Analysis of the Semi-synchronous Approach to Large-scale Parallel Community Finding

Erika Duriakova, Neil Hurley
Insight Centre for Data Analytics
School of Computer Science and Informatics
University College Dublin
Dublin, Ireland
first.last@insight-centre.org

Deepak Ajwani, Alessandra Sala
Bell Laboratories, Dublin
Dublin, Ireland
first.last@alcatel-lucent.com

ABSTRACT

Community-finding in graphs is the process of identifying highly cohesive vertex subsets. Recently the vertex-centric approach has been found effective for scalable graph processing and is implemented in systems such as GraphLab and Pregel. In the vertex-centric approach, the analysis is decomposed into a set of local computations at each vertex of the graph, with results propagated to neighbours along the vertex's edges. Many community finding algorithms are amenable to this approach as they are based on the optimisation of an objective through a process of *iterative local update* (ILU), in which vertices are successively moved to the community of one of their neighbours in order to achieve the highest local gain in the quality of the objective. The sequential processing of such iterative algorithms generally benefits from an asynchronous approach, where a vertex update uses the most recent state as generated by the previous update of vertices in its neighbourhood. When vertices are distributed over a parallel machine, the asynchronous approach can encounter race conditions that impact on its performance and destroy the consistency of the results. Alternatively, a semi-synchronous approach ensures that only non-conflicting vertices are updated simultaneously. In this paper we study the semi-synchronous approach to ILU algorithms for community finding on social networks. Because of the heavy-tailed vertex distribution, the order in which vertex updates are applied in asynchronous ILU can greatly impact both convergence time and quality of the found communities. We study the impact of ordering on the distributed label propagation and modularity maximisation algorithms implemented on a shared-memory multicore architecture. We demonstrate that the semi-synchronous ILU approach is competitive in time and quality with the asynchronous approach, while allowing the analyst to maintain consistent control over update ordering. Thus, our implementation results in a more robust and predictable performance and provides control over the order in which the node labels are updated, which is crucial to obtaining the correct trade-off between running time and quality of communities on many graph classes.

COSN'14, October 1–2, 2014, Dublin, Ireland.

http://dx.doi.org/10.1145/2660460.2660474.

Categories and Subject Descriptors

G.2.2 [**DISCRETE MATHEMATICS**]: Graph Theory — Graph algorithms;
G.1.0 [**NUMERICAL ANALYSIS**]: General — Parallel algorithms

Keywords

Parallel Graph Algorithms; Community Detection Algorithms; Iterative Local Update; Semi-synchronous Graph Algorithms

1. INTRODUCTION

The problem of identifying communities in large and complex networks has received ample attention in recent years, owing to its application in the analysis of social networks, citation networks, the world-wide web, biological networks and so on. Many different definitions of community structure have been proposed and various algorithms have been developed to extract such structure from the complex networks that arise in real-world applications. Common to all definitions is the notion that a community is a cohesive subset of the nodes or vertices of the network graph. Such cohesive subsets often correspond to meaningful structure in the underlying real-world problem. When the network is a social network, they can be interpreted as social communities, but may be understood, in the case of biological networks for instance, as important functional units. Metrics such as modularity and conductance have been proposed to quantify the cohesiveness of communities. Other metrics, such as normalised mutual information, allow a community assignment to be compared against a set of ground-truth communities. (See [4] for a comprehensive overview.)

In the last decade, one of the most studied metrics for community detection is *modularity* [13] of a partition of the graph vertices into a set of non-overlapping communities. Modularity measures the cohesiveness of the given partition on the given graph against the cohesiveness of the same partition on a "null model" consisting of a random graph with the same degree distribution. An alternative approach [16] models community finding as an information theoretic problem of finding a set of modules or communities that minimise

the number of bits required in a particular encoding of a random walk over the graph, resulting in an objective they call the *map equation*. While in many ways very different models of community, both of these approaches propose an objective function over the set of possible partitions, whose optimum corresponds to the 'best' community structure under the model. A number of other methods also seek a community assignment to optimise a functional objective over the set of possible assignments, including methods for overlapping community detection, such as MOSES [12]. Given any such model, we face the computational task of proposing an algorithm to optimise its associated objective.

It transpires that among the best strategies for optimising any of these objectives, an algorithm based on *iterative local update* (ILU) has proven particularly effective, both in run-time performance and quality of the resulting communities. In this approach, the community assignment is initialised by assigning a unique community label to each vertex. Each vertex is visited in some order; the communities of its neighbours are considered and the vertex is moved to the neighbouring community that results in the largest improvement in the objective, if it exists. Several passes through the vertices are made, until no further moves generate an improvement in the objective, or more generally until a threshold α of vertex updates fails to be reached. This algorithm constitutes a single phase of the algorithm. In [1, 16], further phases are applied in which the network is rebuilt by merging connected nodes that share a community and the algorithm is re-applied on the community graph thus formed. The phases are repeated until no further improvement in the objective is obtained.

The most basic community-finding algorithm based on the ILU method described above is the *label propagation* (LP) algorithm [15]. LP does not depend on any explicit objective function, but rather, in each iteration, a vertex is assigned the most frequent label (or the heaviest label in case of weighted graph) among its neighbours and itself. Ties can be broken arbitrarily, randomly, or in favour of the maximum label. As above, this iterative process terminates at an iteration in which no more than a threshold α of vertices has moved. LP has emerged as a particularly effective technique, that has been shown to result in good quality community structure and has since been also used as a fundamental block in many other complex community detection ensemble schemes (e.g., [14]).

In many of the current *parallel* label-propagation implementations (e.g., [17]), the labels of the vertices are asynchronously updated. In the shared-memory context, this implies that whether an old or new label value of neighbours is read at the update stage depends on the architecture and systems-level details, such as when the cores flush their private caches to memory and how is the memory hierarchy shared between the different cores. This makes it impossible to control the sequential order in which the vertices are updated in the ILU algorithm for community detection and precludes any analysis of the outcome of the algorithm in terms of this update ordering.

In this paper, we demonstrate that, for social network graphs with highly non-uniform degree distributions, the vertex update ordering has a large impact on the run-time of the analysis and the quality of the resulting communities. It is thus a key parameter of the computation. This suggests that analysts wishing to explore the structural properties of the network, should be able to control this update order, independently of the details of the machine on which the algorithm is run. Even if the analyst prefers a certain amount of randomisation for the ensemble approaches to work better with the ILU based approaches as basic building blocks, she would prefer to control this randomisation and not leave such a crucial aspect to the architecture and systems-layer details. Thus we explore semi-synchronous parallel ILU algorithms that use an input graph colouring to *impose* a vertex update order. While the set of possible input orderings is restricted by the requirement that it be efficiently computable on the parallel machine, we address the question of how to choose a colouring given a particular desired update order and explore the convergence and run-time characteristics of different colouring strategies.

