Main Memory Evaluation of Recursive Queries on Multicore Machines

Mohan Yang
Department of Computer Science
University of California, Los Angeles
Email: yang@cs.ucla.edu

Carlo Zaniolo
Department of Computer Science
University of California, Los Angeles
Email: zaniolo@cs.ucla.edu

***Abstract*—Supporting iteration and/or recursion for advanced big data analytics requires reexamination of classical algorithms on modern computing environments. Several recent studies have focused on the implementation of transitive closure in multi-node clusters. Algorithms that deliver optimal performance on multi-node clusters are hardly optimal on multicore machines. We present an experimental study on finding efficient main memory recursive query evaluation algorithms on modern multi-core machines. We review SEMINAIVE, SMART and a pair of single-source closure (SSC) algorithms. We also propose a new hybrid SSC algorithm, named SSC12, which combines two previously known SSC algorithms. We implement these algorithms on a multicore shared memory machine, and compare their memory utilization, speed and scalability on synthetic and real-life datasets. Our experiments show that, on multicore machines, the surprisingly simple SSC12 is the only transitive-closure algorithm that is consistently fast and memory-efficient on all test graphs.**

***Keywords*-main memory; multicore; recursive query; transitive closure**

I. INTRODUCTION

The growing importance of applications using big data analytics is promoting a burgeoning research interest in parallel systems, algorithms, and software needed to achieve scalable performance on such systems.

As we have moved from the simple applications originally supported by MapReduce to more advanced applications requiring iteration and/or recursion, it is now clear that classical algorithms designed for traditional architectures should be re-evaluated and re-designed for these new massively parallel systems. For instance, several recent studies [1]–[3] have focused on the implementation of transitive closure (TC) in multi-node clusters, and Afrati et al. [1] showed that a relatively obscure algorithm called SMART [4], [5] outperforms other algorithms on this problem.

However, algorithms that deliver optimal performance on multi-node clusters are hardly optimal on multicore machines, and vice versa: in the rest of the paper we demonstrate this point by an in-depth experimental study of various transitive closure algorithms on multicore machines. Thus, we will first show that many other algorithms are significantly better than SMART, and then propose a hybrid algorithm that achieves the best performance by combining two existing algorithms.

The novel performance findings presented in this paper are hardly surprising in view of the fact that the most previous studies date back to the late 80's and early 90's [6]–[9], and there has been much progress in multicore systems since then. Moreover, we assume here that all our data resides in main memory, whereas past studies on recursive query evaluation [4]–[17] often assumed a database-oriented environment with data residing on secondary storage, whereby query evaluation algorithms were designed to reduce I/O costs rather than in-memory evaluation costs.

The main short-term benefits of the findings presented in this paper are that programmers of multicore systems will be able to use these results by selecting the best algorithm for their transitive-closure-like applications. But our longer-term objective is to enable the compiler to select the best implementation for the the systems at hand, or the most cost-effective configuration when many choices are available as it is often the case in cloud-computing environments. This ambitious objective represents a natural extension of the query optimization approach that has made possible efficient and parallel implementations of declarative database queries. Keeping with this database motif, we will express many recursive queries using Datalog. Datalog entails a concise and elegant expression of these queries; also Datalog's compilation technology combines the well-known query optimization techniques of DBMS with those of recursive programs which are transformed into equivalent ones optimized for the particular query at hand (i.e., techniques such as left-recursion to right-recursion transformations and magic-sets), and then mapped into equivalent iterative ones (e.g., via the seminaive fixpoint computation [18]).

Therefore in this paper, we will express several TC algorithms using Datalog, and then evaluate their performance on massively parallel multicore systems. A discussion of transformations between Datalog programs expressing the various algorithms would take us well beyond our space limitations and is left for later papers. The focus of this paper is instead on efficient implementations. We begin with a study of the TC query evaluation using the seminaive fixpoint computation (denoted by SEMINAIVE) and then SMART. Then we study SSC algorithms that decompose the TC computation into disjoint computations, each computing the closure from a single vertex using linear recursive rules. The two SSC algorithms are SSC1 based on SEMINAIVE, and SSC2 based on the breadth-first search. We also propose a new hybrid SSC algorithm, called

978-1-4799-5666-1/14/$31.00 ©2014 IEEE

SSC12, which integrates the merits of SSC1 and SSC2. We perform an experimental evaluation of our implementations focusing on memory utilization, speed and scalability.

A first contribution of this paper is an extensive experimental comparative evaluation of these parallel algorithms. We show that the little known SSC algorithms outperform other well-studied algorithms in terms of memory utilization and speed. We also identify the crux that prevents each algorithm from scaling linearly. All these algorithms are memory-bandwidth bound, and SEMINAIVE is also bound by control time. Although recursive query evaluation benefits from the current trend towards massive parallel multicore systems, this negative result suggests that it will be difficult to continue benefiting from this trend even if the number of cores per CPU keeps growing.

The second contribution is the proposal of a robust main-memory recursive query evaluation algorithm. The surprisingly simple SSC12 proposed in this paper is the only algorithm that consistently performs well on all test graphs for the TC query evaluation. This simple and efficient algorithm is also applicable to many other recursive queries that express computation similar to TC, including shortest path and similar algorithms expressed using monotonic aggregates [19].

The rest of our paper is organized as follows. Section II describes all compared algorithms, and Section III presents our parallel implementations. Section IV reports the experimental setup and results. Section V discusses related work. The paper concludes in Section VI.

II. TRANSITIVE CLOSURE ALGORITHMS

A. Linear TC Rules and Seminaive Evaluation

Let $\texttt{arc}(\texttt{X},\texttt{Y})$ be a relation that represents the edges of a directed graph, i.e., there is a directed edge from x to y if and only if $\texttt{arc}(\texttt{x},\texttt{y})$ is a fact. The transitive closure (TC) of arc is a relation $\texttt{tc}(\texttt{X},\texttt{Y})$ such that tc contains all pairs $(\texttt{X},\texttt{Y})$ that X can reach Y via a path in the graph. A linear version of TC is given by rules in (1).

$$\begin{aligned}&\texttt{tc}(\texttt{X},\texttt{X}) \texttt{ <- } \texttt{arc}(\texttt{X},\_).\\&\texttt{tc}(\texttt{X},\texttt{Y}) \texttt{ <- } \texttt{tc}(\texttt{X},\texttt{Z}),\texttt{arc}(\texttt{Z},\texttt{Y}).\end{aligned} \tag{1}$$

The first rule is an exit rule[1]. It adds a tuple $(\texttt{X},\texttt{X})$ to tc for each source vertex X. The second rule is a left-linear recursive rule. For every tuple $(\texttt{X},\texttt{Z})$ that is already in tc, it expands the path represented by $(\texttt{X},\texttt{Z})$ with one more edge, then adds the new tuple to tc. The relation tc can be computed by iteratively performing this operation until no more new tuples can be added to it. This procedure is called the *naive evaluation*. This simple procedure may involve redundant derivations since the same path may be generated in several iterations.

[1]A more common way to write the exit rule is $\texttt{tc}(\texttt{X},\texttt{Y})$ <- $\texttt{arc}(\texttt{X},\texttt{Y})$. The exit rule here ensures that tc always contains the tuple $(\texttt{X},\texttt{X})$ for each source vertex X.

