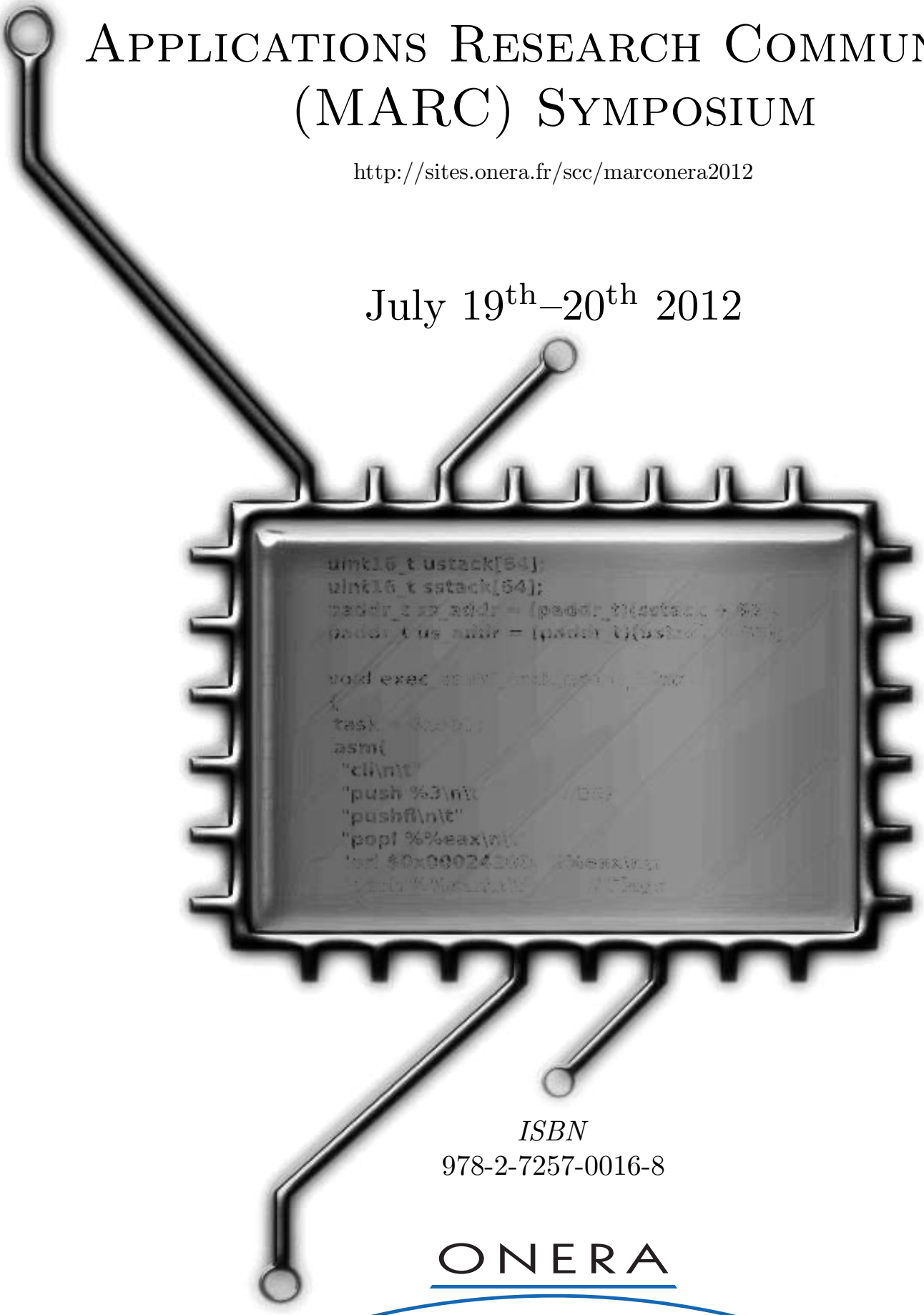
Performance of RDF Query Processing on the Intel SCC

Vasil Slavov, Praveen Rao, Dinesh Barenkala, Srivenu Paturi

Proceedings of the 6th Many-core Applications Research Community (MARC) Symposium

http://sites.onera.fr/scc/marconera2012

July 19th–20th 2012



ISBN
978-2-7257-0016-8

ONERA
THE FRENCH AEROSPACE LAB

 ISBN: 978-2-7257-0016-8

Performance of RDF Query Processing on the Intel SCC

Vasil Slavov, Praveen Rao, Dinesh Barenkala, and Srivenu Paturi

***Abstract*—Chip makers are envisioning hundreds of cores in future processors for throughput oriented computing. These processors, called manycore processors, require new architectural innovations for scaling to a large number of cores as compared with today's multicore processors. We report an early study on the performance of RDF query processing on a manycore processor. In our study, we use the Intel SCC, an experimental manycore processor from Intel Labs. This processor has new architectural features, namely, 48 Pentium cores, a high speed, on-chip mesh network to communicate between cores and access memory controllers, on-chip message passing buffers for high speed message passing, and software controlled fine-grained power management. We classify queries based on their I/O footprint and study the impact of two standard models, namely, task and data parallel programming models. Based on our experiments with synthetic and real RDF datasets on the SCC, we conclude that the task parallelism model provides an immediate way to boost the performance of RDF query processing.**

