Parallel Graph Mining with Dynamic Load Balancing

Nilothpal Talukder
IBM Systems, Poughkeepsie, NY
Email: ntalukd@us.ibm.com

Mohammed J. Zaki
Rensselaer Polytechnic Institute, Troy, NY
Email: zaki@cs.rpi.edu

***Abstract*—Frequent subgraph mining (FSM) has important applications in areas such as bioinformatics, social networks and others. In this paper, we present a highly scalable approach called PARGRAPH that can efficiently mine from a single graph in both distributed as well as shared-memory based systems. In a distributed environment, we can leverage the local memory of multiple compute nodes for storing a large number of intermediate states for enumerating patterns. To address the skewness in the pattern generation tree, our approach uses a novel hybrid load balancing scheme to efficiently distribute workload across both processes and threads. Our experiments demonstrate good speedups using message passing interface (MPI) and OpenMP threads.**

***Keywords*-Parallel Frequent Graph Mining; Dynamic Load Balancing; High Performance Computing**

I. INTRODUCTION

Discovering frequent subgraph patterns from graph structures is a well-studied and very important problem in areas such as computational chemistry, bioinformatics, and social networks. For example, the scientists might be interested in finding common substructures in labeled input graphs representing protein structures, protein-protein interactions or chemical compounds. Similarly, mining frequent subgraphs from a social graph or citation network can help find communities which may interest social scientists. This problem is known as frequent subgraph mining (FSM) and is divided into two broad categories, namely, finding frequent patterns in either i) a *graph database* comprising multiple input graphs (e.g., a set of chemical compounds), or ii) a *single large input graph* setting (e.g., a social network or citation graph).

For both cases, the FSM task requires enumerating possibly an exponential number of candidate subgraph patterns, and checking their presence (subgraph isomorphisms) in the input graph. Finally, it outputs all subgraphs with frequency (or support) above some *minimum support* threshold. In the multiple graphs case, frequency is simply the number of graphs that contain the pattern. However, defining the notion of support in a single graph is more challenging, since it is not enough to simply state whether a pattern exists or not. Instead, we have to find all the distinct isomorphisms from the pattern to the input graph. However, this violates the *anti-monotonicity* principle that is required for effective pruning of the output pattern space. That is, a subgraph can have fewer isomorphisms (i.e., lower support) than its supergraph (e.g., assume that the input graph has only one vertex labeled A, which is connected to 100 vertices labeled B, then the vertex A occurs only once, but the edge $A-B$ occurs 100 times). Several anti-monotonic support measures have been proposed for use in the single graph setting [1].

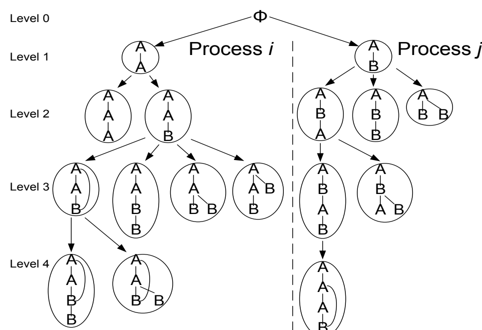


Figure 1. Vertical split of pattern lattice among processes

Most existing parallel [2–5] and mapreduce-based FSM approaches [6–8] adopt the graph database setting. In contrast, our focus is on mining patterns from a single input graph. Existing sequential solutions for this problem include SiGraM [9] and GraMi [10], whereas existing parallel approaches target only shared-memory (SMP) systems [11]. More recent distributed approaches include Arabesque [12], DistGraph [13] and Pregel-based [14]. However, our focus in this paper is to leverage dynamic load balancing to efficiently mine a moderate sized input graph that fits in the memory of a compute node, by effectively leveraging both thread-based and MPI-based parallelism.

In this paper, we present PARGRAPH, a scalable FSM algorithm on a single graph that runs on distributed as well as shared-memory (SMP) based systems. The algorithm assumes that the input graph fits in the memory of a single compute node. The largest processable input graph depends on the size of the local memory of the compute nodes in a distributed system. In the case of shared-memory systems we determine the size of the local memory by dividing the total memory among the processes. It is important to note that programming for distributed environments is considerably more challenging than SMP systems, since it requires interprocess communication in the form of message passing or some other means. Also, the mining task requires us to keep the occurrences/embeddings of the expanding candidate patterns in memory, which can lead to a large amount of intermediate states that has to be maintained, and that can be much larger than the input graph. That is why it is often harder to scale FSM algorithm on a single compute node. On a distributed or cluster environment, we can leverage the local memory of multiple compute nodes, which helps us to scale significantly.

The central idea of PARGRAPH is that we provide the same input graph to all compute nodes/processes and perform the mining task in parallel. In fig. 1 we show an example of a partial candidate pattern generation tree or

pattern lattice resulting from the extension of the patterns. It is obvious that we can divide the tree vertically into computationally independent parts among multiple processes or threads. In addition, since the support of a pattern is not dependant on the other parts of the tree we can do a DFS (depth-first search) based traversal in the candidate generation tree. We also consider canonical code [15] for patterns to remove duplicates from the pattern lattice. From fig. 1 it is also evident that the subtrees in the lattice can have different depths, leading to unbalanced workload for different processes. Therefore, to keep the processors and threads busy at all times, we perform a hybrid workload balancing scheme using MPI (message passing interface) and OpenMP threads. Each process that is out of work can request and obtain work from other active processes, and then can redistribute the workload among its threads. We report experimental results of our parallel graph mining implementation using a multi-core (16-core) shared memory machine and the HPC platfrom, IBM Blue Gene/Q.

The contributions of our work is the following:

- PARGRAPH utilizes both thread-based parallelism within each compute node and distributed approach across compute nodes (using MPI). Hence, it is suitable for a wide variety of environments, such as distributed systems, shared memory systems, and HPC systems.
- PARGRAPH adopts a hybrid dynamic load balancing scheme that shares workload across both processes and threads, making it very efficient and highly scalable in any environment.