SEMINAIVE *Algorithm:* Seminaive evaluation [18] is an optimized variant of naive evaluation. The idea is to use only the new tuples derived in the previous iteration to derive the tuples in the current iteration. The pseudocode is shown in Fig. 1.

```
1: tc := {(X,X)|arc(X,_)}, Δtc := {(X,X)|arc(X,_)}
2: repeat
3:     δtc := π_{X,Y}(Δtc(X,Z) ⋈ arc(Z,Y))
4:     Δtc := δtc − tc
5:     tc := tc ∪ Δtc
6: until Δtc = ∅
```

Fig. 1. SEMINAIVE algorithm for computing tc.

SEMINAIVE evaluates as follows. First, all source vertices in arc are added into tc and Δtc, i.e., tc and Δtc contain tuples for all paths of length 0. Then, in the i-th iteration, the initial Δtc contains tuples for paths of length $i-1$. δtc contains tuples corresponding to paths of length i derived from extending paths of length $i-1$ with one more edge. However, some tuples may already be present in tc. It is not necessary to do derivations on these tuples since it will only derive tuples that are already derived. So the new Δtc excludes these tuples that are already in tc. All tuples in the new Δtc are merged into tc. The algorithm iterates until Δtc becomes empty.

The number of iterations required by SEMINAIVE equals the length of the longest simple path in the graph. The maximal value is $n-1$ for a graph of n vertices. Thus, the algorithm could take $O(n)$ iterations to terminate.

B. Non-Linear TC Rules and the SMART Algorithm

The number of iterations required by SEMINAIVE is large when the length of the longest simple path is very long. However the use of quadratic recursive rules, as shown below, doubles the length of the paths at each iteration thus reducing the number of iterations required to their logarithm.

$$\begin{aligned}&\texttt{tc}'(\texttt{X},\texttt{X}) \texttt{ <- } \texttt{arc}(\texttt{X},\_).\\&\texttt{tc}'(\texttt{X},\texttt{Y}) \texttt{ <- } \texttt{arc}(\texttt{X},\texttt{Y}).\\&\texttt{tc}'(\texttt{X},\texttt{Y}) \texttt{ <- } \texttt{tc}'(\texttt{X},\texttt{Z}),\texttt{tc}'(\texttt{Z},\texttt{Y}).\end{aligned} \tag{2}$$

The SMART algorithm [4], [5] optimizes the computation of these rules by avoiding the generation of the same path multiple times. The pseudocode for SMART is as follows:

```
1: tc' := {(X,X)|arc(X,_)}, δtc' := arc
2: repeat
3:     Δtc' := π_{X,Y}(δtc'(X,Z) ⋈ tc'(Z,Y))
4:     tc' := tc' ∪ Δtc'
5:     δtc' := π_{X,Y}(δtc'(X,Z) ⋈ δtc'(Z,Y)) − tc'
6: until δtc' = ∅
```

Fig. 2. SMART algorithm for computing tc′.

At the beginning of iteration i, tc′ contains all tuples corresponding to paths of length at most $2^{i-1}-1$, and δtc′ contains all tuples corresponding to paths of length exactly

2^{i-1} that are not in $\texttt{tc}'$. This condition holds in the first iteration as $\texttt{tc}'$ is set to the set of tuples corresponding to paths of length $2^0 - 1 = 0$ in line 1, and $\delta\texttt{tc}'$ is set to the set of tuples corresponding to paths of length $2^0 = 1$ in line 1. In line 3, all tuples corresponding to paths of length between 2^{i-1} and $2^i - 1$ are derived by joining $\delta\texttt{tc}'$ and $\texttt{tc}'$. These tuples are merged into $\texttt{tc}'$ in line 4. Now $\texttt{tc}'$ contains all tuples corresponding to paths of at most $2^i - 1$. In line 5, $\delta\texttt{tc}'$ is joined with itself to derive all tuples corresponding to paths of length exactly 2^i. Tuples that are already in $\texttt{tc}'$ are excluded from the new $\delta\texttt{tc}'$. So the condition still holds for iteration $i + 1$. The algorithm iterates until $\delta\texttt{tc}'$ becomes empty. In an n-vertex graph, there is no simple paths of length n. Thus, $\delta\texttt{tc}'$ is empty at the end of iteration $\lceil \log n \rceil$, or the algorithm terminates in $O(\log n)$ iterations.

C. Single-Source Closure Algorithms: SSC1 and SSC2

The previous algorithms compute tuples from different source vertices at the same time, whereas a more parsimonious usage of memory can be achieved by computing the paths that originate from one source vertex one at the time [15].

In Datalog, we can express the optimization by replacing each goal $\texttt{tc}(\texttt{X}, \texttt{Y})$ with the one-column predicate $\texttt{tc}''$ — for each value x that satisfies $\texttt{arc}(\texttt{X}, \_)$:

$$\begin{aligned} &\texttt{tc}''(\texttt{x}). \\ &\texttt{tc}''(\texttt{Y}) \texttt{ <- } \texttt{tc}''(\texttt{Z}), \texttt{arc}(\texttt{Z}, \texttt{Y}). \end{aligned} \tag{3}$$

The closure under the operation defined by rules in (3) contains all vertices reachable from the source vertex x. We call it a single-source closure (SSC). An SSC algorithm computes TC by computing the SSC for every source vertex in the graph. It decomposes the original computation into disjoint computations based on the source vertex.

The SSC1 *Algorithm:* A straightforward way to compute the SSC of a source vertex x is to apply SEMINAIVE to the rules in (3). The algorithm, named as SSC1, is shown in Fig. 3. For each vertex Z in $\Delta\texttt{tc}''$, it finds all the Y that satisfies $\texttt{arc}(\texttt{Z}, \texttt{Y})$, and adds Y to $\delta\texttt{tc}''$. All the vertices that are already in $\texttt{tc}''$ are excluded from the new $\Delta\texttt{tc}''$, and the remaining vertices are added to $\texttt{tc}''$. When the evaluation terminates, $\texttt{tc}''$ contains all the vertices in the SSC of x. We compute the TC by repeating SSC1 on all source vertices.

```
1: tc'' := {x}, Δtc'' := {x}
2: repeat
3:     δtc'' := π_Y(Δtc''(Z) ⋈ arc(Z, Y))
4:     Δtc'' := δtc'' − tc''
5:     tc'' := tc'' ∪ Δtc''
6: until Δtc'' = ∅
```

Fig. 3. SSC1 algorithm for computing $\texttt{tc}''$.

SSC1 performs exactly the same (logical) computation as SEMINAIVE does. The only difference is the computation is partitioned based on the source vertex. The effect of this is similar to hashing. For example, the set difference in line 4 of Fig. 1 is replaced by many set differences in line 4 of Fig. 3 which is equivalent to a hash-based set difference where the hash function simply returns the source vertex of a tuple. As we will see, SSC1 normally outperforms SEMINAIVE which is slower because of the overhead of hashing and related operations.

