# Find Most Probable Worlds of Probabilistic Logic Programs: a Parallel Approach

Amy Sliva
Department of Computer Science
University of Maryland College Park
College Park, MD 20783, USA
asliva@cs.umd.edu

## 1 Introduction

Probabilistic logic programs (PLPs) [NS92] have been proposed as a paradigm for probabilistic logical reasoning with no independence assumptions. PLPs used a possible worlds model based on prior work by [Hai84], [FHM90], and [Nil86] to induce a set of probability distributions on a space of possible worlds. Past work on PLPs [NS91, NS92] focuses on the entailment problem of checking if a PLP entails that the probability of a given formula lies in a given probability interval.

However, we have recently been developing several applications for cultural adversarial reasoning [SAM+07] where PLPs and their variants are used to build a model of the behavior of certain socio-cultural-economic groups in different parts of the world.[1] Such PLPs contain rules that state things like "There is a 50 to 70% probability that group $g$ will take action(s) $a$ when condition $C$ holds." In such applications, the problem of interest is that of finding the most probable action (or sets of actions) that the group being modeled might do. This corresponds precisely to the problem of finding a "most probable world" that is the focus of this paper.

In Section 2 of this paper, we recall the syntax and semantics of such programs [NS91, NS92]. We state the *most probable world* (MPW) problem by immediately using the linear programming methods of [NS91, NS92] - these methods are exponential because the linear programs are exponential in the number of ground atoms in the language. Then, in Section **??**, we discuss several algorithms for solving the MPW problem, including several exact solutions, as well as a heuristic method for obtaining approximate solutions.

While the algorithms given in [KMN+07] are able to reduce the computation time necessary to solve for the MPW and can be applied to problems with (insert number) atoms, we can achive even better results by utilizing the resources provided by a computing cluster. Section 4 then, presents several explicitly parallel algorithms for computing the most probable world.

Section 5 describes a prototype implementation of the parallelized ap-program framework and includes a set of experiments to assess several of the algorithms. We assess both the efficiency of our algorithms, as well as the accuracy of the solutions they produce.

[1] Our group has thus far built models of the Afridi tribe in Pakistan, Hezbollah in the Middle East, and the various stakeholders in the Afghan drug economy.

## 2 Action Probabilistic Logic Programs

Action probabilistic logic programs (ap-programs) are an immediate and obvious variant of the probabilistic logic programs introduced in [NS91, NS92]. We assume the existence of a logical alphabet that consists of a finite set $\mathcal{L}_{cons}$ of constant symbols, a finite set $\mathcal{L}_{pred}$ of predicate symbols (each with an associated arity) and an infinite set $\mathcal{V}$ of variable symbols. Function symbols are not allowed in our language. Terms and atoms are defined in the usual way [Llo87]. We assume that a subset $\mathcal{L}_{act}$ of $\mathcal{L}_{Pred}$ are designated as *action symbols* - these are symbols that denote some action. Thus, an atom $p(t_1, \ldots, t_n)$, where $p \in \mathcal{L}_{act}$, is an *action atom*. Every (resp. action) atom is an (resp. action) wff. If $F, G$ are (resp. action) wffs, then $(F \wedge G), (F \vee G)$ and $\neg G$ are all wffs (resp. action wffs).

**Definition 2.1** *If $F$ is a wff (resp. action wff) and $\mu = [\alpha, \beta] \subseteq [0, 1]$, then $F : \mu$ is called a p-annotated (resp. ap-annotated—short for "action probabilistic" annotated) wff. $\mu$ is called the p-annotation (resp. ap-annotation) of $F$.*

Without loss of generality, we assume that $F$ is in conjunctive normal form (i.e. it is written as a conjunction of disjunctions).

