iPregel: A Combiner-Based In-Memory Shared Memory Vertex-Centric Framework

Ludovic A. R. Capelli
The University of Edinburgh
l.capelli@ed.ac.uk

Zhenjiang Hu
National Institute of Informatics
hu@nii.ac.jp

Timothy A. K. Zakian
University of Oxford
timothy.zakian@cs.ox.ac.uk

ABSTRACT

The expressiveness of the vertex-centric programming model introduced by Pregel attracted great attention. Over the years, numerous frameworks emerged, abiding by the same programming model, while relying on widely different architectural designs. The vast majority of existing vertex-centric frameworks exploits distributed memory parallelism or out-of-core computations. To our knowledge, only one vertex-centric framework is designed upon in-memory storage and shared memory parallelism. Unfortunately, while built on a faster architecture than that of other vertex-centric frameworks, it did not prove to significantly outperform other existing solutions.

In this paper we present *iPregel*: another in-memory shared memory vertex-centric framework. The optimisations developed and presented in this paper particularly target three hotspots of vertex-centric calculations: selecting active vertices, routing messages to their recipient and updating recipients inbox. We compare *iPregel* against the state-of-the-art in-memory distributed memory framework *Pregel+* on three of the most common vertex-centric applications: PageRank, Hashmin and the Single-Source Shortest Path. Experiments demonstrate that the single-node framework *iPregel* is faster than its distributed memory counterpart until at least 11 nodes are used. Further experiments show that *iPregel* completes a PageRank application with an order of magnitude less memory than popular vertex-centric frameworks.

CCS CONCEPTS

• **Computing methodologies → Shared memory algorithms**; **Parallel programming languages**;

KEYWORDS

vertex-centric, lightweight, shared memory, in-memory

ACM Reference Format:
Ludovic A. R. Capelli, Zhenjiang Hu, and Timothy A. K. Zakian. 2018. iPregel: A Combiner-Based In-Memory Shared Memory Vertex-Centric Framework. In *ICPP '18 Comp: 47th International Conference on Parallel Processing Companion, August 13–16, 2018, Eugene, OR, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3229710.3229719

1 INTRODUCTION

Almost a decade ago, the vertex-centric model was introduced through *Pregel* [15]. Thanks to its expressiveness and scalability, this model rapidly found an echo. Vertex-centric applications were soon developed for social network analysis [16] and data analytics [1, 2].

The vertex-centric programming is designed upon calculations that are described at vertex level, a message passing interface between vertices, and the concept of superstep inspired from the Bulk-Synchronous Parallel model [18]. The BSP programming model is structured around three phases. First, elements perform local computations, then communications between elements take place. Finally, a global synchronisation is applied to all elements in order to guarantee they all completed previous phases. An example of a BSP superstep is given in Figure 1. This model provides an execution flow that is clear, and for which parallelisation is easy to design and reason about. Over the years, numerous implementations of the vertex-centric frameworks emerged.

The vast majority of existing frameworks use distributed memory parallelism, out-of-core computations or both. Such architectures allow to process graphs that would be too large to fit in RAM. This flexibility comes at the expense of additional overheads, due to disk IO for out-of-core solutions, and network communications for distributed memory systems. At the other end of the spectrum is the in-memory shared memory architecture, consisting of single-node solutions that use exclusively RAM storage. While being the fastest architecture, it suffers from the amount of RAM available on a single, which is typically very limited.

However, it is reported [20] that the memory consumption of existing vertex-centric frameworks can reach up to 800GB to process a graph occupying 28GB of disk space. The large memory usage of vertex-centric frameworks is also mentioned in [9]. Such an overhead is incompatible with a viable in-memory shared memory framework. There is therefore a need to greatly reduce the memory footprint of vertex-centric frameworks. In addition, the existing in-memory shared memory vertex-centric framework does not prove to clearly outperform other types of frameworks. However, we argue that the potential of the in-memory shared memory architecture, as illustrated with *Ligra* [17] in graph-centric programming, is yet to be leveraged in vertex-centric programming.

In this work, we target a machine that has the performance comparable to that of a modern laptop. This configuration supports the argument that experiments are run on a reasonably affordable hardware.

In this paper, we present *iPregel*; our in-memory shared memory framework, whose multi-version design is introduced in Section 3. Sections 4, 5 and 6 present the three core features provided by our framework *iPregel*, respectively:

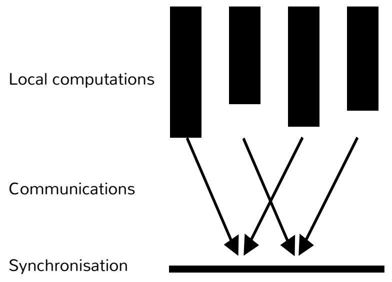

Figure 1: A Bulk-Synchronous Parallel superstep

- **Selection Bypass.** By exploiting communication patterns of vertex-centric applications, *iPregel* detects vertices to run before the selection phase.
- **Efficient Vertex Addressing.** Millions of messages may be sent during a superstep; rapidly finding recipients in memory is therefore essential. *iPregel* semantically enriches vertex identifiers to also represent vertex locations in memory and fasten message delivery.
- **Leverage of Combiners.** Updating recipients inbox is a hotspot of vertex-centric models. Combiners are a known optimisation, *iPregel* exploits them further by implementing several versions, including one that offers a race-free design.

Then, Section 7 relates the experiments conducted, from the methodology, graphs and applications used to the presentation and analysis of the results obtained. Finally, this paper concludes on Section 8.

2 RELATED WORK

The majority of existing vertex-centric frameworks today exploits distributed memory parallelism, such as *Pregel+* [19] and *Giraph* [5] to name a few. These frameworks are also in-memory; they use exclusively RAM for storage. Distributed memory parallelism thus acts as an additional supply of resources. However, as the size of graphs increases, so does the number of nodes needed. Inevitably, this results in larger distributed architectures, which generate more network communications hence exacerbate the fundamental bottleneck of distributed memory parallelism.

Technically, distributed memory systems using in-memory storage can process graphs of any size, given enough nodes. Nonetheless, a recent trend is observed in vertex-centric frameworks to move away from in-memory storage to out-of-core computations. Such frameworks expand their storage capacity from RAM-only to both RAM and disks. Typically, they offload unused data to disk and keep in RAM vertices and edges currently processed. The challenge of out-of-core computations is the disk IO, which needs to be hidden using overlapping techniques for instance. To date, several vertex-centric frameworks exploit both distributed memory parallelism and out-of-core computations, such as *GraphLab* [14], *Pregelix* [4] and *GraphD* [20].

