2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)

Accelerating Message Passing Operation of GDL-Based Constraint Optimization Algorithms Using Multiprocessing

Syeed Abrar Zaoad*, Tauhid Tanjim*, Mir Hasan†, Md. Mamun-or-Rashid*, Ibrahem Abdullah Almansour‡, and Md. Mosaddek Khan*

*Department of Computer Science and Engineering, University of Dhaka, Dhaka, Bangladesh 1000

†Department of Computer Science and Information Technology, Austin Peay State University, Clarksville, TN, USA 37040

‡Department of Computer Science and Information Technology, King Abdulaziz City for Science and Technology, Riyadh (SA).

***Abstract*—This paper develops a new message passing protocol that can be used to speed up the inference process of Generalized Distributive Law (GDL) based constraint optimization algorithms. In particular, we parallelize the inference process of the GDL-based constraint optimization algorithms which have been widely used to solve Constraint Optimization Problems (COPs) in different real world applications, such as wire routing, image analysis, computer-aided gas pipeline operation, and numerical optimization. It is worth noting that the proposed new parallel approach can be applied to accelerate any GDL-based message passing algorithms, such as Max-Sum, Max-Product or Sum-Product. In this work, we make use of the available Central Processing Unit (CPU) cores of a system to minimize the time it requires to execute a given algorithm. Consequently, this reduced computation time will accelerate the GDL-based COP algorithms which can be used in larger systems. However, the main challenge is to maintain the quality of the solution while minimizing the completion time. Our proposed protocol specifically takes this trade-off into account, and our empirical results depict that it is able to produce the same solution quality while accelerating the inference process 2–10 times faster compared to the state-of-the-art.**

***Index Terms*—COP, Generalized distributive law, Parallel computing, Speeding up**

I. Introduction

Constraint Optimization Problems (COPs) [1] are a model used to deal with constraint handling problems and are an extension of a widely used model named Constraint Satisfaction Problems (CSPs). In general, constraint handling problems are formulated as constraint networks that are often represented graphically wherein nodes are used to depict variables, and the constraints that appear between the variables based on their common choice of action are characterized by edges [2]. Each constraint can be described by a set of variables and represented by value (cost) or reward relationship among the set of variables. The constraints in a graphical representation which form the relationships between the agents can be either hard or soft [3]. Hard constraints consider only those relationships that illustrate the accepted joint assignments of the variables. Unlike the hard constraints that return a binary value for a specific assignment, a soft constraint is a real valued function that defines the cost or reward for each joint variable assignment. Whenever a constraint network just considers hard constraints, the problem which finds a variable assignment that satisfies all the constraints is called a CSP. To be specific, a CSP can be identified as an actual problem with the intent of finding a combination of values from their respective domains for all variables that satisfies all constraints. However, even a single constraint error among thousands would make the complete process null and void. Thus, aiming for the best possible solution is more attainable rather than targeting to obtain a specific one. As a consequence, conceptualizing a constraint handling problem as a COP is more appropriate as well as a natural approach than CSP in a number of real-world applications, such as roster scheduling, supply chain management [4], sensor networks [5], bi-clustering [6], and gene pattern detection.

In more detail, COPs deal with soft constraints where each constraint describes individual reward or cost for each variable assignment. The goal of a COP algorithm is to set every variable to a value from its domain, to minimize the constraint violation. To date, a number of algorithms have been developed to solve COPs and its distributed version namely Distributed Constraint Optimization Problems (DCOPs) [7]. These algorithms can be categorised into two approaches: exact and non-exact. Exact algorithms always provide an optimal solution but sacrifice the performance in terms of time and costs. Nevertheless, attaining an optimal solution is an NP-hard problem which shows an exponentially increasing coordination overhead in proportion to the system growth. As the former type always obtains a globally optimal solution, the latter approaches sacrifice some solution quality for scalability and have, therefore, been used more in practice. Non-exact GDL-based algorithms [8] such as Max-Sum [9], [10] and Bounded Max- Sum (BMS) [11], Max-Product [12], and Min-Sum have received particular attention. The GDL-based DCOP algorithms, in general follow a message passing protocol where nodes of corresponding graphical representation of a COP, enumerate and disseminate utilities (or costs) of their neighbouring nodes for each possible value assignments [8], [13]. Therefore, they exactly share the effects of choosing

978-0-7381-2646-3/21/$31.00 ©2021 IEEE
DOI 10.1109/ISPA-BDCloud-SocialCom-SustainCom52081.2021.00102

non-preferred states with the preferred one during inference through a graphical representation, namely factor graph [9], [14]. Eventually, this information helps these algorithms to achieve good solution quality for large and complex problems. However, it has been recently observed that the Standard Message Passing (SMP) protocol followed by the GDL-based algorithms incurs notable delay in completing the message passing process due to it's redundant waiting time [15].

To reduce the waiting time of the message passing process, and as such to minimize the overall completion time of a given GDL-based algorithm, [15] developed a new message passing protocol, named Parallel Message Passing (PMP) for DCOPs in a multi-agent setting. In particular, PMP efficiently distributes the overhead of message passing to the available agents (each having its own processing capability) to exploit their computational and communication power concurrently. It is worth noting that, in the process, PMP provides solution of the same quality as its SMP counterpart. Despite the aforementioned advantages, it is obvious that PMP cannot be directly applied to the traditional COP settings where a single machine is used to compute all the messages of the GDL-based algorithm. In other words, neither PMP nor the traditional SMP are able to utilize the parallel computation ability (i.e. multi-core CPU) of a modern CPU. In such a setting, message passing operation of a COP algorithm takes too long to complete and this problem is even more severe as the system becomes larger.

In the wake of the aforementioned necessity, coupled with the potential significance of the PMP protcol in reducing the completion time of the message passing process, this paper utilizes the idea of PMP which operates on a multi-agent settings to develop a new Centralized Parallel Message Passing (CPMP) protocol for GDL-based algorithms [8] which operates on a single machine. CPMP specifically parallelizes the message passing process utilizing the multi-core processing capability of a given processor. To be precise, CPMP is a centralized version of PMP for solving COPs in single machine where parallelism is obtained by mutli-core processors. In this paper, we accelerate the GDL-based constraint optimization algorithm (Max-Sum) which can be further used to accelerate biclustering [6]. Notably, we also ensure that CPMP produces the same solution quality as SMP while completing the message passing process significantly faster than SMP for a given GDL-based message passing algorithm. We evaluate the performance of the new approach and compare it with that of the SMP for solving COPs in different cases.

