Parallel Community Detection Algorithm Using a Data Partitioning Strategy with Pairwise Subdomain Duplication

Diana Palsetia[1(✉)], William Hendrix[2], Sunwoo Lee[1], Ankit Agrawal[1], Wei-keng Liao[1], and Alok Choudhary[1]

[1] Northwestern University, Evanston, IL, USA
{drp925,slz839,ankitag,wkliao,choudhar}@eecs.northwestern.edu
[2] University of South Florida, Tampa, FL, USA
whendrix@usf.edu

Abstract. Community detection is an important data clustering technique for studying graph structures. Many serial algorithms have been developed and well studied in the literature. As the problem size grows, the research attention has recently been turning to parallelizing the technique. However, the conventional parallelization strategies that divide the problem domain into non-overlapping subdomains do not scale with problem size and the number of processes. The main obstacle lies in the fact that the graph algorithms often exhibit a high degree of data dependency, which makes developing scalable parallel algorithms a great challenge.

We present PMEP, a distributed-memory based parallel community detection algorithm that adopts an unconventional data partitioning strategy. PMEP divides a graph into subgraphs and assigns each pair of subgraphs to one process. This method duplicates a portion of computational workload among processes in exchange for a significantly reduced communication cost required in the later stages. After data partitioning, each process runs MEP on the assigned subgraph pair. MEP is a community detection algorithm based on the idea of maximizing equilibrium and purity. Our data partitioning method effectively simplifies the communication required for combining the local results into a global one and hence allows us to achieve better scalability over existing parallel algorithms without sacrificing the result quality. Our experimental results show a speedup of 126.95 on 190 MPI processes for using synthetic data sets and a speedup of 204.22 on 1225 processes for using a real-world data set.

1 Introduction

Data clustering is a branch of data mining algorithms that organizes a collection of data points into groups based on their similarity [10]. Clustering graph data, also known as community detection, usually refers to the identification of vertex subsets (clusters) that have significantly more internal edges than external ones [7]. Since the past few years, the volume of data has surpassed the

J.M. Kunkel et al. (Eds.): ISC High Performance 2016, LNCS 9697, pp. 98–115, 2016.
DOI: 10.1007/978-3-319-41321-1_6

capabilities of traditional sequential algorithms. For instance, CNM (Clauset, Newman, Moore), a popular community detection algorithm based on maximum modularity takes approximately 18 hours to process a social network data set containing 2,238,731 users and 14,608,137 connections [5,22]. Demands on high-performance solutions have encouraged researchers to develop heuristic and parallel algorithms for tackling large data problems.

Graph problems are data-driven, i.e., the memory access pattern in graph algorithms is often irregular and highly dependent on the network structure. Unlike spatial-based data clustering algorithms where the similarity of two data points can be determined by their distance, most graph algorithms must traverse the edges to calculate the affinity of a vertex to another. Thus, scalable performance can be difficult to achieve for a parallel algorithm because the graph structure is not known a priori [16]. In addition, graph clustering algorithms are iterative in nature with a high degree of data dependency. While there have been a few parallel graph clustering algorithms proposed recently, they suffer from frequent process synchronization and their result quality is affected by the processing order of vertex assignment to communities [1].

We propose a distributed-memory based parallel algorithm called PMEP which parallelizes MEP, a community detection algorithm based on the idea of maximizing equilibrium and purity of communities [23]. MEP has been demonstrated to produce high quality of results for medium to large graphs. To parallelize MEP, we use a data partitioning strategy that duplicates a portion of computational workload in exchange for a lower communication cost required in the later stage when combing local results into a global one. This strategy is motivated by the fact that graph problems are highly data dependent and it is unlikely for a data partitioner to produce subgraphs that can be processed independently on multiple processes without incurring a high cost of synchronization and communication. We employ the Parallel METIS (ParMETIS) graph partitioner [12,13] to break a graph into K subgraphs and assign each subgraph pair of all possible pairwise combinations to one of P processes, where $P = \binom{K}{2}$. This pairwise subgraph partitioning approach assigns each subgraph to $(K-1)$ processes, resulting in $(K-2)$ duplicated computation for processing the subgraph. Once received the assigned subgraph pairs, each process performs MEP on local data independently from other processes. The partial clustering results are then combined by resolving the conflicts on the community memberships found across all processes, which requires only one synchronization.

We used both synthetic and real-world data to evaluate PMEP. Using a synthetic data with 2 million vertices and 35.3 million edges, we achieved a speedup of 126.95 when running PMEP on 190 MPI processes. We evaluated the real-world data collected from Youtube using up to 1225 MPI processes and achieved a speedup of 204.22. We also compare the scalability of PMEP against the MPI implementation of parallel Louvain and observe that PMEP delivers a better performance.

2 Related Work

Fortunato gives a thorough overview of representative community detection algorithms, of which modularity-based methods are the most popular [7]. Since maximizing the graph modularity was proven to be an NP-complete problem [4], there have been several greedy approaches proposed [2,5,19]. Other than modularity-based methods, Zardi et al. introduced MEP, an algorithm that aims to maximize the equilibrium of communities [23]. This technique does not suffer from "resolution limit" problem that small communities are absorbed into large communities, an issue commonly seen in most of the modularity-based methods.

Riedy et al. use maximal matching to solve the parallel modularity maximization problem [17], based on CNM algorithm proposed by Clauset, Newman, and Moore [5]. This parallel approach was implemented using OpenMP and achieved a maximal speedup of 13× on a Cray-XMT shared-memory machine using 80 compute cores and uk-2002 graph data set [3]. Louvain is another popular algorithm that addresses CNM drawbacks by using a hierarchical extraction process [2]. The majority of parallel implementations for Louvain are using OpenMP for shared-memory machines. Bhowmick and Srinivasan proposed a heuristic to eliminate some computations that can be implicitly obtained by computing the modularity [1]. Their OpenMP implementation creates a need for critical sections, which eventually limits the scalability. Staudt and Meyerhenke [18] parallelized the Louvain algorithm using an ensemble learning technique that combines multiple base classifiers or weak classifiers to form a strong classifier, as a preprocessing step. Lu et al. [15] use heuristics of coloring and vertex following to parallelize the Louvain algorithm. All of the above approaches achieved the maximal speedups of 8× on 32 threads.