Our Contribution Differently from previous work on this topic, we explore the use of *independent sets* to compromise between processor data independence and good mixing of community labels during the course of the algorithm. This so-called *semi-synchronous* approach was previously proposed in [9] but not fully implemented and evaluated on a parallel architecture. Specifically,

- We extend the semi-synchronous approach to a general framework for community-finding using the ILU method that includes objectives such as modularity and the map equation.
- We perform a detailed empirical evaluation of various design parameters on the performance of ILU method and the quality of the resultant communities. We show that the vertex update ordering and the colouring technique has a significant effect on the outcome of the ILU algorithms.
- The empirical evaluation of design parameters enables us to carefully engineer the semi-synchronous ILU that is comparable in running time to the state-of-the-art asynchronous implementations and yet provides the consistent control of node update ordering to the developer. Our approach uses many optimisation tricks such as keeping track of active nodes and a blocked colouring strategy.

Outline Section 2 describes the iterative local update methodology and the synchronous, asynchronous and semi-synchronous techniques to parallelize it. Section 3 generalizes the semi-synchronous approach to parallelize iterative local update algorithms for optimising modularity and the map equation as well. After outlining the datasets and machine architectures on which we carry out our study, in Section 4, we examine the effect of different vertex visit orderings on the performance of an ILU algorithm in Section 5. Section 6 discusses vertex ordering based on graph colouring. Finally, Section 7 presents the results of our empirical analysis.

1.1 Related Work

While we focus in this paper on implementation on shared memory multicores, the ILU framework has formed the basis of so-called *vertex-centric* scalable graph processing systems such as Pregel [11], GraphLab [10] and GraphChi [8]. These systems solve graph-based data analytics problems by iterating the computation of a vertex kernel function over all vertices in the graph. Pregel and GraphLab can be

distinguished by the type of synchronisation they support. Pregel supports synchronous algorithms in which computations proceed in a set of *super-steps* followed by a synchronisation. Each vertex kernel running independently in parallel in a super-step cannot access any state changes made by other vertices until the end of the super-step. GraphLab supports asynchronous computations, where vertex kernels can modify state shared with neighbouring vertices, during an update sweep. Recognising the need for analysts to maintain consistency in their results, GraphLab enables *serialisable* asynchronous computations, ensuring that the parallel asynchronous computation is equivalent to some sequential update order.

The original GraphLab system supports a *Chromatic Engine*, analogous to the semi-synchronous approach we explore in this paper. This work was further extended in [7] using colouring techniques to acquire independent sets while also supporting dynamic data-graph computations. However, the recent PowerGraph [5] enhanced system relies instead on a dynamic locking mechanism to ensure that conflicting assesses to shared data are resolved. While this guarantees that the computation is serialisable, the equivalent sequential computation is difficult for the analyst to discern as it is dependent on the parallel architecture.

Another distributed implementation of a constrained variant of the LP algorithm has been used in the graph partitioning system for Facebook's "People You May Know" (PMYK) service [18]. None of these works has considered the semi-synchronous approach to parallelisation.

In other relevant state-of-the-art, different approaches to the parallelisation of LP and greedy modularity maximisation have been evaluated in [17]. In fact, some of the engineering tricks in our work (e.g., the use of *active/inactive* flag described in section 3.1) are similar to those considered in [17]. However the fundamental difference between the two works is that while the implementations in [17] are based on an asynchronous approach, our implementation is based on semi-synchronous ILU algorithms (algorithm framework 2). The engineering of the semi-synchronous algorithm necessitates the study of the impact of graph colouring techniques to identify independent sets and different node orderings, which was not considered in [17].

1.2 Notation

In the rest of the text, we use the following notations. A graph $G(V, E)$ consists of a set of vertices V of size n and a set of edges $E \subseteq V \times V$ of size m. In this paper we consider undirected graphs, such that $(v, w) \in E \Leftrightarrow (w, v) \in E$ with, optionally, an associated edge *weight* $w : E \to \mathbb{R}^+$, with $w(e) = 1 \, \forall e \in E$ in the *unweighted* case. Moreover, we write $\forall v \in V, N_v = \{w \in V | \exists e = (v, w)\}$ for the *neighbourhood* of v; $d_v = |N(v)|$ for the *degree* of v; and $k_v = \sum_{w \in N_v} w(e)$ for the *weighted degree* of v. With a node numbering $l : V \to \{1, \ldots, n\}$ giving a unique integer identifier $i = l(v)$ for each v, we can use i to represent v and interchangeably write k_i for k_v etc. The *adjacency matrix* of the graph is the $n \times n$ matrix, $\mathrm{A} = \{a_{ij}\}$, such that $a_{ij} = w(e)$ whenever $(i, j) \in E$ and $a_{ij} = 0$ otherwise. The output of a community finding algorithm may be represented as the assignment of a community label to each node in the graph, which we write as $\zeta : V \to \mathbb{N}$, where $\zeta(v)$ is the community label of node v. It will be convenient to write $\zeta(A)$ for the set of all labels on a subset of vertices A.

2. PARALLEL ILU

The algorithmic framework of the ILU method is shown in Algorithm 1. Due to its local update of the node labels, the ILU approach is considered to be well suited for a fast and scalable implementation. In the ILU method, the work carried out per vertex is a function of the vertex's neighbourhood size. A natural approach to parallelisation is to partition the vertices among the available processors, updating vertex labels in parallel. Provided the sum of the neighbourhood sizes in each partition is balanced, then the total work should be evenly balanced across the machine, with little interprocessor communication required in the label update. However, this analysis neglects the fact that in the serial algorithm, labels are updated in sequential order. As shown in Algorithm 1, the label applied to vertex v at iteration i, depends on the most recent labels obtained for those neighbours, $N^p(v)$, visited previously to v on the current iteration, and on the old labels for those neighbours, $N^s(v)$, visited later in the loop. In fact, the use of the most recent label values during update is critical to allow fast mixing of the labels, reducing the total number of required iterations for the algorithm to converge to a stable set of communities.

Algorithm 1 Iterative Local Update (ILU)

Input: $G(V, E)$, $\alpha \geq 0$.
1: Initialize labels: $\forall (v \in V), \zeta_0(v) \leftarrow l(v)$ ($l : V \to \{1, \ldots, n\}$ gives a unique integer identifier)
2: $i \leftarrow 0$, updated $\leftarrow n$
3: **while** updated $> \alpha$ **do**
4: $\quad i \leftarrow i + 1$, updated $\leftarrow 0$
5: $\quad$ **for** $\forall (v \in V)$ **do**
6: $\quad\quad \zeta_i(v) = f\left(\zeta_i(N^p(v)), \zeta_{i-1}(N^s(v))\right)$ ($f(.)$ is the local update function)
7: $\quad\quad$ **if** $\zeta_i(v) \neq \zeta_{i-1}(v)$ **then**
8: $\quad\quad\quad$ updated $\leftarrow$ updated $+ 1$
9: $\quad\quad$ **end if**
10: $\quad$ **end for**
11: **end while**
Output: $\forall \zeta_i(v), v \in V$

2.1 Levels of Synchronisation

We summarise the different strategies for parallelisation of an ILU algorithm for community-finding on a shared-memory multicore as follows:

Synchronous Parallelisation In this approach the labels $\zeta_i(v)$ are updated based on the label values from the previous iteration only i.e. line 7 of Algorithm 1 becomes $\zeta_i(v) = f\left(\zeta_{i-1}(N(v))\right)$ and parallelisation is achieved by replacing line 5 of Algorithm 1 with a `parallel for` to distribute the vertices among the shared threads for processing. This approach is preferable from the perspective of minimising shared memory access and avoiding race-conditions. However, it comes at the penalty of poorer mixing of labels.