II. BACKGROUND

In this section, we review some definitions. Let V be the set of *vertices* and $E \subseteq V \times V$ the set of *edges*. A *graph* is defined as $G = (V, E)$. In addition, we assume that L is a labeling function for nodes and edges; we denote by $L(v)$ the label for vertex $v \in V$, and by $L(v_1, v_2)$ the label for the edge $(v_1, v_2) \in E$. In the following, we use $G = (V, E)$ for the *single large input graph*, and we use $P = (V_P, E_P)$ for a *pattern graph*, i.e., one of the subgraphs we wish to mine in G.

Subgraph Isomorphism and Embedding: We say that the pattern $P = (V_P, E_P)$ is *subgraph isomorphic* to $G = (V, E)$, denoted as $P \subseteq G$, if there exists a injective function, $\phi\colon V_P \to V$ such that: 1) $\forall v \in V_P$, $L(v) = L(\phi(v))$, and 2) $\forall (v_i, v_j) \in E_P$, $(\phi(v_i), \phi(v_j)) \in E$ and $L(v_i, v_j) = L(\phi(v_i), \phi(v_j))$. In this case, the isomorphic subgraph in G comprising the vertices $\phi(v_1), \phi(v_2), \cdots, \phi(v_p)$, is also called an *embedding* of the pattern P in the input graph G (here $p = |V_P|$). We use the terms isomorphism and embedding interchangeably, since given the isomorphism function ϕ, we can uniquely identify the corresponding embedding (which is a subgraph of G).

Canonical Code: Given a pattern $P = (V_P, E_P)$, let $Aut(P)$ denote the automorphism group for P, i.e., the set of all graphs isomorphic to P. The unique minimal element of this set is called the *canonical code* for P. There are several approaches for representing the canonical code, such as minimal or canonical adjacency matrices [16–18], and minimal DFS code [15]. In this paper, we use the latter, i.e., we represent P using the minimal DFS code. For example, a pattern A-B-A can have different DFS code representations, such as $(0, 1, A, -, B)(1, 2, B, -, A)$ or $(0, 1, B, -, A)(0, 2, B, -, A)$. Each entry in a DFS code is a five element tuple $(v_i, v_j, L(v_i), L(v_i, v_j), L(v_j))$, where v_i and v_j indicate the vertex ids that are assigned during the DFS traversal from an arbitrary vertex in the pattern. Among different codes for the pattern, $(0, 1, A, -, B)(1, 2, B, -, A)$ happens to be the canonical or minimal DFS code [15].

Pattern Extension: The search for frequent patterns is usually done in a breadth-first or depth-first manner, starting with single edge graphs and extending existing patterns with an extra edge at each level or step. There are two types of edge extensions: 1) a *forward edge* is one that introduces a new vertex and joins an existing vertex to the new one, and 2) a *backward edge* is one that connects two existing vertices creating a cycle. For systematic pattern enumeration, forward extensions are allowed only from the vertices on the rightmost path [15], and backward extensions are allowed only from the rightmost vertex [15], to the rest of the vertices on rightmost path (we omit the details here for brevity). This guarantees that all possible candidate patterns will be generated.

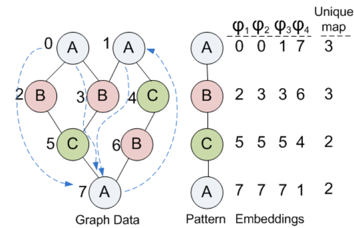


Figure 2. Support in a single graph

Support: For a pattern $P = (V_P, E_P)$ and input graph $G = (V, E)$, let $\Sigma = \{\phi_1, \phi_2, \cdots\}$ denote the set of all isomorphisms/embeddings of P in G. The support of a pattern P is defined based on some function of Σ. For instance, support can be defined as the cardinality of Σ, i.e., the number of embeddings of P in G. However, as we saw in the introduction, in the case of a single input graph, this definition violates the *anti-monotonicity* principle, which requires that the support of a pattern should not be greater than the support of its subgraphs. To address this issue a support measure using the maximum independent set (MIS) of the overlap graph between embeddings was proposed by Kuramochi et al. [9]. However, MIS is known to be a NP-hard problem.

In this paper we use the most restricted node (MRN) support, also called as the minimum image based support proposed in [1], where support is defined as the minimum number of unique vertex mappings over any of the vertices in P. More formally, given the set of embeddings Σ, the set of unique vertex mappings for a node $v \in V_P$ is given as

$$\Phi(v) = \cup_{i=1}^{|\Sigma|} \phi_i(v)$$

and the MRN support of P is defined as:

$$\sigma(P) = \min_{v \in V_P} \{|\Phi(v)|\}$$

Fig.2 shows the embeddings of example pattern $A-B-C-A$ in the input graph. For instance, one of the embeddings is $\phi_1 = \{0, 2, 5, 7\}$ corresponding to the first column, and the set of mappings for the first vertex is $\{0, 1, 7\}$. The number of unique mappings for each pattern vertex are 3, 3, 2 and 2, respectively. Thus the MRN support of P is $\sigma(P) = 2$. One of the advantages of using the MRN support is that even if there are an exponential number of embeddings, the storing the mappings $\Phi(v)$ across all vertices $v \in V_P$ takes at most polynomial space, namely $O(|V_P| \times |V|)$.

Frequent Graph Mining: In the single-graph setting, the input for frequent graph mining is G and a user specified parameter called *minimum support threshold*, or in short, $minsup \in \mathbf{Z}$. The task is to discover all subgraph patterns, such that for each pattern P, we have $\sigma(P) \geq minsup$.