I. INTRODUCTION

Chip makers are envisioning hundreds of cores in future processors for throughput oriented computing. In throughput oriented computing, we expect abundant parallelism opportunities in the workload, and aim to achieve high throughput using a large number of simple cores, while compromising the latency on individual cores [1]. A processor with such large number of cores is called a manycore processor. The cores may be homogeneous or heterogeneous. New architectural innovations for faster on-chip communication and efficient power management are necessary to scale to a large number of cores as compared with today's multicore processors.

In recent years, a few manycore prototypes have emerged (*e.g.*, 80 core processor called Polaris [2], Larrabee [3], Intel Single-chip Cloud Computer (SCC) [4]). Of particular interest to us is the Intel SCC, an experimental manycore processor from Intel Labs. This processor has new architectural features, namely, 48 Pentium cores, a high speed, on-chip mesh network to communicate between cores and access memory controllers, on-chip message passing buffers for high speed message passing, and software controlled fine-grained power management.

In this work, we attempt to understand the benefits and limitations of the SCC for parallel RDF query processing. RDF (Resource Description Framework) is a popular language for representing data on the Web [5]. It enables the interchange and machine processing of data by considering its semantics. The essence of RDF lies in the notion of representing any fact as *subject*, *predicate*, and *object*. Formally, RDF represents resources as a directed, labeled graph where a pair of adjacent nodes denotes two things and the directed, labeled edge represents their relationship. The source node denotes the subject; the sink node denotes the object; and the edge label is the predicate (or property). This "subject-predicate-object" relationship is commonly referred to as an RDF triple. SPARQL is a popular query language for RDF graphs [6]. Using SPARQL, complex graph pattern queries can be expressed on individual RDF graphs as well as across multiple RDF graphs.

In recent years, the RDF data model has become increasingly important in domain-specific applications and the WWW. Through RDF technologies, one can reason over semantic data, which is highly appealing in domains such as healthcare, defense and intelligence, biopharmaceuticals, and so forth. With the rapidly growing size of RDF datasets (*e.g.*, DBPedia [7], Billion Triples Challenge [8]), there is a pressing need for high performance RDF processing tools. With the emergence of manycore processors, it is natural and timely to ask whether *a manycore processor can boost the performance of RDF query processing* – through parallel processing. To the best of our knowledge, there is no published work in this area. Recent studies on the Intel SCC have focused on low level aspects such as on-chip message passing performance, memory access latency, and power and energy consumption on benchmarks from high performance computing [4], [9].

In our study, we adopt standard task parallel and data parallel programming models for parallel RDF query processing. We categorize RDF queries on real and synthetic RDF datasets into two different query workloads based on their I/O footprint – one with small I/O footprint queries and the other with large I/O footprint queries. We study the effect of inter-query parallelism via the task parallel programming model on these workloads. We also study the effect of intra-query parallelism via the data parallel programming model on these workloads.

The rest of the paper is organized as follows. We present background and related work in Section II; we present the methodology of our study in Section III; we present the empirical findings in Section IV; and we conclude in Section V with a note on future work.

II. PRIOR WORK ON RDF QUERY PROCESSING

Today, there are a number of open-source and commercial tools for storing and querying RDF graphs. These tools either

V. Slavov is with the Department of Computer Science and Electrical Engineering, University of Missouri-Kansas City. E-mail: vgslavov@mail.umkc.edu

P. Rao is with the Department of Computer Science and Electrical Engineering, University of Missouri-Kansas City. E-mail: raopr@umkc.edu

D. Barenkala is with the Department of Computer Science and Electrical Engineering, University of Missouri-Kansas City. E-mail: db985@mail.umkc.edu

S. Paturi is with the Department of Computer Science and Electrical Engineering, University of Missouri-Kansas City. E-mail: sp895@mail.umkc.edu

store and process RDF in main-memory, use an RDBMS, or a native RDF database. The popular approach has been to use relational database systems for storing, indexing, and querying RDF [10], [11], [12], [13], [14], [15]. Abadi *et al.* proposed a vertical partitioning approach and leveraged a column-oriented DBMS for achieving an order of magnitude performance improvement over previous techniques [16].