The remainder of the paper is organized as follows. In Section II, we provide a review on related literature and study of the background. The proposed approach is discussed in Section III with a worked example. We then present the empirical results of our method compared to the current state-of-the-art in Section IV. Section V summarizes the contributions of our work and concludes by highlighting the potential areas of future work.

II. Background Study

In this section, we first describe COPs and illustrate how factor graphs can be used to represent COPs. We then detail the GDL framework and discuss how it has been used to solve COPs. Finally, we conclude this section covering the basics of the SMP and PMP message passing protocols.

A. Constraint Optimization Problems

A Constraint Optimization Problem (COP) is a problem that needs to be optimized through a global objective function (maximize or minimize), that is aggregated by a set of constraint functions.

Definition 2.1: A triplet $<X, D, F>$ can define a COP. In details,

- *X defines a set of variables which is finite and discrete. Variables can be defined in a set as* $\{x_0, x_1, .., x_n\}$.
- *D defines a domain set. It can be defined in a set as* $\{D_0, D_1, ..., D_n\}$. *In addition, variables* x_i *may be assigned values from respective or related* $D_i \in D$.
- *F defines constraints set. It can be described in a set as* $\{F_0, F_1..., F_L\}$. *Furthermore,* $F_i \in F$ *is a function of* $x_i \in X$.

In particular, the constraints between the functions and the variables initiate a bipartite graph, generally used as a graphical depiction of such COPs is called factor graph [16], [17]. Thus, the purpose of this model is to assign values from the correlated domain to its related variables in order to either maximize or minimize the global objective function that gradually generates the value of all variables X* (Eq. 1). In details,

$$X^* = \arg\max_x \sum_{i=1}^{L} F_i(x_i) or X^* = \arg\min_x \sum_{i=1}^{L} F_i(x_i) \quad (1)$$

B. Distributed Constraint Optimization Problems

Definition 2.2: A tuple $<A, X, D, F, M>$ can define a DCOPs. where,

- *A defines the set of agents* $\{A_0, A_1, ..., A_k\}$.
- *X,D,F are same as in COPs.*
- *M is a mapping function M:* $\eta \rightarrow A$, *where* η *denotes the functions and variables jointly. Every* η *is held by a single agent.*

In particular, the constraints between the functions and the variables initiate a factor graph as a depiction of such DCOPs [16], [17]. Thus, the purpose of this model is to assign values from the correlated domain to its related variables in order to either maximize or minimize the global objective function.

C. Factor Graph

Definition 2.3: A bipartite graph that illustrates the formation of factorization is called a factor graph. Factor graph contains factor nodes for each local function, variable nodes for each variable and it contains an edge or link between factor node and variable node if and only if the function/factor contains that variable as an argument.

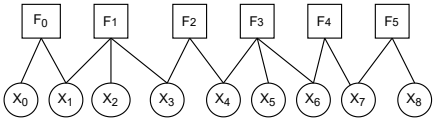


Fig. 1. An illustration of factor graph representing COP. This figure consists factor nodes that is defined by squares and variable nodes that is defined by circles.

In Figure 1, it clearly shows the relationship between the variables and the function of a factor graph that is a graphical depiction of COP. In the presented figure, there are nine variable nodes X = $\{x_0, x_1, ..., x_8\}$ and six function nodes also known as factor nodes F = $\{F_0, F_1, ..., F_5\}$. Furthermore, a set of finite and discrete variable domains D = $\{D_0, D_1, ..., D_8\}$, each domain set $D_i \in D$ is also associated, where each variable $x_i \in X$ in the factor graph. Value of the variables can be any value from the corresponding domain. In the given Figure 1, each function node depends on multiple variable nodes. Here, the global objective functions $F(x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$ is a complex function of six local function- $F_0(x_0, x_1)$, $F_1(x_1, x_2, x_3)$, $F_2(x_3, x_4)$, $F_3(x_4, x_5, x_6)$, $F_4(x_6, x_7)$ and $F_5(x_7, x_8)$ which are represented by factor nodes. To be exact, function nodes perform in the optimization process to either minimize or maximize the global objective function.

D. Standard Message Passing Protocol (SMP)

All GDL-based algorithms maintain a generic and conventional approach that is called Standard Message Passing (SMP) protocol to exchange messages or information among nodes of a factor graph a graphical representation of COPs. In SMP, one node cannot send message to its adjacent node until the messages of all other adjacent nodes are already present. To be exact, a message can be sent from node x to its neighbouring node y through an edge e if and only if all other messages from neighbouring nodes are received by node x through other edges except the edge e [8], [16].

From the above definition, we get a factor graph which represents the previously shown COP formulation follows the SMP protocol to transfer messages using GDL-based message passing algorithms. For message passing of two key GDL-based algorithm Max-Sum and Bounded Max-Sum (BMS) algorithms use Eq. (2) and Eq. (3). So, these algorithms directly can be applied to COPs represented by a factor graph. The variable nodes and the function nodes of this factor graph can uninterruptedly send messages to their neighbors e(function F_j to variable x_i), and (variable x_i to function F_j) depending on Eq. (2) and Eq. (3) respectively to compute the global objective function by building a local objective function Z(x). In Eq. (2) and Eq. (4), M_i represents the connected function nodes to the variable nodes x_i and in Eq. (3) N_j stands for the connected variables to factor nodes F_j. Once functions are formed from Eq. (4), each variable choose a value from its domain that maximizes the function by finding $argmax_{x_i}(Z_i(x_i))$. By making slight modification of these equations, these can be applied to extended version of Max-SUM and Bounded Max-SUM. Those are Bounded Fast Max-SUM (BFMS) [18], Fast Max-Sum (FMS) [14] and BnB-FMS [19]. As a result, the SMP protocol still underpins these algorithms.

