Decentralised Parallel Machine Scheduling for Multi-Agent Task Allocation

Kathryn S. Macarthur[1], Meritxell Vinyals[2], Alessandro Farinelli[3], Sarvapali D. Ramchurn[4], and Nicholas R. Jennings[5]

`{ksm08r, sdr, nrj}@ecs.soton.ac.uk`[1,4,5], `meritxell@iiia.csic.es`[2], `alessandro.farinelli@univr.it`[3]

[1,4,5] School of Electronics and Computer Science, University of Southampton, UK
[2] IIIA, CSIC, Barcelona, Spain
[3] University of Verona, Italy

Abstract. Multi-agent task allocation problems pervade a wide range of real-world applications, such as search and rescue in disaster management, or grid computing. In these applications, where agents are given tasks to perform in parallel, it is often the case that the performance of all agents is judged based on the time taken by the slowest agent to complete its tasks. Hence, efficient distribution of tasks across heterogeneous agents is important to ensure a short completion time. An equivalent problem to this can be found in operations research, and is known as scheduling jobs on unrelated parallel machines (also known as $R||C_{\max}$). In this paper, we draw parallels between unrelated parallel machine scheduling and multi-agent task allocation problems, and, in so doing, we present the decentralised task distribution algorithm (DTDA), the first decentralised solution to $R||C_{\max}$. Empirical evaluation of the DTDA is shown to generate solutions within 86–97% of the optimal on sparse graphs, in the best case, whilst providing a very good estimate (within 1%) of the global solution at each agent.

1 Introduction

Multi-agent task allocation problems pervade a wide range of real-world scenarios, such as search and rescue in disaster management [10], and environmental monitoring with mobile robots [12]. In such problems, a set of agents, such as rescue agents in search and rescue, must work together to perform a set of tasks, often within a set amount of time. In particular, agents are given tasks to perform in parallel, and the performance of the team is usually judged based on the time taken by the slowest member of the team. For example, consider a team of firefighters that must put out fires in a building before medical personnel can enter and provide first aid for civilians — in this case, the medical personnel can enter the building only when all fires have been extinguished. In this case, the task allocation problem we focus on would consist of firefighter agents and firefighting tasks. Each firefighter would be assigned a number of fires to put out, and medical personnel would only be able to enter after the last fire has been extinguished by the last firefighter. Thus, the performance of all the firefighters (i.e., how early the medical personnel can enter the building) is judged on how long the last firefighter has taken to finish. Hence, efficient distribution of tasks across such heterogeneous agents is important to ensure an early completion time. In more detail, in this context, agents must find a solution that ensures all tasks are performed in the shortest amount of time.

Now, an analogue to this particular class of task allocation problems has been widely studied in operations research and is known as *scheduling on unrelated parallel machines*, or $R||C_{\max}$ [4]. In this problem, there are a set of heterogeneous machines, and a set of tasks which must be performed by those machines (equivalent to agents), potentially under some constraints (for example, where a given machine cannot execute certain tasks), such that the maximum finish time across machines, known as the *makespan*, is minimised. However, while many algorithms (for example, [5,6,8], see [9] for a review) have been developed to solve $R||C_{\max}$, they all require the existence of some central authority. However, this introduces a central point of failure: for example, if communication to and from the central authority were to fail, then another authority would have to be found so that it could re-compute the solution and try to communicate it. In addition, in realistic large-scale environments, which can potentially have hundreds of agents, there is a need for an algorithm that will scale well in terms of communication and computation, which centralised algorithms are unable to do. Hence, the challenge we face is to build decentralised algorithms that are robust to failure, and so, there is a clear need for a multi-agent systems solution to solve $R||C_{\max}$ in our domains.

Against this background, in this paper, we develop a novel, decentralised, approximate algorithm to solve the unrelated parallel machine scheduling problem, called the Decentralised Task Distribution algorithm (DTDA). DTDA uses localised message passing through the min-max algorithm to find good quality, per-instance bounded approximate solutions in a distributed, efficient manner. In more detail, the min-max algorithm is a localised message passing algorithm from the GDL (Generalised Distributive Law) family [1], which factorises the global problem into local agent-level sub-problems, by exploiting possible independence among agents' actions. For example, assume two firefighters are in distant parts of a large building, and must decide which of the fires surrounding them they must put out. In this situation, the two firefighters can avoid considering each others' actions but should coordinate their own actions with any firefighters which are close by, and therefore would be making their decisions regarding some of the same fires.

This paper advances the state of the art in the following ways:

– First, we provide a novel representation of the $R||C_{\max}$ problem, in terms of a junction graph, which is a graphical model frequently used to represent factored functions [7].
– Second, we show how we can simplify the min-max algorithm, and then run it over this junction graph representation to generate approximate solutions to the $R||C_{\max}$ problem, with per-instance bounds, through decentralised

message passing between agents. To the best of our knowledge, this is the first known application of min-max to the $R||C_{\max}$ problem.
– Third, we empirically evaluate our approach, by comparing our performance with two benchmark algorithms (optimal and greedy), on graphs with differing average degrees. We find that it produces solutions within 86–97% of the optimal, in the best case, in sparse environments, and outperforms greedy by up to 16%.

The rest of this paper is structured as follows. In Section 2, we formulate $R||C_{\max}$. Next, we decompose the problem in Section 3 so that it can be distributed, and discuss how we simplify the min-max algorithm in Section 4. Then, we present our decentralised algorithm in Section 5. Next, we empirically evaluate the quality of the solutions given by the DTDA in Section 6. Finally, in Section 7, we conclude.

