1 Introduction

Since almost all computers have multiple cores, most software in future would be mutithreaded. Writing parallel (or distributed) programs is hard. There are two aspects of building a working correct program. First and foremost the program must be correct. However, the program must also run fast. Optimizing the program to make it run faster may introduce bugs that affect the correctness of the program. Furthermore, the optimization of the program may be dependent on the platform that it runs. Optimizing the program for various platforms is expensive because it requires maintaining multiple versions of the program for different platforms. In this project, we are developing two main ideas to alleviate these difficulties for the domain of discrete optimization problems.

First, we view the program as consisting of two parts — the *core* part and the *controller* part. The core part relates to the correctness of the program and may be inefficient, even exponentially slower than the final executable. It is, however, correct; it always terminates with the correct answer. This part of the program has massive nondeterminism. The core program can execute any of the enabled events. The second part of the program controls the execution of the enabled events. This controller may run these events in a complete deterministic replayable fashion (to help programmer debug his program), in a parallel fashion where multiple events that are enabled can execute in parallel, or in a distributed fashion where the state of the system is distributed and messages (push or pull) are required to determine whether some event is enabled.

Second, we view the program as a system that is making progress towards desirable predicates. Whenever all the desirable predicates are true, the system is considered to be terminated. These programs are naturally parallel and non-deterministic. Multiple desired predicates may be false and the program is correct if it reaches the correct state irrespective of the sequence of actions that the system takes. While it may appear at first that only a small class of systems can be modeled in this manner, there is a large set of applications that can be modeled in this manner. Rule based systems (e.g. [Hil03]) are primarily based on detecting which rules can be fired and then executing those rules. Languages such as Unity [CM88], are also based on conditional multiple assignment statements. Unity does not have any control flow and any statement whose condition is true can be executed. Dijkstra's guarded command language [Dij78], or Hoare's communicating sequential processes language [Hoa78] can also be viewed in terms on executing statements or commands whose guards are true. Many user interface programs and distributed event based systems [MFP06] are also written using this paradigm.

Our project differs in these earlier efforts in two ways. First, we view programs as finding elements in a distributive lattice [DP90] that satisfy desirable predicates. The notion of order in the search space is fundamental in our system. We are almost always interested in a solution that either minimizes or maximizes certain metric. We exploit the property of the predicate with respect to this order to either optimize the associated action, or check the validity of the associated action. For example, when the predicate is lattice linear (defined later), we know that different threads in a multicore program can update their respective state independently without synchronization. In a distributed system setting, if a predicate is lattice linear, then it is correct to update the local state of a process based on a predicate that is evaluated on the old information from other processes. Second, all the earlier work did not have any component for controlling non-determinism. For example, in Unity or in rule based systems, it is the programmer's responsibility to add enough state if she wanted a particular order of execution of events. This not only adds to the programming burden, but also violates separation of concerns. The program with most non-determinism is generally the simplest and most general to solve the problem and modifying it to control non-determinism for efficiency purposes on a given platform makes it harder to understand and maintain. In this project, we are exploring ways to separate these two concerns. As a toy example, consider the problem of finding the shortest cost from a vertex to all other vertices. Let $G[i]$ be the cost of reaching vertex i from the source vertex. In a parallel implementation, if the array is split across threads, then the details of this partition or synchronization between these threads are separate from the core algorithm. In a distributed implementation, the details of when a change in $G[i]$ is communicated to other processes are separate from the core algorithm. Our system separates these concerns.

While our work is applicable for development of sequential programs, the focus of the project is on parallel (shared memory) and distributed (message based) programs. The fundamental aspect of our system is detecting global predicates in such systems. This work builds on earlier work on predicate detection in distributed systems in 90's. In the context of distributed monitoring, the technique of predicate detection was introduced by Cooper and Marzullo [CM91] and Garg and Waldecker [GW91]. Detection of conjunctive predicates was discussed by Garg and Waldecker in [GW92]. Linear predicates were introduced by Chase and Garg [CG98a], and regular predicates were introduced by Garg and Mittal [GM01]. Distributed on-line algorithms for detecting conjunctive predicates were presented in Garg and Chase [GC95]. Observer-independent predicates were introduced by Charron-Bost, Delporte-Gallet, and Fauconnier [CBDGF95]. Some of this early work has been improved with new methods of tracking causality [TVK18]. This early work on predicate detection also has applications in runtime verification [SSA$^+$18]. Computation slices are abstractions of computations with respect to predicates of interest. Some online algorithms for computing slices with respect to some temporal formulas are described in [CGNM13, NMG14]. Mostafa and Bonakdarpour [MB15, MBFJ16] use online computation slicing [CGNM13] to detect formulas that are in a restricted class of LTL formulas and report an implementation based on their algorithm.

The notion of predicate detection has also been investigated in the context of parallel programming. Autosynch [HG13] describes an automatic signal monitor based on predicate tagging. More recently, [FVGDS18] provides implicit signal monitoring based on symbolic reasoning on predicates at the compile time.

While the initial work on predicate detection was motivated by applications to distributed debugging in distributed systems and efficient conditional synchronization in parallel systems, it turns out that many discrete optimization problems can be cast in terms of predicate detection. For example, the stable marriage problem, the shortest path problem, the market clearing prices problem, the minimum spanning tree problem, the housing allocation problem etc. can all be viewed as detecting a lattice-linear predicate over a distributive lattice. The classical algorithms to solve these problems are the Gale-Shapley algorithm for the stable marriage problem [GS62], Dijkstra's algorithm [Dij59] and Bellman-Ford algorithm [Bel58, For56] for the shortest path problem, Kuhn's Hungarian method to solve the assignment problem [Mun57] (or equivalently, Demange-Gale-Sotomayor auction-based algorithm [DGS86] for market clearing prices), Prim's algorithm for the minimum spanning tree problem, and Gale's top trading cycle algorithm [SS74] for the house allocation problem. The algorithm to detect lattice-linear predicates is a single efficient parallel algorithm that solves all of these problems. Thus, predicate detection technique also serves to unify a large class of algorithms.

