section 4, we present the current implementation of LIMES M/R. Section 5 presents 10 experiments carried out with LIMES M/R and gives insights in the performance of our approach. Finally, we conclude in Section 6.

2. RELATED WORK

Link Discovery is a central task within the Linked Data setting. The most common approach to Link Discovery is based on comparing certain property values of the entities from a source knowledge base S and a target knowledge base T that are to be linked. A link is then generated if and only if the distance between these properties is below a given threshold θ. Several approaches have already been developed for discovering links between data sets. [12] points out that two main categories of Link Discovery frameworks can be differentiated: *domain-specific* and *universal* frameworks. Domain-specific LD frameworks aim at discovering links between knowledge bases from a particular domain such as

I-SEMANTICS 2011, 7th Int. Conf. on Semantic Systems, Sept. 7-9, 2011, Graz, Austria

universities and conferences (RKB-CRS, [6]) as well as music (GNAT, [14]).

Of higher relevance for this paper are universal LD frameworks, which are designed to carry out mapping tasks independently from the domain of the source and target knowledge bases and can thus be confronted with very large link discovery tasks. RDF-AI [15] implements a five-step approach that comprises the preprocessing, matching, fusion, interlinking and post-processing of data sets. SILK [16] (Version 2.2) implements blocking and rough index pre-matching to reach a quasi-linear time-complexity. In addition, the parallelized version of SILK runs on Hadoop, which significantly improves the total runtime of the system. The LIMES approach [12] presupposes that the datasets to link are in a metric space. It then uses the triangle inequality to partition the metric space. Each portion of the metric space is represented by a so-called exemplar, which is used as reference point to compute a pessimistic approximation of distances within that given portion of space. Based on these approximations, LIMES can discard a large number of similarity computations without loosing links. Although the LIMES framework displays significant runtime improvements, it still requires a considerable amount of time to carry out mappings of the order of 10^9 comparisons. In this paper, we show how the time-optimized approach implemented in LIMES can be rendered even faster via parallelization.

3. PARALLELIZATION PARADIGMS

Building programs for parallel computation has been a challenge since parallel computation was invented. Yet during the last decade, the rapid technological development on the hardware side has led to parallel programming becoming increasingly important in order to use the full potential, even of standard computers [1, 3]. In [9], two main categories of approaches for application parallelization are identified, namely *auto parallelization* an *parallel programming*.

Auto-parallelization approaches are realized mainly by parallel compilers and can be applied to automatically parallelize applications that were implemented to run linearly. Yet, previous work shows that automatically parallelized applications are limited in their degree of parallelism and thus lead to a limited speedup of the application [8].

To ensure a higher parallelism, *parallel programming* techniques need to be applied. There are various parallel programming models from which an application developer can choose [3, 4, 1]. [9] identified seven evaluation criteria for parallel programming models and evaluated six models against these criteria. In [1], five critical tasks during the development of parallel applications are named and ten programming models are evaluated against them. Out of these two sets of criteria, the following five are of central importance when parallelizing link discovery frameworks:

- **Task-to-Worker Mapping** defines which task shall be executed on which worker. Automatic (done by the runtime-system) and manual (carried out by the developer) worker mapping can be distinguished.
- **Worker Management** is concerned with the management of the processes. An automatic management means that the system controls the creation or termination of, for example, threads or processes. In manual management, the developer has to specify the life cycle of the workers manually.
- **Data Distribution** describes the task of distributing data over the parallel system to feed the workers with input data to be processed and collect the results from the workers. Again, one can distinguish automatic data distribution and manual data distribution where the developer has to manage the passing of data or the current state of the data.
- **Synchronization** is the task of managing the order of the workers which access shared data. Automatic synchronization indicates that the developer does not need to worry about synchronization details.
- **Programming Methodologies** defines how the parallelism is abstracted, as an example, the available API and the language specification. This can be a main factor in development time when a developer, for instance, uses a new programming model that does not offer an intuitive language.

In the following, we give a brief overview of four current parallel programming models and select the most accurate for the task at hand by using the criteria defined above.

