Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

Energy Efficient Permanence-based Community Detection Algorithm

Hardik Saini* | Vivek Kumar | Tanmoy Chakraborty

[1]Department of Computer Science & Engineering, IIIT-Delhi, New Delhi, India

Correspondence
*Hardik Saini, IIIT-Delhi. Email: hardik18391@iiitd.ac.in

Abstract

Detecting an accurate community structure is a crucial task in network analysis. With the increasing popularity of social networking sites, it's essential to have a community detection algorithm that is not only efficient but also cost-effective for running in data centres. There are several metrics for estimating the accuracy of community detection. However, previous research[1] has shown that permanence, a vertex-centric metric, provides the most precise estimate of a community structure compared to other approaches. Despite this, no study has been conducted on parallelizing a permanence-based community detection algorithm and analyzing its energy efficiency.

This paper introduces Amoeba, a task parallel implementation of a permanence-based community detection algorithm designed for multicore processors. It uses dynamic tasking to schedule the inherent irregular computation, and it can dynamically adapt the total number of parallel threads, which results in improved energy efficiency. We evaluated Amoeba using several real-world and artificial graphs on a multicore server processor. Our experimental results show that Amoeba achieves a geometric mean speedup of 15.3× over its sequential implementation, and due to thread adaptability, it achieves energy savings of 12.4% and a speedup of 6% over its non-adaptive implementation.

KEYWORDS:
Community detection; Permanence; Task-parallelism; Energy efficiency;

1 | INTRODUCTION

Complex systems such as social, biological and technological comprise entities/agents and their interactions. These interactions often show non-trivial characteristics such as structural homogeneity, indicating that nodes with similar properties interact more often than those with different properties. These interactions lead to modular structures within the network, where similar nodes form dense blocks known as communities. A community is a group of nodes that are closely connected to each other internally and have sparse external connections. Community detection has been a significant area of research over the past two decades due to the complexity and diversity of network structures. Most efficient methods for community detection are built on optimizing specific objective functions such as modularity[2], conductance[3], cut-ratio[4], and permanence[5]. Recent research suggests that permanence is a superior metric for community detection[1]. MaxPerm[5] is an algorithm that aims to maximize the permanence

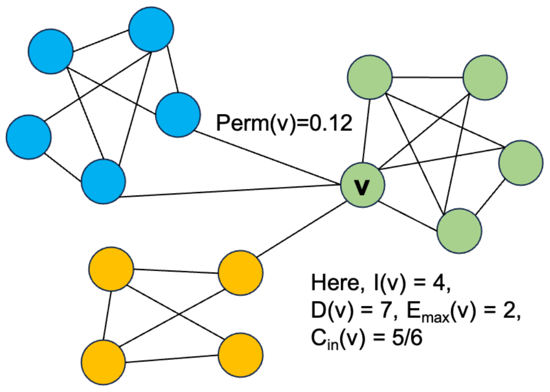


Figure 1 Toy example depicting permanence of a vertex v

of disjoint communities. MaxPerm has emerged as the benchmark algorithm for community detection, outperforming popular graph clustering methods such as Louvain[6], Infomod[7], Infomap[8], and WalkTrap[9].

With the increasing popularity of social networking sites, the community detection algorithm's performance and energy efficiency are equally crucial as these sites rely on large-scale data centres to compute the communities. Permanence being a vertex-centric metric, it is a highly irregular computation as each vertex can have a different amount of parallelism. Due to this reason, it is hard to estimate the optimal thread count to compute the permanence of a given graph in parallel. Having more threads than needed would lead to high energy utilization, whereas fewer threads could inflate execution time. MaxPerm is a sequential algorithm that computes the permanence of vertices one by one in every iteration. This paper proposes a novel parallel algorithm for MaxPerm called Amoeba. It uses dynamic task parallelism for load-balancing the irregular computation at the vertices of a graph. Amoeba tweaks the original iterative MaxPerm implementation so that the vertex with higher parallelism is processed earlier in an iteration than those with lesser parallelism. Amoeba then leverages this opportunity to adapt the total number of active threads participating in the parallel computation. At the start of the iteration, it activates all the available threads as the parallelism is at its maximum. As the parallelism decreases throughout the remaining execution of the iteration, Amoeba periodically decreases or increases the total number of active threads by monitoring the impact on energy utilization. We choose a wide variety of synthetic and real-world graphs to evaluate Amoeba's accuracy, performance, and energy efficiency on a modern server-class multicore processor with 64 hardware contexts. We show that Amoeba achieves accuracy at par with the original sequential MaxPerm. Further, due to thread adaptivity, Amoeba can outperform a non-adaptive parallel implementation in terms of execution time and energy efficiency.

In summary, this paper makes the following contributions:

- *Amoeba*, a novel parallel algorithm for permanence maximization over multicore processors based on dynamic task parallelism.
- A portable implementation for dynamically changing the thread count based on the available parallelism.
- Experimental evaluation of Amoeba on an AMD EPYC 7551 multicore processor using four synthetic and six real-world graphs.

The rest of the paper is structured as follows. Section 2 provides the relevant background. Section 3 discusses our evaluation methodology. Section 4 provides the motivation for Amoeba. Section 5 explains the design and implementation of Amoeba. Section 6 discusses the performance evaluation of Amoeba. Section 7 explains the related work, and finally, Section 8 concludes the paper.

2 | BACKGROUND

2.1 | Permanence

Community detection algorithms, such as modularity or conductance, use designated parameters to assign vertices to communities. However, these algorithms will always produce a set of communities, even without an inherent community structure. These

```
void MergeSort(int Low,int High) {
  if ((High - Low) < THRESHOLD) {
    SequentialSort(Low, High);
    return;
  }
  int Mid = (High + Low)/2;
  finish([=]() {
    async([=]() {
      MergeSort(Low, Mid);
    });
    MergeSort(Mid, High);
  });
  Merge(Low, Mid, High);
}
```

(a) Parallel MergeSort using the `async−finish` APIs

```
void VecorAddition(int Low, int High) {
 finish([=]() {
  forasync(Low, High, TILE, [=](int i) {
    C[i] = A[i] + B[i];
  });
 });
}
```

