Received 20 May 2024, accepted 20 June 2024, date of publication 24 June 2024, date of current version 3 July 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3418815

RESEARCH ARTICLE

MP-HTHEDL: A Massively Parallel Hypothesis Evaluation Engine in Description Logic

EYAD ALGAHTANI
College of Applied Computer Science, Department of Information Systems, King Saud University, Riyadh 145111, Saudi Arabia
e-mail: eaalgahtani@ksu.edu.sa

ABSTRACT We present MP-HTHEDL, a massively parallel hypothesis evaluation engine for inductive learning in description logic (DL). MP-HTHEDL is an extension on our previous work HT-HEDL, which also targets improving hypothesis evaluation performance for inductive logic programming (ILP) algorithms, that uses DL as their representation language. Unlike our previous work (HT-HEDL), MP-HTHEDL is a massively parallel approach that improves hypothesis evaluation performance through horizontal scaling, by exploiting the computing capabilities of all CPUs and GPUs from networked machines in Hadoop clusters. Many modern CPUs, have extended instruction sets for accelerating specific types of computations – especially for data parallel or vector computations. For CPU-based hypothesis evaluation, MP-HTHEDL employs vectorized multiprocessing as opposed to HT-HEDL's vectorized multithreading; though, both MP-HTHEDL and HT-HEDL combine the classical scalar processing of multi-core CPUs with the extended vector instructions of each CPU core. This combination of CPUs' scalar and vector processing, resulted in more extracted performance from CPUs. According to experimental results through Apache Spark implementation, on a Hadoop cluster of 3 worker nodes that have a total of 36 CPU cores and 7 GPUs; the performance improvement achieved using the pure scalar processing power of multi-core CPUs, has yielded a speedup of up to ∼25.4 folds. When combining the scalar-processing and the extended vector instructions of those multi-core CPUs, the performance gains increased from ∼25.4 folds to ∼67 folds, on the same cluster of 3 worker nodes – these large speedups are achieved using only CPU-based processing. In terms of GPU-based evaluation, MP-HTHEDL achieved a speedup of up to ∼161 folds, using the GPUs from the same 3 worker nodes.

INDEX TERMS Machine learning, inductive logic programming, description logic, GPU, big data, MapReduce, Apache Spark.

I. INTRODUCTION

With the constant improvement in computing hardware in terms of processing power and data storage capabilities, machine learning (ML) techniques are now more capable of constructing complex ML models within reasonable time frame; from relatively large training datasets. However, the processing and storage capabilities of a single machine can only be improved or upgraded up to a certain hard limit; this limit will impose an upper boundary on the amount of data that can be processed by the machine, as well as the speed for ML algorithms. An approach for overcoming this single-machine limitation, is by using the aggregated computing resources of several networked computers towards a particular ML task. However, with the rapid generation of data by both humans and machines, which in many cases are too large to be processed by a single machine or even small cluster of machines. Therefore, approaches and algorithms were developed to address the growing need for scalable data processing and storage – for handling rapidly growing data into massive data sizes; those massive data are also known as Big Data [2]. Hadoop [7] and its data processing component (MapReduce [6]), are well known Big Data technologies and infrastructure for handling massive data storage and processing; though in recent years, many MapReduce-based data processing approaches are migrating to Spark [8]

The associate editor coordinating the review of this manuscript and approving it for publication was Mansoor Ahmed.

("Unified engine for large-scale data analytics") – due to its higher processing efficiency, with less overheads than MapReduce.

In the context of this work, we focus on using Big Data infrastructure to improve the performance and to enable Inductive Logic Programming (ILP) algorithms to learn from massive training datasets. ILP is a class of ML techniques which uses logic programming (e.g. Horn clauses) for model and knowledge representation, where the learned ML model in ILP is often described in the form of a textual logic rule or rules; those textual logic rules are also known as a hypothesis (a single rule) or a set of hypotheses (a set of rules). ILP models are capable of describing complex and multi-relational concepts using textual human-readable descriptions, and has been used in many fields [15], [16], [30]. In addition, ILP can be used with other logic formalisms for knowledge and model representation, to achieve different balances between ML model expressivity and computational complexity. Though in this work, we focus on ILPs that use Description Logic (DL) as their logic formalism. The reason for choosing DL-based ILPs, is because they have high expressive capabilities as a knowledge representation formalism, in addition to their reduced computational complexity, as compared to Horn clauses.

The aim of this work, is to address the limitations associated with using a single machine or small cluster of machines to accelerate ILP computations. In our proposed approach, we will present a scalable hypothesis evaluation algorithm that improves hypothesis evaluation performance; when evaluating very large number of input hypotheses on large ILP datasets. Our proposed approach targets improving computations for DL-based ILPs, by exploiting Big Data's existing data processing and storage infrastructure.

II. RELATED WORK

ILP is a very capable ML technique, that can describe complex and multi-relational concepts by constructing logic-based white-box human-interpretable models. A typical ILP learner formulates the learning problem, as a search problem; where an ILP learner searches through the search space of possible candidate hypotheses, i.e. logic-based descriptions; to find either a single or a set of solution hypotheses that matches the ILP training examples. Different ILP learners have different computational complexities, which are highly dependent on the used logic-based representation. To demonstrate how the logic-based representation affects the computational complexity of an ILP learner, we will use the number of supported variables in a predicate, as measurement for the complexity of a given logic formalism; there are other factors as well that effects the complexity of a given logic formalism, although, the number of variables in a single predicate, is sufficient to demonstrate the concept of computational complexity for a given logic formalism. In First Order Logic (FOL), n-ary predicates are supported such as $pred1(v1, v2, \ldots, vn)$, which also subsumes predicates like $pred60(x)$, $pred25(x, y)$, and $pred124(x, y, z)$. In FOL-based ILP learners, the hypothesis search space consists of the possible logic statements, made using the possible combinations of n-ary predicates with also the logic operations supported by FOL; although, there are FOL-based ILP learners that employ techniques that reduce the hypothesis search space, which will be discussed in this section. The hypothesis search space for FOL-based ILP learners, is relatively larger than ILP learners, that uses less expressive logic formalism. In other words, navigating the search space of possible valid FOL-based hypotheses, is computationally more expensive than less expressive logics such as DL and PL (propositional logic). Moreover, evaluating FOL-based hypotheses typically involve executing FOL reasoning algorithms, which is more computationally expensive than reasoning in DL and PL – due to the complexity and the expressive power of FOL.

On the other hand, DL-based ILP learners are limited to unary ($pred60(x)$) and binary ($pred25(x, y)$) predicates, as opposed to FOL's n-ary predicates. As a result, DL-based hypotheses are less expressive than FOL-based hypotheses. However, the hypothesis search space for DL-based hypotheses is much less than the hypothesis search space for FOL-based hypotheses; even though DL-based hypotheses are less expressive than FOL-based hypotheses, yet DL has enough expressive power to represent many real-world multi-relational concepts. In fact, DL is commonly used as the underlying logic-based representation for OWL ontologies in the OWL-DL variation. OWL (Web Ontology Language) [5] is a knowledge representation language that represents knowledge as ontologies, and the role of DL is to perform the reasoning tasks on the knowledge in OWL ontologies. Reasoning tasks on DL are less computationally expensive than FOL, which also translate into less computational effort when performing hypothesis evaluation on DL-based hypotheses.

In terms of PL, PL is limited to hypotheses using unary predicates such as $pred60(x)$, therefore they have very limited expressive power as opposed DL, and FOL. Although, PL-based ILP learning, has much smaller search space and simpler reasoning computations for performing hypothesis evaluation, as opposed to DL and FOL logics. In other words, due to the simplicity of PL, PL-related computations are completed faster than DL and FOL computations, with limitation of having very limited expressive power.