3.1 Parallel programming models

3.1.1 OpenMP

Open Multi-Processing (OpenMP)[1] [13] is an open specification which extends C, C++ and Fortran with parallelization capabilities by offering a set of compiler directives and callable runtime libraries. In OpenMP, the workers are realized as threads whose management is implicit so that the developer does not need to manage their life cycles. Instead, OpenMP provides special compiler directives which indicate that a code section, e. g., a for-loop, is to be executed in parallel. The task-to-worker mapping is managed by OpenMP. Since OpenMP is a programming model for shared-memory architectures, the effort for distributing data between the workers is small. Each worker can access the data in the shared address space. Yet, shared memory renders the synchronization of the access of the shared data difficult. In OpenMP, the synchronization is done implicitly by several mechanisms and the programmer only has to declare sections where a synchronization is needed. OpenMP abstracts away the most critical parallelization tasks from the programmer, like the the workload partitioning or the synchronization mechanisms.

3.1.2 MPI

The Message Passing Interface (MPI)[2] [13] is a standard for inter-process communication in distributed computer systems. Developers can use MPI for the communication in distributed memory environments as well as shared memory computers. MPI has been implemented for languages such as C, C++, Fortran and Java[3]. In contrast to OpenMP, workers in MPI are instantiated as processes. Still, the worker management is mostly taken care of by the MPI runtime. All the application needs to specify is the number processes needed for the current task. The task-to-worker mapping, workload partitioning and data distribution have to be implemented explicitly. The synchronization in MPI

[1] `http://openmp.org/wp/`
[2] `http://www.mcs.anl.gov/research/projects/mpi/`
[3] `http://www.hpjava.org/mpiJava.html`

is carried out by the MPI engine but the developer still has to define where synchronized execution is needed.

3.1.3 UPC

The Unified Parallel C (UPC)[4] [5] programming language is an extension of the C programming language and is intended to be used for parallel programming on HPC machines. It supports both shared memory machines and distributed memory environments. In UPC, the workers are threads and the worker management is done implicitly. All the application needs to specify is the number of threads that are necessitated. The workload partitioning and task-to-worker mapping can be carried out implicitly or explicitly. The synchronization in UPC is more complex than in OpenMP or MPI and offers implicit and explicit synchronization constructs. UPC abstracts from the memory architecture and presents the memory as a partitioned global address space. Furthermore, the UPC-API and the language specification provides various mechanisms for implicit and explicit worker mapping or workload partitioning. This flexibility bears the obvious advantage that the developer can choose to handle single problems manually, e. g., the memory access synchronization, but does not have the need to.

3.1.4 MapReduce

MapReduce is another programming model for HPCs. The core idea of this programming model is based on the *Map* and *Reduce* primitives of functional programming languages. The workers in MapReduce are implemented as processes and managed by the MapReduce cluster. The developer does not have to care about the creation or destruction of workers but there are various options that can be used to configure the cluster, such as the number of map, combine, or reduce tasks. Furthermore, there is no need to specify a task-to-worker mapping during development time. MapReduce maps each task to an available worker and re-executes tasks on other workers if necessary. This ensures a very high fault tolerance and can speed-up the computation in case of slow workers being part of the cluster. The distribution of the input and the generated intermediate data can be carried out fully automatically, but there are also possibilities to influence the distribution over the cluster. MapReduce hides nearly every task of parallel programming by providing Interfaces against which the programmer can develop his application.

3.2 Comparison

Table 1 shows a comparison of the parallel programming models made above. Each criterion is rated as *automatic* if this task is fulfilled by the system, *manual* if the developer has to handle this task manually or *both* if the task is handled by the system but the programmer can choose to take over the task. An exception to the evaluation is the criterion of the programming methodologies where only positive and negative scorings are possible. There , a scoring of "++" means that the programming model hides most of the parallelization tasks while "-" indicates that the programmer has to carry out all parallelization tasks. Models marked with "++" enable the developer to get a quick start with the programming model and set the main focus to the development of the core features of the application.

[4] `http://upc.lbl.gov/`

Table 1 shows that the programming models OpenMP and MapReduce are best suited for parallelizing LIMES. Given that manual data synchronization is not critical, as the data for linking is partitioned and is consequently easy to synchronize, we chose to use MapReduce as it requires a minimal amount of effort for the cluster setup, is supported by a large number of service providers and can be executed on multi-core machines as well as HPCs without any adaptation.

4. LIMES M/R

In its stand-alone and not parallel version, LIMES implements the Link Discovery Process depicted in Figure 1(a) and explicated in [12]. In brief, LIMES begins by querying two knowledge bases S and T and caching their content with respect to constraints defined in the configuration specified by the user. Once the target cache is filled and the used metrics are initialized, LIMES reorganizes the target cache by assigning each instance $t \in T$ to exactly one so-called exemplar e and by storing the distance between t and e. Each exemplar is used to represent a portion T' of T. The actual matching process is then computed between S and all the T'. Thereby, each instance s of S is first compared with each exemplar e. Then, the triangle inequality is used to compute lower bounds of the distance between s and each $t \in T'$. These lower bounds are finally used to reduce the number of actual distance computations that are carried out.