The need for more memory can be satisfied with out-of-core computation and distributed memory parallelism. However, certain frameworks moved from distributed memory to shared memory parallelism, while preserving out-of-core computations. This can be observed with *GraphChi* [12], which is a spin-off of its distributed

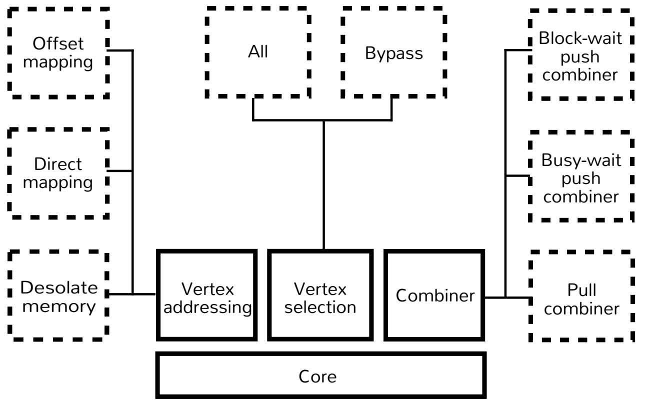

Figure 2: Structure of iPregel framework

memory counterpart *GraphLab* [14]. *FlashGraph* [6] is another example of an out-of-core shared memory vertex-centric framework. While still able to process large graphs, these frameworks benefit from shared memory parallelism, which is easier to program and reason about.

Finally, shared memory parallelism can also be paired with in-memory storage instead of out-of-core. The architecture obtained is free from disk IO and network communications, which makes it the fastest of all architectures presented in this section. *Ligra* [17] demonstrated the potential of in-memory shared memory solutions in graph processing compared to other architectures. Its counterpart in vertex-centric frameworks is *FemtoGraph* [3]. To our knowledge, it is the only in-memory shared memory vertex-centric framework. *FemtoGraph* demonstrated that at best it equals the performance of *GraphLab*, and remains slower until 28 cores are used.

The motivation of the work presented in this paper is twofold. First, despite relying on a faster architecture, no clear performance gains of in-memory shared memory frameworks were observed. This tends to suggest that there is room for optimisation to leverage the potential of such frameworks. Second, it seems that vertex-centric frameworks in general suffer from large memory footprints. As reported in [20], existing vertex-centric frameworks may require up to 800GB of memory to process a graph taking 28GB of disk space. Memory footprint is not a critical challenge to frameworks exploiting out-of-core computation or distributed memory parallelism since they have large amounts of memory available. In-memory shared memory systems however are significantly more limited, and being lightweight is crucial. Moreover, minimising the framework memory overhead maximises the size of graph that can be processed in RAM.

3 OVERVIEW OF IPREGEL

This section introduces the technical aspects of *iPregel*, from its architecture to the vertex representation, through its multi-version design.

3.1 Architecture

The *iPregel* framework is developed in C and parallelised using the shared memory API OpenMP [7]. In programming, certain

```
// Version-dependent
bool IP_get_next_message(struct IP_vertex_t* me,
                         IP_MESSAGE_TYPE* msg;
void IP_send_message(IP_VERTEX_ID_TYPE destID,
                     IP_MESSAGE_TYPE msg);
void IP_broadcast(struct IP_vertex_t* me,
                  IP_MESSAGE_TYPE msg);
void IP_vote_to_halt(struct IP_vertex_t* me);

// Version-independent
size_t IP_get_superstep();
bool IP_is_first_superstep();
size_t IP_get_vertices_count();
```

Figure 3: Functions provided by *iPregel*

optimisations are always applicable but most optimisations only hold under a given set of assumptions. Generally, the latter are not considered because they come at the expense of software flexibility. In *iPregel* however, this is mitigated via the multi-version design illustrated in Figure 2.

It is made of a core that is unique and static, and modules that have alternative implementations. Each module version is optimised for a specific set of assumptions. With this design, *iPregel* remains flexible while able to apply assumption-specific optimisations. The three modules visible in Figure 2, as well as their versions, are presented later in this paper, in Sections 4, 5 and 6.

3.1.1 Multi-Version Module Selection. In order to keep lower-level optimisations concerns hidden from the user, the multi-version layer is abstracted away from them. Nonetheless, the user knowledge about the graph processed or the application written are precious information that may trigger heavy optimisations. As such, they must be provided with a means to express this addition information. In *iPregel* this is achieved via defines passed via compilation flags. It is a mechanism already widely used in programming, it allows to keep application source codes untouched and requires little effort from the user. For instance, most vertex-centric applications use exclusively neighbour-broadcasts when communicating. Should a user write an application exposing this property, they can inform *iPregel* by passing the corresponding define during the compilation phase.

3.1.2 The core. The core is the static and central part of *iPregel*; it is invariant and fulfils numerous roles. For instance it is where the multi-version selection takes place. The core also acts as the interface to *iPregel* by providing all function declarations, which are consistent across module versions. However, there are two types of functions in *iPregel*.

First, there are functions that are declared and defined by *iPregel*. A list of the main ones is given in Figure 3. Among these functions, certain are version-dependent, that is, they have a different implementation in each module version. Nonetheless, they share the same function prototypes. This design choice allows to easily plug-in any additional implementation version. It also allows users to write their code once, and see it adapted to any module version.

```
void IP_compute(struct IP_vertex_t* me);
void IP_combine(IP_MESSAGE_TYPE* old,
                IP_MESSAGE_TYPE new);
```

Figure 4: User-defined functions of *iPregel*

Then, there are functions that are provided by *iPregel* too but whose implementation does not vary across module versions. These functions are in charge of auxiliary features such as keeping track of the supersteps, or the total number of vertices in the application. However, not all functions are defined by *iPregel*.

In addition to functions defined by *iPregel*, there are also functions declared in *iPregel* that must be defined by the user. The two major ones, listed in Figure 4, consist of `compute` and `combine`. The former contains the code to execute on each active vertex at each superstep, and the latter is called each time a vertex with a message in its mailbox receives a new message.

3.2 Vertex Representation

As an in-memory shared memory solution, *iPregel* is bounded to the main-memory available on the single node it runs on. Therefore, reducing its memory footprint is essential. Since the cornerstone of vertex-centric models is the vertices themselves, a great effort was made towards the design of *iPregel* regarding vertex representation.

In vertex-centric frameworks, a vertex contains the attributes that are specified by the user, and internals that are used by the framework itself, such as the active state of the vertex. Generally, vertex-centric frameworks rely on object-oriented languages like C++ and use a base class that contains all internals. That same class also has virtual methods such as `compute` and `combine`, presented in Figure 4. The user then derives their own class from the base class and customises it. However, due to the presence of virtual methods, a virtual table is created. This results in every vertex object carrying a hidden additional pointer, which increases the total memory footprint.