RDF-3X [17] and Hexastore [18] demonstrated that storing RDF data in a single triples table and building exhaustive indexes on the six permutations of (s, p, o) triples can significantly outperform the vertical partitioning approach [16] and also support a larger class of RDF queries efficiently. Recently, BitMat [19] was proposed to overcome the overhead of large intermediate join results in RDF-3X and Hexastore when queries contain low selectivity triple patterns. (Low selectivity implies large result set size.)

There are some RDF stores that operate in shared-nothing clusters (*e.g.*, YARS2 [20], 4store [21], Clustered TBD [22]) by hashing triples/quadruples and distributing them on different nodes in the cluster. Parallel query processing is performed. The scalability of these approaches has been demonstrated on small sized clusters. Weaver *et al.* [23] have studied RDF query processing on supercomputers. More recently, tools for data intensive computing such as Apache Hadoop and Pig have been used for query processing and analytics over RDF data [24], [25], [26]. These approaches are more suitable for batch processing of queries. A few researchers have focused on parallel RDF reasoning [27], [28]. More recently, Huang *et al.* , developed parallel RDF query processing techniques for large RDF graphs [29].

On the Intel SCC, Vidal *et al.* , studied the parallelization of an AI automated planner using a hash-based distribution of tasks [30]. Petrides *et al.* , studied the performance of relational decision support queries on the SCC [31]. However, none of the previous work has studied the performance impact of parallel RDF query processing on the Intel SCC.

III. OUR METHODOLOGY

In this section, we introduce our methodology for parallel RDF query processing on the SCC. Our first approach is to express inter-query parallelism via the task parallel programming model. Our second approach is to express intra-query parallelism via the data parallel programming model. While we adopt standard techniques for task and data parallelism, these techniques provide good insights into the benefits and limitations of the Intel SCC for RDF query processing. The query workloads we study are I/O bound in nature, unlike prior work on SCC [4], [9], which focused on high performance computing benchmarks. We consider two different types of query workloads: one that has relatively smaller I/O footprint and the other that has relatively larger I/O footprint.

A. Message Passing Interface

We use the popular Message Passing Interface (MPI) for writing parallel programs. MPI contains a standard library of routines for writing portable message-passing based programs. The MPI routines that we used for the task parallel and data parallel programming models are listed in Table I. MPI programs essentially create a collection of processes. MPI_Send and MPI_Recv allow a process to exchange messages with another process (point-to-point communication); MPI_Barrier enables processes to synchronize at certain points during execution; and MPI_Bcast, MPI_Scatter, and MPI_Gatherv are collective communication operations, which allow a process to communicate with a group of other processes.

B. Impact of Granularity

In parallel computing, granularity denotes the ratio between the amount of computation to the amount of communication. In fine-grained parallelism, we break a problem into relatively smaller sized computation tasks and therefore, may require more frequent communication between processors. In coarse-grained parallelism, we break a problem into relatively larger sized computation tasks and therefore reduce the frequency of communication between processors. However, fine-grained parallelism enables better load balancing than coarse-grained parallelism. But it may increase communication cost and synchronization overhead. By design, Intel SCC provides a high speed, on-chip network to enable fast communication between cores. Therefore, we attempt to partition the tasks as fine-grained as possible in our experiments. Because the query workloads we study are I/O bound, we use the I/O footprint to characterize the granularity of a task.

MPI routines	Usage
MPI_Send	Is called when a process wants to send a message in its local buffer to another process
MPI_Recv	Is called when a process wants to receive a message from another process
MPI_Barrier	Is called by a process to enter a barrier
MPI_Bcast	Is called by a process to broadcast the message to all processes in the group
MPI_Scatter	Is called by a process to scatter an array of data items to other processes
MPI_Gatherv	Is called by all processes in the group (one receiver, multiple senders) so that the receiver can collect different sized messages from the senders synchronously

TABLE I
MPI ROUTINES USED

C. Task Parallel Programming Model

We express inter-query parallelism via a straightforward task parallel programming model. Each query is regarded as a task. Our model is as follows. On one core, we run the master and on the other cores, we run workers. Algorithm 1 describes the set of actions performed by the master and workers. Lines 1-1 denote the actions taken by the master. Lines 1-1 denote the actions taken by a worker. The master maintains a single task pool. Once the master and workers have started (as MPI processes), each worker sends a message to the master. The master responds to a worker with a query from the task pool. The worker then executes the query locally on the index. (The index is constructed over the entire dataset and is shared by the workers.) Once completed, the worker

Algorithm 1: Task parallel programming model

```
proc @Master()
Create a query pool from a list of SPARQL queries to process
while query pool is not empty do
    MPI_Recv(workerid)
    Remove a SPARQL query q from the pool
    MPI_Send(workerid, q)
execute MPI_Barrier
end
proc @Worker()
while true do
    MPI_Send(master) to request a query
    q ← MPI_Recv(master)
    if q == EOF then
        break
    else
        Execute q locally using the index
execute MPI_Barrier
end
```

repeats the process of requesting and executing queries from the master until the master informs that there are no more queries to execute. Once the master and workers reach the barrier, the task pool has been completely processed.