III. Related Work

FSM is a very well studied problem in both transactional [15–19] and single graph setting [9, 10, 20]. For transactional setting, the earlier algorithms, such as AGM [17] and FSG [16] use the same principle as the Apriori frequent itemsets mining algorithm [21]. The latter algorithms, such as gSpan [15], Gaston [19] and FFSM [18] adopt a unique ordering of the subgraph patterns which results in a significant reduction of the search space of the candidates. GSpan uses minimal DFS code as the canonical representation of the subgraph patterns. Also, gSpan adopts a pattern-growth based (depth-first) approach where a frequent and minimal DFS code is extended recursively until all frequent supergraphs are enumerated. The efforts in sequential single graph mining include SiGraM [9], and more recently GraMi [10]. SiGraM uses maximum independent set (MIS) based approach for computing support which is expensive. The recent approach, GraMi avoids enumerating all isomorphisms and maps the subgraph isomorphism problem into a constraint satisfaction problem (CSP). However, both SiGraM and GraMi are sequential approaches and do not scale. The parallel gSpan algorithm described in [2] performs both breadth-first and depth-first exploration of the pattern space using static and dynamic load balancing of tasks, respectively. However, the algorithm only works in shared-memory systems. A parallel single graph mining method for SMP systems was proposed in [11]. The work parallelizes VSiGraM algorithm [9] using OpenMP taskq/task extensions. It demonstrates good speedup with respect to the sequential VSiGraM, which in turn, uses costly MIS support measure. Arabesque framework [12] proposes a general embeddings centric or "think like an embedding" paradigm for graph problems like FSM, motif discovery, and clique finding. More recently, the distributed single graph mining algorithm DistGraph [13] can mine massive networks. It considers partitioned input graph and performs BFS based exploration on the pattern lattice. Other recent approaches use graph processing platforms, such as Pregel to perform the mining task [14]. In this paper, our focus is on developing a efficient parallel approach that can handle unpartitioned moderated sized input graphs, such as a network with millions of vertices, given sufficient memory.

IV. Parallel Single Graph Mining

In this section, we discuss parallel single graph mining in a step by step approach. The goal is to develop an algorithm that can efficiently run on both distributed and shared-memory based environments. First, we consider the sequential graph mining algorithm and attempt to naïvely parallelize it (Basic-ParGraph algorithm). The algorithm vertically partitions the pattern lattice where each process obtains roughly equal number of subtrees rooted at a single edge pattern, as shown in fig. 1. Then, based on this we develop our parallel algorithm ParGraph that uses dynamic load balancing with inter-process communication, such as message passing. Finally, we discuss how our algorithm performs hybrid load balancing using both threads and processes, making it suitable for both shared-memory and distributed environment.

A. Basic Parallel Approach

We start off with a simple approach Basic-ParGraph (Alg. 1) for single graph mining that performs naïve parallelism on the sequential graph mining algorithm. The algorithm is run by p processes in parallel with the same input graph G. First, we determine all single edge patterns $\mathcal{F}$ from the input graph G (step 1). Then, the processes can equally divide the patterns among them and perform mining on the corresponding set of single edge patterns ($\mathcal{F}_i$ for process i). In other words, each process gets a vertical partition of the pattern lattice and obtains roughly equal number of subtrees rooted at a single edge pattern. In step 2, the process i determines the part of the frequent single-edge patterns it would consider to expand. To determine $\mathcal{F}_i$ no communication among the processes is needed. Each process has access to $\mathcal{F}$ and hence, it can determine the corresponding indices in $\mathcal{F}$ from the process id. The mining process continues in each process independently in depth-first manner by invoking the sequential mining steps, i.e., Seq-Mainloop function in step 5. Inside Seq-Mainloop, we use gSpan's canonical labeling method for the graphs, i.e., DFS code, that provides fast pruning of duplicate subgraph patterns from the candidate generation search tree (step 7). The embeddings $\Sigma(P)$ are kept in memory. Each time a pattern is extended, so is its set of embeddings (step 8). From the embeddings $\Sigma(P)$ the vertex mappings are extracted, and the support $\sigma(P)$ is computed. Finally, the frequent patterns are presented in the output.

B. Single Graph Mining with Dynamic Load Balancing

For a parallel algorithm when a problem size is known (e.g., matrix-matrix or matrix-vector multiplication) and we have a constant number of process or nodes, we can easily distribute the tasks statically. Unfortunately, this does not hold for FSM, as we do not know the number of frequent patterns in advance. One of the prime challenges for parallel graph mining is that the candidate generation tree is usually not balanced. This skewness (one subtree is very deep compared to the others) of the tree can cause the performance of the parallel algorithm to be as bad as it's sequential counterpart. Therefore, we need to dynamically redistribute the work when some process is out of work.

Algorithm 1 Basic Parallel Single Graph Mining

BASIC-PARGRAPH(Graph G, Threshold $minsup$, Number of processes p, Process id i)

1: Compute all single edge patterns $\mathcal{F}$
2: Equally distribute $\mathcal{F}$ among p processes; process i gets $\mathcal{F}_i$
3: **for** each $P \in \mathcal{F}_i$ **do**
4:     Create initial embeddings $\Sigma(P)$
5:     SEQ-MAINLOOP$(G, P, \Sigma(P))$
6: **end for**

SEQ-MAINLOOP(Graph G, Pattern P, Embeddings $\Sigma(P)$)

7: **if** P is canonical **then**
8:     Get all possible extensions $\mathcal{E}(P)$ of P from $\Sigma(P)$ and G
9:     **for** each $e \in \mathcal{E}(P)$ **do**
10:         $P' \leftarrow P$ extended by e
11:         **if** support $\sigma(P') \geq minsup$ **then**
12:             output P'
13:             SEQ-MAINLOOP$(G, P', \Sigma(P'))$
14:         **end if**
15:     **end for**
16: **end if**

In this section, we discuss our parallel single graph mining algorithm PARGRAPH that uses parallel formulation of the depth first search (DFS) [22] on the candidate generation tree. The parallel depth first search relies on dynamic load balancing, where each process can perform a DFS walk on disjoint parts of the tree, since they are computationally independent.