To avoid this memory overhead in *iPregel*, vertices are represented with structures. They contain arbitrary members provided by the user, as well as internals required by the framework. Due to the multi-version design of *iPregel*, internals too have alternative implementations, but again this is abstracted away from the user. In this case, a macro `IP_VERTEX_INTERNALS` is used to conceal the internals of *iPregel*. Concretely, the user defines the structure `struct IP_vertex_t` and includes the macro inside, then they are free to append any additional member needed in their application.

Having multiple possible vertex internals helps *iPregel* keep its memory footprint to a minimum. For instance, some applications require vertices to know both their in and out neighbours, while some others require only the former. On one hand, a single vertex internals design would have to consider the most conservative hypothesis and store both in and out neighbours. On the other hand, *iPregel* proposes several tailor-made internals (in only, out only, in and out) that take into account the module versions selected and the compilation flags passed from the user as explained in Section 3.1.

```
void IP_compute(struct IP_vertex_t* me) {
  IP_MESSAGE_TYPE ref = is_source(me->id)
                      ? 0
                      : UINT_MAX;
  IP_MESSAGE_TYPE m;
  while(IP_get_next_message(me, &m))
    ref = min(ref, m);
  if(ref < me->val) {
    me->val = ref;
    IP_broadcast(me, me->val + 1);
  }
  IP_vote_to_halt(me);
}

void ip_combine(IP_MESSAGE_TYPE* old,
                IP_MESSAGE_TYPE new) {
  if(*old > new) *old = new;
}
```

Figure 5: SSSP implemented in *iPregel*

3.3 Graphs Accepted

Similarly to numerous vertex-centric frameworks, *iPregel* expects vertex identifiers to be integral numbers, which is the case in the vast majority of graphs. In addition, *iPregel* requires vertex identifiers to be consecutive. Finally, it accepts static graphs only.

4 SELECTION BYPASS

The first phase of vertex-centric models consists in selecting the vertices to execute; known to be a delicate part of vertex-centric models [10]. Typically, frameworks (including *Pregel+*) iterate through all vertices at every superstep and check for each vertex its active state and inbox. Then, vertices that are already active or have a message in their inbox are run. However, unfruitful checks (i.e: inactive vertices) result in a waste of time and memory accesses. In this section, we present a technique which allows to bypass the selection phase, hence avoiding unfruitful checks.

It is observed that in many vertex-centric applications, all vertices systematically halt at the end of each superstep. It is visible for instance in the *iPregel* implementation[1] of SSSP, given in Figure 5. This characteristic implies that after the first superstep, a vertex is active if and only if it has received a message at the beginning of the superstep.

In other words, as soon as a vertex sends a message, it knows that the recipient vertex will have to be run during the following superstep because it will have at least one message in its inbox. The technique presented in this section consists in the sender adding its recipient identifier to the list of vertices to execute during next superstep. At the beginning of next superstep, the list established contains all vertices to run, so there is no need any more for checking or selection.

[1]It is assumed that *val* contains the distance from the source vertex, `UINT_MAX` contains a value greater than the maximum distance possible and that all edge weights are equal to 1.

From a parallel programming point of view, the selection bypass also participates to load-balancing by filtering vertices before they are split across threads. Indeed, before the selection phase, each thread receives an equal share of the vertices to process thus are in charge of identical numbers of vertices. With the traditional approach, one cannot know the proportion of active vertices in each share before the actual selection takes place. It follows that each thread may end up with a share containing a very low number of active vertices, or a very high one, resulting in load imbalance. With the selection bypass however, the vertices that are distributed across the threads are already known to be active. In other words, threads are guaranteed to run every vertex they are given. Therefore, with the selection bypass, the fact that threads are assigned equal shares implies that threads run identical numbers of vertices, which improves load-balance.

Note: In algorithms like PageRank where vertices may not halt at the end of each superstep, the implication between "active vertex" and "vertex that has received a message" does not hold. As a result, the selection bypass technique presented in this section is not applicable.

5 EFFICIENT VERTEX ADDRESSING

Vertex-centric models typically have a high number of communications, hence the importance of a quick message delivery. This section focuses on the first step of message delivery; finding the recipient vertex.

The vertex addressing mechanism is conventionally achieved with hashmaps matching vertex identifiers against their locations. This intermediate layer in the vertex addressing process incurs additional memory accesses, grows the memory footprint and exposes bad data locality inherent to hashmaps. In-memory shared memory solutions like *iPregel* typically store all vertices in a single array, so the location of a vertex is its index in that array. Based on the observation that most graphs use integral numbers as vertex identifiers, this paper proposes to semantically enrich vertex identifiers so they represent their vertex location as well.

The first strategy presented in this section is called **Direct Mapping** and is simple though fast: vertices are stored in the global array at the index equal to their identifier. For example, a vertex with identifier 5 resides at index 5 in the vertex array. This approach provides an *overhead-free* addressing mechanism but requires identifiers to start at 0, since *iPregel* is developed in C that is 0-indexed.

However, it may be the case that vertex identifiers start at an arbitrary number, the use of an offset must therefore be considered; resulting in what is called **Offset Mapping** in this paper. This offset is then subtracted to the vertex identifier to find the corresponding location. The offset therefore provides an index-identifier matching consisting of a simple subtraction, which is a marginal overhead.

Direct mapping can still be used in situation normally requiring offset mapping, with the technique named **Desolate Memory**. By forcing direct mapping, vertices will reside at the array index matching their identifier. Since in this case vertex identifiers have an offset, the array elements whose indexes are lower than the actual offset will be unused, resulting in a waste of memory. However, for graphs whose indexes start at 1 for instance, using desolate memory incurs the waste of a single element, which can be argued to be a reasonable memory sacrifice to benefit from direct mapping.

The addressing mechanism presented in this section allows for an efficient mapping between a vertex identifier and its location in memory. In addition, every addressing technique presented in this section can be used in conjunction with the selection bypass mechanism introduced in Section 4.

6 COMBINERS AND DATA CONSISTENCY

Once the location of the recipient vertex is found, the message sent must be appended to the recipient inbox; this is where combination takes place. Concretely, when a vertex sends a message, if its recipient inbox is empty then the message is added, otherwise it is combined with the existing one. It follows that shared memory combiners guarantee vertices to have *at most* one message in their inbox. This greatly reduces memory consumption, as well as making memory consumption more predictable, because no dynamically resizeable structure is needed to represent inbox messages. From an implementation perspective, several techniques are discussed in this section. It must be noted that techniques presented in this section are independent from (thus compatible with) the selection bypass (see Section 4) and the efficient vertex addressing (see Section 5).