Asynchronous Parallelisation This strategy is simply to ignore the race conditions incurred by a direct parallelisation of the serial algorithm, by using a single shared data structure from which to read label values and write label updates. In this case, when updating $\zeta_i(v)$, the label value read for a neighbour w may correspond to $\zeta_i(w)$ or $\zeta_{i-1}(w)$, depending on whether the thread processing w has written

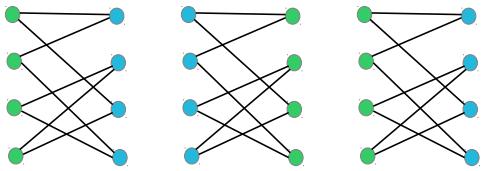

Figure 1: Illustration of label oscillation problem on a bipartite graph, showing the node labels (represented by colours) in iteration i, $i+1$ and $i+2$

its label before that processing v reads it. The update ordering of vertices becomes dependent on the architecture of the machine and moreover, may not correspond to *any* sequential ordering of vertex updates, since race conditions can lead to a situation where two neighbouring vertices *both* use their old values in their update step.

Semi-synchronous Parallelisation The key to this strategy [9] is the use of a *proper graph colouring* i.e. a partitioning of the vertex set V into a set of independent subsets $\{C_1, \dots C_k\}$ such that $C_i \subset V$, $C_i \cap C_j = \emptyset$, $1 \leq i < j \leq k$ and $\bigcup_i C_i = V$; where, $\forall i$, no two vertices in C_i share an edge. Vertex updates occur in order of colour, i.e. the vertices in C_1 are updated before those in C_2 and so on, leading to a parallel algorithm which we call PSILU, of the form shown in Algorithm 2.

Algorithm 2 Parallel Semi-synchronous Iterative Local Update (PSILU)

Input: $G(V, E)$, $\alpha \geq 0$, $\{C_1, \dots, C_k\}$
1: Initialize labels: $\forall (v \in V), \zeta_0(v) \leftarrow l(v)$
2: $i \leftarrow 0$, updated $\leftarrow n$
3: **while** updated $> \alpha$ **do**
4: $i \leftarrow i+1$, updated $\leftarrow 0$
5: *Initialize shared quantities*
6: **for** $\forall C_j$, $j = 1, \dots, k$ **do**
7: **ParallelFor** $\forall (v \in C_j)$
8: $\zeta_i(v) = f\left(\zeta_i(N^p(v)), \zeta_{i-1}(N^s(v))\right)$ ($f(.)$ is the local update function)
9: **if** $\zeta_i(v) \neq \zeta_{i-1}(v)$ **then**
10: updated $\leftarrow$ updated $+ 1$
11: **end if**
12: **EndParallelFor**
13: **Parallel Reduce** *shared quantities*
14: **end for**
15: **end while**
Output: $\forall \zeta_i(v), v \in V$

2.2 Limitations of Synchronisation Levels

Synchronous Parallelisation The main problem of synchronous parallelisation is the poor mixing of labels. An extreme, but common, consequence of this poor mixing is that there may be a cyclic oscillation of the labels of some vertices (already observed in [15]). For instance, Figure 1 shows a bi-partite graph and the node labels represented by colours. In this scenario, the labels of the vertices in the two parts of the graph oscillate from one iteration to another. This problem prevents the convergence to a stable community structure. Current parallel implementations side-step this issue either by terminating when the number of nodes to be updated is below a certain arbitrary threshold or by doing a certain degree of asynchronous updating or both (e.g., [17]).

Asynchronous Parallelisation While updating vertices in a particular serial order is beneficial for mixing of the labels, race conditions in the asynchronous approach make it impossible to control update ordering and thus preclude an analysis of the performance in terms of this ordering. The performance of the algorithm is highly dependent on the low-level details of the architecture. In the shared-memory context, whether an old or new label value is read at the update stage depends on the timing of when the new label value is flushed to memory. If many such updates occur before the neighbourhood values have flushed, it can effectively lead to a synchronous update. Such a behaviour can push the number of iterations required to converge higher than would be expected. Moreover, when new label values do flush to memory, the writing of such shared label values that must subsequently be read by another thread, destroys cache coherency, requiring reads from memory rather than cache.

Semi-synchronous Parallelisation Liu and Murata [9] show that in the semi-synchronous variation of LP, the label oscillation problem disappears in a range of bipartite graphs. This is later extended by Cordasco and Gargano [3] to general graphs. The latter result formally proves that this variant is guaranteed to converge. It also shows that the communities identified by this variant have roughly the same modularity score as the asynchronous approach on a range of graphs. Furthermore, the results of the semi-synchronous variant were shown to be more stable than the asynchronous variant – the standard deviation of the modularity obtained for different test settings was smaller.

Due to the independence property of each colour set, the order in which the vertices are processed within a colour does not affect the outcome of the algorithm. Moreover, as only the vertices within a single colour set are processed in parallel, label values of neighbours are never updated during the parallel loop, removing race conditions from the outcome of the algorithm and avoiding the cache coherency issues of the fully asynchronous approach. Finally, the algorithm still allows for significant label mixing during a single pass of the algorithm through the vertex set, thus reducing the number of iterations required by a synchronous algorithm, while avoiding its convergence issues.

The considerable gains of the semi-synchronous parallelisation in terms of convergence and consistency of results are attained at the cost of additional synchronisation barriers. For this reason, it was assumed to be too slow to be of much practical merit and therefore, was not considered in earlier parallel implementations of ILU in general and label-propagation in particular. We analyse the effects of colouring algorithms and vertex updated orderings on the performance of the semi-synchronous approach and use it to engineer an implementation based on a careful design choices. In Section 7, we show the experimental results analysing the performance of our engineered semi-synchronous implementation.