The SSC2 *Algorithm:* The SSC of x is represented as a set in SSC1. An alternative representation is a Boolean array of size n where the i-th element represents whether x can reach the vertex i in the graph[2]. This array representation converts SSC1 to the SSC2 algorithm shown in Fig. 4. The algorithm essentially performs a breadth-first search starting from x. d, $\Delta\texttt{tc}''$ and $\delta\texttt{tc}''$ are three arrays of size n which are reused throughout the evaluation for all source vertices. Initially, all elements in d are set to **false** except $d[\texttt{x}]$. In each iteration, $\Delta\texttt{tc}''$ and $\delta\texttt{tc}''$ contain the vertices derived in the last iteration and the current iteration, respectively. For each vertex Z in $\Delta\texttt{tc}''$, edges starting from Z are explored to derive new vertices. When a new vertex Y is derived, we check if Y is already in $\texttt{tc}''$ by testing if $d[\texttt{Y}]$ is **true**. If not, we set $d[\texttt{Y}]$ to **true**, and then add Y to $\delta\texttt{tc}''$. The check in line 8 replaces the set difference in line 4 of Fig. 3, while the operation of setting $d[\texttt{Y}]$ to **true** replaces the union in line 5 of Fig. 3. When the algorithm terminates, the actual SSC is constructed by collecting all vertices Y where $d[\texttt{Y}]$ is **true**.

```
 1: set each element in d[] to false
 2: d[x] := true, Δtc''[0] := x, L := 1
 3: repeat
 4:     l := 0
 5:     for i := 0 to L − 1 do
 6:         Z := Δtc''[i]
 7:         for each edge (Z, Y) in arc do
 8:             if d[Y] = false then
 9:                 d[Y] := true, δtc''[l] := Y, l := l + 1
10:     Δtc'' := δtc'', L := l
11: until L = 0
```

Fig. 4. SSC2 algorithm for computing $\texttt{tc}''$.

The array representation allows SSC2 to replace the set operations (insert, set difference and union) with array accesses. This optimization reduces the time of computation (line 2 – 11 in Fig. 4) at the expense of additional time on array initialization (line 1) which is proportional to n. If n is very large, but very few edges are explored during the computation, the time spent on array initialization may be longer than the computation. In this case, SSC2 is slower than SSC1. On the other hand, if the algorithm explores many edges during the computation, the advantage of the array representation becomes clear, and SSC2 becomes faster than SSC1.

D. An Adaptive Single Source Algorithm: SSC12

The performance of the previous two SSC algorithms varies on different source vertices. We propose a hybrid SSC algorithm, named as SSC12, which is a trade-off between SSC1 and

[2]We assume each vertex is encoded as an integer ranging from 0 to $n - 1$ in an n-vertex graph.

SSC2. Evaluation starts with SSC1, and converts to SSC2 when the algorithm predicts that the time would be shorter if it converts to SSC2.

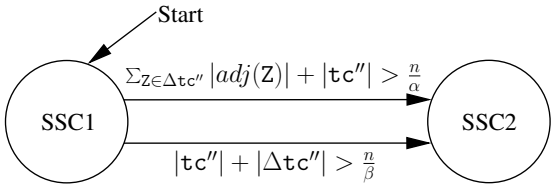

Fig. 5. Control algorithm for SSC12 algorithm.

If SSC2 is faster than SSC1 on a source vertex, the optimal conversion point is the beginning of the evaluation. But the prediction is difficult without computing the SSC. To control the conversion of the hybrid algorithm, we use a heuristic algorithm as shown in Fig. 5. $\delta\mathtt{tc}''$ and $\Delta\mathtt{tc}''$ are represented as sets in SSC1. Assume the cost of set insert and delete is the same. Let $|adj(\mathtt{Z})|$ be the number of Y that satisfies $\mathtt{arc}(\mathtt{Z},\mathtt{Y})$. The number of set operations performed to compute $\delta\mathtt{tc}''$ and $\Delta\mathtt{tc}''$ are $C_\delta = \sum_{\mathtt{Z}\in\Delta\mathtt{tc}''}|adj(\mathtt{Z})|+|\mathtt{tc}''|$ and $C_\Delta = |\mathtt{tc}''| + |\Delta\mathtt{tc}''|$, respectively, which are simple to compute. The algorithm chooses to convert if $C_\delta > n/\alpha$ or $C_\Delta > n/\beta$, where α and β are parameters that control the timing of conversion. It degenerates to SSC1 (SSC2) if $\alpha = \beta = 0$ ($\alpha = \beta = \infty$).

A sufficiently small α ensures that the time becomes shorter if the algorithm chooses to convert since SSC2 is expected to be faster than SSC1 for a large C_δ. But the algorithm may have to execute SSC1 for a long time to obtain such a decision. Thus, the effect of conversion on speedup the computation is diminished as the time of conversion may be too late. On the other hand, large α may lead to wrong predictions which slow down the evaluation. The same dilemma applies to β. We describe how to tune these two parameters in Section IV-B.

III. Algorithm Implementation and Memory Usage

We will next describe in more details the implementation of these algorithms which we then compare in terms of memory utilization, speed and scalability. As discussed in the introduction, the result of this comparison will help programmers implementing recursive applications, and it is actually critical for a compiler optimizing the execution of TC Datalog program on multicore machines. In the previous section, we have seen how a simple rewriting can be used to redirect the execution of linear rules from SEMINAIVE to SSC1, which can in turn be recast as SSC2 and SSC12 (by the compiler using different memory structures). Rewriting rules that transform linear recursive rules into non-linear rules and then these to SEMINAIVE and SMART respectively are also available — although more complex and thus beyond the scope of this paper. Thus all these algorithms represent achievable targets for a parallel Datalog compiler, which will then select the optimal one for the system at hand. Afrati et al. [1] have shown that SMART is optimal for multi-node clusters, and in the rest of the paper we seek to resolve the optimality question for multicore machines.

A. Main Memory Representation

The different algorithms achieve their best performance with different representations. Thus SMART performs well when arc is represented as a collection of tuples, but the performance of SEMINAIVE and the SSC algorithms improves significantly when an *adjacency list* representation is used for arc. The time required for structuring the input data into an adjacency list and building an index for it is included in the total time reported in our experiments (but it is small and only accounts for less than 2% of the total time).

Adjacency List Index: The operation of deriving new tuples in SEMINAIVE can be simply implemented as a nested loop join between $\Delta\mathtt{tc}$ and arc. But it requires a full scan on arc for each tuple in $\Delta\mathtt{tc}$. Note that arc doesn't change during the evaluation. A better strategy is to build an index on arc so that all tuples with a specific source vertex can be accessed directly. The index is an *adjacency list* representation of the graph represented by arc where each vertex X is associated with a unordered list $adj(\mathtt{X})$ describing the set of neighbors of this vertex. It is built by scanning arc twice:

1) The first scan counts how many neighbors each vertex has. These values are stored in an array of size n.
2) A contiguous memory space is allocated for the adjacency list. The starting position within the allocated space for the unordered list associated with each vertex is determined by computing the prefix-sum of the array obtained in step 1).
3) During the second scan, the destination vertex of each tuple is stored in the list associated with the source vertex.

Now, for each tuple $(\mathtt{X},\mathtt{Z})$ in $\Delta\mathtt{tc}$, the algorithm retrieves from the index the list $adj(\mathtt{Z})$ that contains all neighbors of Z. For each vertex $\mathtt{Y} \in adj(\mathtt{Z})$, it generates a new tuple $(\mathtt{X},\mathtt{Y})$. There is no redundant accesses on arc. This index is also used in SSC1, SSC2 and SSC12.

