NVRAM as an enabler to new horizons in graph processing

SN Computer Science (2022) 3:385
https://doi.org/10.1007/s42979-022-01317-4

ORIGINAL RESEARCH

NVRAM as an Enabler to New Horizons in Graph Processing

Ludovic Anthony Richard Capelli[1] · **Nicholas Brown**[2] · **Jonathan Mark Bull**[2]

Received: 24 June 2021 / Accepted: 7 July 2022

Abstract
From the world wide web, to genomics, to traffic analysis, graphs are central to many scientific, engineering, and societal endeavours. Therefore an important question is what hardware technologies are most appropriate to invest in and use for processing graphs, whose sizes now frequently reach terabytes. Non-Volatile Random Access Memory (NVRAM) technology is an interesting candidate enabling organisations to extend the memory in their systems typically by an order of magnitude compared to Dynamic Read Access Memory (DRAM) alone. From a software perspective, it permits to store a much larger dataset within a single memory space and avoid the significant communication cost incurred when going off node. However, to obtain optimal performance one must consider carefully how best to integrate this technology with their code to cope with NVRAM esoteric properties such as asymmetric read/write performance or explicit coding for deeper memory hierarchies for instance. In this paper, we investigate the use of NVRAM in the context of shared memory graph processing via vertex-centric. We find that NVRAM enables the processing of exceptionally large graphs on a single node with good performance, price and power consumption. We also explore the techniques required to most appropriately exploit NVRAM for graph processing and, for the first time, demonstrate the ability to process a graph of 750 billion edges whilst staying within the memory of a single node. Whilst the vertex-centric graph processing methodology is our main focus, not least due to its popularity since introduced by Google over a decade ago, the lessons learnt in this paper apply more widely to graph processing in general.

Keywords NVRAM · DCPMM · Vertex-centric

Introduction

Large-scale graph processing is an important activity which underlies many technologies. Taking the internet as an example, the fact that so many patterns of web-based interaction, from likes and friends on social networking sites, to click throughs can be represented as a graph data structure means that companies generate vast value from analysing such structures. Further afield, many communities including biological research, transportation planners, and communication specialists, also derive significant benefits from graph processing. However with the explosion in data, which is only set to continue, graph sizes are growing exponentially and an important question is how we can support the processing of next-generation graphs in the coming decades.

In graph processing, terabytes of memory can be required to hold large graphs which is orders of magnitude larger than what can be reasonably held within the DRAM of a single node. This is one of the key motivations underpinning the popularity of distributed memory graph processing [5] . In the distributed approach one is scaling across nodes due to memory limits rather than being driven by computational concerns, but this can result in numerous disadvantages. The first of which is the need to, often entirely, rewrite the shared memory implementation, commonly into some form of message-passing abstraction which requires communications to be explicitly programmed. Such inter-node communications are likely to result in significant communication overhead if they are

✉ Ludovic Anthony Richard Capelli
l.capelli@ed.ac.uk

Nicholas Brown
n.brown@epcc.ed.ac.uk

Jonathan Mark Bull
m.bull@epcc.ed.ac.uk

[1] School of Informatics, The University of Edinburgh, Edinburgh, UK

[2] Edinburgh Parallel Computing Centre, The University of Edinburgh, Edinburgh, UK

Published online: 20 July 2022

unstructured, which is the case for graph processing. Furthermore, graph processing is prone to heavy load imbalance due to the power-law distribution underpinning many graphs, and this forms a major obstacle to efficient distributed graph partitioning. An alternative approach especially popular in graph processing is that of out-of-core solutions [11], where the file system is used as a backing store for data and the DRAM as effectively a manual caching mechanism, fetching the data into DRAM as it is required and flushing unneeded data back to disk. Due to the disk being orders of magnitude slower than DRAM, the performance of out-of-core solutions greatly depends on the careful scheduling of those operations. Unfortunately, predicting such behaviour with an irregular, variable, and imbalanced workload is not trivial [2]. In short, graph processing is a prime example of an application where one would ideally stay within the memory space of a single node for as long as possible.