It is obvious that several of these steps can be executed in parallel, for example, fetching the source and target data and filling the corresponding caches. Yet, these steps are usually not the bottleneck of Link Discovery Frameworks. Given a large-scale link discovery task, most of the runtime of any link discovery framework is spent computing the similarity between instances from the source and target knowledge base. Thus, we focus in this paper on parallelizing the similarity computation step. The resulting process is shown in Figure 1(b). Overall, the process implemented by LIMES M/R can be subdivided into two main steps: initialization and execution.

4.1 Initialization

LIMES M/R first checks the MapReduce configuration file that is passed in addition to the LIMES configuration file. If the configuration is valid, the next step of the process creates and configures the MapReduce job by using the information in the configuration file before scheduling the job for execution at the MapReduce framework. If the configuration file be invalid, then the link discovery process is aborted. Else, a job is scheduled for execution. Hadoop then starts the job by executing the MapReduce workflow. During this job execution, the execution of atomic activities switches between LIMES M/R and Hadoop several times as visualized by the swimlanes in Figure 1(b).

The MapReduce job takes care of the distribution of the program to the master and worker nodes. The job execution per se begins with the retrieval of the input splits that are later passed to the mappers of the job in order to find links between the input knowledge bases. This activity is executed on the master node of the cluster and is carried out as follows: First, the LIMES configuration file is validated against its DTD and read. If the validation fails, the framework stops the execution of the job and returns with an appropriate error message. In the other case, the

	Worker management	Task-to-worker mapping	Data distribution	Synchronization	Programming methodologies
OpenMP	automatic	automatic	automatic	automatic	++
MPI	automatic	manual	manual	automatic	+/-
UPC	automatic	both	automatic	both	+
MapReduce	automatic	automatic	automatic	manual	++

Table 1: Evaluation of parallel programming models

source and target knowledge bases are queried and instances matching the restrictions are extracted to the source and target caches. Furthermore, the underlying metrics to be used for the instance comparisons are initialized.

4.2 Execution

4.2.1 Computation of Exemplars

As soon as the target cache is filled and the metrics are initialized, LIMES M/R starts with the reorganization of the target cache as described in [12]. After the completion of the reorganization of the target cache, the computation of the input splits is started. Each input split consist of a subset of $S \cup T$ and is assigned to one of the map tasks of the cluster later in the process. The number and thus size of the input splits is thereby mainly influenced by the number of mappers of the job. The current implementation of LIMES M/R splits the source cache into a number of disjoint cache fragments which equals the number of mappers defined by the user. These source cache fragments are then used as the input keys for the map tasks of the job. We implemented two different variations of input split generators. The first implementation (LIMES M/R) uses the whole reorganized target cache as the value for each input key/value pair, while the second (LIMES M/R V2) splits the reorganized target cache into 10 fragments and uses these fragments as values for the input splits. In the latter case, the current implementation creates 10 times more map tasks than specified by the user.

4.2.2 Split processing

The input splits are processed within the MapReduce framework as depicted in the Hadoop Swimlane of Figure 1(b). The MapReduce framework then goes through a series of procedures. First, the map and reduce tasks are assigned to the worker nodes of the cluster. The implemented standard setting for the number of reducers that are used to join the intermediate key/value pairs coming from the mappers uses one reducer in order to join the mapper output into a single file lying in the distributed file system. In contrast, a MapReduce cluster usually runs a large number of map tasks with the aim at distributing the work over the whole cluster and ensuring a good work load balance. As stated in the previous section, the number of map tasks can be influenced by the user. However, Hadoop uses the generated input splits to feed the assigned map tasks with input data, but in contrast to most other MapReduce programs, LIMES M/R does not store its queried input data or the generated input splits in the distributed file system and passes their locations to the mappers. LIMES M/R transfers the generated input splits directly to the map tasks in order to build in-memory caches for the mappers. This is even for huge knowledge bases possible since the problem size is reduced significantly for the single map tasks by means of the input splitting. Next, Hadoop calls the `RecordReader` for each map task and input split that shall extract the single key/value pairs from the splits in order to be consumed by the mappers. The current implementation does not apply any optimizations to this step and simply creates the caches out of the passed binary input data coming from the input splits.