D. Data Parallel Programming Model

We express intra-query parallelism via a straightforward data parallel programming model. The task of processing a query on the entire dataset is broken down into subtasks, where each subtask consists of executing the query on a different partition of the dataset. Our model for data parallelism is as follows. First, we partition the underlying RDF graph into smaller graphs. We do this by extracting weakly connected directed subgraphs and applying standard graph partitioning techniques if necessary (*e.g.*, METIS [32]). If graph partitioning is applied, then we aim to minimize the number of cut edges. We replicate the cut edges in the partitions. (We ignore the directionality of the edges in the graph while partitioning and assume each edge has unit weight.) In our approach, we may miss results. While overcoming this is a non-trivial research challenge, our goal here is to test whether using partitioned indexes on multiple cores during query processing can provide good speedup for the best case scenario.

Similar to the task parallelism approach described earlier, on one core we run the master, and on the others we run workers. The master selects a query and broadcasts it to the workers and also provides each worker with a bucket id to use during query processing. Each worker executes the query locally on the data in the specified bucket. The partial results are returned to the master. Collecting the results can be done either by sending multiple messages one at a time to the master or using the collective operation MPI_Gatherv. Algorithm 2 describes the steps involved. The master and the workers reach a barrier before the next query is processed.

IV. PERFORMANCE EVALUATION

We used RDF-3X [17], a state-of-the-art RDF query processing engine in our evaluation. RDF-3X was implemented in C++ and was compiled to run on the SCC using a 32 bit GCC compiler. The SCC cores ran Linux and had a NFS mounted file system where the indexes were stored. We did not modify the memory organization/configuration of the SCC and used the default setting.

Algorithm 2: Data parallel programming model

```
proc @Master()
foreach SPARQL query q do
    Let Bid[ ] denote an array of bucket ids
    MPI_Scatter(Bid[ ]) /* Send a different bucket id to each worker */
    MPI_Bcast(q) /* Send the same query to each worker */
    MergeResults()
end
proc @Worker()
while true do
    p ← MPI_Scatter() /* A worker receives one bucket id */
    q ← MPI_Bcast() /* Every worker receives the same query */
    Execute q locally on the index for bucket p
    MergeResults()
end
proc MergeResults()
if Master then
    Collect results from workers using multiple MPI_Recv or single MPI_Gatherv
else
    Send results to master using multiple MPI_Send or single MPI_Gatherv
execute MPI_Barrier
end
```

We implemented the task and data parallel programming models described in Algorithms 1 and 2 using RCKMPI, a modified MPICH2 for the Intel SCC [33]. RCKMPI uses the message passing buffers (MPBs) in the SCC to allow low latency high bandwidth message passing. The SCC platform was initialized to run with tile frequency of 800 MHz, mesh frequency of 800 MHz, and memory controller frequency of 800 MHz.

A. Dataset and Queries

We used two real datasets, namely, YAGO2 [34] and Uniprot [35]. YAGO2 is a semantic knowledge base derived from Wikipedia, WordNet, and Geonames. Uniprot is a comprehensive resource for protein sequence and annotation data. We also generated a synthetic dataset using the Lehigh University Benchmark (LUBM) [36]. The ontology for this dataset is based on a university domain.

Note that the SCC cores generate 32 bit addresses. RDF-3X leverages memory mapping of index files and therefore, recommends 64 bit processors for indexing and querying large RDF datasets. To cope with the 32 bit addressing on the SCC, we indexed a set of triples in each dataset such that the index size was at most 2GB in size, a limit set by the underlying OS. This ensured that RDF-3X successfully ran the queries on the SCC. For YAGO, we indexed 27,331,797 triples; for Uniprot, we indexed 46,972,851 triples; and for LUBM, we

Query	Dataset	I/O footprint	Type	% CPU	Serial time
QY_1	YAGO	14,756 KB	small	29	4.73 secs
QY_2	YAGO	15,004 KB	small	40	9.23 secs
QY_3	YAGO	22,832 KB	small	29	6.51 secs
QY_4	YAGO	33,492 KB	small	21	9.27 secs
QY_5	YAGO	216,564 KB	large	22	82.65 secs
QY_6	YAGO	272,848 KB	large	30	120.08 secs
QY_7	YAGO	332,944 KB	large	43	218.43 secs
QL_1	LUBM	2,668 KB	small	25	1.4 secs
QL_2	LUBM	3,132 KB	small	35	1.47 secs
QL_3	LUBM	9,804 KB	small	19	3.5 secs
QL_4	LUBM	636,204 KB	large	32	299.99 secs
QL_5	LUBM	673,924 KB	large	29	206.58 secs
QU_1	Uniprot	4,468 KB	small	39	2.08 secs
QU_2	Uniprot	10,344 KB	small	39	6.46 secs
QU_3	Uniprot	48,020 KB	large	31	19.39 secs
QU_4	Uniprot	62,188 KB	large	19	15.48 secs
QU_5	Uniprot	166,808 KB	large	17	43.51 secs

TABLE II
INITIAL EVALUATION OF QUERIES

indexed 35,612,176 triples. (The SPARQL queries used for the experiments are listed in a technical report [37].)

