Fairly Allocating Goods in Parallel

Rohan Garg
Purdue University
West Lafayette, IN, USA
rohang@purdue.edu

Alexandros Psomas
Purdue University
West Lafayette, IN, USA
apsomas@cs.purdue.edu

ABSTRACT

We initiate the study of parallel algorithms for fairly allocating indivisible goods among agents with additive preferences. We give fast parallel algorithms for various fundamental problems, such as finding a Pareto Optimal and EF1 allocation under restricted additive valuations, finding an EF1 allocation for up to three agents, and finding an envy-free allocation with subsidies. On the flip side, we show that fast parallel algorithms are unlikely to exist (formally, *CC*-hard) for the problem of computing Round-Robin EF1 allocations.

KEYWORDS

Fair Division; Parallel Algorithms

ACM Reference Format:
Rohan Garg and Alexandros Psomas. 2025. Fairly Allocating Goods in Parallel. In *Proc. of the 24th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2025), Detroit, Michigan, USA, May 19 – 23, 2025*, IFAAMAS, 9 pages.

1 INTRODUCTION

The last two decades have witnessed a remarkable improvement in our computational power, largely due to the widespread adoption of parallel computing. Parallel computing has become the dominant paradigm in computer architecture, mainly in the form of multi-core processors [7, 22]. At the same time, in the overwhelming majority of the AI literature, "efficient algorithm" is a synonym for "efficient sequential algorithm." To bridge this gap, we initiate the study of parallel algorithms for a fundamental problem in fair division: allocating a set of m indivisible items to n agents with additive preferences.

Some classical algorithms for this problem proceed in rounds, e.g. the Round Robin procedure or the Envy-Cycle Elimination procedure [17] that achieve envy-freeness up to one item (henceforth, EF1), while others are computationally intractable (NP-hard), e.g. the maximum Nash welfare (MNW) solution that achieves Pareto efficiency (henceforth, PO) and EF1. *Our goal in this paper is to design, for various, fundamental fair division tasks, algorithms that run in polylogarithmic time and use a polynomial number of processors, or to prove that no such algorithm is likely to exist.*

Large-scale fair resource allocation problems are the prime candidates for parallel fair division algorithms. We give two examples of problems that fit this description here. In the context of high-performance computing, computational workloads have grown in size and complexity considerably. Managing the timely completion of these computational workloads is fundamentally a *resource allocation* problem, e.g. [16, 19]. Another use case is for distributing items for disaster relief. In 2017, after Hurricane Harvey, the United States Federal Emergency Management Agency (FEMA) was in charge of distributing 5.1 million meals, 4.5 million liters of water and over 20,000 tarps to Houston. In the same year, FEMA had to distribute 1.6 million meals, 2.8 million liters of water and roughly 5,000 tarps to Puerto Rico after Hurricane Maria [21]. These large-scale fair resource allocation problems necessitate fast and fair solutions.

1.1 Our contribution

As a warm-up for the reader unfamiliar with the capabilities of parallel algorithms, in Section 3 we consider the basic problem of whether, given an allocation, various fairness properties can be quickly verified in parallel. We show that envy-freeness, EF1, and envy-freeness up-to-any item (EFX) can all be checked efficiently in parallel, i.e. we give *NC* algorithms for verifying these properties. In Section 4, we show how to use algorithms with logarithmic query complexity [20] to get fast parallel algorithms for computing EF1 allocations for two and three agents, as well as how to compute EF1 and fractionally PO allocations for two agents, by mimicking the adjusted winner process. We also give a parallel value-query algorithm that finds an EF1 allocation for two agents with general monotonic valuations. Our last result for this section shows how to find EF1 allocations for n identical additive agents in parallel.

In Sections 5 and 6, we study the complexity of allocating items to *restricted additive agents*, that is when the value of agent i for item j is either 0 or v_j (i.e. each item has an *inherent* value v_j and agent i either sees this value or not), and the value of agent i for a subset of items S is simply $\sum_{j\in S} v_{i,j}$. We first explore the complexity of finding an EF1 allocation. Arguably, the simplest EF1 algorithm in this setting is the Round-Robin procedure (agents choose items one at a time, following a fixed order). In Section 5, we show that, for a given order σ over the agents, one cannot "shortcut" the execution of Round-Robin: the problem is *CC*-hard. Surprisingly, this holds even for the case when each agent positively values at most 3 items and each item is positively valued by at most 3 agents.

Despite this strong negative result, we can efficiently, in parallel, compute an EF1 and PO allocation when there are a constant number of inherent values, even when agents positively value more items, and items are positively valued by more agents. Furthermore, our algorithm outputs allocations that guarantee each agent at least $\lfloor \frac{v_i}{n} \rfloor$ items where v_i is the number of items agent i has a positive value for and n is the number of agents. The complexity of our algorithm is parameterized by t, the number of inherent values: it runs in time $O(\log^2(mn))$ and requires $O(m^{5.5+t}n^{5.5})$ processors. Our

Proc. of the 24th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2025), Y. Vorobeychik, S. Das, A. Nowé (eds.), May 19 – 23, 2025, Detroit, Michigan, USA. © 2025 International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org).

algorithm is via a reduction to the problem of *maximum weight perfect matching* in a bipartite graph. A beautiful result of [18] shows that a *minimum* weight perfect matching (which can be used to find a maximum weight perfect matching) can be found efficiently in parallel when the weight of the heaviest edge is polynomially bounded. Our reduction creates multiple copies of each agent such that the unique item matched to the j-th copy corresponds to the item allocated in the j-th round of *some* Round-Robin procedure (and hence the overall allocation is EF1). The edge weights are increasing (for different copies of the same agent), in a way that every maximum weight matching must give a high-value item to a copy of agent i before giving two high-value items to copies of a different agent. The restriction on the valuations allows us to control the rate at which the weights increase, and specifically bound the maximum weight by a polynomial, so that the algorithm of [18] can be used. We note that when weights are not bounded by a polynomial, the maximum weight matching problem cannot be solved efficiently in parallel (formally, the problem is *CC*-hard), so removing the condition on the valuation functions would require a fundamentally different approach.

Finally, in Section 7, we study the problem of fair allocation with subsidies [4, 12], where the goal is to find an integral allocation of the items as well as payments to the agents, such that the overall solution is envy-free. We give an *NC* algorithm for this problem, and, in fact, prove that one can compute similar solutions in parallel even in the presence of additional constraints on the payments, e.g. "A should not be paid more than B", or "A should be paid no more than 10 dollars." We formulate the problem of finding a constraint-satisfying and envy-eliminating vector as a purely graph-theoretic problem on a graph we call the *payment rejection graph*. The constraints are included by adding edges to this graph. In the most general sense, we can add edges to our graph that correspond to a constraint of the form "if agent i gets paid more than x dollars, then agent j must get paid more than y dollars". Any meaningful overall constraint that can be formulated as a set of such smaller constraints can be added to the problem instance. We highlight that it is not straightforward to implement such constraints in the existing algorithms for the fair division with subsidies problem, especially if one insists on a parallel solution. Our main insight here is that by carefully constructing a large graph to represent the set of all payment vectors, the problem of simultaneously eliminating envy and satisfying constraints can be solved by computing directed reachability in parallel. Details and any missing proofs appear in the full version of the paper [9].

