Invited Paper: A Lattice Linear Predicate Parallel Algorithm for the Housing Market Problem*

Vijay K. Garg[1][0000−0002−5797−4389]

Department of Electrical and Computer Engineering
University of Texas at Austin,
Austin, TX 78712
`garg@ece.utexas.edu`

Abstract. It has been shown that Lattice Linear Predicate (LLP) algorithm solves many combinatorial optimization problems such as the shortest path problem, the stable marriage problem and the market clearing price problem. In this paper, we give an LLP algorithm for the Housing Market problem. The Housing Market problem is a one-sided matching problem with n agents and n houses. Each agent has an initial allocation of a house and a totally ordered preference list of houses. The goal is to find a matching between agents and houses such that no strict subset of agents can improve their outcome by exchanging houses with each other rather than going with the matching. Gale's celebrated Top Trading Cycle algorithm to find the matching requires $O(n^2)$ time. Our parallel algorithm has expected time complexity $O(n \log^2 n)$ with and expected work complexity of $O(n^2 \log n)$.

1 Introduction

The housing market problem proposed by Shapley and Scarf [1] is a matching problem with one-sided preferences. There are n agents and n houses. Each agent a_i initially owns a house h_i for $i \in \{1, n\}$ and has a completely ranked list of houses. There are variations of this problem when the agents do not own any house initially. In this paper, we focus on the version with the initial endowment of houses for the agents. The list of preferences of the agents is given by $pref[i][k]$ which specifies the k^{th} preference of the agent i. Thus, $pref[i][1] = j$ means that a_i prefers h_j as his top choice. The goal is to come up with an optimal house allocation such that each agent has a house and no subset of agents can improve the satisfaction of agents in this subset by exchanging houses within the subset. It can be shown that there is a unique such matching called the *core* for any housing market. The standard algorithm for this problem is Gale's Top Trading Cycle Algorithm that takes $O(n^2)$ time. This algorithm is optimal in terms of

---

* Supported by the NSF Grant CCR-1812351 and Cullen Trust Endowed Professorship.

the time complexity since the input size is $O(n^2)$. Our interest in this paper is to design parallel algorithms for this problem.

The housing market problem has been studied by many researchers [1–8]. Possible applications of the housing market problem include: assigning virtual machines to servers in cloud computers, allocating graduates to trainee positions, professors to offices, and students to roommates. In this paper, we apply the Lattice Linear Predicate (LLP) method [9] to give a parallel algorithm for the housing market problem. This problem has also recently been studied by Zheng and Garg [10] where it is shown that the problem of verifying that a matching is a core is in NC, but the problem of computing the core is CC-hard[1] The paper [10] also gives a *distributed* message-passing algorithm to find the core with $O(n^2)$ messages. In this paper, we focus on computing the core and give a parallel algorithm for finding the core that is nearly linear in the number of agents. Our algorithm takes expected $O(n \log^2 n)$ time and expected $O(n^2 \log n)$ work.

Another goal of this paper is to show applications of the Lattice Linear Predicate (LLP) algorithm for the problem. It has been shown that the Lattice Linear Predicate (LLP) algorithm solves many combinatorial optimization problems such as the shortest path problem, the stable marriage problem and the market clearing price problem [9]. In [11], we show that the LLP algorithm also solves many dynamic programming problems in parallel. These problems include the longest subsequence problem, the optimal binary search tree problem, and the knapsack problem.

The lattice-linear predicate detection method to solve a combinatorial optimization problem is as follows. The first step is to define a lattice of vectors L such that each vector is *assigned* a point in the search space. For the stable matching problem, the vector corresponds to the assignment of men to women (or equivalently, the choice number for each man). For the shortest path problem, the vector assigns a cost to each node. For the housing problem studied in this paper, the vector corresponds to the assignment of agents to houses. The comparison operation ($\leq$) is defined on the set of vectors such that the least vector, if feasible, is the extremal solution of interest. For example, in the stable marriage problem if each man orders women according to his preferences and every man is assigned the first woman in the list, then this solution is the man-optimal solution whenever the assignment is a matching and has no blocking pair. Similarly, in the shortest path problem, the zero vector would be optimal if it were feasible. For the housing problem, each agent orders the list of houses in order of its preference giving us the comparison operator. For two vectors G and H in the lattice, $G \leq H$ if and only if each agent prefers the house assigned to them in G at least as much as the house assigned to them in H.