2 Problem Formulation

In this section, we formally describe the problem we introduced in Section 1. In more detail, our problem consists of finding an allocation of tasks to agents in order to to optimize overall execution performance in terms of the system's makespan. First, in Section 2.1, we provide the basic definitions of the environment. Then, in Section 2.2, we describe our objective function, which formalises the analogy with the $R||C_{\max}$ problem overviewed in Section 1.

2.1 Basic Definitions

Let the set of agents be denoted as $A = \{a_1, a_2, \dots, a_{|A|}\}$, and the set of tasks to be completed as $T = \{t_1, t_2, \dots, t_{|T|}\}$. Each agent a_i can perform a set of tasks $T_i \subseteq T$. For each agent $a_i \in A$ we denote a cost function, $\chi_i : T_i \rightarrow \mathbb{R}^+$, that returns the total run–time incurred at a_i to perform some task $t \in T_i$. Thus, $\chi_i(t_k)$ returns the application-specific runtime required for agent a_i to perform task t_k. A graphical representation of an example environment is given in Figure 1, in which there are 3 agents (circles) and 4 tasks (squares). Each agent is connected to the tasks it can potentially perform by lines in the graph, and edges are labelled with $\chi_i(t_k)$. Thus, for example, agent a_1 will incur a runtime of 30 to perform task t_2 whereas agent a_2 will only incur a runtime of 20.

Given this, the problem is to schedule all of the tasks in T across the agents in A such that all tasks are completed and the makespan is minimised. We formally define this objective in the next section.

2.2 Objective Function

To show the analogy with $R||C_{\max}$, consider the set of jobs as the set of agents' tasks and the set of machines as the set of agents. Specifically, the objective of $R||C_{\max}$ is to find a mapping $m : A \rightarrow 2^T$ from tasks to agents, such that the

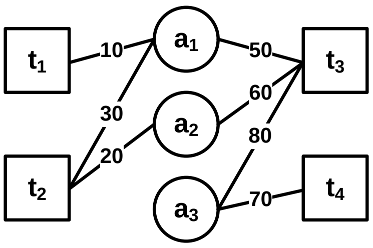


Fig. 1. A graphical representation of a sample $R||C_{\max}$ problem, in which agents are represented by circles, and tasks by squares. Edges between agents and tasks indicate an agent can perform a task, at a cost denoted on the edge.

makespan is minimized. In particular, we wish to find this mapping subject to a number of constraints. First, each task must only be computed by one agent:

$$m(a_i) \cap m(a_j) = \emptyset, \forall a_i, a_j \in A, i \neq j$$

and second, that all tasks must be computed:

$$\bigcup_{a_i \in A} m(a_i) = T$$

in which $m(a_i)$ denotes the set of tasks assigned to agent a_i, under mapping m. Given this, our objective is to find a mapping m^* as follows:

$$m^* = \arg \min_{m \in M} \max_{a_i \in A} \sum_{t_k \in m(a_i)} \chi_i(t_k) \tag{1}$$

where M is the set of all possible mappings. For instance, Figure 2 depicts an optimal mapping of the problem in Figure 1 where optimal assignments from agents to tasks are shown with arrows. Thus, the optimal mapping m^* is defined as: $m^*(a_1) = \{t_1, t_3\}$, $m^*(a_2) = \{t_2\}$ and $m^*(a_3) = \{t_4\}$ with a makespan value of $\max(10 + 50, 20, 70) = 70$.

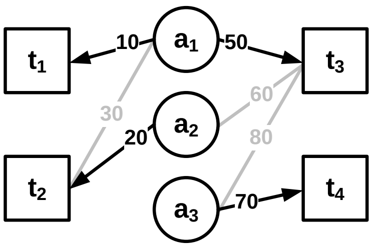


Fig. 2. An optimal mapping from agents to tasks, for the problem in Figure 1. Arrows between agents and tasks depict an agent being assigned to a task.

Now, in order to solve the objective function given in Equation (1) in a decentralised way, we first decompose the problem so that it can be modelled as a junction graph, like that shown in Figure 3.

3 $R||C_{max}$ Representation

In more detail, a junction graph [7] is an undirected graph such that:

- each node i, known as a *clique*, represents a collection of variables, X_i.
- each clique node i in the junction graph has a potential, $\psi_i : X_i \rightarrow \mathbb{R}^+$, which is a function defined over the set of variables in the clique.
- two clique nodes i and j are joined by one edge that contains the intersection of the variables they represent.

Using this representation allows us to explicitly represent the interactions between agents, in terms of the common tasks they can complete. To do this, we represent each agent as a clique in the graph containing variables relating to the tasks that agent can complete. In so doing, the representation facilitates the application of a particular GDL [1] message-passing algorithm, called min-max, which we can use to find a solution to Equation (1) in a decentralised manner. We explain min-max in more detail in Section 4.

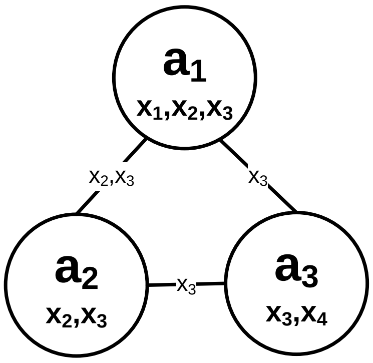


Fig. 3. The junction graph formulation of the scenario given in Figure 1. Large circles are cliques, with the elements of the cliques listed. Edges are labelled with common variables between cliques.