B. Implementation of SEMINAIVE and SMART

Fig. 1 and Fig. 2 contain three basic relational algebra operators: join, union and set difference. The join operator in line 3 of Fig. 1 is implemented as a parallel index nested loop join. We implemented the remaining operators using hash-based parallel relational algebra operators. The join operator in line 3 of Fig. 2 joins two large relations. We implement it using the radix join algorithm [20]. The algorithm first partitions both input relations into smaller relations using multi-pass radix partitioning [21], and then joins each pair of relations resulted from the partitioning. We implement the union operator and the set difference operator similarly — partition both input relations and perform the actual operation. Detailed description of each operator and alternative implementations of both algorithms are discussed in Section 3.1 and Section 4.5 of [22], respectively.

Our implementations use the Pthreads library. Each (working) thread is assigned to a unique physical core on a multicore

machine, and performs all the computation on its assigned core. In addition to the working threads, there is a control thread that coordinates the computation. In each iteration, 1) the control thread assigns the job to each working thread; 2) all working threads perform the computation; 3) the control thread allocates some memory, revokes some memory, assigns the job; 4) each working thread copies the tuples from its local buffer to its assigned location; 5) the control thread decides if the termination condition is satisfied, if not starts the next iteration. The execution time is divided into three parts: 1) *control time*, the time spent by the control thread; 2) *computing time*, when all working threads perform some computation; 3) *copying time*, when each working thread copies tuples to its assigned location.

C. Implementation of SSC Algorithms

Our implementation of different SSC algorithms uses different data structures. In SSC1, we use hash tables to represent $\texttt{tc}''$, $\Delta\texttt{tc}''$ and $\delta\texttt{tc}''$. In SSC2, we use arrays to represent d, $\Delta\texttt{tc}''$ and $\delta\texttt{tc}''$. In SSC12, we use both hash tables and arrays.

An SSC algorithm can be easily parallelized on a shared memory machine as follows: 1) add all source vertices into a queue; 2) each thread removes a vertex from the queue, and computes the SSC of this vertex; 3) repeat until the queue is empty. The queue is implemented as an array with a counter. Each element in the array is a vertex. The initial value of the counter is zero. A thread gets the index of the next vertex by calling the `gcc` atomic memory access function `__sync_fetch_and_add`. Calling the function requires an implicit synchronization between threads that are fetching the counter value at the same time. But the time spent on this synchronization is negligible compared to the control time in SEMINAIVE and SMART.

NUMA Aware Optimization. On non-uniform memory access (NUMA) hardware, the memory is configured into several NUMA regions and each region is attached to a unique CPU as its local memory. A CPU accesses its local memory faster than non-local memory. If the relation `arc` is resident on one CPU's local memory, the threads running on this CPU access the relation faster than the threads running on other CPUs. This unbalanced access speed slows down the computation when we use multiple threads running on different CPUs. If the relation fits in one NUMA region, we can duplicate the relation on each NUMA region such that each thread accesses the (copy of) relation that is resident on the NUMA region attached to the CPU the thread is running on. SEMINAIVE, SSC1, SSC2 and SSC12 adopt this optimization since our experiments are performed on a NUMA machine. The adjacency list index is duplicated on each NUMA region. We discuss the impact of this optimization in Section IV-D1. SMART does not adopt this optimization since `arc` is only used in the first iteration and the time spent on this iteration is very short comparing to the total time.

TABLE I
MEMORY REQUIREMENTS OF IMPLEMENTATIONS.

Algorithm	Memory requirement (bits)
SEMINAIVE	$\geq \mathbf{4bm_c + 2bm}$
SMART	$> \mathbf{6bm_c}$
SSC1	$\leq \mathbf{2bm_c + bm + 6bpn}$
SSC2	$\mathbf{2bm_c + bm + (3b + 1)pn}$
SSC12	$\leq \mathbf{2bm_c + bm + (9b + 1)pn}$

D. Memory Requirements

The main factors determining memory usage are as follows:

n	number of vertices in the graph
m	number of edges in the graph
m_c	number of tuples in the TC
p	number of threads used by an algorithm
b	number of bits to store a vertex

Now, TABLE I summarises the memory requirement of each implementation. It is clear to see the advantage of the SSC algorithms on the memory requirement. They use at most half (one third) of the memory required by SEMINAIVE (SMART) if $m \ll m_c$ (i.e., the TC contains much more tuples than `arc` does). Moreover, the SSC algorithms do not need to access the tuples in the TC. If the TC does not fit in the memory, an SSC algorithm can still executes as long as `arc` fits in the memory and the remaining memory is sufficient to hold the auxiliary data structures used by the algorithm. Instead of storing a newly derived tuple in the main memory, the algorithm now appends it to the end of a file on the disk.

IV. EXPERIMENTAL EVALUATION

All experiments are run on a multicore machine with four AMD Opteron 6376 CPUs and 256GB memory (configured into 8 NUMA regions). Each CPU has 16 cores organized as follows: 1) each core has its own 16KB L1 cache; 2) two cores share a 2MB L2 cache; 3) eight cores share a 6MB L3 cache, and have direct access to 32GB memory. The operating system is Ubuntu Linux 12.04 LTS and the compiler is `gcc` 4.6.3 using `-O3` optimization. Execution time is calculated by taking the average of five runs of the same experiment[3]. Execution time is measured as the number of CPU cycles elapsed from start to finish for computing TC.

In the rest of this section, we first describe the topology of the test graphs, and then describe how the two parameters of SSC12 are determined using some test graphs. Finally, we present the experimental results on serial and parallel execution of the compared algorithms.

A. Topology of Test Graphs

In our experiments, we encode each vertex in an n-vertex graph as a random 32-bit integer ranging from 0 to $n-1$. Each edge is represented as a pair of 32-bit integers. The edges are shuffled into a random order and stored in a file. Recall that the graph has m edges. We say the graph is sparse if m is much less than n^2 (e.g., $m = cn$ or $m = cn \log n$ where c is a

[3]We are not reporting the maximal/minimum execution time since the corresponding line and the average line are almost coincident in the figures.

TABLE II
PARAMETERS OF TEST GRAPHS.

Name	Type	Vertices	Edges	TC size
tree-11	tree	71,391	71,390	876,392
tree-17	tree	13,766,856	13,766,855	251,744,564
grid-150	grid	22,801	45,300	131,698,576
grid-250	grid	63,001	125,500	1,000,203,876
sf-100K	scale-free	100,002	350,604	96,157,950
gnp-0.001	$G(n,p)$	10,000	100,185	100,000,000
gnp-0.01	$G(n,p)$	10,000	999,720	100,000,000
gnp-0.1	$G(n,p)$	10,000	9,999,550	100,000,000
gnp-0.5	$G(n,p)$	10,000	49,986,806	100,000,000
patent	real-life	3,774,769	16,518,948	5,833,193,395
wiki	real-life	1,675,063	2,505,046	8,643,588,110
road	real-life	3,598,623	8,778,114	7,719,381,925
stanford	real-life	281,904	2,312,497	40,044,147,167

small constant). Otherwise, we say it is dense (e.g., $m = cn^2$ or $m = cn^{\frac{3}{2}}$ where c is a small constant). The TC is also a directed graph. Similarly, we say the TC is sparse or dense based on whether the number of tuples in the TC is much less than n^2 or not. We evaluated the algorithms on synthetic graphs of four different topologies and four real-life graphs as shown in TABLE II.

Synthetic Graphs.