**Definition 2.2 (ap-rules)** *If $F$ is an action formula, $A_1, A_2, ..., A_m$ are action atoms, $B_1, \ldots, B_n$ are non-action atoms, and $\mu, \mu_1, ..., \mu_m$ are ap-annotations, then $F : \mu \leftarrow A_1 : \mu_1 \wedge A_2 : \mu_2 \wedge ... \wedge A_m : \mu_m \wedge B_1 \wedge \ldots B_m$ is called an ap-*rule*. If this rule is named $c$, then $Head(c)$ denotes $F : \mu$; $Body^{act}(c)$ denotes $A_1 : \mu_1 \wedge A_2 : \mu_2 \wedge ... \wedge A_m : \mu_m$ and $Body^{state}(c)$ denotes $B_1 \wedge \ldots B_n$.*

Intuitively, the above ap-rule says that an unnamed entity (e.g. a group $g$, a person $p$ etc.) *will take action $F$ with probability in the range $\mu$ if $B_1, \ldots, B_n$ are true in the current state (we will define this term shortly) and if the unnamed entity will take each action $A_i$ with a probability in the interval $\mu_i$ for $1 \leq i \leq n$.*

**Definition 2.3 (ap-program)** *An* action probabilistic logic program *(ap-program for short) is a finite set of ap-rules.*

**Definition 2.4 (world/state)** *A* world *is any set of ground action atoms. A* state *is any finite set of ground non-action atoms.*

Note that both worlds and states are just ordinary Herbrand interpretations. As such, it is clear what it means for a state to satisfy $Body^{state}$.

**Definition 2.5** *Let $\Pi$ be an ap-program and $s$ a state. The* reduction of $\Pi$ w.r.t. *$s$, denoted by $\Pi_s$ is $\{F : \mu \leftarrow Body^{act} \mid s$ satisfies $Body^{state}$ and $F : \mu \leftarrow Body^{act} \wedge Body^{state}$ is a ground instance of a rule in $\Pi\}$.*

***Note that $\Pi_s$ never has any non-action atoms in it.***

A fixpoint operator, $T_{\Pi_s}$, is associated with an *ap*-program $\Pi$ and a state $s$ and maps sets of ground ap-annotated wffs to sets of ground ap-annotated wffs as follows.

**Definition 2.6** *Suppose $X$ is a set of ground action atoms. We first define an intermediate operator $U_{\Pi_s}(X)$ as follows. $U_{\Pi_s}(X) = \{F : \mu \mid F : \mu \leftarrow A_1 : \mu_1 \wedge \cdots \wedge A_m : \mu_m$ is a ground instance of a rule in $\Pi_s$ and for all $1 \leq j \leq m$, there is an $A_j : \eta_j \in X$ such that $\eta_j \subseteq \mu_j\}$.*

Intuitively, $U_{\Pi_s}(X)$ contains the heads of all rules in $\Pi_s$ whose bodies are deemed to be "true" if the action wffs in $X$ are true.

In order to assign a probability interval to each ground action atom,the same procedure followed in [NS91] is used. We use $U_{\Pi_s}(X)$ to set up a linear program $CONS_U(\Pi, s, X)$ as follows. For each world $w_i$, let $p_i$ be a variable denoting the probability of $w_i$ being the "real world". As each $w_i$ is just a Herbrand interpretation, the notion of satisfaction of an action formula $F$ by a world $w$, denoted by $w \mapsto F$, is defined in the usual way. The following constraints are in $CONS_U(\Pi, s, X)$:

1. If $F : [\ell, u] \in U_{\Pi_s}(X)$, then $\ell \leq \Sigma_{w_i \mapsto F}\, p_i \leq u$ is in $CONS_U(\Pi, s, X)$.
2. $\Sigma_{w_i} p_i = 1$ is in $CONS_U(\Pi, s, X)$.

We refer to these as constraints of type (1) and (2), respectively. Our operator $T_{\Pi_s}(X)$ is then defined as follows.

**Definition 2.7** *Suppose $\Pi$ is an ap-program, $s$ is a state, and $X$ is a set of ground ap-wffs. Our operator $T_{\Pi_s}(X)$ is then defined to be $\{F : [\ell(F), u(F)] \mid (\exists \mu) F : \mu \in U_{\Pi_s}(X)\} \cup \{A : [\ell(A), u(A)] \mid A$ is a ground action atom $\}$.*

Thus, $T_{\Pi_s}(X)$ works in two phases. It first takes each formula $F : \mu$ that occurs in $U_{\Pi_s}(X)$ and finds $F : [\ell(F), u(F)]$ and puts this in the result. Once all such $F : [\ell(F), u(F)]$'s have been put in the result, it tries to infer the probability bounds of all ground action atoms $A$ from these $F : [\ell(F), u(F)]$'s. The $T_{\Pi_s}(X)$ operator has a least fixpoint, $T^{\omega}_{\Pi_s}$, which contains all of the ground action atoms in $X$ annotated with tight probability intervals.