6.1 Push-Based Combiner

The first combiner presented in this section is referred to as **Push-Based Combiner**, which designates the action of vertices to put messages in their recipient inbox. In this configuration, multiple vertices may send a message to a same recipient concurrently, which arises potential data-race and must then be prevented via synchronisation. This is achieved with the use of a lock that is acquired in turn by the threads executing the sender vertices to serialise the accesses to the protected data (the receiver vertex mailbox).

The most common locking technique, known as **Block-Waiting Synchronisation**, consists in blocking the threads awaiting to acquire a lock. The threads blocked are paused and put in a waiting queue, from which they will be taken off when they acquire the lock. By putting threads to sleep, this mechanism frees CPU resources that can be allocated to other threads. However, handling features such as a waiting threads queue requires a heavier lock structure, that is, fundamentally, more bytes.

This contrasts with another form of synchronisation called **Busy-Waiting Synchronisation**, where the threads repeatedly attempt to acquire the lock until they eventually succeed. This technique is generally discarded because threads are not put to sleep, hence waste CPU cycles while waiting. Yet, this mechanism presents two advantages over its block-waiting counterpart. First, when the critical section[2] is very small, such as combiners which typically consist of a compare-and-replace operation, busy-waiting locks prove to be more reactive because they do not incur thread pausing and resuming overheads. Also, not handling block-waiting features (i.e: pausing/queuing/dequeuing/resuming) make busy-waiting locks lighter.

In *gcc*, the compiler used in this work for *iPregel*, the block-waiting and busy-waiting synchronisations are implemented with

[2]The region of code protected by a lock.

```
void IP_compute(struct IP_vertex_t* me) {
  if(IP_is_first_superstep())
    me->val = 1.0 / IP_get_vertices_count();
  else {
    IP_MESSAGE_TYPE sum = 0.0;
    while(IP_get_next_message(me, &me->val))
      sum += me->val;
    me->val = 0.15 / IP_get_vertices_count()
            + 0.85 * sum;
  }
  if(IP_get_superstep() < ROUND)
    IP_broadcast(me, me->val
                 / me->out_neighbours_count);
  else
    IP_vote_to_halt(me);
}

void ip_combine(IP_MESSAGE_TYPE* old,
                IP_MESSAGE_TYPE new) {
  *old += new;
}
```

Figure 6: PageRank implemented in *iPregel*

mutexes and spinlocks respectively (the later requires GNU99 extensions). The former weights 40 bytes while the latter is only 4; which is a reduction of 90%. Since there is one lock per inbox and one inbox per vertex, this memory gain is to be multiplied by the total number of vertices. For instance, if considering the Wikipedia and USA graphs used in this research, which are given in Table 1, switching from mutexes to spinlocks drops the memory footprint of the data-race protection from 730 and 958 megabytes to 73 and 96 megabytes respectively. This is all the more valuable in *iPregel* where being lightweight is crucial.

6.2 Pull-Based Combiner

The combination process is understood in Section 6.1 as a sender pushing the message into the recipient's inbox. However, this section presents a mirrored approach where it is up to the recipients to fetch the messages sent by the senders, named **Pull-Based Combiner**. This technique is designed upon the observation that a high number of vertex-centric algorithms use neighbouring broadcasts[3] as their unique means of communications. In other words, every time a vertex communicates, an identical value is sent to all out-neighbours at once. This is visible for instance in PageRank whose *iPregel* implementation is given in Figure 6.

This approach requires to reverse the way communications are designed, and consists of three phases. First, vertices must be given their in-neighbours identifiers to locate the senders from which fetch messages. This is handled automatically in *iPregel*. Second, a sender vertex must buffer the message meant for broadcast in an outbox, as well as updating its internal state to indicate that it has a message to broadcast. Third, at the end of every superstep each vertex must iterate through *all* its in-neighbours outbox, fetch the

[3]By opposition to graph-wise broadcast.

broadcast message (if any) and add it to its own inbox (or combine it with its existing inbox message if any).

With this technique, inter-vertex interactions are exclusively read-only (i.e: fetching messages) while write actions (i.e: combination) are kept intra-vertex. Since a vertex is processed by a single thread, it follows that threads never modify the value of a vertex they do not run. Therefore, there is no risk of data-race, which means that pull-based combiners provide a key improvement for parallelism performance: a race-free design. However, these benefits are impacted by two factors:

(1) **The ratio of active vertices** because each vertex must fetch messages from its in-neighbours at every superstep. Therefore, the more active vertices in these in-neighbours, the fewer unfruitful checks.
(2) **The number of in-neighbours** because each vertex must iterate through every one of its in-neighbours. Consequently, the fewer in-neighbours, the faster.

Nonetheless, locks are no longer needed, which drops the memory footprint of data-race protection to 0.

As said previously, vertices in *iPregel* are given their list of in and out neighbours at creation. Therefore, from an implementation perspective, each vertex stores a pointer to its array of in neighbours as well as a corresponding counter, identically for out neighbours. However, the vertex structure may contain neighbour information that will never be used, resulting in a waste of memory. One may suggest to set the unused counters to 0 and the unused pointers to null. But storing unused pointers and counters wastes several bytes per vertex, surging to a total of approximately 250MB[4] when considering 20 millions vertices such as the graphs selected for experimentation (see Section 7.1.3). It has therefore been decided to indicate unused neighbour information through compilation flags; that allows *iPregel* to select the lightest structure to represent a vertex and keep its memory footprint to a minimum.

6.3 Single Message Mailboxes

With the use of combiners, vertex mailboxes can have two states: empty or containing one message. Upon reception of a new message, an empty mailbox takes it as is, and a mailbox with an existing message combines it with the existing one. Either way, at most one message is contained in a vertex mailbox. This allows *iPregel* to design the mailbox as one message and avoid the use of dynamically resizeable data structures as well as the memory overhead they incur. As a result, it greatly helps *iPregel* keep its memory footprint to a minimum.

7 EXPERIMENTS

7.1 Experimental Setup

7.1.1 Computing Environment. Experiments are run on Amazon EC2 using *m4.large* instances, which provide 8GB of DIMM memory, 2 cores of an Intel(R) Xeon(R) CPU E5-2686 v4, clocked at 2.30GHz, and a maximum bandwidth of 450Mbps. Instances are set-up with Ubuntu 16.04.3 LTS 64-bit operating system.

[4]Assuming a 64-bit operating system, hence 8-byte pointers, and an unsigned int counter, being 4-byte long in most implementations.