Wickramaarachchi et al. [21] implemented a distributed memory parallel Louvain algorithm using MPI and achieved the best speedup of 5× on 128 processes. Similar to our approach, they also used a graph partitioning method. However, they only parallelized the first stage of Louvain.

3 MEP Algorithm

The Maximizing Equilibrium and Purity algorithm (MEP) is a community detection algorithm that identifies a community based on its internal and external connectivity [23]. Let $G = (V, E)$ be an undirected graph, where V and E are the sets of vertices and edges, respectively. MEP partitions G into k communities $C = \{C_1, \ldots, C_k\}$, where $\forall\ i, C_i \subseteq V$ and $\forall\ i \neq j, C_i \cap C_j = \emptyset$. In other words, $C_1, \ldots, C_k$ are non-overlapping communities. The computation of MEP consists of two phases: **region growing** and **community merging**. The region growing phase starts with each vertex as a community containing only itself and grows the communities based on the connectivity of vertices. The communities identified in this phase are the locally optimal solutions, which will later be examined and possibly combined in the merging phase.

In the rest of this paper, we refer $N(v)$ as the set of vertices that have edges directly connecting to vertex v. We denote $d(v)$ as the cardinality (or degree)

Algorithm 1. Region Growing Phase
Input: graph $G = (V, E)$
Output: Communities $C = \{C_1, C_2, \ldots, C_k\}$

```
 1: for each $v_i \in V$ do
 2:     $C_i \leftarrow \{v_i\}$                          ▷ vertex starts out as a singleton
 3: Create $F$, a free list, and add all vertices into $F$
 4: Sort the vertices in $F$ based on their degrees
 5: while $F \neq \emptyset$ do
 6:     Select $v$ from $F$ with the highest degree
 7:     Delete $v$ from $F$
 8:     $\mathrm{NFDN}(v) \leftarrow 0$
 9:     $\mathrm{comp}(v, 1 \cdots N(v)) \leftarrow 0$
10:     for each $u \in N(v)$ do
11:         if $u$ is free then
12:             $\mathrm{NFDN}(v) \leftarrow \mathrm{NFDN}(v) + 1$
13:         else
14:             $\mathrm{comp}(v, C_u) \leftarrow \mathrm{comp}(v, C_u) + 1$
15:     Find $C_x$ whose $\mathrm{comp}(v, C_x)$ is the maximal
16:     if $\mathrm{NFDN}(v) \leq \mathrm{comp}(v, C_x)$ then
17:         Add $v$ to $C_x$
18:     else
19:         RecuriveGrowth($N(v)$, $C_v$, $F$)                ▷ grows $C_v$
```

of v and $d(v) = |N(v)|$. A vertex v is referred as a *free* vertex, if it does not belong to any community but itself. The number of free, direct neighbors of vertex v is denoted as $\mathrm{NFDN}(v) = |\{u|u \in N(v) \wedge u \text{ is free}\}|$. A vertex is said to be *compatible* to a community C_i if most of its direct neighbors are in C_i. Equation 1 defines the compatibility of vertex v to community C_i. The maximum compatibility of vertex v is the maximum among its compatibilities to all communities and its NFDN, as shown in Eq. 2. A vertex is defined as *pure* to a community C_i if and only if its compatibility to C_i is equal to its maximum compatibility.

$$comp(v, C_i) = |\{(v, u)|u \in N(v) \wedge u \in C_i\}| \tag{1}$$

$$comp_{max}(v) = \max\{\max_{C_i \in C} comp(v, C_i), \mathrm{NFDN}(v)\} \tag{2}$$

$$v \text{ is pure to } C_i \text{ iff } comp(v, C_i) = comp_{max}(v) \tag{3}$$

3.1 Region Growing Phase

Algorithm 1 presents the region growing phase. Initially, all vertices start out as singleton communities and are marked as free in list F. The vertices in F are then sorted by their degrees in an increasing order. The algorithm grows communities starting from the vertex with the highest degree, v. If v's maximal compatibility is larger than $\mathrm{NFDN}(v)$, then v is added to the community that has the maximal compatibility. Otherwise, the algorithm will grow C_v by adding direct neighbors of v if they are pure to C_v. For each newly added members, the algorithm recursively adds pure neighbors of those members (indirect connected neighbors of v) to grow C_v. The vertices are removed from the free list F once they were added to a community. This process iterates until the direct neighbors of v are exhausted, at which point the algorithm moves on to the vertex with next highest degree in F. At the end, the region growing phase produces an initial set of communities $C = \{C_1, C_2, \ldots, C_k\}$.

The community initialization in line 2 of Algorithm 1 takes $O(|V|)$ time. In line 4, we sort the vertices by their degree using a counting sort in $O(|V|)$ time. Starting from the vertex with the highest degree in F, we compute its compatibilities to the communities to which its (non-free) direct neighbors belong in lines 10–14. This takes $O(d)$ time. In the worst case, the time becomes $O(\Delta)$, where Δ as the maximum degree of a graph from the most connected vertex in the graph. The next step is to check the purity by comparing the maximal compatibility currently found against the number of free direct neighbors. If the number of free direct neighbors is larger, then we need to recursively check the purity for all the neighbors in line 19. The number of iterations to call procedure RecursiveGrowth is O(d) and in the worst case $O(\Delta)$. Finding the maximal compatibility for each neighbor in line 3 takes $O(|V|)$ time. Thus, each call to RecursiveGrowth takes $O(\Delta|V|)$ time, which makes the complexity of entire recursive call O($\Delta^2|V|$). Assuming the while loop in line 5 repeats ℓ_1 times and $1 \leq \ell_1 \leq |V|/2$. The overall complexity of the region grow phase is O($\ell_1\Delta^2|V|$). The worst case, $\ell_1 = |V|/2$, happens when each vertex in the graph is connected to only one other vertex, making the complexity become O($\Delta^2|V|^2$). The best case, $\ell_1 = 1$, happens when every vertex is connected to every other vertex i.e. the graph being a clique, which makes the complexity become O($\Delta^2|V|$). Therefore, the complexity of region grow depends on how close the neighbors of a vertex v form a clique. This quantity can be measured and is commonly referred to the *local clustering coefficient*, denoted as *lcc*. A high average local clustering coefficient *alcc* of a graph is an indicator of the presence of dense subgraphs [20]. The *alcc* values range from 0 to 1. Value of ℓ_1 decreases as *alcc* value approaches to 1 and increases as *alcc* approaches to 0.