1.2 Related work

Understanding the parallel complexity of various problems has been a central theme in theoretical computer science, with some major recent breakthroughs, e.g. [1]. However, the parallel complexity of problems in fair division remains relatively unstudied. The closest works to ours are that of Zheng and Garg [23] and Friedman [8]. Zheng and Garg [23] study the housing allocation and housing market problems, and give parallel and distributed algorithms. The housing allocation problem asks for a matching between n agents and n houses when agents have strict orderings over the houses. The housing market problem asks for a matching between n agents and n houses when the agents arrive at a market each owning a single house. On the flip side, Zheng and Garg [23] show that finding the core of a housing market is *CC*-hard by showing that the Top-Trading Cycle Algorithm also solves a *CC*-complete problem: Lexicographically First Maximal Matching. Friedman [8] studies the parallel complexity of allocating m divisible homogeneous resources to a set of n agents with nondecreasing utility functions over the amount of each resource received. They show, that for n processors, the parallel time complexity of finding an allocation that has welfare no more than ϵ less than a welfare-maximizing allocation is lower bounded by $\Omega(m \log \frac{1}{n\epsilon})$. They also give an efficient parallel algorithm that computes an approximately accurate solution for $m = 2$ resources.

2 PRELIMINARIES

We consider the problem of allocating a set $\mathcal{M}$ of indivisible goods, labeled by $\{1, \ldots, m\}$, to a set of agents $\mathcal{N}$, labeled by $\{1, \ldots, n\}$. A *fractional* allocation $X \in [0,1]^{n \cdot m}$ defines for each agent $i \in \mathcal{N}$ and $j \in \mathcal{M}$ the fraction of item j that agent i receives. A allocation X is *integral* if $X_{i,j} \in \{0,1\}$ for all $i \in \mathcal{N}$ and $j \in \mathcal{M}$. An allocation $X = (X_1, \ldots, X_n)$ is *complete* if $\cup_{i \in \mathcal{N}} X_i = \mathcal{M}$ and *partial* otherwise. Unless stated otherwise, we use allocation to refer to a complete allocation. We use the term *bundle* to refer to a subset of items, and use $[k]$ to denote the set $\{1, \ldots, k\}$. We use the terms *item* and *good* interchangeably throughout the paper.

Each agent $i \in \mathcal{N}$ has a private valuation function $v_i : 2^{\mathcal{M}} \to \mathbb{R}_+$ which describes the utility agent i receives for each bundle. A valuation function v_i is *additive* if $v_i(X_i) = \sum_{j \in X_i} X_{i,j} \cdot v_i(\{j\})$. A valuation function v_i is *restricted additive* when v_i is additive, and for each item $g \in \mathcal{M}$, $v_i(g) \in \{0, v(g)\}$. To ease notation, we write $v_{i,j} = v_i(\{j\})$ for the value of agent i for item j.

An allocation X is *envy-free* (EF) if $v_i(X_i) \geq v_i(X_j)$ for all agents $i, j \in \mathcal{N}$. Since integral EF allocations don't always exist (e.g. consider the case of a single item and two agents that have positive value for it), the community has turned to notions of approximate fairness. An integral allocation X is *envy-free up to one good* (EF1) if for all agents $i, j \in \mathcal{N}$ there exists a good $g \in X_j$ such that $v_i(X_i) \geq v_i(X_j \backslash g)$ [17]. An integral allocation X is *envy-free up to any good* (EFX) if for all agents $i, j \in \mathcal{N}$, for all goods $g \in X_j$, $v_i(X_i) \geq v_i(X_j \backslash g)$ [5]. The *envy-graph* for an allocation X is the complete weighted directed graph $G_X = (\mathcal{N}, E)$, where there is a vertex for each agent $i \in \mathcal{N}$, and there is an edge $e \in E$ from vertex i to vertex j with the weight $v_i(X_j) - v_i(X_i)$ [17].

An allocation X *Pareto dominates* another allocation Y if $v_i(X_i) \geq v_i(Y_i)$, for all $i \in \mathcal{N}$, and there exists some agent j such that $v_j(X_j) > v_j(Y_j)$. An integral allocation is called *Pareto-Optimal* (PO) or *Pareto-Efficient* (PE) if no other integral allocation Pareto dominates it. An allocation is called *Fractionally Pareto-Optimal* (fPO) if no other (integral or fractional) allocation Pareto dominates it. An allocation is α-PO, for $\alpha \in (0,1]$, if there is no other allocation such that *every* agent's utility is larger by a factor of $1/\alpha$.

Fair division with subsidies. In the problem of fair division with subsidies, we eliminate the envy of an allocation by using payments. An *allocation with payments* $X_{\vec{q}} = (X, \vec{q})$ is a tuple of an integral allocation X and a payment vector $\vec{q} = (q_1, \ldots, q_n)$, where q_i is the payment to agent i. Under such an allocation with payments

$X_{\vec{q}}$, agent i's utility is $v_i(X_i) + q_i$. We can extend the definition of envy-freeness to this setting: an allocation with payments $(X, \vec{q})$ is *envy-free* if $v_i(X_i) + q_i \geq v_i(X_j) + q_j$ for all agents $i, j \in \mathcal{N}$. An allocation X is *envy-freeable* if there exists a payment vector $\vec{q}$ such that $(X, \vec{q})$ is envy-free. For a given envy-freeable allocation X, a payment vector $\vec{q}$ is *envy-eliminating* if the allocation with payments $X_{\vec{q}}$ is *envy-free*. [12] prove that, given an envy-freeable allocation X, one can find an envy-eliminating payment vector for X by computing all-pairs-shortest paths on the envy graph of X with the edge weights negated.

Parallel computation. For sequential algorithms, our model of computation is a single processor that has access to some memory. For parallel algorithms, in this paper, we adopt the CREW (**C**oncurrent **R**ead **E**xclusive **W**rite) PRAM (Parallel RAM) model of computation [14]. The CREW PRAM model allows simultaneous access to any one memory location for read instructions only.[1] We assume a shared memory model where each processor has some local memory to execute its program and all processors can access some amount of global shared memory. Additionally, all computation is *synchronous*, i.e., all processors are coordinated by a common clock.

To describe parallel algorithms, we use p_k to denote the k-th processor. Often we will index processors by items or agents or pairs, e.g. p_j for the processor assigned to item j, or $p_{(i,j)}$ for the processor assigned to the agent i, item j pair. We give the basic notions of efficiency and hardness in the parallel world as well as some useful parallel primitives in the full version of the paper [9].

3 VERIFICATION OF FAIRNESS

As a warm-up, we show that, given an allocation, we can efficiently (in parallel) verify its fairness properties.