In order to apply min-max, we reformulate the objective function in Equation (1) in terms of a junction graph. Figure 3 depicts the junction graph representing the problem in Figure 1. We define the set of variables $X = \{x_1, \ldots, x_{|T|}\}$ to include one variable x_k for each task $t_k \in T$. Thus, the junction graph in Figure 3 contains four variables, $\{x_1, x_2, x_3, x_4\}$, which correspond to the four tasks in Figure 1. Each variable $x_k \in X$ takes a value from its domain, which contains all of the IDs of agents that can complete task t_k. Hence, if $x_k = i$, then we know that agent a_i is allocated to t_k. For instance, the domain of x_2 in Figure 3 is composed of two values, 1 and 2, corresponding to the indices of the agents that can perform task 2, a_1 and a_2. Notice that, in doing this, we enforce the constraint that exactly one agent must perform every task.

Additionally, we use $X_i = \{x_k | t_k \in T_i\}$ as the set of variables representing the tasks agent a_i can perform. With slight abuse of notation, we use $\mathbf{X}_i$ to denote a configuration of the variables in X_i. Given this, in our formulation, an agent a_i's clique will contain all variables in X_i (in Figure 3, labels within circles denote agents' cliques). Thus, in Figure 3, the set of variables corresponding to agent a_2's clique, X_2, is composed of x_2 and x_3, which are the two tasks that a_2 can perform in Figure 1.

Finally, we encode the cost function of agent a_i as a potential function, $\psi_i(\mathbf{X}_i)$, representing the total time that a_i will take to compute the configuration $\mathbf{X}_i$. Formally:

$$\psi_i(\mathbf{X}_i) = \sum_{x_k \in \mathbf{X}_i, x_k = i} \chi_i(t_k) \tag{2}$$

Thus, in Figure 3 the potential function of agent a_2, ψ_2, which is defined over variables x_2 and x_3, returns a runtime of 60 for the configuration $x_2 = 1, x_3 = 2$, which is the runtime incurred at a_2 to complete task 3 in Figure 1.

By the definition of a junction graph, two agents, a_i and a_j, will be joined by an edge in the junction graph if and only if $X_i \cap X_j \neq \emptyset$. In Figure 3 edges are labelled with the intersection of two cliques. Thus, agent a_2 is linked to a_3 by an edge that contains the only common variable in their cliques: x_3. Given this, we denote agent a_i's neighbours, $\mathcal{N}(a_i)$, as the set of agents with which a_i shares at least one variable, and therefore, are neighbours in the junction graph.

Given all this, the junction graph encodes our objective function (Equation (1)) as follows:

$$\mathbf{X}^* = \arg\min_{\mathbf{X}} \max_{a_i \in A} \psi_i(\mathbf{X}_i) \tag{3}$$

where $\mathbf{X}_i$ is the *projection* of $\mathbf{X}$ over variables X_i. In more detail, given two sets of variables $X_i, X_j \subseteq X$, a projection of $\mathbf{X}$ over X_j contains the variable values found in $\mathbf{X}$ for all $x_k \in X_i \cap X_j$ in X_j .

Now that we have a junction graph formulation of the problem, we can decompose our objective function amongst the agents. In order to do this, we compute the marginal function $z_i(X_i)$ at each agent, which describes the dependency of the global objective function (given in Equation (1)) on agent a_i's clique variables. This is computed as follows:

$$z_i(\mathbf{X}_i) = \min_{\mathbf{X}_{-i}} \max_{a_j \in A} \psi_j(\mathbf{X}_j) \tag{4}$$

where $\mathbf{X}_{-i}$ is a configuration of X_{-i}, where $X_{-i} = X \setminus X_i$ and $\mathbf{X}_j$ is the projection of $\mathbf{X}_{-i}$ over the variables in X_j.

Finally, in the presence of a unique solution, the optimal state of a_i's clique variables is:

$$\mathbf{X}_i^* = \arg\min_{\mathbf{X}_i} z_i(\mathbf{X}_i) \tag{5}$$

This decomposition facilitates the application of the min-max GDL algorithm, as our operators here are min and max, to find a decentralised solution to $R||C_{\max}$. Thus, in the next section, we introduce min-max, and prove its most important property: that it will always converge within a finite number of iterations.

4 The min-max Algorithm

The min-max algorithm is a member of the GDL framework [1], which is a framework for localised message passing algorithms. We know from the literature (for example, [3]) that GDL algorithms are efficient (particularly in sparse graphs — in our case, where each agent can only perform a subset of the tasks), and provide generally good solutions, so it fits to apply one here. In addition, GDL algorithms are proven to converge to optimal solutions on acyclic graphs (which, in our case, would be junction trees). In more detail, GDL based algorithms are all based on a commutative semiring. Min-max is based on the commutative semiring $\langle \mathbb{R}^+, \min, \max, \infty, 0\rangle$ where min is the additive operator and max the multiplicative operator.

Given a junction graph, the GDL approach consists of exchanging messages between agents and their neighbours in the junction graph. Let $X_{ij} = X_i \cap X_j$ be the intersection of variables of two neighbours, a_i and a_j, and $X_{i\setminus j} = X_i \setminus X_j$. In the GDL, agent a_i exchanges messages with a neighbour $a_j \in \mathcal{N}(a_i)$ containing the values of a function $\mu_{i\to j} : X_{ij} \to \mathbb{R}^+$.

Initially, all such functions are defined to be 0 (the semiring's multiplicative identity). Once messages have been received, the message is computed as follows:

$$\mu_{i\to j}(\mathbf{X}_{ij}) = \min_{\mathbf{X}_{i\setminus j}} \max \left[\psi_i(\mathbf{X}_i), \max_{a_k \in \mathcal{N}(a_i)|k\neq j} \mu_{k\to i}(\mathbf{X}_{ki})\right] \tag{6}$$

where $\mathbf{X}_{ij}$ is a configuration of X_{ij}, $\mathbf{X}_{i\setminus j}$ is a configuration of $X_{i\setminus j}$, and $\mathbf{X}_i$ and $\mathbf{X}_{ki}$ stand for the projection of $\mathbf{X}_{ij}$,$\mathbf{X}_{i\setminus j}$ over variables in X_i and X_{ki} respectively.