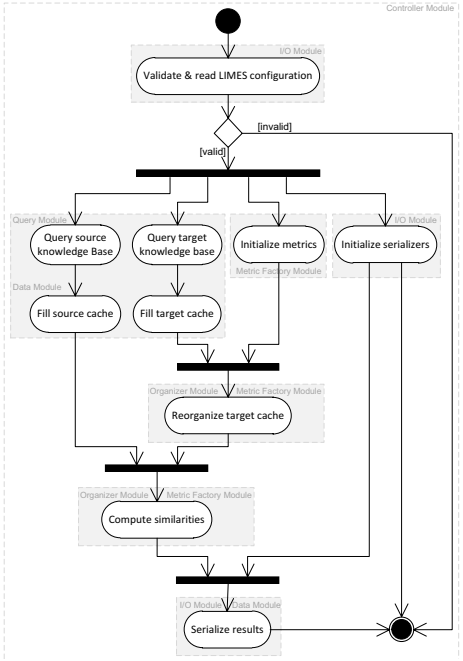
The next two steps are executed by LIMES M/R. The similarity computation process is executed according to the LIMES algorithm in each map task for the input split that has been passed to it. In short, each instance of the key cache (a fragment of the original source cache) is at first compared to each exemplar of the reorganized target cache or its fragment, depending on the version of LIMES M/R, as described for the input split generation. If the similarity lies above a given threshold and the triangle inequality-property of the metric space holds, the instance is compared to each instance in the subspace of the exemplar, as long as the inequality can be applied. As soon as the similarity computation for one instance of the source cache fragment is completed, the mapper emits the resulting key/value pairs as intermediate pairs to the Hadoop output collector which then writes the results to the local disks of the worker that is executing the current map task.

4.2.3 Reduction

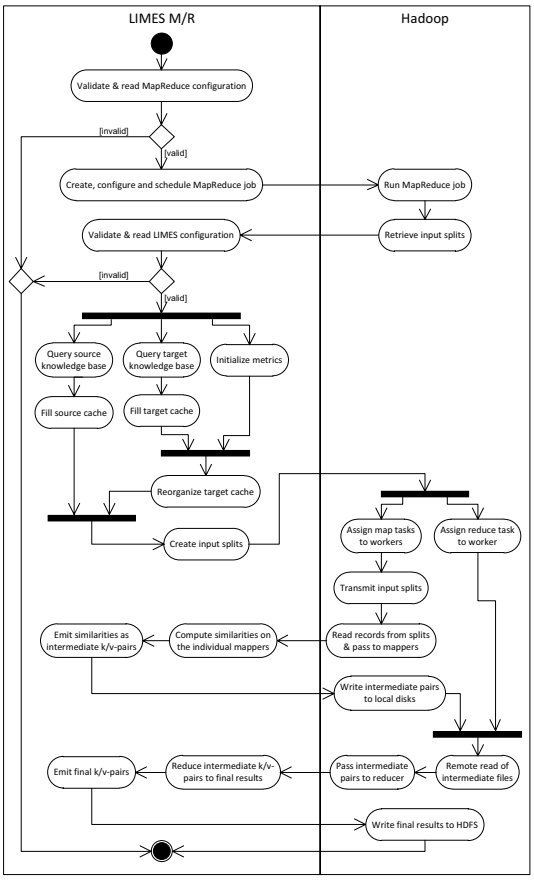
The reduce task is applied as soon as one map task produces intermediate key/value pairs. The job tracker sends the locations of the intermediate results in the distributed file system to the worker node that runs the reduce task of LIMES M/R which then reads these pairs from the file system using Remote Method Invocation (RMI). The above steps are all carried out internally by Hadoop. Once, the intermediate files are fetched, the reduce worker passes the intermediate key/value pairs to the reduce task where these pairs are reduced to the final output of the framework. The reducer of LIMES M/R has to execute just one simple task, the combination of the incoming intermediate results to just one final result file. To achieve this goal, the reducer simply passes all incoming pairs as plain text to the output collector of Hadoop which writes them into a single file on the distributed file system. If no error occurs during the link discovery process, the execution of the LIMES M/R job successfully terminates after the last map task has finished its similarity computation and the so-produced intermediate results are passed to the reducer, which merges all results to one final set of links.