1) Tree. We use tree-d to denote a randomly generated tree of depth d such that the out degree of a non-leaf vertex is a random number between 2 to 6. It is a sparse directed graph whose TC is also sparse.

2) Grid. We use grid-d to denote a $(d+1)\times(d+1)$ square grid of $(d+1)^2$ vertices. It is a sparse directed graph but whose TC is dense.

3) Scale-free. The degree distribution of a scale-free graph follows a power law distribution. sf-100K is generated using the scale-free graph generator in the GraphStream library [23]. Its TC is neither as dense as the TC of a grid, nor as sparse as the TC of a tree.

4) $G(n,p)$. An n-vertex $G(n,p)$ graph (Erdős-Rényi model) is generated by connecting vertices randomly such that each pair of vertices are connected with probability p (the graph can have self-loops). We use gnp-p to denote such a random graph of 10,000 vertices with parameter p.

Real World Graphs.

1) patent is the US patent citation graph [24]. Each vertex represents a patent, and each edge represents a citation between two patents. The graph is a directed acyclic graph.

2) wiki is a subgraph of the Wikipedia knowledge graph [25] . Each vertex in the knowledge graph represents an entity in the Wikipedia. If an entity appears in the infobox of another entity, there is a directed edge between the two corresponding vertices. wiki contains 20% of edges and the related vertices from the knowledge graph.

3) road is the eastern USA road network [26]. Each directed edge represents a road between two points in the road network. The graph has a tree structure where the root is a strongly connected component (SCC) consisting of 2141 vertices (about 0.06% of all vertices). All the paths point toward the root.

4) stanford is the Stanford Web graph [27]. Each directed edge represents a hyperlink between two pages under the stanford.edu domain in 2002. The largest SCC in the graph contains about half of the vertices.

Memory Utilization of Algorithms. TABLE III shows the memory utilization of each algorithm on the test graphs. An X mark indicates that an algorithm is not applicable to a graph because the computation requires more memory than the machine has. The memory utilization is usually higher when an algorithm uses more threads. But adjusting the number of threads does not affect whether an algorithm is applicable to a graph in our experiments. Thus, each value in the table represents a typical memory utilization of an algorithm on a test graph (using the *optimal number of threads*, cf. Section IV-D2). Two values are reported for each SSC algorithm — the memory utilization of computing and storing TC in memory, and the memory utilization of computing TC in memory while storing it to disk. For graphs that fit in memory but whose TC cannot fit in memory, the storing-to-disk option allows an SSC algorithm to compute TC without loosing the speed of in-memory computing. For example, the TC of stanford cannot fit in memory, we can still use the SSC algorithms in the storing-to-disk mode as shown by the last row of TABLE III. Moreover, when the query only needs some aggregates on each source vertex, an SSC algorithm can optimize the evaluation by computing the aggregates on each SSC without storing the whole TC. However, there is no such simple optimization for SEMINAIVE and SMART.

TABLE III
MEMORY UTILIZATION OF ALGORITHMS ON TEST GRAPHS (UNIT GB).

	SEMINAIVE	SMART	SSC1	SSC2	SSC12
tree-11	0.06	0.08	0.02/0.02	0.04/0.04	0.02/0.02
tree-17	6.83	8.77	5.57/3.70	8.79/6.91	5.68/3.80
grid-150	3.75	22.39	1.00/0.02	0.99/0.01	1.00/0.03
sf-100K	2.51	21.96	0.76/0.05	0.77/0.06	0.79/0.08
gnp-0.001	10.95	X	0.76/0.02	0.77/0.01	0.79/0.03
gnp-0.01	97.90	X	0.81/0.07	0.79/0.06	0.81/0.07
gnp-0.1	X	X	1.24/0.50	1.22/0.49	1.24/0.50
gnp-0.5	X	X	3.18/2.44	3.16/2.42	3.18/2.44
grid-250	X	X	7.52/0.07	7.48/0.04	7.50/0.05
patent	X	X	44.99/0.37	45.92/0.85	45.96/0.87
wiki	X	X	64.76/1.54	65.24/2.45	65.26/2.52
road	X	X	58.45/0.94	59.50/1.99	58.45/0.94
stanford	X	X	X/0.50	X/0.23	X/0.26

The algorithms can be ordered as follows based on their memory utilization: SMART > SEMINAIVE > SSC1, SSC2, SSC12. The SSC algorithms always use the least memory which is consistent with our analysis in Section III-D. They are applicable to all test graphs, while SEMINAIVE and SMART are only applicable to some test graphs. Although the TCs of all test graphs can fit in memory (except stanford), SEMINAIVE and SMART are not applicable to some test graphs since the intermediate result (δtc in Fig. 1 and Δtc$'$ in Fig. 2) may be extremely large before deduplication. Moreover, Δtc$'$ is usually larger than the corresponding δtcs since Δtc$'$ in the i-th iteration equals the union of δtcs from iteration 2^{i-1} to $2^i - 1$. Thus, SEMINAIVE is applicable to two more graphs than SMART.

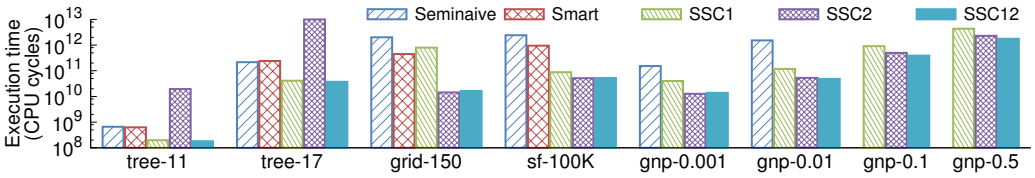

Fig. 7. Serial execution time of algorithms.

The SSC algorithms have a significant advantage over SEMINAIVE and SMART in terms of memory utilization — the memory utilization of SEMINAIVE and SMART is 1.2x-120x of SSC12, or 1.8x-1400x if SSC12 uses the storing-to-disk mode. In the remaining of this section, we compare these algorithms focusing on speed and scalability. We only show the result of an algorithm on a test graph if the algorithm is applicable to the test graph.

B. Tuning the Parameters of SSC12

Now we determine the values of α and β in SSC12. We reduce the search space by forcing both parameters to be powers of two. α is set to a pessimistic value: we find the α which is the smallest power of two such that for $k = n/\alpha$ random integers in the range of $[0, n-1]$, the time of initializing array d and setting the values corresponding to the k integers to **true** is smaller than that of inserting the k integers into δtc$''$. This value is pessimistic since it guarantees that the time becomes shorter if the algorithm chooses to switch. We select $\alpha = 1/8$ as it is the value found by the above procedure for $n = 10^6$.

We tune β by executing SSC12 on four synthetic test graphs, namely tree-17, grid-250, sf-100K and gnp-0.01. We select these graphs since each graph represents a medium workload such that the execution time is neither too short nor too long. The algorithm uses 16 threads on tree-17, 64 threads on the remaining three graphs as these values are optimal in Fig. 13. Fig. 6 shows the execution time for different β. The execution time on tree-17 decreases as β increases, while the execution on the remaining three graphs increases as β increases. We select $\beta = 1/128$ as a trade-off between these two cases.