The second step in our method is to define a boolean predicate B that models the feasibility of the vector. For the stable matching problem, an assignment is feasible iff it is a matching and there is no blocking pair. For the shortest path

---

[1] The class CC (Comparator Circuits) is the complexity class containing decision problems which can be solved by comparator circuits of polynomial size.

problem, the vector G only gives the lower bound on the cost of a path and there may not be any path to vertex v_i with cost $G[i]$. To capture that an assignment is feasible, we define *feasibility* which requires the notion of a *parent*. We say that v_i is a parent of v_j in G iff there is a direct edge from v_i to v_j and $G[j]$ is at least $(G[i] + w[i, j])$. For the shortest path problem, an assignment is feasible iff every reachable node except the source node has a parent. For the housing problem, we say that a housing assignment is feasible if no subset of agents can improve the satisfaction of agents in this subset by exchanging houses within the subset. Fig. 1 gives the feasibility predicate for each of these problems.

Problem	**Feasibility Predicate** B
Shortest Path	every reachable vertex other than the source has a parent
Stable Marriage	the assignment is a matching and there is no blocking pair
Housing market	the assignment is a matching and there is no break away coalition

Fig. 1. The Feasibility Predicate for Various Problems

The third step is to show that the feasibility predicate is a lattice-linear predicate [12, 9]. Lattice-linearity property allows one to search for a feasible solution efficiently. If any point in the search space is not feasible, it allows one to make progress towards the optimal feasible solution without any need for exploring multiple paths in the lattice. Moreover, multiple processes can make progress towards a feasible solution in a *parallel* fashion. In a finite distributive lattice, it is clear that the maximum number of such advancement steps before one finds the optimal solution or reaches the top element of the lattice is equal to the height of the lattice. In this paper, we derive a parallel LLP algorithm that solves the housing market problem using this approach.

This paper is organized as follows. Section 2 gives background on Gale's Top Trading Cycle Algorithm and the LLP method. Section 3 applies LLP method to the unconstrained Housing market problem and derives a high-level parallel algorithm. Section 4 gives a parallel Las Vegas algorithm for the Housing market problem.

2 Background

In this section, we cover the background information on Gale's Top Trading Cycle Algorithm and the LLP Algorithm [9]. Consider the housing market instance shown in Fig. 2. There are four agents a_1, a_2, a_3 and a_4. Initially, the agent a_i holds the house h_i. The preferences of the agents is shown in Fig. 2.

2.1 Gale's Top Trading Cycle (TTC) Algorithm for Housing Market

The Top Trading Cycle (TTC) algorithm attributed to Gale by Shapley and Scarf [1] works in stages. At each stage, it has the following steps:

Agents' Preferences	Initial Allocation	Matching returned by the TTC algorithm
$a_1 : h_2, h_3, h_1, h_4$	$a_1 : h_1$	$a_1 : h_2$
$a_2 : h_1, h_4, h_2, h_3$	$a_2 : h_2$	$a_2 : h_1$
$a_3 : h_1, h_2, h_4, h_3$	$a_3 : h_3$	$a_3 : h_4$
$a_4 : h_2, h_1, h_3, h_4$	$a_4 : h_4$	$a_4 : h_3$

Fig. 2. Housing Market and the Matching returned by the Top Trading Cycle Algorithm

Step 1. We construct the *top choice* directed graph $G_t = (A, E)$ on the set of agents A as follows. We add a directed edge from agent $a_i \in A$ to agent $a_j \in A$ if a_j holds the current top house of a_i. Fig. 3 shows the directed graph at the first stage.

Step 2. Since each node has exactly one outgoing edge in G_t, there is at least one cycle in the graph (possibly, a self-loop). All cycles are node disjoint. We find all the cycles in the top trading graph and implement the trade indicated by the cycles, i.e, each agent which is in any cycle gets its current top house.