There are two components of the proposed project. The first component deals with the theory of predicate detection to build efficient parallel and distributed algorithms for a large class of problems. The programmer specifies the search space and the *progress* predicates. Progress predicates are dual of program invariants. These predicates are not true initially and the goal of the program is to make them true while keeping program invariants true. Depending upon the type of the progress predicate, the system can apply various optimization techniques. The programmer also specifies the mode of execution and possibly a partition of the search space for the parallel and distributed system setting. This is sufficient to build a program that solves the problem in the corresponding setting. A rough analogue of the proposed project is the map reduce system [DG08] such as Hadoop in which the programmer specifies the map and reduce function and the mode of execution to get a working system in a sequential, a local mode or a distributed mode. Instead of map and reduce functions, in our system the programmer specifies progress predicates and the search space. The challenge is to develop efficient execution strategies for different types of predicates. We have developed methods for a class of predicates called lattice-linear. As mentioned earlier, many classical algorithms can be cast as detection of lattice-linear predicates. However, many other problems such as the matching problem, and the max-flow problem cannot be cast in terms of the lattice linear predicates. We are investigating techniques to address predicates that are not lattice linear.

The second component of the project involves designing a programming system that implements the specification of controlling non-determinism in the core part of the program. There are two parts of this specification. The first part specifies a partial order on all the predicates that are eligible for progress (or, all the eligible events). The system maintains the partial order and executes only the minimal events in the partial order. For example, in processing jobs with a prerequisites, the partial order corresponds to the prerequisite order. For the shortest path problem, the order may correspond to the tentative cost of reaching the vertex. For the stable marriage problem, we may choose the man who has been rejected most to advance or we may choose the man with the least identifier. The second part of the specification corresponds to the mode of the execution and the partition of the underlying array into threads or processes. For multicore systems, the program would need to synchronize and we are studying multiple strategies for reducing this overhead. For distributed systems, the state may be distributed and the evaluation of predicates or the corrective action may require sending and receiving of messages. We are developing distributed algorithms for predicate detection that reduce the number of messages. The mode of execution also affects the order in which events are executed. For example, a thread may choose only those events to which are local to that thread. In case, a thread has no local event, it may choose the event from the thread with the most load.

This proposal is organized as follows. Section 2 describes a particular class of predicates, lattice-linear predicates, that is quite useful in solving discrete optimization problems. Section 3 describes our proposed work in specification of the core component of the system. It also gives examples of problems that can be solved using lattice-linear predicates. Section 4 describes our proposed work in specification of the controller component of the system. Section 5 describe other classes of predicates which may also be amenable to efficient generation of the search program.

2 Lattice-Linear Predicates

In this section we describe lattice-linear predicates and its application to deriving solutions for the discrete optimization problems. Let L be the lattice of all n-dimensional vectors of reals greater than or equal to zero vector and less than or equal to a given vector T where the order on the vectors is defined by the component-wise natural $\leq$. The minimum element of this lattice is the zero vector. The lattice is used to model the search space of the combinatorial optimization problem. For simplicity, we are considering the lattice of vectors of non-negative reals; our results are applicable to any distributive lattice. The combinatorial optimization problem is modeled as finding the minimum element in L that satisfies a boolean *predicate* B, where B models *feasible* (or acceptable solutions). We are interested in concurrent algorithms to solve the combinatorial optimization problem with n processes. We will assume that the systems maintains as its state the current candidate vector $G \in L$ in the search lattice, where $G[i]$ is maintained at process i. We call G, the global state, and $G[i]$, the state of process i.

Finding an element in lattice that satisfies the given predicate B, is called the *predicate detection* problem. Finding the *minimum* element that satisfies B (whenever it exists) is the combinatorial optimization problem. We now define *lattice-linearity* which enables efficient computation of this minimum element. Lattice-linearity is first defined in [CG98b] in the context of detecting global conditions in a distributed system where it is simply called linearity. We use the term *lattice-linearity* to avoid confusion with the standard usage of linearity.

A key concept in deriving an efficient predicate detection algorithm is that of a *forbidden* state. Given a predicate B, and a vector $G \in L$, a state $G[i]$ is *forbidden* (or equivalently, the index i is forbidden) if for any vector $H \in L$, where $G \leq H$, if $H[i]$ equals $G[i]$, then B is false for H. Formally,

Definition 1 (Forbidden State [CG98b]). *Given any distributive lattice L of n-dimensional vectors of $\mathbf{R}_{\geq 0}$, and a predicate B, we define $forbidden(G, i, B) \equiv \forall H \in L : G \leq H : (G[i] = H[i]) \Rightarrow \neg B(H)$.*

We define a predicate B to be *lattice-linear* with respect to a lattice L if for any global state G, B is false in G implies that G contains a *forbidden state*. Formally,

Definition 2 (lattice-linear Predicate [CG98b]). *A boolean predicate B is* lattice-linear *with respect to a lattice L iff $\forall G \in L : \neg B(G) \Rightarrow (\exists i : forbidden(G, i, B))$.*

We now give some examples of lattice-linear predicates.

1. **Job Scheduling Problem**: Our first example relates to scheduling of n jobs. Each job j requires time t_j for completion and has a set of prerequisite jobs, denoted by $pre(j)$, such that it can be started only after all its prerequisite jobs have been completed. Our goal is to find the minimum completion time for each job. We let our lattice L be the set of all possible completion times. A completion vector $G \in L$ is feasible iff $B_{jobs}(G)$ holds where $B_{jobs}(G) \equiv \forall j : (G[j] \geq t_j) \wedge (\forall i \in pre(j) : G[j] \geq G[i] + t_j)$. B_{jobs} is lattice-linear because if it is false, then there exists j such that either $G[j] < t_j$ or $\exists i \in pre(j) : G[j] < G[i] + t_j$. We claim that $forbidden(G, i, B_{jobs})$. Indeed, any vector $H \geq G$ cannot be feasible with $G[j]$ equal to $H[j]$. The minimum of all vectors that satisfy feasibility corresponds to the minimum completion time.