## 3 Maximally Probable Worlds

[KMN+07] introduces the problem of finding the most probable world for an ap-program. In this section we briefly summarize the problem and present a naive algorithm for solving it.

**Definition 3.1 (lower/upper probability of a world)** *Suppose $\Pi$ is an ap-program and $s$ is a state. The* lower probability*, $\mathsf{low}(w_i)$ of a world $w_i$ is defined as: $\mathsf{low}(w_i) =$ **minimize** $p_i$ **subject to** $\mathsf{CONS}_U(\Pi, s, T^{\omega}_{\Pi_s})$. The* upper probability*, $up(w_i)$ of world $w_i$ is defined as $up(w_i) =$ **maximize** $p_i$ **subject to** $\mathsf{CONS}_U(\Pi, s, T^{\omega}_{\Pi_s})$.*

Thus, the low probability of a world $w_i$ is the lowest probability that that world can have in any solution to the linear program. Similarly, the upper

probability for the same world represents the highest probability that that world can have. It is important to note that for any world $w$, we cannot *exactly* determine a point probability for $w$. This observation is true even if all rules in $\Pi$ have a point probability in the head because our framework does not make any simplifying assumptions (e.g. independence) about the probability that certain things will happen. As given in [KMN$^+$07] checking if the low (resp. *up*) probability of a world exceeds a given bound is in the class EXPTIME.

**The MPW Problem.** The *most probable world* problem (MPW for short) is the problem where, given an ap-program $\Pi$ and a state $s$ as input, we are required to find a world $w_i$ where $low(w_i)$ is maximal.

**A Naive Algorithm.** A *naive* algorithm to find the most probable world would be:

1. Compute $T^{\omega}_{\Pi_s}$; $Best = NIL$; $Bestval = 0$;
2. For each world $w_i$,
   (a) Compute $low(w_i)$ by minimizing $p_i$ subject to the set $\mathsf{CONS}_U(\Pi, s, T^{\omega}_{\Pi_s})$ of constraints.
   (b) If $low(w_i) > Bestval$ then set $Best = w_i$ and $Bestval = low(w_i)$;
3. If $Best = NIL$ then return any world whatsoever, else return $Best$.

The Naive algorithm does a brute force search after computing $T^{\omega}_{\Pi_s}$. It finds the low probability for each world and chooses the best one.

### 3.1 HOP Algorithms

In [KMN$^+$07], the authors present two algorithms that are more efficient than the naive algorithm, but also still produce an exact solution.

### 3.2 Heuristic Approximation Algorithm

In [KMN$^+$07], the authors also present a binary search style heuristic algorithm. The goal of the heuristic approximation algorithm is to reduce the number of variables in the linear program $\mathsf{CONS}_U(\Pi, s, T^{\omega}_{\Pi_s})$ for a SOMA-program $\Pi$, state $s$, and set of ground action literals $X$.

## 4 Parallel Approximation Algorithms for Finding a Maximally Probable world

In the previous sections we have briefly examined several algorithms that can be used to solve the maximally probable worlds problem. However, even with the given simplifications and heuristic approximation algorithms, the computation time and memory requirements do not always allow us to achieve the desired level of performance. In this section we will present various parallel versions of the the sequential algorithms presented earlier. Parallelism will not only reduce

```
Algorithm P − MPW(Π, s, T^ω_{Π_s}, CONS, m, n){
1.      remainingVars = divideVariables(CONS, m);
2.      while remainingVars not empty {
3.          for each node {
4.          v_j = pop(remainingVars);
5.              for each variable p_i in v_j {
6.                  VAL(p_i) = minimize p_i subject to CONS;
7.                  MAX_j = max VAL(p_i);
8.              }
9.          }
10.      }
11.      BestVal = max MAX_j;
12.      w_k = arg max MAX_j;
13.      return w_k;
}
```

Figure 1: The P-MPW Algorithm. This is the general parallel algorithm for computing the most probable worlds. The variable $remainingVars$ is a stack containing the $j$ sets of variables of size $m$; $divideVariables$ is a procedure that performs the division of $\mathsf{CONS}$ into these sets.

the computation time of the algorithms for finding the most probable worlds, but will also allow us to examine a greater number of worlds, permitting analysis of larger *ap*-programs, and possibly improving the accuracy of the end result.