Step 3. Remove all agents which get their current top houses and remove all houses which are assigned to some agent from the preference list of remaining agents.

The above steps are repeated until each agent is assigned a house. At each stage, at least one agent is assigned a final house. Thus, this algorithm takes $O(n)$ stages in the worse case and needs $O(n^2)$ computational steps.

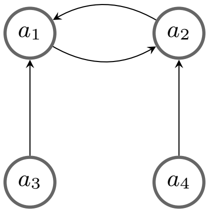

Fig. 3. The top choice graph at the first stage.

2.2 LLP Algorithm

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 lattice is used to model the search space of the combinatorial optimization problem. 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 parallel 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.

Fig. 4 shows a finite poset corresponding to n processes (n equals two in the figure), and the corresponding lattice of all eleven global states.

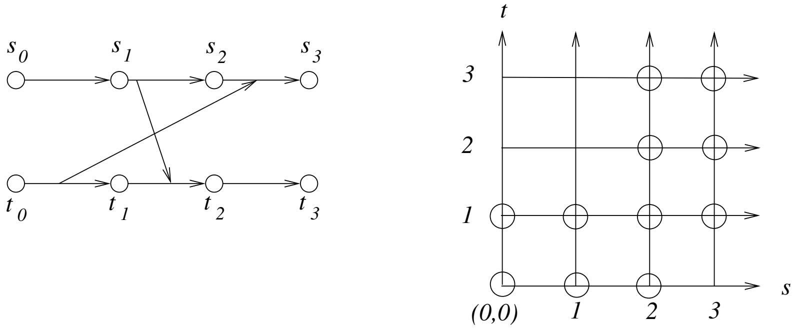

Fig. 4. A poset and its corresponding distributive lattice L

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. 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 $G[j]$ is *forbidden* (or equivalently, the index j is forbidden) if for any vector $H \in L$, where $G \leq H$, if $H[j]$ equals $G[j]$, then B is false for H. Informally, this means that any global state $H \geq G$ which satisfies B must be advanced on index j. Formally,

Definition 1 (Forbidden State [12]). *Given any distributive lattice L of n-dimensional vectors of $\mathbf{R}_{\geq 0}$, and a predicate B, we define* $\text{forbidden}(G, j, B) \equiv \forall H \in L : G \leq H : (G[j] = H[j]) \Rightarrow \neg B(H)$.

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 [12]). *A boolean predicate B is* lattice-linear *with respect to a lattice L iff* $\forall G \in L : \neg B(G) \Rightarrow (\exists j : \text{forbidden}(G, j, B))$.

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, α) 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 $\alpha(G, j, B)$ such that forbidden$(G, j, B, \alpha(G, j, B))$ holds whenever forbidden(G, j, B) is true. With the notion of $\alpha(G, j, B)$, we have the Algorithm LLP. 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 $\alpha(G, j, B)$.

ALGORITHM LLP: To find the minimum vector at most T that satisfies 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
```

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

Lemma 1. *[9, 12] 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) If B_1 and B_2 are lattice-linear then $B_1 \wedge B_2$ is also lattice-linear.

We now give an example of lattice-linear predicates for the 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, j, 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.

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.

2.3 Notation

We now go over the notation used in the description of our parallel algorithms. Fig. 5 shows a parallel algorithm for the job-scheduling problems.

The **var** section gives the variables of the problem. We have a single variable G in the example shown in Fig. 5. G is an array of objects such that $G[j]$ is the state of thread j for a parallel program.

The **input** section gives all the inputs to the problem. These inputs are constant in the program and do not change during execution.

The **init** section is used to initialize the state of the program. All the parts of the program apply to all values of j. For example, the *init* section of the job scheduling program in Fig. 5 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. We will show its use later.

The LLP algorithm gives the desirable predicate either by using the **forbidden** predicate or **ensure** predicate. The *forbidden* predicate has an associated *advance* clause that specifies how $G[j]$ must be advanced whenever the forbidden predicate is true. For many problems, it is more convenient to use the complement of the forbidden predicate. 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 $G[j]$ 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.

3 Applying LLP Algorithm to the Housing Market Problem

We model the housing market problem as that of predicate detection in a computation. There are n agents and n houses. Each agent proposes to houses