Despite the capabilities and advantages of different ILP systems, yet many ILP algorithms are inherently sequential, and difficult to parallelize. Accelerating ILP learning using parallel computing approaches, involves several design challenges and considerations. First, different logic formalisms have different reasoning algorithms; these reasoning algorithms need to be analyzed to find opportunities where parallel computing approaches can be exploited, to accelerate these reasoning algorithms either in part or in whole. Second, since different logic formalisms also have different search spaces, the parallel ILP learner must employ the appropriate parallel search strategy, that takes into account the nature of the search space; resulting from the use of a particular logic formalism. Furthermore, the data structures used for representing the logic-based knowledge, must be optimized or reengineered to allow efficient parallel memory accesses, by the parallel processors. It is worth noting that parallel computing approaches, also have their own design challenges and considerations, which must be also accommodated in addition to the design challenges and considerations related to ILPs' logic formalisms.

Even though parallelizing ILP algorithms involve several challenges and considerations, several approaches and techniques are developed to improve the performance and capabilities of ILP learners; some developed approaches use parallel computing, while others use different approaches. In terms of non-parallel approaches, there are developed approaches that focus on improving the efficiency of existing ILP algorithms. For example, some approaches aim to reduce or eliminate redundant ILP computations such as query packs [4]. While other techniques, focus on utilizing problem-specific or domain knowledge to avoid or shrink redundant and invalid areas of the hypothesis search space; such as Progol [9] for Horn clauses, and DL-Learner [3] and SPILDL [26] for DL-based ILPs.

On the other hand, other performance improvement approaches for ILP are developed, that exploit parallel computing capabilities to accelerate ILP computations; which can be categorized into shared-memory and distributed-memory approaches. In shared memory approaches, ILP computations are accelerated through multi-core CPUs [10], while others use GPUs [21], [22], [23], [24], [25], [26], [27], [28], [29]. Moreover, Other ILP performance improvement approaches are developed that combine both multi-core CPUs with their vector instructions and GPUs [1]. Furthermore, some approaches are developed to improve ILP performance through dedicated hardware accelerators, such as [12] and [31] which are dedicated FPGA-based ILP accelerators. In terms of distributed-memory approaches, ILP researchers has developed approaches that exploit Big Data infrastructure, MapReduce in particular, to improve ILP performance [11], [13], [14].

According to the literature review, ILP computations are successfully accelerated through parallel computing and other efficiency-related approaches; that lead to performance improvements with varying degrees. We can also observe that many ILP acceleration approaches, especially the distributed-memory approaches, focus on improving performance for classical ILPs that uses Horn clauses only; and no distributed-memory approaches are developed to improve computations for DL-based ILPs. Therefore, in this work, we focus on addressing the ILP research gap of no distributed-memory approaches for accelerating DL-based ILPs; by proposing a scalable and massively parallel hypothesis evaluation approach that accelerate the performance of DL-based ILPs, by exploiting the processing and storage capabilities of Big Data infrastructure. Consequently, leading to achieve the capability of handling and learning from massive amount of ILP data in a scalable manner; which also contributes to learning more accurate and more representative DL-based ILP models. In the next section, we describe the details of our proposed approach.

III. MP-HTHEDL: A MASSIVELY PARALLEL HYPOTHESIS EVALUATION ENGINE IN DESCRIPTION LOGIC

We present MP-HTHEDL, a Massively Parallel HT-HEDL, which is an extension on our previous work HT-HEDL [1] that inherit the same HT-HEDL's capabilities of aggregating the computing resources of multi-core CPUs and multi-GPUs, towards accelerating hypothesis evaluation for DL-based ILPs. Unlike our previous work (HT-HEDL), MP-HTHEDL improves hypothesis evaluation performance, by aggregating the computing power of multi-core CPUs and multi-GPUs of all networked machines; instead of only using the computing resources of a single machine. The main goal of MP-HTHEDL is to improve hypothesis evaluation performance when evaluating a very large number of hypotheses on a very large dataset. Therefore, MP-HTHEDL architecture is designed to leverage existing Big Data infrastructure such as Hadoop, towards achieving high hypothesis evaluation performance. See Fig. 1 for an overview of MP-HTHEDL architecture. In Fig. 1, we can observe that MP-HTHEDL uses HDFS (Hadoop Distributed File System) for data storage. In terms of data processing, we can use either MapReduce or Apache Spark to implement MP-HTHEDL's hypothesis evaluation master and worker algorithms.

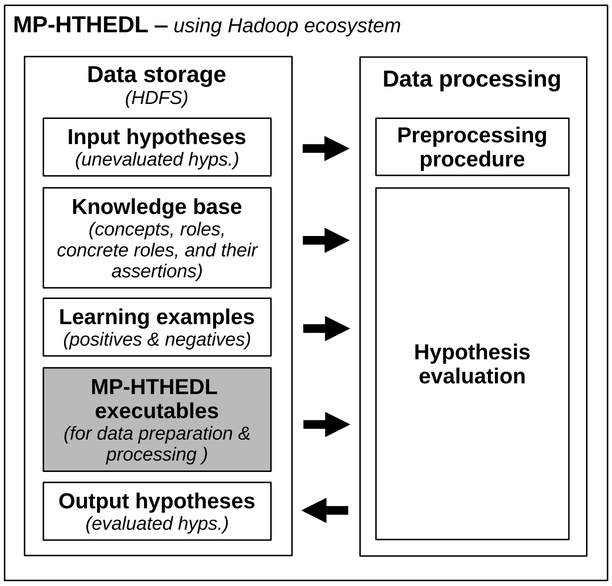
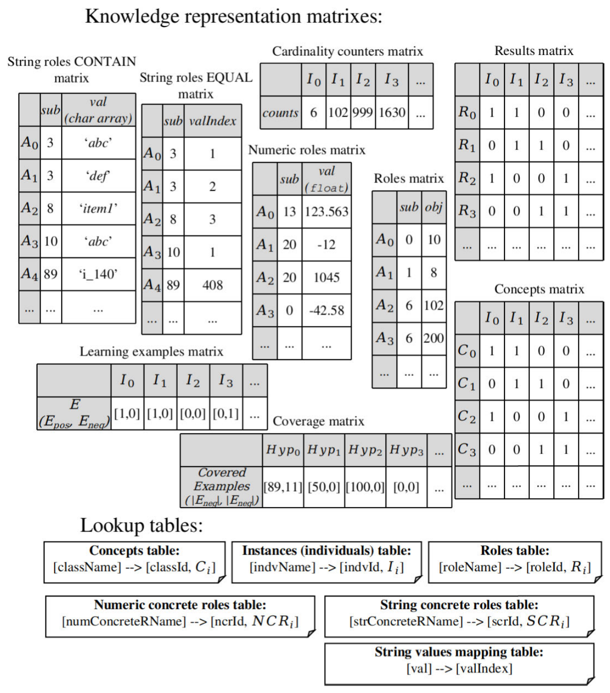

MP-HTHEDL – using Hadoop ecosystem
Data storage (HDFS)
Input hypotheses (unevaluated hyps.)
Knowledge base (concepts, roles, concrete roles, and their assertions)
Learning examples (positives & negatives)
MP-HTHEDL executables (for data preparation & processing)
Output hypotheses (evaluated hyps.)
Data processing
Preprocessing procedure
Hypothesis evaluation

FIGURE 1. An overview of MP-HTHEDL architecture.

A. DATA STORAGE

In terms of data storage, MP-HTHEDL stores in HDFS: the input hypotheses, knowledge base, learning examples, MP-HTHEDL executables, and the resulting output hypotheses. In terms of input and output hypotheses, MP-HTHEDL uses the same internal representation for DL hypotheses as HT-HEDL. In MP-HTHEDL, the input hypotheses are represented using either a single or multiple binary files in the HDFS storage; where a single binary file contain one or more serialized DL hypotheses, from their original HT-HEDL representation. In terms of output hypotheses, they use the same representation as input hypotheses; however, the *hypothesis_info* field in each output hypothesis is updated with examples coverage results – that is, the covered positives and negatives.