3. SEMI-SYNCHRONOUS ILU FOR MODULARITY MAXIMISATION AND MAP EQUATION

In this section, we present our extension of the semi-synchronous ILU for modularity maximisation (MM). The key issue in this extension is that MM requires more states

to be maintained in order to compute the change in modularity that drives the vertex update function. In particular, while modularity depends on the adjacency matrix of the given graph, it also depends on the *volume* of each community, where $\text{Vol}_c = \sum_{i \in c} k_i$. The change in modularity when moving a node v from its current community c to the community of a neighbouring node, d is:

$$\Delta m_{vcd} = \frac{2}{M}(k_{v,d} - k_{v,c}) - \frac{2}{M^2} k_v (\text{Vol}_d - \text{Vol}_c + k_v)$$

where $k_{v,d}$ is the sum of the edge weights from v to d and $M = \sum_i k_i$. The second component of this expression is non-local. Its existence implies that the order of vertex updates even over independent vertex sets impacts on the outcome of the algorithm and moving two vertices simultaneously on different threads may not be equivalent to any sequential ordering. In fact, when vertices are moved simultaneously by different threads, the overall change in modularity can be expressed as $\Delta m = \sum_{t=1}^{T} \Delta m_t - I$, where T is the number of threads, Δm_t is the change in modularity due to moving vertices assigned to thread t and I is a term that expresses the interaction between vertices moved by different threads, which is unknown until threads synchronise after a set of local updates.

The semi-synchronous strategy of restricting parallel updates to independent sets restricts I to

$$I = \frac{1}{M^2} \sum_c \left(\Delta \text{Vol}_c^2 - \sum_{t=1}^{T} (\Delta \text{Vol}_c^t)^2 \right) \quad (1)$$

where ΔVol_c^t is the overall change in the volume of community c due to the movement of vertices by thread t and ΔVol_c is the overall change in the volume of c. The interaction term is small in comparison to that which would apply if non-independent sets of vertices were moved and, if desired, can be easily corrected at each synchronisation point. Vol_c (and I) may be updated by reduction after a set of local updates (line 13 of Algorithm 2). Since in the ILU approach we initialise with a community per node, rather than reduce over such a large number of local values, it is better to recompute the volume using the new community labels.

Similarly, the objective function for the map equation [16] depends on two sets of quantities, namely p_c, the probability that a random walk visits community c and q_c, the exit probability from community c, where these probabilities are based on a random walk over the graph. When an independent set of vertices is updated in parallel, the use of independent sets reduces the error in state during parallel update and the overall change to these shared quantities after the update is simply the sum of the change as calculated independently on each thread.

3.1 Algorithm Optimisations

As the LP and MM algorithm approach convergence, the last number of sweeps through the vertices result in just a few vertices changing their label values. This phase of the algorithm can be needlessly expensive. A key optimisation used in [17], which we adopt in both our LP and MM algorithms, is the maintenance of an `active` flag for each vertex. Only active vertices are considered for update. Whenever a vertex leaves its label unchanged, its active flag is set to `false`; whenever a vertex changes its label, it sets the active flags of its neighbours to `true`, so that they are considered for update in the next pass. Even so, when the stopping criterion is based on the fraction of vertices that move in each sweep, the MM algorithm can perform a large number of update steps as it approaches convergence, resulting in only a very small change in the overall modularity. Hence, we stop the algorithm when the perceived change in modularity (i.e. the estimated Δm, excluding the interaction term I) after an update sweep is less than a given threshold ($= 10^{-4}$ in our experiments).

4. DATASETS AND ARCHITECTURES

Graphs considered Table 1 shows the graphs that we consider in our experimental evaluation. Pokec, Orkut and Livejournal [1] are online social networks and exhibit the complex structure of such networks, such as high clustering and heavy-tailed degree distribution. UK 2002[2] is obtained from a crawl of the `.uk` domain. We choose this network to contrast with the social networks.

Architectures considered To evaluate the performance of the shared memory implementation, we use the following architecture:

- (Config A) A system with 2 x 10-core Intel Xeon E5-2470 v2 2.4GHz processors, each with a shared 25MB L3 cache and 256KB L2 cache per core and a total of 128GB RAM.
- (Config B) A system with 2 x 16-core 2.6 GHz AMD Opteron 6282 SE processors and a total of 96GB RAM. Each core has its private 2MB L2 cache while the 16 MB L3 cache is shared between 16 cores.
- (Config C) A system with 4-core ($\times$ 2 with hyper-threading) 3.6GHz Intel(R) Core i7-3820 processor with a shared 10MB L3 cache and 64GB RAM.
- (Config D) A system with 6-cores ($\times$ 4) 2.4GHz Intel(R) Xeon E7450 processor with a shared 12 MB L3 cache and 128GB RAM.

For the sake of brevity, we only highlight the key findings on Config A and note that the results on Config B, C and D are along similar lines.

Software designed We implemented our ILU LP and MM code, using C++ and OpenMP. The OS has a Linux Ubuntu (12.04 release) installation. We used the option -O3 with the gnu C++ compiler version 4.6.

5. EFFECT OF VERTEX VISIT ORDER ON ASYNCHRONOUS ILU

The performance of the LP and MM ILU algorithms depends intimately on how fast the labels mix, which in turn is dependent on the order in which vertices are updated. Social network graphs generally consist of a core of inter-connected high degree nodes, along with low-degree 'whiskers' outside the core. The node degrees are highly non-uniform, following a heavy-tailed scale-free distribution. For such graphs, in particular, we can expect highly different behaviour of the LP algorithm depending on the ordering in which nodes are traversed. Intuitively, ordering the nodes in decreasing degree so that high-degree nodes are processed first should

[1] downloaded from `http://snap.stanford.edu`
[2] downloaded from `http://law.di.unimi.it`

Table 1: Networks used

Network	# nodes	# edges	Avg. clustering coefficient	Diameter	Effective Diameter
UK 2002	18,520,486	298,113,762	-	218	19.8
Orkut	3,072,441	117,185,083	0.16	9	4.8
Live Journal	4,847,571	42,851,237	0.2742	16	6.5
Pokec	1,632,803	22,301,964	0.1094	11	5.2

allow the labels to mix quickly from the core to the periphery of the network. Alternatively, choosing a breadth-first search (BFS) ordering from a chosen starting node allows the labelling to mix outward from that node. Such orderings can lead to fast convergence of the LP algorithm. We use the Pokec social network to study this mixing issue.

5.1 LP on Pokec

We introduce a work-load measure, W, to quantify the work carried out during the full execution of the algorithm. We define,

$$W = \sum_{i=1}^{\#\text{sweeps}} \sum_{v \in V^i_{\text{active}}} k_v;$$

the sum over all sweeps through the vertices of the degrees of the vertices, V^i_{active}, that are active during sweep i. This is a reasonable approximation of the computation that is carried out in these ILU algorithms. For each active vertex v, the labels of its k_v neighbours are considered for a potential move to the corresponding community, an $O(k_v)$ operation, considering that selecting the best move involves a maximisation of k_v terms. We ignore the constant overhead of processing inactive vertices and the fact that active vertices that change their label must carry out a further $O(k_v)$ operation to set the active flag of their neighbours. In practise, with all other conditions fixed, we find empirically that runtime and W follow an approximately linear relationship and W gives a measure of how much useful computation is carried out, excluding the overhead.

In Figures 2 and 3 we show the modularities and workload that result from running the LP algorithm on the Pokec graph, under a number of different node traversal orderings, as box-plots based on 100 trial runs. Six orderings are chosen,

1. Random ordering;
2. Increasing degree ordering;
3. Decreasing degree ordering;
4. BFS starting from the node of maximum degree and ordering the nodes in each level set in order of decreasing degree;
5. BFS starting from the node of minimum degree and ordering the nodes in each level set in order of increasing degree;
6. BFS starting with a random node and ordering the nodes in each level set randomly.

