iPregel: Strategies to Deal with an Extreme Form of Irregularity in Vertex-Centric Graph Processing

iPregel: Strategies to Deal with an Extreme Form of Irregularity in Vertex-Centric Graph Processing

Ludovic Anthony Richard Capelli
School of Informatics
The University of Edinburgh
Edinburgh, United Kingdom
l.capelli@ed.ac.uk

Nick Brown
Edinburgh Parallel Computing Centre
The University of Edinburgh
Edinburgh, United Kingdom
n.brown@epcc.ed.ac.uk

Jonathan Mark Bull
Edinburgh Parallel Computing Centre
The University of Edinburgh
Edinburgh, United Kingdom
m.bull@epcc.ed.ac.uk

***Abstract*—Over the last decade, the vertex-centric programming model has attracted significant attention in the world of graph processing, resulting in the emergence of a number of vertex-centric frameworks. Its simple programming interface, where computation is expressed from a vertex point of view, offers both ease of programming to the user and inherent parallelism for the underlying framework to leverage. However, vertex-centric programs represent an extreme form of irregularity, both inter and intra core. This is because they exhibit a variety of challenges from a workload that may greatly vary across supersteps, through fine-grain synchronisations, to memory accesses that are unpredictable both in terms of quantity and location.**

In this paper, we explore three optimisations which address these irregular challenges; a hybrid combiner carefully coupling lock-free and lock-based combinations, the partial externalisation of vertex structures to improve locality and the shift to an edge-centric representation of the workload. We also assess the suitability of more traditional optimisations such as dynamic load-balancing and software prefetching. The optimisations were integrated into the iPregel vertex-centric framework, enabling the evaluation of each optimisation in the context of graph processing across three general purpose benchmarks common in the vertex-centric community, each run on four publicly available graphs covering all orders of magnitude from a million to a billion edges.

The result of this work is a set of techniques which we believe not only provide a significant performance improvement in vertex-centric graph processing, but are also applicable more generally to irregular applications.

***Index Terms*—vertex-centric, hybrid combiner, structure externalisation, edge-centric workload, load-balancing, cache efficiency**

I. Introduction

Graphs have become an ubiquitous data structure due to their application throughout very many areas of technology. As such the development of graph processing algorithms is an important and active area of research, with growing interest in the development of programming models for this domain in order to support the expression of complicated algorithms. It follows that there is a growing interest in programming models for graph processing.

In 2010, Google introduced the vertex-centric programming model through Pregel [1], which has attracted great attention, where the user expresses the graph computation from a vertex perspective. In this model, verticies can communicate with each other by sending messages along their outgoing edges and each have a mailbox for incoming messages. The execution first begins with all vertices active, a user-defined function is then applied to each one of these, and vertices can halt and become inactive as appropriate. Working in iterations, also called supersteps, once all vertices are processed for a specific step, message exchanges between vertices are performed and vertices that receive a message become, or remain, active. Once all message exchanges are performed, a new superstep begins: reapplying the user-function to all active vertices, then performing message exchanges before a new superstep begins. This iterative execution repeats until all vertices are inactive.

Whilst this model provides significant programmability advantages, and exposes a large degree of latent parallelism, it also suffers from numerous irregularities that impact performance. In fact, when it comes to common properties associated with irregular applications, the vertex-centric model exhibits many sources of irregularity:

- **Fine-grain synchronisations**: the communications in vertex-centric programs take place at a vertex's mailbox level, hence any data-race protection must be implemented on a per-vertex basis.
- **Unpredictable memory access patterns**: broadcasting a message from a vertex to its neighbours means emitting a vertex-specific number of messages, aggravated by the power-law distribution of the numbers of neighbours per vertex. Also, the recipient vertices are unlikely to reside next to each other in memory, making these memory accesses unpredictable both in terms of quantity and location.
- **Load-imbalance**: the number of active vertices may drastically vary from a superstep to the next, and the numbers of neighbours may drastically vary too from a vertex to the next.