(b) Parallel vector addition using the `forasync-finish` APIs

Figure 2 Examples to demonstrate the task-based parallel programming model by using the `HClib` APIs

algorithms primarily make a choice by arbitrarily breaking ties. In contrast, permanence is a *vertex-centric* metric that encounters fewer tie-breaking situations and considers the maximum number of external connections to any neighbouring community when assigning vertices to communities. Permanence also takes into account the internal clustering coefficient to determine how tightly connected vertices are within their assigned communities. The internal connections of a community are generally considered together as a whole. However, how strongly a vertex is connected to its internal neighbours can differ. To measure this internal connectedness of a vertex, we compute the clustering coefficient of the vertex with respect to its internal neighbours. The higher this internal clustering coefficient, the more tightly the vertex is connected to its community. The permanence of a vertex ranges from 1 (strongly connected) to -1 (weakly connected or possibly wrongly assigned), with a value of 0 indicating equal pull from neighbouring communities. The pull in the metric is modelled as the maximum number of external connections to any neighbouring community. The permanence of a network is the ratio of the sum of the permanence of all vertices and the total number of vertices. It indicates to what extent, on average, the vertices of a network are bound to their communities.

$$Perm(v) = \underbrace{\frac{I(v)}{E_{max}(v)}}_{F1} \times \underbrace{\frac{1}{D(v)}}_{F2} - \underbrace{(1 - C_{in}(v))}_{F3} \tag{1}$$

Equation 1 is used to calculate the permanence of a vertex v. This calculation takes into account several factors, such as $I(v)$, which represents the total number of internal connections of v, $E_{max}(v)$, which is the maximum number of external connections of v to any neighbouring community, $D(v)$, which is the total degree of v, and $C_{in}(v)$, which represents the internal clustering coefficient of v. $F1$, as part of Equation 1, ensures that the vertex v has more internal pull than external pull, and is normalized by the total degree of v ($F2$) to ensure that the product of F1 and F2 falls between 0 (no internal connections) and 1 (no external connections). Additionally, within a specific community, the internal neighbours of the vertex v should be highly connected (high internal clustering coefficient). F3 imposes a penalty based on a low internal clustering coefficient, ranging from 0 (no penalty) to 1 (maximum penalty). To illustrate this concept further, Figure 1 shows a toy example for measuring the permanence of v.

2.2 | Dynamic Task Parallelism

This section provides an overview of the Dynamic Task Parallelism (*DTP*) and Habanero C/C++ Library (`HClib`)[10] used in this paper for the implementation of Amoeba. In DTP, a programmer explicitly exposes the parallelism in the form of tasks that can execute independently. An underlying work-stealing runtime then schedules these tasks across the hardware, keeping idle hardware busy while relieving the overloaded hardware of its burden. This programming model is preferred for parallelizing computations on a multicore processor where the static mapping of tasks to threads cannot provide optimal performance. DTP was made popular by Cilk[11] language, and it is now supported by a wide variety of frameworks, such as Java fork/join[12], Intel CilkPlus[13], Intel TBB[14], OpenMP[15], and `HClib`. Although the APIs to expose the tasks differ across each implementation, they all internally use a work-stealing[16] runtime for scheduling the tasks to threads.

We used DTP supported by HClib to implement Amoeba because of its high performance and easy-to-use APIs. We compared HClib's performance of `forasync` to TBB and OpenMP in Section 4. `HClib` is a library-based implementation that uses C++11

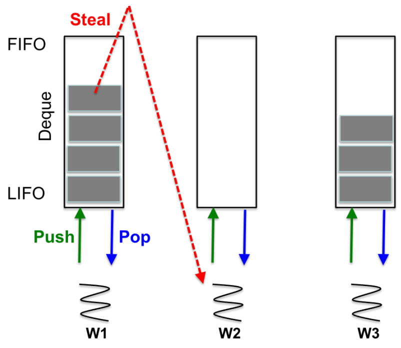


Figure 3 Work-stealing uses a pool of worker threads where each worker maintains a local set of `async` tasks in its deque using push/pop operations. Once a worker runs out of tasks on its deque, it finds a task by stealing from the deque of a random worker

lambda functions in its APIs. An `async` API in `HClib` creates a new task `S1` that can run in parallel to task `S2`. A `finish` is a generalized join operation, and it ensures that both parallel tasks `S1` and `S2` has completed before the execution of statements after the end of the `finish` scope. Both `async` and `finish` can be arbitrarily nested. `forasync` is another parallel API supported by `HClib` that recursively divides a `for` loop iterations into two halves that can execute in parallel. Figure 2(a) and Figure 2(b) shows the pseudocode of two parallel programs using `HClib` APIs. Underlined code shows `HClib` APIs. Figure 2(a) is a parallel recursive merge sort that uses `async` and `finish` APIs. Figure 2(b) shows an implementation of a parallel vector addition program using the `forasync-finish` APIs. The `forasync` API recursively divides a `for` loop iterations into two halves and launches them in parallel using `async` API. The default threshold for the `TILE` is set to one, but programmers can adjust it to control task granularity.

Like Cilk, Intel TBB, and other DTP frameworks, `HClib` also uses a work-stealing runtime for dynamic load-balancing (Figure 3). It schedules work exposed by the programmer (`async`), exploiting idle processors and unburdening those overloaded. Work-stealing uses a pool of worker threads where each worker maintains a local set of `async` tasks in its `deque` using push/pop operations. A worker (victim) remains *busy* as long as it has tasks on its deque. Once a worker becomes idle, it becomes a thief and continuously *searches* for a task (`async`) to execute by stealing from a random worker (victim). Work-stealing maintains the locality as steals are generally rare, and push/pop of local tasks are performed in LIFO order by a victim.

3 | EXPERIMENTAL METHODOLOGY

Before presenting the motivating analysis for Amoeba, we first describe our experimental methodology.

3.1 | Benchmark graphs and MaxPerm implementations

Our study used six Real-World Graphs (R)[17] and four Lancichinetti–Fortunato–Radicchi (LFR)[18] benchmark-generated graphs (Table 1). Real-world graphs were chosen based on increasing degree values, while LFR graphs were selected based on vertices and degrees encompassing our real-world graphs. Each `L1` and `L2` has two variants generated by varying the mut parameter (`m`) as 0.1 and 0.6. Increasing the m increases the ratio of inter-community to intra-community edges in the network, thereby increasing the difficulty in community detection. Hence, our four synthetic graphs are named `L1-V54-D4-m0.1`, `L1-V54-D4-m0.6`, `L2-V1-D58-m0.1`, and `L2-V1-D58-m0.6`. We will refer to the state-of-the-art *sequential* MaxPerm implementation[5] as *Default*. We have made changes in Default to implement *Amoeba*, which can dynamically adapt the total number of threads. We have also implemented the non-adaptive variant of Amoeba, which we will refer to as AmoebaStatic. We did not compare the performance of Amoeba with any other community detection algorithms as it would not be a fair comparison. There are several other algorithms for community detection, but they could have different execution times as they are calculated using different metrics. This approach is consistent with practices adopted by other researchers[19,20,21,22,23,24].