In terms of knowledge representation, MP-HTHEDL uses HT-HEDL's matrix-based representation to represent both the knowledge base and the learning examples. For HT-HEDL's matrix-based representation, see Fig. 2.


Knowledge representation matrixes:

String roles CONTAIN matrix

	sub	*val (char array)*
A_0	3	*'abc'*
A_1	3	*'def'*
A_2	8	*'item1'*
A_3	10	*'abc'*
A_4	89	'i_140'
...	...	...

String roles EQUAL matrix

	sub	*valIndex*
A_0	3	1
A_1	3	2
A_2	8	3
A_3	10	1
A_4	89	408
...	...	...

Cardinality counters matrix

	I_0	I_1	I_2	I_3	...
counts	6	102	999	1630	...

Numeric roles matrix

	sub	*val (float)*
A_0	13	123.563
A_1	20	-12
A_2	20	1045
A_3	0	-42.58
...	...	...

Roles matrix

	sub	*obj*
A_0	0	10
A_1	1	8
A_2	6	102
A_3	6	200
...	...	...

Results matrix

	I_0	I_1	I_2	I_3	...
R_0	1	1	0	0	...
R_1	0	1	1	0	...
R_2	1	0	0	1	...
R_3	0	0	1	1	...
...	...	...	...	...	...

Concepts matrix

	I_0	I_1	I_2	I_3	...
C_0	1	1	0	0	...
C_1	0	1	1	0	...
C_2	1	0	0	1	...
C_3	0	0	1	1	...
...	...	...	...	...	...

Learning examples matrix

	I_0	I_1	I_2	I_3	...
E (E_{pos}, E_{neg})	[1,0]	[1,0]	[0,0]	[0,1]	...

Coverage matrix

	Hyp_0	Hyp_1	Hyp_2	Hyp_3	...
Covered Examples $(\lvert E_{neg}\rvert, \lvert E_{neg}\rvert)$	[89,11]	[50,0]	[100,0]	[0,0]	...

Lookup tables:

Concepts table: [className] --> [classId, C_i]

Instances (individuals) table: [indvName] --> [indvId, I_i]

Roles table: [roleName] --> [roleId, R_i]

Numeric concrete roles table: [numConcreteRName] --> [ncrId, NCR_i]

String concrete roles table: [strConcreteRName] --> [scrId, SCR_i]

String values mapping table: [val] --> [valIndex]

FIGURE 2. **HT-HEDL matrix-based representation [1].**

In Fig. 2, HT-HEDL uses the following to represent knowledge bases in the $\mathcal{ALCQI}^{(\mathcal{D})}$ DL language:

- **Concepts matrix** for representing concepts and their assertions.
- **Roles matrix** for representing roles and their assertions.
- **Numerical roles matrix** for representing numerical concrete roles and their assertions.
- **String roles EQUAL matrix** for representing and computing EQUAL restrictions on string assertions.
- **String roles CONTAIN matrix** for representing and computing CONTAIN restrictions on string assertions.
- **Cardinality counters matrix** for computing role cardinality restrictions.
- **Results matrix** for storing intermediate results of DL operations when evaluating DL hypotheses.
- **Learning examples matrix** for representing the positive and negative learning examples.
- **Coverage matrix** for storing examples coverage results for set of evaluated input hypotheses.

In terms of HT-HEDL lookup tables, they are used to map concepts, roles, and concrete role names; to their numerical IDs. Concepts table for mapping concepts, Roles table for mapping roles, numeric concrete roles table for mapping numeric concrete roles, etc. However, the String values mapping table maps the string values of string assertions to numerical IDs, to be used in the String roles EQUAL matrix; to compute EQUAL string restrictions. After describing MP-HTHEDL's knowledge base, including learning examples, and DL hypothesis representations which are based on HT-HEDL's representations; we next describe MP-HTHEDL executables. MP-HTHEDL executables is a set of programs and scripts used by MP-HTHEDL's data processing component, to perform preprocessing and then hypothesis evaluation computations on the data stored in HDFS. MP-HTHEDL executables consists of the *PreprocessProc* script and *miniEvaluator* worker program, which are discussed next.

B. DATA PROCESSING

MP-HTHEDL's data processing component is responsible for performing hypothesis evaluation computations. To evaluate hypotheses, MP-HTHEDL performs three main steps. First, MP-HTHEDL asynchronously downloads the preprocessing procedure script (*PreprocessProc*) from HDFS, that is MP-HTHEDL executables in particular, to the local file system of each node in the cluster. Second, the downloaded preprocessing script is executed, to prepare the nodes in the Hadoop cluster for the hypothesis evaluation task. Each node in the Hadoop cluster, execute its own copy of the preprocessing script, in parallel and independently of other nodes in the cluster. See Algorithm 1 for the Preprocessing procedure's pseudocode.

In Algorithm 1, the procedure starts by downloading if not already exist from HDFS, a copy of the knowledge base and the learning examples to the node's local file system. Next, a copy of the *miniEvaluator* program is downloaded from HDFS to the node's local file system. The *miniEvaluator* program is a smaller variation of HT-HEDL that evaluate a set of input hypotheses using either a single scalar/vectorized CPU or a single GPU; on startup, the *miniEvaluator* program binds to a specific CPU core or a GPU, with affinity towards binding to GPUs rather than CPUs – to achieve better evaluation performance. For example, assuming a node with 4 GPUs and 6 CPU cores; running 6 instances of the *miniEvaluator* program, will result in 4 *miniEvaluator* instances binding to the 4 GPUs, and 2 instances binding to 2 CPU cores. MP-HTHEDL uses a set of *miniEvaluator* processes on each node in the cluster, to carry out actual hypothesis evaluation computations. See Fig. 3 for the architecture of the *miniEvaluator* worker program.

Algorithm 1 The Pseudocode for MP-HTHEDL's Preprocessing Procedure (*PreprocessProc*)

```
DownloadFromHDFSToLocalSystem(
KB_HDFSPath, KB_LocalPath);

DownloadFromHDFSToLocalSystem(
miniEvalHDFSPath, miniEvalLocalPath);

CPUCoresNum = getCPUcoresNum();
evalSlotAllocTable~=
CreateEvalSlotAllocTable(CPUCoresNum);

miniEvaluator evals[CPUCoresNum];
for (int i=0; i<CPUCoresNum; i++)
    evals[i] = startMiniEvalProcess(
    KB_LocalPath, evalSlotAllocTable[i]);

for (int i=0; i<CPUCoresNum; i++)
    evals[i].waitUntilReady()();
```

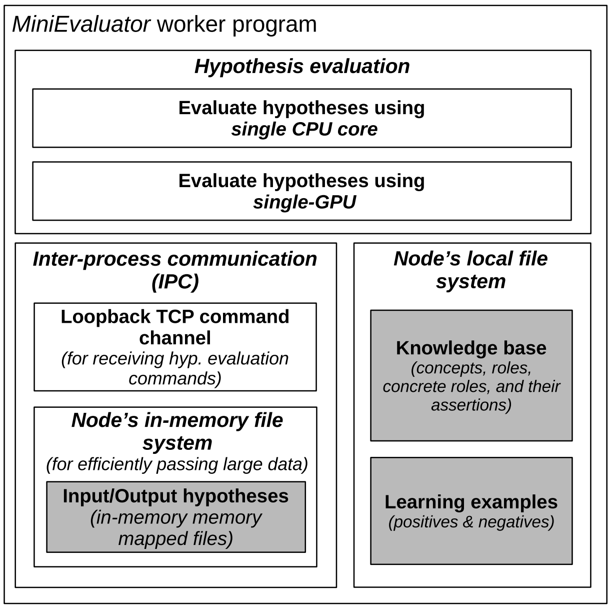

MiniEvaluator worker program
Hypothesis evaluation
Evaluate hypotheses using single CPU core
Evaluate hypotheses using single-GPU
Inter-process communication (IPC)
Loopback TCP command channel (for receiving hyp. evaluation commands)
Node's in-memory file system (for efficiently passing large data)
Input/Output hypotheses (in-memory memory mapped files)
Node's local file system
Knowledge base (concepts, roles, concrete roles, and their assertions)
Learning examples (positives & negatives)