Theorem 3.1. *Given an allocation X and the valuation functions of n additive or restricted additive agents, the problem of deciding whether X satisfies EF is in NC.*

Proof. We wish to test whether or not each agent prefers their own bundle to any other agent's bundle. For each ordered pair of agents (i, j), we assign $|X_i|+|X_j| \leq m$ processors. First, we compute the value of $v_i(X_i)$ and $v_i(X_j)$ using parallel sum procedures; each sum takes $O(\log m)$ time. Next, we test whether $v_i(X_i) \geq v_i(X_j)$. For each ordered pair of agents (i, j), we assign one bit in memory, initially set to 0. If $v_i(X_i) \geq v_i(X_j)$, processor $p_{(i,j)}$ will flip the bit indexed by the agent pair (i, j) to 1. Setting this bit for all ordered pairs is done simultaneously. Finally, using n^2 processors, we take the minimum across these bits to find if there is any pair of agents that does not respect envy-freeness; this step takes $O(\log n^2)$ time; if the minimum is 1, then the allocation is envy-free. We overall used at most $O(n^2 m)$ processors, and the total time was $O(\log m + \log n)$. □

To test if an allocation is EF1, we use similar ideas to that of testing EF. For every ordered pair of agents, we allocate $O(m)$ processors to test whether or not the removal of each item from j's bundle eliminates i's envy. For each item, we set a separate bit to 1 to signify whether or not that item's removal eliminates envy; we take the maximum across all bits to see if there is any one item for which the EF1 condition holds. Then we ensure that the minimum for all ordered pairs of agents is 1.

To test if an allocation is EFX, we run the same procedure as that of testing EF1 except instead of computing maximums of the inequality bits, we compute minimums.

Theorem 3.2. *Given an allocation X and the valuation functions of n additive or restricted additive agents, the problem of deciding whether X satisfies EF1 is in NC.*

Theorem 3.3. *Given an allocation X and the valuation functions of n additive or restricted additive agents, the problem of deciding whether X satisfies EFX is in NC.*

4 EF1 ALLOCATIONS FOR TWO AND THREE ADDITIVE AGENTS

In this section, we discuss how to efficiently, in parallel, compute EF1 allocations for two and three agents. We first focus on the case of two and three *additive* agents and give algorithms that work via a reduction. We then give a family of algorithms (parametrized by a trade-off parameter) that compute an EF1 allocation for two agents with general *monotonic* valuations in the value query model.

4.1 Two and three additive agents

Our algorithms work via a reduction. Specifically, Oh et al. [20] prove that EF1 allocations can be found using a logarithmic number of value queries[2] for two agents with monotonic utilities and three agents with additive utilities. For sequential algorithms and additive agents, implementing a query takes $O(m)$ time, since one needs to sum the values of the items in a subset. However, using $O(m)$ processors, one can implement a value query in $O(\log m)$ time. Therefore, the results of [20] can be directly translated to our setting.

Theorem 4.1. *For the case of additive agents, if there exists a query algorithm that uses k queries to compute an allocation X, then there exists a parallel algorithm that uses $O(m)$ processors and computes X in time $O(k \log m)$.*

Proof. Consider any sequential fair division algorithm $\mathcal{A}$ for additive agents that only has query access to agents' valuations, and specifically, it can ask a query, $query(S, i)$, to learn the value of subset S for agent i. Suppose $\mathcal{A}$ requires k query calls. We give a parallel algorithm that efficiently implements $query(S, i)$. In order to implement $query(S, i)$, we run a parallel-sum procedure using $O(m)$ processors on the elements specified by S, using the valuation function of agent i. In $O(\log m)$ time, we then have the sum of all item values in S for agent i. Since $\mathcal{A}$ uses k queries, and for each query we compute a sum, we get an overall runtime of $O(k \log m)$ using $O(m)$ processors. □

As corollaries, we can derive *NC* algorithms that produce EF1 allocations for two or three agents with additive utilities via the

[1] It is well known that the strongest PRAM model, the CRCW PRAM model, with p processors can be simulated by the weakest PRAM model, the EREW PRAM model, with p processors with at most a $O(\log p)$ factor slowdown [13].

[2] A value query on input $i, S \subseteq \mathcal{M}$ returns the value $v_i(S)$ of agent i for the subset S of items.

algorithms of Oh et al. [20], since these algorithms have polylogarithmically many value queries.

Corollary 4.2. *The problem of finding an EF1 allocation for two and three additive agents is in NC.*

Next, we notice that the two-agent algorithm of Oh et al. [20] mimics the classic cut-and-choose algorithm from continuous cake-cutting. The authors show that for any ordering of indivisible items on a line, there exists a way for the first agent to cut (split the items into two pieces) such that when the second agent selects her favorite piece, the overall allocation is EF1. The main difficulty is, of course, finding this cut using only a logarithmic number of queries. Here, we observe that since such a cut can be found for an arbitrary ordering of the items, by ordering the items in increasing $v_{1,j}/v_{2,j}$ (mimicking the adjusted-winner process [3]) we can also guarantee fractional Pareto efficiency (fPO). Since the basic operations (sorting, adding, etc) in the adjusted winner process can be done in parallel, we overall get a fractionally PO and EF1 *NC* algorithm.

Theorem 4.3. *The problem of finding an fPO and EF1 allocation for two additive agents is in NC.*

4.2 Two general monotonic agents

General monotonic valuations may not have a succinct representation in memory. As such, we have to adopt a different parallel model - the *adaptive* model of computation. In the adaptive model, we are allowed to directly issue *value queries*. Further, in each round our algorithm can make polynomially many (in n and m) simultaneous queries per round. The goal is to minimize the number of *rounds* of computation required for the algorithm to terminate.

Studying the adaptive complexity of various problems has seen a surge of interest in recent years, specifically in the areas of submodular maximization and minimization. We refer the reader to [2, 6] for more details on adaptive complexity.

The algorithm given by [20] works by placing items on a line and finding a *cut point*. They do this by finding a good g such that agent 1 is almost indifferent between the set of goods to the left of g and the set of goods to the right of g. We follow the same approach of placing the items on a line and finding the *cut point*, but utilizing parallel queries we can find the cut point faster.

We give an algorithm parameterized by a value k where k represents a desired trade-off between the total number of rounds and the number of queries per round.

The algorithm of [20] begins by placing the items on a line. For each good g, let L_g and R_g represent the set of goods to the left and to the right of good g respectively. The goal is to find the rightmost good g such that $v_1(L_g) \leq v_1(R_g \cup \{g\})$. After finding this good, we check to see if $v_1(L_g) \leq v_1(R_g)$. If this inequality holds, we present the partition $(L_g \cup \{g\}; R_g)$ to agent 2 and let them select their preferred half. If the inequality does not hold, we present the partition$(L_g; R_g \cup \{g\})$ to agent 2 and let them select their preferred half. In [20], the authors find the good g by using binary search. Translated directly, this becomes a $O(\log m)$-round adaptive algorithm with one query per round.

The first observation is that with m processors, we can, in parallel, evaluate each item as a potential *cut point* and identify the rightmost good g in exactly one round of computation. Evaluating the two halves to determine the partition takes another query and letting agent 2 select their favorite half takes another query. In total, this takes 3 rounds and requires $O(m)$ queries per round. As a result, we get the following theorem.

Theorem 4.4. *There is a 3-round adaptive algorithm that makes $O(m)$ queries per round that finds an EF1 allocation of m indivisible items for two general monotonic agents.*

More carefully, we can use a recursive approach to introduce a trade-off parameter k between the total number of rounds and the number of queries per round.