Graphs (Actual Name)	Vertices	Degree	Graphs (Short Name)
Lancichinetti Fortunato Radicchi (LFR)	54001	4	L1-V54-D4-m0.1
Lancichinetti Fortunato Radicchi (LFR)	54001	4	L1-V54-D4-m0.6
Lancichinetti Fortunato Radicchi (LFR)	1801	58	L2-V1-D58-m0.1
Lancichinetti Fortunato Radicchi (LFR)	1801	58	L2-V1-D58-m0.6
email-enron-large.mtx	33697	5	R-V33-D5
fb-pages-media.edges	27917	7	R-V27-D7
soc-gemsec-HR.edges	54600	9	R-V54-D9
scc infect-dublin.mtx	10973	16	R-V10-D16
bio-HS-CX.edges	4500	24	R-V4-D24
socfb-Mississippi66.mtx	10522	58	R-V10-D58

Table 1 Graphs used in this paper. "R" denotes Real-World graphs and "L" denotes LFR graphs

3.2 | Experimental Setup

Our experiments were conducted on a 32-core AMD EPYC 7551 processor with two hardware contexts per physical core, totalling 64. This processor has a minimum frequency of 1.2GHz and a maximum frequency of 2GHz. We maintained the default system settings with the CPU frequency scaling governor set to `ondemand`, allowing it to select CPU frequencies based on the CPU load. The AMD EPYC 7551 is a NUMA processor with four NUMA domains and 64GB of RAM. To ensure optimal performance during parallel executions, we utilized the `numactl --interleave` command, which binds physical memory pages in a round-robin across all the NUMA nodes. The Operating System was Ubuntu 20.04.5 LTS, and we compiled our benchmarks using the LLVM compiler version 16.0.6 and the -O3 flag. We sourced the HClib implementation from its official GitHub repository, commit ID ab310a0, and used the oneAPI TBB version oneTBB 2021.9.0. Our parallel executions maintained a one-to-one mapping between application threads and hardware contexts. We utilized the LIKWID API[25] to measure processor energy consumption and monitored the processor package energy counter `RAPL_PKG_ENERGY:PWR1`. It is a socket-wide counter that measures the energy the entire processor uses (cores, shared caches, memory controllers, and interconnects). We executed each parallel execution of benchmark graphs eights times and reported the mean execution time.

4 | MOTIVATING ANALYSIS

As permanence is a vertex-centric metric, total computation at each vertex in a graph could differ depending on its total number of neighbours and the neighbouring community. A straightforward parallel implementation (AmoebaStatic) can thus lead to high energy usage and low performance. We describe it using a motivational analysis in this section. We begin our discussion by comparing the accuracy of MaxPerm with the popular modularity maximization. We then describe the performance and energy usage by using AmoebaStatic.

4.1 | Correspondence to Ground-Truth Structure

We analyzed the accuracy of communities obtained through the MaxPerm (Default) and modularity maximization[6]. Our evaluation was based on four synthetic graphs (Section 3), and we used Normalized Mutual Information (NMI)[26] to measure the accuracy of the detected communities against the ground-truth community structure. The results of this experiment are presented in Table 2. A lower value of m indicates a stronger community in the network. We observed that MaxPerm consistently outperformed modularity maximization, even when dealing with weaker community structures (higher m values).

Benchmark	L1-V54-D4-m0.1		L1-V54-D4-m0.6		L2-V1-D58-m0.1		L2-V1-D58-m0.6	
Algorithm	M	P	M	P	M	P	M	P
Accuracy	0.376	**0.963**	0.57	**0.72**	0.043	**0.647**	0.045	**0.323**

Table 2 Analysis comparing the accuracy of MaxPerm ("P") with modularity maximization ("M"). Higher the better. Large "m" implies poor community structure

Graphs	Default Execution (Minutes)	Speedup of AmoebaStatic over Default (Higher the better)			Energy Savings in AmoebaStatic over Default (Higher the better)		
		T=16	T=32	T=64	T=16	T=32	T=64
L1-V54-D4-m0.1	5.9	4.9	**5.4**	4	**67%**	61.4%	41.9%
L1-V54-D4-m0.6	11.2	9.9	**15.1**	13.3	83.2%	**85.3%**	81.8%
L2-V1-D58-m0.1	6.7	11.8	21.6	**26.5**	86.3%	89.7%	**90.6%**
L2-V1-D58-m0.6	2.9	10.6	18.1	**22.7**	85%	88.2%	**89.3%**
R-V33-D5	51	7.8	**10.6**	9.2	**79.5%**	79.1%	73.7%
R-V27-D7	10.4	9.5	**14.1**	13	83.1%	**84.6%**	81.7%
R-V54-D9	15.3	10.1	**15.5**	14.2	83.8%	**85.7%**	82.9%
R-V10-D16	2.1	8.5	**11.4**	10	81%	**81.1%**	76.5%
R-V4-D24	12.6	12.6	23.3	**27.2**	87.1%	90.1%	**90.8%**
R-V10-D58	107.9	12.3	22.9	**27.4**	86.9%	89.9%	**90.9%**

Table 3 Analysis comparing the speedup and energy savings in AmoebaStatic relative to Default at different thread count for different graphs

4.2 | Execution time and Energy Usage

We have modified the Default sequential implementation to use loop-level parallelism via the `forasync` APIs of HClib. In Section 5, we provide a detailed explanation of these modifications. One of the resulting parallel implementations, AmoebaStatic, is a straightforward implementation that does not adapt the thread count. Unlike Amoeba, it requires the user to specify the thread pool size. We have found that relying on the user-selected thread count may not always provide optimal performance and energy usage. To demonstrate this, we executed each graph using AmoebaStatic and varied the thread count while calculating the speedup and energy relative to the Default sequential implementation. The results of this experiment are presented in Table 3. For four graphs (L2-V1-D58-m0.1, L2-V1-D58-m0.6, R-V4-D24, and R-V10-D58), the optimal thread count was found to be 64 as it provided the best speedup and energy savings. For another group of graphs (L1-V54-D4-m0.6, R-V27-D7, R-V54-D9, and R-V10-D16), the optimal thread count was 32. As explained in Section 2.2, worker threads in a work-stealing runtime remain busy if they have tasks to execute. Otherwise, it randomly steals a task for a victim with extra tasks. Having more workers than needed increases idleness in the work-stealing runtime, which causes workers to attempt to steal more often. Frequent task movement across the NUMA domains hampers the locality, resulting in lower performance and high energy usage. The remaining two graphs (L1-V54-D4-m0.1 and R-V33-D5) achieved the best speedup when using 32 threads, but energy savings were maximum when executed with 16 threads. This behaviour suggests that the optimal thread count may change during execution for certain graphs.