5. EXPERIMENTS AND RESULTS

To ensure the correctness and the effectiveness of the approach presented in the previous section, we carried out ten experiments of different size.



(a) Activity diagram of LIMES



(b) Activity diagram of LIMES M/R

Figure 1: Internal Link Discovery Processes

5.1 Experimental Setup

The test cluster used in all experiments reported below was created by using the Amazon Elastic Compute Cloud (EC2) Web Service[5] and consisted of 9 computers[6] (nodes). One of these computers was exclusively used as the master node of the cluster and the other 8 nodes were designated as worker nodes of the cluster (the slaves). Since the master node was not intended to perform large computations, this node was launched as a standard small EC2 instance (*m1.small*), while the slave nodes were launched as high-CPU medium EC2 instances (*c1.medium*) in order to provide enough computation resources.

The small instance was assigned $1.7GB$ of memory and 1 EC2 Compute Unit which is provided by 1 virtual core with 1 EC2 Compute Unit. The high-CPU medium instances with $1.7GB$ of memory and up to 5 EC2 Compute Units which were provided by 2 virtual cores with 2.5 EC2 Compute Units each. Both instance types are built on a 32-bit platform architecture and provided a moderate I/O performance. The cluster nodes were launched with the 32-bit version of the Ubuntu 10.04 LTS (Lucid Lynx) Server Edition operating system. The MapReduce cluster was set up with Apache Hadoop Version 0.20.2. The cluster was configured to run at most 2 map tasks per node in parallel which results in a total number of 16 parallel map tasks for the whole cluster. This value was chosen due to the hardware restrictions of the cluster nodes. The Jave Virtual Machines of the slave nodes were allocated $1024MB$ of memory.

We carried out two series of experiments. In the first series experiments, we performed four link discovery tasks using different settings for linkage. The setup for this first series of experiments is summarized in Table 2. In the second series of experiments, we were concerned with measuring the effect of a change of threshold on our approach. Thus, we stuck with the same data sets and carried out a link discovery task with six different thresholds.

5.2 Results

The results of our first series of experiments are shown in Figure 2. Our results clearly show that we outperform both LIMES (single-core) and SILK (parallelized version running on 16 cores) with respect to runtime and reach a speedup of up to 11 on the Diseases dataset when running on 16 processors in parallel. Interestingly, LIMES M/R V2 leads to longer runtimes than LIMES M/R with 16 workers. Overall, we obtain the best speedup with 16 workers as awaited. Setting the numbers of workers to a value above 16 leads to a higher communication overhead between the master and the slaves as well as a higher effect of the network latency. The second series of experiments reveals that lowering the threshold leads to improved speedup values. This is simply due to the network overhead being constant for all experiments and the runtime being larger for lower similarity thresholds which is a result of the LIMES similarity computation approach. Consequently, the relative runtime decreases, leading to better speedup values. Note that in all experiments LIMES M/R achieved the same results as LIMES.

[5] `http://aws.amazon.com/ec2/`

[6] Amazon EC2 provides compute capacity on demand. Using this cloud-based service, it is possible to create virtual computer instances matching the personal requirements.

	Organisms	Diseases	Cities	Films
Source data set	DBpedia[7]	MeSH[8]	DBpedia	DBpedia
Target data set	STITCH[9]	LinkedCT[10]	DBpedia	DBpedia
Source instance class	`dbp-o:Species`	`meshr:Concept`	`dbp-o:City`	`dbp-o:Film`
Target instance class	`stitch:organisms`	`lct:condition`	`dbp-o:City`	`dbp-o:Film`
Source property	`rdfs:label`	`dc:title`	`rdfs:label`	`rdfs:label`
Target property	`rdfs:label`	`linkedct:condition_name`	`rdfs:label`	`rdfs:label`
|S|	146000	23600	12600	49000
|T|	100	5000	12600	49000
Complexity(|S| × |T|)	14.6×10^6	118×10^6	159×10^6	2.4×10^9
Threshold	0.9	0.75	0.95	0.95

Table 2: Summary of experimental setup

The runtime of LIMES could be improved by implementing a data portioning approach that makes better use of the distribution of data across the metric space. Currently, the exemplar computation is carried out without taking the skew within the data into consideration. Thus, it can lead to an unbalanced data distribution across the exemplars and thus to a sub-optimal distribution of the workload amongst the workers. A distribution-aware approach for exemplar computation still needs to be integrated in LIMES. Such an approach could then be used directly by LIMES M/R thank to the minimally invasive parallelization followed within this paper.