FIGURE 3. ***miniEvaluator*** **worker program architecture.**

In Fig. 3, when a process of the *miniEvaluator* worker program is created, it will load the knowledge base and learning examples from the local system into main memory. After that, it will create a shared memory area in the in-memory file system; to provide an efficient means of receiving large number of input hypotheses, with minimal overhead. After the shared memory area is created, *miniEvaluator* will open a loopback TCP communication server as a means of IPC (inter-process communication); to receive and process hypothesis evaluation commands from MP-HTHEDL's main master hypothesis evaluation algorithm. The reason for using TCP communication for IPC instead of other methods (Message Queues, Named Pipes, Unix domain sockets, etc.), is because many of Big Data technologies are implemented and used with Java, while HT-HEDL's vectorized CPU and GPU algorithms, are implemented and used with C/C++. Therefore, using TCP communication for only receiving commands, and not for transferring hypotheses; will result in better interoperability between Java and C/C++ software, and fast-enough speed for sending/receiving commands. In other words, it is indeed true that new Java versions such as Java 16 natively support IPC mechanisms such as Unix domain sockets; in fact, we have considered using Unix domain sockets instead of TCP sockets for IPC command signaling, because Unix domain sockets are typically faster due to less overheads than TCP sockets. However, since we use Hadoop 3.3.5 in our Hadoop cluster, which is compiled with Java 8 and can support up to Java 11 runtime only according to [17]; therefore we limit our Java code to Java 8 to ensure best compatibility with Hadoop. Since Unix domain sockets are not supported in Java 8, we therefore use TCP sockets instead, for IPC command signaling. In terms of named pipes, we can use them for IPC, however, we preferred using sockets either Unix sockets or TCP sockets, because sockets are portable and compatible with many programming languages and operating systems; which is an important feature when creating interfaces between programs written in different programming languages. In addition, sockets are intuitive interfaces for command/response communications.

In terms of *miniEvaluator*, once the shared memory area and TCP communication channel are created, *miniEvaluator* is now ready for performing hypothesis evaluation computations. To evaluate hypotheses in *miniEvaluator*, input hypotheses are copied to *miniEvaluator*'s shared memory area. After that, a command is sent to *miniEvaluator*'s through its TCP communication channel, to signal that the data is ready to be processed. Next, *miniEvaluator* will evaluate the input hypotheses stored in the shared memory using either a GPU or a single core CPU, and then add examples coverage results in *hypothesis_info* for each hypothesis – in the same shared memory. Since *miniEvaluator*'s output is stored in shared memory, the output is also visible and accessible for other software in the given node – in particular, MP-HTHEDL's main hypothesis evaluation algorithm.

In terms of Algorithm 1, Once HDFS downloads of knowledge base, learning examples, and *miniEvaluator* program are completed; an *evalSlotAllocTable* is created in a shared memory area as a binary file stored in the in-memory file system. *evalSlotAllocTable* is an array of single-byte elements, with an array size equal to the number of CPU cores in the given node. For each element in *evalSlotAllocTable*, an instance or a process of the *miniEvaluator* program is created; where each created *miniEvaluator* program instance, is bound to specific index in the *evalSlotAllocTable* array. If a particular *miniEvaluator* program instance is busy in evaluating hypotheses, it will set the value '1' at its corresponding index in the *evalSlotAllocTable* array; otherwise, the value is '0' is set, which indicates that the given *miniEvaluator* instance is available, i.e. ready to perform new computations. The first indexes in *evalSlotAllocTable*, are bound to the node's GPUs first, and the remaining indexes are used for the node's CPU cores. For example, assuming a node of 10 CPU cores and 4 GPUs, *evalSlotAllocTable* will contain 10 elements where the indexes [0-3] are used for the node's 4 GPUs, and [4-9] indexes are used for the node's CPUs. Since using *evalSlotAllocTable* for allocating *miniEvaluator* workers, entails parsing the table from its [0] index for finding available CPU/GPU evaluators; as a result, GPU-based evaluators are allocated first, then CPU-based evaluators in case of all GPUs are busy.

The role of the *evalSlotAllocTable* array, is to keep track of busy and available *miniEvaluator* instances; the information in the *evalSlotAllocTable* array is used by MP-HTHEDL's main hypothesis evaluation algorithm, to facilitate efficient in-node scheduling of hypothesis evaluation workloads.

In terms of MP-HTHEDL's third step, the hypothesis evaluation starts once all instances of *miniEvaluator* worker program are ready on all nodes. The architecture for MP-HTHEDL's hypothesis evaluation is described in Fig. 4. In Fig. 4, MP-HTHEDL's hypothesis evaluation algorithm starts by receiving the input hypotheses from HDFS, which are then partitioned into multiple smaller data partitions. The data partitions are distributed to the nodes in the Hadoop cluster, where a single node may process one or multiple data partitions – in parallel with other nodes in the cluster. Once all data partitions are fully processed by MP-HTHEDL, the output is the updated list of input hypotheses, that contain examples coverage results for each input hypothesis.

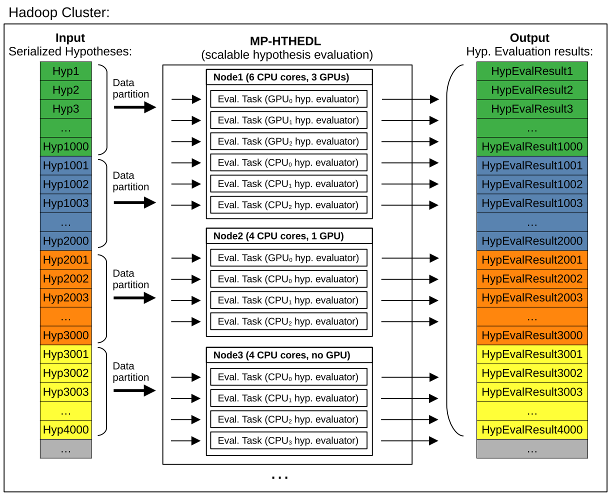


FIGURE 4. **The architecture for MP-HTHEDL's Hadoop-based hypothesis evaluation.**

MP-HTHEDL employ a pool of vectorized CPU and GPU hypothesis evaluators, in each node in the Hadoop cluster; this pool of evaluators is used by in-node Eval. tasks, for performing hypothesis evaluation computations. See Fig. 5 for processing parallel in-node Eval. tasks.