Since the challenges faced in vertex-centric programs echo those of a larger class of irregular problems, optimisations that are successful for vertex-centric are likely to be applicable or adaptable to other heavily irregular applications such as social network analysis or graph databases for instance.

Vertex-centric programs are inherently difficult to optimise

We thank the reviewers for their helpful feedback and suggestions. This research was supported by the UK Engineering and Physical Sciences Research Council (grant number EP/L01503X/1, CDT in Pervasive Parallelism)

because they tend to take the form of short user provided source code, resulting in the fact that the underlying framework is provided with little information to leverage for performance. In the meantime, since programmability is the essence of vertex-centric, any performance optimisations must not impact the programmability properties of the framework.

There is a serious tension here – performance optimisation against programmability [2] – and many attempts to optimise the vertex-centric model have negatively impacted the ability to easily program this paradigm. By contrast, our approach integrates the preservation of vertex-centric programmability in its core. To that end, all the optimisations discussed in this paper are entirely encapsulated within the iPregel framework, requiring no user source modification to take advantage of them.

The main contributions presented in this paper, which can be summarised as follows:

- **A hybrid combiner** designed to couple lock-free and lock-based interactions in order to efficiently handle fine-grain synchronisations.
- **The externalisation of vertex attributes** to better cope with unpredictable memory accesses by improving the cache efficiency through the grouping of vertex attributes based on their access frequency.
- **An edge-centric workload representation** that improves load-imbalance and preserves both the vertex-centric paradigm and Pregel user interface.

The rest of this paper is organised as follows: Section II depicts the context in which this research takes place. Sections III, IV and V introduce the optimisations considered in this paper. Section VI describes the environment in which experiments were conducted while Section VII presents and analyses the results obtained. This paper then concludes in Section VIII; summarising the findings of this work as well as discussing potential future work directions.

II. Related work

Over the last decade, the vertex-centric programming model has received great attention due to the programmability it offers to the user and the amount of parallelism inherently provided. There have been some attempts by the vertex-centric community to address the implications of irregular workloads present in graph processing algorithms.

Bypassing the costly selection of active vertices has been addressed, whether it has to be done manually by the user such as GraphChi [3] or automatically thanks to the analysis of algorithmic patterns as introduced in early iPregel work [4]. However the dispatching of active vertices to threads remains an unsolved challenge. Indeed, accurately evaluating the workload contained in these active vertices is key to an efficient workload dispatch. The common approach in vertex-centric frameworks consists in distributing an equal number of active vertices to each worker. However, this approach is sub-optimal due to the power-law degree distribution that typically underpins the graphs processed. This observation led to the development of PowerGraph [5], where the authors adopted a more edge-centric approach, but which resulted in an entirely new interface based on the scatter-apply-gather design instead of the typical Pregel single user-defined function. As such, many of the abstraction and programmability benefits of the vertex-centric model were lost.

The edge-centric approach was taken one step further by X-Stream [6], whose implementation and interface are entirely designed from an edge-centric perspective; exposing an edge-centric Gather-Apply-Scatter interface to the user. The underlying motivation for this design was to address the randomness of memory accesses, by reading the graph's edges sequentially. Nonetheless, such edge-centric frameworks, which by definition are no longer vertex-centric hence cannot provide the same benefits, demonstrate that addressing the vertex-centric challenges without sacrificing certain aspects of the actual programming model is not a trivial task.

This is also illustrated in the implementations for the fine-grain synchronisations required in vertex-centric programs during message exchange. Typically, either communications are redesigned as pull-based [3] so they are lock-free, or a semaphore is needed for each vertex mailbox where appropriate. An alternative approach for the latter is to implement the message combination as a compare-and-swap, as illustrated in Ligra [7]. However, despite providing performance benefits, this design again reduces the level of abstraction and requires the end programmer to interact with the framework at a lower level, potentially requiring a rewriting of certain parts of their code, in addition to raising additional restrictions that will be discussed in details later in this paper.