One possible solution to this challenge is the use of commodity Non-Volatile Random Access Memory (NVRAM) which has recently emerged on the market through Intel's Optane DC Persistent Memory Modules (DCPMM). Whilst this is slightly slower than DRAM, although much faster than disk, the significantly increased per-DIMM capacity compared to DDR4 DRAM, means that byte-addressable NVRAM can provide much larger RAM-style memory pools than DRAM alone. Furthermore, NVRAM can act as an extension to DRAM in a totally transparent manner, meaning applications can leverage this technology without any code changes required. Whilst DCPMM is Intel's specific NVRAM implementation, in this paper we use these terms interchangeably. Therefore for large-scale graph processing the use of NVRAM could be a game changer, enabling much larger graphs to be processed without incurring the performance overheads of moving inter-node or to out-of-core solutions. In this paper, we explore the use of Intel's DCPMM in the context of vertec-centric graph processing, to understand the role that this could play and the most appropriate techniques to obtain optimal performance. Our contributions are:

- Experimenting with and analysing the scalability of shared memory vertex-centric graph processing using NVRAM. Resulting in, as far as we are aware, a new world record for the size of graph processed within a single node without the use of out-of-core computation.
- Evaluating the need for manual tuning of existing codes to most efficiently exploit NVRAM.
- Quantifying the impact of NVRAM data placement based on the type of memory access performed.
- Discussing the price and power properties of using NVRAM for large-scale graph processing compared to alternate distributed approaches.

The rest of this paper is organised as follows: "Related work" presents related work and introduces the persistent memory technology and vertec-centric, before "Persistent memory modes" focuses on the Intel Optane DC Persistent Memory Module. "Experimental environment" describes the environment in which experiments are conducted, followed by "Results" which analyses the results obtained, before we draw conclusions and discuss potential future work directions in "Conclusion".

Related Work

Interest in non-volatile memory technologies has grown over the past decade [4], with hardware advances in the last two years now resulting in this technology becoming commodity and a realistic proposition for use in the data-centre and High-Performance Computing (HPC) machines. One such recent NVRAM technology is Intel's Optane DC Persistent Memory (DCPMM) [9]. Released in April 2019, in addition to featuring byte-addressability and non-volatility, the product is provided in a standard DRAM DIMM form-factor and at a significantly lower cost per byte than previous DRAM and NVRAM solutions. The byte-addressability means that the CPU can access any location in the DCPMM, effectively meaning that DCPMM has the ability to be used as either an extra storage disk, or an additional pool of RAM. The focus in this paper is most interested in the second benefit, where the between five and ten times increase in per-DIMM capacity when compared to DDR4 DRAM, results in the ability to provide a very large memory spaces. Therefore, whilst DCPMM is slower than DRAM, it makes possible the ability to equip nodes with an additional layer of memory hierarchy of TBs in size, at much lower energy and purchase cost than if this was all DRAM. DCPMM's read bandwidth is quoted as around 2.4 times lower than DDR4 DRAM and write bandwidth around 6 times lower than DDR4 DRAM [18], however this is still far faster than disk and can often be ameliorated in an application either by using the node's DRAM as an additional layer of cache (which is an automatic feature of the technology), or by the programmer explicitly controlling data placement.

Previous studies have been conducted around the use of NVRAM for a variety of applications, including [19] which specifically focuses on the use of DCPMM for scientific codes. An exciting result has been to show that applications which scale poorly in the distributed memory environment, can exploit NVRAM's large memory space to significantly increase the local problem size (effectively the data which can fit into a node's memory space) and ultimately improve performance. This has been highlighted as an important facet of the technology, extending the memory capacity of a node to enable applications most suited to shared memory

operation to reach a scale hitherto unobtainable with DRAM exclusively.

In recent years graph size has grown exponentially to reach today's scale which routinely involves hundreds of billions of edges, and even over a trillion for the largest graphs reported [14, 16, 20]. The amount of memory required to process such graphs increases with graph size and now stands at TBs for the largest graphs. The actual computation required for processing the graph is typically fairly low, with the codes themselves limited by the amount of memory that can be provided. Whilst the ideal is to stay within a single node, when processing the largest graphs the memory required is beyond what the vast majority of machines can hold or reasonably affordable to provide. Traditionally, there have been two possible approaches to tackle processing such large graphs. The first is that of distributed parallelism, often via MPI, but graph processing applications tend to be communication intensive, resulting in poor inter-node scaling. Furthermore, the high load-imbalance frequently found in graphs, especially within social networks, greatly increases the complexity of developing efficient distributed memory solutions. This was demonstrated in [12], where a 70 trillion edge graph (the overall world record for graph size in a distributed memory environment) was processed on 38,656 compute nodes of Sunway TaihuLight (with each node containing 260 CPU cores). This required over a million CPU cores to process the graph, and in their scaling experiments they highlighted that performance was limited by the bisection bandwidth due to static routing in InfiniBand, and the increasing volume of the graph cut in their distributed algorithm. The second possible solution is the use of out-of-core techniques, where significant chunks of a large graph are held on disk (typically SSDs) and the DRAM is effectively used as a cache managed explicitly by the programmer. However, this approach also tends to perform poorly, because of the relatively long latencies and low bandwidth of disk accesses.