In Fig. 5, each parallel Eval. task receives its own set of input hypotheses. Next, each Eval. task allocate

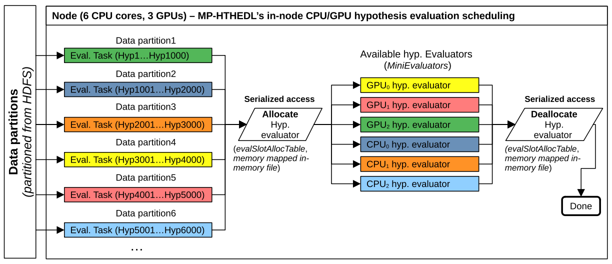

Node (6 CPU cores, 3 GPUs) – MP-HTHEDL's in-node CPU/GPU hypothesis evaluation scheduling
Data partitions
(partitioned from HDFS)
Data partition1
Eval. Task (Hyp1...Hyp1000)
Data partition2
Eval. Task (Hyp1001...Hyp2000)
Data partition3
Eval. Task (Hyp2001...Hyp3000)
Data partition4
Eval. Task (Hyp3001...Hyp4000)
Data partition5
Eval. Task (Hyp4001...Hyp5000)
Data partition6
Eval. Task (Hyp5001...Hyp6000)
Serialized access
Allocate Hyp. evaluator
(evalSlotAllocTable, memory mapped in-memory file)
Available hyp. Evaluators
(MiniEvaluators)
GPU₀ hyp. evaluator
GPU₁ hyp. evaluator
GPU₂ hyp. evaluator
CPU₀ hyp. evaluator
CPU₁ hyp. evaluator
CPU₂ hyp. evaluator
Serialized access
Deallocate Hyp. evaluator
(evalSlotAllocTable, memory mapped in-memory file)
Done

FIGURE 5. **MP-HTHEDL's in-node Eval. tasks.**

a CPU/GPU hypothesis evaluator from evaluators pool *evalSlotAllocTable*, with affinity towards allocating available GPU-based evaluators, rather than CPU-based evaluators; the process for allocating/deallocating evaluators is serialized through locks, to prevent race condition between parallel allocation requests issued by in-node Eval. tasks. Once an evaluator is allocated for a given Eval. task, the Eval. task will then evaluate the hypotheses in its data partition using the assigned evaluator. After that, hypotheses evaluation results from the Eval. task's assigned evaluator, are written to the Eval. task's output. Once a Eval. task completes writing its output, it will then release or deallocate the used evaluator back to the given node's evaluators pool. All Eval. tasks are performing their tasks in parallel with one another, except any operation related to allocating/deallocating evaluators, which is serialized.

In terms of processing a single data partition in a Eval. task, the procedure is described in Algorithm 2.

Algorithm 2 A C-Like Pseudocode for MP-HTHEDL's Eval. Task

```
// INPUT: a single data partition (inHyps),
// a set of input hypotheses

// OUTPUT: out, a~set of evaluated
// hypotheses

//get an available evaluator
lock(evalSlotAllocTable);
evalID=getAvailableEvaluator();
evalSlotAllocTable[evalID]=1; //busy
unlock(evalSlotAllocTable);


//assign work to evaluator
connectToEvaluator(evalID);
SerializeAndWriteHypsIntoBuf(inHyps,
inmemMemMappedBuf[evalID]);

startAndWaitForHypEvaluation(evalID);

//parse evaluator output
for(hypthesis h:inmemMemMappedBuf[evalID])
    out.write(h);

//deallocate evaluator
lock(evalSlotAllocTable);
evalSlotAllocTable[evalID]=0; //available
unlock(evalSlotAllocTable);
```

In Algorithm 2, MP-HTHEDL's Eval. task starts by first calling the function *getAvailableEvaluator*(), that checks the *evalSlotAllocTable* table and then returns the index of an available evaluator; that is, the index of a first encountered element with '0' (available) value. Since several parallel instances of the Eval. task may execute on a single node, we use locks to coordinate the access and to prevent race conditions when accessing the *evalSlotAllocTable* table. Once the index or the ID of an available evaluator is acquired in *evalID* for the given Eval. task, the Eval. task connects to the given evaluator's TCP loopback server. After the TCP connection is established, Eval. task writes through *SerializeAndWriteHypsIntoBuf*(...) the data partition's binary data which is the set of hypotheses, to the evaluator's *inmemMemMappedBuf*[*evalID*]; which is the evaluator's memory mapped input hypotheses file, stored in the in-memory file system (''/dev/shm''). After input hypotheses in the data partition are written to *inmemMemMappedBuf*[*evalID*], the Eval. task calls the function *startAndWaitForHypEvaluation*(...), that send a TCP packet to the evaluator to start evaluate the assigned hypotheses; the *startAndWaitForHypEvaluation*(...) function blocks until the evaluator completes its assigned computations. Once the computations are completed, *inmemMemMappedBuf*[*evalID*] now contains the same set of input hypotheses, though updated with their evaluation results – that is, the covered positives and negatives for each input hypothesis. The Eval. task then directs the evaluator's output, to the Eval. task's output. Once the Eval. task's output is completely written, the used evaluator is then released through *evalSlotAllocTable*[*evalID*] $= 0$;, so that it can later be used by other Eval. tasks.

Next, we describe implementation and evaluation details for MP-HTHEDL's hypothesis evaluation algorithm.

IV. IMPLEMENTATION AND EVALUATION

For implementing MP-HTHEDL's algorithms, we use Apache Spark instead of MapReduce for many reasons. First, Spark processes and caches data in memory, while MapReduce uses disks for loading/storing intermediate data; this difference, make Spark generally much faster than MapReduce. Second, Spark supports iterative data processing, which is an important capability needed for ILPs inherent iterative learning approaches; unlike MapReduce which is much more suitable for batch processing tasks. We implement MP-HTHEDL using Spark with Hadoop's YARN (Yet Another Resource Negotiator), where Spark uses Hadoop's infrastructure to process large amount of hypotheses data. MP-HTHEDL's Spark application is written using the JAVA language, and the *miniEvaluator* worker program is written in C/C++. For implementing HT-HEDL's hypothesis evaluation algorithms, which also includes HT-HEDL's algorithms in the *miniEvaluator* worker program; we use Nvidia CUDA API [19] for implementing the GPU-based evaluation algorithms. For the CPU-based evaluation algorithms, we use OpenMP [20] for CPU multithreading, and Intel Intrinsics [18] for utilizing the CPUs' SSE (Streaming SIMD Extensions) vector instructions. In terms of data storage, MP-HTHEDL's Spark application uses HDFS for data storage.

To evaluate MP-HTHEDL, we use a Hadoop cluster of 4 nodes, that have 1 master node and 3 worker nodes. The details for the used Hadoop cluster are described in Table 1.

TABLE 1. The used 4-nodes cluster for MP-HTHEDL experiments.

Node name	CPU	RAM	GPU(s)
Master node	AMD Ryzen 7 3750H (4 cores, 8 threads)	16 GB	Integrated graphics
Worker node 1 (M_1)	AMD Ryzen 9 5950X (16 cores, 32 threads)	32 GB	–Nvidia RTX 3090 (GPU_1) –Nvidia RTX 3070 (GPU_2) –Nvidia GTX 1070 (GPU_3) –Nvidia GTX 970 (GPU_4)
Worker node 2 (M_2)	AMD Ryzen 9 5800X (8 cores, 16 threads)	32 GB	–Nvidia RTX 3080 Ti (GPU_1) –Nvidia RTX 3070 (GPU_2)
Worker node 3 (M_3)	AMD Ryzen 9 5900X (12 cores, 24 threads)	32 GB	Nvidia RTX 3080 Ti (GPU_1)

We evaluate MP-HTHEDL using 3 types of experiments. In the first experiment (see Table 2), we study how the number of hypotheses affect performance. In the second experiment (see Table 3), we study how the dataset size in terms of the number of individuals, affect evaluation performance. In terms of the third experiment (see Table 4), we study the performance impact of adding more nodes to the cluster. In the experiments, we use replicated copies of the same hypothesis, where the replicated hypothesis is a conjunction of 5 concepts – to reflect an average size hypothesis. All measured execution times for evaluating hypotheses in Table 2-4, are reported in milliseconds; the execution times cover hypothesis evaluation computations only, and doesn't include MP-HTHEDL's initialization steps – because MP-HTHEDL is initialized once and reused multiple times, by DL-based large-scale ILP learners for their hypothesis evaluation. Since MP-HTHEDL inherits the hypothesis

TABLE 2. Experiment 1: study performance impact of evaluating #*hyp* hypotheses on 10 million individuals dataset.