2. **Shortest Path Problem**: We are given a weighted directed graph and a fixed vertex s. We are required to find the cost of the shortest path from s to all vertices. Let the input be specified as $w[i, j]$ as the cost of going from i to j. Here our objective is to output maximum $G[j]$ subject to constraints that $G[j]$ is less than or equal to $G[i] + w[i, j]$ for all $i \in pre(j)$. One can view $G[j]$ as an upper bound on the cost of reaching j. We assume that there are no negative cycles and thus $G[s]$ equals zero. For this problem, the order on the underlying lattice is inverted. The lattice is defined on the value of $G[j]$ for all vertices except the source vertex. The minimum element is the vector with all components as ∞. It is easy to check that the predicate $G[j] \leq \min\{G[i] + w[i, j] \mid i \in pre(j)\}$ is lattice-linear. If $G[j] > G[i] + w[i, j]$ for some (i, j) then it will continue to hold until G is advanced on j, i.e., $G[j]$ value is reduced at least to $G[i] + w[i, j]$.

3. **Continuous Optimization Problem**: We are required to find minimum nonnegative x and y such that $B \equiv (x \geq 2y^2 + 5) \wedge (y \geq x - 4)$. We view this problem as finding minimum (x, y) pair such that B holds. It is easy to verify that B is lattice-linear. If the first conjunct is false, then x is forbidden. Unless x is increased the predicate cannot become true, even if other variables (y for this example) increase. If the second conjunct is false, then y is forbidden.

4. **A Non Lattice-Linear Predicate** As an example of a predicate that is not lattice-linear, consider the predicate $B \equiv \sum_j G[j] \geq 1$ defined on the space of two dimensional vectors. Consider the vector G equal to $(0, 0)$. The vector G does not satisfy B. For B to be lattice-linear either the first index or the second index should be forbidden. However, none of the indices are forbidden in $(0, 0)$. The index 0 is not forbidden because the vector $H = (0, 1)$ is greater than G, has $H[0]$ equal to $G[0]$ but it still satisfies B. The index 1 is also not forbidden because $H = (1, 0)$ is greater than G, has $H[1]$ equal to $G[1]$ but it satisfies B.

In Section 3, we give many other examples of lattice-linear predicates including that for stable marriage and market clearing prices.

The following Lemma is useful in proving lattice-linearity of predicates.

Lemma 1. *[Gar18a] Let B be any boolean predicate defined on a lattice L of vectors.*
(a) Let $f : L \to \mathbf{R}_{\geq 0}$ be any monotone function defined on the lattice L of vectors of $\mathbf{R}_{\geq 0}$. Consider the predicate $B \equiv G[i] \geq f(G)$ for some fixed i. Then, B is lattice-linear.
(b) Let L_B be the subset of the lattice L of the elements that satisfy B. Then, B is lattice-linear iff L_B is closed under meets.
(c) If B_1 and B_2 are lattice-linear then $B_1 \wedge B_2$ is also lattice-linear.

For the job scheduling example, we can define B_j as $G[j] \geq max(t_j, max\{G[i] + t_j \mid i \in pre(j)\})$. Since $f_j(G) = max(t_j, max\{G[i] + t_j \mid i \in pre(j)\})$ is a monotone function, it follows from Lemma 1(a) that B_j is lattice-linear. The predicate $B_{jobs} \equiv \forall j : B_j$ is lattice-linear due to Lemma 1(c). Also note that the problem of finding the minimum vector that satisfies B_{jobs} is well-defined due to Lemma 1(b).

We now discuss detection of lattice-linear predicates which requires an additional assumption called the *efficient advancement property* [CG98b] — there exists an efficient (polynomial time) algorithm to determine the forbidden state. This property holds for all the problems discussed in this proposal. Once we determine j such that $forbidden(G, j, B)$, we also need to determine how to advance along index j. To that end, we extend the definition of forbidden as follows.

Definition 3 (α-forbidden). *Let B be any boolean predicate on the lattice L of all assignment vectors. For any G, j and positive real $\alpha > G[j]$, we define $forbidden(G, j, B, \alpha)$ iff*

$$\forall H \in L : H \geq G : (H[j] < \alpha) \Rightarrow \neg B(H).$$

Given any lattice-linear predicate B, suppose $\neg B(G)$. This means that G must be advanced on all indices j such that $forbidden(G, j, B)$. We use a function $\alpha(G, j, B)$ to determine the value that $G[j]$ can advance to safely. With the notion of $\alpha(G, j, B)$, we have the algorithm LLP shown in Fig. 1. The algorithm LLP has two inputs — the predicate B and the top element of the lattice T. It returns the least vector G which is less than or equal to T and satisfies B (if it exists). Whenever B is not true in the current vector G, the algorithm advances on all forbidden indices j *in parallel*. This simple parallel algorithm can be used to solve a large variety of combinatorial optimization problems by instantiating different $forbidden(G, j, B)$ and $\alpha(G, j, B)$.

```
vector **function** getLeastFeasible($T$: vector, $B$: predicate)
**var** $G$: vector of reals initially $\forall i : G[i] = 0$;
**while** $\exists j : forbidden(G, j, B)$ **do**
    **for all** $j$ such that $forbidden(G, j, B)$ **in parallel**:
        **if** $(\alpha(G, j, B) > T[j])$ then return null;
        **else** $G[j] := \alpha(G, j, B)$;
**endwhile**;
**return** $G$; // the optimal solution
```

Figure 1: Algorithm LLP to find the minimum vector less than or equal to T that satisfies B

For the job scheduling example, we get a parallel algorithm to find the minimum completion time by using $forbidden(G, j, B_{jobs}) \equiv (G[j] < t_j) \vee (\exists i \in pre(j) : G[j] < G[i] + t_j)$, and $\alpha(G, j, B_{jobs}) = \max\{t_j, \max\{G[i] + t_j \mid i \in pre(j)\}\}$.