6. CONCLUSION AND FUTURE WORK

In this paper, we presented our extension of the LIMES framework using the Hadoop framework. We gave insights in the design principles and in the current implementation of LIMES M/R. We carry out ten experiments within which we compared LIMES, LIMES M/R and SILK. We showed that our implementation leads to speedup values up to 11 on 16 processors. In all experiments, we outperformed SILK ran in parallel on 16 processors.

In future work, we will aim to develop an exemplar computation approach that takes the skew in the data distribution into consideration and thus enable a better load balancing across the Hadoop slaves. Furthermore, we will investigate new algorithms for Link Discovery and aim to parallelize them so as to enable LIMES to run on problems of complexity beyond 10^9 in less than a minute.

7. REFERENCES

[1] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] Sören Auer, Chris Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. DBpedia: A nucleus for a web of open data. In *ISWC*, pages 722–735. Springer, 2008.

[3] Liang T. Chen and Deepankar Bairagi. Developing Parallel Programs – A Discussion of Popular Models. Technical report, Oracle Corporation, September 2010.

[4] Ali Ebnenasir and Rasoul Beik. Developing parallel programs: A design-oriented perspective. In *IWMSE '09*, pages 1–8, 2009.

[5] Tarek El-Ghazawi and Francois Cantonnet. Upc performance and potential: a npb experimental study. In *Supercomputing*, pages 1–26, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[6] Hugh Glaser, Ian C. Millard, Won-Kyung Sung, Seungwoo Lee, Pyung Kim, and Beom-Jong You. Research on linked data and co-reference resolution. Technical report, University of Southampton, 2009.

[7] Tom Heath and Christian Bizer. *Linked Data: Evolving the Web into a Global Data Space*. Morgan & Claypool, 2011.

[8] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, fourth edition, 2007.

[9] Henry Kasim, Verdi March, Rita Zhang, and Simon See. Survey on Parallel Programming Model. In *Network and Parallel Computing*, pages 266–275. Springer Berlin / Heidelberg, 2008.

[10] Vanessa Lopez, Victoria Uren, Marta Reka Sabou, and Enrico Motta. Cross ontology query answering on the semantic web: an initial evaluation. In *K-CAP*, pages 17–24, New York, NY, USA, 2009. ACM.

[11] James McCusker and Deborah McGuinness. Towards identity in linked data. In *Proceedings of OWL Experiences and Directions Seventh Annual Workshop*, 2010.

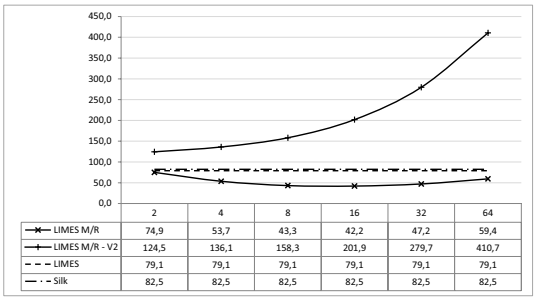
[12] Axel-Cyrille Ngonga Ngomo and Sören Auer. A time-efficient approach for large-scale link discovery on the web of data. In *IJCAI*, 2011.

[13] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003.

[14] Yves Raimond, Christopher Sutton, and Mark Sandler. Automatic interlinking of music datasets on the semantic web. In *Proceedings of the 1st Workshop about Linked Data on the Web*, 2008.

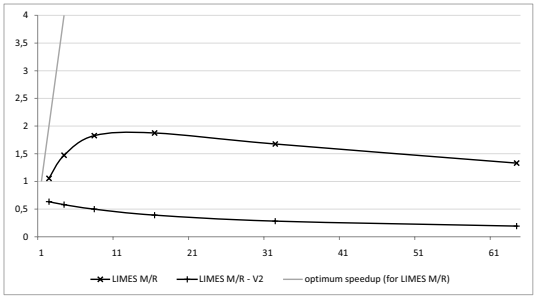
[15] François Scharffe, Yanbin Liu, and Chuguang Zhou. Rdf-ai: an architecture for rdf datasets matching, fusion and interlink. In *Proc. IJCAI 2009 workshop on Identity, reference, and knowledge representation (IR-KR), Pasadena (CA US)*, 2009.

[16] Julius Volz, Christian Bizer, Martin Gaedke, and Georgi Kobilarov. Discovering and maintaining links on the web of data. In *ISWC*, pages 650–665, 2009.