These results show that employing more threads than the optimal number can lead to increased energy usage, while using fewer threads can increase execution time. Further, to validate our selection of HClib, we conducted a speedup comparison of HClib, Intel oneAPI TBB, and OpenMP implementations of AmoebaStatic. The geometric mean speedup relative to the Default sequential for (`HClib`, TBB, and OpenMP) are (9.5×, 9.1×, 7.7×) using 16 threads, (14.6×, 13.4×, 10.6×) using 32 threads, and (14.5×, 14.5×, 8.6×) using 64 threads. In Section 6 we have discussed the execution time of the graphs reported in Table 3.

5 | DESIGN AND IMPLEMENTATION

The previous sections identified the permanence calculation as a highly irregular computation and highlighted the inefficiency of using a straightforward DTP approach for permanence maximization. We approach the problem by automatically increasing and decreasing the number of active threads in Amoeba. The insight is that reverse sorting the vertices based on their total computation would reduce the number of parallel tasks over time. Hence, Amoeba can leverage this opportunity to dynamically control the total number of active threads by monitoring the energy wasted by idle threads. Our contribution is designing and implementing Amoeba, a new community detection algorithm that implements our above insight. While we used HClib for our implementation, Amoeba can be effectively utilized with any other DTP framework since we have not modified the HClib runtime.

5.1 | Permanence Maximization using DTP

Algorithm 1 demonstrates how to maximize permanence in Amoeba using DTP. The changes made to the original sequential MaxPerm implementation are highlighted with underlined code. Input to the MaxPerm algorithm is the graph for which net permanence and community structure need to be calculated. At the start, each vertex is assigned its seed community and added to an array (Line 2). The iterative implementation of permanence maximization calculates new permanence for each vertex until the difference in net permanence between two consecutive iterations is less than a specified threshold (Line 4, set to 2%). This iterative algorithm works, as mentioned below.

5.2 | Reverse sorting the vertices

The vertex array created at Line 2 is reverse sorted by Amoeba at Line 7. This sorting is based on the total computation at each vertex, which is determined by the product of the number of neighbours (`Neighbours(v)`) and neighbouring communities (`NeighbouringCommunity(v)`) for each vertex. During the first iteration of the `while` loop, the primary community structure is calculated, and sorting is avoided at Line 6. If K is 100%, it indicates that thread adaption will not occur (Section 5.4). The `while` loop iterations continue refining the communities generated from previous iterations. Reverse sorting produces an array where the total computation at each vertex decreases while iterating the vertex array from start to end (Line 9). Amoeba leverages this property to dynamically control the total active threads and improve overall energy utilization (Line 11). Thread adaption is triggered at fixed intervals (Line 10) to regulate the adaptation frequency (explained in Section 5.4).

Reverse sorting of the vertex array can lead to minor changes in the overall community structure, as certain vertices may relocate to a different community due to variations in tie-breaking. For instance, in the MaxPerm algorithm, a vertex v_x could potentially be grouped within the community of either vertices v_y or v_z. When not reverse sorted, the visitation order for the vertices is v_x, v_y, and v_z, resulting in v_x being grouped with v_y. On the contrary, when reverse sorted, the visitation order could become v_x, v_z, and v_y, thus leading to the grouping of v_x with v_z. However, the overall change in the net permanence due to reverse sorting is negligible, as demonstrated in our experimental evaluations (Section 6.1).

5.3 | Permanence of individual vertices

It is important to note that as a vertex's permanence increases, it indicates a rise in the number of internal connections and/or a decline in the number of external connections to its neighbouring communities. The primary objective of the MaxPerm algorithm is to attain high permanence values for each vertex. The Algorithm 1 achieves this by following a five-step process for every vertex in G, utilizing a `for` loop at Line 9. $Step-1$ calculates the vertex's permanence (Line 13), and if it has already reached maximum permanence, further processing stops (Line 16). If not, $Step-2$ calculates the neighbour's permanence (Line 20), and $Step-3$ moves the vertex to each neighbouring community, calculating its new permanence (Lines 23–24). As the movement affects the neighbours of the vertex, $Step-4$ calculates the new permanence of the neighbours (Lines 26–27). Finally, $Step-5$ (Lines 29–32) assesses whether the vertex should move to the new community or stay in its original community. The vertex moves to the new community if there is an improvement in its permanence and its neighbour's permanence (Line 29). All four steps, except Step-1, have been modified to introduce parallelism. The `for` loops inside Steps 2-4 have been parallelized using `finish-forasync` APIs, and thread-local variables have been used to avoid the use of locking APIs (not shown in Algorithm 1 for simplicity). Step-5 is sequential and is the same as in Default, highlighted as modified due to support for thread-local variables.

Algorithm 1 Permanence maximization algorithm in Amoeba

```
procedure MaxPerm($G(V, E)$)                                   ▷ Input is Graph G(V vertices, E edges)
    Add each vertex v from G in an array V and assign them unique communities
    $Perm_{\text{Old}} \leftarrow -1$; $Perm_{\text{New}} \leftarrow 0$; $Primary_{\text{Comm}} \leftarrow$ true
    while $(Perm_{\text{Old}} - Perm_{\text{New}}) > THRESHOLD$ do
        $Perm_{\text{Old}} \leftarrow Perm_{\text{New}}$; $Perm_{\text{New}} \leftarrow 0$;
        if not $Primary_{\text{Comm}}$ And $(K < 100)$ then                          ▷ K from Equation 2
            $Sort(V, TotalComputation(v))$                         ▷ Reverse sort vertices by computations
        end if
        for $v \in V$ do
            if $(!Primary_{\text{Comm}})$ And $(K < 100)$ And $(v\%V_{\text{Count}} == 0)$ then
                $AdaptAmoebaThreads()$
            end if
            $vPerm_{\text{CurrComm}} \leftarrow Permanence(v)$
            if $vPerm_{\text{CurrComm}} == 1$ then                       ▷ MaxPermanence attained by v is 1
                $Perm_{\text{New}} \mathrel{+}= 1$
                continue
            end if
            $vPerm_{\text{CurrNeigh}} \leftarrow 0$
            parallel_for $u \in \{Neighbors(v)\}$ do                ▷ Parallel for using finish & forasync
                $vPerm_{\text{CurrNeigh}} \mathrel{+}= Permanence(u)$
            end parallel_for
            parallel_for $C \in NeighbouringCommunity(v)$ do       ▷ Parallel for using finish & forasync
                Move v to New Community C
                $vPerm_{\text{NewComm}} \leftarrow Permanence(v)$
                $vPerm_{\text{NewNeigh}} \leftarrow 0$
                parallel_for $u \in \{Neighbors(v)\}$ do            ▷ Parallel for using finish & forasync
                    $vPerm_{\text{NewNeigh}} \mathrel{+}= Permanence(u)$
                end parallel_for
                if $(vPerm_{\text{CurrComm}} < vPerm_{\text{NewComm}})$ And $(vPerm_{\text{CurrNeigh}} < vPerm_{\text{NewNeigh}})$ then
                    $vPerm_{\text{CurrComm}} \leftarrow vPerm_{\text{NewComm}}$
                else
                    Replace v to its original community
                end if
            end parallel_for
            $Perm_{\text{New}} \mathrel{+}= vPerm_{\text{CurrComm}}$
        end for
        $AwakeAmoebaThreads()$
        $Primary_{\text{Comm}} \leftarrow$ false
    end while
    return $Perm_{\text{New}}/size(V)$                 ▷ Detected community structure & NetPermanence
end procedure
```