### 4.1 Parallelism for Reducing Computation Time

All of our algorithms given above lend themselves to being parallelized in a straightforward way. This new class of algorithms, the P-MPW (Parallel Maximally Probable World) algorithms, operate identically to the serial algorithms, except that the computation of $\mathsf{low}(w_i)$ or $\mathsf{up}(w_i)$ for all of the worlds $w_i$ is distributed among $n$ nodes of a computing cluster such that $m$ worlds at a time are given to each node. Figure 1 contains the basic P-MPW algorithm in pseudo code.

The number $m$ of worlds for which to compute the *low* or *up* values can be determined in several ways. The most obvious is to simply divide the problem evenly across all of the $n$ nodes such that $m = \frac{|\mathsf{CONS}_U(\Pi, s, T^\omega_{\Pi_s})|}{n}$ where $|\mathsf{CONS}_U(\Pi, s, T^\omega_{\Pi_s})|$ is the number of variables in $\mathsf{CONS}_U(\Pi, s, T^\omega_{\Pi_s})$.

The $\mathsf{CONS}$ parameter of the P-MPW algorithm can be either $\mathsf{CONS}_U(\Pi, s, T^\omega_{\Pi_s})$, $\mathsf{RedCONS}_U(\Pi, s, T^\omega_{\Pi_s})$, $\mathsf{S\_RedCONS}_U(\Pi, s, T^\omega_{\Pi_s})$, or the $\mathsf{CONS}'$ returned by the binary heuristic. The basic P-MPW and the PAMPW algorithms allow for, in the best case, a computation time improvement of up to a factor of $n$, where $n$

```
Algorithm PAMPW − MS(Π, s, T^ω_{Π_s}, m, n, ε, r, CONS){
1.      for each node {
2.          CONS_{n_j} = Binary(T^ω_{Π_s}, r, ε, CONS)
3.          for each variable p_i in CONS_{n_j} {
4.              VAL(p_i) = minimize p_i subject to CONS_{n_j}
5.              MAX_{n_j} = max VAL(p_i)
6.          }
7.      }
8.      BestVal = max MAX_{n_j}
9.      w_k = arg max MAX_{n_j}
10.      return w_k
}
```

Figure 2: The PAMPW-MS Algorithm. This is a parallel algorithm for computing an approximation of the most probable world using the binary heuristic. On each node, the binary heuristic algorithm is used to select a different reduced set of constraints containing $r$ variables. The most probable world is then that with the maximum low probability across all of the nodes.

is the number of nodes in the cluster.

### 4.2 Parallelism for Improving Solution Accuracy of Heuristics

We can also utilize parallel algorithms to improve the quality of the final solutions. In this section we will describe explicitly parallel algorithms that are able to take into account additional samples of worlds for the heuristic approximations, propagating the most probable worlds from each sample throughout the successive computations and allowing more thorough comparisons between the most probable worlds found by each iteration of the parallel computation.

The first algorithm, PAMPW-MS (Parallel Appproximation of the Maximally Probable Worlds-Multi-Sample), allows us to examine a greater proportion of the possible worlds in computing the most probable world. With this method, each parallel computation investigates a distinct sample of possible worlds for the binary heuristic constraint selection algorithm; the resulting most probable worlds from each sample are then compared to find the most probable world overall. Using the PAMPW-MS algorithm we are able to look at larger samples of the possible worlds and thereby have a better chance of finding an approximate solution that is more accurate with respect to the solutions returned by the naive, HOP or SemiHOP algorithms. Figure 2 contains pseudo code for the PAMPW-MS algorithm.

Using PAMPW-MS we can further generalize the functionality of the algorithm to find the $k$ most probable worlds from each node and compare these sets of worlds to find the $k$ worlds that are the most probable overall.