B. Query Workload Classification

The queries used in our evaluation are I/O bound in nature. Using the `iostat` command, we measured the I/O footprint of each query. (We dropped the file system buffer cache before running each query by issuing `echo 3 > /proc/sys/vm/drop_caches`.) Based on the I/O footprint, we classified the queries into two categories, `small` and `large`. Queries that were classified `small` had relatively smaller I/O footprint. Queries that were classified `large` had relatively larger I/O footprint. Table II shows the queries and their classification after running each query serially. (The block size used by the filesystem was 4096 bytes.) In addition, the serial time (on a single core) and the % CPU utilization for each query is shown. Queries that had higher CPU utilization (*e.g.*, QY_7), typically returned more results. Note that internally RDF-3X stores long string literals in a *mapping dictionary*, and uses ids in the indexes. At the end of query processing, it maps back these ids to literals using the dictionary. For queries returning large number of results, this cost of mapping becomes non-negligible [17].

C. Evaluation Metrics

We measured the effectiveness of parallel RDF query processing by computing the *speedup* and *efficiency* as the number of available cores was increased. Suppose T_s is the time taken to execute a workload of SPARQL queries on a single SCC core. Suppose T_p is the time taken to execute the queries in parallel (using either data or task parallel programming models) on n SCC cores. (On n cores, we run one master and $n-1$ workers.) The speedup on n cores is computed by the ratio $\frac{T_s}{T_p}$; the efficiency on n cores is computed by the ratio $\frac{speedup}{n}$.

D. Data Partitioning Approach

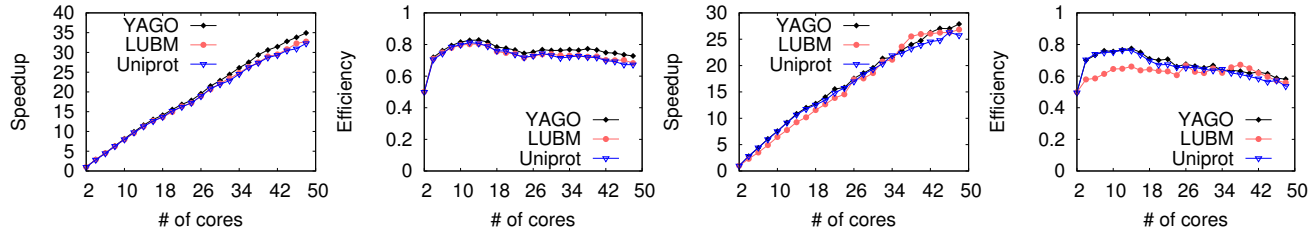
For the data parallel programming model, we partitioned a dataset depending on how many cores were available to run the workers. (Note that partitioning was done once before executing all the queries.) Each worker was assigned one partition and used the index for that partition during query processing. Different approaches were followed for each of the three datasets. The main goal was to assign the triples corresponding to weakly connected directed subgraphs in the RDF graph into buckets. For LUBM, as the generator produced separate RDF files, we grouped the triples from one file and placed it in a bucket. All the files were distributed across the buckets in a round-robin fashion. For Uniprot, we had one single XML/RDF file, and we created fragments of this XML file at points where a new protein was described. The triples from each fragment were stored together in a bucket. All the fragments were distributed across the buckets in a round-robin fashion.

The YAGO2 dataset was available in N-Triples format. First, we extracted graphs of a particular type from the dataset, which we call *star-shaped* graphs. A star-shaped graph is a weakly connected directed graph, where the degree of all vertices except one is exactly 1. All the triples from a star-shaped graph were put into a bucket. On the remaining non-star graphs, we ran the METIS [32] algorithm to partition the graphs. After obtaining n partitions, we assigned the triples for each partition into one bucket. (We replicated the cut edges in each partition.) As mentioned earlier, our approach may miss results.

E. Results

We focus on four possible combinations of workload and parallel programming models, namely, ST (small I/O footprint, task parallelism), LT (large I/O footprint, task parallelism), SD (small I/O footprint, data parallelism), and LD (large I/O footprint, data parallelism). We will refer to these as the ST, LT, SD, and LD models in subsequent discussions. Note that all I/O requests go through the MCPC connected to the SCC platform via the PCIe bus. We measured wall clock time by ensuring a cold cache scenario. (We dropped the file system buffer cache before a query was executed on a core.)