#hyp	Baseline	MP-HTHEDL on 4 nodes Hadoop cluster			
		Scalar	Vector	GPUs only	Vector + GPUs
1	32	1,668	1,617	1,677	1,664
10	307	2,018	1,647	1,666	1,660
100	3,056	4,702	1,923	1,716	1,727
1,000	29,898	10,239	3,157	2,368	2,431
10,000	308,156	27,136	8,919	5,559	8,315
100,000	3,060,591	120,750	45,797	19,055	24,506
1M	-*	1,221,557	440,553	157,984	184,571

* The experiment was aborted after few hours of running, since it is anticipated to take $\geq$ 8 hours to complete.

TABLE 3. Experiment 2: study performance impact of evaluating 100,000 hypotheses on varying dataset of #*ivsNum* individuals.

#ivsNum	Baseline	MP-HTHEDL on 4 nodes Hadoop cluster			
		Scalar	Vector	GPUs only	Vector + GPUs
10	9	12,406	12,162	11,142	9,676
100	37	12,223	12,251	11,703	9,946
1,000	305	12,501	12,249	10,807	9,577
10,000	2,992	12,789	12,132	10,918	10,001
100,000	29,953	13,788	13,594	11,432	9,858
1M	302,976	22,826	12,006	11,248	10,067
10M	2,994,485	121,777	45,537	19,419	24,687
100M	-*	1,158,008	494,565	115,731	181,296

* Similar to Table 2, the experiment took too long to complete – anticipated to take $\geq$ 8 hours to complete.

TABLE 4. Experiment 3: study performance impact of evaluating 100,000 hypotheses on 10 million individuals dataset, using a cluster of #*WNode* worker nodes.

#WNode	Baseline	MP-HTHEDL on 4 nodes Hadoop cluster			
		Scalar	Vector	GPUs only	Vector + GPUs
HT-HEDL (M_1)	3,060,591	281,934	74,753	23,141	31,219
M_1		286,373	96,032	25,903	29,697
M_1, M_2		183,523	72,320	21,880	26,185
M_1, M_2, M_3		120,750	45,797	19,055	24,506

evaluation capabilities of HT-HEDL in terms of using CPUs and GPUs; therefore, we also report CPU-based and GPU-based evaluation performance for each experiment type. We use computer-generated synthetic data for all experiments. In terms of experiments baseline, we use the sequential scalar CPU performance of M_1 machine, which is reported in the "Baseline" column on all experiment tables (Table 2-4). The reason for using scalar single CPU performance as the baseline, is because it is the evaluation method used by traditional ILP learners and also used by DL-based ILPs like the DL-learner; which is the state of the art in DL-based ILP learning. According to the literature review, parallel ILP approaches focused on either using: GPUs, or the scalar performance of multi-core CPUs, through multithreading for example. In other words, vectorized CPU-based hypothesis evaluation is not used in the ILP literature yet. In fact, one of the key contributions of our previous work HT-HEDL, is the combination of multithreading with vector instructions of modern multi-core CPUs to improve hypothesis evaluation. Therefore, we consider vectorized CPU-based evaluation to be an accelerated hypothesis evaluation method, and not a traditional or a baseline method; especially since vectorized CPU-based evaluation require explicit use of CPUs extended vector instructions set. Moreover, the vector instructions of CPUs are typically not as sophisticated as the vector instructions of GPUs, because the vector instructions of CPUs have more data alignment requirements; such as the data must be loaded and stored in continuous and adjacent memory addresses. The enabler for MP-HTHEDL's vectorized CPU-based hypothesis evaluation, is the matrix-based representation of HT-HEDL and its data processing algorithms, which are optimized for both CPU-based vector instructions and also GPU-based processing.

In all experiment tables, the "Scalar" column refers to the use of scalar CPU cores without vector instructions for computing hypotheses evaluation results. The "Vector" column refers to the use of CPU cores in addition to each core's vector instructions for performing hypothesis evaluation computations. The "GPUs only" column refers to using only GPUs and ignoring CPUs in each worker node for performing computations. In "Vector + GPUs", the performance of all GPUs and all vectorized CPUs in each worker node, are used for performing computations.

V. DISCUSSION

The experimental results provide a clear evidence that MP-HTHEDL indeed introduces major speedups. In experiment 1 (visualized in Fig. 6), we can observe MP-HTHEDL starts to introduce performance improvements, when the number of input hypotheses is $\geq$ 100 hypotheses; since on and above 100 input hypotheses, Hadoop and Spark related overheads, are masked or justified by the introduced performance speedups. The speedups grow larger as the number of input hypotheses increases. For example, at 100 input hypotheses, a speedup of 1.78 folds is achieved using GPUs only; whereas at 100,000 input hypotheses, a speedup of ~161 folds is achieved using only GPUs.

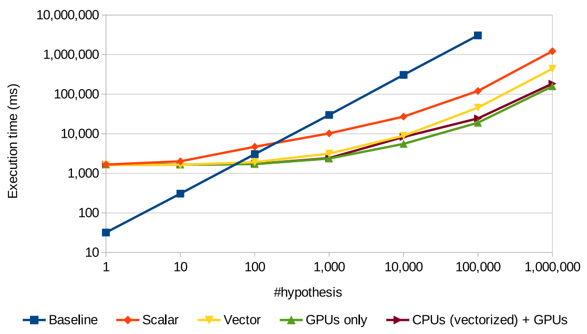


FIGURE 6. Visualizing experiment 1 results using log-log plot: the performance impact of evaluating #*hyp* hypotheses on 10 million individuals.

In terms of experiment 2 (visualized in Fig. 7), performance speedups start to appear at 100,000 individuals; because the time spent in hypothesis evaluation computations is large enough, to justify the overheads associated with: Hadoop, Spark, and MP-HTHEDL's in-node scheduler. The speedups grow larger with the increase in the number of individuals in the dataset. For example, at 100,000 individuals, a speedup of 3 folds is achieved by combining CPUs and GPUs; on the other hand, at 10 million individuals, a speed up of 154 folds is achieved using GPUs only.

We can deduce from experiment 1 and experiment 2, that in order to achieve speedups and especially large speedups, the number of input hypotheses and also the number of individuals in the knowledge base, has to be large

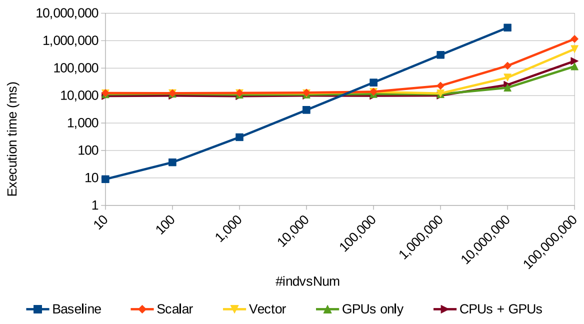


FIGURE 7. Visualizing experiment 2 results using log-log plot: the performance impact of evaluating 100,000 hypotheses on ***#ivsNum*** individuals.

enough – to justify the overheads related to Hadoop, Spark, and MP-HTHEDL's scheduler.

For experiment 3 (visualized in Fig. 8), we can observe in the "HT-HEDL (M_1)" experiments, that running a single instance of HT-HEDL on M_1, will introduce similar performance gains as running MP-HTHEDL on a single node cluster that only has the M_1 node; which clearly suggest that the performance of MP-HTHEDL on a single node, is comparable with the performance of running a single HT-HEDL instance on that same node. Although, there are some performance differences between HT-HEDL and MP-HTHEDL. First, we can observe that among the different evaluation methods, HT-HEDL's "vector" evaluation is superior to MP-HTHEDL's "vector" evaluation. Second, we can observe that the performance of HT-HEDL on the "vector" and "GPUs only" experiments, is roughly in the middle point between the performance of MP-HTHEDL's "M_1" and "M_1+M_2" experiments. HT-HEDL is able to extract more performance from the M_1 machine, because of much less scheduling and communication overheads as opposed to MP-HTHEDL's overheads. Also, the employed scheduling strategy has a key role in affecting the performance for both HT-HEDL and MP-HTHEDL.