Similarly, for each clique a_i, GDL computes an approximation of the marginal contribution of its variables, $\tilde{z}_i : \mathbf{X}_i \to \mathbb{R}^+$, as:

$$\tilde{z}_i(\mathbf{X}_i) = \max \left[\psi_i(\mathbf{X}_i), \max_{a_j \in \mathcal{N}(a_i)} \mu_{j\to i}(\mathbf{X}_{ji})\right] \tag{7}$$

Now, the idempotency of max, the multiplicative operator [11], allows us to make a number of changes to the standard GDL formulation, which we explain below.

Idempotency implies that, for all $r \in \mathbb{R}^+$, $\max(r, r) = r$. Hence, the idempotency of the multiplicative operator implies that repeatedly combining the same information will not produce new information. Moreover, when an operator is idempotent, it defines a partial ordering over the set $\mathbb{R}^+$. In our case, both operations are idempotent. For the min operator, the order is the natural order of real numbers: i.e., $r \leq s$ if and only if $\min(r, s) = r$. Meanwhile, for the max operator, the order is the inverse of the natural ordering of numbers: i.e., $r \geq s$ if and only if $\max(r, s) = r$. From these, we can deduce that, as min orders the elements in exact inverse to max, $\min(r, \max(r, s)) = r$.

Given all this, due to the idempotency of the min-max commutative semiring, the following equality holds for any $\mathbf{X}'_i$, $\mathbf{X}_i$ where $X'_i \subseteq X_i$:

$$\max \left[\psi_i(\mathbf{X}_i), \min_{\mathbf{X}'_i} \psi_i(\mathbf{X}'_i) \right] = \psi_i(\mathbf{X}_i) \tag{8}$$

This idempotency property is a feature we exploit in our implementation of min-max, to improve efficiency. In more detail, the idempotency of the min-max semiring, a property not shared with other non-idempotent GDL semirings, allows us to *simplify* the GDL equations such that:

- in Equation (6), when an agent a_i sends a message to a neighbour a_j, it does not need to marginalise out any previously received messages from a_j, thus reducing computation at agents.
- in Equation (7), the agent's marginal contributions can be computed recursively by combining messages from multiple iterations, which, again, reduces computation at the agent. This is because repeatedly combining messages from previous iterations will not change the approximate marginal contribution at an agent.

Thus, in the next section, we will introduce min-max based on these simplified equations, instead of the original GDL Equations ((6) and (7)), allowing us to simplify the computation at each agent when sending messages.

In addition to this, the idempotency property provides two further properties that make the min-max algorithm more efficient than non-idempotent GDL algorithms: (1) it is guaranteed to converge, even in cyclic graphs (Theorem 1); and (2) it provides an online per-instance bound on the optimal solution value of the problem that it approximates (which we will discuss later on, in Section 5.3). In what follows, we provide the formal proof of convergence.

Theorem 1. *The min-max algorithm is guaranteed to converge in a finite number of iterations.*

Proof. [2] prove the termination of idempotent commutative semirings (Theorem 8). Given the fact that min-max is an idempotent semiring, the min-max algorithm must terminate.

Now that we have shown why the min-max algorithm carries useful properties, in the next section, we present our decentralised task distribution algorithm (DTDA). Our algorithm consists of an algorithmic instantiation of the min-max algorithm, which, when combined with a value propagation phase, allows online per-instance bounds on solution quality to be obtained at each agent.

5 Decentralised Task Distribution Algorithm

Broadly speaking, the DTDA consists of two key steps: applying the min-max algorithm to compute an allocation of tasks to agents, and value propagation to ensure all agents choose the same assignment, and to compute the per-instance bound.

In more detail, the first step of the min-max algorithm propagates information across the agents to produce a set of solutions (we will explain how this can be a set later on). In the second phase (value propagation), agents are arranged in a tree structure, and one solution is chosen that is consistent with all other agents' solutions. We elaborate on these steps in what follows.

5.1 Applying min-max

In the first step of the DTDA, we apply the min-max algorithm over the junction graph described in Section 3, in order to find a set of solutions (distributions of tasks to agents).[1]

Algorithm 1 `minmax()` at agent a_i.

```
1: procedure initialise
2: Initialize messages $\mu_{i\to j}(\mathbf{X}_{ij}) = 0 \quad \forall j \in \mathcal{N}(i)$
3: $\tilde{z}_i(\mathbf{X}_i) = \psi_i(\mathbf{X}_i) = \sum_{x_k \in \mathbf{X}_i, x_k = i} \chi_i(t_k)$ // Intialise marginal contribution
4: Run procedure send_messages.
5:
6: procedure received $\mu_{j\to i}$
7: if $stored(j) \neq \mu_{j\to i}$ then // received different message
8:     $stored(j) = \mu_{j\to i}$ // update stored message
9:     Run procedure update_marginal_contribution
10:    Run procedure send_messages.
11: end if
12:
13: procedure send_messages
14: for $j \in \mathcal{N}(i)$ do
15:     Send $\mu_{i\to j}(\mathbf{X}_{ij})$ to $a_j$
16: end for
17:
18: procedure find_solutions
19: $\mathbf{X_i^*}$ = all states with value $\min_{\mathbf{X}_i} \tilde{z}_i(\mathbf{X}_i)$
```

We present the pseudocode for min-max in Algorithm 1. Now, an agent begins by running the procedure `initialise` (lines 1–4). Each agent starts by

[1] Note that we specify that a set of solutions is produced, because it is possible for more than one solution to have the same value. This is because the solution value is taken to be the largest makespan at one agent — therefore, many allocations of tasks and agents could give the same makespan.

initialising its stored outgoing messages to 0 (line 2), and its marginal contribution function, $\tilde{z}_i(\mathbf{X}_i)$, to the agent's potential, ψ_i, which encodes the agent's own cost function, computed as given in (2) (line 3).