1) The ST Model: The query workload for each dataset consisted of queries marked `small` in Table II. The task pool consisted of these queries put in order and scaled by a factor of 100. (For example, the task pool for YAGO consisted of queries QY_1, QY_2, QY_3, QY_4, ..., QY_1, QY_2, QY_3, QY_4, ...) Figure 1(a) shows the speedup obtained for parallel RDF query processing using Algorithm 1. On 48 cores (1 master + 47 workers), a promising speedup of 34.92, 32.74, and 32.27 was obtained for YAGO, LUBM, and Uniprot, respectively. Figure 1(b) shows the efficiency. For all three datasets, the efficiency reached close to 70% on 48 cores. The tasks were relatively fine-grained due to their small I/O footprints and were well distributed across the workers. There was effective load balancing of tasks across the workers resulting in good speedup and efficiency. (This is evident from the mean and standard deviation of the number of tasks processed by each



(a) Speedup (ST model) (b) Efficiency (ST model) (c) Speedup (LT model) (d) Efficiency (LT model)

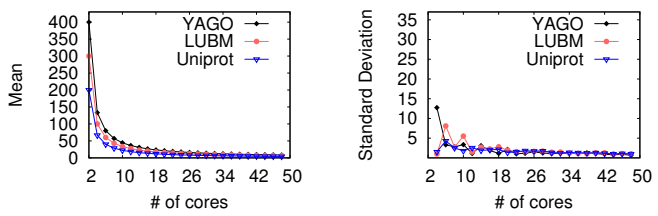
Fig. 1. Results for the task parallel programming model

worker as shown in Figures 2(a) and 2(b).) As shown in Figure 4, the average CPU utilization varied marginally (from 2 to 48 cores), indicating negligible I/O contention in ST.

2) The LT Model: The query workload for each dataset consisted of queries marked `large` in Table II. Similar to ST, the task pool consisted of these queries put in order and scaled by a factor of 33, 50, and 33 for YAGO, LUBM, and Uniprot, respectively. Figures 1(c) and 1(d) show the speedup and efficiency for parallel RDF query processing using Algorithm 1. On 48 cores, the speedup ranged between 25 to 30 for the three datasets. This is promising given that the queries had larger I/O footprint than those used in the ST model. The load was fairly well distributed across the workers. (See Figures 3(a) and 3(b).) As shown in Figure 4, the drop in the average CPU utilization (from 2 to 48 cores) was higher for LUBM and Uniprot as compared to YAGO, indicating higher I/O contention for these datasets.

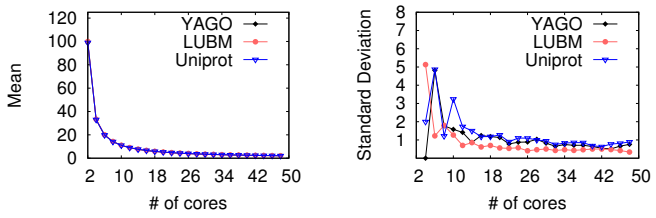
3) The SD Model: The query workload for each dataset consisted of queries marked `small` in Table II. Each query was run multiple times using Algorithm 2. Although the data parallel approach created fine-grained tasks for a query with increasing number of cores, there was load imbalance as many of the workers returned no results on their partitions. This resulted in poor speedup and efficiency as the number of cores was increased. We show the plots in Figures 5(a) and 5(b).

4) The LD Model: The query workload for each dataset consisted of queries marked `large` in Table II. Each query was run multiple times using Algorithm 2. As more cores were used to process a query, I/O contention became an issue. This is evident from the fact that the average CPU utilization for LD was lower than that for LT on all datasets. As a result, poor speedup and efficiency were obtained. We show the plots in Figures 5(c) and 5(d).



(a) Mean (# of tasks per worker) (b) Standard deviation

Fig. 2. Load distribution - ST model



(a) Mean (# of tasks per worker) (b) Standard deviation

Fig. 3. Load distribution - LT model

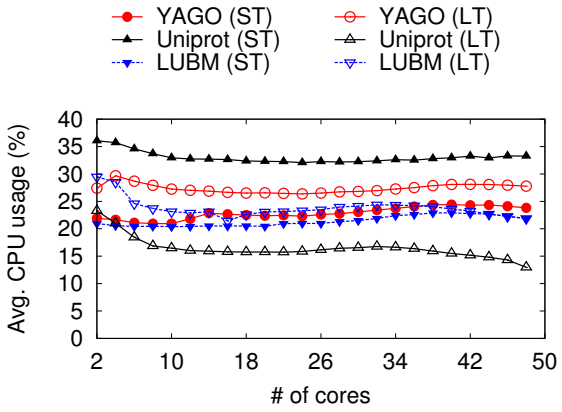


Fig. 4. Average CPU usage with increasing number of cores

F. Summary of Results on the SCC

- The task parallel programming model yielded good speedup and efficiency for parallel RDF query processing. This was true for both small I/O and large I/O footprint queries. The ST model, however, gave better results than the LT model.
- Although the data parallel programming model created fine-grained tasks, the speedup and efficiency for both the SD and LD models were poor due to either load imbalance or I/O contention. Further research is necessary to address these issues.