Algorithm 1: SMP protocol on a factor graph

Data: A set of functions and variables of a factor graph F_G that represents a COP

1 Find *initNode*$\in F_G$;
2 *sendMessage*(F_G, *initNode, initNode.allNeighbours, NULL*);
3 **while** *every node in* F_G *yet to propagate message to all of their neighbours* **do**
4 | *sendMessage*(F_G, *pNodes, pNodes.allNeighbours, generatedMessage*);
5 | **if** *a variable get all the message from its neighbour* **then**
6 | | x_i.val= $\text{argma}_{x_i} Z_i(x_i)$;
7 | **end**
8 **end**

$$Q_{x_i \to F_j}(x_i) = \sum_{F_k \in M_i \setminus F_j} R_{F_k \to x_i}(x_i) \tag{2}$$

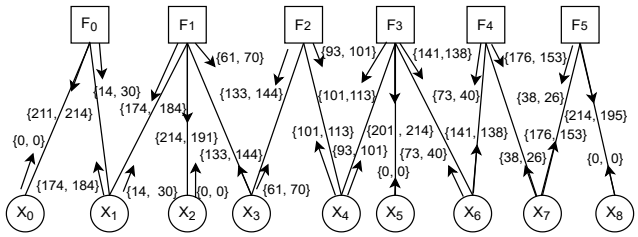
$$R_{F_j \to x_i}(x_i) = \max_{x_j \setminus x_i} \left[F_j(x_j) + \sum_{x_k \in N_j \setminus x_i} Q_{x_k \to F_j}(x_k) \right] \tag{3}$$

$$Z_i(X_i) = \sum_{F_j \in M_i} R_{F_j \to x_i}(x_i) \tag{4}$$

To solve COPs, different GDL-based message passing algorithms such as Max-Sum, Max-Product and Bounded Max-Sum (BMS) operate on factor graph or junction tree.

Algorithm 1 operates on factor graph that gives a clear overview of how SMP works on factor graph. Here, each of the function nodes F_i consists of number of variable nodes x_i in factor graph F_G.

The aforementioned algorithm is operated on the factor graph of a centralized system. Figure 2 gives a clear view on how this algorithm operates on factor graph. It depicts how messages passes between variable nodes and function nodes based on the utility table. At first, the variable nodes and the function nodes of factor graph F_G which are attached to a minimum number of neighbouring nodes, implied by *initNode*, are allowed to transfer messages to their neighbours. Line 1 of Algorithm 1 finds *initNode* which are connected only with one factor graph. Now, these *initNode*s are allowed to send message to their neighbor. Particularly, the function *sendMessage()* represents the messages being sent by allowed nodes to their neighbours. To be precise, the SMP protocol makes sure that no function node or variable nodes within a factor graph cannot create and send a message to its significant neighbour before obtaining messages from the other neighbour(s). Nodes yet to get message from only one neighbor are the permitted nodes (*pNodes*) and their corresponding allowed neighbours *(pNodes.pNeighbours)* are determined for a particular time. At first *initNode* send NULL values to all their neighbouring nodes (Line 2). Now, *While* loop in Lines 3-7 finds the *pNodes* and generates messages using Eq. (2) or Eq. (3) for there



F_0	X_0	X_1
14	R	R
27	R	B
7	B	R
30	B	B

F1	X_1	X_2	X_3
8	R	R	R
5	R	R	B
22	R	B	R
30	R	B	B
31	B	R	R
40	B	R	B
28	B	B	R
15	B	B	B

F2	X3	X4
32	R	R
11	R	B
20	B	R
31	B	B

F3	X4	X5	X6
28	R	R	R
16	R	R	B
2	R	B	R
9	R	B	B
27	B	R	R
4	B	R	B
40	B	B	R
37	B	B	B

F4	X6	X7
35	R	R
12	R	B
1	B	R
14	B	B

F5	X7	X8
38	R	R
19	R	B
0	B	R
26	B	B

Fig. 2. SMP operation on the factor graph of Fig. 1. Here, domain of each variable is {R, B}. Each table represent the equivalent local utility of a function. Direction of sending messages is represented by arrow and the messages represented by values within a curly bracket.

only remaining neighbor *(pNodes. pNeighbours)* yet to send message. And this will continue till every nodes of the factor graph sends messages to all their neighbours. When a variable x_i gets messages from all of its neighbours, it can develop a local objective function $Z_i(x_i)$, and find the value of the variable which maximizes the objective function by finding $argmax_{x_i}(Z_i(x_i))$(Lines 5–7).

Through this message passing operation of GDL-based algorithms, many Constraint Optimization Problems can be solved. Many applications are dependent on this message passing operation. But this message passing operation has many drawbacks and limitations.

SMP protocol uses simple approach and cannot send a message to other neighbour node unless it receives or gets all the messages from all its other neighbours. So, it takes too much time or has to wait for a long time to generate messages. This dependency is common for all nodes and it leads to an increasing amount of average time for message passing operations. Thus, when the systems get larger (resources allocation/tasks), the completion time exponentially increases with proportion to the amount of tasks or resources.

In this paper we presented a parallel message passing algorithm which will outperform the aforementioned SMP Algorithm 1 as benchmark for centralised system.

E. Parallel Message Passing Protocol (PMP)

Another approach of GDL-based message passing algorithm for large-scale Distributed COPs is called Parallel Message Passing (PMP) protocol. In fact, this approach accelerates SMP protocol of Max-Sum using cluster formation and parallel message passing. To be exact, PMP algorithm is an improved version of SMP. Significantly, in SMP each node exchanges message among the neighbouring nodes of a constraint graph representing distributed COPs. Furthermore, a node is not allowed to send message to one of its neighbour until it get all other neighbours' messages similar to SMP.