Theorem 4.5. *There is a $k+2$-round adaptive algorithm that makes $O(\sqrt[k]{m})$ queries per round that finds an EF1 allocation of m indivisible items for two general monotonic agents.*

Proof. Let ℓ be the length of the line segment we are considering and i be the round number we are currently on. Let g^* be the rightmost good such that $v_1(L_{g^*}) \leq v_1(R_{g^*} \cup \{g^*\})$. In round one, we consider the entire line of items. We set $\ell = m$ and $i = 1$. We query the goods with indices that are multiples of the value $(\sqrt[k]{m})^{k-i}$. For each good g in this set, We check to see if the inequality $v_1(L_g) \leq v_1(R_g \cup \{g\})$ holds. After this set of queries, we will have identified a line *segment* of size $(\sqrt[k]{m})^{k-1}$ that contains good g^*. Now, we move to round two and only consider this smaller line segment. So, $\ell = (\sqrt[k]{m})^{k-1}$ and $i = 2$. We recursively repeat this process. In the i'th iteration, $\ell = (\sqrt[k]{m})^{k-(i-1)}$. After $k-1$ iterations, we have a line segment of length exactly $\sqrt[k]{m}$. We then apply Theorem 4.4 to query all items in this smaller segment to find the correct good g^* in one round. Finally, we require two more rounds: one to determine the correct partition and one to find agent 2's preferred half. □

4.3 Identical additive agents

Finally, we show that for n identical and additive agents, there exists a simple *NC* algorithm for finding an EF1 allocation. For identical agents, it is easy to predict what item will be allocated in the k-th round of the Round-Robin procedure: since all agents have the same ranking over the items, the k-th item allocated is precisely the k-th favorite item.

Theorem 4.6. *The problem of finding an EF1 allocation for n identical and additive agents is in NC.*

5 TRADITIONAL EF1 ALGORITHMS ARE INHERENTLY SEQUENTIAL

In this section, we give limits to what parallel algorithms can achieve in our setting. Specifically, we show that "Round-Robin looking" allocations cannot be found efficiently in parallel. We consider the following problem, which we call Fixed-Order Round-Robin: Given a set $\mathcal{M}$ of m items, a set $\mathcal{N}$ of n agents, a strict ordering $\sigma = \{\sigma_1 \succ \cdots \succ \sigma_n\}$ over the agents, and a designated agent, item pair (i^*, j^*), decide if agent i^* is allocated item j^* by Round-Robin with σ as the order over the agents. We give a log-space reduction from Lexicographically-First Maximal Matching to Fixed-Order Round-Robin.

Theorem 5.1. *Fixed-Order Round-Robin is CC-Hard, even for the case of n restricted additive agents, i.e. $v_{i,j} \in \{0, v(j)\}$, where every*

agent positively values at most 3 *items and every item is positively valued by at most* 3 *agents.*

Proof. We reduce 3-Lexicographically-First Maximal Matching (3-LFMM) to Fixed-Order Round-Robin. In the LFMM problem, we are given a bipartite graph $G = (X, Y, E)$ where $X = \{x_i\}_{i=1}^{n}$, $Y = \{y_i\}_{i=1}^{m}$, and $E \subseteq X \times Y$. The lexicographically first maximal matching of G, M_{lex}, is produced by successively matching vertices in X, in the order $x_1, \ldots, x_n$, each one with the available vertex in Y that has the smallest index. The LFMM problem is to decide if a designated edge belongs to the lexicographically first maximal matching of a bipartite graph G. In the 3-LFMM problem, each vertex in G has degree at most 3. [15] prove that 3-LFMM is *CC*-complete.

Let $G = (X, Y, E)$ with a designated edge e^* be an instance of the 3-LFMM problem. Without loss of generality, let $|X| \geq |Y|$. We construct an instance of Fixed-Order Round-Robin as follows. For each vertex $x_i \in X$ we create an agent, and for each vertex $y_j \in Y$ we create an item. For each $e = (x_i, y_j) \in E$, we set $v_{i,j} = m - j + 1$. For $e = (x_i, y_j) \notin E$, $v_{i,j} = 0$. By construction, since each vertex in G has degree at most 3, each agent values positively at most 3 items, and each item is valued positively by at most 3 agents. Let the ordering of the vertices in X correspond to σ, i.e. $\sigma_i = i$. This construction takes logarithmic space. Therefore, to conclude the proof of Theorem 5.1, it suffices to show that $e^* = (x_{i^*}, y_{j^*}) \in M_{lex}$ iff agent i^* gets item j^* in the execution of Round-Robin that corresponds to σ. We prove a stronger statement, using induction.

Our inductive hypothesis is that, for a given number k, for any $j \in [m]$, $(x_k, y_j) \in M_{lex}$ iff agent k gets item j in the k-th round of the execution of Round-Robin that corresponds to σ. For $k = 1$, we have that $(x_1, y_j) \in M_{lex}$ iff $j = argmin_{\ell \in [m]}\{(x_1, y_\ell) \in E\}$, which, by construction, happens iff $v_{1,j} > v_{1,\ell}$ for all $\ell \in [m]$, i.e., iff agent 1 picks item j in the execution of Round Robin, noting that agent 1 is first in σ and that agents don't pick items they have zero value for. Assume the hypothesis is true for numbers less than or equal to k, and that $(x_{k+1}, y_j) \in M_{lex}$. By the inductive hypothesis, all edges $(x_i, y_\ell) \in M_{lex}$ for $i \leq k$ correspond to items allocated in the first k rounds in the execution of Round-Robin. $(x_{k+1}, y_j) \in M_{lex}$ iff j is the smallest index among all unmatched neighbors of x_{k+1} at the $(k+1)$-st step of building the lexicographically first maximal matching. Since smaller indices (of edges) correspond to strictly higher valuations, we have that, by construction, $(x_{k+1}, y_j) \in M_{lex}$ iff $v_{k+1,j} > v_{k+1,\ell}$ for all items $\ell \in [m]$ that have not been allocated in the first k rounds in the execution of Round-Robin. This holds iff j is the item selected by agent $k+1$ in the $(k+1)$-st round of Round-Robin (noting once again that, in Round-Robin, agents don't pick items with zero value for them). □

6 EF1 + PO FOR RESTRICTED ADDITIVE WITH A BOUNDED NUMBER OF VALUES

In this section, we present a new randomized parallel algorithm that gives an EF1 and PO allocation for assigning m indivisible items to n agents with *restricted additive* valuations. Recall that a valuation function v_i is *restricted additive* if v_i is additive, and for each item $g \in \mathcal{M}$, $v_i(g) \in \{0, v(g)\}$. The complexity of the algorithm is parameterized by t, the number of "inherent" item values, i.e. the number of different values $v(g)$ can take. Formally, our parallel algorithm has *polylog(m, n)* time complexity and requires $poly(m, n) \cdot O(m^t)$ processors.