V. Conclusions and Future Work

We have presented an early study of the performance of parallel RDF query processing on the Intel SCC, an experimental manycore processor. Using real and synthetic RDF datasets, we studied how inter-query parallelism (via

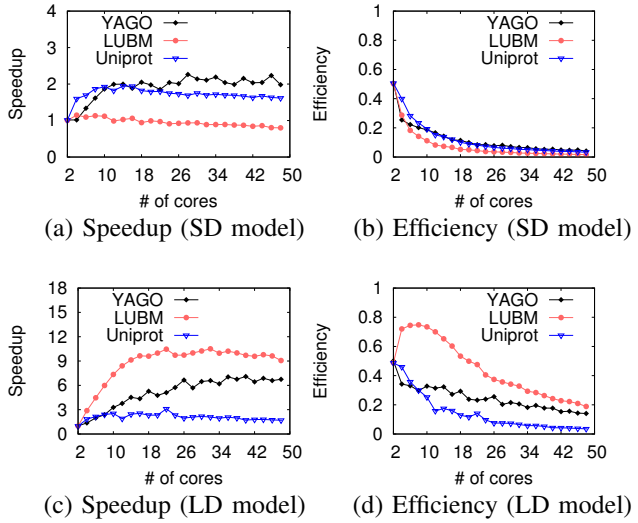


Fig. 5. SD and LD models

the task parallel programming model) and the intra-query parallelism (via the data parallel programming model) affected the performance of RDF query processing. We conclude that the task parallel model provides an immediate way to boost the query processing performance. In the future, we plan to develop new RDF query processing strategies to overcome the challenges posed by the data parallel programming model. We would also like to study the effect of dynamic voltage and frequency scaling of the SCC cores on the performance of RDF query processing.

ACKNOWLEDGEMENTS

We are grateful to the MARC team at Intel Labs for granting us access to the SCC hardware and related software tools. Special thanks to Mark Aughenbaugh and Ted Kubaska for their prompt help. This work was supported in part by a grant from the National Science Foundation (IIS-1115871).