This is where we believe NVRAM can be of great benefit for graph processing, enabling one to process much larger graphs within a single node before being forced to move to distributed memory, based on a technology which is hundreds of times faster than disk for access [19]. Furthermore, graph processing represents a workload with a highly irregular memory access pattern, which in itself is an important application pattern to explore within the context of how best one can leverage NVRAM most effectively, with lessons learnt applying more widely across other codes which also exhibit similar irregular memory access patterns. There have been a couple of previous studies of NVRAM with graph processing, for instance [8] where the authors compared a number of existing graph frameworks on NVRAM without optimising them specifically for this technology, and [6] where the authors developed their own placement algorithm exploiting the asymmetry between NVRAM read and write operation performance. Both these studies concluded that NVRAM is a worthwhile technology for graph processing, and in this paper we build upon the existing work, studying the use of NVRAM to hold much larger graphs (our largest graph contains 750 billion edges, compared to 128 billion and 225 billion edges respectively of these previous studies) and we undertake a more detailed exploration from the perspective of the programmer looking to obtain optimal performance from the DCPMM technology for their graph code.

Vertex-Centric and iPregel

The vertex-centric programming model is a popular approach for graph processing. This model, invented in 2010 by Google [13], is designed so that one can express graph computation from a vertex perspective. The design of the vertex-centric programming model is inspired from the Bulk Synchronous Parallel (BSP) model developed by Valiant [17]. This choice is motivated by the fact that BSP makes it "easier to reason about program semantics when implementing algorithms, and ensures that Pregel programs are inherently free of deadlocks and data races common in asynchronous systems" [13].

As a result, the vertex-centric execution flow consists of a sequence of iterations called supersteps, where the user-defined function containing the logic of a given algorithm is applied to vertices conceptually in parallel. Vertices can communicate via messages, thanks to the message-passing abstraction provided. Each vertex has a mailbox from which it can read messages that were emitted by their senders during the previous superstep. Similarly, vertices can send messages, which will be accessible by their recipient during the next superstep.

The application termination relies on the notion of active vertex. Every vertex begins the first superstep as active and has the ability to halt by calling a halt function from within the user-defined function. Upon halting, a vertex becomes inactive, and will stay so until it receives a message. At the beginning of a superstep, vertices that are inactive are skipped, and the user-defined function is applied to active vertices, conceptually in parallel. During execution, messages are sent and read as explained above, and inactive vertices having received a message are re-activated. When all vertices have been processed, every message transmitted and every inactive vertex re-activated where appropriate, the superstep ends, as illustrated in Fig. 1. If there is at least one active vertex, a new superstep begins, else the application terminates.

The vertex-centric programming model quickly gained traction due to the simplicity and programmer productivity of the interface it exposes. In the last decade, numerous vertex-centric frameworks have been implemented, and a range

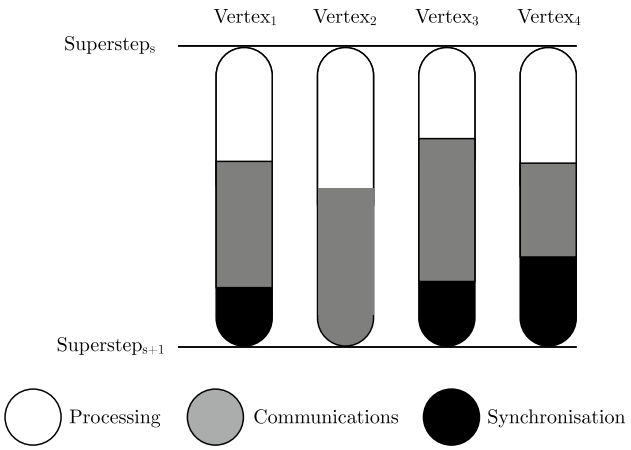

Fig. 1 A bulk-synchronous parallel superstep

of solutions that span the shared memory, distributed memory, and out-of-core techniques have grown popular. This has become a popular graph processing model employed by numerous organisations including web companies, and continued advances in the vertex-centric methodology mean that this is likely to become even more prevalent in the coming years.