On account of Lemma 1(c), if we have a parallel algorithm for a problem, then we also have one for the constrained version of that problem. Let LLP be the parallel algorithm to find the least vector G that satisfies B_1 if one exists. Then, LLP can be adapted to find the least vector G that satisfies $B_1 \wedge B_2$ for any lattice-linear predicate B_2 with the following changes: $forbidden(G, j, B_1 \wedge B_2) \equiv forbidden(G, j, B_1) \vee forbidden(G, j, B_2)$, and
$\alpha(G, j, B_1 \wedge B_2) = \max\{\alpha(G, j, B_1), \alpha(G, j, B_2)\}$.

For example, suppose that we want the minimum completion time of jobs with the additional lattice-linear constraint that $B_2(G) \equiv (G[1] = G[2])$. B_2 is lattice-linear with $forbidden(G, 1, B_2) \equiv (G[1] < G[2])$ and $forbidden(G, 2, B_2) \equiv (G[2] < G[1])$. From Lemma 1(c), we get a parallel algorithm for the constrained version.

3 Proposed Work: A Programming System based on Lattice-Linear Predicates (LLP)

We first go over the notation used in the proposed system. We have a single variable G in all the examples shown in Fig. 2. All other variables are derived directly or indirectly from G. G is an array of objects such that $G[j]$ is the state of thread j for a parallel program (process j for a distributed program or an index j for a sequential program). From now on, we will simply use the paradigm of parallel programming, though the concepts are applicable to distributed or sequential programs. There are three sections of the program.

The **init** section is used to initialize the state of the program. All the parts of the program are applicable to all values of j. For example, the *init* section of the job scheduling program in Fig. 2 specifies that $G[j]$ is initially $t[j]$. Every thread j would initialize $G[j]$.

The **always** section defines additional variables which are derived from G. The actual implementation of these variables are left to the system. They can be viewed as macros. For example, in the stable marriage problem, for any thread $z = mpref[j][G[j]]$. This means that whenever $G[j]$ changes, so does z (just like a formula in a spreadsheet).

The third section gives the desirable predicate either by using **ensure** or its complement by using **forbidden**. The *ensure* section specifies the desirable predicates of the form $(G[j] \geq expr)$ or $(G[j] \leq expr)$. The statement *ensure* $G[j] \geq expr$ simply means that whenever thread j finds $G[j]$ to be less than $expr$; it can advance $G[j]$ to $expr$. Since $expr$ may refer to G, just by setting $G[j]$

```
$P_j$:
var $G$: array[1..$n$] of 0..$maxint$; // common declaration for all the programs below

job-scheduling:
    input: $t[j] : int$, $pre(j)$: list of 1..$n$;
    init: $G[j] := t[j]$;
    ensure: $G[j] \geq \max\{G[i] + t[j] \mid i \in pre(j)\}$;

shortest path from node $s$: Bellman-Ford
    input: $pre(j)$: list of 1..$n$; $w[i,j]$: int for all $i \in pre(j)$
    init: if $(j = s)$ then $G[j] := 0$ else $G[j]$ := maxint;
    ensure: $G[j] \leq \min\{G[i] + w[i,j] \mid i \in pre(j)\}$

Shortest path from node s: Dijkstra's algorithm
    input: $pre(j)$: list of 1..$n$; $w[i,j]$: positive int for all $i \in pre(j)$
    init: $G[j] := 0$;
    always: $parent[j,i] = (i \in pre(j)) \wedge (G[j] \geq G[i] + w[i,j])$;
        $fixed[j] = (j = s) \vee (\exists i : parent[j,i] \wedge fixed[i])$
        $H = \{(G[i] + w[i,k]) | (i \in pre(k)) \wedge fixed(i) \wedge \neg fixed(k)\}$;

    forbidden: $\neg fixed[j]$
        advance: $(G[j] \geq \min H)$

Man-optimal stable marriage: Gale Shapley algorithm
    input: $mpref[i,k]$: int for all $i,k$; $rank[k][i]$: int for all $k,i$;
    init: $G[j] := 1$;
    always: $z = mpref[j][G[j]]$;

    forbidden: $(\exists i : \exists k \leq G[i] : (z = mpref[i][k]) \wedge (rank[z][i] < rank[z][j]))$
        advance: $G[j] := G[j] + 1$;

Market Clearing Prices: Demange Gale Sotomayor algorithm
    input: $v[b,i]$: int for all $b,i$
    init: $G[j] := 0$;
    always: $E = \{(k,b) \mid \forall i : (v[b,k] - G[k]) \geq (v[b,i] - G[i])$;
        $demand(U') = \{k \mid \exists b \in U' : (k,b) \in E\}$;
        $overDemanded(J) \equiv \exists U' \subseteq U : (demand(U') = J) \wedge (|J| < |U'|)$

    forbidden: $(\exists minimal J : OverDemanded(J) \wedge (j \in J)$
        advance: $G[j] := G[j] + 1$;
```

Figure 2: Example of Programs with LLP Predicates

```
job-scheduling:
    mode: sequential nondeterministic;
    var $count[j]$: int initially $pre(j).size()$;
    always $dep(j) := \{i | j \in pre(i)\}$;
    On ensure enabled if $(count[j] = 0)$
        for $(i \in dep(j))$: $count[i] := count[i] - 1$;

shortest path from node $s$: Bellman-Ford
    mode: parallel tasks;
    worklist ordered by $G[j]$;
    on ensure:
        for $(i \in dep(j))$: $worklist.add(i)$;

Shortest path from node s: Dijkstra's algorithm
    mode: parallel threads;
    worklist ordered by $G[j]$;

Man-optimal stable marriage: Gale Shapley algorithm
    mode: distributed push;

Market Clearing Prices: Demange Gale Sotomayor algorithm
    mode: parallel threads;
    worklist ordered by $j$;
```

Figure 3: Controller Component of Programs with LLP Predicates

equal to *expr*, there is no guarantee that $G[j]$ continues to be equal to *expr* — the value of *expr* may change because of changes in other components. We use *ensure* statement whenever *expr* is a monotonic function of G and therefore the predicate is lattice-linear. For some examples such as the stable marriage problem, the desirable predicate is not of the form $(G[j] \geq expr)$ or $(G[j] \leq expr)$. In this case, we specify the forbidden predicate and the corresponding advance statement. The forbidden section gives the predicate under which thread j is forbidden and the *advance* statement gives the action that must be executed whenever the forbidden predicate is true. For the stable marriage problem, the action is to increment $G[j]$ (i.e., proposing to the next woman in his list).