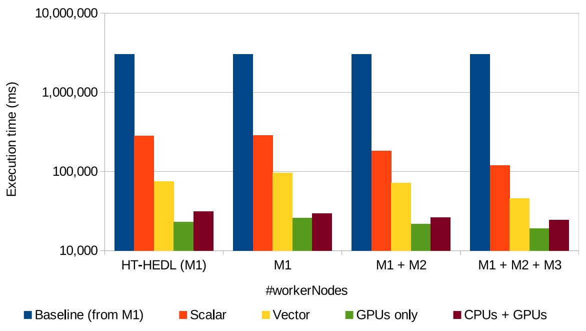


FIGURE 8. Visualizing experiment 3 results using log scale: the performance impact of evaluating 100,000 hypotheses on 10 million individuals dataset, using ***#WNode*** cluster.

Despite MP-HTHEDL's overheads, MP-HTHEDL introduced performance gains higher than HT-HEDL, especially when more worker nodes are added into the cluster; for example, in the "M_1+M_2" experiments, MP-HTHEDL provided faster performance than HT-HEDL. In fact, the more worker nodes are added, the larger the performance gap between HT-HEDL and MP-HTHEDL – in favor of MP-HTHEDL. At 1 worker node (M_1), performance increase of 118 folds is achieved using GPUs only; by adding M_2 node to the cluster, the performance speedup is increased to ∼140 folds over baseline, using only GPUs. When using all three $M_1 - 3$ nodes, the performance speedup is further increased to ∼161 folds using GPUs only over baseline. It is worth noting that the speedups introduced by adding a given node to the cluster, is highly dependent on that node's computing capabilities; which also include the node's added contribution to the clusters' overall computing capabilities – such as the number and performance of each added CPU core, and the number and performance of each added GPU. In other words, adding a node with very weak computing capabilities, in comparison to other existing nodes in the cluster; will hinder the cluster overall performance, due to the introduced performance bottleneck from the weaker node.

In terms of the "GPUs only" experiments, we can also observe that adding more GPUs results in more performance being achieved by the cluster. However, adding more GPUs through different machines, result in achieving smaller increases in performance as opposed to the "Scalar" and "Vector" experiments. The reason for such smaller performance gains for the "GPUs only" experiments, is because MP-HTHEDL's in-node scheduler assigns a fixed number of hypotheses to each available CPU or GPU evaluator in a given node, which doesn't take into account the individual computing capabilities of each GPU and CPU. Depending on the GPU's computing performance, powerful GPUs completes their assigned hypotheses sooner, and will remain idle until the next set of input hypotheses is assigned; since the powerful GPU is idle more often than the less powerful GPU, therefore it will reduce the utilization efficiency of the powerful GPU. On the other hand, less powerful GPUs will take longer to process their assigned hypotheses. By the time the less powerful GPU completes its assigned computations, the next set of input hypotheses are already added in the GPU's processing queue; since the processing queue of the less powerful GPU is often saturated with computation tasks, this will result in efficient use of the GPU's computing resources.

In the "Vector + GPUs" experiments, the introduced performance gains are higher than "Vector" experiments and lower than "GPUs only" experiments. The reason for why the performance gains of "Vector + GPUs" experiments are less than the "GPUs only" experiments, is because the GPUs are much more powerful than the CPUs; by which, the GPUs complete their assigned computations sooner than the CPU cores, whereas the CPU cores are still processing their assigned computations. In other words, the CPU cores become the performance bottleneck because the CPUs cannot complete the same workload as fast as the GPU can. Moreover, we can observe that "Vector + GPUs" performance in HT-HEDL is slightly slower than MP-HTHEDL (on $\#WNode = M_1$); the reason for this performance, is because HT-HEDL's scheduling algorithm, sends GPU-based hypothesis evaluation workloads to all GPUs first, and then proceed with executing the CPU's assigned hypothesis evaluation workloads. By the time these GPU-related workload assignments are sent to all GPUs, some GPUs - especially the powerful ones - may have already finished some or all of their assigned computations – while the CPU is still busy sending GPU-related workload assignments to other GPUs. Since the CPU starts executing its assigned hypothesis evaluation workloads later, after some or all GPUs may have already finished their assigned computations, therefore the CPU negatively affects or bottlenecks the CPU-GPU hypothesis evaluation performance.

On the other hand, in MP-HTHEDL's "Vector + GPUs" performance, CPU-based and GPU-based hypothesis evaluation workloads are assigned and started at the same time; unlike HT-HEDL, where GPU-based hypothesis evaluation workloads are started on all GPUs first, and then CPU-based hypothesis evaluation workloads are started. As a result of this workload scheduling difference between HT-HEDL and MP-HTHEDL, we can observe that MP-HTHEDL extracted more performance from the same hardware, as opposed by HT-HEDL.

The solution to improve the utilization efficiency for both powerful and less powerful GPUs in the cluster, is to employ a GPU scheduling strategy that assigns the appropriate workloads based on the computing capabilities of each GPU – to maximize the cluster's GPU-based evaluation performance. In HT-HEDL, we have already developed a capability-aware scheduler that uses information about the performance of each CPU and GPU; to schedule the appropriate workloads to each CPU and GPU based on their individual computing capabilities in a single machine environment. Although, employing HT-HEDL's capability-aware scheduler as MP-HTHEDL's in-node scheduler will remain as a potential future work.

Based on all experimental results, we can deduce the following. In the first deduction, the input data in terms of hypotheses number and knowledge base size, has to be large enough to hide the aforementioned overheads; MP-HTHEDL provides large speedups on large input data, whereas on small input data, MP-HTHEDL is unable to compete with baseline performance. In the second deduction, the largest speedups by MP-HTHEDL, are achieved by using GPUs only, whereas the least speedups are from scalar CPU performance. Moreover, GPUs only evaluation achieved more performance than the combined vectorized CPU + GPU evaluation; the reason is because GPUs have much more powerful vector processing capabilities than multi-core CPUs with vector instructions. Therefore, a GPU will finish its assigned vector-based workload much faster than a vectorized multi-core CPU, which consequently leading the CPU into becoming the performance bottleneck. In certain scenarios, a combination of multi-core CPUs and GPUs will introduce more performance, such as when combining a multi-core CPU with older generation GPUs; which resulted in more performance speedups, as demonstrated from experimental results in our previous work, HT-HEDL [1]. Combining CPUs and GPUs to achieve more performance, is highly affected by the computing capabilities, and the number of used CPUs and GPUs; in addition, the workload distribution and scheduling policy also affects the CPU-GPU performance.

In the third deduction, we can observe that MP-HTHEDL's in-node scheduler perform well with CPU-based evaluators as opposed to GPU-based evaluators, because the strategy of scheduling fixed workloads to multi-core CPUs that have symmetric processing cores, will result in significant CPU-based performance gains. In the fourth deduction, the speedups achieved by MP-HTHEDL through different hypothesis evaluation types using all three worker nodes ($M_1 - 3$); can be ranked from smallest to largest speedups based on the experimental results: Scalar only ($\sim$25.4 folds)$\rightarrow$Vector only ($\sim$67 folds)$\rightarrow$ combined: CPUs+GPUs ($\sim$125 folds)$\rightarrow$GPUs only ($\sim$161 folds)

VI. CONCLUSION

The aim of this work, is to propose and evaluate an approach that improves hypothesis evaluation performance (a key task for ILP learners), which also improves hypothesis evaluation throughput; by aggregating the computing resources of networked machines. The proposed approach targets DL-based ILPs and designed as a massively parallel algorithm that exploits computing resources of existing Big Data infrastructure, namely Hadoop. In terms of computing performance, the proposed approach not only uses CPU cores in their classical scalar processing fashion, but rather combines CPU's scalar processing capabilities with each CPU core's vector processing instructions – typically available on modern multi-core CPUs. The combination of CPU's scalar processing with its vector-processing instructions, has consequently lead into improving the performance of CPU-based hypothesis evaluation even more, according to experimental results.