After executing these five steps for each vertex, Amoeba resets the number of active threads to the maximum at Line 37 before initiating a new `while` loop iteration.

5.4 | Dynamically adjusting thread count

The parallelism at a vertex can be calculated by multiplying its total number of neighbours by the neighbouring community. It could change across the `while` loop iterations (Line 4 in Algorithm 1) due to changes in the node's neighbouring communities,

Algorithm 2 Code for dynamically changing the total number of active workers in Amoeba

```
1:  procedure AdaptAmoebaThreads
2:      static $Action_{\text{Last}} \leftarrow DECREMENT$
3:      static $Workers_{\text{Active}} \leftarrow 2 \times N_{\text{Cores}}$
4:      $Workers_{\text{ToChange}} \leftarrow 2$
5:      if ($called\ for\ first\ time$) Or ($energy\ reduced\ since\ last\ call$) then
6:          if $Action_{\text{Last}} == DECREMENT$ then
7:              $sleep(Workers_{\text{ToChange}})$
8:          else
9:              $awake(Workers_{\text{ToChange}})$
10:         end if
11:     else
12:         if $Action_{\text{Last}} == DECREMENT$ then
13:             $awake(Workers_{\text{ToChange}})$
14:             $Action_{\text{Last}} \leftarrow INCREMENT$
15:         else
16:             $sleep(Workers_{\text{ToChange}})$
17:             $Action_{\text{Last}} \leftarrow DECREMENT$
18:         end if
19:     end if
20:     if $Action_{\text{Last}} == DECREMENT$ then
21:         $Workers_{\text{Active}} -= Workers_{\text{ToChange}}$
22:     else
23:         $Workers_{\text{Active}} += Workers_{\text{ToChange}}$
24:     end if
25: end procedure
```

but the *total parallelism* over the entire `while` loop iterations remains constant. Total parallelism will also differ with/without reverse sorting (Line 7 in Algorithm 1) as the reverse sorting could slightly alter the community structure (Section 5.2).

The degree of a vertex determines its total number of neighbours, which means that parallelism will increase with an increase in both degrees and the total number of vertices. As a result, a fixed frequency of invoking the thread adaptation routine cannot be used for all input graphs. The total number of active threads should be adapted less frequently for graphs with high parallelism and more frequently for those with lower parallelism. Additionally, the frequency of thread adaptation should depend on the total number of physical cores (N_{Cores}), occurring more frequently for high $N_{\text{Cores}}$and less frequently for low N_{Cores}. To implement this, we use specific criteria to invoke thread adaptation after every V_{Count}increase in total vertices processed (Line 10). For a given graph with an average degree of D and total vertices of V, launched on a machine with N_{Cores}number of physical cores, $V_{\text{Count}}$is calculated as $K\% \times V$, where K is as mentioned below:

$$K = min(max(0.25, \frac{D^{1+Q}}{N_{\text{Cores}}}), 100), where\ Q = \frac{D}{N_{\text{Cores}}} \tag{2}$$

The value of K is determined by a range with a lower limit of 0.25% and an upper limit that is dependent on both the degree and N_{Cores}, as demonstrated in Equation 2. An additional exponential weight, Q, is assigned to the degree to calculate the upper limit of K, as shown in the same equation. The Q value is adjusted dynamically to ensure fewer calls for thread adaptation are made when $N_{\text{Cores}}$is already low. For high N_{Cores}, the Q value is smaller, while for low N_{Cores}, it is larger. K's value increases exponentially when parallelism is estimated to be high in relation to the total physical core count, resulting in occasional thread adaptation. When parallelism is low, K is assigned a smaller value, resulting in frequent thread adaptation in Amoeba. When K is calculated as 100%, thread adaptation is unnecessary in Amoeba as the parallelism is estimated to be significantly higher. All the graphs used in this paper are shown in Table 4, along with their corresponding K values.

The routine $AdaptAmoebaThreads$ (Line 11) shown in Algorithm 2 controls the number of active threads. Amoeba sets the thread pool size in `HClib` to match the total hardware contexts (64 in our case) at the start of any graph execution (Line 3). As

Graphs	L1-V54-D4-m0.1	L1-V54-D4-m0.6	L2-V1-D58-m0.1	L2-V1-D58-m0.6	R-V33-D5	R-V27-D7	R-V54-D9	R-V10-D16	R-V4-D24	R-V10-D58
Net Permanence (NP) in Default	0.596	-0.520	0.108	-0.143	0.134	0.016	-0.181	0.437	-0.098	-0.052
NP in Amoeba over Default	Same	2% lesser	Same	Same	Same	3% more	Same	Same	3.3% more	Same
Total Parallelism in AmoebaStatic	4.9×10^7	12.1×10^7	5.8×10^7	7.7×10^7	23.8×10^7	12.9×10^7	16×10^7	2.5×10^7	6.3×10^7	108×10^7
K in Amoeba (Equation 1)	0.25%	0.25%	**100%**	**100%**	0.25%	0.33%	0.52%	2%	8.13%	**100%**

Table 4 Net permanence, total parallelism and value of K in Amoeba

explained in Section 5.1, the vertex array is reverse sorted by Amoeba to reduce the parallelism at each vertex with the increasing iterations of the for-loop (Line 9 in Algorithm 1). This approach aims to minimize the number of active threads as the loop progresses and limit the number of idle threads that would constantly query the underlying thread pool for tasks. To achieve this, $AdaptAmoebaThreads$ puts two active threads to sleep (Line 4) and calculates the difference in processor package energy consumption during the next invocation of this routine. If there is an increase in energy consumption, the previous decision is invalidated, and two idle threads are awakened (Line 23). However, if there is a decrease in energy consumption, the earlier decision is validated, and the active thread count is further reduced by two (Line 21). This cycle continues until the number of active threads is reset to match the total number of hardware contexts before starting the next iteration of the $while$ loop (Line 37 in Algorithm 1).

