A Parallel Bottom-Up Resolution Algorithm Using Cilk

Reza Basseda
Department of Computer Science
Stony Brook University
Stony Brook, NY, 11794, United States
Email: rbasseda@cs.stonybrook.edu

Rezaul Alam Chowdhury
Department of Computer Science
Stony Brook University
Stony Brook, NY, 11794, United States
Email: rezaul@cs.stonybrook.edu

***Abstract*—Rapid developments of multicore processors in the last ten years have accelerated the advancements in concurrency platforms. Performance of bottom-up resolution algorithms used in logic programming and artificial intelligent systems, can potentially be improved using the parallel programming constructs offered by these platforms (e.g., `OpenMP`, `Cilk++`, etc.). In this work we use `Cilk++` to implement a parallel bottom-up resolution algorithm, and study how different parallel programming constructs affect its performance. Our experimental results show that a careful `Cilk++` implementation of the algorithm can lead to significant speedup w.r.t. its traditional serial implementation.**

***Keywords*-Parallel Logic Programming; Parallel Resolution Algorithm; Bottom-up Reasoning**

I. Introduction

Concurrency platforms such as OpenMP [1] and Cilk++ [2], considerably simplify the implementation of parallel algorithms for modern multicore machines. These developments became essential as the technology for multicore processors started to advance rapidly in recent years. Availability of these multicore machines has lead to a revolution in the area of algorithm design, particularly when dealing with big data. On the other hand, sizes of data in different domains are also growing fast, and in many cases such growth is inevitable. Resolution algorithms in logic programming must also be prepared to face such revolutionary growths as now-a-days program sizes in terms of rules and identifiers are becoming noticeably huge. While trying to redesign and reimplement our algorithms to handle these large programs efficiently on multicores, we should choose a concurrency platform after careful consideration of the opportunities and ease of parallelization offered by various such platforms. We have decided to use `Cilk++` for our implementations as it allows easy parallelization based on only three keywords (i.e., `cilk_for`, `cilk_spawn` and `cilk_sync`), uses a provably good work-stealing scheduler for efficient load-balancing, has good support for *fork-join* paralleism which will be useful for our algorithms based on recursive divide-and-conquer, and has a library of hyperobjects (e.g., reducers) with provable performance guarantees [2].

Several research projects aim to develop and study parallel resolution algorithms. In [3], Ishida not only explains the semantics of a parallel logic programming language, but also shows that manipulations must take place at the algorithmic level in addition to the program level of rule based systems. In [4], Bellia and Occhiuto consider the complexity class of parallel unification algorithms. Parallelism based on message passing inter-process communication also have been studied in Datalog systems. In [5] and [6], the complexity of parallel processing of recursive Datalog queries has been studied. In [7], Valduriez et al. proposes two parallel algorithms for evaluating the transitive closure of a relation in a parallel data server. In [8], Wolfson proposes a method of parallelizing the evaluation of a set of Datalog programs. He syntactically characterizes several classes of Datalog programs that can be shared using this method. He uses a notion related to bottom up evaluation and program classification. In [9], Robinson and Lin provide an arithmetic approach for load balancing and communication in evaluation of linear recursive queries in distributed parallel Datalog programs. In [10], Ganguly et al. present several methods for the parallel, bottom-up evaluation of Datalog queries. They introduce the notion of a discriminating predicate, based on hash functions, that partitions the computation between the processors in order to achieve parallelism.

While most of the developments in parallel logic programming languages and Datalog use parallel processes with message passing inter-process communication, recent evolution of multicore processors makes parallelism under shared memory more feasible. However, shared-memory parallel programming comes with its own set of challenges. In addition to being able to efficiently handle race conditions arising from concurrent writes to the same memory location, fast shared-memory parallel algorithms must use the shared cache hierarchy (a.k.a. memory hierarchy) on a multicore machine efficiently. In this paper we explore how the `Cilk++` concurrency platform helps in meeting these challenges in the context of resolution algorithms. While the reducer hyperobjects provided by `Cilk++` helps in efficient handling of race conditions without locks, its randomized work-stealing scheduler supports cache-efficient scheduling of concurrent threads. We utilize these two resources along with the parallel programming constructs

supported by Cilk++ to implement a parallel resolution algorithm that scales well on multicores.