PMP protocol works on a factor graph where the variable and the function nodes are held by a set of agents. To form the

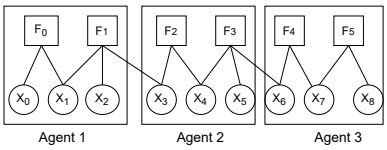


Fig. 3. A sample factor graph representation of a DCOP. It also represents relation among variables, factors and agents of a DCOP. Variable nodes, factor nodes and agents are depicted by circles, rectangles and octagons respectively.

cluster in a decentralized manner, PMP finds a special function node which initiates the cluster formation procedure. In this protocol, all operations in each cluster is performed in parallel. So, average waiting time of nodes is reduced. However, links between clusters are ignored during cluster formation. As a result, to get the same result as SMP protocol it needs two round of message passing and an intermediate step. After the first round of message passing messages, ignored links are recovered in the intermediate step. An agent takes the responsibility to perform computation and communication of intermediate step for the corresponding cluster.

Figure 3 represent a DCOP where $\{X_0, X_1,....,X_8\}$ are the variable nodes and $\{F_0, F_1,..., F_5\}$ are the function/factor nodes. $F(X_0, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8)$ represents the global function. Though SMP and PMP algorithms exchange messages among themselves so privacy of each agents may be compromised. We can use the algorithm of PMP protocol to accelerate SMP in centralized system where cores of processors would execute the clusters rather than agents.

III. The CPMP Algorithm

In this section, We introduce CPMP that tailor the Parallel Message Passing Protocol (PMP) has been tailored for centralised system (i.e. for single-agent settings). To be precise, similar to PMP, CPMP splits the factor graph representation of a COP into several clusters; however, CPMP utilizes available multiple physical cores of a single agent to reduce the overall completion time of the message passing process. Here, each of the clusters execute independently on different core of a processor. In doing so, CPMP reduces the waiting time of each node of the given factor graph, and in effect, minimize total completion time to $\frac{T_{smp}}{\mathcal{N}_\mathcal{C}}$. Here, T_{smp} is the completion time of SMP and N_c is the total number of clusters. However, it is worth noting that while we form the clusters (Algorithm 3) by splitting the original factor graph, some links are ignored from the original factor graph. Thus, messages from those links cannot be incorporated during the inference process. To overcome this problem, we use the same concept of intermediate step developed in PMP to recover those missing values. Thus we can ensure the same solution that can be achieved by the SMP protocol. To complete the message passing, CPMP needs two round of messages passing and an intermediate step. As the cluster computation execute in parallel, total completion time of each round of message passing depends on the largest cluster (Eq. (5)).

$$T_{CPMP} = T_{r1c_{largest}} + T_{intermediate} + T_{r2c_{largest}} \quad (5)$$

Algorithm 2 idetnfies the steps in the CPMP protocol. A Factor graph F_G and a set of functions are given. Initially,

Algorithm 2: CPMP protocol

Data: A set of functions and variables of a factor graph F_G that represent a COP

1 { C,S } ← *makeCluster(F_G, N_c)*, each cluster c_i is sub-factor graph of F_G;
// first round of message passing
2 **for** *every cluster c_i having only neighbour in PARALLEL* **do**
3 | *setVal(S)*$_{c_i}$ ← *clusterCalculation(c_i,NULL)* :: virtual processor
4 **end**
// Intermediate step
5 **for** *each cluster c_i in C* **do**
6 | **for** *each separated node s_j in S_i in PARALLEL* **do**
7 | | *neglectVal*$_{S_i}$← *IntermediateStep(c_i, S_i, s_j, setVal(S), F_G)* :: virtual processor
8 | **end**
9 **end**
// Second round of message passing
10 **for** *every cluster c_i in F_G in PARALLEL* **do**
11 | X_i.val← *clustercalculation(c_i, neglectVal(S)*$_{c_i}$*)* :: virtual processor
12 **end**

factor graph f_G divides into N_c clusters (sub graph of F_G) where N_c is the total number of clusters (Line 1). From the makeCluster() function (Algorithm 3) we get *negNodes* and set of separate nodes *S* of each cluster where *negNodes* denotes the variable nodes (*vnode*) which connect two or more clusters and set of separated nodes *S* represent the *vnodes* within a cluster which are the member of *negNodes*. After the formation of clusters, CPMP executes the first round of message passing. In this round, only those cluster having only one neighbouring cluster can participate (Line 2-4).

A. Cluster Formation

Before starting the message passing of CPMP, we need to split the F_G into N_c clusters which is explained in Algorithm 3. Line 1 and Line 2 computes the total factor nodes and the maximum node in a cluster. At first, take a factor node with only one neighboring factor node and initiate *fnode* with it. Lines 6-15 add factor node to a cluster c_i. Lines 16-22 find the variable nodes (*vnodes*) which connect two clusters. Then add them to split nodes of the corresponding clusters. Figure 5 shows an example of cluster formation. Here, initnode can be F_0 or F_5, we consider F_0 as the initnode randomly. In this example, number of total clusters is three. Thus, maximum factor nodes (*fnodes*) will be two. Finally, factors $\{F_0$ and $F_1\}$ are in cluster 1. Then $\{F_2, F_3\}$ and $\{F_4, F_5\}$ form cluster 2 and 3, respectively. Since x_3 and x_6 are connecting two clusters, $\{x_3\}$, $\{x_3, x_6\}$ and $\{x_6\}$ are the set of split nodes S_1, S_2 and S_3 respectively for the clusters 1, 2 and 3.

B. Message Passing within a Cluster

During the first and the second round of message passing, clusters execute in parallel in different cores of an agent. Calculation of each cluster is given in Algorithm 4. To be

Algorithm 3: makeCluster(Factor graph F_G , number of cluster N_c)

Data: A factor graph is consist of a variables set $X=\{x_1,x_2,...,x_m\}$ and a set of Factor nodes $F=\{x_1,x_2,...,x_m\}$