The region-growing phase exhibits a high degree of data dependency, because the processing of vertex v depends on the results of processing all the vertices with higher degrees than v. However, this priori information is not known. Such a high degree of data dependency exhibited in graph problems in general makes the parallelization of any graph clustering algorithm an extremely challenging task.

3.2 Community Merge Phase

This phase first checks whether the initial communities found in the region growing phase are in *equilibrium* or not. The concept of equilibrium is simply the definition of strong communities. A community is strong if it has more internal connections (also defined as *compactness*) than the average external connections (also defined as *separation*). Equation 4 defines *compactness* of a community as the number of edges within the community and Eq. 5 defines *separation* of two

Algorithm 2. Recursive Growth
Input: a set of vertices N, Community C, and free list F
Output: updated C and F

```
Procedure RecursiveGrowth($N, C, F$)
1: $newN \leftarrow \emptyset$
2: for each $u \in N$ do
3:     Find $comp_{max}(u)$
4:     if $comp(u, C) = comp_{max}(u)$ then                ▷ if $u$ is pure
5:         Add $u$ to $C$ and delete $u$ from $F$
6:         Add $u$ to $newN$
7: if $newN \neq \emptyset$ then
8:     RecursiveGrowth($newN, C, F$)
```

communities as the number of edges between them. Equation 6 defines the average separability of a community. Equation 7 describes the equilibrium condition for a community, i.e. its average separation over all other communities is less than its compactness. Communities that are not in equilibrium will be merged to the community with which it has the highest separation. After the merge, the overall purity of a community may decrease. In this case, the impure vertices are moved to communities in which they are pure. Algorithm 3 describes the community merge phase.

$$compact(C_i) = |\{(v,u)|(v,u) \in E, v \in C_i \wedge u \in C_i\}| \tag{4}$$

$$sep(C_i, C_j) = |\{(v,u)|(v,u) \in E, v \in C_i \wedge u \in C_j\}| \tag{5}$$

$$sep_{avg}(C_i) = \frac{1}{|C|} \sum_{j=1 \wedge j \neq i}^{|C|} sep(C_i, C_j) \tag{6}$$

$$sep_{avg}(C_i) < compact(C_i) \tag{7}$$

We denote $|C'|$ to be the number of communites after the region growing phase. Algorithm 3 initially populates a $|C'| \times |C'|$ matrix with the pairwise separability and calculates the compactness for each community in C'. It iterates through all the edges of the graph, which takes $O(|E|)$ time. Finding the maximum and average separability for each community, as well as updating the matrix for any merged community (lines 8–12), take $O(|C'|)$ time, for a total of $O(|C'|^2)$ time per iteration of the `while` loop in line 5. The algorithm iterates through all the vertices after each merge round for reassigning the membership (lines 15–18), at a cost of $O(|E|)$. While this `while` loop could potentially iterate $O(|C'|)$ times, we find empirically that the number of iterations is a small constant. If the number of iterations is ℓ_2, then the overall complexity for the merge phase becomes $O(\ell_2(|C'|^2 + |E|))$. This phase of the MEP algorithm has a high data dependency on the order of communities being processed, because when a community pair is merged, all connections with those two communities need to be updated.

Algorithm 3. Community Merge Phase
Input: graph $G = (V, E)$
Input: initial communities $C = \{C_1, C_2, \ldots, C_k\}$
Output: modified communities $C = \{C_1, C_2, \ldots, C_{k'}\}$

```
 1: for each $C_i \in C$ do
 2:     Calculate $compact(C_i)$
 3:     for each $C_j \in C \wedge j \neq i$ do
 4:         Calculate $sep(C_i, C_j)$
 5: while true do
 6:     merge_count $\leftarrow$ 0
 7:     for each $C_i \in C$ do
 8:         if $sep_{avg}(C_i) > compact(C_i)$ then
 9:             Find $C_j$ in $C$ where $sep(C_i, C_j)$ is maximal
10:             Merge $C_i$ and $C_j$ into a new $C_{k'}$
11:             Delete $C_i$ and $C_j$ from $C$
12:             Update $compact(C_{k'})$ and $sep(C_{k'})$
13:             Add $C_{k'}$ to C
14:             merge_count $\leftarrow$ merge_count + 1
15:             for each $v$ in $C_{k'}$ do
16:                 if $comp(v, C_{k'}) \neq comp_{max}(v)$ then
17:                     Find $C_t$ in $C$ that has $comp_{max}(v)$
18:                     Add $v$ to $C_t$ and delete $v$ from $C_{k'}$
19:     if merge_count = 0 then
20:         break the while loop
```

4 Design and Implementation

Conventional parallelization strategies often consist of three steps: breaking the problem domain into a set of subproblems, solving subproblems independently and concurrently, and combining the subproblem solutions into a global solution for the original problem instance. Following the same principle, our parallelization divides a graph into subgraphs, detects communities within each subgraph independently using MEP, and merges the local communities to get the global solution. Contrary to the conventional approach that often seeks to generate non-overlapped subproblems, we adopt a data partitioning method that duplicates workload among processes. Our idea is motivated by the fact that graph problems are highly data dependent and it is unlikely for a data partitioner to produce subgraphs that can be processed independently without a high cost of process synchronization and communication at a later stage. For instance, if a non-overlapping partitioning is used, then the local results computed in each process must be sent to all other processes for merging, because any subgraph may have external edges connecting to all other subgraphs. Such a complete all-to-all personalized communication may be required multiple times when the algorithm traverses edges across multiple subgraphs to grow a community. Thus, a high communication cost is inevitable for such an approach. Owing to this, we design a strategy that duplicates a portion of computational workload among processes in exchange for a lower communication cost and hence heavier local computational workload. Our parallel algorithm consists of the following phases:

4.1 Parallel Read

The input graph data is stored in a file of compressed storage format (CSR), a widely used text format for storing graphs. In a CSR file, each line corresponds to a vertex and its adjacency list (vertex IDs of direct neighbors). This format is understood by ParMETIS [11], the data partitioner employed by our parallel algorithm (discussed in the next section). To enable parallel I/O, we convert the input file into a binary form but in the same data layout. During this off-line conversion, we also calculate the file starting offsets of adjacent lists and store the offsets in a separate file. In our parallel read phase, we partition vertices evenly into disjoint blocks among all processes. At first, all processes read the total number of vertices and edges to calculate the ranges of vertices to be assigned to individual processes. Through the offset file, each process can perform a file seek operation to jump to the file location containing the vertex subset to be read. We use an MPI collective read to read the graph data in parallel. A low cost is expected for this phase, as the I/O pattern from the above partitioning method is known to be highly scalable on state-of-the-art parallel file systems.