6 | EXPERIMENTAL EVALUATION

We now discuss the experimental evaluation of Amoeba. It frees the user from the task of determining the ideal thread count, which may differ from one graph to another (as detailed in Section 4). Instead, it sets the thread count to the maximum number of hardware contexts available (in our case, 64) and proceeds to dynamically evaluate the optimal thread count that delivers optimal performance and energy efficiency.

6.1 | Net Permanence

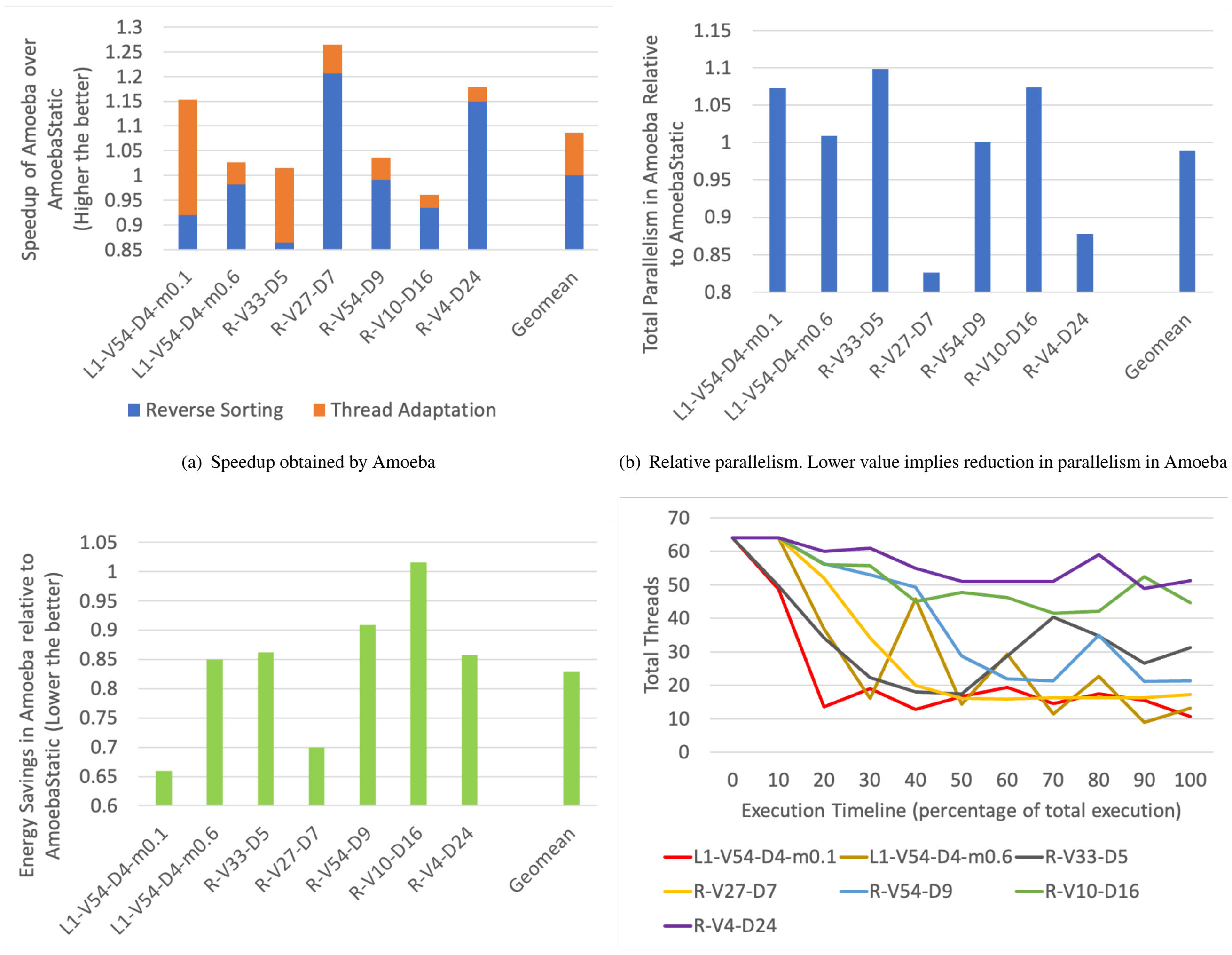
The NP calculated by Default and AmoebaStatic is identical for any given graph, ranging between -1 to $+1$, with a higher value indicating stronger community structure. Amoeba's reverse sorting of the vertex array may cause slight changes in the NP (as described in Section 5.2). However, the difference in NP is minimal and falls within the -2% to $+3.3\%$ range.

6.2 | Execution time

Table 3 demonstrates that the calculation of permanence is compute-intensive, with the sequential execution times ranging from 2 minutes to 1.8 hours. The execution time is influenced by the graph's degree, the number of vertices, and the NP value. It is possible to process a graph with a higher degree faster than one with a lower degree, as seen in the examples of R-V4-D24 and R-V33-D5. Higher NP values result in better community structure but also increase execution time, as the $while$ loop in Algorithm 1 takes more iterations to refine the communities. Therefore, graphs with high NP values and many vertices, such as R-V33-D5 and R-V27-D7, have longer execution times. However, due to the difference in vertex count, R-V10-D16 executes faster than R-V54-D9, even though the former has a higher NP value. R-V10-D58 has a high execution time (1.8 hours) because of its high degree. Increasing m in LFR graphs increases the ratio of inter to intra community edges, thereby resulting in poor community structure. It reduces execution time as the $while$ loop in Algorithm 1 would converge sooner. However, L1-V54-D4-m0.1 is an exception with a small degree and m value, leading to community assignment within a few iterations.