1 N_F← $|F|$;
2 $N \leftarrow N_F/N_c$;
3 fnode ← initnode;
4 iNodes ← ∅;
5 S ← ∅;
6 **for** *i=1 to N_c* **do**
7 | nodecount ← 0;
8 | **while** *nodecount < N* **do**
9 | | **if** *fnode is added to any cluster yet* **then**
10 | | | *c_i.add()*← *fnode*;
11 | | | *nodecount ← nodecount+1*;
12 | | **end**
13 | | *fnode ← adj(fnode)*;
14 | **end**
15 **end**
16 **for** *every vnode* **do**
17 | **if** *vnode connect two clusters c_i and c_j* **then**
18 | | iNodes.add() ← vnode;
19 | | S_i ← vnode;
20 | | S_j ← vnode;
21 | **end**
22 **end**
23 **return** $\{c_1,c_2,...,c_{N_c}\}$,S;

exact, it works as an individual SMP. Notably, similar to SMP, within a cluster, *vnode* and fnode can send a message to a neighbor only if they receive messages from all its other neighbours. Now, a variable x_i sends a message to factor node F_j within cluster following Eq. (2) and a factor node F_i sends a message to variable x_j following Eq. (3). Line 1 finds initial nodes of the cluster with nodes having only one neighbor. Line 2 sends message from *iNodes* to its neighbor. In while loop, nodes with receiving message from all neighbors except one (permitted nodes) send message to the remaining neighbors depending on the local function. If a *vnode* gets all messages from the *fnode* then calculation of the local value from local objective function as Eq. (4) (Line 5-8). In the first round of message passing only those clusters having one neighboring cluster participate but in the second round of message passing all cluster participate and initial value of split nodes restored in intermediate step (discussed in the following section). Here CPMP follows the same process prescribed in PMP. After first round of message passing, values in split nodes are called *readyVal*. Figure 4 shows that after first round of message passing *readyVal* of x_3 is (61,70) and (73,40) for x_6. After getting the *ignVal* from the intermediate step, CPMP starts the second round of message passing. Finally, global objective can be gained through executing each cluster in parallel. Thus each variable node gets its desired value.

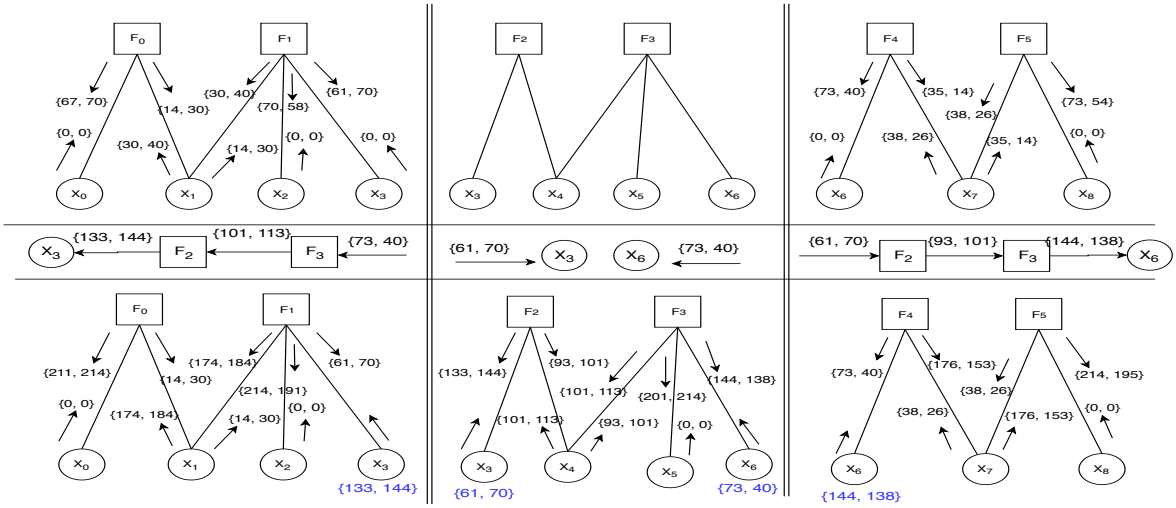


Fig. 4. Overview of CPMP

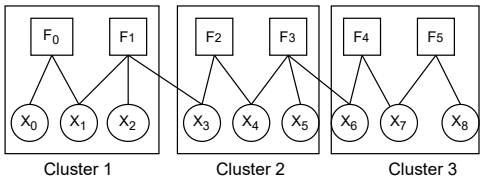


Fig. 5. Cluster formation

Algorithm 4: clustercalculation(cluster c_i, initial value of splite nodes S)

```
1  Find iNodes∈ c_i;
2  messageUpdate(c_i, iNodes, iNodes.allNeighbours, NULL);
3  while every node in c_i yet to propagate message to all of their neighbours do
4      messageUpdate(c_i, pNodes, pNodes.allNeighbours, generatedMessage);
5      if a variable get all the message from its neighbour then
6          x_i calculate a local objective function Z_i(x_i) ;
7          x_i.val= argmax_{x_i} Z_i(x_i) ;
8      end
9  end
```

C. Intermediate Step

In the first round of message passing, *readyVal* nodes from the set of split nodes of participating clusters are calculated. Now Algorithm 5 shows intermediate step for all split nodes. Line 2 and Line 3 find the adjacent clusters connected via split node s_j and total neighbors (*cCount*) of that cluster. If a cluster has only adjacent cluster (Line 4-6) then CPMP transfers *readyVal* of split node that connecting the adjacent cluster into *ignVal* of S_j. If *cCount* is greater than 1 then using s_j as a root a dependent acyclic graph(D_G) is created (Line 7-10) where leaf nodes are *vnode* with *readyVal* or *vNode* with only one neighbor. After creating D_G, values of the node are passed to its parent. Then *fnode* calculates the value using its local function and pass it to its parent, and this will continue until s_j gets the value. In Figure 4, cluster 2 has two adjacent clusters with only one neighbor. So values of x_3 and x_6 are gained from the *readyVal*. For split nodes x_3 in cluster 1 and x_6 in cluster 2 form a graph D_G and calculate its *ignVal* from it shown in Figure 4.