4.2 Graph Partitioning

There are several graph partitioning techniques proposed in the literature [8]. A high-quality partitioner can produce subgraphs that are well connected within each subgraph and fewer edges between them. Our parallel algorithm employs the Parallel METIS (ParMETIS) graph partitioner. METIS is a multilevel partitioning algorithm that produces high quality partitions by minimizing the resulting inter-subdomain connectivity and enforcing contiguous partitions [12,13]. Implemented using MPI, ParMETIS partitions a graph into K disjoint subgraphs in parallel, given K as a user input parameter.

Given P compute processes, one naive parallelization strategy is to partition a graph into P subgraphs and assign each subgraph to a process. However, when using this approach, the external edges between two subgraphs cannot be used to grow communities during the local computation, as processes possess no vertex data on the remote subgraphs connected through those external edges. To continue growing or merging the local communities, the intermediate results must be distributed among processes, which could involve multiple levels of data synchronization. Because a subgraph may have external edges connecting to all other subgraphs, a high communication cost is anticipated if this naive approach is used. To avoid such problem, we choose to duplicate a portion of computational workload in exchange for a lower communication complexity.

Our data partitioning method assigns every possible combination of subgraph pairs to a unique process. In this approach, the external edges between every two subgraphs can be used by a process to grow the communities. Given a graph and P processes, we call the ParMETIS library subroutine `ParMETIS_V3_PartKway` to partition the graph into K subgraphs such that $P = \binom{K}{2}$. A process is assigned all edges in subgraphs k_i, and k_j, along with the edges between them. This strategy requires the number of processes P to be $\binom{K}{2}$. For instance, when $K = 2$ we have one process, which is the serial case. When $K = 10$, our parallel program must run on $\binom{10}{2} = 45$ processes. This subgraph partitioning approach assigns each subgraph to exactly $(K - 1)$ processes, resulting in duplicated computation for detecting communities within the subgraph. This is the cost we intend to trade for achieving a lower communication cost later on.

The subroutine `ParMETIS_V3_PartKway` consists of three phases: graph coarsening, initial partitioning, and refinement. According to [11], coarsening and refinement take $O(|E|)$ time with $O(\log(|V|))$ stages. The partitioning phase takes $O(|E'|)$ time, where E' is the number of edges in the coarsened graph, and scales relative to $\sqrt{P}$. This partitioning function and hence our data partitioning phase takes $O(|E| \log(|V|) + |E'|/\sqrt{P})$ time.

4.3 Subgraph ID Distribution

The output of `ParMETIS_V3_PartKway` is an array in each process containing the subgraph IDs for the local vertices. Thus only the subgraph IDs of the local vertices assigned in the read phase are known. To achieve the pairwise subgraph duplication, we must also obtain the subgraph IDs for the vertices in the local vertices' adjacency lists. The subgraph IDs will be used to calculate the ranks of processes to which a vertex and its edges are to be duplicated. Because vertices are divided among processes in a block fashion in the read phase, the rank of a process that possesses the subgraph ID of a given vertex can be calculated by simply dividing the vertex ID by the block size.

In order to minimize the communication, we sort each list based on the vertex IDs and remove repeated vertices. The inter-process communication is carried out in three steps. First, an `MPI_Alltoall` is called to exchange the number of vertices to be sent and received among all processes. Next, send and receive buffers, one for each remote process, are allocated and a hash table lookup is performed to fill the send buffer with the requested subgraph IDs. The last step uses asynchronous communication calls (*isend* and *irecv*) to complete the communication. On average, each process is assigned $|E|/P$ edges and out of which $|V|/P$ vertices' subgraph IDs are already known. Thus, the complexity of this phase in terms of communication is $O(|E|/P)$.

4.4 Pairwise Subgraph Duplication

Given K subgraphs, there are $\binom{K}{2}$ combinations of subgraph pairs. We assign each process a pair of subgraphs along with the internal and external edges connecting the pair. If the two vertice of an edge belong to the same subgraph, the edge is internal. Otherwise the edge is external. An internal edge is identified when the subgraph IDs of its two vertices agree. To assign an external edge to a process, we use Eq. 8 to calculate the process rank p, where k_i and k_j

are subgraph IDs of the edge's two vertices, respectively. Using our duplication scheme, an external edge will be assigned to one and only one process and an internal edge is assigned in duplication to $(K-1)$ processes.

$$p = k_j * (k_j + 1)/2 - k_i - 1, \quad \text{for} \quad j > i \tag{8}$$

To start the duplication, each process first scans and packs all the edges from its local adjacency lists to send buffers if they are for remote processes. Adopting the similar communication method used in the previous phase, we call MPI asynchronous *isend/irecv* functions to distribute the workload. Edge scanning and packing takes $O(|E|/P)$ time. Assuming ParMETIS evenly divides the edges into K subgraphs, there are at most $O(|E|/K)$ external edges between each subgraph pair. Therefore, the complexity of this phase is $O(|E|/\sqrt{P})$ as $K = \sqrt{P}$.

4.5 Local Graph Construction

We use adjacency lists to represent the vertices and edges of the subgraph pair assigned to each process. To achieve a constant time for a vertex lookup in the later local MEP phase, we use a hash table to store the adjacency lists. The timing of creating a hash table depends on the efficiency of hashing function and the frequency of hash collision. In our implementation we use Jenkins' hash[1]. Assuming that it takes $O(h)$ time for adding an edge to the hash table, the time complexity of this phase is $O(h|E|/P)$, as each process is assigned $O(|E|/P)$ edges on average.

4.6 Local Region Growing

We implement the region growing phase of MEP algorithm using a union-find data structure to keep track of a vertex's community membership [6]. We also store the maximum compatibility (Eq. 3) of each vertex denoted as its *purity* and use it later in the global resolution phase to finalize the vertex's membership. The sequential complexity of this phase is $O(\ell_1 \Delta^2 |V|)$. Because of our duplication strategy, each subgraph is duplicated in $\sqrt{P}$ processes and thus the complexity of this phase is $O(\ell_1 \Delta^2 |V|/\sqrt{P})$.