Here, we describe how our algorithm works. We construct a weighted bipartite graph G where on one side of the graph, we have vertices corresponding to items, and on the other side, we have vertices corresponding to copies of agents. We ensure that the two sides have the same number of vertices by adding $mn - m$ dummy items that all agents have zero value for. We first describe the vertices representing the set of items. Let this side be A. To populate A, we create a vertex a_j for each $j \in \mathcal{M}$. We will think of A as partitioned in buckets $\mathcal{M}_1 \ldots \mathcal{M}_t$, where t is the number of different item values. M_i is the set of items with the i'th highest value. Finally, we add vertices that correspond to dummy items. Let the set of dummy vertices be $\mathcal{M}_d$. On the other side of the bipartition, we have vertices corresponding to copies of agents. Let this side be B. We create m buckets of n vertices where each of these n vertices represents an agent. Formally, we create a set of vertices $\{b_{1,j}, b_{2,j}, \ldots, b_{n,j}\}$ for $j \in [m]$. The c-th bucket will be called $\mathcal{N}_c$. For each $j \in \mathcal{M}_f$ and $i \in \mathcal{N}$, if $v_{i,j} > 0$, we add, for all $c \in [m]$, the edge $(a_j, b_{i,c})$ with weight $-m^{t-f} \cdot c$. For each dummy item $j \in \mathcal{M}_d$ and $i \in [n]$, we add, for all $c \in [m]$, the edge $(a_j, b_{i,c})$ with weight 0. We refer to this weight function as $w(\cdot)$. We give an example of the weighted bipartite graph in Figure 1 where there are three agents, three items in buckets $\mathcal{M}_1$ and $\mathcal{M}_f$, and some dummy vertices in $\mathcal{M}_d$.

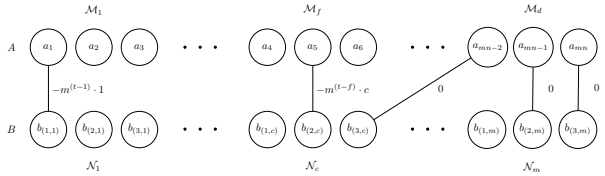

Figure 1: G for an instance with three agents.

Once the graph $G = (X \cup Y, E, w)$ is constructed, we compute a maximum-weight perfect matching M^* and return the allocation corresponding to M^*. We assume that every (non-dummy) item is valued by *someone*. This is without loss of generality since, if an item is not valued by anyone, this can be checked efficiently in parallel, and the item can be discarded. The formal description of the algorithm is given in Algorithm 1. We prove that this algorithm always outputs an EF1 and PO allocation.

Lemma 6.1. *Algorithm 1 outputs a PO allocation.*

Proof. We show that the resulting maximum-weight matching saturates the left side of the bipartition. As a result, all items are allocated to agents that value those items since an edge in the graph is only present between an item-agent pair when the agent values that item.

We show this by using Hall's Marriage Theorem [11]. Hall's Theorem characterizes necessary and sufficient conditions for a bipartite graph to have a perfect matching. Recall Hall's Theorem:

Algorithm 1 Parallel Algorithm for Division of Goods with Restricted Additive Values

Input: n agents, m items, $v_i(g) \in [0, v(g)]\ \forall i \in \mathcal{N}, g \in \mathcal{M}$
Output: EF1 + PO Allocation X

Sort the items into buckets by value $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_t$
Add a set of $mn - m$ dummy items $\mathcal{M}_d$ to $\mathcal{M}$
Create an empty, weighted, undirected graph G
for all $j \in \mathcal{M}$ in parallel **do**
    Add a vertex a_j
for all $j \in \mathcal{M}$, $i \in \mathcal{N}$ in parallel **do**
    Add a vertex $b_{i,j}$
for all $j \in \mathcal{M}$, $b_{i,c}$ for $i \in \mathcal{N}$ and $c \in [m]$ in parallel **do**
    if $(v_{i,j} > 0)$ **then**
        Add the edge $(a_j, b_{i,c})$ to G with weight $-m^{t-f} \cdot c$ if item $j \in \mathcal{M}_f$
    if $(a_j \in \mathcal{M}_d)$ **then**
        Add the edge $(a_j, b_{i,c})$ to G with weight 0
Compute the Max-Weight Perfect Matching M^* in G
for all $j \in \mathcal{M}$, $b_{i,c}$ for $i \in \mathcal{N}$ and $c \in [m]$ in parallel **do**
    if $((a_j, b_{i,c}) \in M^*)$ **then**
        $X_i = X_i \cup j$
return: X

Theorem 6.2 (Hall's Theorem [11]). *A bipartite graph $G = (L \cup R, E)$ contains an L-saturating perfect matching if and only if for every subset W of L, its neighborhood, $N_G(W)$, satisfies $|N_G(W)| \geq |W|$.*

This holds for the graph G used in Algorithm 1. Consider any subset W of A. Since W is comprised of vertices corresponding to non-dummy items and vertices corresponding to dummy items, it suffices to show that a vertex of either type has a large enough neighborhood in B. Every item $j \in \mathcal{M}$ is associated with a vertex $a_j \in A$. A vertex a_j corresponding to the non-dummy item j has at least m edges coming out of it: an edge to the same agent i in each of the m blocks. Any vertex corresponding to a dummy item is connected to all vertices in B. Thus, any subset W of A will result in a neighborhood of size at least $|W|$ in B. So, G will always contain at least one perfect matching that saturates A. Since our allocation corresponds to this matching, each item is given to an agent that values it. In the restricted additive setting, this corresponds to a PO allocation. □

The next lemma is crucial for showing the EF1 guarantee of Algorithm 1.

Lemma 6.3. *For any two agents i and j, and $c \in [m-1]$, i weakly prefers the item matched to her in bucket $\mathcal{N}_c$ to the item that is matched to j in bucket $\mathcal{N}_{c+1}$.*

Proof. Let vertex j be matched to $M^*(j)$ in M^*. We want to show that $v_i(M^*(b_{i,c})) \geq v_i(M^*(b_{j,c+1}))$. Assume that this is not true. Then, the following holds for matching M^*. Agent i is matched to item ℓ from $\mathcal{M}_{f+h}$ for some $h \in [t-f]$ in bucket $\mathcal{N}_c$ and agent j is matched to item ℓ' from $\mathcal{M}_f$ in bucket $\mathcal{N}_{c+1}$. However, we know that agent i values item ℓ'. So the edge $(a_{\ell'}, b_{i,c})$ exists in G. We show that we can augment M^* and increase its weight, thus proving that it was not the maximum weight matching in the first place; a contradiction. Towards this, consider matching item ℓ' to agent i in bucket $\mathcal{N}_c$ and matching item ℓ to agent i in any bucket $\mathcal{N}_p$ for $p > c$ where $b_{i,p}$ is unmatched. We show that the new matching has a higher total weight.