We now explain the solution of various problems in the LLP system shown in Fig. 2. We have only shown the core component of the programs. The core component is sufficient for the correctness. These programs can be executed in sequential, parallel or distributed mode depending upon the controller specified later.

1. *Job Scheduling*: This example illustrates specification of computation over acyclic graphs. We have earlier shown that the ensure predicate is lattice-linear. The program returns the least G that satisfies the predicate. Any j that does not satisfy *ensure*(j) triggers an action that updates $G[j]$ with the expression on the right hand side.

2. *Shortest Path Problem: Bellman-Ford*: For this problem our goal is to maximize $G[j]$ subject to constraints that $G[j]$ is less than or equal to $G[i] + w[i, j]$ for all $i \in pre(j)$. The variable $G[i]$ is initialized to ∞ for all indices except for the source vertex which is initialized to 0.

Since the predicate is lattice-linear, the program returns the optimal cost vector.

3. *Shortest Path Problem: Dijkstra's Algorithm (a slight variant)* :
   We use $G[j]$ as a lower bound on the cost of reaching j initialized to 0 for all vertices j. For simplicity, we assume that all edge weights are positive and all nodes are reachable from the source vertex. The key property of the lower bound is that to reach j for any vertex other than the source vertex, it must come through a vertex which has a lower cost. Thus, $G[j] \geq \min\{G[i] + w[i,j] \mid i \in pre(j)\}$. We define $parent[j,i]$ to be true if $G[j]$ is bigger than $G[i] + w[i,j]$ for some i in $pre(j)$ (a node may have multiple parents). We define $fixed[j]$ to be true if a path to vertex j has been found with cost equal to $G[j]$ and therefore the value of $G[j]$ cannot change. Initially, only the source node is fixed. If a node j has a parent i that is fixed, then node j is also fixed. Our goal is to advance G to fix as many nodes as possible. We keep H as the set of all costs that go from fixed nodes to non-fixed nodes. If any node j is not fixed, then it is safe to advance $G[j]$ to the minimum value in H. It is easy to show that the predicate $\forall j : fixed[j]$ is a lattice-linear predicate.

   The LLP algorithm is a slight variant of Dijkstra's algorithm because Dijkstra's algorithm fixes exactly one node whenever it removes a node from the priority queue; whereas, multiple nodes may get fixed in the LLP algorithm [Gar18b].

4. *Stable Marriage Problem: Gale-Shapley Algorithm (a slight variant)*:
   For this problem, we are given as input n men and n women. We are also given a list of men preferences as $mpref$ where $mpref[i][k]$ denotes k^{th} top choice of man i. The women preferences are more convenient to express as a $rank$ array where $rank[i][j]$ is the rank of man j by woman i. A matching between man and woman is stable if there is no *blocking pair*, i.e., a pair of woman and man such that they are not matched and prefer each other to their spouses. We let $G[i]$ be the choice number that man i has proposed to. Initially, $G[i]$ is 1 for all men. In the *always* section, we define some convenient notation. The variable z is defined to be the woman that corresponds to choice $G[j]$ for man j. Now, we can define j to be forbidden if there exists a man i such that z prefers man i to man j and man i prefers z to his current choice, i.e., man i and woman z would form a blocking pair for G. If man j is forbidden, it is clear that any vector in which man j is matched with z and man i is matched with his current or a worse choice can never be a stable marriage. Thus, it is safe for man j to advance to the next choice.

   The LLP algorithm is a slight generalization of Gale-Shapley algorithm. Instead of starting from the top choice, G can be initialized to any vector of choices. So long as there exists a stable marriage greater than or equal to that vector the LLP algorithm will find that stable marriage.

5. *Market Clearing Price Problem: Demange-Gale-Sotomayor Algorithm*

   Let I be a set of indivisible n items, and U, a set of n bidders. Every item $i \in I$ is given a *valuation* $v[b,i]$ by each bidder $b \in U$. The valuation of any item i is a number between 0 and $T[i]$. Each item i is given a price $G[i]$ which is also a number between 0 and $T[i]$. The goal is to assign items to the bidders to maximize the total payoff. This problem is simply a reformulation of the weighted bipartite matching problem.

   Given a price vector G, we define the bipartite graph $(I, U, E(G))$ as

$$(k,b) \in E(G) \equiv \forall i : (v[b,k] - G[k]) \geq (v[b,i] - G[i]).$$

An edge exists between item i and bidder b if the payoff for the bidder (the bid minus the price) is maximized with that item. A price vector G is a *market clearing price*, denoted by $B_{clearingPrice}(G)$ if the bipartite graph $(I, U, E(G))$ has a perfect matching. It can be easily shown that this predicate is lattice-linear. If there is no perfect matching, then there must be a minimal set of items that is overdemanded. Any item in the overdemanded set is forbidden and its price must be incremented. Note that in Fig. 2, we have given a brute force implementation of checking for overdemanded sets for simplicity. There exists efficient polynomial algorithms to compute minimal overdemanded sets (for example, by using an augmenting BFS algorithm).

Many of the problems discussed in this proposal such as the shortest path, the stable marriage, and the weighted bipartite problems can also be modeled using linear programs. However, applying linear programming algorithms such as simplex or interior point methods do not yield the special case combinatorial algorithms that are generally more efficient than the general algorithm for Linear Programming.