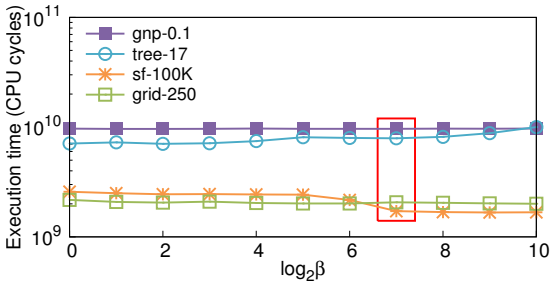

Fig. 6. Execution time of SSC12 for different β.

The meaning of this parameter setting is: SSC12 chooses to switch if the number of tuples in the SSC is greater than $n/128$ or the number of derivations in an iteration is greater than $n/8$. The setting derived from these four graphs proved effective at delivering good SSC12 performance throughout the wide range of test graphs tested in our experiments.

C. Serial Execution Performance

We first study the serial execution performance of the compared algorithms. Fig. 7 shows the serial execution time of each algorithm. The computation of SSC2 on tree-17 did not finish in one day. We use a value of 10^{13} in the figure to indicate this.

Effectiveness of SSC algorithms. Overall, the SSC algorithms achieve the shortest serial execution time on all the test graphs reported in Fig. 7: SSC2 is the fastest on grid-150, sf-100K and gnp-0.001, and SSC12 is the fastest on the remaining graphs. We say a source vertex is a type I (II) vertex if SSC1 is much faster (slower) than SSC2, otherwise it is a type III vertex. Since most vertices in trees (e.g., tree-11 and tree-17) are type I vertices, SSC2 performs poorly on trees, while SSC1 is much faster than SSC2 on trees. On the other hand, most vertices in grid-150 are type II vertices, thus SSC2 is much faster than SSC1. SSC2 is only slightly faster than SSC1 on the remaining graphs since most vertices are type III vertices. The hybrid algorithm, SSC12, is able to select the more efficient algorithm for every source vertex in the graph. The ability to make this intelligent selection enables SSC12 to outperform both SSC1 and SSC2 on many test graphs (e.g., tree-11 and gnp-0.01). It consistently performs well on all the test graphs.

Linear Recursion Algorithms. SEMINAIVE and the SSC algorithms are based on linear recursion. Fig. 7 shows that SEMINAIVE is always slower than SSC1 on all the test graphs, which is consistent with the analysis in Section II-C. This result shows the effectiveness of the partitioning by source vertex optimization employed by SSC1. Algorithm I.1 [7] and strategy TCr [8] share the same idea with SSC1. But both compute the tuples from a set of source vertices at the same time. They are expected to be slower than SSC1 for the same problem that SEMINAIVE suffers from.

Linear vs. Non-linear Recursion. Fig. 7 shows an empirical comparison between the non-linear recursion based SMART algorithm and the linear recursion based algorithms. SMART is always faster than some linear recursion based algorithms on the four test graphs that it is applicable, which exhibits the advantage of smaller number of iterations during the TC computation. However, the linear recursion based SSC12 algorithm outperforms SMART on all four graphs as it uses more efficient data structures.

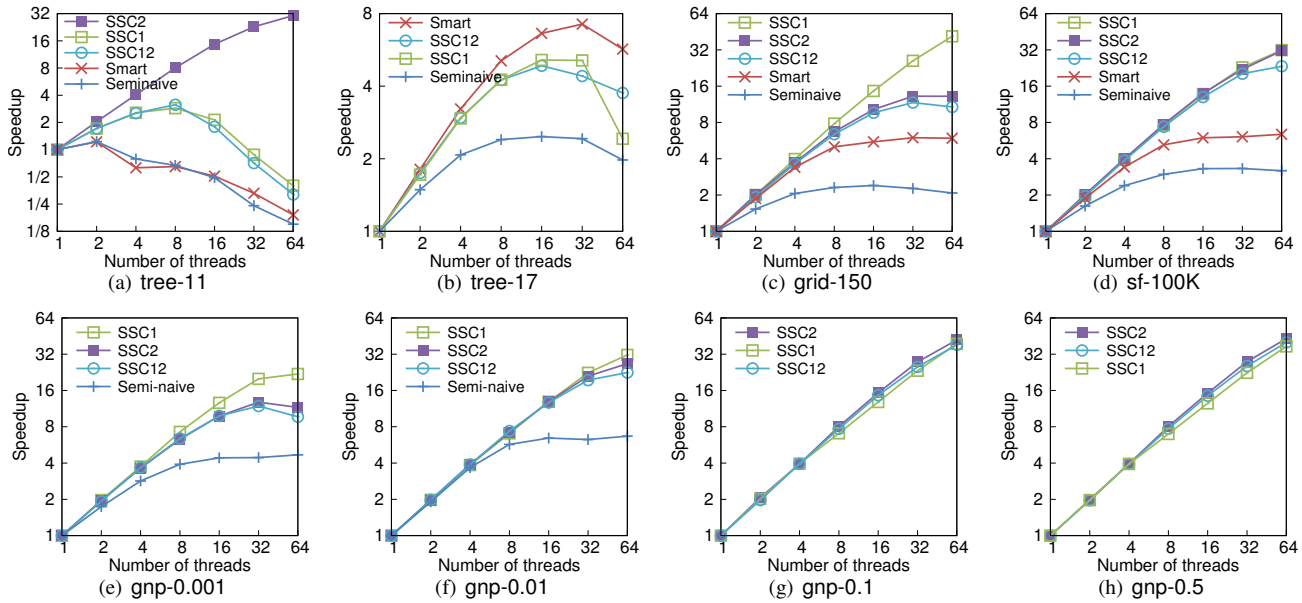

Fig. 8. Speedup of algorithms (w.r.t. the serial execution) using different number of threads.

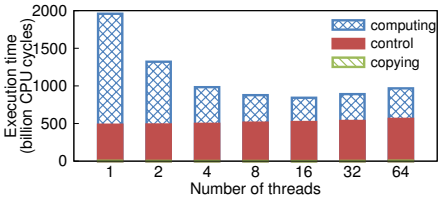

Fig. 9. Execution time breakdown of SEMINAIVE on grid-150.

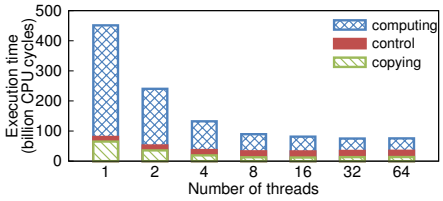

Fig. 10. Execution time breakdown of SMART on grid-150.

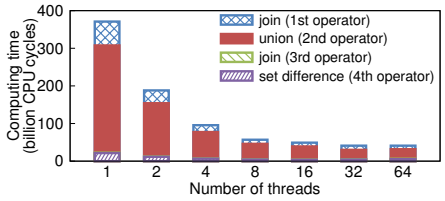

Fig. 11. Computing time breakdown of SMART on grid-150.

D. Parallel Execution Performance

Now we study the parallel execution performance of the compared algorithms on the test graphs.

1) Speedup of Algorithms: Fig. 8 shows the speedup of each algorithm w.r.t. the serial execution. Overall, every algorithm exhibits some speedup as the number of threads increases: SEMINAIVE and SMART show very limited speedup, while the SSC algorithms achieve almost linear speedup. A general rule is that the speedup does not increase linearly with the number of threads, but it becomes progressively less due to the increased overhead of synchronizing more threads. In the extreme cases where the time for synchronization dominates real computation time, we see the overall computation taking a longer time when we increase the number of threads — see, e.g., SEMINAIVE and SMART on tree-11.