As described in this section, the application of optimisations for dealing with irregular workloads in vertex-centric frameworks typically results in the sacrifice of vertex-centric aspects, features or programmability. By contrast, in previous work [2], we demonstrated that vertex-centric optimisations could be designed without generating such unwanted side effects. In this paper, we continue this direction and focus on developing optimisations to cope with vertex-centric irregularities without sacrificing the actual vertex-centric programmability or requiring a single user source code rewrite.

III. Fine-grain synchronisations

Vertex-centric programs require each vertex mailbox to be protected against potential data-races. The write-interactions with that mailbox are achieved during combination, which makes combiners a key area for optimisations since any improvement in their design will have a direct impact on the performance observed. To implement them, two designs are available:

- **lock**: a classic design where vertices acquire the lock held on the recipient vertex, check if that recipient vertex already received a message, and if so the sender vertex combines the existing message with the new one. Otherwise, it simply pushes the new message, before releasing the lock.
- **compare-and-swap**: a lock-free design where vertices retrieve the existing message from the recipient's vertex

mailbox, combine it and push it back using a compare-and-swap operation that atomically checks if the value of the mailbox message is identical to that read earlier. If so, it updates it with the new value and returns *true*; otherwise, it means the value changed, which implies that another vertex updated this recipient's mailbox first, in which case the entire operation is repeated until it eventually succeeds.

The second approach has the advantage of avoiding locks, thus resulting in a performance gain. However, it systematically combines the new message with the existing one, therefore relying on the assumption that mailboxes begin with a default message value that is neutral to the combination operation applied. For instance, in a combination operation that sums messages, vertices mailboxes would begin each superstep with a message value of 0. This need of a neutral value implies that either the user must be constrained to a set of predefined combination operations whose neutral values are hardcoded, or else the user must somehow declare the neutral value for the combination operation they write. In the Ligra version of PageRank, for instance, the combination operation is a sum (thus having the neutral value 0). For the user, this results in having to manually reset each vertex mailbox to 0 at the end of every superstep.

The second drawback of a pure compare-and-swap design comes from the lack of a notion of empty mailboxes. Indeed, mailboxes always have a message, either representing the result of a combination, or being the neutral value by default. Therefore, a vertex knows it has received a message if the mailbox message value is different from the neutral value. Yet, in a scenario where the combination operation would result in the neutral value itself, the vertex would assume it has not received a message, while in fact it has. In vertex-centric programs, this can lead to incorrect outputs since receiving messages is what reactivates inactive vertices.

In order to make the best of both designs, that is; exploiting compare-and-swap while keeping the notion of an empty mailbox, as well as letting the user define any arbitrary combination operation, we designed a hybrid combiner that leverages lock-based and lock-free interactions with the recipient mailbox. Its implementation is provided in Figure 1, where vertex attributes have been shortened for brevity. In this example, *ip_combine* is the user-defined combination function, *has_msg_next* is the flag indicating whether the vertex already received a message during this superstep and *msg_next* is the message itself (whose value is meaningful only if the flag is *true*).

As shown in Figure 1, the hybrid combiner carefully couples lock-free and lock-based interactions. Correctness comes from the guarantee that if the *has_msg_next* flag of a recipient vertex is *true*, the value held in that vertex mailbox is set. Indeed, as soon as the *has_msg_next* flag is *true*, potential compare-and-swap combinations may happen concurrently on that vertex from other threads. Therefore, the value they will fetch from that recipient vertex mailbox must have been set by that time.

To satisfy this guarantee, when a thread pushes the first message to a recipient mailbox, it stores the message (line 25)

```
1  void apply_cas(IP_VERTEX_TYPE* dst,
2                 IP_MESSAGE_TYPE msg){
3    IP_MESSAGE_TYPE old_msg = dst->msg_next;
4    IP_MESSAGE_TYPE new_msg = old_msg;
5    ip_combine(&new_msg, msg);
6    while(new_msg != old_msg &&
7          !atomic_compare_exchange_strong(
8            &dst->msg_next, &old_msg, new_msg)) {
9      old_msg = dst->msg_next;
10     new_msg = old_msg;
11     ip_combine(&new_msg, msg);
12 } }
13
14 void ip_send_message(IP_VERTEX_ID_TYPE dst_id,
15                      IP_MESSAGE_TYPE msg) {
16   IP_VERTEX_TYPE* dst=ip_get_vertex_by_id(dst_id);
17   if(dst->has_msg_next) apply_cas(dst, msg);
18   else {
19     ip_lock_acquire(&dst->lock);
20     if(dst->has_msg_next) {
21       ip_lock_release(&dst->lock);
22       apply_cas(dst, msg);
23     } else {
24       dst->message_next = msg;
25       dst->has_message_next = true;
26       ip_lock_release(&dst->lock);
27 } } }
```