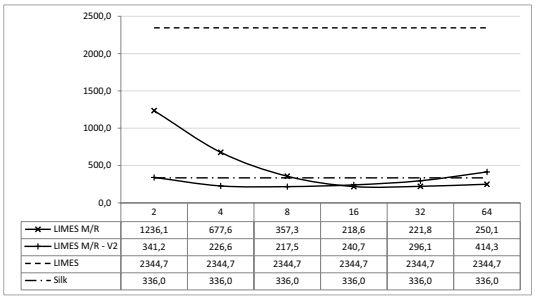


	2	4	8	16	32	64
LIMES M/R	74,9	53,7	43,3	42,2	47,2	59,4
LIMES M/R - V2	124,5	136,1	158,3	201,9	279,7	410,7
LIMES	79,1	79,1	79,1	79,1	79,1	79,1
Silk	82,5	82,5	82,5	82,5	82,5	82,5

(a) Runtimes – Organisms

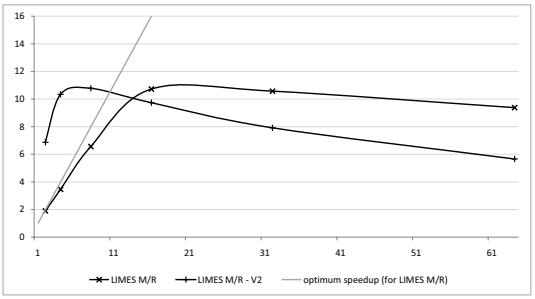


(b) Speedup graph – Organisms

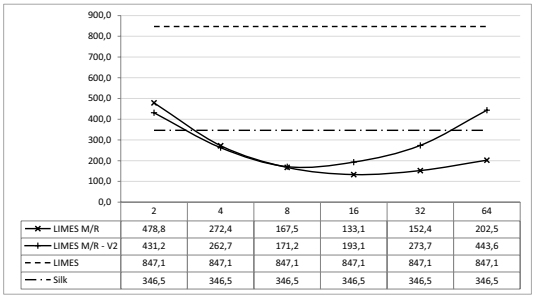


	2	4	8	16	32	64
LIMES M/R	1236,1	677,6	357,3	218,6	221,8	250,1
LIMES M/R - V2	341,2	226,6	217,5	240,7	296,1	414,3
LIMES	2344,7	2344,7	2344,7	2344,7	2344,7	2344,7
Silk	336,0	336,0	336,0	336,0	336,0	336,0

(c) Runtimes – Diseases

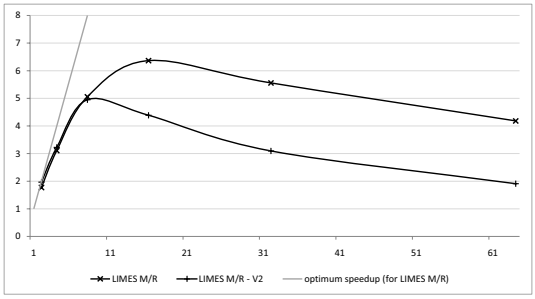


(d) Speedup graph – Diseases

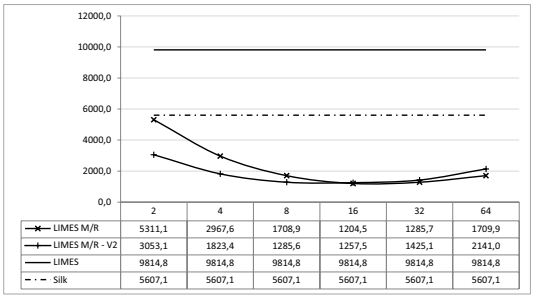


	2	4	8	16	32	64
LIMES M/R	478,8	272,4	167,5	133,1	152,4	202,5
LIMES M/R - V2	431,2	262,7	171,2	193,1	273,7	443,6
LIMES	847,1	847,1	847,1	847,1	847,1	847,1
Silk	346,5	346,5	346,5	346,5	346,5	346,5