The wide variation in results is striking. Choosing the nodes from high to low degree results in fast mixing and hence low workload, but also results in low modularity, because the labels mix into a giant community. A random ordering gives a wide variation of workloads and modularities.

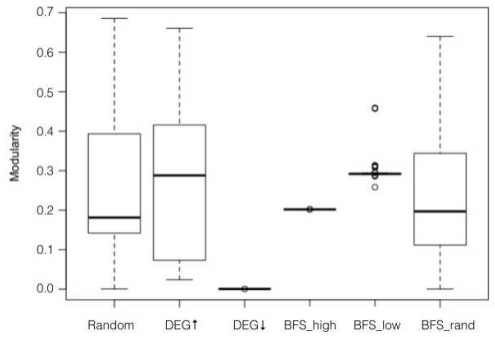

Figure 2: Modularity of LP communities for different traversal Orderings on the Pokec network.

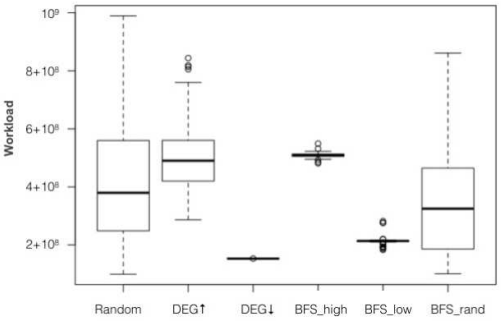

Figure 3: Workload of LP algorithm for different traversal Orderings on the Pokec network.

BFS starting from a random node results in lower workload on average than a fully random ordering and a similar range of modularities. On the other hand BFS starting from low degree node results in low workload and high modularity with practically no variance. It would appear to be the best compromise in terms of workload and quality.

5.2 MM on Pokec

The MM algorithm is less prone to getting trapped in undesirable local maxima than LP. We find that, running on the above orderings, the algorithm converges to solutions that vary in modularity in a range between 0.658 and 0.676. Nevertheless, the work required to reach this solution varies according to update order. The decreasing degree order requires $W = 2.058 \times 10^8$, while the increasing degree order requires more than twice as much work ($W = 5.369 \times 10^8$). Random ordering requires $W = 3.784 \times 10^8$ on average while

a BFS from a random node requires slightly less work on average ($W = 3.682 \times 10^8$) and from the highest degree node, less again ($W = 3.481 \times 10^8$)

Similar behaviour is seen in other social network graphs of highly non-uniform degree distribution such as the Orkut social network graph. It follows that an analyst may prefer the decreasing degree order to speed convergence.

5.3 Asynchronous Ordered Computations

On a shared-memory multicore, a simple approach to approximating a desired update order is to apply an asynchronous update by parallelising the sequential loop that iterates through the vertices in the desired order using a `parallel for` operation, on the assumption that race conditions will have only a minor impact on the outcome. However, the impact is quite dramatic. On the Orkut network, the workload experienced by the asynchronous version of the decreasing degree ordering on 4 threads is on average 3.7 times greater than on a single thread and this grows to 4.6 times when 16 threads are used. Moreover, it is clear that a static scheduling of the loop would assign all the high-degree (and therefore work-intensive) vertices to the same thread, leading to high load imbalance. Our empirical evaluation shows that even an adaptive dynamic schedule (using the OpenMP `guided` scheduler) still leads to high load imbalance. The result is that a speed-up of only 1.17 is achieved on eight threads.

It transpires that much better control of the work-load can be achieved by adopting the semi-synchronous approach. Nevertheless, this also introduces extra overhead, through the synchronisation steps – this trade-off will be explored later in the paper.

6. COLOURING ALGORITHMS

The performance of the semi-synchronous algorithm is highly dependent on the choice of colouring algorithm. A commonly used heuristic is the *greedy* heuristic [2] summarised in Algorithm 3, which depends, firstly, on the ordering in which the vertices are visited and, secondly, on how a colour is chosen from amongst those not forbidden. Hasenplaugh et. al. provide a comprehensive overview of sequential an parallel graph colouring algorithms using a number of different node ordering heuristics [6]. In many applications, an optimal colouring is considered to be one in which the total number of colours is minimised. In this case, a good choice for the choose(.) function is min(.) and vertices may be traversed in order of degree. Using the minimum function can lead to a highly unbalanced colouring, in which a few colour sets contain many more vertices than the others. For the PSILU algorithms considered here, each colour set is separately parallelised by distributing its vertices among the available threads. Thus, a highly unbalanced colouring does not necessarily mean an unbalanced workload, as the work carried out by each thread is proportional to the sum of the degrees of the vertices allocated to the thread in a complete pass through the vertex set. In fact, if `static` scheduling is used to distribute the vertices within each colour, then the ordering of the vertices *within* each colour has a larger impact on the load balance. In particular, if vertices were ordered by degree within each colour set, then some threads would get a disproportional amount of work. A `guided` schedule over a randomly ordered colour set is found to be the most effective in ensuring small load imbalance.

A further consideration is the total number of iterations required for the algorithm to converge and the total workload, W. Iterating over an independent set slows mixing as, by construction, no vertex in the colour set shares a neighbour with any vertex. In particular, if the first colour set traversed is very large (as may be typical of an algorithm that seeks to minimise the total number of colours), then all the vertices within this set are effectively updating in synchronous mode, as their updates all depend on the labels of neighbours from the previous pass. This suggests that to speed mixing, the colour sets should not be too large, while if they are too small, the speed of a single pass through the algorithm will be affected by a large number of synchronisations after each colour is processed. Thus, an effective heuristic for colouring for ILU community-finding algorithms is

1. To minimise the number of iterations, choose a *balanced* colouring algorithm, in which the work per colour is as balanced as possible across all colours;

2. To minimise the load imbalance per iteration, choose an vertex ordering within each colour set, so that the work per thread is as balanced as possible and use a dynamic scheduling of work.

Given a particular vertex visit preference order, we can also ask if a colouring strategy can be found that adheres to that preference as closely as possible. For instance, if decreasing degree update order is preferred in the ILU algorithm, then colouring the vertices in order of decreasing degree, ensures that high-degree vertices in general get low colour values and hence are processed early in the ILU update sweep. Thus a simple strategy is to colour the vertices in the order in which we would like them to be processed by the ILU algorithm.

As for choosing the non-forbidden colour, we distinguish between three variants:

- **Random Greedy Heuristic** (RG): choose(.) = a randomly chosen non-forbidden colour.

- **First Greedy Heuristic** (FG): choose(.) = min(.) – This variant aims at minimising the total number of colours. However, this can lead to a highly unbalanced colouring.

- **Smallest Greedy Heuristic** (SG): choose(.) = the non-forbidden colour, such that the number of nodes associated with the colour is minimised. This leads to highly balanced colours, though the number of colours can be larger than First greedy approach.