Fig. 1. Implementation in iPregel of the hybrid combiner

before setting the flag to *true* (line 26). In addition, in order to avoid a potential out-of-order execution, a full memory barrier is required in-between. It is achieved by declaring the *has_msg_next* flag as atomic, using C11 atomics, which implicitly enforces a sequentially consistent memory model. Without it, an out-of-order execution could result in a recipient vertex entering a state where its *has_msg_next* flag is set to *true* while not having its *msg_next* message set yet. Another thread meaning to push a message to that vertex mailbox would therefore check the flag, see it is *true* and apply a compare-and-swap with the very message that is still unset. Finally, having the *has_msg_next* flag as atomic implies that the read at line 18 and write at line 26 are atomic too; guaranteeing that a read cannot happen on a flag partially written to memory.

The rest of the hybrid combiner is rather straightforward; threads check if the recipient vertex already has a message and if so they use a compare-and-swap combination, otherwise they acquire the lock. When the lock is acquired by a thread, it checks once again the recipient vertex flag in case, while it was waiting to acquire the lock, another thread that was holding that lock pushed the first message into that recipient mailbox. In this event, the recipient vertex now has a mailbox containing a set message therefore the thread can release the lock and use the compare-and-swap combination. Otherwise, it holds the lock and performs the first message push to that recipient vertex mailbox, making sure the store operations are issued in the order explained earlier.

IV. Unpredictable memory access patterns

In vertex-centric programs, the irregularity in memory accesses is two-fold. First, the power-law distribution that typically underpins the graphs processed results in vertices having widely different numbers of neighbours. Second, the inherent irregular structure of graphs allows each vertex to be connected with any other arbitrary vertex. In other words, the data for neighbouring vertices may reside at any location in memory, thus unlikely to be contiguous with each other. As a consequence, when a vertex broadcasts a message to its neighbours, there are an arbitrary number of memory accesses to perform each at an arbitrary memory location.

These memory access patterns, although unpredictable, expose one regularity: the attributes accessed from these neighbours vertex structure. In iPregel, when using the pull-based version, the vertex structure contains among other attributes a *flag* indicating if it has data to broadcast and a variable for the actual *message*. The combination consists in iterating through neighbours, checking their *flag* and fetching their *message*.

However, while never accessed during this combination process, the other vertex attributes are still loaded into cache along with the meaningful attributes sharing the same vertex structure. Nonetheless, this cache pollution can be minimised by reorganising the vertex structure; externalising the frequently accessed attributes into their own structure. In other words, one array would contain structures made of the *flag* and *message* attributes, while the other array would contain structures made of the rest of the vertex attributes. Therefore, such a design allows cache lines to be loaded only with useful attributes.

V. Irregular workload

A common irregularity that parallel programs face is that of load imbalance, where processes or threads have associated with them different amounts of work. Vertex-centric programs, where vertices can become inactive during execution and contain different numbers of edges, are therefore prone to load imbalance.

A. Workload evaluation proxy

Finding the right proxy to evaluate the workload is crucial since it lays down the foundations on which build more advanced strategies like load-balancing. Logically, implementations of the vertex-centric programming model represent workload in terms of vertices. Although this is accurate with regular data structures, the graphs processed by vertex-centric programs typically follow a power-law distribution, resulting in widely different number of neighbours per vertex. In addition, the runtime of typical vertex-centric programs is dominated by communications and not computation. While the latter is related the number of vertices, the former depends on the number of edges. Based on this observation, our hypothesis was that the workload of a thread, which results in the number of combination operations performed and memory writes, or reads, is better expressed as being correlated to the number of outgoing, or incoming, neighbours.

