Speeding Up GDL-based Message Passing Algorithms for Large-Scale DCOPs

Md. Mosaddek Khan, Sarvapali D. Ramchurn
Long Tran-Thanh and Nicholas R. Jennings

School of Electronics and Computer Science,
University of Southampton, UK
{mmk1g14, sdr, ltt08r, nrj}@ecs.soton.ac.uk

Abstract. This paper develops a new way of speeding up the GDL-based message passing algorithms that solve large-scale Distributed Constraint Optimization Problems (DCOPs) in multi-agent systems. In particular, we minimize the computation and communication costs in terms of time for algorithms such as Max-Sum, Bounded Max-Sum, Fast Max-Sum, Bounded Fast Max-Sum, and BnB Max-Sum. To do so, our approach contains three major components: splitting the original DCOP representation into clusters, efficiently distributing the computation/communication overhead among agents and a time efficient domain pruning technique. We empirically evaluate the performance of our proposed method in different settings and find that it brings down the completion time by around 45% for $100-600$ nodes and by up to around 60% for $3000-10000$ nodes compared to the standard approaches that are currently used in DCOPs.

1 Introduction

Distributed Constraint Optimization Problems (DCOPs) are a widely studied framework for solving constraint handling problems of co-operative multi-agent systems (MAS) [Modi *et al.*2005]. They have been applied to many real world applications such as disaster response, sensor networks, traffic control in the form of task allocation [Ramchurn *et al.*2010,Zivan *et al.*2014], meeting scheduling [Maheswaran *et al.*2004b] and coalition formation [Cerquides *et al.*2013]. In DCOPs, such problems are formulated as constraint networks that are often represented graphically. In particular, the agents are represented as nodes, and the constraints that arise between the agents depending on their joint choice of action are represented by the edges. Each constraint can be defined by a set of variables held by the corresponding agents related to that constraint. In more detail, each agent holds one or more variables, each of which takes values from a finite domain. The agent is responsible for only setting the value of its own variable(s) but can communicate with other agents. The goal of a DCOP algorithm is to set every variable to a value from its domain to minimize the constraint violation.

Over the last decade, a number of algorithms have been developed to solve DCOPs under two broad categories: exact and non-exact algorithms. The former always find an optimal solution. However, finding an optimal solution for a DCOP is an NP-hard problem so they exhibit an exponentially increasing coordination overhead as the system gets larger (e.g. ADOPT [Modi *et al.*2005], BnB ADOPT [Yeoh *et al.*2008]), DPOP

[Petcu2005]). On the other hand, non-exact algorithms sacrifice some solution quality, but scale well compared to the exact algorithms. These algorithms are further categorized into greedy and Generalized Distributive Law (GDL) based inference methods. Greedy approaches (e.g. DSA [Fitzpatrick2003], MGM [Maheswaran *et al.*2004a]) rely on local greedy moves which often come up with a solution far from the optimal one for large and complex problems. On the other hand, GDL-based inference algorithms perform well in practical applications and provide optimal solutions for cycle free constraint graphs (e.g. Max-Sum [Farinelli *et al.*2008]) and acceptable solutions for a cyclic one (e.g. Bounded Max-Sum [Rogers *et al.*2011]). Notably, these algorithms make efficient use of constrained computational and communication resources and effectively represent and communicate complex utility relationships through the network. Nevertheless, due to the following reason scalability remains an open issue for such GDL-based algorithms. Specifically, they perform repetitive maximization operations for each constraint to select the locally best configuration of the associated variables given the local function and a set of incoming messages. To be precise, a constraint that depends on n variables having domains composed of d values each, will need to perform d^n computations for a maximization operation. As the system scales up, the complexity of this step grows exponentially and makes Max-Sum/Bounded Max-Sum too expensive in terms of computational cost. While several attempts have been sought to reduce the cost of the maximization operation, they typically limit the applicably of Max-Sum/Bounded Max-Sum within few problem domains [Ramchurn *et al.*2010,Macarthur *et al.*2011,Zivan *et al.*2015] and others rely on a preprocessing step, thus denying the opportunity of obtaining local utilities at runtime [Stranders *et al.*2009,Kim and Lesser2013].

More specifically, previous attempts at scaling up GDL-based algorithms have mainly focused on reducing the overall cost of the maximization operator mentioned earlier. However, they overlook an important concern that all the GDL-based DCOP algorithms follow a Standard Message Passing (SMP) protocol where *a message is sent from a node v on an edge e if and only if all messages received at v on edges other than e* [Aji and McEliece2000]. This dependency produces increasing amounts of average waiting time for agents as the graphical representation of a DCOP becomes larger. As a consequence, the required time to obtain the solution from the DCOP algorithm (i.e. completion time) increases, which makes it unusable in practice for a large real world problem.

To date, this challenge has not been addressed by the DCOP community. As such, this paper aims to fill this gap by proposing a new general framework (Parallel Message Passing- PMP) that can be applied to the existing GDL-based message passing algorithms to obtain the same overall result but in significantly less time. Here, we consider both the computation and communication cost of an algorithm in terms of time. Thus, we reduce the completion time of a GDL-based algorithm by replacing its SMP by PMP while maintaining the same outcome in terms of solution quality. It is noteworthy that the GDL-based algorithms which deal with cyclic graphical representations of a DCOP (e.g. Bounded Max-Sum, Bounded Fast Max-Sum) initially remove the cycle from the original constraint graph, then apply the SMP protocol on the acyclic graph to provide a bounded approximate solution of the problem. In this paper, we reduce the completion time of the standard message passing procedure. Therefore, our framework can be applied on cyclic DCOP by using the same tree formation technique used by Bounded Max-Sum. Afterwards, PMP can be applied instead of SMP on the transformed acyclic graph.