REFERENCES

[1] M. Garland and D. B. Kirk, "Understanding Throughput-Oriented Architectures," *Commun. of ACM*, vol. 53, pp. 58–66, November 2010.
[2] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar, "An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS," in *Solid-State Circuits Conference*, 2007, pp. 98–589.
[3] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: A Many-Core x86 Architecture for Visual Computing," *ACM Transactions on Graphics*, vol. 27, pp. 18:1–18:15, August 2008.
[4] T. Mattson, R. Van der Wijngaart, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe, "The 48-core SCC Processor: the Programmer's View," in *Proc. of Intl. Conf. for High Performance Computing, Networking, Storage and Analysis*, Nov 2010, pp. 1–11.
[5] "Resource Description Framework (RDF)," http://www.w3.org/RDF.
[6] "SPARQL Query Language for RDF," http://www.w3.org/TR/rdf-sparql-query/.
[7] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann, "DBpedia - A crystallization point for the Web of Data," *Journal of Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 7, no. 3, pp. 154–165, September 2009.
[8] "Semantic Web Challenge," http://challenge.semanticweb.org/.
[9] P. Gschwandtner, T. Fahringer, and R. Prodan, "Performance Analysis and Benchmarking of the Intel SCC," in *Proc. of Intl. Conf. on Cluster Computing*, Sept. 2011, pp. 139–149.
[10] K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds, "Efficient RDF Storage and Retrieval in Jena2," in *Proc. of SWDB'03*, 2003, pp. 131–150.
[11] S. Harris and N. Gibbins, "3store: Efficient Bulk RDF Storage," in *Practical and Scalable Semantic Systems*, 2003.
[12] J. Broekstra, A. Kampman, and F. van Harmelen, "Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema," in *Proc. of ISWC '02*, pp. 54–68.
[13] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan, "An Efficient SQL-Based RDF Querying Scheme," in *Proc. of the 31st VLDB Conference*, Trondheim, Norway, 2005, pp. 1216–1227.
[14] L. Ma, Z. Su, Y. Pan, L. Zhang, and T. Liu, "RStar: an RDF storage and query system for enterprise resource management," in *Proc. of CIKM '04*, Washington, D.C., USA, 2004, pp. 484–491.
[15] J. J. Levandoski and M. F. Mokbel, "RDF Data-Centric Storage," in *Proc. ICWS '09*, Washington, DC, 2009, pp. 911–918.
[16] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach, "Scalable Semantic Web Data Management Using Vertical Partitioning," in *Proc. of the 33rd VLDB Conference*, 2007, pp. 411–422.
[17] T. Neumann and G. Weikum, "The RDF-3X engine for scalable management of RDF data," *The VLDB Journal*, vol. 19, no. 1, pp. 91–113, 2010.
[18] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: Sextuple Indexing for Semantic Web Data Management," *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 1008–1019, 2008.
[19] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler, "Matrix "Bit" Loaded: A Scalable Lightweight Join Query Processor for RDF Data," in *Proc. of the 19th Intl. Conference on World Wide Web*, Raleigh, North Carolina, USA, 2010, pp. 41–50.
[20] A. Harth, J. Umbrich, A. Hogan, and S. Decker, "YARS2: A Federated Repository for Querying Graph Structured Data From the Web," in *Proc. of ISWC'07/ASWC'07*, Busan, Korea, 2007, pp. 211–224.
[21] S. Harris, N. Lamb, and N. Shadbolt, "4store: The Design and Implementation of a Clustered RDF Store," in *Proc. of 5th Intl. Workshop on Scalable Semantic Web Knowledge Base Systems*, 2009, pp. 94–109.
[22] A. Owens, A. Seaborne, N. Gibbins, and M. Schraefel, "Clustered TDB: A Clustered Triple Store for Jena," in *Technical Report, Electronics and Computer Science, University of Southampton*, 2008.
[23] J. Weaver and G. T. Williams, "Scalable RDF Query Processing on Clusters and Supercomputers," in *Proc. of the 5th International Workshop on Scalable Semantic Web Knowledge Base Systems*, 2009.
[24] P. Castagna, A. Seaborne, and C. Dollin, "A Parallel Processing Framework for RDF Design and Issues," HP Labs, Bristol, Tech. Rep., 2009, www.hpl.hp.com/techreports/2009/HPL-2009-346.pdf.
[25] R. Sridhar, P. Ravindra, and K. Anyanwu, "RAPID: Enabling Scalable Ad-Hoc Analytics on the Semantic Web," in *Proc. of ISWC '09*, 2009, pp. 715–730.
[26] M. F. Husain, J. McGlothlin, M. M. Masud, L. R. Khan, and B. Thuraisingham, "Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing," *IEEE Transactions on Knowledge and Data Engineering*, vol. 23, pp. 1312–1327, 2011.
[27] J. Weaver and J. A. Hendler, "Parallel Materialization of the Finite RDFS Closure for Hundreds of Millions of Triples," in *Proc. of ISWC '09*, Chantilly, VA, 2009, pp. 682–697.
[28] J. Urbani, S. Kotoulas, E. Oren, and F. Harmelen, "Scalable Distributed Reasoning Using MapReduce," in *Proc. of ISWC '09*, 2009, pp. 634–649.
[29] J. Huang, D. J. Abadi, and K. Ren, "Scalable SPARQL Querying of Large RDF Graphs," *PVLDB*, vol. 4, no. 11, pp. 1123–1134, 2011.
[30] V. Vidal, S. Vernhes, and G. Infantes, "Parallel AI Planning on the SCC," in *4rd Many-core Applications Research Community Symposium (MARC 2011)*, Potsdam, Germany, 2011, pp. 1–6.
[31] P. Petrides, A. Diavastos, and P. Trancoso, "Exploring Database Workloads on Future Clustered Many-Core Architectures," in *3rd Many-core Applications Research Community Symposium (MARC 2011)*, Ettlingen, Germany, 2011, pp. 81–84.
[32] G. Karypis and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM Journal on Scientific Computing*, vol. 20, pp. 359–392, December 1998.
[33] I. A. C. Urena, "RCKMPI User Manual," *Intel Braunschweig*, 2011.
[34] J. Hoffart, F. M. Suchanek, K. Berberich, E. Lewis-Kelham, G. de Melo, and G. Weikum, "YAGO2: Exploring and Querying World Knowledge in Time, Space, Context, and Many Languages," in *Proc. of WWW '11*, 2011, pp. 229–232.
[35] "Uniprot RDF Distribution," http://www.uniprot.org/downloads.
[36] Y. Guo, Z. Pan, and J. Heflin, "LUBM: A benchmark for OWL knowledge base systems," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 3, pp. 158–182, October 2005.
[37] V. Slavov, P. Rao, D. Barenkala, and S. Paturi, "Towards RDF Query Processing on the Intel SCC," University of Missouri-Kansas City, Tech. Rep. TR-DB-2012-01, 2012, http://r.web.umkc.edu/raopr/TR-DB-2012-01.pdf.