4.7 Local Community Merge

We implement the community merge phase of MEP algorithm using a sparse community matrix M to represent the number of edges within and between the communities. As mentioned in Sect. 3.2, a vertex may change membership upon merging. When this happens, we update the vertex's *purity* value, which will be used in the next phase for resolving membership conflicts. As each process has $O(|E|/\sqrt{P})$ edges and $|C'|/P$ communities, where $|C'|$ denotes the number of communities found after the region growing phase, the time complexity of this phase is $O(\ell_2(|C'|^2/P + |E|/\sqrt{P}))$.

[1] http://burtleburtle.net/bob/.

4.8 Global Resolution

The locally detected communities are to be merged globally. Because each vertex is assigned to $(K-1)$ processes, the memberships calculated by different processes may disagree. When such conflicts occur, we resolve them based on the vertex's purity. We divide this global resolution task among all processes based on the vertex IDs, in a block partitioning fashion. In other words, process rank i is responsible for vertices of IDs from $(|V|/P) \cdot i$ to $(|V|/P) \cdot (i+1)$. All processes only redistribute the vertices' purities and their root IDs, using MPI asynchronous communication (*isend* and *irecv*). The overall communication message size exchanged among processes is $2|V|$ integers.

Each process receives $(K-1)$ purities and root IDs for each of $|V|/P$ vertices it is responsible. To resolve a conflict, we let the community with higher purity win the conflict. Essentially, we treat the purity as the support of a vertex to a community. When root IDs differ but the purities are equal, we assign the vertex to the community with a larger root ID. Since the operation of finding the maximum is both associative and commutative, this strategy ensures the convergence, no matter in what order the resolution is performed on the partial results. Once all conflicts are resolved, we use an MPI collective write function to write the community IDs to a shared file in parallel. The computation time complexity of this phase is $O(K|V|/P) = O(|V|/\sqrt{P})$ and communication complexity is $O(|V|/P)$.

4.9 Complexity Analysis

The overall complexity of PMEP becomes $O(|E|log(|V|) + \ell_1 \varDelta^2|V|/\sqrt{P} + \ell_2(|C'|^2/P + |E|/\sqrt{P}))$, which corresponds to the graph partitioning and local MEP phases. Our complexity analysis implies that the computation time of PMEP is to be dominated by these two phases.

5 Experiments and Performance Evaluation

We implement PMEP in C using Message Passing Interface (MPI) for communication and I/O. Our experiments were carried out on Hopper, a Cray XE6 supercomputer at the National Energy Research Scientific Computing (NERSC) Center. Each compute node on Hopper contains two twelve-core AMD Magny-Cours 2.1-GHz processors and 32 GB of memory. We use both real-world and synthetic graph data sets. Table 1 provides some graph properties of the data sets used in our experiments, which include number of vertices ($|V|$), number of edges ($|E|$), maximum degree ($\varDelta$), number of ground truth communities ($|C|$), and average local clustering coefficient (*alcc*). Note that higher the *alcc* value, denser the graph [20].

5.1 Synthetic Graphs

For a better control on the quality of community results with various graph properties, we synthesize four large graphs, $g1$, $g2$, $g3$, and $g4$, with ground truth

Table 1. Graph properties of data set used in our experiments.

Graph	$\lvert V\rvert$	$\lvert E\rvert$	Δ	$\lvert C\rvert$	alcc
g1	2.00 M	35.37 M	88	39,685	0.111
g2	2.00 M	35.06 M	88	35,318	0.299
g3	6.00 M	88.88 M	75	122,750	0.299
g4	6.00 M	88.85 M	75	122,471	0.587
Youtube	1.13 M	2.99 M	28,754	8,385	0.081

M: million

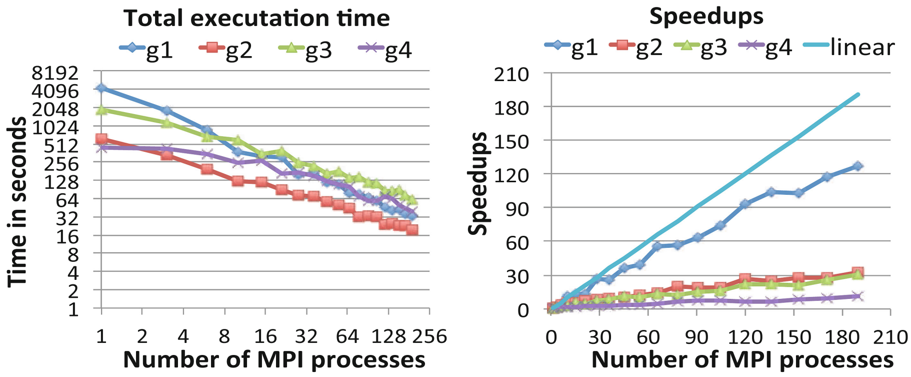

Fig. 1. Execution time and speedups for synthetic graphs (Color figure online).

using LFR benchmark [14]. The graphs become denser from $g1$ to $g4$ as the *alcc* values increase. The LFR benchmark allows us to set *alcc* values by changing the fraction of edges a vertex shares with others in different communities, while keeping other parameters constant, such as $|V|$, $|E|$, and Δ. Figure 1 presents the execution time and speedup of PMEP. Given two graphs of the same size, we observe that lower the *alcc* value higher the execution time. For example, $g1$'s $alcc = 0.111$ is lower than $g2$'s 0.299 and $g1$ has a higher execution time than $g2$. Similarly, $g3$ has a smaller *alcc* value than $g4$ and thus takes more time to complete. This performance trend matched our complexity analysis in Sect. 3 that the lower *alcc* value corresponds to higher values in Δ, ℓ_1, ℓ_2, and $|C'|$. Given a fixed *alcc* value, the execution time increases as the number of vertices and edges. In our experiment, graphs $g2$ and $g3$ have the same *alcc* value, 0.299, and because $g3$ has more vertices and edges, its execution time is higher than $g2$.