```
$P_j$: Code for thread $j$
// common declaration for all the programs below
var $G$: array[$1..n$] of $0..maxint$;// shared among all threads
input: $t[j]$ : $int$, $pre(j)$: list of $1..n$;
init: $G[j] := t[j]$;

job-scheduling:
    forbidden: $G[j] < \max\{G[i] + t[j] \mid i \in pre(j)\}$;
        advance: $G[j] := \max\{G[i] + t[j] \mid i \in pre(j)\}$;

job-scheduling:
    ensure: $G[j] \geq \max\{G[i] + t[j] \mid i \in pre(j)\}$;
```

Fig. 5. LLP Parallel Program for (a) job scheduling problem using forbidden predicate (b) job scheduling problem using ensure clause

in the decreasing order of preferences. These proposals are considered as events executed by n processes representing the agents. Thus, we have n events per process. Each event is labeled as (i, h, k), which corresponds to the agent i proposing to the house h as his choice number k.

The global state corresponds to the number of proposals made by each of the agents. Let $G[i]$ be the number of proposals made by the agent i. We will assume that in the initial state every agent has made his first proposal. Thus, the initial global state $G = [1, 1, .., 1]$. We extend the notation of indexing to subsets $J \subseteq [n]$ such that $G[J]$ corresponds to the subvector given by indices in J.

We now model the possibility of reallocation of houses based on any global state. Recall that $pref[i][k]$ specifies the k^{th} preference of the agent a_i. Let $wish(G, i)$ denote the house that is proposed by a_i in the global state G, i.e.,

$$wish(G, i) = pref[i][G[i]]$$

A global state G satisfies *matching* if every agent proposes a different house, i.e.,

$$matching(G) \equiv \forall i, j : i \neq j : wish(G, i) \neq wish(G, j).$$

We generalize *matching* to refer to a subset of agents rather than the entire set.

Definition 4 (submatching). *Let $J \subseteq [n]$. Then, submatching(G, J) iff wish(G, J) is a permutation of indices in J.*

Intuitively, if *submatching*(G, J) holds, then all agents in J can exchange houses within the subset J.

For any G, it is easy to show that

Lemma 2. *For all* G*, there always exists a nonempty* J *such that* $submatching(G, J)$*.*

Proof: Given any G, we can create a directed graph as follows. The set of vertices is agents and there is an edge from i to j if $wish(G, i) = j$. There is exactly one outgoing edge from any vertex in $[n]$ to $[n]$ in this graph. This implies that there is at least one cycle in this graph (possibly, a self-loop). The indices of agents in the cycle gives us such a subset J. ■

We now show that

Lemma 3. $submatching(G, J_1)$ *and* $submatching(G, J_2)$ *implies that* $submatching(G, J_1 \cup J_2)$*.*

Proof: Any index $i \in J_1 \cup J_2$ is mapped to J_1 if $i \in J_1$ and J_2, otherwise. ■

Hence, there exists the biggest submatching in G. Note that $matching(G)$ is equivalent to $submatching(G, [n])$.

Definition 5 (Feasible Global State). *A global state* G *is feasible for the housing market problem iff it is a matching and for all global states* $F < G$*, there does not exist any submatching which is better in* F *than in* G*. Note that if there exists a submatching* J *which is better in* F *than* G*, then the agents in* J *can improve their allocation by just exchanging houses within the subset* J*. Formally, let*

$$B_{housing}(G) \equiv matching(G) \wedge (\forall F < G : \forall J \subseteq [n] : submatching(F, J) \Rightarrow F[J] = G[J]).$$

We show that $B_{housing}(G)$ is a lattice-linear predicate. This result will let us use the lattice-linear predicate detection algorithm for the housing market problem.

Theorem 1. *The predicate* $B_{housing}(G)$ *is lattice-linear.*

Proof: Suppose that $\neg B_{housing}(G)$. This implies that either G is not a matching or it is a matching but there exists a smaller global state F that has a submatching better than G.

First, consider the case when G is not a matching. Let J be the largest set such that $submatching(G, J)$. Consider any index $i \notin J$ such that $wish(G, i) \in J$. We claim that $forbidden(G, i, B_{housing})$. Let H be any global state greater than G such that $G[i] = H[i]$. We consider two cases.

Case 1: $H[J] > G[J]$.
Then, from the second conjunct of $B_{housing}$, we know that $\neg B_{housing}(H)$ because $submatching(G, J)$ and $H[J] \neq G[J]$.

Case 2: $H[J] = G[J]$.
Since $wish(H, i) = wish(G, i)$, $wish(G, i) \in J$, and $G[J] = H[J]$, we get that H is not a matching because the house given by $wish(G, i)$ is also in the wish list of some agent in J.

Now consider the case when G is a matching but $\neg B_{housing}(G)$. This implies

$$\exists F < G : \exists J \subseteq [n] : submatching(F, J) \wedge F[J] < G[J]).$$

However, the same F will also result in guaranteeing $\neg B_{housing}(H)$ for any $H \geq G$. ■

It is also easy to see from the proof that if an index is part of a submatching, then it will never become forbidden.

This theorem gives us the algorithm shown in Fig. 6. Let G be the initial global state. Let $S(G)$ be the biggest submatching in G. All agents such that they are not in $S(G)$ and wish a house which are part of $S(G)$ are forbidden and can move to their next proposal. The algorithm terminates when no agent is forbidden. This algorithm is a parallel version of the top trading cycle (TTC) mechanism attributed to Gale in [1].