In this paper, we briefly explain a simple bottom-up parallel resolution algorithm that can be implemented in Cilk++. Our algorithm is similar to the SLD resolution algorithm as it tries to find a resolvent for each pair of knowledge base clauses in each resolution step. Cilk++ is designed for general-purpose parallel programming, but is especially effective for exploiting dynamic, highly asynchronous parallelism, which can be difficult to write in data-parallel or message-passing style [11]. Its run-time thread scheduler uses work-stealing which on P processing cores leads to an expected running time of of $T_1/P + O(T_\infty)$ for any fully strict computation with work T_1 and critical-path length T_∞, including scheduling overhead[1] [12]. As Cilk++ employs a provably efficient scheduling algorithm, it delivers guaranteed performance for our application. Implementation of our resolution algorithm in Cilk++ also shows how the parallel programming constructs provided by it can be used to have an efficient parallel resolution algorithm. In our implementation of the algorithm, we use parallel loop constructs in Cilk++ which not only make the implementation easy but also provide a very decent thread creation method for parallel loops. We also show that use of reducers in the body of parallel loops in our implementation have a great impact on the speedup of the algorithm.

We will explain our methodology and algorithm in the next section. Section 3 will describe our case study and experiments. We will briefly discuss our results in Section 4, and Section 5 will draw a conclusion for our study.

II. Methodology and Algorithm

Any bottom-up resolution algorithm can be considered a good candidate for parallelization. As the purpose of our study is to consider the role of parallel programming frameworks and implementation techniques on the performance of parallel resolution algorithms, we will simply study these algorithms in their simplest and basic form. Hence we consider a bottom-up parallel resolution algorithm for ground logic programs which can be expressed using propositional logic [13]. Although the considered resolution algorithm works only for logic programs without function symbols and variables, it still can be useful in different applications of knowledge base systems. This assumption makes the scope of the resolution algorithm clear. In this section, we first briefly describe the resolution algorithm followed by the implementation details in Cilk++.

A. Basic bottom-up resolution algorithm

Given a set of propositional logic CNF clauses as KB, the goal of the knowledge base system is to decide whether $KB \models \alpha$ for some propositional logical sentence α. Basically, for a bottom-up parallel inference procedure, the solution usually has two steps. In such systems, the set of all of the derivable logical sentences out of the current state of knowledge base will be built, then we will search to find the query clauses or their negation inside the expanded knowledge base. Although the search problem is a well studied problem in parallel algorithms, the interesting component of this solution for us is the knowledge base expansion part.

[1] T_p is the running time on p processing cores

```
1:  function PL_RESOLUTION(KB)        ▷ returns KB'
2:      clauses ← the set of clauses of KB
3:      new ← ∅
4:      for all C_i, C_j ∈ clauses do
5:          resolvents ← PL_RESOLVE(C_i, C_j)
6:          if resolvents contains an empty resolvent then
7:              exit
8:          end if
9:          if resolvents ≠ NULL then
10:             new ← new ∪ resolvents
11:         end if
12:     end for
13:     if new ⊆ clauses then
14:         return clauses
15:     end if
16:     return PL_RESOLUTION(new ∪ clauses)
17: end function
```

Figure 1: A simple sequential bottom-up resolution algorithm.

Figure 1 shows a common sequential resolution algorithm for the first component of our inference procedure. The input to this algorithm is a set of CNF clauses. The algorithm applies the resolution rule to that set of clauses recursively. Each pair containing complementary literals is resolved to produce a new set of clauses, called resolvents, which is added to the set. Then we call the resolution function recursively. This process continues until one of two things happens:

- There are no new clauses that can be added, in which case we get the complete set of clauses inferred from the original knowledge base. This will lead to successful termination of the algorithm.
- Two clauses resolve to yield the *empty* clause in which case the knowledge base has logical inconsistency.