When a process is done finding all frequent patterns in its corresponding part of the tree, it actively requests an unexplored part of the tree from other processes. In our message passing architecture, the *requester* process asks for work and the *donor* process responds to the request with either some work from its search space or with a reply indicating no availability of work. The work in our case is the DFS code prefixes of the patterns stored at each level of the candidate generation tree. We maintain a DFS prefix queue for each level of the tree. The queue is populated in the order of the DFS code extensions. Fig. 3 shows DFS prefix queues for five levels. The DFS code of a candidate pattern can be constructed by concatenating the currently expanded prefixes from level 1 to the current level.

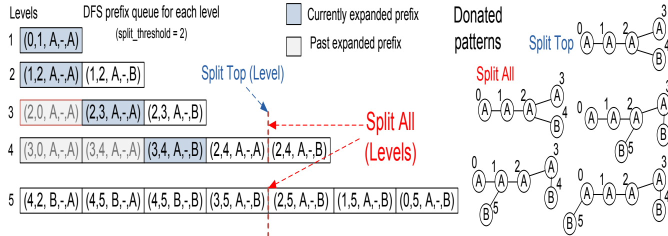


Figure 3. Work split strategies of DFS code prefix queues: Split Top and Split All

The expanded DFS code prefixes are highlighted in Fig. 3 for the sake of illustration (however, in the actual implementation we remove the already expanded prefixes from the queue and put the currently expanded DFS code in a separate stack). Here, $(0,1,A,-,A)(1,2,A,-,A)$ $(2,3,A,-,A)(3,4,A,-,B)(4,2,B,-,A)$ is an example DFS code. We refer the DFS queue as work queue in our algorithm.

Algorithm 2 Parallel Single Graph Mining

PARGRAPH(Graph G, Threshold $minsup$, Number of processes p, Process id i)

1: Compute all single edge patterns $\mathcal{F}$
2: Equally distribute $\mathcal{F}$ among p processes; process i gets $\mathcal{F}_i$
3: Populate patterns $\mathcal{F}_i$ in work queue $\mathcal{Q}_i$
4: $state \leftarrow ACTIVE$
5: **while** termination not detected **do**
6:     **for** each $P \in \mathcal{Q}_i$ **do**
7:         Create initial embeddings $\Sigma(P)$
8:         PARGRAPH-MAIN$(G, P, \Sigma(P))$
9:     **end for**
10:     $state \leftarrow IDLE$
11:     DYN-LOAD-BALANCE()
12: **end while**

PARGRAPH-MAIN(Graph G, Pattern P, Embeddings $\Sigma(P)$)

13: **if** P is canonical **then**
14:     Get all possible extensions $\mathcal{E}(P)$ of P from $\Sigma(P)$ and G
15:     DYN-LOAD-BALANCE()
16:     **for** each $e \in \mathcal{E}(P)$ **do**
17:         $P' \leftarrow P$ extended by e
18:         **if** support $\sigma(P') \geq minsup$ **then**
19:             output P'
20:             PARGRAPH-MAIN$(G, P', \Sigma(P'))$
21:         **end if**
22:     **end for**
23: **end if**

DYN-LOAD-BALANCE()

24: **if** Process i has pending msg **then**
25:     Get next msg from msg_queue
26:     **if** msg is a work request from $requester$ process **then**
27:         **if** $state = ACTIVE$ and i can donate work **then**
28:             Send work to $requester$ process
29:         **else**
30:             Reply $requester$ with no work availability
31:         **end if**
32:     **end if**
33:     **if** msg is a work reply from $donor$ process **then**
34:         **if** $donor$ has sent work **then**
35:             Populate work in $\mathcal{Q}_i$
36:             $state \leftarrow ACTIVE$
37:         **end if**
38:     **end if**
39: **end if**
40: **if** $state = IDLE$ and termination not detected **then**
41:     Send work request to next process
42: **end if**

The pseudocode of PARGRAPH is shown in Alg. 2. Every process maintains a list of available work or unexpanded candidate patterns in the form of DFS codes. We skip the details of work queue $\mathcal{Q}_i$ here for simplicity of the algorithm, and revisit it in the next section. Now, each process runs Alg. 2 and starts off with it's share of single edges (length 1 DFS codes) $\mathcal{F}_i$ (step 2) and populating them in work queue $\mathcal{Q}_i$ (step 3). We maintain a *state* of the process ($ACTIVE$ or $IDLE$) to indicate whether it is currently working or out of work. The basic structure of the algorithm is still the same as Alg. 1; the new addition is DYN-LOAD-BALANCE() invoked in steps 11 and 15. The work queue $\mathcal{Q}_i$ gets repopulated after a successful work request by process

i (step 35). Therefore, the requesting process will again be in $ACTIVE$ state and will continue to repeat the procedure until a global termination of the distributed algorithm has been reached (step 5). The termination is reached when all processes become $IDLE$, i.e., there are no more work across the whole system to be shared among the processes. However, the detection of termination is nontrivial in a distributed setting. When one process becomes $IDLE$ it continues to ask for work from other processes, even in the situation when all other processes are $IDLE$. The processes could perform a periodic broadcast of the states. Obviously, it incurs high communication cost. In PARGRAPH we use Dijkstra's token based termination detection algorithm [23] for this purpose. For a total number of p processes, the runtime of this algorithm is $O(p)$, which works well for small number of processes (8 or 16).

In DYN-LOAD-BALANCE() function, the process i probes for incoming messages (step 24) A work request from a $requester$ process is responded to accordingly based on the current state of the process and work availability. The availability of work is determined by the size of a DFS code prefix queue being above a threshold ($split_threshold$). We set $split_threshold = 2$ in our implementations. We discuss work split strategies in subsection IV-D. When a process receives a work response it populates the prefix queues with the DFS codes obtained from the $donor$ process. The embeddings or even the vertex mappings can be very large and are not sent through message passing interface, because of the large network communication overhead. For example, for the $donor$ has to send $O(|V_P| \times |V|)$ mappings for each donated pattern $P = (V_P, E_P)$. Rather, they are regenerated locally with much less overhead by the $requester$ process.