Among the four graphs, we observe that $g1$ scales much better than the rest and $g4$'s speedups are the worst. To help understand the differences, we collected the timing breakdown for individual phases of PMEP. In Fig. 2, the upper four charts show the percentages of timing for all phases and the bottom four charts show the speedups for the top three phases that dominate the overall execution time. From the percentage charts, we can see that the top three phases are the

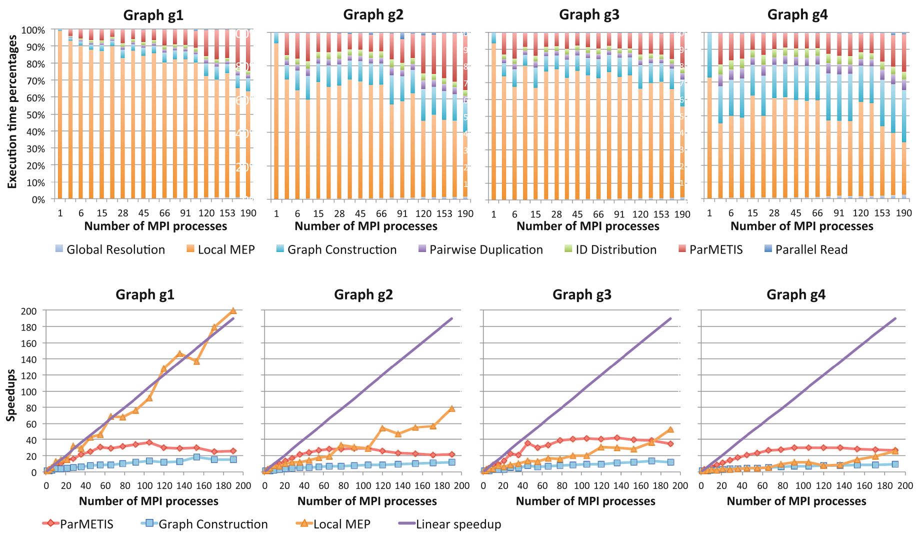

Fig. 2. Timing breakdown and speedup for individual phases of PMEP. (Color figure online)

local MEP, ParMETIS, and graph construction. For $g1$, the local MEP takes about 64 % to 93 % of the total time and because its speedup curve is quite close to the linear line, the overall speedups follow the similar trend. The best speedup for $g1$ is 126.95 when running 190 MPI processes.

For graphs $g2$, $g3$, and $g4$, the non-MEP phases start to take larger portions of the total time. However, since the local MEP still shows the dominant percentages in most of the cases, its scalability remains a strong influence to the overall speedup. The top three speedup charts show that the local MEP achieves lower speedups than the ones in $g1$'s chart and similar trends for the other two phases. The lower speedups altogether from the top three phases explain the lower overall speedups for $g2$, $g3$, and $g4$.

The high timing percentage of local MEP phase in $g1$ can be explained by its lower *alcc* value. As discussed in Sect. 3, a smaller *alcc* corresponds to a larger ℓ_1, meaning more iterations required on checking the compatibility and purity of a vertex's neighbors. In addition, the sparser graph $g1$ produces more fragmented intermediate communities $|C'|$ and hence a larger ℓ_2. Therefore, small *alcc* in $g1$ makes both the region growing and community merging of the local MEP phase the most expensive phase. This behavior is consistent with the local MEP's complexity analysis.

As the number of processes increases, the timing percentages of graph partitioning phase (ParMETIS) increases proportionally, taking a significant percentage of the overall execution time. From its speedup charts, the scalability of ParMETIS flattens quickly as the number of processes reaches to 40. Recall that ParMETIS consists of three phases, among which only the initial partitioning phase scales with respect to $\sqrt{P}$, and the other two remain constant regardless of P. As the number of processes increases, the two non-scalable phases start to dominate and hence explain the speedup curve. This behavior implies the PMEP's overall performance could be limited by the scalability of ParMETIS.

The local graph construction phase noticeably takes a larger portion of the overall time for dense graphs. In addition, the percentages increase significantly as the number of processes. As we build the vertex adjacency lists into a hash table, the timing depends on the efficiency of the hash function and the frequency of hash collision. For dense graphs, a high number of hash collisions is expected because more vertices sharing the same neighbors. When using vertex IDs as hash keys, there is a high chance for a densely connected graph that the same keys (vertices) are used when inserting new edges to the hash table. The effect of increasing cost on hash collisions can be seen when comparing the percentages of $g2$ to $g1$ and $g4$ to $g3$.

The subgraph ID distribution phase occupies only a small fraction of the overall execution time and the main cost of this phase is the communication. Its complexity in term of communication amount is $O(|E|/P)$ per process. When there are sufficiently large workloads, this phase appears negligible compared to other phases. Both the pairwise subgraph duplication and global resolution phases are mainly communication tasks. The complexity of pairwise duplication phase $O(|E|/\sqrt{P})$ is the worst case scenario and in the real timing results show much smaller, as seen from the timing percentage breakdown charts for all the synthetic graphs. The global reduction phase also occupies very a small percentage of the execution time. The small communication amount complexity $O(|V|/P)$ explains the observed results. The I/O cost takes less than 1 % of the overall time. As we use MPI collective I/O to read/write files stored in the Lustre parallel file system on Hopper, the low I/O cost is expected.

5.2 Real World Data Set

The real-world dataset, *youtube*, used in our experiments was obtained from the collection of the Stanford Large Network Dataset Collection[2]. Although there are other real-world graphs that come with the ground truth communities, but most of them do not contain disjoint communities. This graph is considered sparse, it has very low *alcc* value 0.081. Figure 3 shows the overall execution time, speedups, percentage breakdown, and speedups of the top-three phases. The youtube graph has a small *alcc* value similar to the synthetic graph $g1$ and both numbers of vertices and edges are less than $g1$. However, the maximum degree ($\Delta = 28,754$) of youtube graph is much higher than all the synthetic graphs. According to the overall complexity derived in Sect. 4.9, such a high Δ value makes the local MEP workload much higher than the rest of phases. This effect can be seen from the breakdown percentage chart where the local MEP phase dominates the overall execution time. As the local MEP speedups show good scalability, the overall speedups follow the similar trend. The highest speedup

[2] http://snap.stanford.edu/data/index.html.

