Beyond worst-case analysis
Observed low depth for a P-complete problem

Uzi Vishkin
vishkin@umd.edu
University of Maryland
College Park, Maryland, USA

Abstract
The performance of a simple parallel algorithm for 3CNF Horn-SAT is observed. The algorithm requires linear work. The algorithm also exhibits low parallel time ("depth") for central Horn-SAT formulae benchmarks. The work optimality of the algorithm, its observed low depth, P-completeness of the problem and algorithm-specific modeling put together demonstrate a way for going beyond worst-case for parallel algorithms. The questioning of the near exclusivity of the worst-case analysis mode in typical algorithms courses has received considerable attention recently for a broad range of algorithms. The current paper suggests a new line, unique to parallel algorithms, for questioning the worst-case analysis mode, dramatically underlining the broad perception that a P-completeness result is nothing short of "a nightmare for parallel processing", as phrased in an authoritative textbook.

CCS Concepts: • General and reference → Empirical studies; • Theory of computation → Timed and hybrid models; Theory and algorithms for application domains.

Keywords: Parallel algorithms, P-Completeness, Beyond worst-case

ACM Reference Format:

1 Introduction
It has been widely understood that the observed performances of some important algorithms for clustering, linear programming, neural network training and SAT solvers and many more applications have gone unexplained by analyses limited to worst-case. Still, as the recent edited book [6] points out, typical courses on the design and analysis of algorithms view worst-case analysis of algorithms as a silver bullet. Motivated by this apparent tension between specialized computer science knowledge and practice, on one hand, and such courses, on the other hand, this edited book sought to popularize alternative algorithms and analyses. Its 30 chapters provide representative examples from broad areas of algorithm research. However, none follows from a beyond worst-case (henceforth, BWC) behavior of a parallel algorithm point of view. (While deep learning relies on a parallel matrix multiplication routine through back propagation and stochastic gradient descent, the routine itself is within the worst-case domain.) This is particularly noteworthy since once processor clock rates started to plateau circa 2004, parallelism has been the dominant mode for performance improvements. One possible explanation is the lack of commodity general-purpose manycore CPUs, since the observation of runtime performance is often tied to standard computer systems, as the SAT solvers example demonstrates. While the SAT problem is NP-Complete, the improved performance of SAT solvers was observed with respect to standard CPU-based systems. This enabled demonstrating that SAT solvers often outperform their worst-case exponential complexity bounds.

The current paper provides an empirical evidence example for a parallel algorithm whose observed performance improves on what worst-case analysis teaches. Luckily, the specific algorithm at hand allows modeling the depth of the parallel algorithm around a certain algorithm-specific parameter. Monitoring the quantitative behavior of the parameter with respect to representative inputs derived from the main benchmarks for the problem allows observing parallel performance, thereby circumventing the lack of standard general-purpose manycore systems.

Developing evidence of strong performance (especially when the theory parallel algorithms suggests otherwise) has clear merit. For example, it could help influence vendors to finally pursue general-purpose manycore systems. Therefore, we hope that this paper will stimulate others to develop more such evidence, perhaps along similar lines.

The focus of this short paper is a linear work parallel algorithm for 3CNF Horn-SAT. The algorithm achieves linear speedup over a known breadth-first-search-like linear time serial algorithm [1] up to some number of p processors on
a PRAM. While the parallelism technique is not new, the algorithm has special merit since:

1. The problem is P-complete (as noted in [1]). This makes it unlikely that there will be an NC algorithm for it, namely an algorithm that runs in poly-log time using a polynomial number of processors. An NC-algorithm for a P-complete problem would immediately imply an NC algorithm for any problem in the class P. In other words, any problem that can be solved using a (serial) polynomial time algorithm, which is considered unlikely by complexity theorists. So, linear speedup not reaching poly-log time is the best one can hope for in the worst case. The exact upper bound is $O\left(\frac{N}{p} + h\log N\right)$ for an input of $N$ literals on a $p$-processor Arbitrary CRCW PRAM. Every iteration of the parallel algorithm seeks to concurrently commit the truth value of as many variables as possible, based on truth values committed in prior iterations. The parameter $h$ is the length of the longest chain of such commit iterations, given the input formula (by limited analogy to BFS). This means that the best parallel time (known also as “depth”) that the parallel algorithm can provide would be $O(h \log N)$, and it will need $N/(h \log N)$ processors to achieve that. None of the above is surprising.