Table 1: Graphs used in the comparison with Pregel+

Name	$\|V\|$	$\|E\|$
Wikipedia	18,268,992	172,183,984
USA Road network	23,947,347	58,333,344

iPregel is compiled with *gcc* version 5.4.0, using C99 standard by default and GNU99 extensions when using spinlocks (see Section 6.1). The optimisation level is set to -O2, and to exploit both cores available on the EC2 instance, two OpenMP threads are used.

Pregel+ is compiled with *mpic++* (MPICH version 3.2), using *g++* version 5.4.0 with C++11 standard as the underlying C++ compiler. The optimisation level is set to -O2, and to exploit both cores available on each EC2 instance, two MPI processes are created per node.

7.1.2 Methodology. The experiments are initially run 5 times, and are repeated until the margin of error obtained represents less than 1% of the average runtime, given a confidence level of 99%. The timings collected report superstep execution time; that is, graph preprocessing and graph loading are not included. Memory consumption is measured as well, represented by the *maximum resident set size*[5] as returned by the bash command *time -v*.

7.1.3 Graphs Used. Experiments presented in this paper are conducted on graphs presented in Table 1. The Wikipedia is publicly available[6] at KONECT [11], so is the USA roads graph[7] at DIMACS [8]. Both graphs are made of contiguous indexes starting at 1, and are processed in *iPregel* using offset mapping with desolate memory as explained in Section 5.

7.1.4 Applications Selected. In this work, three applications have been selected for experiments, namely: PageRank, Hashmin and SSSP. These three applications are widely used in vertex-centric experiments and thus act as standards. They also expose three different evolutions of the number of active vertices: constantly all active in PageRank, decreasing from all active to none in Hashmin and in SSSP it starts with one active vertex typically followed by a bell evolution (increasing then decreasing).

Also, it is observed that all three applications exclusively use broadcasts, and are therefore compatible with the pull-based combiner presented in Section 6.2. However, only Hashmin and SSSP are compatible with the selection bypass of Section 4 because, unlike PageRank, their vertices vote to halt at the end of every superstep.

Note: PageRank experiments are run with 30 iterations and SSSP experiments use the vertex identified by '2' as the source.

7.2 Performance of iPregel Versions

The round of experiments presented in this section intends to evaluate the impact of the selection bypass and combination techniques presented in Sections 4 and 6 on performance. Concretely, each of the three applications presented in Section 7.1.4 is performed using

[5]It stands for the maximum amount of memory taken by a program throughout its execution.
[6]http://konect.uni-koblenz.de/networks/dbpedia-link
[7]http://www.dis.uniroma1.it/challenge9/data/USA-road-d/USA-road-d.USA.gr.gz

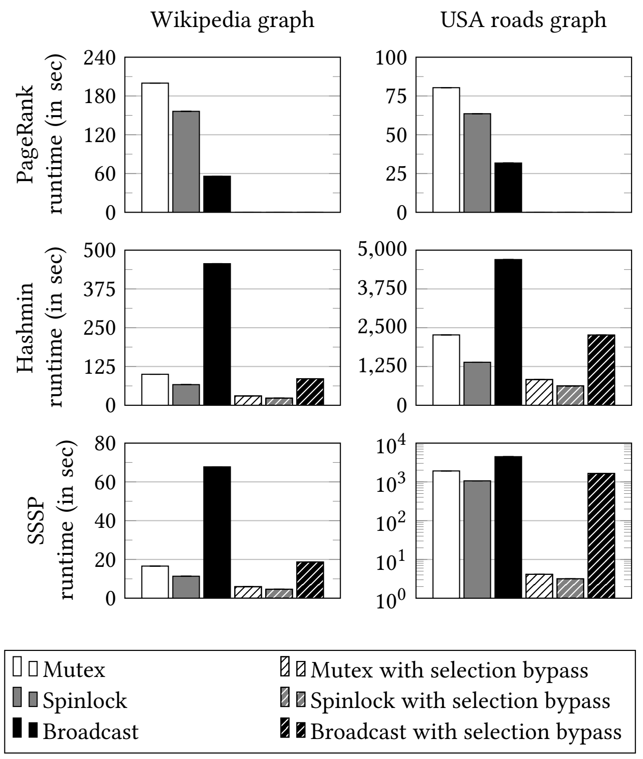

Figure 7: Runtime (in seconds) of iPregel on PageRank, Hashmin and SSSP as the version varies

every compatible version of *iPregel*. There are in total six versions possible: three combiners (see Section 6) which can be used with or without the selection bypass (see Section 4). Except for PageRank, which is implemented only in the three versions without selection bypass, Hashmin and SSSP are implemented in all six versions. The execution times collected are given in Figure 7.

On both graphs, we observe that PageRank execution times drop by about 30% between the mutex and spinlock versions. It is the broadcast version however which provides the best performance by halving the spinlock runtime and becoming the only version able to process the 30 PageRank iterations under a minute.

The runtimes of Hashmin and SSSP are similar in that that broadcast versions are slower than mutexes, themselves slower than spinlocks. In addition, all combiner versions become faster when they exploit the selection bypass. Therefore, the best version is always the spinlock combiner with selection bypass and the worst is always the broadcast version without selection bypass. By moving from the Wikipedia to the USA roads graph, the speed-up between these two fastest and the slowest versions increases from 7.5 to 20 for Hashmin, but surges from 15 to 1,400 for SSSP.

To analyse these differences, two factors must be considered: the **ratio of active vertices** compared to the total number of vertices, and the **graph density**.

The first factor directly impacts the performance of the broadcast version (a.k.a: pull-based combiner) as explained in Section 6.2. It turns out that PageRank offers an optimal ratio since all vertices stay active during the entire execution time. On the other hand, Hashmin and SSSP expose lower ratios, continuously decreasing and constantly low respectively. This first factor explains why the broadcast version performed well in PageRank and badly in both Hashmin and SSSP.

The second factor is the key to explain the surge in SSSP performance. A lower density means a smaller average out degree, which results in a slower propagation of messages, thus a high number of supersteps to completely reach a graph. In SSSP, the number of active vertices is constantly very low, which is optimal for the selection bypass. It is the lower density of the USA graph paired with the very low number of active vertices in SSSP that explain the gap between the versions using selection bypass to those which do not. Hashmin reaches a very low number of active vertices only in the late supersteps, which partially mitigates the benefits of a lower graph density.

In this first round of experiments, the broadcast version proves to be the fastest for PageRank, so is the spinlock with selection bypass version for both Hashmin and SSSP. Each of these versions is now meant to be compared against the state-of-the-art in-memory distributed memory solution: *Pregel+*.