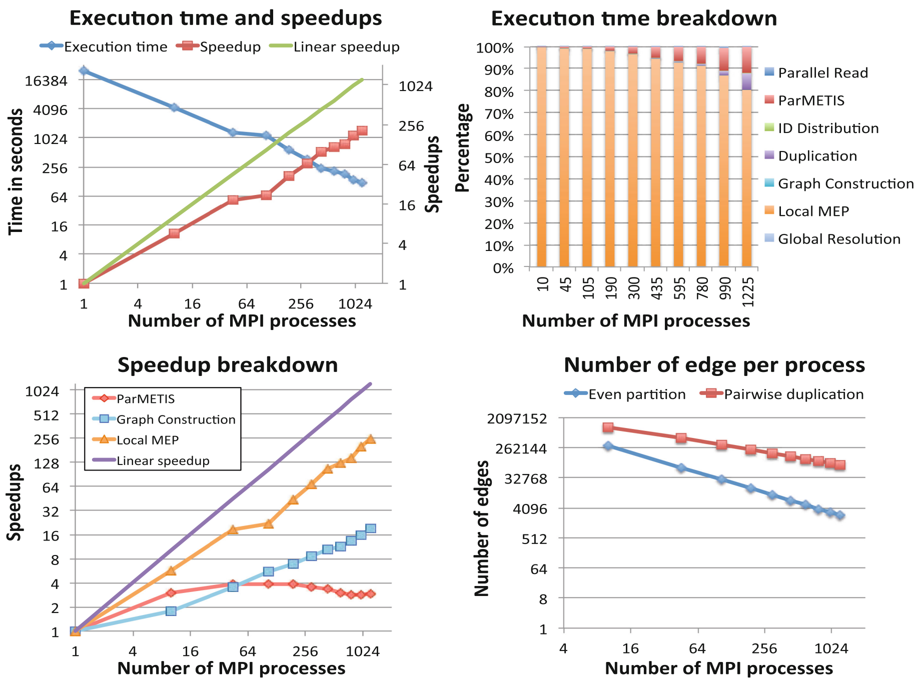

Fig. 3. Performance results of the youtube graph. (Color figure online)

achieved is 204.22 when running PMEP on 1225 MPI processes. Although this number is far from the linear speedup, we consider it a very good result for a parallel graph algorithm, given the fact that a high degree of data dependency in graph problems.

In Fig. 3, we also show the number of edges assigned to each process for our pairwise subgraph duplication strategy and the even data partitioning method. The number of edges per process for our approach is $O(|E|/\sqrt{P})$, while the even partitioning method is $O(|E|/P)$. As the number of processes increases, the chart shows the average number of edges of our approach deviates increasingly from the even partitioning method. This indicates that achieving a linear speedup is unlikely for PMEP if the computation workload of the graph problem is determined by the number of edges. However, our pairwise duplication approach produces a very small cost of inter-process communication, which is unlikely achievable by the even partitioning method.

5.3 Result Quality Analysis

In order to evaluate the quality of the community structures output from our algorithm against the known ground truth generated by the benchmark, we adopt a metric called adjusted rand index (ARI) [9]. Given two communities X and Y, the overlap between the two communities can be summarized by a contingency table where each entry t_{xy} denotes the number of vertices in

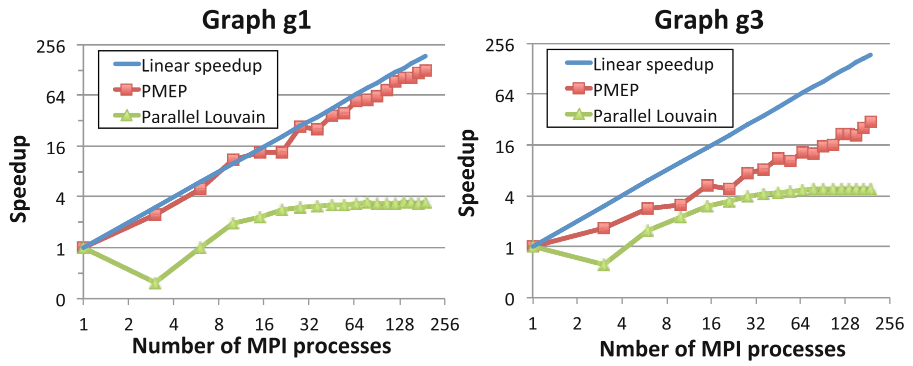

Fig. 4. Comparison between PMEP and parallel Louvain (Color figure online).

common between X and Y. ARI outputs a score ranging from -1 to 1, where 1 indicates that the two communities match perfectly and -1 indicates that two communities are in complete disagreement.

In terms of quality, we find PMEP results within 95 % accuracy to ground truth for both synthetic and real-world dataset. The quality results for graphs $g1$ to $g4$ for selected numbers of MPI process counts are shown in Table 2. These results demonstrate PMEP produces a high quality clustering solution when we increase the number of processes. Duplicating the internal edges implies that communities within the subgraphs are not split up, while still allowing to grow communities across multiple subgraphs. Thus, the heuristic used in our global resolution phase is shown to work well in producing high quality results.

5.4 Comparison with Parallel Louvain

We compare PMEP with the MPI implementation of parallel Louvain[3] [21]. Louvain is designed to minimize the value of modularity (Q), a popular metric in the graph community for measuring the strength of division of a graph into communities. Modularity is defined in Eq. 9, where e_i and a_i denote the fractions of internal and external edges in community C_i, respectively.

$$Q = \sum_{i=1}^{|C|} (e_i - a_i^2) \tag{9}$$

Although the design goal of PMEP is different (PMEP is to maximize equilibrium and purity), we are interested to see the scalability of the parallel Louvain and its comparison to PMEP. Note that a direct comparison in execution time is not fair because MEP and Louvain are two completely different algorithms. In our experiments, we set the stop condition ϵ to 0.1, meaning when the change of modularity value from the previous iteration increases to more than ϵ. In Fig. 4, we present the speedup chart for $g1$ and $g3$. PMEP scalability is significantly

[3] https://github.com/usc-cloud/parallel-louvain-modularity.

Table 2. ARI and modularity for synthetic graphs.