Our framework consists of following major contributions. First, PMP provides same result compared to its SMP counterpart within less time. Here, we do not change the computation method of the messages, rather we efficiently distribute the overhead of message passing to the agents to exploit their computational power concurrently. Thus, we reduce the average waiting time of the agents. To do so, we split the graphical representation of a DCOP into several parts (clusters) and execute message passing on them in parallel. As a consequence of this cluster formation we have to ignore inter cluster links. Therefore, PMP requires two rounds of message passing and an intermediate step to recover the values of the ignored links. Second, to further reduce the required time of expensive intermediate step we introduce a domain pruning algorithm. By doing so, we advance the state of the art in the following ways:

1. We introduce a new cluster based framework, Parallel Message Passing (PMP), which efficiently distributes the computational and communicational overhead to the agents to exploit their computational power concurrently and reduce their average waiting time before sending messages to solve large-scale DCOPs.
2. We empirically evaluated the performance of our framework in terms of completion time and compare it with the benchmark algorithms that follow the SMP protocol in different settings (up to 10,000 of nodes). Our results show a speed up of 45% to 60% with no reduction in solution quality.

2 Problem Formulation

Even though, PMP is a general framework that can be applied to any DCOP related application domain, we opt for a model that illustrates its application in the task allocation perspective. Assume, we have a set of discrete variables, $X = \{x_0, x_1, \dots, x_m\}$, each variable x_i represents a geographic location where an agent needs to be allocated to do a task(s). Then, $F = \{F_1, F_2, \dots, F_n\}$ is a set of functions/factors and each of the them represents each task. Here, each task F_i might need to be performed by more than one agent. In this formulation, each agent can do multiple tasks in a single allocation. For example, an information gathering agent might gain situation awareness by observing two different locations or a rescue agent could perform two different tasks one after the other based on their priorities. Moreover, $D = \{D_0, D_1, \dots, D_m\}$ is a set of discrete and finite variable domains, each variable x_i can take the value from the domain D_i. To be exact, the values in the domain D_i is the set of potential agents, $A = \{A_0, A_1, \dots, A_{k_i}\}$ to be deployed to the location represented by the variable x_i. By definition, functions have a relation with variables by means of locality, meaning to accomplish a task (represented by a function) one or more agents need to be deployed in the location(s) represented by the variable(s). These dependencies generate a bipartite graph, namely factor graph which is commonly used as a graphical representation (constraint networks) of such DCOPs [Kschischang *et al.*2001]. Within this model, the target is to find the value (agents' allocation to tasks) of each variable, x^*, such that all the tasks (functions) in the system can be performed in the best possible way.

$$x^* = \underset{x}{argmax} \sum_{i=1}^{n} F_i(\mathbf{x}_i) \tag{1}$$

For instance, Figure 1 depicts the relationship among variables, functions and agents as well as a possible outcome of the DCOP (task allocation) problem. Initially, agent $A1$ takes responsibilities of task1 represented by the factor F_1 and the variable x_0. Similarly, task2 (F_2), variables x_1 and x_2 are held by agent $A2$. Now, only these two agents participate in the optimization process instead of all available agents. Therefore, unlike traditional DCOP formulation where each agent must take the responsibility of at least one node (variable/function) of the factor graph, this formulation permits us not to involve all the agents take this responsibility. Thus, we can avoid unnecessary agents' involvement in the message passing procedure. Nonetheless, each agent participates in the optimization process as a domain value for certain variables. At the end of the process, the location represented by the variable x_0 can be occupied by any of the following agents $\{A1, A2, A3, A4, A5\}$. Similarly, the sets of agents $\{A1, A2, A3, A6, A7\}$ and $\{A4, A2, A5, A6, A7\}$ define the domain of the variables x_1 and x_2 respectively. The ultimate goal is to find the appropriate agent from the domain of a particular variable to be employed in a position conforming to that variable to do the corresponding task represented by the factor connected to it. Here, the bottom part of Figure 1 presents a sample scenario of the final result of the optimization process where agents $A2, A1$ and $A5$ are going to be placed at the location mapped by the variables x_0, x_1, and x_2 respectively. Specifically, Task1 should be done by agent $A2$ and $A1$, similarly with the help of agent $A1$, $A5$ will accomplish task2.

3 The Parallel Message Passing Framework

All GDL-based DCOP algorithms follow the SMP Protocol and this adds a substantial delay to converge for large-scale systems. To address this, we introduce a new framework (Parallel Message Passing-PMP). PMP is a general framework that can be applied to any GDL-based DCOP formulation. Notably, both Max-Sum and Bounded Max-Sum use Equations 2, 3 and 4 for their message passing and they can be directly applied to the factor graph representation of a DCOP. Here, the variable and function nodes of the factor graph continuously exchange messages (*variable (x_i) to function (F_j)* (Equation 2), *function (F_j) to variable (x_i)* (Equation 3)) to compute an approximation of the impact that each of the agent's actions has on the global objective function by building a local objective function $Z_i(x_i)$. Here, M_i is the set of functions connected to x_i and N_j represents the set of variables connected to F_j. Once the function is built (Equation 4), each agent picks the value of a variable that maximizes the function by finding $\arg\max_{x_i}(Z_i(x_i))$. Now, even though some extensions of Max-Sum algorithm (e.g. Fast Max-Sum [Ramchurn *et al.*2010], BnB FMS [Macarthur *et al.*2011]) modify these equations, our framework can still be applied to those algorithms as we are not