Figure 2 shows the function resolving a pair of clauses. This function implements a resolution rule which is proved to be sound [14]. It is also easy to show that the sequential resolution implemented by $PL\_RESOLUTION(KB)$ is complete [13].

Apparently, the loop in line 4 of $PL\_RESOLUTION(KB)$ can be be parallelized which can speedup the algorithm as the number of clauses grows. It is also obvious that the loop in line 2 of $PL\_RESOLVE(C_i, C_j)$ also can be parallelized which may improve the speedup when we have large clauses.

```
1: function PL_RESOLVE(C_i, C_j)                ▷ returns C_k
2:     for all l_i ∈ C_i, l_j ∈ C_j do
3:         if l_i = ¬l_j then
4:             C'_i ← C_i \ {l_i}
5:             C'_j ← C_j \ {l_j}
6:             for all m_i ∈ C'_i, m_j ∈ C'_j do
7:                 if m_i = ¬m_j then
8:                     return NULL
9:                 end if
10:            end for
11:            return C'_i ∪ C'_j
12:        end if
13:    end for
14:    return NULL
15: end function
```

Figure 2: The resolving function for sequential bottom-up resolution algorithm.

```
1: function PAR_REC_RESOLUTION(KB) ▷ returns KB'
2:     clauses ← the set of clauses of KB
3:     S_1 ← the first half of clauses
4:     S_2 ← the second half of clauses
5:     new ← ∅
6:     spawn new_{1,1} ← SUBRESOLVE(S_1, S_1)
7:     spawn new_{1,2} ← SUBRESOLVE(S_1, S_2)
8:     spawn new_{2,1} ← SUBRESOLVE(S_2, S_1)
9:     new_{2,2} ← SUBRESOLVE(S_2, S_2)
10:    sync
11:    new ← new_{1,1} ∪ new_{1,2} ∪ new_{2,1} ∪ new_{2,2}
12:    if new ⊆ clauses then
13:        return clauses
14:    end if
15:    return PAR_REC_RESOLUTION(new ∪ clauses)
16: end function
```

Figure 3: A simple parallel bottom-up resolution algorithm based on recursive divide and conquer.

Large clauses are expected when we have large rules and a large number of logical terms. In this work we only study what happens when the number of clauses grows, and we do not consider large clauses, even though the parallelism approach we are describing can be applied in both cases.

B. Parallelism using parallel recursive calls

The divide-and-conquer technique employed in many sequential algorithms can often be parallelized easily. As the sub-problems are solved independently, their solutions can be computed in parallel.

Parallel function call is a common approach in parallel programming. Basically, this technique is suitable when you have a divide and conquer algorithm and all of the recursive calls are mutually independent from each other. In practice, this means that there should not be a memory location accessed by two or more concurrent function calls at least one of which writes to the location. In our case, we can design a divide-and-conquer algorithm equivalent to the algorithm shown in Figure 1, which applies the resolution rule on different parts of the set of clauses. This algorithm has been shown in Figure 3. The cornerstone of this structure is a recursively invoked function $SUBRESOLVE(S_i, S_j)$ shown in Figure 4. Inside $SUBRESOLVE(S_i, S_j)$, the problem is either split into four smaller subproblems according to the input set of clauses or it is directly solved as described before. Basically, recursion continues until the sizes of input set of clauses are too small to be solved in parallel and the overhead of creating parallel threads start to dominate. We compare the size of the larger of the two sets of clauses with a threshold named $grainsize$ which depends on various parameters, such as the cache size and the spawning overhead. In fact, this becomes important specifically as (1) the overhead of performing further splits and merges significantly degrades performance, or (2) the size of the problem is optimal for the target system due to system parameters (e.g., cache size).