7.3 Comparison with Pregel+

The existing in-memory shared memory vertex-centric framework is *FemtoGraph*. Unfortunately, we have not been able to observe correct results from this framework, even when using the PageRank implementation provided. Consequently, the comparison between these two frameworks sharing the same architecture cannot be made. Therefore, the comparison must include a framework that has an architecture different from that of *iPregel*. In other words, either out-of-core computation or distributed memory parallelism must be allowed. It has been decided to compare against a distributed memory system because despite suffering from network communications, it benefits from additional memory and processing power.

Yet, comparing shared memory and distributed memory solutions on a single node is unfair, so is comparing them over multiple nodes. The former disadvantages distributed memory solutions, which have an overhead for distributed parallelism, while the latter disadvantages shared memory solutions that by definition are single-node. As a consequence, experiments presented in this section are twofold: to compare the performance of *Pregel+* and *iPregel* on a single node, then determine the number of nodes required by *Pregel+* to outperform *iPregel* (referred to as *lead change* in the rest of this paper).

Results presented in this section are collected from experiments that are run with a maximum of 16 nodes. The *lead change* may not always be observed within this interval, in which case extrapolation is used by assuming the efficiency between 8 and 16 nodes to stay constant every time the number of nodes is doubled[8]. The same extrapolation method is used backward to estimate the runtimes for the number of nodes under which *Pregel+* fails to complete due to insufficient memory. The timings that have been collected are presented in Figure 8.

[8]Given an efficiency of x between 8 and 16 nodes, the runtime of 32 nodes is projected assuming an efficiency of x between 16 and 32 nodes.

Across both graphs, the execution time of *Pregel+* on PageRank remains stable at approximately 200 seconds. On the other hand, that of *iPregel* decreases by 43%, from slightly less than a minute to about 30 seconds. The *lead change* happens at 11 nodes on the Wikipedia and is estimated at 30 nodes for the USA graph.

To analyse the timings obtained for PageRank, it must first be reminded that the *iPregel* version used for that application implements the pull-based combiner introduced in Section 6.2. As explained in Section 7.2, PageRank characteristics are optimal for the pull-based combiner, making *iPregel* several times faster than *Pregel+*; by a factor of 3.57 and 6.47 on Wikipedia and USA graphs respectively.

The drop in the runtime of *iPregel* is caused by the variation of the number of in-neighbours between the two graphs. Indeed, the graph density is the second performance factor of pull-based combiner (see Section 6.2). Consequently, *iPregel* takes advantage of the fact that this number is divided by three between the two graphs and almost halves its runtime.

The round of experiments run on SSSP exposes two particularities: insufficient memory failures and the biggest difference between the performance of *Pregel+* and *iPregel*. On the Wikipedia graph, *iPregel* proves to be approximately 7 times faster than *Pregel+*, with almost 5 seconds against more than 33 seconds. This difference is multiplied by an order of magnitude on the USA graph, where the runtime of *Pregel+* grows by more than 560% to reach 221 seconds while that of *iPregel* falls by 30% to approximately 3 seconds; making it 70 times[9] faster than its distributed rival. Although the *lead change* is reached at 13 nodes for the Wikipedia graph, it is estimated that it would require more than 15,000 nodes for the USA graph. The low number of active vertices and the low graph density of the USA graph provide optimal conditions to exploit the potential of selection bypass.

The third and last set of experiments is run on Hashmin, and contains the longest execution times observed; up to almost one hour. Although *iPregel* and *Pregel+* process the Wikipedia graph in less than 25 and 150 seconds respectively, their runtime surge to more than 10 and 50 minutes[10] when it comes to the USA road network graph. The *lead change* remains constant however, requiring 11 nodes for both graphs.

The specificity of Hashmin is its variation of the number of active vertices throughout the computation: starting with all vertices active, progressively, they halt. Consequently, most of the execution time is spent on supersteps containing a medium number of active vertices, which does not permit to the selection bypass technique to reach its full potential as in SSSP. Furthermore, the low density of the USA graph makes the deactivation propagation all the more slower, which causes the soar in the runtimes observed. Hashmin is the only application in which the speed-up between *iPregel* and *Pregel+* decreases from the Wikipedia to the USA graph. Indeed, while *iPregel* is 6.5 times faster than *Pregel+* on the former, it is only 5 times faster on the latter.

Across all experiments, *iPregel* proves to be faster than *Pregel+* on a single node, where the former was naturally advantaged due to its shared memory design. However, it is not less than 10 additional nodes that are needed by *Pregel+* to outperform *iPregel*. Certain configurations like SSSP on the USA graph contain too few active vertices during too many supersteps, and make impossible for *Pregel+* to outperform *iPregel* within a reasonable number of nodes.

[9]Exact results are 220.97/3.17 = 69.70.

[10]Exact results are 624.13 and 3,065.03 seconds.

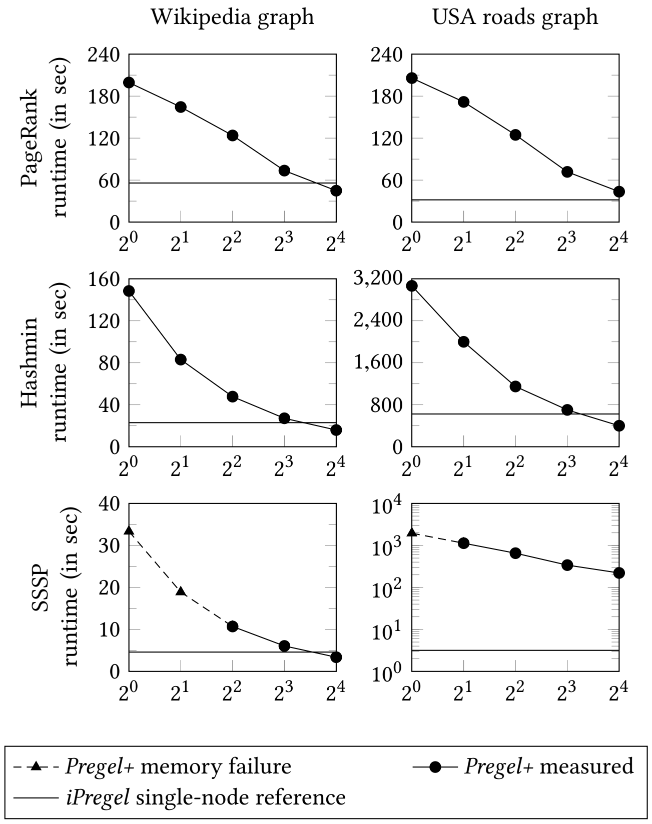

Figure 8: Evolution of the Pregel+ runtime (in seconds) of PageRank, Hashmin and SSSP as the number of nodes varies

7.4 Memory Footprint

7.4.1 Measured. The memory footprint of vertex-centric models is a known weakness [9]. Yet, being lightweight is crucial to in-memory shared-memory frameworks like *iPregel*. Indeed, being lighter increases the amount of vertices and edges that can be processed under a specific amount of memory. This is why the memory footprint of *iPregel* was measured too during experiments presented in Section 7.