Notice that besides this item switch, all other edges remain the same. So, we need to show:

$$w(a_{\ell'}, b_{i,c}) + w(a_\ell, b_{i,p}) > w(a_{\ell'}, b_{j,c+1}) + w(a_\ell, b_{i,c}).$$

Expanding using the weight function, we have:

$$w(a_{\ell'}, b_{i,c}) + w(a_\ell, b_{i,p}) = -cm^{t-f} - pm^{t-(f+h)}$$
$$w(a_{\ell'}, b_{j,c+1}) + w(a_\ell, b_{i,c}) = -(c+1)m^{t-f} - cm^{t-(f+h)}.$$

Subtracting the weight of the old edges from the modified matching edges, we have:

$$-cm^{t-f} - pm^{t-(f+h)} + (c+1)m^{t-f} + cm^{t-(f+h)} = m^{t-f} + (c-p)m^{t-(f+h)}.$$

Since we know that $c \geq 1$ and $p \leq m$, the smallest value that $(c-p)$ can take is $(1-m)$. So, we get:

$$\begin{aligned} m^{t-f} + (c-p)m^{f+h} &\geq m^{t-f} + (1-m)m^{t-(f+h)} \\ &> m^{t-f} + (-m)m^{t-(f+h)} \\ &= m^{t-f} - m^{(t-f-h+1)}. \end{aligned}$$

The largest value the second term can take is when $h = 1$. This gives us,

$$m^{t-f} - m^{(t-f-h+1)} \geq m^{t-f} - m^{t-f} = 0.$$

Thus, we can strictly increase the weight of the matching; a contradiction. □

The repeated application of Lemma 6.3 gives us Lemma 6.4.

Lemma 6.4. *Algorithm 1 outputs an EF1 allocation.*

Proof. Notice that every vertex in B is matched to some item in A (the matched item may be a dummy item of value 0). By Lemma 6.3, for any two agents i and j, we have that $v_i(M^*(b_{i,c})) \geq v_i(M^*(b_{j,c+1}))$. So, in particular, we have that agent i weakly prefers the item they received in bucket $\mathcal{N}_1$ to the item agent j receives in bucket $\mathcal{N}_2$. Agent i also weakly prefers the item they received in bucket $\mathcal{N}_2$ to the item agent j receives in bucket $\mathcal{N}_3$ and so on. As a result, we know that agent i has at least the same value for the set of items she receives in buckets $\mathcal{N}_1$ through bucket $\mathcal{N}_m$ as that of the set of items agent j receives in buckets $\mathcal{N}_2$ through bucket $\mathcal{N}_m$. Thus, by removing the item agent j receives in $\mathcal{N}_1$, agent i will certainly have no envy for agent j. □

Finally, we show that Algorithm 1 runs in randomized polylogarithmic time using $f(m,n) \cdot O(m^t)$ processors where f is a polynomial (in m and n) function.

Lemma 6.5. *Algorithm 1 takes $O(\log^2(mn))$ time using $O(m^{5.5+t}n^{5.5})$ processors.*

Combined, Lemmas 6.1, 6.4, and 6.5 give us the following theorem.

Theorem 6.6. *Algorithm 1 is a parallel algorithm that returns an EF1 and Pareto Optimal allocation of m indivisible items to n agents with restricted-additive valuations, from a set of t different inherent item-values, in time $O(\log^2(mn))$ using $O(m^{5.5+t}n^{5.5})$ processors.*

Notice that binary valuations are a special case of restricted additive valuations (with one inherent item value). Thus, we get an *RNC* algorithm for binary valuations.

Corollary 6.7. *The problem of finding an EF1 and Pareto Optimal allocation for n additive agents with binary valuations is in RNC.*

We note here that for a given instance of restricted additive fair division, we can reduce the number of inherent item values at the expense of some loss in the EF1 and PO guarantees. Concretely, if we round the valuations $v_{i,j}$ to $v'_{i,j}$ such that $v'_{i,j} \in [\alpha \cdot v_{i,j}, v_{i,j}]$ then an EF1 and PO allocation in v' is an α-EF1 and α-PO allocation with respect to v. Assuming the item values are in the range $[1, V]$, one can use such a rounding to create $\lceil \log_{\frac{1}{\alpha}}(V+1) \rceil$ intervals.

Theorem 6.8. *For n restricted additive agents and m indivisible items such that $v(g) \in [1, V]$, an α-EF1 and α-PO allocation can be computed in time $O(\log^2(mn))$ using $O(m^{5.5+\lceil \log_{\frac{1}{\alpha}}(V+1) \rceil} n^{5.5})$ processors.*

Proof. We begin by rounding the valuations v to new valuation functions v' where $v'_{i,j} \in [\alpha \cdot v_{i,j}, v_{i,j})$ for some $\alpha \in [0, 1)$. Specifically, all values in the interval $[1, 1/\alpha)$ will be rounded down to 1, values in the interval $[1/\alpha, (1/\alpha)^2)$ will be rounded down to $1/\alpha$, and so on. This creates $\lceil \log_{\frac{1}{\alpha}}(V+1) \rceil$ intervals, and therefore $t = \lceil \log_{\frac{1}{\alpha}}(V+1) \rceil$ different inherent item-values in v'. Using Algorithm 1, we can compute an EF1 and PO allocation X with respect to v'. We claim that X is an α-EF1 and α-PO allocation with respect to v.

First, we show the α-EF1 guarantee. Since X is EF1 with respect to v', for every pair of agents i, j, there exists some good g in agent j's bundle such that (1) $v'_i(X_i) \geq v'_i(X_j \setminus \{g\})$. Since $\alpha \in [0, 1)$, we have that (2) $v_i(X_i) \geq v'_i(X_i)$. By the construction of v', we also have (3) $v'_i(X_j \setminus \{g\}) \geq \alpha \cdot v_i(X_j \setminus \{g\})$. Stitching (1), (2), and (3) together, we the α-EF1 guarantee:

$$v_i(X_i) \geq v'_i(X_i) \geq v'_i(X_j \setminus \{g\}) \geq \alpha \cdot v_i(X_j \setminus \{g\}).$$

Next, we show the α-PO guarantee. Consider some other allocation X'. Since X is PO with respect to v', we know, for any other allocation X', X' does not Pareto dominate X. That is, there exists at least one agent i such that $v'_i(X'_i) \leq v'_i(X_i)$. By construction of v', we have that, for every subset of items S, $\alpha \cdot v_i(S) \leq v'_i(S) \leq v_i(S)$. Therefore, we have:

$$\alpha \cdot v_i(X'_i) \leq v'_i(X'_i) \leq v'_i(X_i) \leq v_i(X_i).$$

That is, agent i's utility cannot be improved by a factor more than $1/\alpha$, and therefore X is α-PO. □

7 FAIR ALLOCATIONS WITH SUBSIDIES IN PARALLEL

In this section, we study fair division with subsidies. First, we show how to adjust the algorithm of [12] and compute an envy-freeable allocation and corresponding envy-eliminating payment vector in parallel. Second, we give an efficient parallel algorithm that computes a payment vector that not only eliminates envy but additionally satisfies other user-specified constraints (defined later in this section).

7.1 Envy-Freeable allocations and payments in *NC*

We prove that the algorithm of [12], for finding an envy-freeable allocation and envy-eliminating payment vectors can be parallelized.