(e) Runtimes – Similar Cities

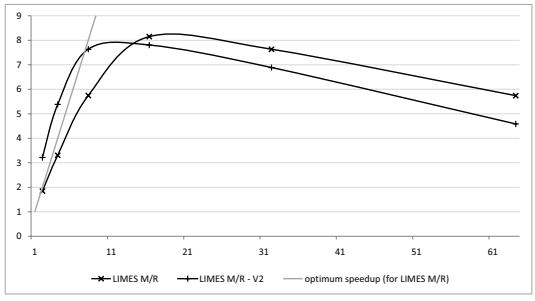


(f) Speedup graph – Similar Cities



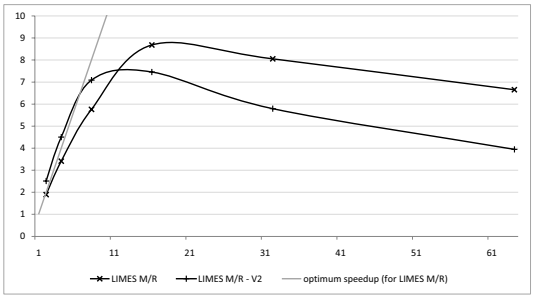
	2	4	8	16	32	64
LIMES M/R	5311,1	2967,6	1708,9	1204,5	1285,7	1709,9
LIMES M/R - V2	3053,1	1823,4	1285,6	1257,5	1425,1	2141,0
LIMES	9814,8	9814,8	9814,8	9814,8	9814,8	9814,8
Silk	5607,1	5607,1	5607,1	5607,1	5607,1	5607,1

(g) Runtimes – Similar Films

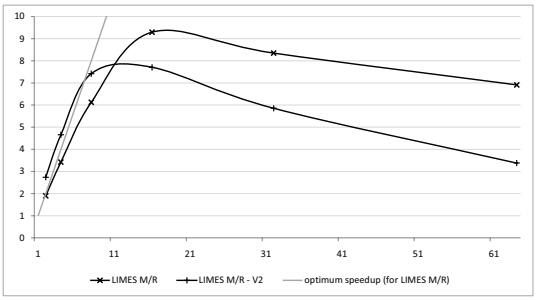


(h) Speedup graph – Similar Films

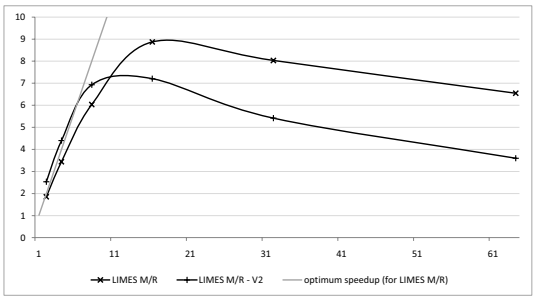
Figure 2: Experimental results for the first series of experiments. All runtimes are in seconds. The runtime of SILK is shown for 16 processors.



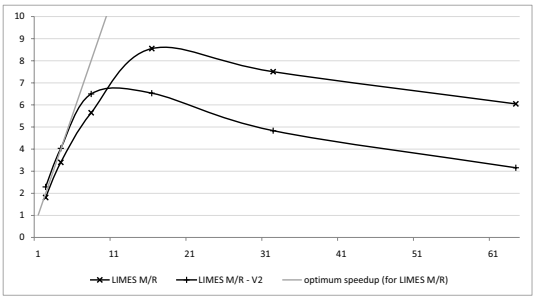
(a) Threshold = 0.5



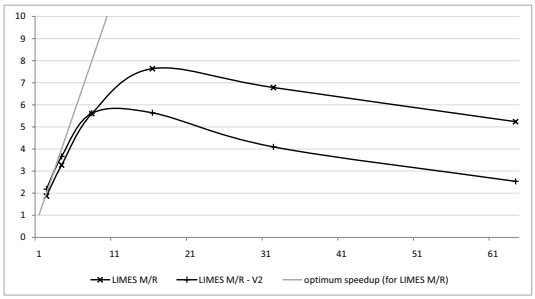
(b) Threshold = 0.75



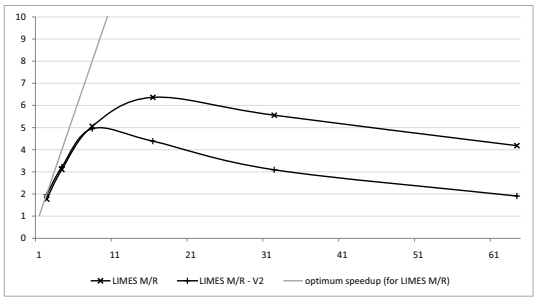
(c) Threshold = 0.8



(d) Threshold = 0.85



(e) Threshold = 0.9



(f) Threshold = 0.95

Figure 3: Experimental results for the second series of experiments