The LLP algorithm has many useful properties which are applicable to all the problems. First, it is non-deterministic. Given a global state G, there may be multiple indices j for which $G[j]$ is forbidden. The LLP algorithm is correct irrespective of the order in which these indices are updated. The efficiency of the algorithms may differ depending upon the order in which these indices are updated, but the correctness is independent of the order. Second, it allows *parallel* evaluation of forbidden predicates. Suppose that G is shared among different threads such that thread j is responsible for evaluating $forbidden(G, j)$. While this thread is evaluating this predicate other threads may have advanced on other indices, i.e., thread j may have old information of $G[i]$ for $i \neq j$. However, this would still keep the algorithm correct. For example, in the stable marriage problem, men can propose to women in parallel. In the shortest path algorithm, multiple vertices can update the estimate of their lower bounds and their parents in parallel. Third, it is *online*. There is no lookahead required for evaluation of the forbidden predicate. The LLP algorithm determines whether an index j is forbidden depending upon only the current global state G. This means that these algorithms are applicable in online settings where the future part of the lattice is revealed only when a forbidden index needs to advance. For example, in the stable marriage problem, when we are computing the man-optimal stable marriage, a man may not reveal his preference list. Only when he is rejected (his state is forbidden), he needs to advance on his choices and therefore reveal the next woman on his list.

4 Proposed Work: The Controller Component

In this section, we describe specification of the controller component for a LLP program (shown in Fig. 3). The core component of a LLP program can be run in many modes as specified by the controller.

Sequential mode: In this mode, the program can be either run in a deterministic or non-determinstic submodes. There are multiple reasons to make the program deterministic. First, some deterministic order may be more efficient than others. The programmer can specify this order in the controller component. Another reason to make the program deterministic is to make it easier for debugging purposes. If the programmer does not specify any deterministic strategy, the system can choose one. For example, the system may choose the least index j which violates the ensure predicate. If multiple predicates are violated by index j, then by ordering all the predicates one can make the schedule deterministic. The other strategy is to make the system replayable by logging the index and the predicate for every corrective action.

In the non-deterministic mode, the system chooses any execution. Although it will not be the focus of this project, the system may use performance data of the execution to choose the next enabled event. For example, if some forbidden events are faster to execute then they may be given preference. This analysis can be carried out at the compile time or the runtime.

The job scheduling problem illustrates the sequential mode. For this example, the core program is correct but inefficient because a component $G[j]$ may be updated multiple times. The controller component makes the program efficient by enabling updates on only those $G[j]$ for which all the prerequisites have been updated. This strategy is implemented by keeping $count[j]$ for all components j. The count keeps track of the number of prerequisite jobs for which completion times have been updated. With this strategy, it is easy to show that every $G[j]$ is updated exactly once. Since there may be multiple j for which $count[j]$ is zero and they are enabled, the program is still nondeterministic. By changing the keyword from nondeterministic to deterministic, the system can make the execution deterministic by always choosing the least j of all j for which $count[j]$ equals zero. The job scheduling problem is an example where the value of a node is updated based on the value of all prerequisite nodes. This example can be generalized to solve most dynamic programming problems where recurrence can be set up in an acyclic fashion.

Parallel mode: In this mode, the program can be run in a thread submode or a task submode. In the thread submode, the components of G are partitioned across a static pool of threads. The user may choose to specify how the partition is carried out. In the task submode, each forbidden event is viewed as a task. These tasks are maintained in a worklist ordered by the user specified partial order. A thread removes any minimal task from the worklist and executes it.

The shortest path problem with Bellman-Ford illustrates the parallel task mode. Each j such that $forbidden[j]$ holds is part of a shared worklist between threads. Any idle task can extract an ensure task from the worklist and run it. The worklist is updated whenever any component j is updated. The shortest path problem with Dijkstra's algorithm illustrates the parallel thread mode. The components of G are partitioned across threads. A thread responsible for component $G[j]$ updates it whenever it is not fixed. The other data structures such as H, $fixed$ and $parent$ also get updated. The market clearing price problem uses the parallel thread mode in which the prices of lower numbered items are increased first.

Distributed mode: In this mode, the global state vector G is distributed across multiple nodes. Whenever a process needs the most recent state from other processes, it can *pull* the value. Alternatively, whenever G changes, the node can *push* the value to all nodes specified in the dependency list.

The stable marriage problem illustrates the distributed mode. Each man i keeps its local state $G[i]$ and estimate of $G[j]$ for any other man j. Whenever $G[i]$ is updated it is sent to all other nodes who can then determine if they need to advance. This algorithm ends up sending $O(n)$ messages per update. By additional optimizations specific to the stable marriage problem. For example, by making man i also responsible for woman i, and sending the update message only to the man who is responsible for the current choice of man j, the message complexity can be reduced to $O(1)$ per update. Such optimizations are part of the proposed work.

For some problems, a natural definition of feasibility predicate may not be lattice-linear. For example, consider the problem of finding minimum spanning tree of a graph. One approach would be to define the feasibility predicate as the set of edges that form a spanning tree. However, the meet of two spanning trees may not be a spanning tree. However, these problems can still be solved using LLP system as follows. We first reformulate the problem such that there is a unique solution. If all edge weights are unique, then there is a unique minimum spanning tree. Any singleton element of a distributive lattice is trivially closed under meets (and joins). Thus, we can define the feasibility predicate in such a manner such that only the minimum spanning tree satisfies that predicate. With these observations, it is easy to implement other algorithms using LLP system such as Prim's algorithm for the minimum spanning tree problem, and Gale's top trading cycle algorithm for the housing allocation problem.

5 Proposed Work: Non-LLP Predicates

There are many problems for which the desired solution may not be unique or closed under meets. Many of these problems can still be solved efficiently in a sequential manner. We will explore techniques for efficient parallelization of predicates for these problems. Some examples of classes of these problems are:

- *Sequential Greedy Algorithms*: There are many problems that are solved by casting the problem as a sequence of decisions. At each decision, the algorithm takes the choice that minimizes the objective function at that point (while maintaining feasibility of the solution). For example, Kruskal's algorithm for the minimum spanning tree problem maintains a forest as a feasible solution. It chooses an edge one at a time such that the next edge is of minimum weight that does not create cycle with existing chosen edges. More generally, matroids and greedoids are structures that allow efficient greedy algorithms. We propose to investigate techniques for efficient parallel and distributed implementation of matroids and greedoids.

- *Local Exchange Algorithms*: There are many problems that start with a feasible solution and move to a solution with improved objective function by local swaps or exchanges whenever possible. The algorithm may not result in efficient algorithm if there can be a long chain of tiny improvements. Furthermore, it may be difficult to come up with generic efficient parallel and distributed implementation.