Algorithm 5: IntermediateStep(cluster c_i,set of splite nodes S_i ,ready value calculated from first phase ready-Val(S), Factor grpah F_G)

```
1  for All s_j in S_i do
2      c_a ← adjcluster(s_j,c_i);
3      cCount_{c_a} ← totalAdjacent(c_a);
4      if cCound_{c_a} == 1 then
           // cluster having only one adjacent cluster
5          ignVal.s_j ← readyVal(S).s_j;
6      else
           // cluster having more than one adjacent cluster
7          cNode ← adjNode(c_a,s_j);
           // create depending acyclic graph
8          while cNode≠ ∅ do
9              D_G(s_j) ← cNode ;
               // Synchronous operation on every edge of D_G.
10             cNode ← adjNode(cNode);
11         end
12         s_j.value ← syn(D_G(s_j));
13         ignVal(c_i)← append(s_j.values);
14     end
15 end
16 return ignVal(c_i);
```

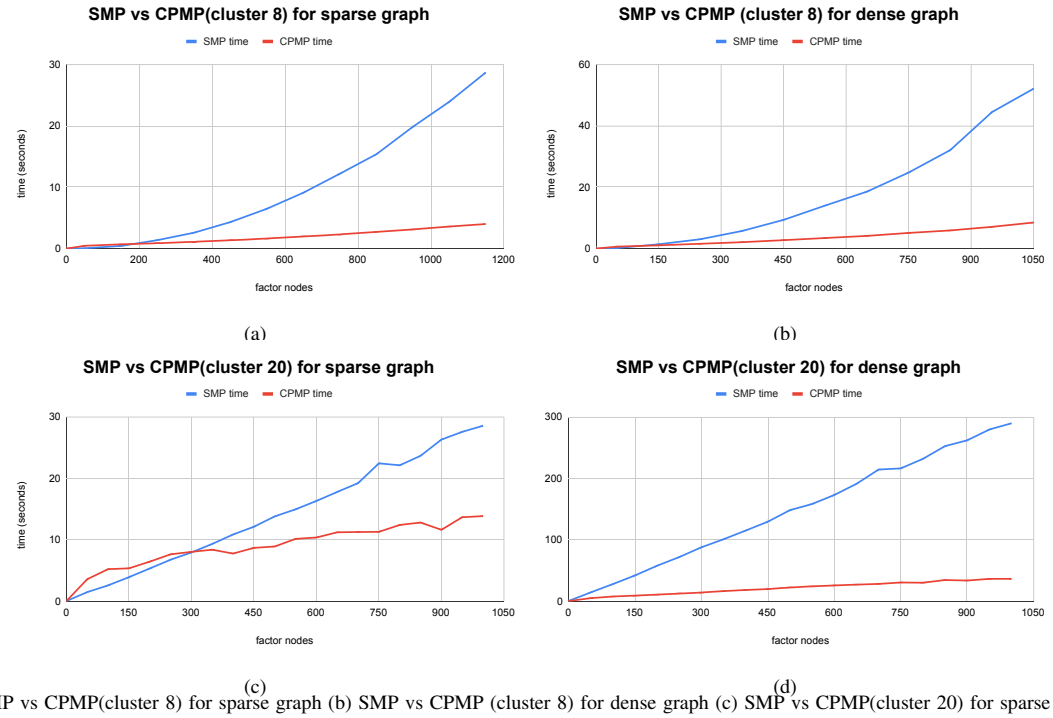


Fig. 6. (a) SMP vs CPMP(cluster 8) for sparse graph (b) SMP vs CPMP (cluster 8) for dense graph (c) SMP vs CPMP(cluster 20) for sparse graph (d) SMP vs CPMP(cluster 20) for dense graph

IV. Empirical Results

In this section, we empirically evaluate and analyze the performance of our proposed CPMP protocol, and compare it with the performance of the SMP protocol in terms of completion time. In our experiments, we consider factor graphs having $100-1500$ factor nodes, each of which is connected with a random number of variable nodes ranging from $2-10$. We categorize the factor graphs into sparse and dense based on the connected number of variable nodes per factor node. For the sparse setting, this number is randomly taken from the range $2-7$, while the range is 7-14 for the dense graph. For the sake of simplicity, we set a variable's domain size to 2. It is worth noting that our reported comparative results are also comparable to larger settings. In this paper, we report the performance of CPMP as well as SMP specified on a different number of clusters ranging from 2 to 20. As we only focus on the performance of CPMP compared with SMP regarding the completion time, we choose to conduct our experiment on a generic setting instead of a application specific one. Note that all of the experiments are performed on a simulator deployed on the following two machines.

- **Machine 1** is built with a 1.8 GHz Intel Core i5 CPU with 8 GB RAM having 8 virtual cores.
- **Machine 2** is built with a Intel(R) Xeon(R) 3.00GHz CPU having 20 cores physical cores.

Python 3 is used as the programming language and its multi-processing package to use the virtual cores in parallel.

A. Experiment E1: Compare CPMP with SMP for fixed number of clusters.

In Figure 6(a) and 6(b), we illustrate the comparative performance of CPMP with 8 clusters and SMP in terms of completion time for sparse graph and dense graphs respectively on Machine 1. It is observed that from the results depicted in Figure 6(a) and 6(b) that for the smaller number of clusters SMP works better than CPMP. This is because from Eq. (5) we know that completion time CPMP depends on two rounds of message passing and an intermediate step. These two rounds of message passing depend on the largest cluster. For the smaller number of factor nodes in a factor graph, number of messages are small. As a result, the completion time is also short. There is some overhead in creating a cluster and set up virtual processors for parallel computing. For small factor graph completion time of SMP is so short. In effect, for smaller number of factor nodes completion time in SMP can not overcome the overhead of CPMP. As a result, total completion time in SMP is less than CPMP for a smaller factor graph. Nevertheless for the larger factor graph completion time of CPMP works much better than SMP. In larger factor graph total computation time is much larger and computation time of each cluster in CPMP also larger compared to the overhead. Therefore, as the graph becomes larger, CPMP works better. In CPMP, factor graphs are split into 8 clusters. and execute them in parallel. Hence, theoretically the computation time of each cluster is $T_{smp}/8$. Therefore, the total completion time of CPMP will be,