After initialisation, each agent exchanges a message, $\mu_{i\to j}$, with each of its neighbours, $a_j \in \mathcal{N}(a_i)$, in order to find out their approximated marginal contribution for each configuration of the variables they share. This is done via the procedure `send_messages` (lines 13–16). The message $\mu_{i\to j}(X_{ij})$ is sent over all combinations of the variables in X_{ij} (i.e., the intersection of X_i and X_j). The content of the message from an agent a_i to a_j is, therefore, agent a_i's marginal contribution function:

$$\mu_{i\to j}(\mathbf{X}_{ij}) = \min_{\mathbf{X}_{i\setminus j}} \tilde{z}_i(\mathbf{X}_i) \tag{9}$$

When an agent a_i receives a message, it runs the procedure `received` (lines 6–11), in which the agent checks if the message it has received differs from the last message it received from that agent. This is important to ensure that the messages stop being sent when they stop changing, so the algorithm converges to a solution. If the message does differ (line 7), then the a_i updates its *stored* entry for the sending agent (line 8). Afterwards, the agent a_i runs the procedure `update_marginal_contribution` (line 9), which updates its marginal contribution values as follows:

$$\tilde{z}_i(\mathbf{X}_i) = \max\left\{\tilde{z}_i(\mathbf{X}_i), \max_{a_j\in\mathcal{N}(a_i)} \mu_{j\to i}(\mathbf{X}_{ji})\right\} \tag{10}$$

This marginal contribution is approximate because, as we said earlier, GDL algorithms can only compute exact solutions on acyclic graphs (i.e., a junction tree instead of a junction graph). Finally, agent a_i re-sends all of its messages (line 10) in case its marginal contribution value has changed (for example, if a new maximum $\mu_{j\to i}(\mathbf{X}_{ij})$ has been found).

These messages are passed amongst agents until their content no longer changes — at which point, the agent will ascertain the best states for its variables using the procedure `find_solutions` (lines 18–19). In more detail, this is done by assessing the configuration $\mathbf{X}_i^*$, with cost $\tilde{z}_i^*$, that minimises the agent's marginal contribution (line 19):

$$\mathbf{X}_i^* = \arg\min_{\mathbf{X}_i} \tilde{z}_i(\mathbf{X}_i) \tag{11}$$

Hence, this equation provides an approximation of (3). We show in our empirical evaluation (in Section 6) that the solutions DTDA gives are of very good quality on a general problem. Next, we describe our value propagation phase which ensures that all agents apply the same solution and that computes the online per-instance bound of the approximate solution.

5.2 Value Propagation

Once the messages amongst agents have converged, and no further messages need to be sent, we introduce a two-part value propagation phase to ensure the agents all set their values to produce a valid state, and are aware of the quality of their bound. This is required in part due to the likelihood of multiple solutions being present.

In the first part of this phase (see Algorithm 2, lines 1–10), we arrange the agents into a Depth-First Search (DFS) tree using distributed DFS. In particular, a DFS tree must always ensure that agents which are adjacent in the original graph are in the same branch. This ensures relative independence of nodes in different branches of the tree, allowing parallel search. For any given graph, our DFS tree is considered 'valid' if no ties (variable overlaps) between agents in different subtrees of the DFS tree exist. The outcome of this DFS is that each agent has set the values of its *parent* and *children* variables shown in Algorithm 2. Once this has occurred, the root node decides on a configuration of its variables to propagate, $\mathbf{X}_i^*$, as computed in (11), and sends this, along with $v_i = \psi_i(\mathbf{X}_i^*)$ (the actual value of the current solution) and $z_i(\mathbf{X}_i^*)$ (the value of the current solution as computed by min-max) to the node's children. Each of these children adds their own variables onto $\mathbf{X}_i^*$ (line 2), takes the maximum of v and z with what they have received (lines 3 and 4, respectively), and sends these new values onto their own children (line 5).

Algorithm 2 `valueprop` at agent a_i

```
Require: parent, children
 1: procedure received(⟨$\mathbf{X}_p^*, v_p, \tilde{z}_p^*$⟩) from parent
 2: $\mathbf{X}_i^* = \arg\min_{\mathbf{X}_{i\setminus p}} \tilde{z}_i(\mathbf{X}_{i\setminus p}; \mathbf{X}_p^*)$
 3: $v_i = \max(v_p, \psi_i(\mathbf{X}_i^*))$
 4: $\tilde{z}_i^* = \max(\tilde{z}_p^*, \tilde{z}_i(\mathbf{X}_i^*))$
 5: Send ⟨$\mathbf{X}_i^*, v_i, \tilde{z}_i^*$⟩ to all $a_j \in$ children
 6: if children $= \emptyset$ then
 7:     Send ⟨$v_i, \tilde{z}_i^*$⟩ to parent
 8: end if
 9:
10: procedure received(⟨$v_p, \tilde{z}_p^*$⟩) from child
11: if Received messages from all child $\in$ children then
12:     $v_i = \max(v_p, \psi_i(\mathbf{X}_i^*))$
13:     $\tilde{z}_i^* = \max(\tilde{z}_p^*, \tilde{z}_i(\mathbf{X}_i^*))$
14:     $\rho = \tilde{z}_i^* / v_i$
15:     Send ⟨$v_i, \tilde{z}_i^*$⟩ to parent.
16: end if
```