There are several choices for the load balancing schemes [22] to request work from the "next" donor process (step 41).

- *Asynchronous or Local Round Robin (ARR)*: Each process maintains a local variable for the next process and increments it (as in a logical ring formation). The next process is $n \leftarrow (n+1) \bmod p$, where p is the total number of processes.
- *Global Round Robin (GRR)*: Each process probes for a global variable for the next process (needs lock for exclusive access). It is stored in a designated process.
- *Random Polling (RP)*: The next process is randomly chosen by each process.

We chose the first and third options (ARR and RP) in our algorithm since they do not have locking contention and have been shown [22] to have demonstrated good performance with parallel DFS traversal.

C. Work Queue

As mentioned earlier, every process keeps a list or available work or unexpanded DFS codes. For that we maintain a DFS code prefix queue for each level of candidate generation tree. Fig. 3 shows the prefix queues for top five levels from the root. The DFS code of a candidate pattern can be constructed by concatenating the currently expanded prefixes from level 1 to the current level. When a pattern is expanded by right-most path extensions, the extensions are put into the next level queue. The expanded DFS code prefixes are highlighted in Fig. 3 for convenience. However, as mentioned earlier, in the actual implementation we remove the already expanded prefixes from the queue. An example DFS code is $(0, 1, A, -, A)(1, 2, A, -, A)(2, 3, A, -, A)(3, 4, A, -, B)$ $(4, 2, B, -, A)$. That is, when a process is done finding all frequent patterns in its corresponding part of the tree, it actively requests an unexplored part of the tree from other processes. In our message passing architecture, the $requester$ process asks for work and the $donor$ process responds to the request with either some work from its search space or with a reply indicating no availability of work.

D. Work Split Strategies

Now, we discuss how the work requests from a process are handled by a $donor$ process. As mentioned earlier, the size of a DFS prefix queue is indicated by the number of unexplored candidate prefixes in the queue. When a process receives a work request, first it checks whether it is in $ACTIVE$ state. Then, it checks the size of DFS code prefixes beginning from level 1. The work availability is determined by the size of the queue being above $split_threshold$. We term the first level to have such a criterion, the "topmost available level". If we consider $split_threshold = 2$, in Fig. 3 the topmost available level is level 4. It is shown that splitting the work queue in half usually achieves the best performance [22]. Therefore, the queue of DFS code prefixes is split into half and given to the requesting process. Also, it is better to give work from the top level rather than the levels below [5]. The reason is that work from the top level guarantees more expansions of the tree and hence more work. We explore two work split strategies in our algorithm as explained in the following.

- *Split Top*: The donor process only splits the queue at the topmost available level and donates half of it.
- *Split All*: The donor process splits queues from the topmost available level to the currently expanding level. Then, it donates half of each queue.

In Fig. 3 for Split Top we split only level 4, whereas for Split All we split levels 4 and 5. As mentioned earlier, in our algorithm we only communicate DFS codes, and the embeddings are regenerated by the receiving process.

E. Hybrid Approach: Distributed and Shared Memory

We consider a hybrid approach for PARGRAPH on a cluster of computers or a distributed environment. Each computing unit in a cluster or a distributed system is usually a SMP machine. The basic idea of our algorithm is that we run an instance of parallel graph mining algorithm (Alg. 2) per compute node and inside each node we can run threads to divide the work even further. Each process works on the same input graph. It is possible to run a threads-based load balancing inside a computing unit, and a process based load balancing among the computing units. This gives rise to a situation where one can conveniently choose the number of processes or threads for the computation. For example, p processes (1 per compute node) and k threads per process can be used. Our threads load balancing is very similar

to the process based dynamic load balancing. However, in addition to the private work queues, we now maintain a shared work queue and a message queue for each thread. Also, the messages do not need to be communicated through network, rather they can be directly written to the *donor* thread's message queue. Other threads might also want to write to the same message queue and hence we need to use locks to prevent concurrent accesses. Upon receiving a work request, the donor thread can now directly populate the shared queue of the requesting thread instead of sending it through the network. The work split strategy also remains unchanged. When choosing the next thread for requesting work ARR or RP schemes are used. If a *donor* thread finds a work request, it determines whether it can split the work and then directly populates the work queue of the $requester$ thread. Since the $requester$ thread is $IDLE$ at this point, we don't need a lock. The *donor* thread also changes the state of the $requester$ thread to $ACTIVE$ so it can start working again.

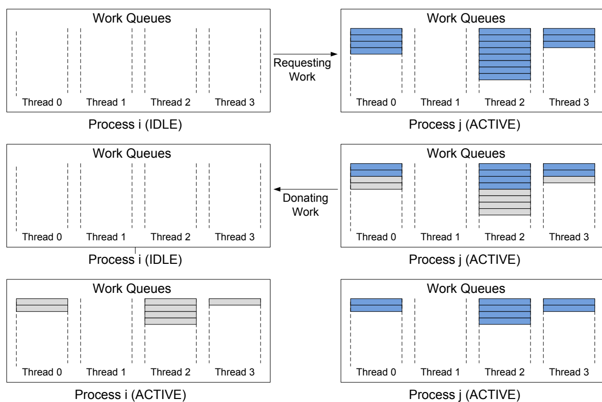


Figure 4. Work splitting in hybrid load balancing

The process state becomes $IDLE$ when all threads become $IDLE$. We do not need a termination detection algorithm, since the thread states can be accessed by any thread. In this situation, the designated thread from the thread group performs process based dynamic load balancing (DYN-LOAD-BALANCE() in Alg. 2) or requests work from other thread groups, i.e., processes. The designated thread in the donor process then asks the rest of the threads to donate work. All threads donate half of their work in the same way we discussed earlier. The designated thread then ships this work to the requesting process. Again, the corresponding designated thread on the requesting side receives and redistributes the work among the $IDLE$ threads. Thus, the snapshot of the work queue on the donor process is effectively halved and given to the threads of the requesting process. The threads that obtained work switch to the $ACTIVE$ state and so does the process itself. Fig. 4 shows a conceptual diagram of how the work sharing is done in the hybrid approach. In this scenario, process i is initially in $IDLE$ state and requests work from the process j. Each process has 4 threads, and the state of the work queue of each thread is shown. Upon obtaining the work request, all threads in process j split work in half and donate to process i. As a result, process i becomes $ACTIVE$.