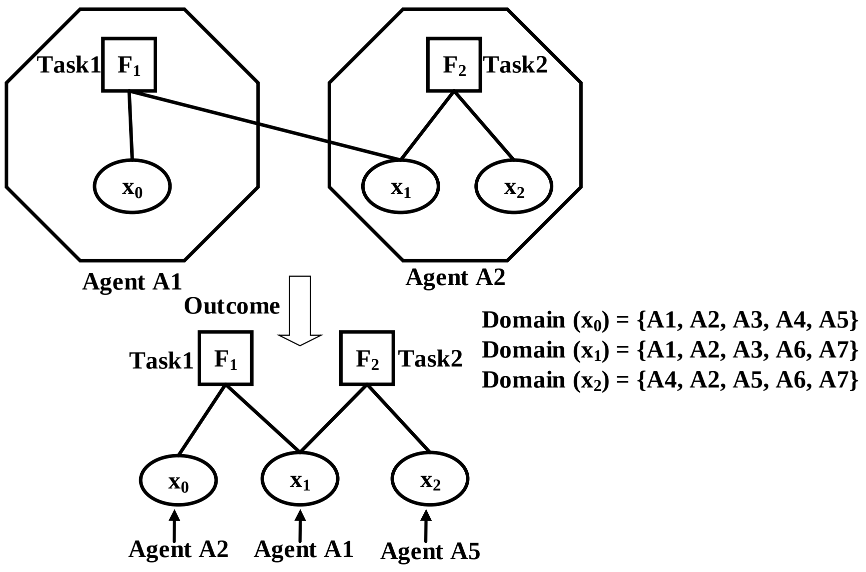

Task1
F1
x0
Agent A1
F2
Task2
x1
x2
Agent A2
Outcome
Domain (x0) = {A1, A2, A3, A4, A5}
Domain (x1) = {A1, A2, A3, A6, A7}
Domain (x2) = {A4, A2, A5, A6, A7}
Task1
F1
F2
Task2
x0
x1
x2
Agent A2
Agent A1
Agent A5

Fig. 1: A sample factor graph shows a possible end result of the task allocation problem, a DCOP as well as the relationship among variables, factors and agents.

altering any equation of the original algorithm.

$$Q_{i \to j}(x_i) = \sum_{k \epsilon M_i \backslash j} R_{k \to i}(x_i) \tag{2}$$

$$R_{j \to i}(x_i) = \max_{\mathbf{x}_j \backslash i}[F_j(\mathbf{x}_j) + \sum_{k \epsilon N_j \backslash i} Q_{k \to j}(x_i)] \tag{3}$$

$$Z_i(x_i) = \sum_{j \epsilon M_i} R_{j \to i}(x_i) \tag{4}$$

Definition 1. *(Cluster). A cluster c_i is a subgraph of a factor graph F_G preserving the following property: two clusters c_i and c_j are neighbours if and only if they share a common variable node (i.e. split node) x_{ij}.*

Definition 2. *(Ignored Values of a Cluster, $ignVal(c_i)$). To achieve the similar solution quality compared to the algorithms that follow the SMP protocol, it is necessary to recover the incoming messages (through the split nodes) overlooked during the formation of clusters. The intermediate step of PMP takes the responsibility of finding those missing values named as $ignVal(c_i)$ for each cluster c_i.*

Definition 3. *(Dependent Acyclic Graph, $\mathcal{D}_{\mathcal{G}}(\mathcal{I}_j)$). A $\mathcal{D}_{\mathcal{G}}(\mathcal{I}_j)$ is an acyclic directed graph from the furthest node of the factor graph F_G from a split node x_{ij} towards it. During the intermediate step, synchronous operations are performed at the edges of this graph in the same direction to compute each ignored value of a cluster, $ignVal(c_i)$.*

Algorithm 1: Parallel Message Passing

```
Data: A factor graph, $F_G$ consists a set of variables, $X = \{x_1, x_2, \ldots, x_m\}$ and a set of
      functions $F = \{F_1, F_2, \ldots, F_n\}$
1  $\mathcal{N_F} \leftarrow |F|$
2  $\mathbb{N} \leftarrow \mathcal{N_F}/\mathcal{N_C}$
3  $firstVar \leftarrow min\_deg(X)$
4  $node \leftarrow firstVar.Function$
5  for $i \leftarrow 1$ to $\mathcal{N_C}$ do                              // Cluster formation
6      $count \leftarrow 0$
7      while $count < \mathbb{N}$ do        // distribute the nodes to each cluster
8          $c_i.member() \leftarrow node$
9          $c_i.member() \leftarrow adj(node)$
10         $node \leftarrow adj(adj(node))$
11         $count \leftarrow count + 2$
12     $c_i.member() \leftarrow node$
13 foreach $cluster\ c_i \in \mathcal{S_N}$ in $\mathbb{PARALLEL}$ do           // First round
     of message passing
14     $\forall Q(c_i) \leftarrow \emptyset$
15     $\forall R(c_i) \leftarrow \emptyset$
16     $Max - Sum(c_i)$: Message Passing Only              // Equation 2 and
        Equation 3
17 for $i \leftarrow 1$ to $\mathcal{N_C}$ in $\mathbb{PARALLEL}$ do              // Intermediate step
     of PMP
18     $ignVal(c_i) \leftarrow intermediateStep(c_i)$
19 for $i \leftarrow 1$ to $\mathcal{N_C}$ in $\mathbb{PARALLEL}$ do                // Second round of
     message passing
20     $(\forall Q(c_i) \setminus Q_{ignEdge}(c_i)) \leftarrow \emptyset$
21     $Q_{ignEdge}(c_i) \leftarrow ignVal(c_i)$
22     $\forall R(c_i) \leftarrow \emptyset$
23     $Max - Sum(c_i)$: Complete          // Equation 2, Equation 3 and
        Equation 4
```