Once this first phase is complete (i.e., the messages have reached the leaf nodes), the leaf nodes pass their marginal contribution and makespan values (the actual value of the solution) back up the tree (lines 6 and 7), to ensure every agent can compute the quality of the solution. Then, when an agent has received such a message from each of its children (line 11), it will update its v and $z_i(\mathbf{X}_i^*)$ values (lines 12 and 13, respectively), calculate its approximation ratio ρ (line 14) — which is the agent's per-instance bound (this will be clarified in the next section), and send the new v and $z_i(\mathbf{X}_i^*)$ values to its parent (line 15). Once these messages have reached the root of the tree, the DTDA is complete, all variables have values consistent across all agents, and all agents are aware of the quality of their solution.

5.3 Proving the Per-instance Bound

Next, we prove that the maximum value of the solution that minimises the approximate marginal contributions of the agents in min-max, or, more formally, $\tilde{z} = \max_{a_i \in A} \min_{\mathbf{X}_i} \tilde{z}_i(\mathbf{X}_i)$, is a lower bound on the cost of the optimal solution of the $R||C_{\max}$ problem, as formulated in Equation (3) (Theorem 2). This lower bound can be used by agents to assess the quality of the min-max solution by bounding its error.

Before proving Theorem 2, we define the relation of *equivalence* among two functions.

Definition 1 (equivalence). *Given two functions $\alpha(X_\alpha)$ and $\beta(X_\beta)$ we say they are equivalent if they: (1) are defined over the same set of variables $X_\alpha = X_\beta$; and (2) return the same values for all possible configurations, $\alpha(\mathbf{X}) = \beta(\mathbf{X})$. We denote such relation of equivalence as $\alpha \equiv \beta$.*

Next, we prove two lemmas, that help to assess Theorem 2. First, in Lemma 1, we state that, at any iteration of the min-max algorithm, the function that results from the combination of the agents' marginal contributions, namely $\tilde{Z}(\mathbf{X}) = \max_{a_i \in A} \tilde{z}_i(\mathbf{X}_i)$, is *equivalent* to the objective function to minimise in $R||C_{\max}$, $\Psi(\mathbf{X}) = \max_{a_i \in A} \psi_i(\mathbf{X}_i)$. Thus, under Lemma 1, $\tilde{Z} \equiv \Psi$. Second, in Lemma 2, we state that $\tilde{z}$, defined as the maximum of the individual agents' marginal solutions, is a lower bound on the value of the optimal value of function $\tilde{Z}$.

We provide formal proofs for these two lemmas below.

Lemma 1. *At any iteration τ of the min-max algorithm, function $\tilde{Z}^\tau(\mathbf{X}) = \max_{a_i \in A} \tilde{z}_i^\tau(\mathbf{X}_i)$ is equivalent to the to the objective function to minimise in $R||C_{\max}$, $\Psi(\mathbf{X}) = \max_{a_i \in A} \Psi_i(\mathbf{X}_i)$.*

Proof. We prove this by induction on τ.

For $\tau = 0$ the case is trivial, $\tilde{Z}^0(\mathbf{X}) = \max_{a_i \in A} \tilde{z}_i^0(\mathbf{X}_i) = \max_{a_i \in A} \psi_i(\mathbf{X}_i) = \Psi(\mathbf{X})$.

Then we prove $\tau = n+1$: that is, that $Z^{n+1} \equiv Z^n$, assuming that $\tau = n$ holds. $\tilde{Z}^{n+1}(\mathbf{X}) = \max_{a_i \in A} \max(\tilde{z}_i^n(X), \max_{a_j \in N(a_i)} \min_{\mathbf{X}_{j \setminus i}} \tilde{z}_j^n(\mathbf{X}_{j \setminus i}))$. Since the max operator is commutative and associative, $\tilde{Z}^{n+1}(\mathbf{X})$ can also be written as $\max_{a_i \in A} \max(\tilde{z}_i^n(\mathbf{X}_i), \max_{a_j \in \mathcal{N}(a_i)} \min_{\mathbf{X}_{i \setminus j}} \tilde{z}_i^n(\mathbf{X}_{i \setminus j}))$. Then, by exploiting the idempotency of the max operator (see Equation (8)), $\tilde{Z}^{n+1}(\mathbf{X})$ simplifies to $\max_{a_i \in A} \tilde{z}_i^n(\mathbf{X}_i)$ and $\tilde{Z}^{n+1} \equiv \tilde{Z}^n \equiv \Psi$. □

Lemma 2. *Given $\tilde{Z}(\mathbf{X}) = \max_{a_i \in A} \tilde{z}_i(\mathbf{X}_i)$, let $\tilde{z}^*$ be the value of the assignment x^* that minimises $\tilde{Z}(\mathbf{X})$. Then, $\tilde{z} = \max_{a_i \in A} \min_{\mathbf{X}_i} \tilde{z}_i(\mathbf{X}_i)$ is a lower bound on $\tilde{z}^*$, $\tilde{z} \leq \tilde{z}^*$.*

Proof. We prove this by contradiction. Assume that there is an assignment $\mathbf{X}$ of X such that $\tilde{Z}(\mathbf{X}) \leq \max_{a_i \in A} \min_{\mathbf{X}_i} \tilde{z}_i(\mathbf{X}_i)$. This leads to a contradiction, because it implies that at least one function $\tilde{z}_i$ evaluated at x is lower than its minimum, $\min_{\mathbf{X}_i} \tilde{z}_i(\mathbf{X}_i)$. □

Finally, we combine these two lemmas to prove our main result, in Theorem 2.