V. EXPERIMENTS

In this section, we report results from our experiments with PARGRAPH.

A. Setup

We implemented our parallel graph mining algorithm PARGRAPH using C++ (compiled using g++ (v. 4.4.7) and O3 optimization flag). We use the OpenMP (v. 3.0) library for thread-based parallelism within compute nodes, and the portable, open-source and freely available MPI implementation MPICH2 (v. 1.5) for distributed computation across nodes. Our PARGRAPH code is available for download at https://github.com/zakimjz/DistGraph/tree/master/src/parallel.

In addition, we used two different environments to run the experiments: 1) An SMP machine with 16-core, 2.9Ghz AMD Opteron 6272 processor, 256 GB shared memory and running Ubuntu SMP OS; and 2) the IBM Blue Gene/Q supercomputer with upto 16 compute nodes, and each node consisting of a 16-core 1.6 GHz A2 processor, with 16 GB of DDR3 memory.

The datasets used are described in Table I. The single graph datasets in the experiments were obtained from the PDB dataset [24], with a small, a medium and a large instance. The graphs represent protein structures, where amino acids with different labels are connected with one another.

Table I
DATASETS AND THEIR PROPERTIES: VERTICES $|V|$, EDGES $|E|$, LABELS $|L|$, AVG. DEGREE AD, AND CLUSTERING COEFFICIENT CC

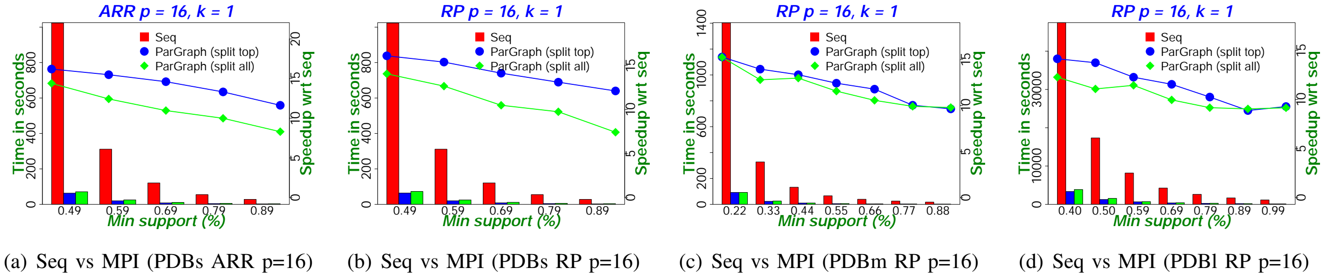
dataset	$\|V\|$	$\|E\|$	$\|L\|$	AD	CC
PDBs	20226	83356	22	8.2	0.57
PDBm	90982	349779	22	3.9	0.57
PDBl	2020188	7905260	22	3.9	0.55

B. Results

The bar plots in Fig. 5 show the performance comparison of the sequential run and our multi-core SMP machine runs. The left Y axis shows the total time, and the bars show the time for different methods with different minimum support values. A value s on the X axis indicates $minsup$ of $s\%$ with respect to $|V|$, i.e., we are using relative support for each pattern. So for a pattern to be frequent it must satisfy the condition $\sigma(P) \geq |V| \times s/100$. In most cases the right Y axis records the speedup with respect to a baseline method in the same plot.

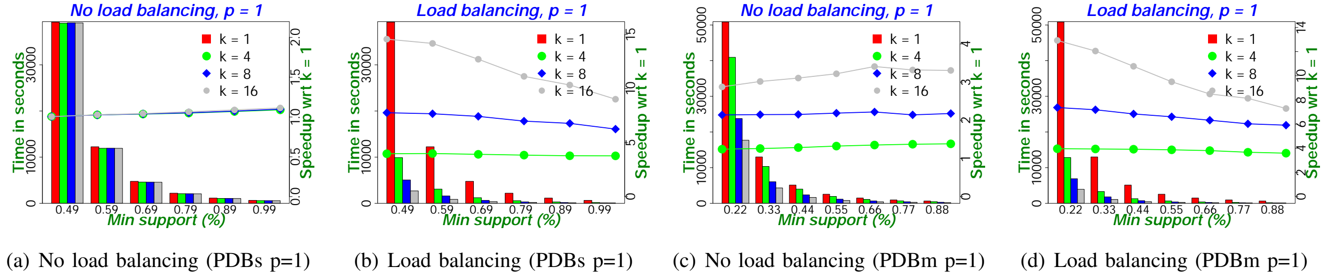
Fig. 5(a) and 5(b) show the results we obtained with PDBs dataset, using 16 processes, and having two different load balancing schemes (ARR and RP). On both plots we observe that the work split strategy "split top" performs better than the "split all". This may be due to the high communication cost incurred for the "split all" strategy. However, we do not observe any significant difference between the performance of ARR or RP. With the smallest support value 0.49%, both ARR and RP strategies achieve a speedup of just over 15 with "split top".