- *Marching towards Optimality Algorithms*: For many problems whenever a feasible solution is not optimal, there exists an augmentation of the current solution to increase the objective function. For example, for the maximum cardinality bipartite matching problem, if the given matching is not of maximum cardinality then there exists an augmenting path. For max-flow problems, if the given flow is not maximum, then there exists a path from the source to the destination vertex in the residual flow graph (Ford-Fulkerson's algorithm). For min cost max flow problem, if the current flow does not have min cost, then there exists a negative cost cycle (Klein's cycle canceling algorithm). All these problems can be solved by formulating an appropriate predicate and an advance function. However, these predicates are not closed under meets and independent updates of various threads is not possible. We propose to investigate efficient techniques for parallelization and distribution for these problem.

References

[Bel58] Richard Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90, 1958.

[BG13] Bharath Balasubramanian and Vijay K. Garg. Fault tolerance in distributed systems using fused data structures. *IEEE Trans. Parallel Distrib. Syst.*, 24(4):701–715, 2013.

[BG14] Bharath Balasubramanian and Vijay K. Garg. Fault tolerance in distributed systems using fused state machines. *Distributed Computing*, 27(4):287–311, 2014.

[CBDGF95] B. Charron-Bost, C. Delporte-Gallet, and H. Fauconnier. Local and temporal predicates in distributed systems. *ACM Trans. on Programming Languages and Systems*, 17(1):157–179, January 1995.

[CG98a] C. Chase and V. K. Garg. Detection of Global Predicates: Techniques and their Limitations. *Distributed Computing*, 11(4):191–201, 1998.

[CG98b] Craig M Chase and Vijay K Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11(4):191–201, 1998.

[CG13] Himanshu Chauhan and Vijay K. Garg. Democratic elections in faulty distributed systems. In Frey et al. [FRS$^+$13], pages 176–191.

[CG15a] Yen-Jung Chang and Vijay K. Garg. A parallel algorithm for global states enumeration in concurrent systems. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '15*, 2015.

[CG15b] Yen-Jung Chang and Vijay K. Garg. Quicklex: A fast algorithm for consistent global states enumeration of distributed computations. In *Proc. of the 19th International Conference on Principles of Distributed Systems*. Springer-Verlag, 2015.

[CG15c] Himanshu Chauhan and Vijay K. Garg. Necessary and sufficient conditions on partial orders for modeling concurrent computations. In Sajal K. Das, Dilip Krishnaswamy, Santonu Karkar, Amos Korman, Mohan Kumar, Marius Portmann, and Srikanth Sastry, editors, *Proceedings of the 2015 International Conference on Distributed Computing and Networking, ICDCN 2015, Goa, India, January 4-7, 2015*, pages 34:1–34:10. ACM, 2015.

[CG16] Yen-Jung Chang and Vijay K. Garg. Predicate detection for parallel computations with locking constraints. In Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone, editors, *20th International Conference on Principles of Distributed Systems, OPODIS 2016, December 13-16, 2016, Madrid, Spain*, volume 70 of *LIPIcs*, pages 17:1–17:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[CG17a] Himanshu Chauhan and Vijay K. Garg. Space efficient breadth-first and level traversals of consistent global states of parallel programs. In Shuvendu K. Lahiri and Giles Reger, editors, *Runtime Verification - 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings*, volume 10548 of *Lecture Notes in Computer Science*, pages 138–154. Springer, 2017. **Best Paper Award**.

[CG17b] Himnashu Chauhan and V. K. Garg. Fast enumeration of counting and stable predicates. In *Proc. of the 21st International Conference on Principles of Distributed Systems*. Springer-Verlag, 2017.

[CGNM13] Himanshu Chauhan, Vijay K. Garg, Aravind Natarajan, and Neeraj Mittal. A distributed abstraction algorithm for online predicate detection. In *IEEE 32nd Symposium on Reliable Distributed Systems, SRDS 2013, Braga, Portugal, 1-3 October 2013*, pages 101–110. IEEE Computer Society, 2013.

[CM88] K Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation Addison-Wesley*, volume 59. 1988.

[CM91] R. Cooper and K. Marzullo. Consistent detection of global predicates. pages 163–173, Santa Cruz, CA, May 1991.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[DGS86] Gabrielle Demange, David Gale, and Marilda Sotomayor. Multi-item auctions. *Journal of Political Economy*, 94(4):863–872, 1986.

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, Dec 1959.

[Dij78] Edsger W Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. In *Programming Methodology*, pages 166–175. Springer, 1978.

[DP90] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, UK, 1990.

[For56] L. A. Ford. Network flow theory. Technical report, 1956.

[FRS$^+$13] Davide Frey, Michel Raynal, Saswati Sarkar, Rudrapatna K. Shyamasundar, and Prasun Sinha, editors. *Distributed Computing and Networking, 14th International Conference, ICDCN 2013, Mumbai, India, January 3-6, 2013. Proceedings*, volume 7730 of *Lecture Notes in Computer Science*. Springer, 2013.

[FVGDS18] Kostas Ferles, Jacob Van Geffen, Isil Dillig, and Yannis Smaragdakis. Symbolic reasoning for automatic signal placement. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 120–134, New York, NY, USA, 2018. ACM.

[GAO14] Vijay K. Garg, Anurag Agarwal, and Vinit A. Ogale. Modeling, analyzing and slicing periodic distributed computations. *Inf. Comput.*, 234:26–43, 2014.

[Gar02] V. K. Garg. *Elements of Distributed Computing*. Wiley & Sons, 2002.

[Gar04] V. K. Garg. *Concurrent and Distributed Computing in Java*. Wiley & Sons, 2004.

[Gar13] Vijay K. Garg. Maximal antichain lattice algorithms for distributed computations. In Frey et al. [FRS$^+$13], pages 240–254.

[Gar15] Vijay K. Garg. *Introduction to Lattice Theory with Computer Science Applications.* Wiley and Sons, 2015.