Theorem 7.1. *The problem of finding an envy-freeable allocation X and an envy-eliminating payment vector for X for n additive agents is in NC.*

First, note that the welfare-maximizing allocation, which gives each item to the agent with the highest value for it, can be shown to be envy-freeable. Now, given an envy-freeable allocation X, the algorithm of [12] for finding envy-eliminating payments at a high-level, constructs the envy-graph G_X, negates all the edge weights in G_X, and computes all-pairs-shortest-paths on the modified G_X. Then, for each agent, i, the algorithm singles out the path with the lowest overall weight (out of n shortest paths) that starts at i's vertex in G_X. One can show that paying agent i the sum of the edge weights along this path results in an envy-eliminating payment. We show that all these steps can be parallelized efficiently, noting that one can apply known techniques to solve the all-pairs-shortest-paths problem in parallel.

7.2 Computing constrained envy-eliminating payment vectors in *NC*

In this section, we give a different algorithm for finding an envy-eliminating payment vector, $\vec{q}$. A key feature of our algorithm is that it allows for additional constraints on the final solution.

Formally, we are given an allocation X of m items to n additive agents each with a valuation function v_i that takes integer values, and a set C of constraints of the form "*if agent i is paid more than x dollars, then agent j must be paid more than y dollars.*" We are interested in computing a payment vector $\vec{q}$ that is envy-eliminating and satisfies all such constraints in C, or deciding that no such vector exists. We call this problem Constrained Payments. We assume that no agent is paid more than $m\Delta$ dollars, where $\Delta = \max_{i,j} v_{i,j}$. This is because, for any meaningful solution, we need not pay any one agent more than $m\Delta$ dollars as this is the maximum value any agent can have for the entire set of items.

We note that many non-trivial constraints on the payment vector can be formulated as a set of these smaller individual constraints. For example, the constraint "*agent 1 should not be paid more than agent 2*" can be imposed by adding the constraint "*if agent 1 is paid more than x dollars, then agent 2 is paid more than x dollars*" for all $x \in [m\Delta]$. Or, the constraint "*agent 1 should not be paid more than 10 dollars*" can be imposed by adding the constraint "*if agent 1 is paid more than 10 dollars, then agent 2 is paid more than $m\Delta$ dollars*". When C is empty, we solve the original problem of finding an unconstrained envy-eliminating payment vector.

Theorem 7.2. *If $v_{i,j}$ is integral for all $i \in [n]$ and $j \in [m]$, Constrained Payments can be solved in $O(\log^2(mn\Delta))$ time using $O(n^3m^3\Delta^3)$ processors, where $\Delta = \max_{i,j} v_{i,j}$.*

Note that C is upper-bounded by $O(n^2m^2\Delta^2)$, and hence the size of C does not appear in the bounds of Theorem 7.2. The main challenge with incorporating constraints into the final payment vector is that the previous approach of running all-pairs-shortest-paths on the envy graph does not allow us to isolate specific dollar amounts for which we want to impose a constraint on. To resolve this, we construct a larger, modified graph where each vertex corresponds to an agent *coupled* with a specific payment amount. We call this graph *the payment rejection graph*. Our goal is to select a single vertex for each agent from the payment rejection graph, which will define the final payment vector. An edge in the payment rejection graph will exactly represent the causal relationship defined by a constraint.

Formally, the payment rejection graph is a directed graph $G_p = (V, E)$ with a total of $nm\Delta$ vertices. We arrange the $nm\Delta$ vertices on an $n \times m\Delta$ two-dimensional grid. Vertex (i, j) corresponds to agent i being paid $\vec{q}_i = j$ dollars. We use the term *rejecting a vertex* (i, j) to denote that $\vec{q}_i = j$ will not be in the final payment vector. We use the term *payment row* for an agent i when referring to the set of vertices $\{(i, j) \mid j \in [m\Delta]\}$. An edge from node (i, j) to (k, ℓ), denoted by $(i, j) \rightarrow (k, \ell)$, signifies that if we have rejected all vertices (i, j') for $j' \leq j$, then we also reject all vertices (k, ℓ') for $\ell' \leq \ell$. The idea of modeling rejections as edges in a directed graph was first used to find consistent global states in distributed systems [10]. We adapt this approach to find envy-eliminating payments.

We maintain a "current" payment vector and initialize it to the all-zero payment vector (i.e we select the vertex $(i, 0)$ for each agent i). Then, we iteratively increase the agents' payments by one dollar until no envy is present. Although this process seems sequential, we show that we can quickly, in parallel, determine which payment components are not part of *any* envy-eliminating payment vector. To see this, consider two agents i and j and a current payment vector $\vec{q}$. We can compute the envy i has for j (or, similarly, j has for i) subject to these two payments by comparing i's value for i's bundle and payment $(v_i(X_i) + \vec{q}_i)$ to that of j's $(v_i(X_j) + \vec{q}_j)$. If it is the case that i envies j subject to the payments $\vec{q}_i$ and $\vec{q}_j$, we must increase i's payment by one dollar. So, we will increment $\vec{q}_i$ to $\vec{q}_i + 1$. Now, if there is any other agent k that envies i subject to the new payment, we know we will have to increase k's payment by one dollar as well. As a result, we can make the following inference: if we pay agent i more than $\vec{q}_i$ dollars, we have to pay agent k more than $\vec{q}_k$ dollars. So, we can place an edge $(i, \vec{q}_i) \rightarrow (k, \vec{q}_k)$. Notice that the meaning of these edges holds transitively (i.e if $(i, \vec{q}_i) \rightarrow (j, \vec{q}_j)$ and $(j, \vec{q}_j) \rightarrow (k, \vec{q}_k)$, then $(i, \vec{q}_i) \rightarrow (k, \vec{q}_k)$). Since this observation does not require us to use any information about other vertices in the graph besides the set $\{(i, \vec{q}_i), (i, \vec{q}_i + 1), (k, \vec{q}_k)\}$, by using a separate processor for each pair of vertices, we can place all edges in the graph simultaneously. An example of a payment rejection graph for a specific valuation profile can be found in the full version of the paper [9].

Now, we identify which vertices will not be a part of any envy-eliminating payment vector initially, and then follow edges from these vertices. These vertices are of the form $(i, 0)$ where there is some other vertex $(j, 0)$ where $v_i(X_i) < v_i(X_j)$. Agent i must be paid and so vertex $(i, 0)$ *will* be rejected. To find all vertices that are reachable from initially rejected vertices, we take the transitive closure of G_p, which can be done efficiently in parallel [13]. Vertices that are reachable from any initially rejected vertex will be marked as rejected. Then, we find the minimum payment component for each agent using a parallel reduction operator. If there is no minimum component (i.e all vertices along some agent's payment row have been rejected), then we output "No satisfying vector". If all agents have a valid payment, we output the envy-eliminating payment vector $\vec{q}$. Since edges in G_p correspond exactly to a constraint in C, all constraints can be added to G_p simultaneously in parallel. Now, the algorithm identifies the first envy-eliminating payment vector that respects these constraints.

The algorithm and proof of Theorem 7.2 are in the full version of the paper [9]. As a direct result, we get an *NC* algorithm when Δ is bounded by a polynomial of n and m.