B. Work distribution

When parallelising a *for* loop in OpenMP by using the *for* construct, one can apply the *schedule* clause, which is provided with a scheduling kind describing how chunks of iterations will be distributed to threads as well as an optional parameter specifying how many iterations make a chunk.

One of the scheduling kinds provided by OpenMP is *dynamic*, specifying that chunks of iterations will be distributed on a first-come-first-served basis. This allows threads that have been assigned lighter chunks to be assigned more chunks, thus improving load-balancing.

To be compatible with this technique, the code must be within a *for* loop whose iteration set distribution can be freely managed by OpenMP. This is compatible with all versions of iPregel that do not rely on the workload shift from vertex-centric to edge-centric described in Subsection V-A. Indeed, the edge-centric workload negates the use of OpenMP dynamic scheduling because the workload is represented as edges and not vertices. The assigned chunks therefore represent workloads on a per-vertex basis instead of edge-centric one.

VI. Experimental environment

This section describes the conditions and configurations in which the experiments presented in this paper were conducted.

A. Computing environment

Experiments are run on a standard compute node of an HPE 8600 cluster, set up with CentOS 7 Linux and containing two 2.1 GHz, 18-core Intel Xeon E5-2695 (Broadwell) series processors and 256GB of memory split in two 128GB non-uniform memory access regions, one local to each processor.

The compilation is achieved by using the GCC compiler version 8.2.0 with OpenMP version 4.5. Compilation flags passed enable the support for C11 standard (*-std=c11*) and level 3 optimisations (*-O3*).

B. Graph configurations

Table I lists the graphs processed in the experiments presented in this paper. All four are real-world graphs publicly available in the Stanford Network Analysis Project [8] online collection. The smallest graph, the Database and Logic Programming Bibliography graph (DBLP), represents the eponymous computer science bibliography while LiveJournal, Orkut and Friendster are network graphs about blogging, social and gaming respectively. These graphs cover all orders of magnitude from a million to a billion edges and are undirected, meaning that the total number of directed edges is twice the amount presented.

C. Benchmarks

Experiments presented in these paper are conducted on three benchmarks commonly used by the vertex-centric community.

PageRank, or PR, was originally presented in [9]. This iterative algorithm ranks webpages by evaluating their importance; taking into account a ratio between incoming and outgoing hyperlinks. In iPregel, PR is best implemented using the

TABLE I
NUMBER OF VERTICES AND EDGES IN THE GRAPHS SELECTED FOR EXPERIMENTS

Name	Vertex count	Edge count
DBLP	317,080	1,049,866
Live Journal	4,036,538	34,681,189
Orkut	3,072,441	117,185,083
Friendster	65,608,366	1,806,067,135

single-broadcast version, where communications are achieved by pulling messages from their sender's outbox.

Connected Components, also abbreviated CC, locates in a graph all the subgraphs in which any two vertices are connected to each other by paths but connected to no vertex outside that subgraph. In iPregel, the CC benchmark is best implemented using the single-broadcast with selection bypass version, where in addition to leveraging pull-based communications, the framework keeps track of active vertices to better dispatch them to threads at every supersteps.

Single-Source Shortest Path, or SSSP, consists in selecting a vertex and finding the shortest path from that vertex to each other vertex it can reach. Experiments presented in this paper use the unweighted version of SSSP, where all edges represent a distance of 1. In iPregel, SSSP is best implemented using the selection bypass version described above.

Further details about each internal version of iPregel as well as a detailed analysis of the benchmarks can be found in [4] and [2]. As discussed in Section I, we have designed the optimisations of Sections III, IV and V in a manner that requires no modifications in user code. As such the versions of the benchmarks in [4] and [2] have remained unchanged in the experiments of this paper.

VII. RESULTS

The results presented in Table II consist in applying every optimisation individually and calculating the speed-up obtained against the baseline version, before repeating the process with a version aggregating all applicable optimisations.