$$T_{cpmp} = 2 * T_{smp}/8 + T_{intermediate} + overhead$$

According to experimental results, we can report that

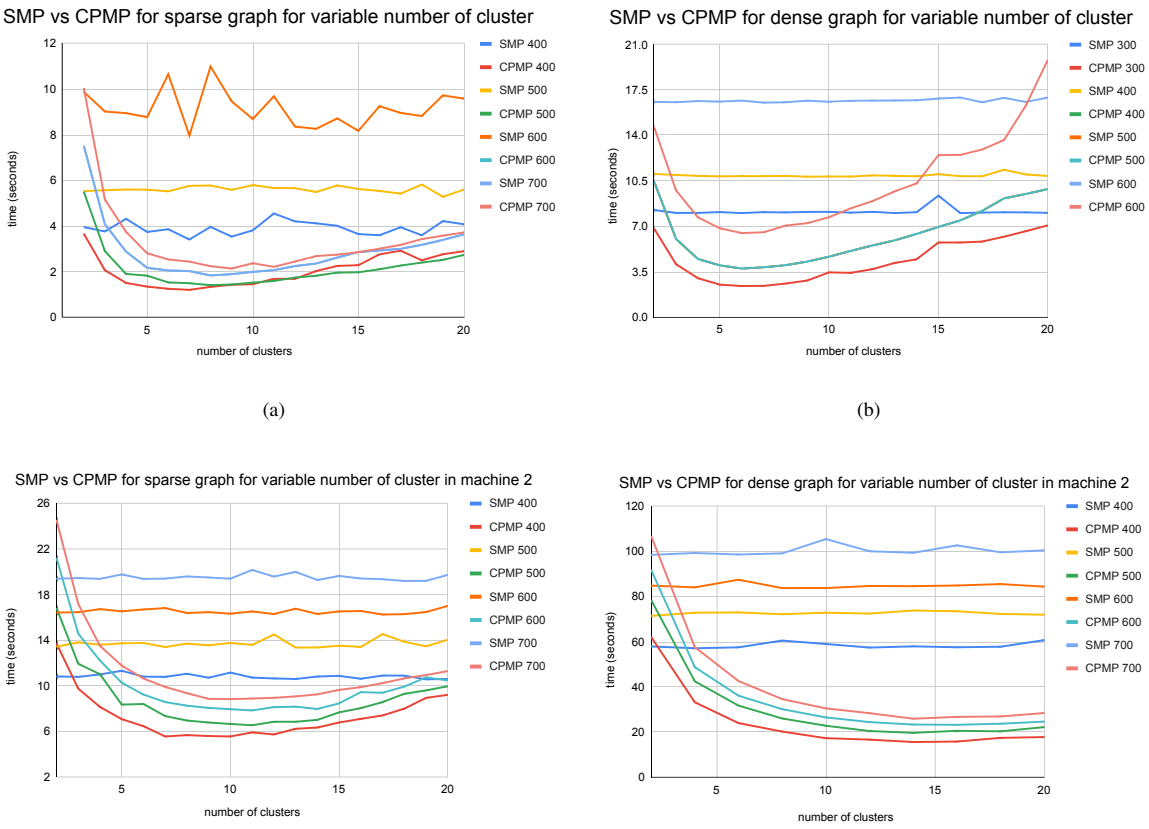


Fig. 7. (a) SMP vs CPMP for sparse graph for variable number of cluster in machine 1(b) SMP vs CPMP for sparse graph for variable number of cluster in machine 1(c) SMP vs CPMP for sparse graph for variable number of cluster in machine 2(d) SMP vs CPMP for sparse graph for variable number of cluster in machine 2

CPMP works around 2-5 times better than SMP for larger factor graph.

In Figure 6(c) and 6(d), we illustrate the comparative performance of CPMP with 20 clusters and SMP in terms of completion time both for sparse graph and dense graph which are executed for both on Machine 2. The outcome is similar to the results we have observed in the previous experimental settings. For smaller factor graph, CPMP with 20 clusters works worse than SMP. Figures show that CPMP works better for dense graph and large factor graph.

In CPMP factor graphs are split into 20 clusters and execute them in parallel. Therefore, theoretically computation time of each cluster is $T_{smp}/20$. So the total completion time of CPMP will be,

$$T_{cpmp} = 2 * T_{smp}/20 + T_{intermediate} + overhead$$

According to experimental results shown in Figure 6(c) and 6(d) , we can report that CPMP works approximately 2-6 times better than SMP for larger factor graph.

B. Experiment E2: compare CPMP with SMP based on variable clusters.

In the experimental setting E2, we compare the performance of SMP and CPMP depending on the completion time for the factor graph with total 300-700 factor nodes executed on both Machine 1 and Machine 2. For each factor graph, we run SMP and CPMP changing the total number of clusters ranging from 2-20 both for sparse graph and dense graph. In each instance, we create factor graph randomly from the given number of factor nodes and variable nodes. In Figure 7(a) and 7(b) , we represent the comparative results for sparse graph and dense graph that are executed on Machine 1. For a fixed number of factor nodes two lines represents the completion time for SMP and CPMP. From the overview of section III, we know that for CPMP factor graph split into n clusters and every cluster executed on different virtual cores in two rounds of message passing. Intermediate step for each *neglectNode* also executes on virtual cores. Hence, from the Eq. (5) we can say that CPMP reduces the completion time of SMP algorithm approximately by n/2. The more the number of clusters n increases, the faster the algorithm. Our experiment executed on quadcore Intel core i5 processor where there were only 8 virtual cores available. Therefore, maximum 8 concurrent execution can execute on this machine in parallel. If the number of the processor is greater than the number of virtual cores then the system assigns one of 8 processors to each process temporarily at a time. A process may get stopped then another process will start to run. For too many processes

file I/O or network I/O will become the bottleneck and the program will slow down cause additional overhead needs to switch between the processes. Because of the randomly generated factor graph with a fixed number of factors nodes, SMP time changed each time. In effect, the spike in SMP is observed. As our machine only consists of 8 virtual processors, in Figure 7(a) and 7(b), it can be observed that the performance of CPMP increases with the number of clusters and starts degrading the result from the number of clusters 6-8 because of the extra overhead to create and switch between processes.