Now, PMP (Algorithm 1) uses a similar means to compute messages as its SMP counterpart. For example, Max-Sum messages are used when our framework is applied to the Max-Sum algorithm. Similarly, Fast Max-Sum messages are used when applied to the Fast Max-Sum. Even so, PMP reduces the completion time by splitting the factor graph into clusters (Definition 1) and independently running the message passing on those clusters in parallel. As PMP ignores inter cluster links (i.e. messages) during the formation of cluster, it is not possible to obtain the similar solution quality as the original algorithm by executing only one round of message passing. This is why PMP includes two rounds of message passing with the addition of an intermediate step. The role of the intermediate step is to generate the ignored values (Definition 2) for the split node(s) of a cluster so that the second round can use these as initial values for the split node(s) to generate the same messages as the original algorithm.

Cluster Formation: PMP operates on a factor graph F_G of a set of variables X and a set of functions F. Specifically, lines $1 - 14$ of Algorithm 1 generate $\mathcal{N_C}$ *clusters* by

Algorithm 2: intermediateStep(Cluster c_i)

```
Input: A set of clusters, $\mathcal{C} = \{\mathfrak{c}_1, \mathfrak{c}_2, \ldots, \mathfrak{c}_\mathfrak{r}\}$ with their corresponding Cluster Heads,
       $\mathcal{CH} = \{\mathfrak{ch}_1, \mathfrak{ch}_2, \ldots, \mathfrak{ch}_\mathfrak{r}\}$
Output: Ignored values, $iVal(\mathfrak{c}_i)$ of dependent edges, $\mathcal{I} = \{\mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_k\}$ for the
        cluster $\mathfrak{c}_i$
1  $\mathfrak{ch}_i \leftarrow StatusMessage(\forall\mathcal{CH} \setminus \mathfrak{ch}_i) \wedge M_r$   // required utility and messages received by $\mathfrak{ch}_i$
2  $\mathfrak{l} \leftarrow |\mathcal{I}|$
3  for $j \leftarrow 1$ to $\mathfrak{l}$ do
4      $\mathfrak{c}_\mathfrak{p} \leftarrow adjCluster(\mathfrak{c}_i, \mathcal{I}_j)$
5      $dCount_{c_p} \leftarrow totalAdjCluster(\mathfrak{c}_\mathfrak{p})$
6      if $dCount_{c_p} == 1$ then   // Cluster having only one neighbour cluster
7          $\mathcal{D}_G(\mathcal{I}_j) \leftarrow READY\_\mathcal{D}_G(\mathcal{I}_j)$
8      else if $dCount_{c_p} > 1$ then   // Cluster having more than one neighbour cluster
9          if $\mathcal{I}_j.node_p \in c_p$ then
10             $dNode \leftarrow \mathcal{I}_j.node_p$
11         while $dNode \neq \emptyset$ do   // Formation of dependent acyclic graph
12             $\mathcal{D}_G(\mathcal{I}_j) \leftarrow dNode$
13             $dNode \leftarrow adj(dNode)$
14         if $\{F_{n_1}, F_{n_2}, \ldots, F_{n_g}\} \in \mathcal{D}_G(\mathcal{I}_j)$ then
15             $\mathcal{I}_j.values \leftarrow sync(F_{n_1} \oplus F_{n_2} \oplus \ldots \oplus F_{n_g})$// Operations on $\mathcal{D}_G(\mathcal{I}_j)$
16             $iVal(\mathfrak{c}_i) \leftarrow \mathcal{I}_j.values$
17 return $iVal(\mathfrak{c}_i)$
```

splitting F_G. Line 3 gets a special variable node ($firstVar$), which is a variable having minimum degree and related to only one function ($min\_deg(X)$). Then, line 4 initializes the node (function) which is connected to $firstVar$. The *for* loop of lines $5 - 14$ iteratively adds the member nodes to each cluster c_i. The *while* loop (lines $7 - 12$) iterates as long as the member count for a cluster c_i remains less than the maximum nodes per cluster ($\mathbb{N}$). For instance, in Figure 2 original factor graph is divided into 3 clusters: c_1, c_2, c_3. The *for* loop in lines $13 - 16$ acts as the **first round** of message passing, which involves only computing and sending the *variable to function* (Equation 2) and *function to variable* (Equation 3) messages within those clusters having only single neighbouring cluster (only c_1 and c_2 in Figure 2) in parallel. Unlike the first round, all the clusters participate in the **second round** of message passing in parallel (lines $19 - 23$). In the second round, instead of using the null values for initializing all *variable to function* messages, here in line 21 we exploit recovered ignored values (Definition 2) from the intermediate step (lines $17 - 18$) as initial values for split variable nodes and the rest of the messages are initialized as null (lines $20, 22$). We briefly explain this intermediate step of PMP shortly. Finally, PMP will converge with Equation 4 by computing the value $Z_i(x_i)$ and hence finding $\arg\max_{x_i}(Z_i(x_i))$.