6.3 | Speedup and Energy Savings in Amoeba

Figure 4 reports the experimental evaluation of Amoeba compared to AmoebaStatic for all the graphs used in this paper (total threads set to 64), except for the three graphs where $K = 100\%$. In such cases, there is no difference in the execution of Amoeba



(a) Speedup obtained by Amoeba

(b) Relative parallelism. Lower value implies reduction in parallelism in Amoeba

(c) Energy savings in Amoeba

(d) Active thread count during execution

Figure 4 Experimental evaluation of Amoeba over AmoebaStatic. Results not reported for graphs having $K = 100\%$ (L2-V1-D58-m0.1, L2-V1-D58-m0.6, and R-V10-D58)

and AmoebaStatic, as the parallelism is estimated to be at its maximum (as stated in Section 5.4). Figure 4(a) shows the speedup achieved by Amoeba, factoring in contributions from reverse sorting (Section 5.1) and thread adaptation (Section 5.4) into the net speedup. The total parallelism in Amoeba and AmoebaStatic may differ due to reverse sorting (see Section 5.4 and Section 5.2). Table 4 shows the total parallelism in AmoebaStatic, and Figure 4(b) displays the relative parallelism in Amoeba. The speedup in Amoeba due to reverse sorting alone is inversely proportional to the change in parallelism.

The thread adaptation has the maximum speedup for L1-V54-D4-m0.1 and R-V33-D5 in Figure 4(a). As explained in Section 4.2, these graphs are where AmoebaStatic achieved maximum energy savings using 16 threads. Figure 4(d) confirms this observation, as Amoeba reduced the active thread count in these two graphs right after the execution started. Thread reduction helps to minimize the NUMA effect on our machine, which in turn improves parallel performance. Among all the graphs, Amoeba has the shortest execution time of 13.3 seconds for R-V10-D16, while the others range between 23.6 seconds (R-V4-D24) and 328.5 seconds (R-V33-D5). However, the *while* loop iterations inside R-V10-D16 are extremely short (Line 4 in Algorithm 1, $\approx$ 1.4 seconds), making it difficult for Amoeba to effectively measure the impact of thread adaptations, resulting in a slowdown of 4%. R-V4-D24 also has a relatively short execution time, but it can achieve speedup due to the reduction in parallelism from reverse sorting of vertices. Figure 4(c) illustrates the energy savings in Amoeba over AmoebaStatic, which is directly proportional to the speedup obtained by Amoeba over AmoebaStatic. Overall, across all ten graphs, Amoeba achieved a

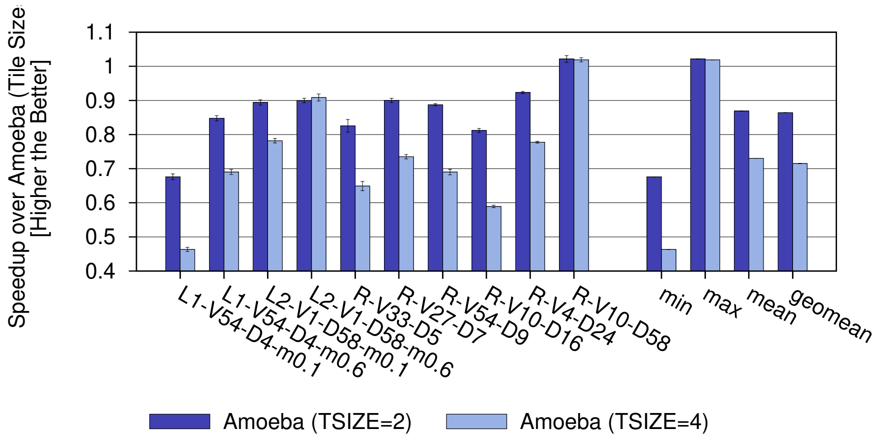


Figure 5 Amoeba does not benefit from large TILE_SIZE in `forasync`, as reverse sorting of vertices results in lesser computation with increasing iterations in `forasync`

geomean speedup and energy savings of 6% and 12.4%, respectively. Amoeba obtained a geomean speedup and energy savings of 15.3% and 85.4% over Default, respectively.

6.4 | Effect of Controlling Task Granularity

As mentioned in Section 5.1, Amoeba would not benefit by controlling task granularity in `forasync` APIs due to highly irregular parallelism at each vertex. To confirm this, we executed Amoeba using two different `TILE_SIZE` (2 and 4) and compared the respective execution time with the default setting that uses `TILE_SIZE=1`. This experiment is shown in Figure 5, indicating that increasing the task granularity leads to an increase in execution time.

7 | RELATED WORK

7.1 | Techniques for community detection

There are several other methods to detect communities other than Permanence, such as spectral graph partitioning[27], random walk[28], and consensus clustering[29]. Modularity[2] is one of the most widely used metrics. It measures a graph's structure, measuring the density of edges within a group or community. However, it suffers from the resolution limit problem where it tends to form large communities by merging smaller than a certain threshold[30]. SPart is a vertex-based metric that considers each vertex's contribution and its neighbour in the community formation[31]. Community score tries to detect communities on the principle that the density of internal edges in a community is higher than that of external edges[32]. We refer readers to[1] for an in-depth analysis of different community detection metrics.

7.2 | Multicore parallelism in community detection

Several prior works exist for multicore parallelism in community detection algorithms. Staudt *et al.*[33] proposed OpenMP[15] based parallel implementation for modularity maximization by using the Louvain method[6]. Lu *et al.*[34] implemented a parallel Louvain algorithm and proposed heuristics to overcome the original algorithm's inherent sequential barrier. Scalable Community detection (SCD) uses OpenMP to parallelize Weighted Community Clustering (WCC) on multicore processors for overlapping community detection[35]. Riedy *et al.*presented a highly parallel agglomerative implementation for the Clauset–Newman–Moore algorithm[23]. Kuzmin *et al.*[22] proposed a Speaker Listener Label Propagation (SLLP) algorithm for overlapping community detection that uses OpenMP and TBB for parallelization. GLEAM is another OpenMP based community detection framework[36]. Hydetect is a recent hybrid CPU-GPU implementation of parallel community detection using Louvain's algorithm[19].

FDT[37] and Varuna[38] are popular frameworks that also use online techniques to adapt the number of threads in a parallel program. FDT is suitable for data-parallel OpenMP work-sharing loops, whereas Varuna is programming model-independent and can work even with DTP applications. Like Amoeba, Varuna uses an exploration-based analytic model to determine the optimal number of threads for a given parallel program. However, unlike Varuna, as Amoeba is integrated inside the MaxPerm algorithm, Amoeba tweaks the iterative MaxPerm such that the vertices with higher parallelism are processed earlier in an iteration than those with lesser parallelism. Hence, Amoeba begins the iteration with maximum threads and then aims to decrease the thread count until that iteration finishes. Amoeba also avoids the exploration overheads by adapting the exploration frequency based on the network graph's degree and vertex count.