We are now in a position to perform an asymptotic analysis of the algorithm. Let n be the number of clauses. If we assume that the resolvent sets shown in line 24 of $SUBRESOLVE(S_i, S_j)$ can be merged in $T_\infty(n) = \Theta\left(\log^2 n\right)$ and $T_1(n) = O(n)$ using the parallel merge algorithm given in [15], we will have $T_\infty(n) = \Theta(\log^2 n)$ and $T_1(n) = \Theta(n)$ for $SUBRESOLVE(S_i, S_j)$. Then the maximum speedup one can expect from function $SUBRESOLVE(S_i, S_j)$ due to parallelism will be $\Theta(\frac{n}{\log^2 n})$.

C. Parallelism using parallel loop constructs

Parallelization of loop iterations is another technique for parallelization of algorithms. We can use the parallel loop constructs as we need to perform the same independent resolution operation for each pair of clauses for a fixed number of iterations. The iterations of a loop will be independent if they do not write to memory locations that are accessed by other iterations. The syntax of a parallel loop is very similar to the `for` loops we already know, but the parallel loop iterations runs in parallel on a computer that has cores available. Another difference is that unlike a sequential loop, the order of execution isn't defined for a parallel loop. Iterations often execute at the same time, in parallel. Sometimes, the order in which two iterations execute can be the opposite of their sequential order. The only guarantee is that all of the loop's iterations will have run by the time the loop finishes.

Figure 5 shows a parallel bottom-up resolution algorithm using parallel loop constructs. Although the struc-

```
1: function SUBRESOLVE(S_i, S_j)                    ▷ returns S'
2:     size ← max(|S_i|, |S_j|)
3:     new ← ∅
4:     if size < grainsize then
5:         for all C_i ∈ S_i, C_j ∈ S_j do
6:             resolvents ← PL_RESOLVE(C_i, C_j)
7:             if resolvents contains an empty resolvent
   then
8:                 exit
9:             end if
10:            if resolvents ≠ NULL then
11:                new ← new ∪ resolvents
12:            end if
13:        end for
14:    else
15:        S_{1_1} ← the first half of S_1
16:        S_{1_2} ← the second half of S_1
17:        S_{2_1} ← the first half of S_2
18:        S_{2_2} ← the second half of S_2
19:        spawn new_{1,1} ← SUBRESOLVE(S_{1_1}, S_{2_1})
20:        spawn new_{1,2} ← SUBRESOLVE(S_{1_1}, S_{2_2})
21:        spawn new_{2,1} ← SUBRESOLVE(S_{1_2}, S_{2_1})
22:        new_{2,1} ← SUBRESOLVE(S_{1_2}, S_{2_1})
23:        sync
24:        new ← new_{1,1} ∪ new_{1,2} ∪ new_{2,1} ∪ new_{2,2}
25:    end if
26:    return new
27: end function
```

Figure 4: The recursive function in parallel bottom-up resolution algorithm based on recursive divide and conquer.

```
1: function PAR_LOOP_RESOLUTION(KB)                 ▷
   returns KB'
2:     clauses ← the set of clauses of KB
3:     new ← ∅
4:     for all C_i ∈ clauses do Parallel
5:         for all C_j ∈ clauses do Parallel
6:             resolvents ← PL_RESOLVE(C_i, C_j)
7:             if resolvents has an empty resolvent then
8:                 exit
9:             end if
10:            if resolvents ≠ NULL then
11:                new ← new ∪ resolvents
12:            end if
13:        end for
14:    end for
15:    if new ⊆ clauses then
16:        return clauses
17:    end if
18:    return PAR_LOOP_RESOLUTION(new ∪
   clauses)
19: end function
```