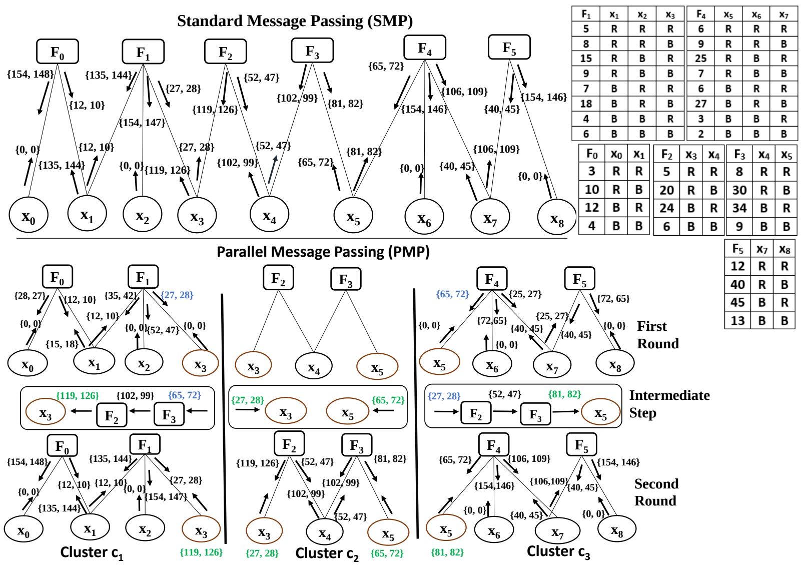
Detail of Intermediate Step: A key part of PMP is the *intermediate step* (Algorithm 2) which takes a cluster (c_i) provided by line 18 of Algorithm 1 as an input and returns ignored values (Definition 2) for each of the ignored links of that cluster. A representative of each cluster $\mathfrak{c}_i$ (cluster head $\mathfrak{ch}_i$) performs the operation of intermediate

Algorithm 3: Domain pruning to compute $D_{F_j \to F_p}(x_i)$ in intermediate step of PMP

```
Input: Local utility of factor $F_j(\mathbf{x}_j)$: sorted independently by each states of the domain
       of $\mathbf{x}_j$; Incoming messages from the neighbour(s) of $F_j$ other than $F_p$.
Output: Range of values of the states over which maximization will be computed.
1  $Let,\ \{\mathbf{d_1}, \mathbf{d_2}, \dots, \mathbf{d_r}\}\ be\ the\ states\ of\ the\ domain$
2  $m \leftarrow \sum_{k \in (C_j \setminus F_p)} max(D_{F_k \to F_j}(x_t))$
3  for $i \leftarrow 1$ to r do                    // for each states of the domain
4      $p \leftarrow max_{\mathbf{d_i}}(F_j(\mathbf{x}_j))$
5      $b \leftarrow \sum_{k \in (C_j \setminus F_p)} val_p(D_{F_k \to F_j}(x_t))$
6      $t \leftarrow m - b$
7      $s \leftarrow getVal()$          // pick a value from $\mathbf{d_i}$ less than $p$
8      if $t \geq p - s$ then
9          return $valueRange_{\mathbf{d_i}}[p, s)$
10     else
11         goto Line 7
```

step. Initially, each cluster head needs to receive the *StatusMessages* from rest of the cluster heads (line 1 of Algorithm 2). Each *StatusMessage* contains the factor graph structure along with the utility information. A cluster head also requires a factor to split variable message from each of the participated clusters of the first round (M_r). Notably, *StatusMessages* can be formed and exchanged during the time of the first round, thus it does not incur additional delay. The *for* loop in lines $3-16$ computes the ignored values for each split node of the cluster c_i by generating a Dependent Acyclic Graph, $\mathcal{D}_{\mathcal{G}}(\mathcal{I}_j)$ (Definition 3). If a neighbouring cluster related to a split node has no other cluster(s) to depend on then there is no need for further computation as the ignored value is immediately ready after the first round (lines 6-7). On the other hand, if it has other clusters to rely on then further calculations in the graph are required, so need to find each node of that graph $\mathcal{D}_{\mathcal{G}}(\mathcal{I}_j)$ (lines 8-13). Line 10 initializes the first node of $\mathcal{D}_{\mathcal{G}}(\mathcal{I}_j)$. The *while* loop (lines $11-13$) repeatedly forms that graph through extracting the adjacent nodes from the first selected node, $dNode$. Finally, synchronous executions from the farthest node to the start node (split node) of $\mathcal{D}_{\mathcal{G}}(\mathcal{I}_j)$ produce the desired value, $\mathcal{I}_j.values$ for the edge, $\mathcal{I}_j$ (line 15), which is eventually the required value for that split node of the cluster c_i, which will be used as initial value during the second round of message passing. For example, x_3 is a split node for cluster c_1 in Figure 2 and $\{119, 126\}$ is the ignored value computed by the intermediate step. Now, instead of $\{0, 0\}$ the second round uses $\{119, 126\}$ as initial value for node x_3.

During intermediate step, the entire operation is performed on a single machine (**cluster head**) for each cluster. Therefore, apart from receiving the M_r values which is literally a message from the participating clusters of first round, there is no communication cost in this step. This produces a significant reduction of communication cost (time) in PMP. Moreover, Figure 3 shows how we can avoid the computation of *variable to factor* messages in the intermediate step as they are redundant and produce no further significance in this step. For instance, it is redundant to compute *variable to factor* messages $(x_4 \to F_8, x_1 \to F_8, x_2 \to F_8, x_3 \to F_8)$ during the computation of the message



F_1	x_1	x_2	x_3
5	R	R	R
8	R	R	B
15	R	B	R
9	R	B	B
7	B	R	R
18	B	R	B
4	B	B	R
6	B	B	B

F_4	x_5	x_6	x_7
6	R	R	R
9	R	R	B
25	R	B	R
7	R	B	B
6	B	R	R
27	B	R	B
3	B	B	R
2	B	B	B

F_0	x_0	x_1
3	R	R
10	R	B
12	B	R
4	B	B

F_2	x_3	x_4
5	R	R
20	R	B
24	B	R
6	B	B

F_3	x_4	x_5
8	R	R
30	R	B
34	B	R
9	B	B

F_5	x_7	x_8
12	R	R
40	R	B
45	B	R
13	B	B

Fig. 2: Worked example of SMP and PMP (participating clusters: first round- (c_1, c_2), second round- (c_1, c_2, c_3)).