8 | CONCLUSION

There are several metrics for evaluating the quality of a community structure, among which permanence stands out as superior to other popular community detection metrics. However, the computation of permanence is expensive and can lead to an irregular workload. This paper presented a solution by implementing dynamic task parallelism to parallelize the permanence calculation. However, as this approach is prone to worker starvation and improper energy utilization, we proposed Amoeba, which tweaks the permanence calculation algorithm and dynamically adapts the active thread count based on available parallelism. Our empirical results show that Amoeba improves energy efficiency and overall performance. We plan to extend Amoeba's support for hybrid CPU-GPU computing in future work.

References

1. Chakraborty T, Dalmia A, Mukherjee A, Ganguly N. Metrics for community analysis: A survey. *ACM CSUR* 2017; 50(4): 1–37.
2. Newman ME. Modularity and community structure in networks. *Proceedings of the national academy of sciences* 2006; 103(23): 8577–8582.
3. Lu Z, Wahlström J, Nehorai A. Community detection in complex networks via clique conductance. *Scientific reports* 2018; 8(1): 1–16.
4. Leskovec J, Lang KJ, Mahoney M. Empirical comparison of algorithms for network community detection. In: ACM. ; 2010: 631–640.
5. Chakraborty T, Srinivasan S, Ganguly N, Mukherjee A, Bhowmick S. On the Permanence of Vertices in Network Communities. In: ACM. ; 2014: 1396–1405
6. Blondel VD, Guillaume JL, Lambiotte R, Lefebvre E. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment* 2008; 2008(10): P10008.
7. Rosvall M, Bergstrom CT. An information-theoretic framework for resolving community structure in complex networks. *National Academy of Sciences* 2007; 104(18): 7327–7331.
8. Rosvall M, Bergstrom CT. Maps of random walks on complex networks reveal community structure. *National Academy of Sciences* 2008; 105(4): 1118–1123.
9. Pons P, Latapy M. Computing communities in large networks using random walks. In: Springer. ; 2005: 284–293.
10. Kumar V, Zheng Y, Cavé V, Budimlić Z, Sarkar V. HabaneroUPC++: A Compiler-free PGAS Library. In: ACM. ; 2014: 5:1–5:10
11. Frigo M. Multithreaded Programming in Cilk. In: ACM. ; 2007: 13–14
12. Lea D. A Java Fork/Join Framework. In: ACM. ; 2000: 36–43
13. Robison AD. Composable Parallel Patterns with Intel Cilk Plus. *Computing in Science and Engg.* 2013; 15(2): 66–71. doi: 10.1109/MCSE.2013.21

14. Reinders J. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc. first ed. 2007.

15. OpenMP ARB . *OpenMP Application Programming Interface Version 5.0* . November 2018.

16. Frigo M, Leiserson CE, Randall KH. The implementation of the Cilk-5 multithreaded language. In: ACM. ; 1998: 212–223

17. Rossi RA, Ahmed NK. The Network Data Repository with Interactive Graph Analytics and Visualization. In: AAAI Press. ; 2015.

18. Lancichinetti A, Fortunato S, Radicchi F. Benchmark graphs for testing community detection algorithms. *Phys. Rev. E* 2008; 78: 046110. doi: 10.1103/PhysRevE.78.046110

19. Bhowmik A, Vadhiyar S. HyDetect: A Hybrid CPU-GPU Algorithm for Community Detection. In: IEEE. ; 2019: 2-11.

20. Ghosh S, Halappanavar M, Tumeo A, et al. Distributed louvain algorithm for graph community detection. In: IEEE. ; 2018: 885–895.

21. Riedy J, Bader DA, Meyerhenke H. Scalable multi-threaded community detection in social networks. In: IEEE. ; 2012: 1619–1628.

22. Kuzmin K, Shah SY, Szymanski BK. Parallel overlapping community detection with SLPA. In: IEEE. ; 2013: 204–212.

23. Riedy EJ, Meyerhenke H, Ediger D, Bader DA. Parallel community detection for massive graphs. In: Springer. ; 2011: 286–296.

24. Zhang Y, Wang J, Wang Y, Zhou L. Parallel community detection on large networks with propinquity dynamics. In: ACM. ; 2009: 997–1006.

25. Treibig J, Hager G, Wellein G. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In: IEEE. ; 2010: 207–216.

26. Danon L, Diaz-Guilera A, Duch J, Arenas A. Comparing community structure identification. *Journal of statistical mechanics: Theory and experiment* 2005; 2005(09): P09008.

27. Newman ME. Spectral methods for community detection and graph partitioning. *Physical Review E* 2013; 88(4): 042822.

28. Pons P, Latapy M. Computing communities in large networks using random walks. In: Springer. ; 2005: 284–293.

29. Lancichinetti A, Fortunato S. Consensus clustering in complex networks. *Scientific reports* 2012; 2: 336.

30. Fortunato S, Barthelemy M. Resolution limit in community detection. *Proceedings of the national academy of sciences* 2007; 104(1): 36–41.

31. Chira C, Gog A, Iclănzan D. Evolutionary detection of community structures in complex networks: A new fitness function. In: IEEE. ; 2012: 1–8.

32. Pizzuti C. A multiobjective genetic algorithm to find communities in complex networks. *IEEE Transactions on Evolutionary Computation* 2011; 16(3): 418–430.

33. Staudt CL, Meyerhenke H. Engineering Parallel Algorithms for Community Detection in Massive Networks. *IEEE Transactions on Parallel and Distributed Systems* 2016; 27(1): 171-184.

34. Lu H, Halappanavar M, Kalyanaraman A. Parallel heuristics for scalable community detection. *Parallel Computing* 2015; 47: 19 - 37. doi: https://doi.org/10.1016/j.parco.2015.03.003

35. Prat-Pérez A, Dominguez-Sal D, Larriba-Pey JL. High Quality, Scalable and Parallel Community Detection for Large Real Graphs. In: ACM. ; 2014: 225–236

36. Bu Z, Cao J, Li HJ, Gao G, Tao H. GLEAM: a graph clustering framework based on potential game optimization for large-scale social networks. *Knowledge and Information Systems* 2018. doi: 10.1007/s10115-017-1105-6

37. Suleman MA, Qureshi MK, Patt YN. Feedback-Driven Threading: Power-Efficient and High-Performance Execution of Multi-Threaded Workloads on CMPs. In: ACM. ; 2008: 277–286

38. Sridharan S, Gupta G, Sohi GS. Adaptive, Efficient, Parallel Execution of Parallel Programs. In: ACM. ; 2014: 169–180

$\square$