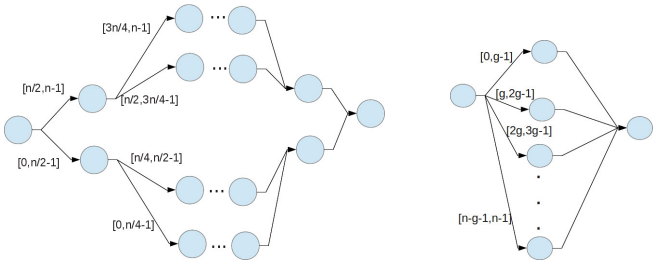
Figure 5: A simple parallel bottom-up resolution algorithm based on parallel loop pattern.

ture of this algorithm is very similar to the sequential bottom-up resolution algorithm shown in Figure 1, the execution model is very different. Observe that line 4 of $PL\_RESOLUTION(KB)$ is mapped to lines 4–5 of $PAR\_LOOP\_RESOLUTION(KB)$. But that is not the only difference between the two algorithms. In line 11 of $PAR\_LOOP\_RESOLUTION(KB)$, the parallel loop threads have a write access to a shared memory space which may give rise to race condition. The standard solution for this problem is guarding the shared memory data via mutual exclusion. Locks can easily provide the required mutual exclusion. However, such solution may reduce the speedup of the parallel algorithm drastically. `Cilk++` provides a family of flexible data structures called *reducers* which addresses the problem of having write access to a shared memory data in parallel code. Cilk's parallel loop construct provides an efficient runtime model of thread creation which affects the asymptotic analysis of the algorithm. We will briefly explain these two features, and then analyze the time complexity of the algorithm.

1) Parallel loop constructs in Cilk: Parallel loop construct's runtime thread creation model is not the same as spawning each loop iteration. In fact, the compiler converts the loop body to a function that is called recursively using a divide-and-conquer strategy. Figure 6a shows the runtime thread creation model in Cilk which provides significantly better performance as the overhead of thread creation decreases. Each node on the left side of the picture represents a thread creation call and each node on the right side of the graph shows a join of two threads. The label of each edge also shows the range of iterations carried by the corresponding thread. This process of creating threads continues until each thread gets at most a certain number of iterations which is called grain size. This run time execution model for parallel loop constructs decreases the time complexity of a loop of n iteration from $O(n)$ to $O(\log n)$.

On the other hand, we can call spawn to create a thread directly for each loop iteration. We call this non-recursive parallel loop construct. Figure 6b shows how threads are created in non-recursive parallel loop constructs. In this model, for each loop iteration (or for each set of at most g number of iterations), we directly call spawn. Theoretically[2], run time execution model for parallel loop constructs decreases the time complexity of a loop of n iteration from $O(n)$ to $O(\frac{n}{g})$. In practice, however, it may have an opposite result when the number of iterations gets large as the scheduler struggles with a large number of spawn calls which join all together. Our experiments show that using Cilk run time scheduler, this kind of parallelization of loops ruins the performance

[2]assuming all threads can be launched simultaneously



(a) Recursive. (b) Non-recursive.

Figure 6: Thread creation models of parallel loop constructs

of the parallel algorithm.

2) Reducers in Cilk: `Cilk++` provides a shared object construct, called a *reducer* that allows many strands to coordinate in updating a shared variable or data structure independently by providing different but coordinated views of the object to different threads at the same time. Reducers have a great impact on the performance of parallel loops in our algorithm. In all our implementations, we have used reducers for all memory objects shared by parallel loop threads.

Now let us compute the speedup of $PAR\_LOOP\_RESOLUTION(KB)$. Let's assume that we implement lines 4–5 of that function with parallel loops. Then, using recursive parallel loop constructs, the function will take $T_\infty(n) = \Theta(\log n)$ and $T_1(n) = O(n^2)$. The maximum expected speedup for the function will be $\Theta\left(\frac{n^2}{\log n}\right)$. The asymptotic analysis for the non-recursive parallel loop constructs is similar, and the function will take $T_\infty(n) = \Theta(\frac{n^2}{g^2})$ and $T_1(n) = O(n^2)$. Hence, the maximum expected speedup for the function will be $\Theta(g^2)$ which is equal to $\Theta(1)$ as g is a constant factor. Observe that we assumed the cost of thread creation for parallel loop construct to be zero. Because of such optimistic assumption, the experimental results show different behaviour.