A. Individual optimisations

The results presented in Table II show that the hybrid combiner improves the performance of SSSP on all graphs. It also proves to be the optimisation yielding both the biggest speed-up overall, up to 4.07 on Friendster, and on average, with a geometrical mean of 1.81. In addition, as the size of the graph increases, so does the speed-up. The reason for this is that the number of combinations depends on the number of edges, the benefit of improving the combination therefore grows along with the number of combinations generated. These results demonstrate that efficiently handling fine-grain synchronisations does not have to sacrifice programmability.

Similarly, the externalised structure optimisation is beneficial in all graph-benchmark pairs tested, with a speed-up of 1.30 on average. The results in Table II show that externalising vertex attributes generates the best speed-ups

TABLE II
SPEED-UPS OBTAINED FROM EACH OPTIMISATION APPLIED INDIVIDUALLY, THEN TOGETHER, FOR EACH BENCHMARK, USING 32 THREADS, ON ALL GRAPHS ORDERED BY ASCENDING NUMBER OF EDGES

	DBLP	Live Journal	Orkut	Friendster
PR (10 iterations)				
Baseline	1.00	1.00	1.00	1.00
Externalised structure	1.31	1.27	1.51	1.13
Edge-centric workload	1.01	2.31	1.67	1.36
Dynamic scheduling	1.23	2.31	1.99	1.44
Final	1.61	3.14	3.07	1.63
CC				
Baseline	1.00	1.00	1.00	1.00
Externalised structure	1.58	1.66	1.47	1.65
Edge-centric workload	0.56	1.12	1.27	1.41
Dynamic scheduling	1.23	1.67	1.69	1.20
Final	2.05	2.96	2.41	2.12
SSSP				
Baseline	1.00	1.00	1.00	1.00
Hybrid combiner	1.01	1.12	2.35	4.07
Externalised structure	1.08	1.01	1.07	1.10
Edge-centric workload	0.91	0.87	1.28	1.29
Dynamic scheduling	1.11	1.33	1.55	1.69
Final	1.09	1.75	3.18	5.63

for CC and the worst ones for SSSP. The explanation is twofold, firstly, PR and CC benefit more because they rely on iPregel versions that use pull-based communications that are lock-free by design. As a consequence, the memory accesses performed during the communications are not interleaved with lock acquisition or release, which reduces further the number of vertex attributes that are frequently accessed. Secondly, PR and CC rely on different algorithms; the one underpinning PR has one loop that can leverage structure externalisation while the one for CC has two. The overall benefit obtained on CC is therefore greater since it can leverage this optimisation in two parts of the code. Overall, structure externalisation therefore demonstrates that heavily irregular memory access patterns may exhibit certain regular aspects when analysed from a different angle, which can be leveraged for performance gains.

The timings reported in Table II also indicate that shifting to the edge-centric workload proves to be beneficial in 75% of the experiments; yielding a speed-up of 1.19 on average. The extremes are observed for PR on Live Journal with a speed-up of 2.31 and Connected Components on DBLP with only 0.56. In fact, the edge-centric workload representation performs better on PR than on any of the two other benchmarks. The reason for this is that CC and SSSP rely on an iPregel implementation leveraging the *selection bypass* optimisation introduced in [4], which helps cope with variable number of active vertices but requires the edge-centric workload distribution to be recalculated at every superstep, therefore increasing the total overhead.

The OpenMP *dynamic* scheduling is the fourth optimisation explored in these experiments. The results presented in Table II are obtained with an empirically determined chunk size of 256. Unlike the edge-centric optimisation, the *dynamic* scheduling turns out to improve the performance in all experiments;

resulting in speed-ups between 1.11 and 2.31. With an average speed-up of 1.50, the first-come-first-served dispatch pattern proves to be an efficient load-balancing strategy.

B. Aggregated optimisations

The optimisations combined, which exclude edge-centric workload in favour of the better performing dynamic scheduling, do not include the hybrid combiner in the case of PR and CC since they rely on iPregel versions that are lock-free. For PR and CC, the speed-up patterns exhibited are identical; the smallest and biggest graphs benefit the least. Live Journal and Orkut show better speed-ups, with Live Journal benefiting the most. In the case of PR, the speed-ups obtained range from 1.61 up to 3.14. For CC, the speed-up range is less spread; reaching only a maximum of 2.96 but never going below 2.05.