2. Quantitative empirical studies of $h$ for all Horn-SAT benchmarks suggested to us by three expert SAT-solver researchers demonstrate low values, which imply overall low depth, and suggest very moderate growth with the input size, arguably, in contradiction to how the theory of P-completeness has been presented and interpreted.

Satisfiability problems have received extensive attention in the literature. Simon Kasif has published quite a few interesting papers on parallel complexity of various satisfiability instances, such as [3]. However, none is directly related to the current paper. Still: (i) the simplicity of the presented algorithm, (ii) its linear work, and (iii) mostly, its low empirical depth, relative to parallel NC algorithms for satisfiability instance are worth noting.

2 Basics

Input for 3CNF Horn-SAT. In each clause: up to 3 literals and up to one of them is positive. Let $N$ be the total number of literals.

Comment: CNF Horn SAT, which allows an unlimited number of literals per clause up to one of them positive, has a simple reduction to 3CNF Horn-SAT, as demonstrated next. Consider the clause $(x_1 \lor \neg x_2 \lor \neg x_3 \lor \neg x_4)$. Replace it by the following two 3CNF Horn-SAT clauses $(x_1 \lor \neg x_2 \lor \neg x_5)$ and $(x_5 \lor \neg x_3 \lor \neg x_4)$, where $x_5$ is a new variable.

Input form. Array of size $k$ for variables $P_1, \ldots, P_k$. Array of size $m$ for clauses $C_1, \ldots, C_m$. Each clause has a link to a subarray of size at most 3 comprising its literals. Each variable has a link to a subarray of its literals. Each clause vertex has a pointer to the beginning of its literals subarray. Each edge in the bipartite graph appears twice: once for each of its vertices. Finally, each edge has a pointer to its other copy. Note: only an arbitrary subset of these pointers is shown.

Figure 1. Left side: The bipartite graph for the 3CNF formula example. Middle column: the vertex array. Variable vertices followed by the clause vertices. Right column: The edge array. Each variable vertex has a pointer to the beginning of the subarray of its literals. Each clause vertex has a pointer to the beginning of its literals subarray. Each edge in the bipartite graph appears twice: once for each of its vertices. Finally, each edge has a pointer to its other copy. Note: only an arbitrary subset of these pointers is shown.

3 The Algorithm

Similar to [1], we assume that Horn formulae are in a “reduced” form, i.e., that there are no duplicate clauses and no duplicate literals within clauses.

The loop of the parallel algorithm (parallel unit propagation):

While there are clauses with singleton literals (“unit clauses”):

satisfy all of these literals, do {

In parallel, “Remove”: 

(i) all satisfied clauses, and 
(ii) all implied FALSE literals in every clause in which they appear. For every affected clause, perform the following case analysis:

a. No literals remain. Conclude: the formula is unsatisfiable. Terminate the algorithm.
b. Only one literal remained. Include this literal in the set of singleton literals for the next iteration.

Optional step. If the current set of singleton literals does not include any positive literal, conclude: the formula is satisfiable. Assign FALSE to all remaining variables to derive a satisfying assignment. Terminate the algorithm.

Final step. If the above did not already conclude that the formula is satisfiable or unsatisfiable, conclude: The formula is satisfiable. Assign FALSE to all remaining variables to derive a satisfying assignment.

The default assumption in this presentation is that the optional step is not used.

**Correctness.** When there are no unit clauses upon reaching the final step, each remaining clause must contain at least one negative literal. Therefore, FALSE assignment to all their variables will satisfy all of these literals and their clauses.

**Elaborating on the complexity upper bound.** As commonly understood in the parallel algorithms field, the $O(N/p + h \log N)$ time bound means $O(N/p)$ time for $p \leq \frac{N}{h \log N}$ processors. However, for the bound to have merit, $h \log N$ must not exceed $O(N)$. As explained next, the possibility that this would not be the case is a moot point:

1. Per the algorithm $h \leq v$, where $v$ is the total number of variables in the 3CNF formula. Namely, in the worst-case $h = O(v)$.
2. $v \log N$ would need to exceed $O(N)$.
3. The experimental results reported later in this paper exhibited low values for $h$ for random Horn-SAT formulae as well as for other benchmarks.

So, while it is not impossible that $h \log N$ would exceed $O(N)$, several relatively extreme conditions need to be met for this to happen. Finally, a parallel algorithms folklore trick ensures that our parallel algorithm will never require more than $O(N)$ time: Just dove-tail it with the serial $O(N)$ time algorithm.