On the Wikipedia graph, both mutex versions (with and without selection bypass) took 2GB of memory, while their spinlock counterparts needed 1.5GB. However, the use of selection bypass increased the memory footprint of the broadcast version from 1.5GB to 2.5GB. This is due to the out-neighbours information that are needed by the selection bypass on top of the in-neighbours information required by the pull-based combiner used in the broadcast version. Then, for the USA graph, it is observed that the memory consumption of all versions increased by 10%. It is due to the fact that vertices are heavier than edges that are typically just integers.

Table 2: Graphs used for further iPregel memory footprint experiments

Name	$\|V\|$	$\|E\|$
Twitter (MPI)	52,579,682	1,963,263,821
Friendster	68,349,466	2,586,147,869

In this case, between the Wikipedia graph to the USA graph, the 100M fewer edges do not compensate for the 5M additional vertices.

Overall, it is between 1.5GB and 2.8GB of memory that were necessary to *iPregel*, out of the 8GB available.

7.4.2 Breaking point. Throughout experiments presented in this paper, iPregel used at most 35% of the 8GB available in main-memory. To estimate the maximal size of graphs that *iPregel* can process under 8GB, additional experiments were conducted. The Twitter (MPI) graph presented in Table 2 is a KONECT [11] graph publicly available[11]. This graph is selected due to existing results about the memory footprint of *Pregel+* and *GraphLab* on this same graph. It must be reminded that the memory footprint of in-memory frameworks includes that of the graph itself. In order to distinguish the part required to store the graph from the memory overhead generated by the framework, the graph binary size is calculated. It takes into account that vertices store their identifier as well as those of their out-neighbours, and assumes 4-byte vertex identifiers. However, it excludes information specific to vertex-centric applications (such as the rank value in PageRank) and internals required by frameworks, which are considered as parts of the memory overhead. The binary size of the Twitter graph is calculated to 8GB, it follows that *iPregel* cannot process this graph with 8GB of RAM. Instead, an incremental approach was used to determine the breaking point of *iPregel* with 8GB of memory. Concretely, several synthetic graphs were generated[12], with a number of vertices and edges proportional to the original Twitter graph. That is, a synthetic graph described as 20% contains a fifth of the number of vertices and a fifth of the number of edges of the original Twitter graph. PageRank was then run by *iPregel* on each of the synthetic graphs, from the smallest to the largest, until it runs out of memory. The results obtained are shown in Figure 9. It turns out that up to 70% of the Twitter graph can be processed before memory failure occurs. In other words, *iPregel* is able to run PageRank on a graph made of 37 million vertices and 1.4 billion edges under 8GB of memory.

7.4.3 Projections. However, existing results related in [20] about *Pregel+* and *Giraph* consider PageRank run on the entire graph. Regarding *iPregel*, linear extrapolation drawn in Figure 9 indicates that 11GB would be sufficient. To verify this statement, a new Amazon EC2 instance was deployed, the *m4.xlarge*, which has 16GB of memory. PageRank was then run on a synthetic graph with a size identical to that of the original Twitter graph. In total, *iPregel* needs 11.01GB to run PageRank on the complete graph, compared to *Pregel+* that requires 109GB and *Giraph* which needs 264GB.

[11]http://konect.uni-koblenz.de/networks/twitter_mpi

[12]The out-degree distribution is not preserved but this has no impact on the size of the graph or the memory footprint of *iPregel*

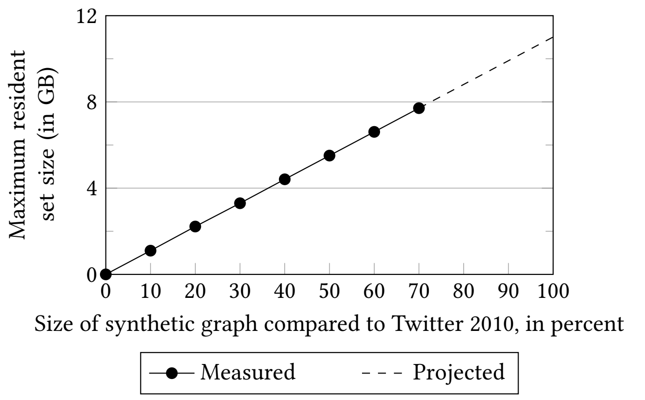

Figure 9: Evolution of the iPregel maximum resident set size (in GB) of PageRank as the size of synthetic Twitter graphs varies

The memory footprint of *iPregel* is therefore 10 times smaller than that of *Pregel+* and 25 times smaller than that of *Giraph*. Excluding the 8GB allocated to the graph itself, out of the 11GB taken by *iPregel*, 3GB are due to its overhead. Comparatively, the overheads of *Pregel+* and *Giraph* are 101GB and 256GB respectively; equivalent to 33 and 85 times that of *iPregel*.

A last experiment was run to estimate the biggest graph that *iPregel* could process under 16GB of RAM. To that end, two online graph collections KONECT [11] and SNAP [13] were parsed and the largest graph available overall was selected. It turns out to be the friendster graph from KONECT, publicly available[13]. As shown in Table 2, this graph is made of approximately 70 million vertices and 2.5 billion edges. The results collected reveal that *iPregel* is able to process PageRank on the Friendster graph with 14.45GB of memory. In other words, *iPregel* is able to process a multi-billion edge graph under 16GB of RAM.

7.4.4 Memory footprint analysis. The gap observed in memory footprints is explained by factors that are of two types.

To begin with, there are advantages inherent to the in-memory shared memory design. For instance, shared memory systems manage local communications only. This contrasts with distributed memory systems, in which messages between remote vertices are passed over the network, and typically involves the storage of sending and receiving buffers like in *Pregel+*. In addition, for the receiver node to know how to dispatch the messages received to their recipient vertex, messages are wrapped with the vertex identifier of the recipient vertex. This results in heavier messages, hence a memory overhead. Another advantage of the shared-memory structure is to avoid the storage of redundant information. Indeed, frameworks that use exclusively distributed memory exploit intra-node parallelism by creating multiple distributed workers per node. This leads to multiple instances of both the application and the distributed software environment to be stored in the memory of every node. The redundant copies therefore waste memory. Finally, frameworks that rely on distributed memory or out-of-core computations manage vertices that may reside in main-memory, on disk, or on a

[13]http://konect.uni-koblenz.de/networks/friendster

remote node. They must therefore use an additional addressing layer storing where each vertex currently resides, which increases the overall memory footprint too.