In Figure 7(c) and 7(d), we represent the comparative results for sparse graph and dense graph that are executed on Machine 2. The outcome is similar to the results we have observed for Machine 1. Since this machine consists of 20 cores, maximum 20 concurrent executions can execute on this machine in parallel. Additionally, the outcome shows that the performance of CPMP increases with the number of clusters. It starts degrading the result from the number of clusters 10-12 for sparse graph and 14-16 for dense graph because of the extra overhead to create processes and switch between processes. Therefore, theoretically, we can say that if the number of cores in the computer increase then completion time CPMP also will decrease and CPMP works better for dense graph.

V. Conclusions and Future work

In this paper, we focus on one of the expensive parts of GDL-based COP algorithms that is the message passing operation. In fact, we propose and develop a new protocol of message passing that accelerates the message passing operation of GDL-based COP algorithms. In so doing, we improve the practical use and scalability of these algorithms. Significantly, we do so without sacrificing the quality of their solution. To be precise, we have developed a new generic message passing protocol for GDL-based COP algorithms, namely Centralized Parallel Message Passing (CPMP) protocol. Our approach is mainly dependent on the efficient use of existing computational resources available in a system. To evaluate the performance of our approach, we empirically observe the comparative results in two different settings. Our results illustrate that the CPMP works $3-7$ times better than SMP on the system with 8 and 20 cores. The trend that we observe in our empirical results also indicates that if an agent possess more processor with more cores, CPMP can perform better. This is because the reduction of completion time depends on the efficient use of the number of clusters. Finally as CPMP is a generic message passing protocol, it can be applied to any GDL-based algorithms to solve large-scale COPs regardless of the application domain.

The findings in this work has opened a number of new research opportunity. In the future, we want to explore the practical applicability of our method in different real-world problems. Moreover, we want to see whether existing domain pruning techniques [15], [20], [21] for GDL-based algorithms can be deployed to further accelerate our approach. It may help us to accelerate message passing operations of GDL-based algorithms.

Acknowledgment

This research is primarily supported by the ICT Innovation Fund of Bangladesh Government. We also acknowledge the use of the BdREN High Performance Computing Facility.

References

[1] A. Meisels, Constraints optimization problems-cops, in: Distributed Search by Constrained Agents, Springer, 2008, pp. 19–26.

[2] R. Dechter, D. Cohen, et al., Constraint processing, Morgan Kaufmann, 2003.

[3] J. Kendall, Hard and soft constraints in linear programming, Omega 3 (6) (1975) 709–715.

[4] R. Sun, B. Chu, R. Wilhelm, J. Yao, A csp-based model for integrated supply chain, in: Proceedings of AAAI-99 Workshop on Artificial Intelligence for Electronic Commerce, Orlando, FL, 1999, pp. 92–100.

[5] A. Farinelli, A. Rogers, N. R. Jennings, Agent-based decentralised coordination for sensor networks using the max-sum algorithm, Autonomous agents and multi-agent systems 28 (3) (2014) 337–380.

[6] M. Denitto, A. Farinelli, M. A. Figueiredo, M. Bicego, A biclustering approach based on factor graphs and the max-sum algorithm, Pattern Recognition 62 (2017) 114–124. doi:10.1016/j.patcog.2016.08.033. URL http://dx.doi.org/10.1016/j.patcog.2016.08.033

[7] S. Mahmud, M. Choudhury, M. M. Khan, L. Tran-Thanh, N. R. Jennings, Aed: An anytime evolutionary dcop algorithm, in: Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems, 2020.

[8] S. M. Aji, R. J. McEliece, The generalized distributive law, IEEE Transactions on Information Theory 46 (2) (2000) 325–343. doi:10.1109/18.825794.

[9] A. Farinelli, A. Rogers, A. Petcu, 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, International Foundation for Autonomous Agents and Multiagent Systems, 2008, pp. 639–646.

[10] M. M. Khan, L. Tran-Thanh, N. R. Jennings, A generic domain pruning technique for gdl-based dcop algorithms in cooperative multi-agent systems, in: Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems, 2018, p. 1595–1603.

[11] A. Rogers, A. Farinelli, R. Stranders, N. R. Jennings, Bounded approximate decentralised coordination via the max-sum algorithm, Artificial Intelligence 175 (2) (2011) 730–759.

[12] Y. Weiss, W. T. Freeman, On the optimality of solutions of the max-product belief-propagation algorithm in arbitrary graphs, IEEE Transactions on Information Theory 47 (2) (2001) 736–744.

[13] M. M. Khan, Speeding up gdl-based distributed constraint optimization algorithms in cooperative multi-agent systems, Ph.D. thesis, University of Southampton (2018).

[14] S. D. Ramchurn, A. Farinelli, K. S. Macarthur, N. R. Jennings, Decentralized coordination in robocup rescue, The Computer Journal 53 (9) (2010) 1447–1461.

[15] M. M. Khan, L. Tran-Thanh, S. D. Ramchurn, N. R. Jennings, Speeding up gdl-based message passing algorithms for large-scale dcops, The Computer Journal 61 (11) (2018) 1639–1666.

[16] F. R. Kschischang, B. J. Frey, H.-A. Loeliger, et al., Factor graphs and the sum-product algorithm, IEEE Transactions on information theory 47 (2) (2001) 498–519.

[17] H.-A. Loeliger, An introduction to factor graphs, IEEE Signal Processing Magazine 21 (1) (2004) 28–41.

[18] K. Macarthur, Multi-agent coordination for dynamic decentralised task allocation, Ph.D. thesis, University of Southampton (2011).

[19] K. S. Macarthur, R. Stranders, S. Ramchurn, N. Jennings, A distributed anytime algorithm for dynamic task allocation in multi-agent systems, in: Twenty-Fifth AAAI Conference on Artificial Intelligence, 2011.

[20] M. M. Khan, L. Tran-Thanh, N. R. Jennings, A generic domain pruning technique for gdl-based dcop algorithms in cooperative multi-agent systems, in: Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems, International Foundation for Autonomous Agents and Multiagent Systems, 2018, pp. 1595–1603.

[21] Z. Chen, X. Jiang, Y. Deng, D. Chen, Z. He, A generic approach to accelerating belief propagation based incomplete algorithms for dcops via a branch-and-bound technique, in: Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 33, 2019, pp. 6038–6045.