III. Experiments and Results

We have implemented algorithms shown in Figures 3 and 5. For parallel resolution algorithm with parallel loop constructs, we have two implementations based on the two parallel loop constructs mentioned in previous section. Now in order to compare these algorithms and consider their performance and speedup in practice, we need to have an appropriate case study. We should be able to easily scale our case study to explore the effect of growing the size of the program. We should also be able to measure the size of the program easily. Based on these criteria we have chosen the *Wumpus World* [13] game as our case study. Wumpus World is a well-known toy example in artificial intelligence research. Basically, it is an agent navigating a grid with a few types of perceptions and trying to achieve a goal while avoiding a few types of enemies and obstacles.

The agent's simple knowledge base can be represented using propositional logic. Changing the size of the grid, the example can be scaled clearly, and we can easily measure the number of clauses and facts. Note that we have chosen our case study as simple as possible in order to be able to scale the input size easily. This will allow us to study how the size of the input affects the performance of the parallel resolution algorithm. We have tested the algorithms with different grid sizes. Note that the number of clauses has a linear relation to the number of positions in the grid.

In the above mentioned algorithms, we can keep all clauses resolved in the previous step in the memory in order to avoid the repeated application of the resolution rule on those clauses. This means that, in our case study, a large number of clauses will resolve with each other when we are initially adding clauses from input program. But the load of resolution will decrease as we memoize the previously resulted resolvents. This technique is similar to the tabling technique in logic programming frameworks. This technique can be used when our agent acts in a static environment and the perceptions and facts added in the previous steps of executions are not removed (or changed) from the knowledge base. The memoization technique cannot be used in dynamic environments in which the previously added facts to the knowledge base may not be true at the time of reasoning. As the set of facts in such environments should be refreshed frequently, we cannot use previously added clauses to the knowledge base. This increases the number of resolving steps. Since our analysis showed that the parallel iterative algorithm may lead to better speedup, we only focused on this algorithm in our experiments.

We run our experiments on one compute node of the Lonestar supercomputing cluster located at Texas Advanced Computing Center. The node has two processors, each a Xeon 5680 series 3.33GHz hex-core processor with a 12MB unified L3 cache. Peak performance for the 12 cores is 160 GFLOPS. We also used the `Cilk++` compiler compatible with gcc version 4.2.4.

As shown in Figure 7, memoization technique decreases the speedup of parallel resolution algorithm using `Cilk++` parallel loop constructs because it reduces the number of resolving clauses in each iteration. Basically, using memoization technique, we avoid resolving clauses we matched in previous steps. That decreases the size of sets in the recursive algorithm, and the number of iterations in the iterative algorithm. On the other hand, our experiments show that we can have a speedup factor close to the number of cores when we avoid the memoization technique. For example, Figure 8 shows the speedup trend of the algorithm shown in Figure 5 using parallel loop constructs. This trend shows that we can get benefits of efficient cilk parallel programming constructs when the memoization technique is not used. This trend also highlights the role of `Cilk++` parallel loop constructs as explained in previous section.

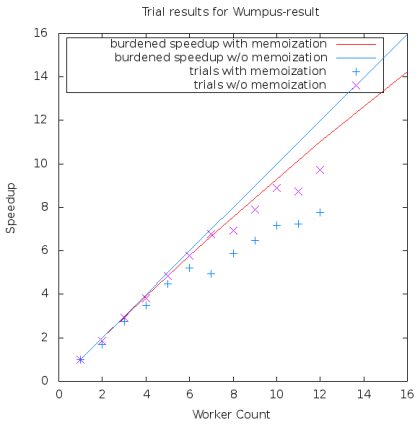


Figure 7: The effect of memoization on the speedup of parallel algorithms using `Cilk++` loop constructs.