6.1 Analysis of Coloured Orderings on Pokec

For each of the RG, FG and SG colouring strategies, we form the colour sets using three different node traversal orderings – namely a BFS ordering starting from a random node and decreasing and increasing degree ordering. We examine the modularity and workloads that are obtained using the LP algorithm and the different resulting vertex orderings in Figures 4 and 5. For SG, the BFS traversal ordering leads to a similar set of modularity values and workloads as using

Algorithm 3 Sequential Greedy Colouring

Input: $G(V, E)$
1: $\forall v \in V$, $c(v) \leftarrow 0$, $i \leftarrow 1$
2: $\pi \leftarrow$ ordering of the vertices
3: **while** $i \neq n$ **do**
4: $v \leftarrow \pi(i)$, $i \leftarrow i + 1$, $F \leftarrow \emptyset$ (F holds the forbidden colours for v)
5: **for** each $n \in N(v)$ **do**
6: $F \leftarrow F \cup c(n)$
7: **end for**
8: $c_m = \text{choose}\{j > 0 : j \notin F\}$, $c(v) = c_m$,
9: **end while**
Output: $\forall v \in V\, c(v)$

the asynchronous LP with BFS ordering starting from random source (as presented in figures 2 and 3), though with greater variance. The decreasing degree visit ordering leads to small workload, but also low modularities. These low modularity values are avoided while low work-load is maintained when MM is used. It would appear that FG with decreasing degree node ordering would be the best choice as it results in the highest modularity value on average. However this approach also results in the highest workload. On the other hand RG with increasing degree node ordering seems to be the best compromise between high average modularity and low average workload.

It is important to note that the workload is independent of the number of threads and hence the coloured orderings have the potential to achieve greater speed-up in semi-synchronous mode, than that achieved by the asynchronous method.

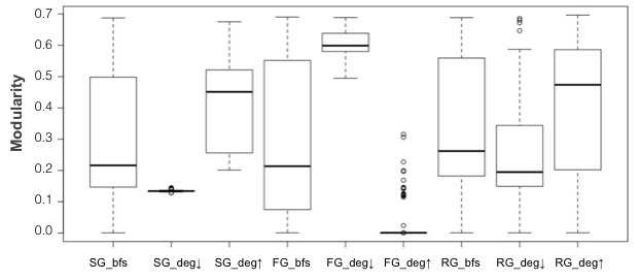

Figure 4: Modularity and workload for different colour orderings on the Pokec network.

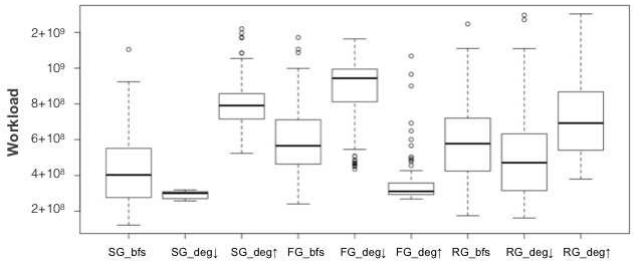

Figure 5: Modularity and workload for different colour orderings on the Pokec network.

6.2 Blocked Colouring Strategy

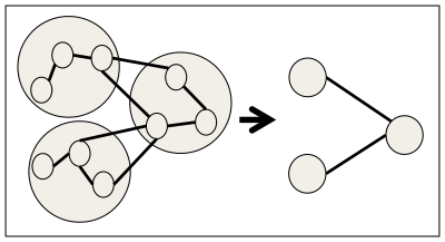

Figure 6: Merging vertices into larger units of work

Another strategy to mitigate against slow mixing when updating on independent sets of vertices is to carry out a blocked colouring strategy. In the standard ILU, the tasks that are allocated to each thread consist of a single vertex update. A generalisation is to combine multiple vertices into a single task. See Figure 6 in which a block graph is formed by merging groups of three vertices. A colouring of the block graph allows non-conflicting tasks to be updated in parallel. As each block of vertices is updated on a single thread, vertex updates within a block are immediately available to their neighbours in the same block – thus, label mixing can begin even in the first colour update. One drawback of this approach is that task unbalance can be exacerbated when a number of high degree vertices are allocated to the same block. Also, depending on the network, the block graph may be significantly more dense than the original graph, hence requiring more colours. On the other hand, when the network exhibits hierarchical structure, blocking can be an effective way to ensure fast mixing in the lower levels of the hierarchy. In our implementation, given a block size, we visit vertices in a given order and form blocks by merging them with neighbouring vertices in a breadth-first manner.

7. EVALUATION

In this section, we report our analysis on the performance of our semi-synchronous implementation and the design choices involved in its engineering. In the LP algorithm, we use random tie breaking and whenever there is a random element to the algorithm, e.g. when random colouring is applied, we report average run-times, speed-up over multiple runs. Note that we have carefully benchmarked our code with that presented in [17] and have confirmed that our asynchronous implementation on a single thread produces identical output with that code.

Our evaluation attempts to answer the following questions:

- How is the work-load W affected by different vertex traversal orderings?
- How does the overhead of synchronisation affect the performance of semi-synchronous algorithms?
- To what extent does load imbalance affect the speed-up of the algorithms?

While the total work-load W provides a measure of the total amount of computation that is carried out by the algorithms, run-time is also affected by overhead. In the asynchronous method, threads synchronise after each pass

Table 2: LP on 1 and 8 Processors : RG and Asynchronous with Decreasing Degree Ordering.

Network	Method	# Cols	# Sweeps	$W_{\times 10^9}$	t_{col}	t_1	t_8	S	m_1	m_8
Orkut	Async Deg ↓	1	4.4/28.6	0.46/1.37	$0s$	$24.4s$	$37.2s$	0.65	~ 0	0.07
Orkut	RG Deg ↓	40.6	25.8	1.00	$8.65s$	$174.4s$	$35.5s$	4.90	0.53	0.55
Pokec	Async Deg ↓	1	15.4/16.3	0.15/0.13	$0s$	$7.87s$	$3.48s$	2.25	0.0003	0.01
Pokec	RG Deg ↓	39	52.25	0.45	$1.33s$	$33.8s$	$7.55s$	4.47	0.17	0.30
UK 2002	Async Deg ↓	1	30.2/30.2	1.15/1.41	$0s$	$51.4s$	$41.5s$	1.23	0.97	0.95
UK 2002	RG Deg ↓	944	30	1.23	$4.5s$	$120.7s$	$29.7s$	4.06	0.96	0.96

through the vertices; the semi-synchronous methods have more synchronisation points by a factor of the number of colours. Hence the number of colours and the number of sweeps through the vertices give an indication of the amount of overhead. Semi-synchronous methods also have the overhead of computing the colouring.