Then, come the differences that are due to the design of *iPregel*. The leverage of combiners, as explained in Section 6.3, permits to store at most one message per vertex mailbox. This avoids the use of dynamically resizeable structures like queues, replaced instead with a single variable of the message type. In addition, the memory overhead of data-race protection can be reduced to zero when using the lock-free structure provided by the pull-based combiner presented in Section 6.2. The multi-version design of *iPregel* also plays an important role in its overall memory footprint. Indeed, by selecting at compile-time the best structures to use, *iPregel* does not include vertex attributes that would be unused or left empty, such as in-neighbours. Finally, as explained in Section 3.2, the use of structures in *iPregel* avoids the hidden virtual pointer that is embedded in each vertex object when using derived classes like in *Pregel+*.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we propose a vertex-centric model that uses in-memory storage and shared memory parallelism. By allowing the user to pass additional information about the graph and the application via compilation flags, *iPregel* applies techniques optimised specifically for such contexts. The capacity of *iPregel* to adapt is provided by its underlying multi-version design, which permits to exploit assumption-specific optimisations without sacrificing software flexibility. In addition, this paper presented several optimisation techniques, from selection bypass to direct mapping, through a race-free combiner.

Experiments demonstrate that *iPregel* outperforms state-of-the-art in-memory distributed memory. On a single-node, timings collected show that *iPregel* is always faster than *Pregel+*, by a median factor of 6.5. At worst, it is 3.5 times faster, and up to more than 600 times in the best case. In addition, the performance achieved by *iPregel* remains competitive even when considering multi-nodes. Indeed, timings collected reveal that at least 11 nodes are needed by *Pregel+* to equal or outperform *iPregel*. The results obtained fulfil the first half of this work motivation; witnessing clear performance gains from an in-memory shared memory vertex-centric framework.

In addition, the performance gains of *iPregel* do not come at the expense of memory consumption. In fact, the memory footprint of *iPregel* is an order of magnitude smaller than its in-memory distributed memory counterparts; needing 11GB when the latter require up to a quarter of a terabyte. Further experiments demonstrated that *iPregel*, albeit single-node, can process multi-billion edge graphs under 16GB of memory, or the USA road network using less memory than that of a smartphone. The results collected demonstrate that there is room for optimisation in the memory footprint of vertex-centric frameworks, which satisfies the second half of this work motivation.

To conclude, *iPregel* demonstrates that the in-memory shared memory architecture is competitive in vertex-centric programming too. Further investigations about load-balancing strategies and internal parallelism would certainly benefit *iPregel*.

REFERENCES

[1] I. Abdelaziz, R. Harbi, S. Salihoglu, and P. Kalnis. 2017. Combining Vertex-Centric Graph Processing with SPARQL for Large-Scale RDF Data Analytics. *IEEE Transactions on Parallel and Distributed Systems* 28, 12 (Dec 2017), 3374–3388. https://doi.org/10.1109/TPDS.2017.2720174

[2] Ibrahim Abdelaziz, Razen Harbi, Semih Salihoglu, Panos Kalnis, and Nikos Mamoulis. 2015. Spartex: A vertex-centric framework for RDF data analytics. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1880–1883.

[3] Ballmer Alex, Walters Benjamin, and Raicu Ioan. [n. d.]. FemtoGraph: A Pregel Based Shared-memory Graph Processing Library. ([n. d.]). Poster at SC'16.

[4] Yingyi Bu. 2013. Pregelix: Dataflow-based Big Graph Analytics. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC '13)*. ACM, New York, NY, USA, Article 54, 2 pages. https://doi.org/10.1145/2523616.2525962

[5] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1804–1815.

[6] Disa Mhembere Da Zheng, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S Szalay. 2015. FlashGraph: Processing billion-node graphs on an array of commodity SSDs. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*. 45–58.

[7] L. Dagum and R. Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering* 5, 1 (Jan 1998), 46–55. https://doi.org/10.1109/99.660313

[8] The Center for Discrete Mathematics and Theoretical Computer Science (DIMACS). 2006. 9th DIMACS Implementation Challenge. http://www.dis.uniroma1.it/challenge9/download.shtml. (2006).

[9] Vasiliki Kalavri, Vladimir Vlassov, and Seif Haridi. 2016. High-Level Programming Abstractions for Distributed Graph Processing. (07 2016). arXiv:1607.02646 https://arxiv.org/abs/1607.02646

[10] Arijit Khan. 2016. Vertex-Centric Graph Processing: The Good, the Bad, and the Ugly. (12 2016). arXiv:1612.07404 https://arxiv.org/abs/1612.07404

[11] Jérôme Kunegis. 2013. KONECT: The Koblenz Network Collection. In *Proceedings of the 22Nd International Conference on World Wide Web (WWW '13 Companion)*. ACM, New York, NY, USA, 1343–1350. https://doi.org/10.1145/2487788.2488173

[12] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. 2012. Graphchi: Large-scale graph computation on just a pc. USENIX.

[13] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data. (June 2014).

[14] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. 2010. Graphlab: A new framework for parallel machine learning. arXiv preprint. *arXiv preprint arXiv:1006.4990* 1 (2010).

[15] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*. ACM, New York, NY, USA, 135–146. https://doi.org/10.1145/1807167.1807184

[16] Louise Quick, Paul Wilkinson, and David Hardcastle. 2012. Using Pregel-like Large Scale Graph Processing Frameworks for Social Network Analysis. In *Proceedings of the 2012 International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2012) (ASONAM '12)*. IEEE Computer Society, Washington, DC, USA, 457–463. https://doi.org/10.1109/ASONAM.2012.254

[17] Julian Shun and Guy E. Blelloch. 2013. Ligra. *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '13* (2013). https://doi.org/10.1145/2442516.2442530

[18] Leslie G Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111.

[19] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2015. Effective techniques for message reduction and load balancing in distributed graph computation. In *Proceedings of the 24th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 1307–1317.

[20] Da Yan, Yuzhen Huang, Miao Liu, Hongzhi Chen, James Cheng, Huanhuan Wu, and Chengcui Zhang. 2017. GraphD: Distributed Vertex-Centric Graph Processing Beyond the Memory Limit. *IEEE Transactions on Parallel and Distributed Systems* (2017).