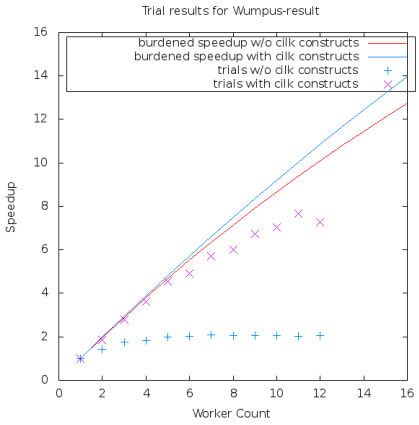


Figure 8: Trend of speedup in different implementations.

IV. Conclusion

Resolution algorithms play an important role in logic programming and knowledge base systems. The efficiency of those algorithms directly affects the performance of such systems. This is particularly true when a small set of updates in a knowledge base leads to a huge number of resolution steps in order to update resulting set of facts or check some consistency constraints. In this paper, we briefly studied the practical aspects of parallel resolution algorithms. In our experiments, we showed how implementation details in parallel programming may affect the performance of the parallel resolution algorithms. Our results also show that our parallel resolution algorithms without memoization achieves a speedup factor close to the number of cores.

The practical details of parallel programming should be discussed in other parallel logic programming areas as well. For example, parallel algorithms for unifications or updating indices also need to be studied in the presence of recent developments in parallel programming frameworks.

Acknowledgement

This research was supported by XSEDE grant number CCR120012. We would like to thank Michael Kifer and IV Ramakrishnan.

References

[1] "OpenMP application program interface, version 4.0," OpenMP specification, July 2013. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

[2] C. E. Leiserson, "The Cilk++ concurrency platform," *Journal of Supercomputing*, vol. 51, no. 3, pp. 244–257, March 2010.

[3] T. Ishida, "Methods and effectiveness of parallel rule firing," in *Artificial Intelligence Applications, 1990., Sixth Conference on*, 1990, pp. 116–122 vol.1.

[4] M. Bellia and M. E. Occhiuto, "N-axioms parallel unification," *Fundam. Inf.*, vol. 55, no. 2, pp. 115–128, Aug. 2002.

[5] F. Afrati and C. Papadimitriou, "The parallel complexity of simple chain queries," in *Proceedings of the sixth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, ser. PODS '87. New York, NY, USA: ACM, 1987, pp. 210–213.

[6] F. Afrati and C. H. Papadimitriou, "The parallel complexity of simple logic programs," *J. ACM*, vol. 40, no. 4, pp. 891–916, Sept. 1993.

[7] 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.

[8] O. Wolfson, "Parallel evaluation of datalog programs by load sharing," *The Journal of Logic Programming*, vol. 12, no. 4, pp. 369 – 393, 1992.

[9] J. Robinson and S. Lin, "Arithmetic for parallel linear recursive query evaluation in deductive databases," in *PARLE '93 Parallel Architectures and Languages Europe*, ser. Lecture Notes in Computer Science, A. Bode, M. Reeve, and G. Wolf, Eds. Springer Berlin Heidelberg, 1993, vol. 694, pp. 648–659.

[10] S. Ganguly, A. Silberschatz, and S. Tsur, "Parallel bottom-up processing of datalog queries," *The Journal of Logic Programming*, vol. 14, no. 12, pp. 101 – 126, 1992.

[11] *Cilk 5.4.6 Reference Manual*, Supercomputing Technologies Group, Massachusetts Institute of Technology Laboratory for Computer Science, Nov. 2001.

[12] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM*, vol. 46, no. 5, pp. 720–748, Sept. 1999.

[13] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.

[14] J. W. Lloyd, *Foundations of logic programming; (2nd extended ed.)*. New York, NY, USA: Springer-Verlag New York, Inc., 1987.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.