To further increase our ability to examine a larger sample of the possible

```
Algorithm iPAMPW − MS(Π, s, T^ω_{Π_s}, m, n, ε, r, i, k, CONS){
1.      for iteration=1 to iteration=i {
2.         for each node {
3.            prevWorlds_{n_j,i} = ∅
4.            CONS_{n_j,i} = modConstraints(prevWorlds_{n_j,i}, Binary(T^ω_{Π_s}, r, ε, CONS))
5.            for each variable p_i in CONS_{n_j,i} {
6.               VAL(p_i) = minimize p_i subject to CONS_{n_j,i}
7.               MAX_{n_j,i} = max_k VAL(p_i)
8.               prevWorlds_{n_j,i} = k arg max MAX_{n_j,i}
9.            }
10.         }
11.      }
12.      BestVal = max MAX_{n_j,i}
13.      w = arg max MAX_{n_j,i}
14.      return w
}
```

Figure 3: The iPAMPW-MS Algorithm. This algorithm solves multiple iterations of the PAMPW-MS, using the $modConstraints(prevWorlds, Binary(T^{\omega}_{\Pi_s}, r, \epsilon, \mathsf{CONS}))$ function to perform the Binary heuristic, replacing $k$ of the $r$ constraints with the variables in the set prevWorlds. The most probable world is then that with the maximum low probability across all of the nodes and iterations.

worlds, we have also developed an iterative version of the PAMPW-MS algorithm, called the iPAMPW-MS. In iPAMPW-MS, we first compute the $k$ most probable worlds on each node of the cluster as described in PAMPW-MS. Then, we again generate a set of constraints for each node, propagating the $k$ most probable worlds from the first iteration into the second sample set. For example, if our world selection method selects 1000 worlds and $k$ is chosen to be 20, then in the second iteration we only select 980 worlds and automatically include the 20 most probable worlds from previous computation. Using this new set of constraints, the $k$ most probable worlds are again computed and propagated into the next iteration. We continue this process until we have completed $I$ iterations of the algorithm, choosing the final $k$ most probable worlds from the last sets of $k$ obtained across all processors. This iterative process allows us to progressively refine the solution set of the $k$ most probably worlds, improving the accuracy of the approximation heuristic algorithms. The steps in the iPAMPW-MS algorithm are given in the pseudo code in Figure 3.

### 4.3 Parallelism for increasing computation capacity

Last, but not least, rather than simply distributing the MPW algorithms and performing the same computation in a shorter amount of time, we can also

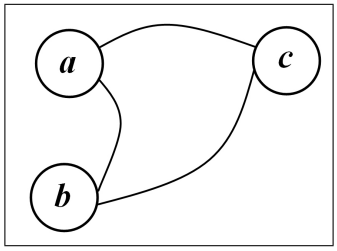


Figure 4: The literal-relationship graph $G^{\Pi}$ for a simple *ap*-program $\Pi$.

design a "pleasantly parallel" algorithm for finding the most probable world of larger *ap*-programs, i.e. programs with a larger number of ground atoms. To accomplish this task, we must be able to distribute the set of constraints among nodes in a cluster, rather than simply dividing the task of computing *low* or *up* for each of the worlds $w_i$.

An *ap*-program can be represented as a graph in which the vertices are literals in the program and an edge indicates that its two endpoints occur together in a *ap*-rule.

**Definition 4.1 (Literal Relationship (LR) Graph)** *Let* $\Pi$ *be an* ap*-program. The* literal relationship graph $G^{\Pi} = (V, E)$ *is an* undirected *graph defined as follows.*
$V = \{l \mid l$ *is a literal (positive or negative) appearing in a rule in* $\Pi\}$.
$E = \{(l_i, l_j) \mid l_i, l_j \in V$ *and* $l_i$ *and* $l_j$ *are either complementary literals or they both appear in a rule in* $\Pi\}$.
*We use* $G^{\Pi} = (V, E)$ *to denote the* Literal Relationship Graph *for the program* $\Pi$.