### 3.1 Implementation comments

We assume the Arbitrary CRCW PRAM, in which $p$ synchronous processors can read simultaneously any shared memory location, as well as write simultaneously. In the case of several simultaneous writes into the same memory locations an arbitrary one succeeds.

The computation below applies balanced-binary-tree prefix-sum computations. For each such balanced binary data structure we assume without loss of generality that the number of elements (leaves in the tree) is always a power of 2, as it is always possible to round the number of leaves to the next power of 2 (achieving the same complexity).

In $O(m)$ work and $O(1)$ time, find all clauses which are unit clauses. For each unit clause assign a satisfying (TRUE or FALSE truth value) to its variable. This may involve concurrent writes.

Apply prefix-sum over the $k$ variables to assign serial numbers to all the just assigned variables.

Now, do a prefix sum computation over the degrees of these assigned variables to allocate a serial number to every one of their literals (within their respective clauses). This takes $O(s)$ work and $O(\log s)$ time for $s$ assigned variables. There are 4 cases once the literal of a clause gets its final truth value from its variable:

**Case 1:** The clause is satisfied. In this case, remove the clause.

When the literal is FALSE there are 3 cases:

**Case 2:** Two unassigned literals remain in the clause. Then do nothing.

**Case 3:** One unassigned literal remains. Namely, a new unit clause was found. Then the variable of that literal will be marked, possibly involving concurrent writes. Using parallel prefix sum computation, the variable will be given a serial number and included in the next iteration.

**Case 4:** No unassigned literals remain. Then the whole formula is declared unsatisfiable.

To complete the loop, each occurrence of **Case 3** above will induce a “thread” of activity. Inductively, the next iteration is now ready to begin. In total, the algorithm requires $O(N)$ work and $O(h \log N)$ time, where $h$ is the length of the longest chain of commit iterations in the input formula.

### 3.2 Experimental results

Ananth Hari performed the experimental studies reported below.

#### 3.2.1 Random Horn-SAT formulae

The complexity analysis of the algorithms suggests the question: what values for $h$ can we expect?

Personal communication with three constraint satisfaction experts suggested that the model that [5] discussed for random Horn-SAT formulae is probably the most relevant in the literature: For two real number $d_1 < 1$ and $d_2$, let the random Horn-SAT formula $H(N, d_1, d_2)$ be the conjunction of:

- a single negative literal $x_1$;
- $d_1 N$ positive literals chosen uniformly without replacement from the variables $x_2, \ldots, x_N$; and
- $d_2 N$ Horn clauses chosen uniformly with replacement from the $\binom{N(N-1)(N-2)}{2}$ possible Horn clauses with 3 variables where one literal is positive.
The experimental study reported below was done within the above model. The experiments considered three values for \(N\): 10,000, 100,000, and 1,000,000. \(d_1\) ranged between 0.02 and 0.2 and \(d_3\) between 1 and 5.

Figure 2a shows that the average value of \(h\) exceeded 60 only for one category: \(d_3 = 2\) and \(d_1 = 0.18\). Per Figure 2b, the largest value observed was slightly above 200 for the same category. Note that, for example, for \(N = 1,000,000\) the value of \(h\) would be lower by nearly 4 order of magnitude or more, leading to a low parallel time. This is noteworthy for our P-complete problem.

Comments:

1. It would be interesting to consider the practical value of the optional step in the context of a specific implementation platform. On one hand, the optional step would eliminate the need for any more iterations if the condition that there are no singleton positive literals holds true at the end of an iteration. On the other hand, the optional step requires extra work. Namely, checking whether the condition holds true.

2. The experimental results reported above are generally in line with those of [5]. In particular, [5] reported an inflection point for satisfiability of Horn-SAT formulae. Given a fixed value for \(d_1\):
   a. for lower values of \(d_3\) more formulae tend to be satisfiable, while
   b. for higher \(d_3\) values more formulae tend to be unsatisfiable.

   The parallel algorithm terminates (resulting in low value of \(h\)) once:
   a. it iteratively runs out of singleton clauses (concluding that the formula is satisfiable), or
   b. reaches an emptied clause (concluding that the formula is unsatisfiable).

   It appears that around inflection points some straddling between these two possibilities leads to higher \(h\) values explaining the peaks that for different values of \(d_1\), occur at different places between \(d_3 = 1\) and \(d_3 = 5\).