```
Algorithm Housing-Market:
var
    $G$: array[$1..n$] of int initially $1$;// every agent starts with the top choice
    $T = (n, n, ..., n)$; //maximum number of proposals at $a_i$
always
    $S(G)$ = largest $J$ such that $submatching(G, J)$
    $\text{forbidden}(G, j, B) \equiv (j \notin S(G)) \wedge (wish(G, j) \in S(G))$

while $\exists j$ : $\text{forbidden}(G, j, B)$ do
    for all $j$ such that $\text{forbidden}(G, j, B)$ in parallel:
        if $(G[j] = T[j])$ then return null;
        else $G[j] := G[j] + 1$;
endwhile;
return $G$; // the optimal solution
```

Fig. 6. A high-level parallel algorithm to find the optimal house market

We now show that

Theorem 2. *There exists at least one feasible global state G such that $B_{housing}(G)$.*

Proof: Every agent has his own house in the list of preferences. If he ever makes a proposal to his own house, he forms a submatching. That particular event is never forbidden because it is a part of a submatching. Hence, lattice-linear predicate detection algorithm will never mark that event as forbidden. Since such an event exists for all processes, we are guaranteed to never go beyond this global state. ■

The above proof also shows that agents can never be worse-off by participating in the algorithm. Each agent will either get his own house back or get a house that he prefers to his own house.

4 An Efficient Parallel Algorithm for the Housing Market Problem

We now present an efficient parallel algorithm for the housing market problem. We note here that [10] gives a distributed algorithm with $O(n^2)$ messages for the housing market problem. In this paper, we focus on computing the core and give a parallel algorithm for finding the core that is nearly linear in the number of agents. Our algorithm takes expected $O(n \log^2 n)$ time and expected $O(n^2 \log n)$ work.

By renumbering houses, if necessary, we assume that initially agent a_i has the house h_i. We assume that the preference list is provided as two data structures: $prefList$ and $prefPointer$. The variable $prefList$ is an array of doubly linked list such that $prefList[i]$ points to the list of preferences of agent i. As the algorithm executes, we advance on $prefList$ and the head of the $prefList[i]$ corresponds to the variable $wish$ for agent a_i in Fig. 6.

To facilitate the quick deletion of houses from this list, we also have a data structure $prefPointer$. The variable $prefPointer$ is a two dimensional array such that $prefPointer[i][j]$ points to the node corresponding to house h_j in the doubly-linked list of agent a_i. If at any stage in the algorithm, we find out that the house h_j has been permanently allocated to some other agent than a_i, then we need to remove the house h_j from the preference list of a_i. Since $prefPointer[i][j]$ points to that node in the doubly linked list $prefList[i]$, we can delete the house in $O(1)$ time. Due to these deletions, we maintain the invariant that the head of $prefList[i]$ always corresponds to the top choice of the agent a_i. Note that if the input is given as the two dimensional array $pref$, where $pref[i][j]$ is the top j^{th} choice for the agent a_i, then it can be converted into $prefList$ and $prefPointer$ in $O(n)$ time with $O(n)$ processors.

We keep the array $fixed$ such that $fixed[i]$ indicates that the agent i has been assigned its final house. If an agent i is fixed, then it can never be forbidden in Fig. 6. Once all agents are fixed, we get that no agent is forbidden and the algorithm terminates.

At every iteration, we keep the array $inCycle[i]$ that indicates agents that are in Top Trading Cycle at that iteration. In Fig. 6, these agents correspond to $S(G)$ in the global state G. Algorithm LLP-TTC uses a *while* loop to fix some number of agents in every iteration. At least one agent is fixed in every iteration, and therefore there are at most n iterations of the while loop.

Each iteration has four steps. In the first step, we initialize *inCycle* to be false by default. In the second step (function *markRoots*) we use symmetry breaking via randomization and pointer jumping to mark one node called *root* in every cycle as belonging to a cycle. The reader is referred to [13] for symmetry breaking and pointer jumping. During the process of pointer jumping, we also construct a tree rooted at a vertex such that it consists of all the nodes in the cycle. In the third step (function *informTree*), we inform all the agents that are in some rooted tree that they are in a cycle. In the fourth step, we fix all the agents that are in cycles and remove their houses from $prefList$. This step corresponds to advancing G in Fig. 6.

ALGORITHM LLP-TTC: Parallel LLP Top Trading Cycle Algorithm