Based on the experimental results, even on a cluster with only a single worker node, the proposed approach has yielded very large speedups up to 118 folds. When the number of worker nodes is increased to 2, the performance speedups has increased from 118 folds to $\sim$140 folds; the performance was increased further when 3 nodes were used, which resulted in the speedups to increase from $\sim$140 folds using 2 nodes, to $\sim$161 folds using 3 nodes. However, due to the overheads related to Hadoop, Spark, and MP-HTHEDL's in-node scheduler; the input data has to be large enough to cancel out or justify the those overheads. Since the proposed approach is based on the Big Data infrastructure paradigm, this makes it inherit Hadoop's and Spark's large scale processing capabilities; that can utilizes large clusters of 100+ or 1,000+ nodes – to accelerate computations. Therefore, the proposed approach in this work, will act as a scalable hypothesis evaluation tool or a scalable component, for handling very heavy hypothesis evaluation tasks against large knowledge bases for DL-based ILPs.

In terms of future work, other performance accelerators can also be integrated with MP-HTHEDL to further improve performance, such as hardware based accelerators like FPGAs for example. Moreover, MP-HTHEDL's scheduling algorithm can be improved to increase the utilization efficiency of GPUs in the cluster. Furthermore, MP-HTHEDL's logic-based representation can be extended to support other expressive logic formalisms such as Horn clauses.

REFERENCES

[1] E. Algahtani, "HT-HEDL: High-throughput hypothesis evaluation in description logic," *ACM Trans. Knowl. Discovery Data*, 2024.

[2] N. Elgendy and A. Elragal, "Big data analytics: A literature review paper," in *Proc. Ind. Conf. Data Mining*, 2014, pp. 214–227.

[3] J. Lehmann, "Learning OWL class expressions," Ph.D. thesis, Fac. Math. Comput. Sci., Univ. Leipzig, Leipzig, Germany, 2010.

[4] H. Blockeel, L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, and H. Vandecasteele, "Executing query packs in ILP," in *Proc. 10th Int. Conf. Inductive Logic Programming*, London, U.K., Jul. 2000, pp. 60–77.

[5] OWL. *OWL Web Ontology Language*. Accessed: May 18, 2024. [Online]. Available: https://www.w3.org/TR/2004/REC-owl-features-20040210

[6] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[7] (2010). *Apache Software Foundation*. Accessed: May 19, 2024. [Online]. Available: https://hadoop.apache.org

[8] (2023). *Apache Spark: Unified Engine for Large-scale Data Analytics*. Accessed: May 19, 2024. [Online]. Available: https://spark.apache.org/

[9] S. Muggleton, "Inverse entailment and progol," *New Gener. Comput.*, vol. 13, nos. 3–4, pp. 245–286, Dec. 1995.

[10] A. Meissner, "A simple parallel reasoning system for the ALC description logic," in *Proc. 1st Int. Conf. Comput. Collective Intell.*, 2009, pp. 413–424.

[11] A. Srinivasan, T. Faruquie, and S. Joshi, "Exact data parallel computation for very large ILP datasets," in *Proc. 20th Int. Conf. Inductive Logic Program.*, Florence, Italy, 2010, pp. 1–8.

[12] E. Algahtani, "A hardware approach for accelerating inductive learning in description logic," *ACM Trans. Embedded Comput. Syst.*, May 2024. https://dl.acm.org/doi/10.1145/3665277.

[13] A. Srinivasan, T. A. Faruquie, and S. Joshi, "Data and task parallelism in ILP using MapReduce," *Mach. Learn.*, vol. 86, no. 1, pp. 141–168, Jan. 2012.

[14] J. Corte-Real, I. Dutra, and R. Rocha, "Prolog programming with a map-reduce parallel construct," in *Proc. 15th Symp. Princ. Pract. Declarative Program.*, Madrid, Spain, Sep. 2013, pp. 285–296.

[15] A. K. Debnath, R. L. Lopez de Compadre, G. Debnath, A. J. Shusterman, and C. Hansch, "Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. Correlation with molecular orbital energies and hydrophobicity," *J. Medicinal Chem.*, vol. 34, no. 2, pp. 786–797, Feb. 1991.

[16] C. Feng, "Inducing temporal fault diagnostic rules from a qualitative model," in *Proc. 8th Int. Conf. Mach. Learn.*, 1991, pp. 403–406.

[17] (2020). *Hadoop Java Versions*. Accessed: May 11, 2024. [Online]. Available: https://cwiki.apache.org/confluence/display/HADOOP/Hadoop+Java+Versions

[18] (2024). *Intel Intrinsics Guide*. Accessed: May 11, 2024. [Online]. Available: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#techs=SSE_ALL

[19] *Nvidia CUDA Toolkit*. Accessed: May 11, 2024. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[20] *OpenMP*. Accessed: May 11, 2024. [Online]. Available: https://www.openmp.org/

[21] E. Algahtani and D. Kazakov, "GPU-accelerated hypothesis cover set testing for learning in logic," in *Proc. 28th Int. Conf. Inductive Logic Program.*, Ferrara, Italy, 2018, pp. 6–20.

[22] E. Algahtani and D. Kazakov, "CONNER: A concurrent ILP learner in description logic," in *Proc. 29th Int. Conf. Inductive Logic Program.*, Plovdiv, Bulgaria, 2019, pp. 1–15.

[23] C. A. Martínez-Angeles, H. Wu, I. Dutra, V. S. Costa, and J. Buenabad-Chávez, "Relational learning with GPUs: Accelerating rule coverage," *Int. J. Parallel Program.*, vol. 44, no. 3, pp. 663–685, Jun. 2016.

[24] C. Chantrapornchai and C. Choksuchat, "TripleID-Q: RDF query processing framework using GPU," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 9, pp. 2121–2135, Sep. 2018.

[25] P. Makpisit and C. Chantrapornchai, "SPARQL query optimizations for GPU RDF stores," *ECTI-CIT Trans.*, vol. 17, no. 2, pp. 235–244, Jun. 2023.

[26] E. Algahtani, "A scalable and parallel inductive learner in description logic," Ph.D. thesis, Comput. Sci., Univ. York, York, U.K., 2020.

[27] M. Kharma, "GRUEL: An EL reasoner using general purpose computing on a graphical processing unit," M.S. thesis, Comput. Softw. Eng., Concordia Univ., Montreal, QC, Canada, 2017.

[28] P. Makpaisit and C. Chantrapornchai, "VEDAS: An efficient GPU alternative for store and query of large RDF data sets," *J. Big Data*, vol. 8, no. 1, p. 125, Sep. 2021.

[29] J. Pang, "TriAG: Answering SPARQL queries accelerated by GPU," in *Proc. 10th Int. Workshop Comput. Sci. Eng.*, Shanghai, China, Jun. 2020, pp. 300–306.

[30] I. Bratko and S. Muggleton, "Applications of inductive logic programming," *Commun. ACM*, vol. 38, no. 11, pp. 65–70, 1995.

[31] A. Fidjeland, W. Luk, and S. Muggleton, "Scalable acceleration of inductive logic programs," in *Proc. IEEE Int. Conf. Field-Programmable Technol.*, Hong Kong, Dec. 2002, pp. 252–259.

EYAD ALGAHTANI received the Ph.D. degree in computer science with a focus on scalable machine learning from the University of York, U.K. In his studies, he is working on developing scalable inductive logic programming (ILP) algorithms, capable of constructing human-interpretable machine learning (ML) models, from large amount of real-world data; through both high-performance computing (HPC) and non-HPC approaches. He is currently an Assistant Professor with King Saud University, Saudi Arabia. His main research interests include scalable and high-performance human-interpretable ML.