Fig. 5(c) and 5(d) show the results we obtained with the PDBm and PDBl datasets, respectively, using 16 processes. Again, with the PDBm dataset and the lowest support value



(a) Seq vs MPI (PDBs ARR p=16) (b) Seq vs MPI (PDBs RP p=16) (c) Seq vs MPI (PDBm RP p=16) (d) Seq vs MPI (PDBl RP p=16)

Figure 5. Results from multi-core

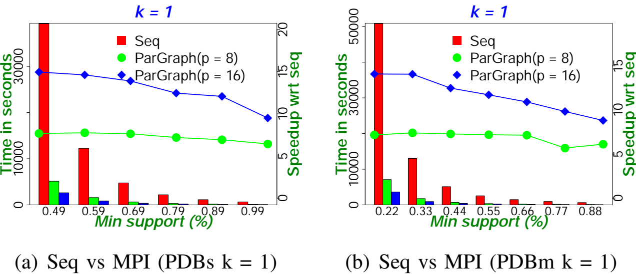


(a) No load balancing (PDBs p=1) (b) Load balancing (PDBs p=1) (c) No load balancing (PDBm p=1) (d) Load balancing (PDBm p=1)

Figure 6. Effect of load balancing Blue Gene/Q

of 0.22%, we achieve a speedup of 15.3 . However, with the PDBl dataset the speedup is just under 15.

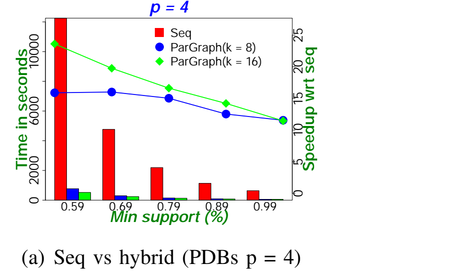
We show the effects of our dynamic load balancing scheme in Fig. 6 on BG/Q platform. The communication between the compute nodes is performed using MPI (for example thread 0 is designated for the task of communicating with external nodes during dynamic load balancing). On BlueGene/Q each compute node has 16GB of shared memory for 16 cores. Therefore, each core gets roughly 1GB of memory for its independent use. In our hybrid approach we load a single copy of the graph into the shared memory of each compute node. The plots report results using thread based parallelism, using 1, 4, 8 and 16 threads. Fig. 6(a) and 6(c) show results with no load balancing with the PDBs and PDBm graphs, respectively.



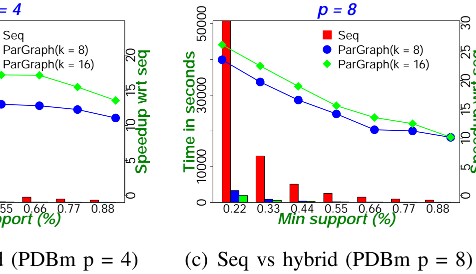
(a) Seq vs MPI (PDBs k = 1) (b) Seq vs MPI (PDBm k = 1)

Figure 7. Distributed run from Blue Gene/Q

There is almost no speedup with any number of threads with the PDBs graph. However, there is a very low speedup w.r.t the single-thread version, in the case of the PDBm graph. For example, with 16 threads, the speedup is just around 3. Due to the absence of dynamic load balancing, many threads became *IDLE* very soon and without work till the end of the run. BG/Q was not able to mine the PDBl graph consisting more than 2M vertices. On the other hand, Fig. 6(b) and 6(d) show effects of multi-threaded dynamic load balancing. With PDBs and PDBm graphs we achieve almost linear speedup. We observe speedups of 15 and 13.8, respectively at the lowest support values.



(a) Seq vs hybrid (PDBs p = 4)



(b) Seq vs hybrid (PDBm p = 4) (c) Seq vs hybrid (PDBm p = 8)

Figure 8. Hybrid load balancing results from Blue Gene/Q

Fig. 7 shows the results from the multi-node run on BG/Q. In this case, we ran a single thread instance (process) on 8 and 16 compute nodes and performed dynamic load balancing through MPI. For both PDBs and PDBm datasets we observed an speedup around 7 and 15, respectively. This shows that our implementation of the MPI based load balancing is also very efficient.

Finally, Fig. 8(a) and 8(c) show the results with the hybrid load balancing on BG/Q, using 4 and 8 MPI processes with 8 and 16 threads for both runs. The speedups obtained are reasonable. For 4 processes and 16 threads, we have 64 instances running. However, with both PDBs and PDBm datasets, we achieve a speedup right around 25. The reason is that when a process (a group of threads) is out of work, it has to wait much longer than in the case of the MPI based dynamic load balancing. Each thread in the donor process has to co-ordinate to donate tasks to the requesting process. The performance suffers due to that.

C. Discussions

From the results we see that both the MPI and the OpenMP thread based parallelism achieves almost linear speedup in both distributed and SMP platforms. The performance of the hybrid load balancing approach dampens a little due to the coordination among the threads. Other possible approaches for thread based load balancing including work donating (sender-initiated) and work stealing (receiver-initiated) [3]. The status of local task queues are updated in a global data structure, called global empty list (sender-initiated case) and global available list (receiver-initiated case). However, these approaches may require high locking contention due to high number of accesses by the threads. Therefore, careful design of the algorithm is required to extract maximum performance or near-linear speedups using these approaches. Our approach is very similar to "work donating" except that we do not use a global empty list which usually has high contention for locking. We rather reduce the contention by using separate message queues. However, for fewer number of threads (8 or 16) the performance of these approaches usually have little difference.

VI. Conclusion

Most modern systems are distributed and equipped with multi-core processors, and sometimes accelerators such as GPUs. Our distributed algorithm leverages hybrid architecture on a large distributed system. It is possible to design a truly hybrid algorithm which combines message passing, threads and accelerators for maximum performance. Tasks such as the support computation of a pattern during the single graph mining are currently done sequentially in a thread. With accelerators we can perform the support computation using multiple cores in parallel [4]. Furthermore, our work can be extended to other graph mining tasks, such as closed graph [25],and maximal graph [26] on a variety of platforms.

Acknowledgment

This work was supported by NSF Award IIS-1302231. We also thank Robert Kessl for initial discussions on implementing dynamic load balancing.