Theorem 2. *Let $\tilde{z}_i^\tau(\mathbf{X}_i)$ be agent a_i's marginal contribution function at iteration τ of the min-max algorithm. Then, $\tilde{z} = \max_{a_i \in A} \min_{\mathbf{X}_i} \tilde{z}_i^\tau(\mathbf{X}_i)$ is a lower bound on the value of the optimal solution, namely $\tilde{z} \leq \min_{\mathbf{X}} \Psi(\mathbf{X})$, where $\Psi(\mathbf{X}) = \max_{a_i \in A} \psi_i(\mathbf{X}_i)$.*

Proof. Since the optimal solution of two equivalent functions is the same, the result follows directly from Lemmas 1 and 2. □

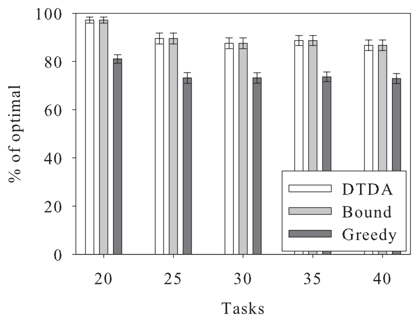
Therefore, under Theorem 2, at each iteration of the min-max algorithm, the maximum of the agents' marginal contributions, $\tilde{z} = \max_{a_i \in A} \min_{\mathbf{X}_i} \tilde{z}_i^\tau(\mathbf{X}_i)$, is a lower bound on the value of the optimal solution. Notice that, at each iteration, the agents' marginal contribution functions combine information from the messages using the max operator, so $\min_{\mathbf{X}_i} \tilde{z}_i^\tau(\mathbf{X}_i) \leq \min_{\mathbf{X}_i} \tilde{z}_i^{\tau+1}(\mathbf{X}_i)$. Therefore, the sequence of lower bounds is guaranteed to monotonically increase over iterations of min-max, thus providing a better approximation of the value of the optimal solution at each iteration. As shown in section 5.2, agents can, at the end of the min-max algorithm, assess this lower bound value to bound the error of the approximate solution found when running the min-max algorithm.

In the next section, we present our empirical evaluation of the DTDA. It is necessary for us to do this to show our algorithm finds good solutions, as the bound we provide on the quality of the approximations we give is per-instance, as opposed to an overall offline bound.

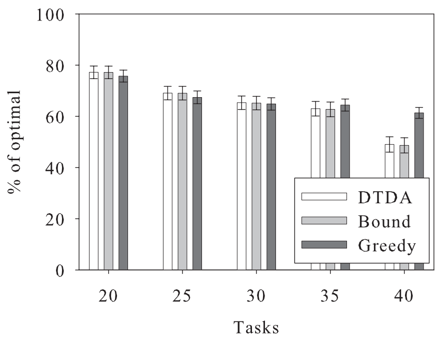
6 Empirical Evaluation

In this section, we compare the approximation found by the DTDA to a number of other algorithms, thus establishing the first decentralised benchmark for $R||C_{\max}$. Namely, we compare the DTDA against an optimal centralised algorithm, and a greedy algorithm. In more detail, the *optimal centralised algorithm (CA)* operates by solving a mixed integer program to find the optimal solution. We formulate the problem as a binary integer program, and then use IBM ILOG CPLEX[2] to find an optimal solution assigning tasks to agents. Next, in the *global greedy algorithm (Greedy)*, tasks are allocated to the agent that can complete them the fastest, and are considered in order of time required, from highest to lowest. In both these cases, we consider exactly the same problem the DTDA does — i.e., each agent can only perform a subset of the tasks. In addition, we plot the maximum bound found at an agent after executing the DTDA, ρ, as

[2] See `www.ibm.com/software/integration/optimization/cplex-optimizer/`



(a) $\sigma_t = 2$



(b) $\sigma_t = 3$

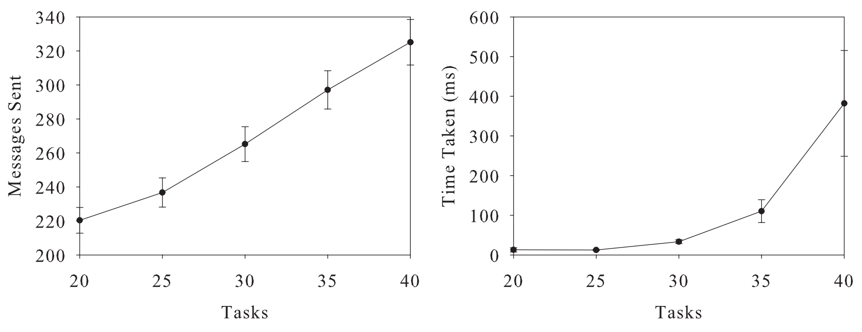
Fig. 4. Empirical Results: Utility gained for varying graph density.

computed in the value propagation phase, found in Section 5.2. Note that we do not compare to any existing approximate algorithms for $R||C_{\max}$ because, as we said earlier, there exist *no* decentralised algorithms for $R||C_{\max}$. Hence, our result establishes the first communication and computation benchmark for distributing the solution of $R||C_{\max}$ problems.

To evaluate the performance of DTDA, we plot the solutions obtained as a mean percentage of the optimal centralised solution, with error bars representing 95% confidence intervals in the mean. We calculate the mean approximation ratio of solutions obtained by each of these algorithms by dividing the achieved makespan by the optimal makespan (i.e., those obtained by CA), over 100 random scenarios, and use this to plot the percentage of the optimal obtained. In addition, we plot the mean total number of messages sent by DTDA, and the mean time taken to find a solution by DTDA.