7.1 LP Algorithm

We summarise results obtained for 1 and 8 threads on the two social network graphs and on the UK 2002 network in Tables 2 and 3, contrasting the decreasing degree ordering with increasing degree ordering. In the case of the asynchronous method, the two values given for # sweeps and W correspond to the averages obtained on 1 and 8 threads; t_1, t_8 are the total run-times; t_{col} is the time taken to colour the graph; S is the speedup on 8 threads with respect to the same algorithm run on a single thread; m_1, m_8 are the modularity values obtained on the corresponding number of threads.

Note the larger speed-up obtained for the semi-synchronous algorithms, compared with the asynchronous approach. Except for the Pokec network, the run-time obtained on 8 processors is smaller with semi-synchronous method, despite the greater number of synchronisation points and colouring overhead. Moreover, the RG coloured ordering avoids the giant community local minimum that LP falls into when run in a strictly decreasing degree ordering on the social network graphs. As pointed out in Section 5.1, the results on the social network graphs are highly volatile. On the other hand, the UK 2002 graph shows much more stability, with modularity values generally robust to the ordering of the algorithm. When increasing degree order is used, generally there is significantly more work done in order for LP to converge, which leads to large run-times. Colouring in increasing degree order tends to result in more colours and hence more synchronisation points for the semi- synchronous algorithm. Furthermore, the work is much more balanced by the dynamic scheduler in the asynchronous method, resulting in an increase in speed-up. The overhead of the semi-synchronous method now means that it lags behind the asynchronous method in run-time.

7.2 MM Algorithm

In terms of work-load and scalability, we can reach many of the same conclusions with the MM algorithm as are reached with the LP algorithm. The convergence criterion is important in this case. Generally, if the MM algorithm was allowed to complete to full convergence (i.e. until no more vertex updates occur in a sweep of the algorithm), then all orderings would reach similar modularity scores, but the decreasing degree ordering does so much more quickly. For instance, on the Orkut network, decreasing degree ordering reaches a modularity score of 0.66 with a work-load of 2.11×10^9 ($160s$ on a single thread) while increasing degree ordering reaches a slightly smaller modularity of 0.63, but with a work-load of 3.37×10^9 ($282s$). On the other hand, if we apply a stopping condition based on Δm, then decreasing degree ordering still succeeds in reaching the high modularity score, while the other orderings exit with comparably lower modularity scores (see Figure 8). If synchronisation is applied after each colour update to keep Δm exact, then semi-synchronous cannot compete with asynchronous in terms of run-time. However, with synchronisation applied only at the end of a full vertex sweep, we can see the advantage of a coloured ordered ordering. The run-time of the asynchronous and RG decreasing degree strategies is shown in Figure 7. Again, asynchronous decreasing degree ordering fails to achieve speed-up on multiple threads due to the load imbalance and varying work-loads that it incurs when running on multiple threads. The peak at 8 threads, is due to the fact that, on our architecture, race conditions lead to the method requiring more than twice as much work as on a single thread. In contrast, the semi-synchronous strategy maintains a relatively consistent work-load, which is larger than that of pure decreasing degree ordering on a single thread, but yields better performance after 4 threads.

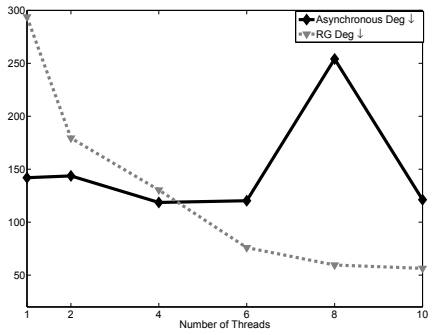

Figure 7: Asynchronous Runtime v RG Runtime on the Orkut Network

7.3 Blocked Colouring

We explore the effect of blocking with two of the networks, Orkut and UK 2002. Following a decreasing degree order, we merge neighbouring vertices in a breadth-first manner to form blocks. Using the RG colouring strategy in decreasing degree order, we run a single thread LP algorithm on the Orkut and UK 2002 graphs for block sizes in the range 1 to 128. The results are shown in Figure 9. Grouping just two vertices together results in more thaw a halving of the run-time after which further increases in block-size give just

Table 3: LP on 1 and 8 Processors : RG and Asynchronous with Increasing Degree Ordering.

Network	Method	# Cols	# Sweeps	$W_{\times 10^9}$	t_{col}	t_1	t_8	S	m_1	m_8
Orkut	Async Deg ↑	1	32.4/57.6	1.60/1.76	$0s$	$88.4s$	$23.1s$	3.81	0.51	0.54
Orkut	RG Deg ↑	304.6	56	2.8	$7.4s$	$195.0s$	$41.0s$	4.75	0.52	0.52
Pokec	Async Deg ↑	1	76/80	0.5/0.58	$0s$	$33.18s$	$5.94s$	5.58	0.24	0.39
Pokec	RG Deg ↑	90.75	86	0.57	$1.14s$	$48.5s$	$8.5s$	5.71	0.48	0.55
UK 2002	Async Deg ↑	1	40/38	1.54/1.551	$0s$	$70.8s$	$20.0s$	3.54	0.92	0.95
UK 2002	RG Deg ↑	944	44.4	1.57	$6.1s$	$145.5s$	$40.89s$	4.19	0.92	0.92

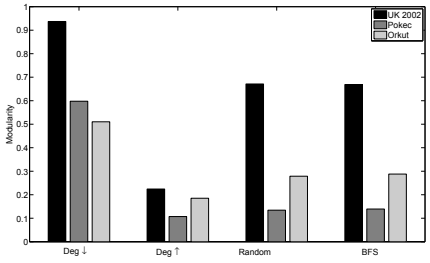

Figure 8: Modularities Achieved with RG colouring and different orderings.

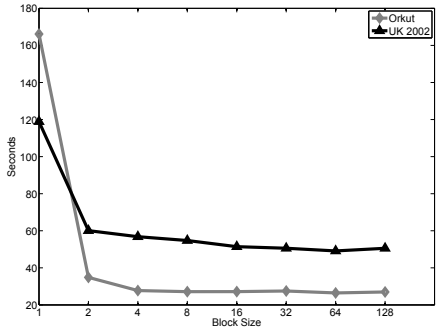

Figure 9: Comparison of Runtime of LP against Block Size, Single Thread, RG with Deg ↓ ordering

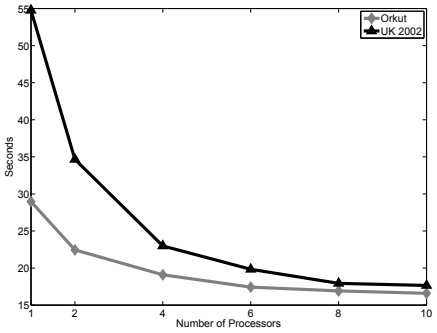

Figure 10: Comparison of Runtime of LP against # Threads, RG with Deg ↓ ordering, Block Size=4 (Orkut), =16(UK 2002)

small gains in run-time. Based on these results, we choose a block size of 4 for the Orkut network and 16 for the UK 2002 network and run the algorithm on multiple threads, in Figure 10. The strong interconnectedness of vertices at the core of social networks, means that blocking tends to group high-degree nodes together which are very highly connected within the rest of the core. This results in a blocked graph that is more dense than the original graph. As a result, more colours are required to colour the graph. On Orkut, while on average 117 colours are required for the original graph, 545 are required when vertices are merged into blocks of size 4. On the other hand, UK 2002 web graph has a more hierarchical structure which blocking helps to reveal, resulting in a blocked graph which can be coloured with 92 colours on average with blocks of size 16, compared with the 944 required on the original graph. The resulting fewer synchronisation points, along with the fact that labels can immediately mix between connected vertices within blocks gives the blocked strategy an advantage over the non-blocked method. On the other hand, scalability is hampered by the larger work units, which can lead to more imbalance. While both networks maintain their performance improvement over a block size of 1, the UK 2002 network maintains a better speed-up (3.05 on 8 processors) compared with Orkut (only 1.71 on 8 processors).