We next discuss the speedups of the various algorithms in detail. We use the result of grid-150 (cf. Fig. 8(c)) as an example since the same trend is also observed in Fig. 8(d), 8(e) and 8(f).

Speedup of SEMINAIVE. The execution time of SEMINAIVE is the sum of computing time, control time and copying time (cf. Section III-B). Fig. 9 shows the execution time breakdown of SEMINAIVE on grid-150. The bars for copying time are unnoticeable since copying time accounts for less than 1% of the execution time. As the number of threads increases from 1 to 64, control time remains almost unchanged, while computing time decreases by about two thirds. (Note that computing time increases slightly when the number of threads increases from 16 to 64 as a result of increased overhead of synchronizing more threads.) Control time accounts for more than half of the execution time when 64 threads are used. Although there is speedup in computing time, the unchanged control time limits the overall speedup in SEMINAIVE. This result is very common for graphs that require many iterations while the computation in each iteration is very fast. However, for graphs like gnp-0.001 and gnp-0.01 that SEMINAIVE terminates in a few iterations (8 and 4, respectively) while the computation in each iteration takes a long time, the execution time is still dominated by computing time, and the speedup curve of SEMINAIVE is similar to that of SMART. As we will see next, the speedup of this kind of evaluation is bound by the memory bandwidth.

Speedup of SMART. Fig. 10 shows the execution time breakdown of SMART on grid-150. Computing time in SMART scales much better than that in SEMINAIVE, while control time accounts for a smaller percentage in SMART. Thus, SMART scales better than SEMINAIVE on grid-150. However, the speedup of computing time is only 8 when 64 threads are used. We further investigated the time spent on each relational algebra operators in computing time. There are four operators

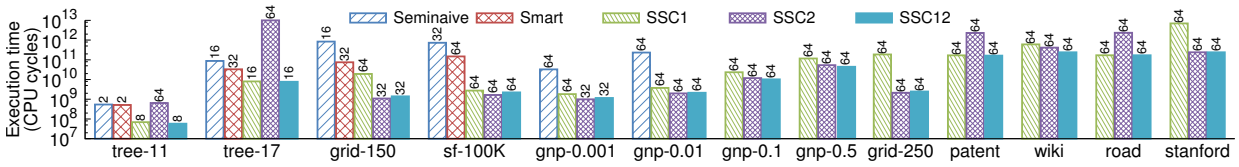

Fig. 13. Execution time for optimal number of threads.

in each iteration, namely join, union, join and set difference. Fig. 11 shows the total time spent on each operator. The time spent on the union operator dominates the computing time. A further breakdown of the time spent on the union operator reveals that the overall speedup is bound by the speedup of partition phase. A union operator works in two phases — partition phase and union phase. Both input relations are partitioned into smaller relations in the partition phase. Each thread computes the union of two partitioned relations in the union phase. Fig. 12 shows the maximal speedup of each phase in each iteration. The speedup of union phase is twice as much as that of partition phase from iteration 5 to 8, while the time spent on these 4 iterations accounts for 97% of the execution time on the union operator. The speedup of partition phase is limited by the memory bandwidth [20]. Thus, the overall speedup of SMART is bound by the memory bandwidth.

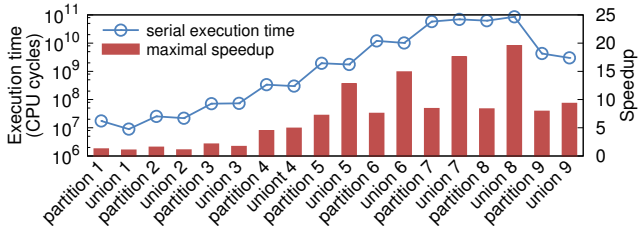

Fig. 12. Maximal speedup of partition phase and union phase in the union operator of SMART on grid-150. A label "partition i" ("union i") on the x-axis shows the maximal speedup of partition (union) phase in i-th iteration.

Speedup of SSC algorithms. An SSC algorithm is expected to achieve linear speedup since the computation in one thread does not interfere with the computation in other threads. However, its speedup stays below eight as the number of threads increases from 8 to 64 when the NUMA aware optimization described in Section III-C is not enabled[4]. The main reason for this behavior is due to the limited memory bandwidth. The NUMA region that the shared relation is resident on is the hotspot since all threads request data from it concurrently. When the memory is saturated (i.e., it cannot handle more memory load requests in a time unit), increasing the number of threads does not increase the speedup.

The memory saturation problem is alleviated by the NUMA aware optimization. This optimization 1) distributes the memory load requests to each NUMA region and 2) ensures each thread always accesses data on the local memory. The SSC algorithms exhibit linear speedup as shown in Fig. 8. But this optimization does not change the fact that the SSC algorithms are memory bandwidth bound. The evaluation of the all-pairs shortest path query described in Section 5 of [22] accesses more data than that of the TC query does. None of the SSC algorithms scale linearly on the graph that does not fit in the L3 cache as a result of memory saturation.

[4]We are referring to the case that the graph does not fit in the L3 cache, otherwise the algorithms scale better.

2) Minimal Execution Time: Finally, we compare all the algorithms in terms of their execution time using multiple threads. The execution time does not always decrease as the number of threads increases. The *optimal number of threads* is the number of threads such that the execution time of an algorithm on a graph is minimal among all other numbers. Fig. 13 shows the execution time of each algorithm using the optimal number of threads.

SMART is faster than SEMINAIVE on all four graphs that both algorithms are applicable as a result of better speedup, which is consistent with the conclusion of [1] that SMART has advantages over SEMINAIVE on TC computation. However, neither algorithms is the fastest on any test graph. The SSC algorithms achieve the minimal execution time on all the test graphs, while only SSC12 consistently performs well.

Fig. 13 also shows the minimal execution time of the SSC algorithms on grid-250 and four real-life graphs. These graphs have much larger TCs such that only the SSC algorithms are applicable. SSC1 is faster than SSC2 on patent and road since both graphs have tree structures. SSC2 is faster than SSC1 on the remaining three graphs. Nevertheless, SSC12 is the only algorithm that performs well on all five graphs.

Besides the speed, SSC12 is more memory efficient than SEMINAIVE and SMART. These two advantages make SSC12 an ideal choice for main memory parallel TC evaluation.

V. RELATED WORK

The TC of a binary relation is a much-studied recursive query. The earliest work dates back to 1962, when Warshall [10] proposed the Floyd-Warshall algorithm that computes the TC of an n-vertex graph in $\Theta(n^3)$ time. One line of research tries to speed up the computation by exploiting the special property of the problem itself. Warren and Henry [11] proposed a variant that works faster for sparse graphs in a paging environment. Agrawal and Jagadish [12] studied I/O efficient variants, the Blocked Warshall algorithm and the Blocked Warren algorithm, under the assumption that the memory size is small compared to the result relation size. The I/O cost is further reduced in algorithms based on depth-first search and a marking optimization [13], [14]. [16] and [17] compared I/O costs of TC algorithms using different implementations. Our study compares the serial execution performance of several TC algorithms. But we focus on main memory evaluation, and our implementations use cache conscious algorithms.