$F_8 \rightarrow x_0$ (i.e. $F_8 \rightarrow F_7$) in the intermediate step of PMP. However, each synchronous execution within the $\mathcal{D}_{\mathcal{G}}(\mathcal{I}_j)$ is as expensive as a *factor to variable* message and can be computed using the Equation 5. This equation retains similar properties as Equation 3 but the receiving node is a function node instead of a variable node. Here, C_j denotes the set of indexes of the functions connected to function F_j in the Dependent Acyclic Graph, $\mathcal{D}_{\mathcal{G}}(\mathcal{I}_j)$ of intermediate step and variable $x_t \in \{adj(F_k) \wedge adj(F_j)\}$.

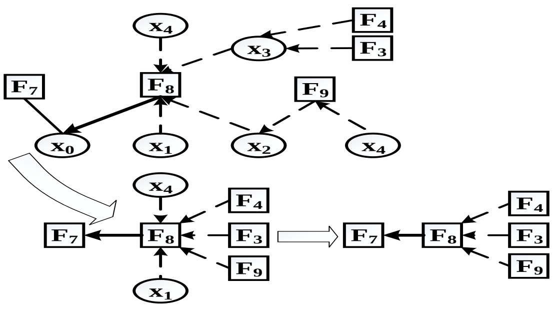


Fig. 3: Single Computation within intermediate step.

$$D_{F_j \rightarrow F_p}(x_i) = \max_{\mathbf{x}_j \setminus x_i} [F_j(\mathbf{x}_j) + \sum_{k \in C_j \setminus F_p} D_{F_k \rightarrow F_j}(x_t)] \tag{5}$$

Domain Size Reduction During Intermediate Step: Note that Equation 5 still requires a significant amount of computation due to the potentially large parameter domain size. Given this, in order to further improve the computational efficiency, we propose a novel algorithm to reduce the domain size over which maximization needs to be computed (Algorithm 3). This algorithm requires incoming messages from the neighbour(s) of a function in $\mathcal{D}_{\mathcal{G}}(\mathcal{I}_j)$ and each local utility be sorted independently by each state of the domain. Specifically, this sorting can be done before computing the *StatusMessage* during the time of the first round. Therefore it does not incur an additional delay. The *for* loop in lines $3-11$ generates the range of the value for which we will always find the maximum value and discards the rest. In more detail, we generate a base case (t) by subtracting the addition of corresponding values for the maximum of the a state (p) from the addition of maximum values of the incoming message(s) (lines 4-5). Then, a value s is picked from the sorted list of that state which is less than p (line 7). Finally, if it is greater than or equal to $p-s$ then the desired maximization must be found within the range of $[p, s)$ otherwise we need to pick smaller value of s and repeat the checking (lines 8-11). Note that, PMP itself reduces completion time significantly. Thus, only if the maximization operator has to deal with a large domain size then Algorithm 3 should be used alongside.

4 Empirical Evaluation

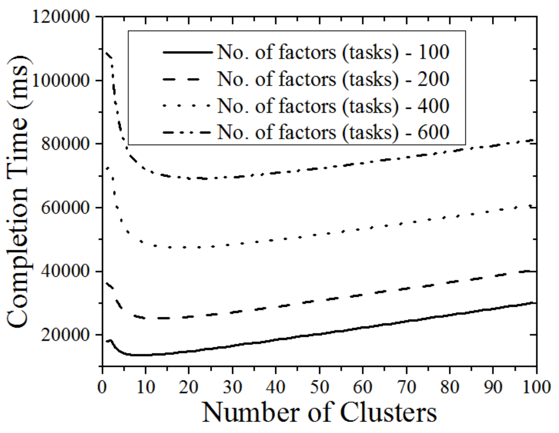


Fig. 4: Completion Time: Standard Message Passing (No. of Cluster=1); Parallel Message Passing (No. of Cluster>1)). Number of Tasks: $100-600$ (Without domain pruning).

We now evaluate the performance of PMP to show how effective it is in terms of completion time compared to the benchmarking algorithms that follow SMP protocol. We generated different instances of the task allocation problem that have varying numbers of tasks $100-10,000$. In our settings, we consider those tasks as function nodes

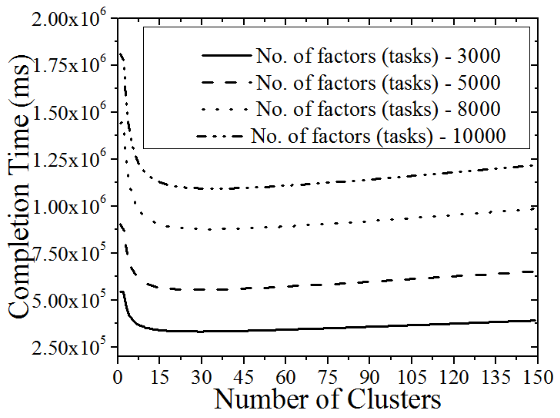


Fig. 5: Completion Time: Standard Message Passing (No. of Cluster=1); Parallel Message Passing (No. of Cluster>1)). Number of Tasks: $3000 - 10000$ (Without domain pruning).