8. CONCLUSION

In the past, the parallel and distributed ILU implementations did not consider using a semi-synchronous approach, as it was considered to be too slow to be practically viable. In this paper, we have shown that on shared-memory platforms, this algorithm can be engineered to be as fast (and sometimes even faster than) the asynchronous approach, without the race conditions and the related cache-coherency issues associated with the latter. A major advantage of our engineered approach is that it gives the control of node update ordering back to the algorithm developer. The number of iterations and the quality of the resulting clusters produced by our approach is invariant to architecture and systems layer issues.

We find that node ordering and graph colouring techniques have a significant impact on the running time of the semi-synchronous approach and the quality of the resulting communities. Thus, to achieve good performance, a user needs to carefully select the node ordering and graph colouring technique based on the structure of the graph being considered (if known in advance) and the overall application objective.

For the node ordering, our results show that decreasing node ordering results in fast label mixing but it often results in one giant community yielding low modularity values. On the other hand, increasing degree node ordering produces a large variation in results, requiring multiple runs of the algorithm in order to maximise the modularity value. For social

network graphs, the BFS with increasing degree node ordering seems to be the best choice, as it reduces the variation both in modularity and workload.

The modularity of the communities identified by the MM algorithm is more robust to node ordering than the LP algorithm. However, similar to LP, the workload for MM algorithm varies a lot with the node update ordering. Thus, a decreasing degree ordering is preferable for the MM algorithm, to speed the convergence.

Using the right colouring strategy one can preserve the preferred node ordering while the LP algorithm is run in parallel on a number of threads. That is the semi-synchronous LP is completely deterministic and therefore it returns the control over the node ordering back to the developer. Our findings suggest that even simple and highly parallelisable colouring strategies can achieve the desired node ordering. For instance the SG colouring with BFS node ordering strategy results in similar modularity and workload values as pure BFS node ordering.

Our blocked colouring strategy can speed the asynchronous LP considerably if used on networks with hierarchical structure.

Our engineering of the semi-synchronous approach is based on a careful selection of node ordering and graph colouring strategy in order to maximise the run time performance and the quality of the resulting communities.

Most scalable graph processing systems proposed in recent years (e.g., Pregel [11], GraphLab [10], GraphChi [8]), are based on a vertex-centric model of computation. In this model, each node and edge is associated with some labels and the computation consists of iteratively updating these labels. ILU is a natural framework for community detection in this computation model. The insights learnt from our work can be particularly useful in engineering ILU for these systems. In fact, our results can be more generally useful for designing and engineering many other algorithms in the vertex-centric computation model, where the mixing rate of labels is an important concern.

Acknowledgements

The Insight Centre for Data Analytics is supported by Science Foundation Ireland under Grant Number SFI/12/RC/2289.

9. REFERENCES

[1] Blondel, V.D., Guillaume, J.L., Lambiotte, R., Lefebvre, E.: Fast unfolding of communities in large networks. Journal of Statistical Mechanics: Theory and Experiment 2008(10), P10008 (2008), http://stacks.iop.org/1742-5468/2008/i=10/a=P10008

[2] Çatalyürek, Ü.V., Feo, J., Gebremedhin, A.H., Halappanavar, M., Pothen, A.: Graph coloring algorithms for multi-core and massively multithreaded architectures. Parallel Computing (2012)

[3] Cordasco, G., Gargano, L.: Label propagation algorithm: a semi-synchronous approach. International Journal of Social Network Mining (IJSNM) 1(1), 3–26 (2012)

[4] Fortunato, S.: Community detection in graphs. CoRR abs/0906.0612 (2009)

[5] Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: Powergraph: Distributed graph-parallel computation on natural graphs. In: Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI) (2012)

[6] Hasenplaugh, W., Kaler, T., Schardl, T.B., Leiserson, C.E.: Ordering heuristics for parallel graph coloring. In: Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures. pp. 166–177. ACM (2014)

[7] Kaler, T., Hasenplaugh, W., Schardl, T.B., Leiserson, C.E.: Executing dynamic data-graph computations deterministically using chromatic scheduling. In: Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures. pp. 154–165. ACM (2014)

[8] Kyrola, A., Blelloch, G., Guestrin, C.: Graphchi: Large-scale graph computation on just a pc. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12). pp. 31–46. USENIX Association (2012)

[9] Liu, X., Murata, T.: How does label propagation algorithm work in bipartite networks? In: Proceedings of the 2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology. pp. 5 – 8. IEEE (2009)

[10] Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.M.: Graphlab: A new framework for parallel machine learning. In: Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence (UAI). pp. 340–349 (2010)

[11] Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: A system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD'10). pp. 135–146. ACM (2010)

[12] McDaid, A., Hurley, N.: Detecting highly overlapping communities with model-based overlapping seed expansion. In: Advances in Social Networks Analysis and Mining (ASONAM), 2010 International Conference on. pp. 112–119. IEEE (2010)

[13] Newman, M.E.: Modularity and community structure in networks. Proc Natl Acad Sci U S A 103(23), 8577–8582 (2006)

[14] Ovelgönne, M., Geyer-Schulz, A.: An ensemble learning strategy for graph clustering. In: Graph Partitioning and Graph Clustering. Contemporary Mathematics, vol. 588, pp. 187–206. American Mathematical Society (2012)

[15] Raghavan, U.N., Albert, R., Kumara, S.: Near linear time algorithm to detect community structures in large-scale networks. Physical Review E 76(036106), 1 – 12 (2007)

[16] Rosvall, M., Bergstrom, C.T.: Maps of random walks on complex networks reveal community structure. Proceedings of the National Academy of Sciences 105(4), 1118–1123 (2008)

[17] Staudt, C., Meyerhenke, H.: Engineering high-performance community detection heuristics for massive graphs. In: Proceedings of the 42nd International Conference on Parallel Processing (ICPP). pp. 180–189. IEEE (2013)

[18] Ugander, J., Backstrom, L.: Balanced label propagation for partitioning massive graphs. In: Proceedings of the Sixth ACM International Conference on Web Search and Data Mining (WSDM). pp. 507–516. ACM (2013)