We compare our algorithms in a number of average cases: specifically, sparse random graphs, and dense random graphs. In more detail, we generated 500 random scenarios with $|A| = 20$, $|T| = \{20, 25, 30, 25, 40\}$, and $\sigma_t \in \{2, 3\}$, where σ_t is the average degree of each task. The time taken for agent a_i to perform task t_j was calculated as $c_i \times c_j$, where $c_i \in \{1, \ldots, 100\}$ and $c_j \in \{0.1, \ldots, 1.1\}$, where c_i and c_j are both taken from uniform distributions. We present the utility results of these experiments in Figure 4, and the communication and computation results in Figure 5.

Figure 4(a) shows the performance of the DTDA versus greedy in a sparse environment, where each task can, on average, only be performed by two agents. Conversely, in Figure 4(b), we have the performance of DTDA versus greedy in a more dense environment, with an average of three agents being able to perform each task. The DTDA clearly outperforms greedy in the sparse graph by up to 16%; however, in the more dense graph, it is clear that the performance of DTDA does not warrant its application over greedy in this case. This shows that the DTDA is best applied on sparse graphs, as we intended, and is consistent with



(a) Messages Sent.

(b) Computation Time.

Fig. 5. Empirical Results: Communication and Computation used where $\sigma_t = 2$.

other GDL algorithms [1]. Nevertheless, the DTDA's performance ranges from 97% of the optimal to 86% in the sparse graphs. In addition, the graphs show that the bound produced by the DTDA provides a very accurate estimation of the solution gained by the DTDA — so much so, that the two lines on the graph are barely distinguishable. Finally, in terms of communication and computation, Figure 5 (a) shows that the number of messages sent by DTDA increases almost linearly as the number of tasks increases. In contrast, Figure 5 (b) shows that the computation time increases exponentially in the number of tasks. Note that in Figure 5 we only plot results for $\sigma_t = 2$, as $\sigma_t = 3$ gave similar results.

7 Conclusions and Future Work

We have presented the first decentralised algorithm for finding solutions to the scheduling on unrelated parallel machines problem, known as $R||C_{\max}$. Our algorithm (DTDA) is also the first known application of the min-max algorithm to solve $R||C_{\max}$ in the literature. In addition, we are able to provide a per-instance bound on the quality of the solutions given, online. Empirically, we showed that the bound we find provides an accurate estimation of the global solution value, that the communication required by the DTDA scales linearly in the size of the environment, and that DTDA is able to find good quality solutions in environments which can be formulated as a sparse graph (from 97–86% of the optimal). In addition, we drew the parallel between $R||C_{\max}$ and multi-agent task allocation problems. However, we found that the DTDA holds no advantage over a greedy algorithm in more dense environments, partly because the state space explored at each agent in DTDA grows exponentially, and partly because the approximation given by using the min-max algorithm is not of high enough quality. While using the algorithm makes sense in task allocation environments where an agent only considers a limited number of tasks, the computation needed scales exponentially in the size of the environment. Therefore, future work will focus on reducing the state space at each agent (e.g., by using techniques such as branch and bound), using spanning trees to improve solution quality on denser graphs, so that we can successfully apply DTDA to a wider range of problems, and evaluating DTDA's performance on other graph topologies, such as scale-free graphs.

References

1. Aji, S.M., McEliece, R.J.: The generalized distributive law. IEEE Transactions on Information Theory 46(2), 325–343 (2000)
2. Bistarelli, S., Gennari, R., Rossi, F.: Constraint propagation for soft constraints: Generalization and termination conditions. In: CP. pp. 83–97 (2000)
3. Farinelli, A., Rogers, A., Petcu, A., Jennings, N.R.: Decentralised coordination of low-power embedded devices using the max-sum algorithm. In: Proc. AAMAS–08. pp. 639–646 (2008)
4. Graham, R.L., Lawler, E.L., Lenstra, J.K., Rinnoy Kan, A.H.G.: Optimization and approximation in deterministic sequencing and scheduling: a survey. Annals of Discrete Mathematics 5, 287–326 (1979)
5. Horowitz, E., Sahni, S.: Exact and approximate algorithms for scheduling nonidentical processors. Journal of the ACM 23, 317–327 (April 1976)
6. Ibarra, O.H., Kim, C.E.: Heuristic algorithms for scheduling independent tasks on nonidentical processors. Journal of the ACM 24(2), 280–289 (1977)
7. Jensen, F.V.: An Introduction to Bayesian Networks. Springer-Verlag (1996)
8. Lenstra, J.K., Shmoys, D.B., Tardos, E.: Approximation algorithms for scheduling unrelated parallel machines. Mathematical Programming 46, 259–271 (1990)
9. Potts, C.N., Strusevich, V.A.: Fifty years of scheduling: A survey of milestones. Journal of the Operational Research Society 60, 41–68 (2009)
10. Ramchurn, S.D., Farinelli, A., Macarthur, K.S., Jennings, N.R.: Decentralised Coordination in RobocupRescue. The Computer Journal 53(9), 1447–1461 (2010)
11. Rossi, F., Beek, P.v., Walsh, T.: Handbook of Constraint Programming (Foundations of Artificial Intelligence). Elsevier Science Inc., New York, NY, USA (2006)
12. Stranders, R., Farinelli, A., Rogers, A., Jennings, N.: Decentralised coordination of mobile sensors using the max-sum algorithm. In: Proc. IJCAI–09. pp. 299–304 (2009)