3.2.2 Horn-SAT formulae derived from an application. The second benchmark we considered was based on an algorithmic approach presented in the paper [8]. As explained there, temporal synthesis is the automated design of a system that interacts with an environment, using the declarative specification of the system’s behavior. A popular language for providing such a specification is Linear Temporal Logic, or LTL. LTL synthesis has been a hard problem to solve in practice in the general case. So that paper, like many others, focused on developing synthesis procedures for specific fragments of LTL, with an easier synthesis problem. [8] focuses on “Safety LTL.” Seeking to model a safety property, what makes that paper relevant to the current one is that the approach reduces fragments of LTL to Horn-SAT formulae, giving rise to a wealth of Horn-SAT instances derived from using Horn-SAT as a tool, thereby complementing our first benchmark. Recall, that the first benchmark was Horn-SAT instances derived directly from a randomized Horn-SAT model rather than from an application.

The above results are very encouraging, as \(h\), the parameter dominating the parallel time (or depth) of our algorithm does not exceed 18 in spite of considering pretty large Horn-SAT formulae.

**More about the experimental work.** The paper [8] set a timeout of 60 seconds and considered all benchmarks that exceeded the time limit as unsolved. They were able to solve only 159 cases out of the 395 cases in the benchmark using the Horn-SAT approach, explaining it as follows: “The performance of Horn-SAT decreases sharply…since formula generation dominates the synthesis time.” Ananth Hari set a timeout of 15 minutes just to generate Horn formulae and was able to generate 248 Horn formulae (from the same 395 cases). The remaining cases timed out. Table 1 summarizes the results for these 248 formulae.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Min</th>
<th>Max</th>
<th>Mean</th>
<th>Standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(k)</td>
<td>4</td>
<td>134e6</td>
<td>7.6e6</td>
<td>26.4e6</td>
</tr>
<tr>
<td>(f_1)</td>
<td>0.21</td>
<td>0.997</td>
<td>0.067</td>
<td>0.24</td>
</tr>
<tr>
<td>(f_2)</td>
<td>0.005</td>
<td>0.96</td>
<td>0.38</td>
<td>0.27</td>
</tr>
<tr>
<td>(h)</td>
<td>1</td>
<td>18</td>
<td>5.52</td>
<td>4.83</td>
</tr>
</tbody>
</table>

**Table 1.** \(k\) is the number of variables in the formulae. \(f_1\) is the number of positive unit clauses divided by \(k\). \(f_2\) is the number of all remaining clauses divided by \(k\). \(h\) is the parameter we have been studying throughout this paper. Note: \(Xe6\) implies \(X \times 10^6\).

4 What’s Next?

In response to requests from some of the reviewers the following text provides some general thoughts on how one may go about BWC parallel algorithms for other problems.

The first issue is where to look for parallel algorithms BWC. The current paper implies starting with problems for which there is a proof that desired worst-case solutions are unlikely to emerge, such as the problem of Horn-SAT, whose P-completeness implies that an NC algorithm is unlikely. One way to broaden this would be for problems whose worst-case algorithms have not provided good enough empirical solutions.

However, empirical runtime evaluation of a BWC parallel algorithm is more challenging than it should be. The introduction section suggested that developing evidence of strong performance of a parallel algorithm for a significant application could drive vendors to pursue general-purpose manycore systems. However, the other side of the same coin is that parallel industry grade platforms are currently lacking. So, how to evaluate runtime on a platform, which is
not yet available? A key insight of the current paper is to isolate the parameter $h$ from the parallel algorithm and center its parallel runtime evaluation around it. This insight circumvents the current “chicken and egg” impasse of the field. Industry-grade systems are either too difficult to program or provide too few cores to enable demonstrating how effective manycore systems, which are easier to program, could be. Unfortunately, it is not always possible to similarly reduce reasoning about runtime to a single parameter for demonstrate efficacy. So, while we hope that this paper will stimulate others to develop more evidence regarding appealing performance of parallel algorithms, it important to understand this challenge.

Finally, [4, Chapter 9] cited a P-completeness result as “a nightmare for parallel processing”. In contrast, this paper demonstrates that in and by itself P-completeness of a problem does not preclude a pragmatic, and even simple, parallel algorithm for it. This new knowledge is of general interest, as it dramatically qualifies the pragmatic impact of (the so-called) NC theory.

Acknowledgments
Moshe Vardi suggested the problem considered in this paper after seeing [2]. Moshe Vardi, Rina Dechter and Simon Kasif provided many helpful comments and suggestions on an earlier draft of this paper, as well as suggesting the benchmarks studied in the paper. Ananth Hari performed all the experimental studies above. All this help is gratefully acknowledged.

References