For SSSP, the speed-up pattern exhibited is different: the bigger the graph, the higher the speed-up as Table II shows. This can be explained by the presence of the hybrid combiner, which provides such a speed-up pattern, in addition to the dynamic scheduling also exhibiting that pattern for SSSP in Table II. Starting at 1.09 on the smallest graph, the speed-up obtained increases until reaching 5.63 on the biggest graph.

Overall, when fixing the number of threads at 32, the optimised versions prove to be beneficial for all benchmarks on all graphs tested. On average, the optimised versions cut almost two thirds (59%) of the runtime measured, with extremes cases observed of 8% and 82%.

VIII. Conclusions and future work

In this paper we explored multiple optimisations to address the irregular challenges inherent to vertex-centric programs.

The fine-grain synchronisations that underpin message combinations presented in Section III was the first one. By carefully coupling compare-and-swap and lock-based operations, we designed a hybrid combiner that, while transparent to the user, can drop the runtime by up to 75% as shown in Table II.

In Section IV, we analysed the unpredictable memory access patterns from a different perspective and found that although one cannot know which structure will be accessed next, one knows which structure attribute will. We therefore exploited this characteristic to redesign vertex structures for cache efficiency; decreasing the runtime measured by up to 40% as shown in Table II. We also considered the temporality of these accesses to leverage software prefetching, despite yielding no performance benefit in this case due to the bandwidth-intensive nature of vertex-centric programs.

Our approach for the third challenge, the load imbalance presented in Section V, was to better evaluate the workload by representing it with an edge-centric metric while preserving the user interface. Table II reports that this shift, beneficial in 75% of the tests and providing a runtime reduction of up to 57%, proved to be less productive than the OpenMP *dynamic* scheduling which never resulted in performance degradation while equalling the maximum runtime gain of edge-centric.

Overall, the experiments conducted in this work show that the optimisations considered yield performance benefits in 37 out of the 40 graph-benchmark pairs tested. When combined, these successful optimisations proved to yield performance benefits in all graph-benchmark pairs tested as shows Table II. This demonstrates that although the vertex-centric model exhibits many sources of irregularity, they can be efficiently addressed, reducing the runtime measured by up to 82%.

Future directions for this work could include the integration of work-stealing in the edge-centric workload, for example by designing an affinity schedule tailored for edge-centric. Another direction could be that of incrementalisation [10]; an optimisation area under-explored in vertex-centric but which could unlock a new level of performance. Finally, focusing on distributed-memory architectures would raise new challenges, calling for new optimisations to be designed.

References

[1] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807184

[2] L. A. R. Capelli, Z. Hu, T. A. K. Zakian, N. Brown, and J. M. Bull, "ipregel: Vertex-centric programmability vs memory efficiency and performance, why choose?" *Parallel Computing*, vol. 86, pp. 45 – 56, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819118303788

[3] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 31–46. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387884

[4] L. A. R. Capelli, Z. Hu, and T. A. K. Zakian, "ipregel: A combiner-based in-memory shared memory vertex-centric framework," *Proceedings of the 47th International Conference on Parallel Processing Companion - ICPP '18*, 2018. [Online]. Available: http://dx.doi.org/10.1145/3229710.3229719

[5] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX, 2012, pp. 17–30. [Online]. Available: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez

[6] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 472–488. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522740

[7] J. Shun and G. E. Blelloch, "Ligra," *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '13*, 2013. [Online]. Available: http://dx.doi.org/10.1145/2442516.2442530

[8] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[9] S. Brin and L. Page, "Reprint of: The anatomy of a large-scale hypertextual web search engine," *Computer Networks*, vol. 56, no. 18, pp. 3825 – 3833, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128612003611

[10] T. A. Zakian, L. A. Capelli, and Z. Hu, "Incrementalization of vertex-centric programs," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 1019–1029.