References

[1] B. Bringmann and S. Nijssen, "What is Frequent in a Single Graph?" in *Proc. 12th Pacific-Asia Conf. on Knowl. Discovery and Data Mining (PAKDD)*, 2008, pp. 858–863.

[2] G. Buehrer, S. Parthasarathy, and Y.-K. Chen, "Adaptive Parallel Graph Mining for CMP Architectures," in *Proc. 6th IEEE Int. Conf. on Data Mining (ICDM)*, 2006.

[3] T. Meinl, M. Wörlein, I. Fischer, and M. Philippsen, "Mining Molecular Datasets on Symmetric Multiprocessor Systems," in *Proc 2nd IEEE Int. Conf. on Syst., Man, and Cybern.*, 2006, pp. 1269–1274.

[4] R. Kessl, N. Talukder, P. Anchuri, and M. J. Zaki, " Parallel Graph Mining with GPUs," in *Proc. 3rd Int. Workshop on Big Data, Streams and Heterogeneous Source Mining*, 2014, pp. 1–16.

[5] G. D. Fatta and M. R. Berthold, "Dynamic Load Balancing for the Distributed Mining of Molecular Structures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, no. 8, pp. 773–785, Aug. 2006.

[6] B. Wu and Y. Bai, "An Efficient Distributed Subgraph Mining Algorithm in Extreme Large Graphs," in *Proc. Int. Conf. on Artifi. Intell. and Comput. Intell.: Part I (AICI)*, 2010, pp. 107–115.

[7] W. Lin, X. Xiao, and G. Ghinita, "Large-Scale Frequent Subgraph Mining in MapReduce," in *Proc. 30th IEEE Int. Conf. on Data Eng. (ICDE)*, 2014, pp. 844–855.

[8] M. Bhuiyan and M. Al Hasan, "An Iterative MapReduce Based Frequent Subgraph Mining Algorithm," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 3, pp. 608–620, Mar. 2015.

[9] M. Kuramochi and G. Karypis, "Finding Frequent Patterns in a Large Sparse Graph," *J. Data Mining and Knowl. Discovery*, vol. 11, no. 3, pp. 243–271, Nov. 2005.

[10] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis, "GRAMI: Frequent Subgraph and Pattern Mining in a Single Large Graph," *Proc. VLDB Endow.*, vol. 7, no. 7, pp. 517–528, Mar. 2014.

[11] S. Reinhardt and G. Karypis, "A Multi-Level Parallel Implementation of a Program for Finding Frequent Patterns in a Large Sparse Graph." in *Proc. 12th Int. Workshop on High-Level Parallel Program. Models and Supportive Environments (HIPS)*, 2007.

[12] C. H. C. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga, "Arabesque: A System for Distributed Graph Pattern Mining," in *Proc. 25th ACM Symp. on Operating Syst. Principles (SOSP)*, 2015, pp. 425–440.

[13] N. Talukder and M. J. Zaki, "A Distributed Approach for Graph Mining in Massive Networks," *J. Data Mining and Knowl. Discovery*, vol. 30, no. 5, pp. 1024–1052, 2016.

[14] X. Zhao, Y. Chen, C. Xiao, Y. Ishikawa, and J. Tang, "Frequent Subgraph Mining Based on Pregel," *Comput. J.*, 2016.

[15] X. Yan and J. Han, "gSpan: Graph-Based Substructure Pattern Mining," in *Proc. 2nd IEEE Int. Conf. on Data Mining (ICDM)*, 2002, pp. 721–724.

[16] M. Kuramochi and G. Karypis, "Frequent Subgraph Discovery," in *Proc. 1st IEEE Int. Conf. on Data Mining (ICDM)*, 2001, pp. 313–320.

[17] A. Inokuchi, T. Washio, and H. Motoda, "An Apriori-Based Algorithm for Mining Frequent Substructures from Graph Data," in *Proc. 4th European Conf. on Principles of Data Mining and Knowl. Discovery (PKDD)*, 2000, pp. 13–23.

[18] J. Huan, W. Wang, and J. Prins, "Efficient Mining of Frequent Subgraphs in the Presence of Isomorphism," in *Proc. 3rd IEEE Int. Conf. on Data Mining (ICDM)*, 2003, pp. 549–552.

[19] S. Nijssen and J. Kok, "A Quickstart in Frequent Structure Mining can make a Difference," in *Proc. 10th ACM SIGKDD Int. Conf. on Knowl. Discovery and Data Mining*, 2004, pp. 647–652.

[20] D. J. Cook and L. B. Holder, "Substructure discovery using minimum description length and background knowledge," *J. Artif. Intell. Res.*, vol. 1, no. 1, pp. 231–255, Aug. 1993.

[21] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," in *Proc. 20th Int. Conf. on Very Large Data Bases (VLDB)*, 1994, pp. 487–499.

[22] V. Kumar, A. Y. Grama, and N. R. Vempaty, "Scalable Load Balancing Techniques for Parallel Computers," *J. Parallel Distrib. Comput.*, vol. 22, no. 1, pp. 60–79, Jul. 1994.

[23] E. W. Dijkstra, W. H. J. Feijen, and A. J. M. van Gasteren, "Derivation of a Termination Detection Algorithm for Distributed Computations," *Inform. Process. Lett.*, vol. 16, no. 5, pp. 217–219, Jun. 1983.

[24] RCSB, "Protein Data Bank." [Online]. Available: http://www.rcsb.org/pdb/home/home.do

[25] X. Yan and J. Han, "CloseGraph: mining closed frequent graph patterns," in *Proc. 9th ACM SIGKDD Int. Conf. on Knowl. Discovery and Data Mining*, 2003, pp. 286–295.

[26] J. Huan, W. Wang, and J. Prins, "SPIN: Mining Maximal Frequent Subgraphs from Graph Databases," in *Proc. 10th ACM SIGKDD Int. Conf. on Knowl. Discovery and Data Mining*, 2004, pp. 581–586.