```
// By renumbering houses, ensure that initially agent $a_i$ is assigned house $h_i$
var
    $prefList$:array[1..n] of list initially $\forall i : prefList[i]$ has preferences for $a_i$;
    $prefPointer$:array[1..n, 1..n] of pointer to the node in $prefList$;
    $fixed$: array[1..n] of booean initially $\forall i : fixed[i] = false$;
    $inCycle$: array[1..n] of booean initially $\forall i : inCycle[i] = false$;
    $children$: array[1..n] of set of nodes that $a_i$ traversed initially {};
while $(\exists i : \neg fixed[i])$
    // Step 1: initialize $inCycle$
    forall $i : \neg fixed[i]$ in parallel do: $inCycle[i] := false$;
    // Step 2: Mark one node in every cycle as the root
    markRoots();
    // Step 3: inform all the agents in any rooted tree that they are in a cycle
    informTree();
    // Step 4: Now delete all the agents that are in cycle
    forall $i : \neg fixed[i] \wedge inCycle[i]$, $j : \neg[fixed[j] \wedge \neg inCycle[j]]$ in parallel do
            delete the node $prefPointer[j][i]$ from the linked list $prefList[j]$;
    forall $i : \neg fixed[i] \wedge inCycle[i]$ in parallel do
        $fixed[i] := true$;
endwhile
return $prefList$; //$prefList[i]$ points to the house assigned to the agent $a_i$
```

The function *markRoots* uses variable *active* to denote agents that are active. Initially, all agents are active. The variable *succ*[*i*] is used to point to the next active agent. Initially, *succ*[*i*] points to the agent who has the top choice house of agent *i*. The variable *done*[*i*] indicates whether a cycle has been discovered in the subgraph that agent *i* is pointing to. Once, a cycle has been discovered then any active agent knows that it cannot be part of any cycle and it becomes inactive.

The function *markRoots* uses a *while* loop at line 5 to run while there is any active node. Every active agent flips a coin at line 7. If its own coin is a head and its successor gets a tail, then this agent becomes inactive at line 9. It is clear that two consecutive agents can never become inactive in the same round because we require an agent to get "head" and its successor to get "tail" to become inactive. It is also clear that the number of active agents is reduced by a constant fraction in every round of coin toss in expectation. Thus, the outer while loop at line 5 is executed expected $O(\log n)$ times.

If an agent is active, it traverses its *succ* pointer till it reaches the next *active* node. This is done using the *while* loop at line 11. This traversal has a length of one or zero because there cannot be two consecutive inactive agents due to the rule of becoming active.

If agent *i* reaches itself as the next active node at line 15, it marks *inCycle* to be true. It also sets *done*[*i*] to be true so that any active node *j* that points to *i* knows that a cycle has been found and that the node *j* can stop looking for the cycle. If the successor of the node is different, then we check if the successor is done. If the successor is done, then this node is not part of the cycle and can therefore make itself inactive and also mark itself as done. Since all agents execute the statements in *forall* in parallel, we get that the function *markRoots()* has parallel expected time complexity of $O(\log n)$. Also, for every cycle in the graph, there is exactly one node that sets its *inCycle* to be true.

The function *informTree* uses variable *rootSet* to initially include all the roots found in the function *markRoots*. Once all the nodes in any rooted tree have been informed, the root is deleted from the *rootSet*. To inform agents in the tree, we follow the usual method of broadcasting a value from the root to its children. To detect that all agents in the tree have been notified, we let any subtree that has finished informing its subtree to leave the tree by deleting itself from the children set of its parent. If the agent is a root, then it deletes itself from the *rootSet*. Once all roots have deleted themselves, the function terminates. Since the height of any tree is expected to be $O(\log n)$ and the number of children of any node is also $O(\log n)$, we get that the algorithm takes $O(\log^2 n)$ time.

ALGORITHM markRoots: Function markRoots for the Parallel LLP Top Trading Cycle Algorithm