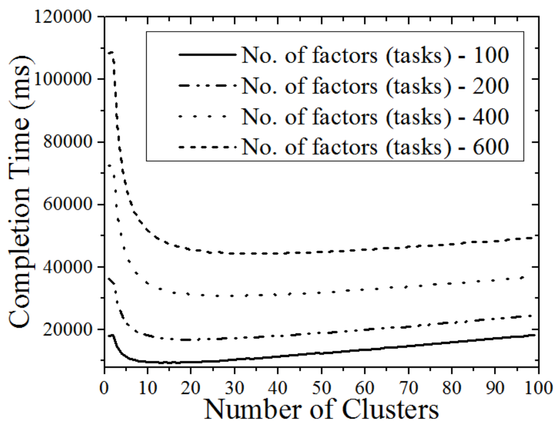


Fig. 6: Completion Time: Standard Message Passing (No. of Cluster=1); Parallel Message Passing (No. of Cluster>1)). Number of Tasks: $100 - 600$ (With domain pruning).

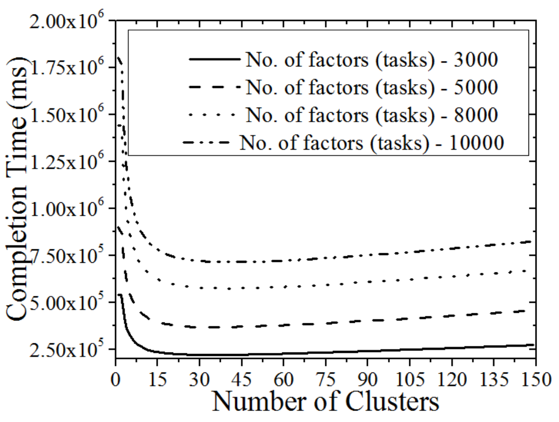


Fig. 7: Completion Time: Standard Message Passing (No. of Cluster=1); Parallel Message Passing (No. of Cluster>1)). Number of Tasks: $3000 - 10000$ (With domain pruning).

and randomly generating factor graphs by connecting variable nodes to them. We consider the number of clusters $2 - 99$ for the factor graph of $100 - 600$ function nodes and $2 - 149$ for the rest. This ranges were chosen because the best performances are always found within this range.

Figures 4-7 illustrates the comparative measure on completion time for algorithms that follow the SMP and PMP protocols for different factor graphs. Even though PMP can be applied to any existing GDL-based algorithms, for the results of Figures 4-7 we consider the Max-Sum and Bounded Max-Sum. Afterwards, we illustrate other possible scenarios (extensions of Max-Sum) in Table 1 to summarize their performance. Each line of Figures 4-7 represents the result of both SMP (Number of Clusters$=1$) and PMP (Number of Clusters>1). Note that, domain pruning in the intermediate step (Algorithm 3) of PMP is not applied for the results shown in the Figure 4 and 5 (non-expensive maximization operator), however, this method is applied to the Figure 6 and 7 (expensive maximization operator). According to the graphs of Figures 4-7, the best performance of PMP can be found if the number of clusters are picked from the range around $(5-25)$. However, for the smaller factor graphs this range becomes smaller. For example, in Figure 4, where we are dealing with a factor graph of 100 function nodes the best results are found within the range (5-15) clusters, afterwards, PMP performs worse compared to its SMP counterpart. Notably, for the larger factor graphs the comparative performance of PMP is more substantial in terms of completion time. Moreover, it can be ascertained from the results that after reaching to its pick with a certain number of clusters, performance of PMP drops steadily with the increase of the number of clusters and this trend is common for each scenario. As observed, PMP running over a factor graph with 100-200 function nodes achieves around 18% to 30% performance gain (Figure 4) over its SMP counterpart. On the other hand, if we apply our domain pruning technique on this same setting, the gain increases to around 37% to 53% (Figure 6). Remarkably, when larger factor graphs (400-600 functions) are considered, PMP takes 31% to 36% less time than Max-Sum if domain pruning is not applied and 43% to 58%

less time is consumed if applied. Finally, Figure 5 and 7 depict that this performance gain reaches around 33% to 39% for the factor graph having 3000 to 10,000 function nodes without domain pruning and 45% to 61% with domain pruning.

Extensive simulation results show that a PMP based algorithm always take less time than its SMP counterpart if it can be applied by carefully considering the number of clusters for a certain scenario. However, completion time of such algorithms mainly depends on few issues. First, average time to compute an expensive Factor to Variable message (T_{cp_1}). Second, average time to compute a non-expensive variable to factor message (T_{cp_2}). Finally, average time to transmit a message between nodes (T_{cm}). The ratio of these values would be different as in the literature we have several extensions of Max-Sum, which eventually produce differences in the performance gain we show in Figures 4-7. In summary, we present the range of performance gain in Table 1 considering different possible scenarios with corresponding examples.

Table 1: Performance of PMP compared to SMP in different possible scenarios in terms of completion time.

Factor Graph (1000 functions)	Gain (%)	Example
$(T_{cp_1} > T_{cp_2}) \wedge (T_{cp_1} \approx T_{cm})$	35-60	MS
$(T_{cp_1} \gg T_{cp_2}) \wedge (T_{cp_1} > T_{cm})$	24-44	MS
$(T_{cp_1} \approx T_{cp_2}) \wedge (T_{cp_1} \approx T_{cm})$	35-61	FMS or MS
$(T_{cp_1} \approx T_{cp_2}) \wedge (T_{cp_1} < T_{cm})$	37-64	FMS or MS

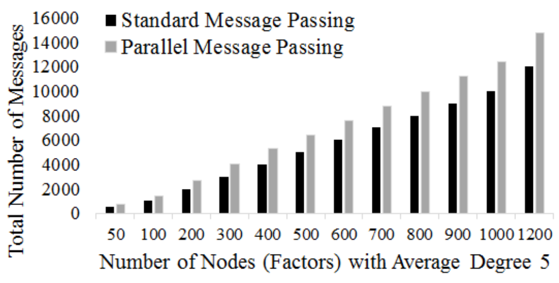


Fig. 8: Total Number of messages: SMP vs PMP.