Corollary 7.3. *The problem of finding an envy-eliminating and constraint-satisfying payment vector is in NC if* $\Delta = \max_{i,j} v_{i,j}$ *is polynomial in n and m.*

8 DISCUSSION

Our results show that many problems in fair division admit efficient parallel solutions. Our hardness result shows that the traditional Round Robin allocation cannot be efficiently computed in parallel. We leave open many interesting research directions. Is the problem of finding any EF1 allocation *CC*-Hard? Are any problems in fair division *P*-Complete? Can we give deterministic parallel algorithms for restricted additive fair division?

ACKNOWLEDGMENTS

The authors are supported in part by an NSF CAREER award CCF-2144208, a Google AI for Social Good award, and research awards from Google and Supra.

REFERENCES

[1] Nima Anari and Vijay V Vazirani. 2020. Planar graph perfect matching is in NC. *Journal of the ACM (JACM)* 67, 4 (2020), 1–34.
[2] Eric Balkanski and Yaron Singer. 2018. The adaptive complexity of maximizing a submodular function. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing* (Los Angeles, CA, USA) *(STOC 2018)*. Association for Computing Machinery, New York, NY, USA, 1138–1151. https://doi.org/10.1145/3188745.3188752
[3] Steven J Brams and Alan D Taylor. 1996. A procedure for divorce settlements. *Mediation Quarterly* 13, 3 (1996), 191–205.
[4] Johannes Brustle, Jack Dippel, Vishnu V. Narayan, Mashbat Suzuki, and Adrian Vetta. 2020. One Dollar Each Eliminates Envy. In *Proceedings of the 21st ACM Conference on Economics and Computation* (Virtual Event, Hungary) *(EC '20)*. Association for Computing Machinery, New York, NY, USA, 23–39. https://doi.org/10.1145/3391403.3399447
[5] Ioannis Caragiannis, David Kurokawa, Hervé Moulin, Ariel D. Procaccia, Nisarg Shah, and Junxing Wang. 2019. The Unreasonable Fairness of Maximum Nash Welfare. *ACM Trans. Econ. Comput.* 7, 3, Article 12 (sep 2019), 32 pages. https://doi.org/10.1145/3355902
[6] Chandra Chekuri and Kent Quanrud. [n.d.]. *Submodular Function Maximization in Parallel via the Multilinear Relaxation.* 303–322. https://doi.org/10.1137/1.9781611975482.20 arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9781611975482.20
[7] Keith D. Cooper. 2014. Making Effective Use of Multicore Systems A software perspective: The multicore transformation (Ubiquity symposium). *Ubiquity* 2014, September, Article 3 (Sept. 2014), 8 pages. https://doi.org/10.1145/2618407
[8] Eric J. Friedman. 1993. The Complexity of Allocating Resources in Parallel: Upper and Lower Bounds. In *Complexity in Numerical Optimization*. WORLD SCIENTIFIC, 107–127. https://doi.org/10.1142/9789814354363_0007
[9] Rohan Garg and Alexandros Psomas. 2023. Fairly Allocating Goods in Parallel. arXiv:2309.08685 [cs.GT] https://arxiv.org/abs/2309.08685
[10] Vijay K. Garg and Rohan Garg. 2019. Parallel Algorithms for Predicate Detection. In *Proceedings of the 20th International Conference on Distributed Computing and*

Networking (Bangalore, India) *(ICDCN '19)*. Association for Computing Machinery, New York, NY, USA, 51–60. https://doi.org/10.1145/3288599.3288604

[11] P. Hall. 1935. On Representatives of Subsets. *Journal of the London Mathematical Society* s1-10, 1 (1935), 26–30. https://doi.org/10.1112/jlms/s1-10.37.26 arXiv:https://londmathsoc.onlinelibrary.wiley.com/doi/pdf/10.1112/jlms/s1-10.37.26

[12] Daniel Halpern and Nisarg Shah. 2019. Fair Division with Subsidy. In *Algorithmic Game Theory: 12th International Symposium, SAGT 2019, Athens, Greece, September 30 – October 3, 2019, Proceedings* (Athens, Greece). Springer-Verlag, Berlin, Heidelberg, 374–389. https://doi.org/10.1007/978-3-030-30473-7_25

[13] Joseph F. JáJá. 1992. *An Introduction to Parallel Algorithms*. Addison-Wesley.

[14] Richard M. Karp and Vijaya Ramachandran. 1990. CHAPTER 17 - Parallel Algorithms for Shared-Memory Machines. In *Algorithms and Complexity*, Jan Van Leeuwen (Ed.). Elsevier, Amsterdam, 869–941. https://doi.org/10.1016/B978-0-444-88071-0.50022-9

[15] Dai Tri Man Lê, Stephen A Cook, and Yuli Ye. 2011. A formal theory for the complexity class associated with the stable marriage problem. In *Computer Science Logic (CSL'11)-25th International Workshop/20th Annual Conference of the EACSL*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[16] Tan N. Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. 2020. AlloX: compute allocation in hybrid clusters. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 31, 16 pages. https://doi.org/10.1145/3342195.3387547

[17] R. J. Lipton, E. Markakis, E. Mossel, and A. Saberi. 2004. On Approximately Fair Allocations of Indivisible Goods. In *Proceedings of the 5th ACM Conference on Electronic Commerce* (New York, NY, USA) *(EC '04)*. Association for Computing Machinery, New York, NY, USA, 125–131. https://doi.org/10.1145/988772.988792

[18] Ketan Mulmuley, Umesh V. Vazirani, and Vijay V. Vazirani. 1987. Matching is as Easy as Matrix Inversion. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing* (New York, New York, USA) *(STOC '87)*. Association for Computing Machinery, New York, NY, USA, 345–354. https://doi.org/10.1145/28395.383347

[19] Deepak Narayanan, Fiodar Kazhamiaka, Firas Abuzaid, Peter Kraft, Akshay Agrawal, Srikanth Kandula, Stephen Boyd, and Matei Zaharia. 2021. Solving Large-Scale Granular Resource Allocation Problems Efficiently with POP. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 521–537. https://doi.org/10.1145/3477132.3483588

[20] Hoon Oh, Ariel D. Procaccia, and Warut Suksompong. 2019. Fairly Allocating Many Goods with Few Queries. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence* (Honolulu, Hawaii, USA) *(AAAI'19/IAAI'19/EAAI'19)*. AAAI Press, Article 265, 8 pages. https://doi.org/10.1609/aaai.v33i01.33012141

[21] Aimee Thomas, Gloria Young, and Peter Meyer. 2018. Lessons from a Hurricane: Supply Chain Resilience in a Disaster - An Analysis of the US Disaster Response to Hurricane Maria. https://conservancy.umn.edu/items/a78b1577-9e0a-4871-b562-94b8525ba1fc Accessed: 2024-10-16.

[22] Walter Tichy. 2014. The Multicore Transformation Opening Statement: The multicore transformation (Ubiquity symposium). *Ubiquity* 2014, May, Article 1 (May 2014), 8 pages. https://doi.org/10.1145/2618393

[23] Xiong Zheng and Vijay K. Garg. 2020. Parallel and Distributed Algorithms for the Housing Allocation Problem. In *23rd International Conference on Principles of Distributed Systems (OPODIS 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 153)*, Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 23:1–23:16. https://doi.org/10.4230/LIPIcs.OPODIS.2019.23