```
function markRoots()
    succ: array[1..n] of 1..n initially $\forall i : succ[i] = prefList[i].head()$; //successor
  of $a_i$ which is active
    active: array[1..n] of booean initially $\forall i : active[i] = true$;
    done: array[1..n] of booean initially $\forall i : done[i] = false$;
    while ($\exists i : active[i]$)
        forall $i : \neg fixed[i] \wedge active[i]$ in parallel do
            $coin[i]$ := "head" or "tail" // based on the flip of a coin
            if ($coin[i]$ = "head") $\wedge$ ($coin[succ[i]]$ = "tail") then
                $active[i] := false$;
            else // node $i$ is active
                while $\neg active[succ[i]]$ do
                    $children[i] := children[i] \cup succ[i]$
                    $succ[i] := succ[succ[i]]$
                endwhile
                if ($succ[i] = i$) // found a cycle
                    $done[i] := true$
                    $inCycle[i] := true$
                    $active[i] := false$
                else if $done[succ[i]]$ then
                    $active[i] := false$
                    $done[i] := true$
        endforall
    endwhile
```

ALGORITHM informTree: Function informTree for the Parallel LLP Top Trading Cycle Algorithm

```
function informTree()
    $informed$: array[1..$n$] of booean initially $\forall i : informed[i] = false$;
    $parent$: array[1..$n$] of 1..$n$ initially $\forall i : parent[i] = i$;
    $rootSet$: set of 1..$n$ initially $\{i \mid inCycle[i]\}$
    while ($rootSet \neq \{\}$) do
        forall $i : \neg fixed[i]$ in parallel do
            if ($inCycle[i] \wedge \neg informed[i]$) then
                $informed[i] := true$
                for ($j \in children[i]$) do
                    $inCycle[j] := true$
                    $parent[j] := i$
                endfor
            if ($inCycle[i] \wedge informed[i] \wedge (children[i] = \{\})$) then
                if ($parent[i] = i$) then $rootSet.remove(i)$;
                else  $children[parent[i]] := children[parent[i]] - \{i\}$
        endforall
    endwhile
```

We first show the correctness of the parallel algorithm LLP-TTC.

Theorem 3. *The algorithm LLP-TTC returns the core of the housing market problem.*

Proof: It is sufficient to show that the algorithm LLP-TTC finds all top trading cycles in each iteration. Consider any top trading cycle of size 1 at node i. The function markRoot can never mark node i in the cycle as inactive due to the requirement of the coin turning at node i as head and its successor, itself, as tail. Furthermore, since $succ[i]$ equals i, node i is marked as $inCycle$. Now, consider any top trading cycle of size $k > 1$. Since we require the successor of the node to have a different toss to turn inactive, all nodes cannot turn inactive. The active nodes keep the inactive nodes following it as its children. After every coin toss, the length of the cycle for active nodes is expected to shrink by a constant factor. Hence, in expected $O(\log n)$ coin tosses, the cycle reduces to size 1 and the former case applies.

Now consider any node i that is not in any top trading cycle. Since our graph is functional (every vertex has out-degree exactly one), node i leads to a cycle by following the $succ$ edge. By previous discussion in $O(\log n)$ expected time, one of the nodes in that cycle, say j will set $inCycle[j]$ and $done[j]$ to be true. Since any path of active nodes reduces by a constant factor, in $O(\log n)$ expected time node i will point to a node that is $done$ and will also mark itself as $done$.