[Gar17] Vijay K. Garg. Brief announcement: Application of predicate detection to the stable marriage problem. In *Distributed Computing - 31st International Symposium, DISC 2017 Vienna, Austria, October 12-16, 2017, Proceedings*, 2017. to appear.

[Gar18a] Vijay K. Garg. Applying predicate detection to the constrained optimization problems. *CoRR*, abs/1812.10431, 2018.

[Gar18b] Vijay K. Garg. Removing sequential bottleneck of dijkstra's algorithm for the shortest path problem. *CoRR*, abs/1812.10499, 2018.

[GC95] V. K. Garg and C. Chase. Distributed algorithms for detecting conjunctive predicates. In *Proc. of the IEEE Intnatl. Conf. on Distributed Computing Systems*, pages 423–430, Vancouver, Canada, June 1995.

[GG17] V. K. Garg and R. Garg. Parallel algorithms for predicate detection. Technical report, ECE Department, The University of Texas at Austin, 2017.

[GG19] Vijay K. Garg and Rohan Garg. Parallel algorithms for predicate detection. In R. C. Hansdah, Dilip Krishnaswamy, and Nitin Vaidya, editors, *Proceedings of the 20th International Conference on Distributed Computing and Networking, ICDCN 2019, Bangalore, India, January 04-07, 2019*, pages 51–60. ACM, 2019.

[GM01] V. K. Garg and N. Mittal. On slicing a distributed computation. In *21st Intnatl. Conf. on Distributed Computing Systems (ICDCS' 01)*, pages 322–329, Washington - Brussels - Tokyo, April 2001. IEEE.

[GMS03] V. K. Garg, N. Mittal, and A. Sen. Applications of lattice theory to distributed computing. *ACM SIGACT Notes*, 34(3):40–61, September 2003.

[GS62] David Gale and Lloyd S Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.

[GW91] V. K. Garg and B. Waldecker. Detection of unstable predicates. In *Proc. of the Workshop on Parallel and Distributed Debugging*, Santa Cruz, CA, May 1991.

[GW92] V. K. Garg and B. Waldecker. Detection of unstable predicates in distributed programs. pages 253–264. Springer Verlag, December 1992. Lecture Notes in Computer Science 652.

[HCG15] Wei-Lun Hung, Himanshu Chauhan, and Vijay K. Garg. Activemonitor: Asynchronous monitor framework for scalability and multi-object synchronization. In *Proc. of the International Conference on Principles of Distributed Systems (OPODIS)*, 2015.

[HG13] Wei-Lun Hung and Vijay K. Garg. Autosynch: an automatic-signal monitor based on predicate tagging. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 253–262. ACM, 2013.

[Hil03] Ernest Friedman Hill. *Jess in Action: Java Rule-Based Systems*. Manning Publications Co., Greenwich, CT, USA, 2003.

[Hoa78] Charles Antony Richard Hoare. Communicating sequential processes. In *The origin of concurrent programming*, pages 413–443. Springer, 1978.

[MB15] Menna Mostafa and Borzoo Bonakdarpour. Decentralized runtime verification of ltl specifications in distributed systems. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 494–503. IEEE, 2015.

[MBFJ16] Ramy Medhat, Borzoo Bonakdarpour, Sebastian Fischmeister, and Yogi Joshi. Accelerated runtime verification of ltl specifications with counting semantics. In *International Conference on Runtime Verification*, pages 251–267. Springer, 2016.

[MFP06] Gero Mühl, Ludger Fiege, and Peter Pietzuch. *Distributed event-based systems*. Springer Science & Business Media, 2006.

[MHVG15] Hammurabi Mendes, Maurice Herlihy, Nitin H. Vaidya, and Vijay K. Garg. Multidimensional agreement in byzantine systems. *Distributed Computing*, 28(6):423–441, 2015.

[Mun57] James Munkres. Algorithms for the assignment and transportation problems. *Journal of the society for industrial and applied mathematics*, 5(1):32–38, 1957.

[NMG14] Aravind Natarajan, Neeraj Mittal, and Vijay K. Garg. Online algorithms to generate slices for regular temporal logic predicates. In Mainak Chatterjee, Jian-Nong Cao, Kishore Kothapalli, and Sergio Rajsbaum, editors, *Distributed Computing and Networking - 15th International Conference, ICDCN 2014, Coimbatore, India, January 4-7, 2014. Proceedings*, volume 8314 of *Lecture Notes in Computer Science*, pages 165–180. Springer, 2014.

[SS74] Lloyd Shapley and Herbert Scarf. On cores and indivisibility. *Journal of mathematical economics*, 1(1):23–37, 1974.

[SSA$^+$18] César Sánchez, Gerardo Schneider, Wolfgang Ahrendt, Ezio Bartocci, Domenico Bianculli, Christian Colombo, Yliés Falcone, Adrian Francalanza, Srdan Krstic, JoHao M. Lourenço, Dejan Nickovic, Gordon J. Pace, Jose Rufino, Julien Signoles, Dmitriy Traytel, and Alexander Weiss. A survey of challenges for runtime verification from advanced application domains (beyond software), 2018.

[TVK18] Vidhya Tekken Valapil and Sandeep Kulkarni. Biased clocks: A novel approach to improve the ability to perform predicate detection with o(1) clocks. In Zvi Lotker and Boaz Patt-Shamir, editors, *Structural Information and Communication Complexity*, pages 345–360, Cham, 2018. Springer International Publishing.

[VG13] Nitin H. Vaidya and Vijay K. Garg. Byzantine vector consensus in complete graphs. In Panagiota Fatourou and Gadi Taubenfeld, editors, *PODC*, pages 65–73. ACM, 2013.

[ZG19] Xiong Zheng and Vijay K. Garg. An optimal vector clock algorithm for multithreaded systems. In *Proc. of the International Conference on Distributed Computing Systems*. IEEE, 2019.

[ZHG18] Xiong Zheng, Changyong Hu, and Vijay K. Garg. Lattice agreement in message passing systems. In Ulrich Schmid and Josef Widder, editors, *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, volume 121 of *LIPIcs*, pages 41:1–41:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.