Consider the simple *ap*-program $\Pi$:

$$\begin{array}{lll} (a \vee b) & : [0.7, 1] & \leftarrow . \\ ((a \wedge b) \vee (b \wedge c)) & : [0.2, 0.6] & \leftarrow . \\ (a) & : [0.4, 0.4] & \leftarrow . \end{array}$$

Figure 4 shows the LR-graph associated with $\Pi$.

The *rank* of an LR-graph is the maximum cardinality of the connected components of the graph.

**Definition 4.2 (Rank of an *LR*-Graph)** *Let* $\Pi$ *be an* ap*-program, and* $G^{\Pi} = (V, E)$ *be the LR-Graph for* $\Pi$. *We say that graph* $G^{\Pi}$ *has* rank $k$*, if* $k$ *is the maximum cardinality of any connected component in* $G^{\Pi}$.

For example, when the rank of the LR-graph $G^{\Pi}$ is 1, this means that all rules in $\Pi$ are only literals, and there are no complementary literals. Note that the graph in our example has rank 3. On the other hand, if we deleted the second probabilistic statement from the program $\Pi$ for Figure 4, we would have a graph of rank 2. Note that we can compute the rank of an LR-graph in polynomial

```
Algorithm PAMPW − LR(Π, s, T^ω_{Π_s}, m, n, ϵ, r, CONS){
1.      G = buildLR(Π)
2.      components = getComponents(G)
3.      while components not empty {
4.          for each node {
5.          c_j = pop(components)
6.          CONS_{c_j} = Binary(c_j, s, r, ϵ, CONS)
7.              for each variable p_i in CONS_{c_j} {
8.                  VAL(p_i) = minimize p_i subject to CONS_{c_j}
9.                  MAX_{c_j} = max VAL(p_i)
10.             }
11.         }
12.     }
13.     BestVal = max MAX_{c_j}
14.     w_k = arg max MAX_{c_j}
15.     return w_k
}
```

Figure 5: The PAMPW-LR Algorithm. This is a parallel algorithm for computing an approximation of the most probable world. The variable *components* is a stack containing the $j$ connnected components in $G^{\Pi}$; *getComponents* is a procedure that returns these connected components.

time (w.r.t. the size of the graph), and we can also compute the LR-graph itself in polynomial time. As a consequence, checking if the LR-graph's rank is below some *a priori* set bound $b$ is a polynomial-time operation.

Each connected component $c$ in an LR-graph $G^{\Pi}$ represents a subprogram $\Pi_c$ of $\Pi$ that utilizes only the literals in that component. Therefore, each connected component comprises a separate set of linear constraints $\mathsf{CONS}(\Pi_c, s, T^{\omega}_{\Pi_s})$, $\mathsf{RedCONS}_U(\Pi_c, s, T^{\omega}_{\Pi_s})$, $\mathsf{S\_RedCONS}_U(\Pi_c, s, T^{\omega}_{\Pi_s})$, or $CONS'$. By finding the maximally probable world in each component, we can compare these individual solutions and find the maximally probable world across all components of the original set of constraints for $\Pi$. The PAMPW-LR algorithm uses this methodology to divide a much larger *ap*-program into smaller pieces that can be computed in parallel. On a cluster with $n$ nodes, the PAMPW-LR algorithm assigns a connected component of $G^{\Pi}$ to each node and then computes the most probable world of each component. The algorithm will then aggregate the results and return the most probable world overall. While the PAMPW-LR approach does not provide any time savings with regards to the computation time, it does allow the analysis of much larger *ap*-programs that can be divided into computationally feasible parallel components. Figure 5 contains pseudo code for the PAMPW-LR algorithm.

| Worlds | Naive | SemiHOP | $\text{Naive}_{Binary}$ | $\text{HOP}_{Binary}$ | $\text{SemiHOP}_{Binary}$ |
|---|---|---|---|---|---|
| 32 | 50.48 | 02.73 | 111.21 | 94.40 | 76.64 |
| 64 | 66.15 | 40.88 | 123.72 | 97.78 | 80.26 |
| 128 | 83.23 | 47.97 | 121.58 | 97.10 | 78.06 |
| 256 | 89.47 | 52.01 | 86.80 | 13.61 | 44.73 |
| 512 | 90.29 | 55.84 | 89.95 | 29.15 | 9.68 |
| 1024 | 72.45 | 49.16 | 89.81 | 1.86 | 27.88 |
| Avg | 75.35 | 41.43 | 103.84 | 55.65 | 52.87 |