Our implementations of parallel TC algorithms are inspired by previous studies on parallel TC computation [6]–[9]. The idea of implementing SEMINAIVE and SMART using hash-based parallel relational algebra operators is attributed to Valduriez and Khoshafian [6]. Agrawal and Jagadish [7] and Wolfson et al. [8] proposed to partition the computation by the source vertices so that each core applies SEMINAIVE on a set of source vertices. The idea is similar to that of SSC1 except SSC1 applies SEMINAIVE on one source vertex one at the time. Cacace et al. [9] provided a survey on parallel TC algorithms. Previous studies use theoretical models to analyse the performance of algorithms, whereas our study focuses on experimental evaluation. In another experimental study [22] that includes the parallel Floyd algorithm [7], we showed that the Floyd algorithm achieves competitive performance for small dense graphs. But its memory requirement is impractical for large sparse graphs. Moreover, it is outperformed by SSC12 in the experiments.

VI. CONCLUSION AND FURTHER WORK

In this paper, we compared several recursive query evaluation algorithms on a modern multicore machine. A clear conclusion emerging from these experiments is that, for multicore machines, the simple SSC algorithms perform better than other algorithms in terms of speed and significantly better in terms of memory utilization. We thus introduced an algorithm, called SSC12, which combines the strengths of SSC1 and SSC2, and thus provides the obvious target algorithm for the compiler of our Datalog system DeAL [28] on multicore machines. However, our experiments also confirmed that performance of SSC12 (and other algorithms) on multicore machines will always be limited by the memory bandwidth bottleneck. Higher level of scalability through parallelism are however achievable on multi-node clusters. For multi-node clusters, the SMART algorithm has been shown to be optimal [1] — a conclusion that is confirmed by our recent investigation. The objective of this ongoing investigation is understanding the cost-performance tradeoffs on different multicore and multi-node systems for TC-like applications. From this study we seek to derive simple criteria for deciding which system, out of the many available on the cloud, can be most cost-effective for the application at hand. The ability of our Datalog compiler [28] to retarget recursive queries for different platforms is based on its ability to transform linear recursive rules into non-linear ones, which was described through simple examples in Section II. Many important algorithms that use TC-like rules in conjunction with monotonic aggregates [19] are amenable to such platform-driven porting and optimization.

ACKNOWLEDGMENT

This work was supported by NSF grants IIS 1218471 and IIS 1118107. We would like to thank the reviewers, Tyson Condie, Kai Zeng, Shi Gao and Alexander Shkapsky for their comments. We would like to thank the authors of [29], [30] for making their source code available online.

REFERENCES

[1] F. N. Afrati, V. Borkar, M. Carey *et al.*, "Map-reduce extensions and recursive queries," in *EDBT*. ACM, 2011, pp. 1–8.
[2] F. N. Afrati and J. D. Ullman, "Transitive closure and recursive datalog implemented on clusters," in *EDBT*. ACM, 2012, pp. 132–143.
[3] M. Shaw, P. Koutris, B. Howe *et al.*, "Optimizing large-scale semi-naive datalog evaluation in hadoop," in *Datalog in Academia and Industry*. Springer, 2012, pp. 165–176.
[4] Y. E. Ioannidis, "On the computation of the transitive closure of relational operators," in *VLDB*, vol. 86, 1986, pp. 403–411.
[5] P. Valduriez and H. Boral, "Evaluation of recursive queries using join indices," in *Expert Database Conf.*, 1986, pp. 271–293.
[6] P. Valduriez and S. Khoshfian, "Parallel evaluation of the transitive closure of a database relation," *International Journal of Parallel Programming*, vol. 17, no. 1, pp. 19–42, 1988.
[7] R. Agrawal and H. Jagadish, "Multiprocessor transitive closure algorithms," in *Proceedings of the first international symposium on Databases in parallel and distributed systems*. IEEE, 1988, pp. 56–66.
[8] O. Wolfson, W. Zhang, H. Butani *et al.*, "Parallel processing of graph reachability in databases," *International Journal of Parallel Programming*, vol. 21, no. 4, pp. 269–302, 1992.
[9] F. Cacace, S. Ceri, and M. Houtsma, "A survey of parallel execution strategies for transitive closure and logic programs," *Distributed and Parallel Databases*, vol. 1, no. 4, pp. 337–382, 1993.
[10] S. Warshall, "A theorem on boolean matrices," *JACM*, vol. 9, no. 1, pp. 11–12, 1962.
[11] H. S. Warren Jr, "A modification of warshall's algorithm for the transitive closure of binary relations," *CACM*, vol. 18, no. 4, pp. 218–220, 1975.
[12] R. Agrawal and H. Jagadish, "Direct algorithms for computing the transitive closure of database relations." in *VLDB*, vol. 87, 1987, pp. 1–4.
[13] Y. E. Ioannidis and R. Ramakrishnan, "Efficient transitive closure algorithms." in *VLDB*, vol. 88, 1988, pp. 382–394.
[14] R. Agrawal and H. Jagadish, "Hybrid transitive closure algorithms." in *VLDB*, 1990, pp. 326–334.
[15] H. Jakobsson, "Mixed-approach algorithms for transitive closure," in *PODS*. ACM, 1991, pp. 199–205.
[16] R. Kabler, Y. E. Ioannidis, and M. J. Carey, "Performance evaluation of algorithms for transitive closure," *Information Systems*, vol. 17, no. 5, pp. 415–441, 1992.
[17] S. Dar and R. Ramakrishnan, "A performance study of transitive closure algorithms," in *ACM SIGMOD Record*, vol. 23, no. 2. ACM, 1994, pp. 454–465.
[18] C. Zaniolo, S. Ceri, C. Faloutsos *et al.*, "Advanced database systems," 1997.
[19] A. Shkapsky, N. Lu, and C. Zaniolo, "Towards more powerful datalog systems: The implementation and optimization of recursive queries with monotonic aggregates in deals," UCLA, Tech. Rep. 140004, 2014.
[20] C. Kim, T. Kaldewey, V. W. Lee *et al.*, "Sort vs. hash revisited: fast join implementation on modern multi-core cpus," *PVLDB*, vol. 2, no. 2, pp. 1378–1389, 2009.
[21] S. Manegold, P. Boncz, and M. Kersten, "Optimizing main-memory join on modern hardware," *TKDE*, vol. 14, no. 4, pp. 709–730, 2002.
[22] M. Yang and C. Zaniolo, "Main memory evaluation of recursive queries on multicore machines," UCLA, Tech. Rep. 140014, 2014.
[23] "Graphstream," http://graphstream-project.org.
[24] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over time: densification laws, shrinking diameters and possible explanations," in *SIGKDD*. ACM, 2005, pp. 177–187.
[25] "Wikipedia," http://www.wikipedia.org.
[26] "Eastern usa road network," http://www.dis.uniroma1.it/challenge9/data/USA-road-d/USA-road-d.E.gr.gz.
[27] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.
[28] A. Shkapsky, K. Zeng, and C. Zaniolo, "Graph queries in a next-generation datalog system," *PVLDB*, vol. 6, no. 12, pp. 1258–1261, 2013.
[29] S. Blanas, Y. Li, and J. M. Patel, "Design and evaluation of main memory hash join algorithms for multi-core cpus," in *SIGMOD*. ACM, 2011, pp. 37–48.
[30] C. Balkesen, J. Teubner, G. Alonso *et al.*, "Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware," in *ICDE*, 2013, pp. 362–373.