Even though, PMP reduces the completion time of message passing algorithms, it requires more messages to be exchanged because of the additional round of message passing. Figure 8 illustrates the comparative results of PMP and SMP in terms of total number of messages for factor graphs with number of factors $50-1200$ with average 5 variables connected to a factor. PMP needs $27-45\%$ more messages than SMP for factor graph having less than 500 function nodes and $15-25\%$ more messages for factor graph more than 500 nodes. As more messages are exchanged at the same time in PMP due to the parallel execution, this phenomena does not affect the performance in terms of completion time.

5 Conclusions

We propose a framework which significantly reduces the required completion time of GDL-based message passing algorithms. Our approach is applicable to all the extensions of the Max-Sum algorithm and adaptable to any DCOP formulation which uses the factor graph as a graphical representation. We provide a significant reduction in completion time of such algorithms up to around $45 - 60\%$ for different scenarios with large number of nodes. To achieve this performance, we introduce a cluster based method to parallelize the message passing procedure. Additionally, a domain reduction algorithm is proposed to further minimize the cost of the expensive maximization operator. Given this, by using the PMP framework, we now can indeed use GDL-based algorithms to efficiently solve large real world DCOPs. Future work will look at finding a method to determine the appropriate number of clusters for a certain scenario.

References

Aji and McEliece2000. S. M. Aji and R.J. McEliece. The generalized distributive law. *Information Theory, IEEE Transactions on*, 46(2):325–343, 2000.

Cerquides *et al.*2013. J. Cerquides, A. Farinelli, P. Meseguer, and S. D. Ramchurn. A tutorial on optimization for multi-agent systems. *Computer Journal*, 57:799–824, 2013.

Farinelli *et al.*2008. A. Farinelli, A. Rogers, A. Petcu, and N. R. Jennings. Decentralised coordination of low-power embedded devices using the max-sum algorithm. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 2*, pages 639–646, 2008.

Fitzpatrick2003. Lambert Fitzpatrick, Stephen; Meertens. Distributed coordination through anarchic optimization. In *Distributed Sensor Networks*, pages 257–295. Springer, 2003.

Kim and Lesser2013. Y. Kim and V. Lesser. Improved max-sum algorithm for DCOP with n-ary constraints. *Proceedings of the 12th International Confer- ence on Autonomous Agents and Multiagent Systems (AAMAS 2013)*, pages 191–198, 2013.

Kschischang *et al.*2001. F. R. Kschischang, B. J Frey, and H.A. Loeliger. Factor graphs and the sum-product algorithm. *Information Theory, IEEE Transactions on*, 47(2):498–519, 2001.

Macarthur *et al.*2011. K. S. Macarthur, R. Stranders, S. D. Ramchurn, and N. R. Jennings. A Distributed Anytime Algorithm for Dynamic Task Allocation in Multi-Agent Systems Fast-Max-Sum. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence A*, pages 701–706, 2011.

Maheswaran *et al.*2004a. R. T. Maheswaran, J. P. Pearce, and M. Tambe. Distributed algorithms for dcop: A graphical-game-based approach. In *ISCA PDCS*, pages 432–439, 2004.

Maheswaran *et al.*2004b. R. T. Maheswaran, M. Tambe, E. Bowring, J. P. Pearce, and P. Varakantham. Taking dcop to the real world: Efficient complete solutions for distributed multi-event scheduling. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 310–317. IEEE Computer Society, 2004.

Modi *et al.*2005. P. J. Modi, W. Shen, M. Tambe, and M. Yokoo. Adopt: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 161(1):149–180, 2005.

Petcu2005. Faltings B. Petcu, A. A scalable method for multiagent constraint optimization. In *Proceedings of the 19th international joint conference on Artificial Intelligence*, pages 266–271, 2005.

Ramchurn *et al.*2010. S. D. Ramchurn, A. Farinelli, K. S. Macarthur, and N. R. Jennings. Decentralized Coordination in RoboCup Rescue. *The Computer Journal*, 53:1447–1461, 2010.

Rogers *et al.*2011. A. Rogers, A. Farinelli, R. Stranders, and N.R. Jennings. Bounded approximate decentralised coordination via the max-sum algorithm. *Artificial Intelligence*, pages 730–759, 2011.

Stranders *et al.*2009. R. Stranders, A. Farinelli, A. Rogers, and N. R Jennings. Decentralised Coordination of Mobile Sensors Using the Max-Sum Algorithm. pages 299–304, 2009.

Yeoh *et al.*2008. W. Yeoh, A. Felner, and S. Koenig. Bnb-adopt: An asynchronous branch-and-bound dcop algorithm. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 2*, pages 591–598. International Foundation for Autonomous Agents and Multiagent Systems, 2008.

Zivan *et al.*2014. R. Zivan, H. Yedidsion, S. Okamoto, R. Glinton, and K. Sycara. Distributed constraint optimization for teams of mobile sensing agents. *Autonomous Agents and Multi-Agent Systems*, 29(3):495–536, 2014.

Zivan *et al.*2015. R. Zivan, T. Parash, and Y. Naveh. Applying max-sum to asymmetric distributed constraint optimization. In *Proceedings of the 24th International Conference on Artificial Intelligence*, pages 432–438. AAAI Press, 2015.