Table 1: Percent Speedup Achieved with P-MPW Algorithms

## 5 Implementation and Experiments

In [KMN+07] the authors implemented several of the serial algorithms mentioned in this paper—the naive, HOP, SemiHOP, and the binary heuristic algorithms—using approximately 6,000 lines of Java code. In addtion to the basic MPW algorithms, we also implemented the P-MPW algorithm and the PAMPW-MS iterative algorithm, using about 6,700 lines of Java code; these implementations will be described in more detail in this section. Experiments were performed for P-MPW applied to the naive, HOP, SemiHOP and binary heuristic serial algorithms. Our experiments were performed on a Linux computing cluster comprised of 64 8-core, 8-processor nodes with between 10GB and 20GB of RAM. The linear constraints were solved using the QSopt linear programming solver library, and the logical formula manipulation code from the COBA belief revision system and SAT4J satisfaction library were used in the implementation of the HOP and SemiHOP algorithms.

For each experiment using P-MPW, we held the number of rules constant at 10 and did the following: (i) we generated a new *ap*-program and sent it to each of the three algorithms, (ii) varied the number of worlds from 32 to 16,384, performing at least 10 runs for each value and recording the average time taken by each parallel algorithm relative to the serial experiments in [KMN+07] and the other parallel running times, and (iii) we also measured the quality of the SemiHOP and all algorithms that use the binary heuristic by calculating the average distance from the solution found by the exact algorithm. In the discussion below we use the metric $ruledensity = \frac{\mathcal{L}_{act}}{card(T^{\omega}_{\Pi_s})}$ to represent the size of the *ap*-program; this allows for the comparison of the naive and HOP and SemiHOP algorithms as the number of worlds increases.

### 5.1 Parallel Implementations

Each experimental run utilized 16 processors in parallel on the above mentioned computing cluster to solve a single MPW problem. As expected, the P-MPW algorithms produce a marked speedup in the computation time for finding the most probable world (Figure 6). Where the basic naive algorithm requires

almost 4 hours (13,636.23 seconds) for problems with 1,024 possible worlds, the naive P-MPW algorithm completed the same computation in only about an hour (4016.83 seconds). This is a very promising result for situations where an exact solution is necessary. We see a similar speedup for the SemiHOP and heuristic algorithms; the P-MPW SemiHOP algorithm uses slightly under 6 minutes (339.65 seconds) as opposed to 33.47 minutes (2,008.1 seconds) to solve for 1,024 worlds, and the naive heuristic improves from 136.08 seconds to 21.78 seconds. In some cases, however, the P-MPW version of the SemiHOP algorithm actually performs worse as compared to the serial SemiHOP algorithm. This anomoly occurs in those instances where there are no subpartitions with only a single satisfying interpretation; in such cases, we do not actually need to solve an MPW computation (as described in Section **??**), so the overhead of managing parallel threads is greater than the running time of the serial version. In most instances, though, the P-MPW algorithm greatly improves the efficiency of computing the most probable world. Table 1 contains the average speedup achieved by using the P-MPW algorithms compared to their serial counterparts. Similar improvements can be seen when using the P-MPW heuristic algorithms on large numbers of worlds, providing an average speedup of about 66%. These running times are shown in Figure 7.

## 6 Conclusions

The three algorithms and the Binary heuristic developed in [KMN+07] scalably find the most probable world of an *ap*-program. In this paper, we have expanded on the scalability and efficiency of the naive, HOP, and SemiHOPalgorithms by creating several explicitly parallel versions of the algorithms. Through parallel computing, we can improve the computation time and solution quality by utilizing greater resources, and can calculate the most probable world for larger *ap*-programs by distributing the workload. For this paper, we implemented the basic P-MPW algorithm and demonstrated significant speed up from this pleasantly parallel approach as compared with the serial algorithms. As future work, we plan to implement both the PAMPW-MS and PAMPW-LR algorithms to test the effectivness of algorithms to increase solution quality and computational capacity when approximating the most probable world.