The function $informTree$ simply sets the variable $inCycle$ of all nodes in the cycle to be true. Finally, step 4 removes all houses and agents that are in any cycle and thus implements the top trading cycle mechanism. ∎

We now analyze the time and work complexity of LLP-TTC.

Theorem 4. *LLP-TTC takes expected $O(n \log^2 n)$ time and expected $O(n^2 \log n)$ work.*

Proof: Since every functional graph has at least one cycle, there exists at least one new node that finds itself in a cycle in every iteration of the while loop. Hence, there are at most n iterations of the while loop. In each iteration, Step 1 takes $O(1)$ time and $O(n)$ work. Step 2 takes expected $O(\log n)$ time and expected $O(n \log n)$ work. Step 3 takes $O(\log^2 n)$ time and $O(n)$ work. Let α_k be the number of agents that are fixed in the k^{th} iteration of the while loop. Step 4 takes $O(\alpha_k)$ time and $O(n\alpha_k)$ work. Adding up over all iterations, we get the desired time and work complexity. ■

5 Acknowledgments

I thank Changyong Hu, Robert Streit, and Xiong Zheng for various discussions on the housing allocation problem. I also thank the anonymous reviewers for comments on the paper.

References

1. Lloyd Shapley and Herbert Scarf. On cores and indivisibility. *Journal of mathematical economics*, 1(1):23–37, 1974.
2. Aanund Hylland and Richard Zeckhauser. The efficient allocation of individuals to positions. *Journal of Political economy*, 87(2):293–314, 1979.
3. Lin Zhou. On a conjecture by Gale about one-sided matching problems. *Journal of Economic Theory*, 52(1):123–135, 1990.
4. Atila Abdulkadiroğlu and Tayfun Sönmez. Random serial dictatorship and the core from random endowments in house allocation problems. *Econometrica*, 66(3):689–701, 1998.
5. Atila Abdulkadiroğlu and Tayfun Sönmez. House allocation with existing tenants. *Journal of Economic Theory*, 88(2):233–260, 1999.
6. Alvin E Roth and Andrew Postlewaite. Weak versus strong domination in a market with indivisible goods. *Journal of Mathematical Economics*, 4(2):131–137, 1977.
7. Alvin E Roth. Incentive compatibility in a market with indivisible goods. *Economics letters*, 9(2):127–132, 1982.
8. Manlove David. *Algorithmics of matching under preferences*, volume 2. World Scientific, 2013.
9. Vijay K. Garg. Predicate detection to solve combinatorial optimization problems. In Christian Scheideler and Michael Spear, editors, *SPAA '20: 32nd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, July 15-17, 2020*, pages 235–245. ACM, 2020.
10. Xiong Zheng and Vijay K. Garg. Parallel and distributed algorithms for the housing allocation problem. In Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller, editors, *23rd International Conference on Principles of Distributed Systems, OPODIS 2019, December 17-19, 2019, Neuchâtel, Switzerland*, volume 153 of *LIPIcs*, pages 23:1–23:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.
11. Vijay K. Garg. A lattice linear predicate parallel algorithm for the dynamic programming problems. *CoRR*, abs/2103.06264, 2021.

12. Craig M Chase and Vijay K Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11(4):191–201, 1998.
13. Joseph JáJá. *An introduction to parallel algorithms*, volume 17. Addison-Wesley Reading, 1992.