P	$g1$			$g2$			$g3$			$g4$		
	ARI	Q	Q(PL)	ARI	Q	Q(PL)	ARI	Q	Q(PL)	ARI	Q	Q(PL)
1	0.997	0.504	0.504	1.000	0.700	0.703	1.000	0.700	0.701	1.000	0.899	0.900
3	0.999	0.504	0.502	0.999	0.699	0.695	1.000	0.700	0.699	1.000	0.899	0.900
10	0.986	0.501	0.504	0.998	0.699	0.698	0.980	0.700	0.699	0.998	0.899	0.900
28	0.957	0.504	0.503	0.974	0.699	0.700	0.964	0.700	0.700	0.991	0.899	0.900
55	0.946	0.504	0.504	0.969	0.700	0.699	0.950	0.700	0.698	0.988	0.899	0.900
91	0.962	0.502	0.504	0.980	0.700	0.699	0.955	0.700	0.700	0.985	0.899	0.900
136	0.950	0.502	0.504	0.994	0.700	0.700	0.972	0.700	0.700	0.985	0.899	0.900
190	0.971	0.504	0.500	0.998	0.700	0.700	0.982	0.700	0.700	1.000	0.899	0.900

better for both graphs and in the meanwhile the parallel Louvain's speedup curve starts to flatten when P reaches 32. We note that parallel Louvain currently only parallelizes its first phase that computes the initial communities based on modularity maximization. To show that PMEP can also deliver high quality results in term of modularity, the measured values are provided in Table 2. We observe PMEP's modularity measures are consistent with parallel Louvain (PL).

6 Conclusion

Community detection for large graphs is extremely challenging due to a lack of *a priori* information of the graph structure and a high degree of data dependency. The scalability and quality of a parallel algorithm is significantly impacted by the data partitioning scheme it employs. Our proposed PMEP addresses this challenge by adopting a pairwise subdomain duplication partitioning approach that aims to trade some additional computation workload for significant reduction in communication cost. The experimental results show that PMEP successfully achieves this goal and in the meanwhile maintains a high quality of clustering results. Our future work include the investigation of the ParMETIS and graph construction phases, as they have shown poor scalability in the timing breakdown charts.

Acknowledgment. This work is supported in part by the following grants: NSF awards CCF-1029166, IIS-1343639, CCF-1409601; DOE awards DE-SC0007456, DE-SC0014330; AFOSR award FA9550-12-1-0458; NIST award 70NANB14H012; DARPA award N66001-15-C-4036.

References

1. Bansal, S., Bhowmick, S., Paymal, P.: Fast community detection for dynamic complex networks. In: Mangioni, G. (ed.) CompleNet 2010. CCIS, vol. 116, pp. 196–207. Springer, Heidelberg (2011)

2. Blondel, V.D., Guillaume, J.L., Lambiotte, R., Lefebvre, E.: Fast unfolding of communities in large networks. J. Stat. Mech: Theory Exp. **2008**(10), P10008 (2008)
3. Boldi, P., Codenotti, B., Santini, M., Vigna, S.: Ubicrawler: a scalable fully distributed web crawler. Softw.: Pract. Experience **34**(8), 711–726 (2004)
4. Brandes, U., Delling, D., Gaertler, M., Görke, R., Hoefer, M., Nikoloski, Z., Wagner, D.: On finding graph clusterings with maximum modularity. In: Brandstädt, A., Kratsch, D., Müller, H. (eds.) WG 2007. LNCS, vol. 4769, pp. 121–132. Springer, Heidelberg (2007)
5. Clauset, A., Newman, M.E.J., Moore, C.: Finding community structure in very large networks. Phys. Rev. E **70**(6), 066111 (2004)
6. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. The MIT Press, Cambridge (2009)
7. Fortunato, S.: Community detection in graphs. Phys. Rep. **486**(3–5), 75–174 (2010)
8. Hendrickson, B., Kolda, T.G.: Graph partitioning models for parallel computing. Parallel Comput. **26**(12), 1519–1534 (2000)
9. Hubert, L., Arabie, P.: Comparing partitions. J. Classif. **2**(1), 193–218 (1985)
10. Jain, A.K., Murty, M.N., Flynn, P.J.: Data clustering: a review (1999)
11. Karypis, G., Kumar, V.: Parallel multilevel k-way partitioning scheme for irregular graphs. In: Proceedings of the 1996 ACM/IEEE Conference on Supercomputing (1996)
12. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. **20**(1), 359–392 (1998)
13. Karypis, G., Kumar, V.: Multilevel k-way partitioning scheme for irregular graphs. Parallel Distrib. Comput. **48**(1), 96–129 (1998)
14. Lancichinetti, A., Fortunato, S., Radicchi, F.: Benchmark graphs for testing community detection algorithms. Phys. Rev. E Stat. Nonlin. Soft Matter Phys. **78**(4 Pt 2), 046110 (2008)
15. Lu, H., Halappanavar, M., Kalyanaraman, A., Choudhury, S.: Parallel heuristics for scalable community detection. In: Proceedings of the International Workshop on Multithreaded Architectures and Applications (MTAAP), IPDPS Workshops (2014)
16. Meyerhenke, H., Gehweiler, J.: On dynamic graph partitioning and graph clustering using diffusion. In: Algorithm Engineering. Dagstuhl Seminar Proceedings, vol. 10261 (2010)
17. Riedy, E.J., Meyerhenke, H., Ediger, D., Bader, D.A.: Parallel community detection for massive graphs. In: Graph Partitioning and Graph Clustering, pp. 207–222 (2012)
18. Staudt, C., Meyerhenke, H.: Engineering high-performance community detection heuristics for massive graphs. In: ICPP, pp. 180–189 (2013)
19. Wakita, K., Tsurumi, T.: Finding community structure in mega-scale social networks:[extended abstract]. In: Proceedings of the 16th International Conference on World Wide Web, pp. 1275–1276. ACM (2007)
20. Watts, D.J., Strogatz, S.H.: Collective dynamics of'small-world'networks. Nature **393**(6684), 409–10 (1998)
21. Wickramaarachchi, C., Frincu, M., Small, P., Prasanna, V.: Fast parallel algorithm for unfolding of communities in large graphs. In: 2014 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–6, September 2014
22. Zafarani, R., Liu, H.: Social computing data repository at arizona state university. School Comput. Inf. Decis. Syst. Eng. (2009)
23. Zardi, H., Romdhane, L.B.: An $o(n^2)$ algorithm for detecting communities of unbalanced sizes in large scale social networks. Know.-Based Syst. **37**, 19–36 (2013)