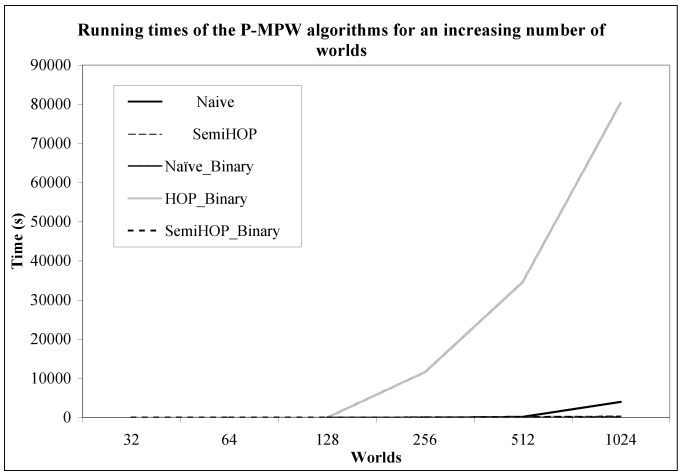


Figure 6: Running time of the P-MPW versions of the naive, SemiHOP, naive$_{bin}$,HOP$_{bin}$, and SemiHOP$_{bin}$ algorithms for an increasing number of worlds.

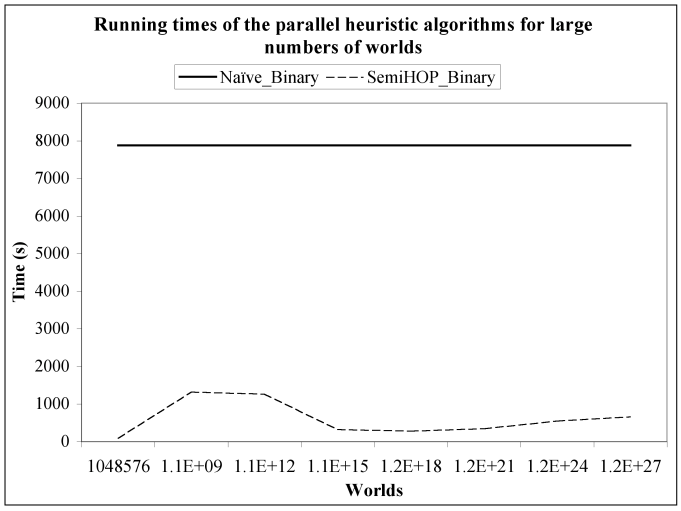


Figure 7: Running time of the P-MPW versions of the naive$_{bin}$ and SemiHOP$_{bin}$ algorithms for large numbers of worlds.

## References

[FHM90] Ronald Fagin, Joseph Y. Halpern, and Nimrod Megiddo. A logic for reasoning about probabilities. *Information and Computation*,

87(1/2):78–128, 1990.

[Hai84] T. Hailperin. Probability logic. *Notre Dame Journal of Formal Logic*, 25 (3):198–212, 1984.

[KMN+07] Samir Khuller, Maria Vanina Martinez, Dana Nau, Gerardo Simari, Amy Sliva, and VS Subrahmanian. Finding most probable worlds of probabilistic logic programs. In *International Conference on Scalable Uncertainty Management (SUM 2007)*. Springer-Verlag (To Appear), 2007.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming, Second Edition*. Springer-Verlag, 1987.

[Nil86] Nils Nilsson. Probabilistic logic. *Artificial Intelligence*, 28:71–87, 1986.

[NS91] Raymond T. Ng and V. S. Subrahmanian. A semantical framework for supporting subjective and conditional probabilities in deductive databases. In Koichi Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 565–580. The MIT Press, 1991.

[NS92] Raymond T. Ng and V. S. Subrahmanian. Probabilistic logic programming. *Information and Computation*, 101(2):150–201, 1992.

[SAM+07] V.S. Subrahmanian, M. Albanese, V. Martinez, D. Reforgiato, G. Simari, A. Sliva, O. Udrea, and J. Wilkenfeld. CARA: A Cultural Reasoning Architecture. *IEEE Intelligent Systems*, 22(2):12–16, 2007.