One such framework is iPregel [1], which is arguably the most advanced shared memory vertex-centric framework, outperforming all other comparable technologies in its class both in terms of performance and memory usage. In [1] iPregel was tested against the distributed memory vertex-centric framework Pregel+ then state-of-the-art. iPregel proved to outperform Pregel+ by a median factor of 6.5, as well as demonstrating a higher memory efficiency by needing only 11GB of RAM to process a Twitter graph made of 2 billion edges while Pregel+ needed 109GB, and Giraph 264GB. iPregel is especially interesting here as it has been demonstrated to perform comparatively against other frameworks having sacrificed vertex-centric features or abstraction layers for performance [3]. This addresses the traditional disadvantage of vertex-centric where previously some performance was sacrificed for programmer productivity, while iPregel demonstrated that both performance and memory efficiency could be maintained while preserving the core advantage of vertex-centric: its programmability. Recent work with iPregel [2] focused on strategies to efficiently cope with the multiple challenges in vertex-centric programming such as load balancing or fine-grain synchronisations. Notably, the introduction of a novel hybrid combiner automatically switching from lock-based to lock-less execution permitting to go as far as quadrupling the performance of iPregel. This variety of techniques, aspects and problems investigated in these works mean that the lessons learnt in this paper can apply, not only to the vertex-centric model specifically but also more widely.

Persistent Memory Modes

In addition to the hardware itself, the Intel Optane DC persistent memory modules also ship with libraries that enable them to be used in different modes, offering different levels of granularity in the control of data placement. These modes can be activated by rebooting the node, which usually takes approximately 20 min, making it relatively easy to switch between DCPMM modes without hindering the overall throughput of jobs on a cluster.

Memory Mode

The first mode presented in this paper is referred to as memory mode, which is is very convenient due to it being entirely transparent to applications. This requires no application modification because the NVRAM provided by the DCPMM becomes the main memory space whist the DRAM effectively becomes the last level cache. By default, all allocations (both static and dynamic) take place on the DCPMM.

App-Direct Mode

The app-direct mode is the second mode presented in this paper and, unlike memory mode, does not provide automatic access to the DCPMM. Instead, existing DRAM remains main memory while the NVRAM can also be accessed by explicit load and store operations. Allocating memory on the DCPMM can be achieved by mounting a file system upon it, and using a special malloc interface from the *libvmem* library which is part of the Persistent Memory Development Kit (PMDK) [15].

DCPMM's app-direct mode can also be used in conjunction with the *libvmmalloc* library, which intercepts all dynamic allocations calls including *malloc*. These dynamic allocations, which without this library would have taken place in DRAM (as described in previous paragraph), will now take place within DCPMM. In other words, dynamic allocations now take place on the DCPMM while static allocation continue to be placed in DRAM. Furthermore, unlike the general app-direct mode and explicit use of PMDK, this mode does not require application rewriting other than the inclusion of the *libvmmalloc.h* header file.

Table 1 Hardware specification of a NextGenIO node

Metric	Value
Processor	2 × Xeon Platinum 8260M 24-core @ 2.4 GHz
Volatile memory (DRAM)	192 GB (12 × 16 GB)
Non-volatile memory (NVRAM)	3072 GB (12 × 256 GB)

Table 2 Graphs selected

Name	Number of vertices	Number of directed edges
S-250 / C-250	250,000,000	250,000,000,000
S-750 / C-750	750,000,000	750,000,000,000
Kronecker 25 500	33,554,432	33,554,432,000
Kronecker 28 500	268,435,456	268,435,456,000
Kronecker 33 16	8,589,934,592	274,877,906,944

Experimental Environment

This section describes the conditions in which our experiments were run, from the hardware and software used, to the graphs and application selected.

Hardware and Software

The experiments presented in this paper have been run on the cluster built as part of the NextGenIO project whose per-node specifications are given in Table 1. The cluster contains 34 identical nodes totalling over 100 TB of NVRAM and 6.5 TB of DRAM. Given the shared memory nature of the work presented in this paper, only one node of this cluster was used at any given time during the experiments discussed in "Results".

The iPregel framework was compiled with GCC 8.3.0 with support for OpenMP version 4.5 enabled. OpenMP threads are placed on physical cores and are pinned to them to prevent thread migration. Also, threads are placed on consecutive physical cores; meaning that the first 24 OpenMP threads are placed on the same NUMA region. The libraries *libvmem* and *libvmmalloc* have been used and these can be found in Intel's Persistent Memory Development Kit [15].

Graphs Selected

Table 2 lists the graphs that have been used in the experiments of "Results". Of the five graphs, three of them have been created using a Kronecker graph generator, where the name of these Kronecker graphs contains the parameters to reproduce them. The first number represents the logarithm base 2 of the number of vertices, and the second number the logarithm base 2 of the average out-degree. The graphs generated vary in sparsity, with an average degree of up to 500 which mimics those typically found in large social network graphs [14].

The two other graphs have been generated using our own graph generator, which provides a finer control over vertex adjacency lists. Each graph was generated in two forms; consecutive and scattered, where both versions comprise the same number of vertices, edges and degrees. The difference is in the locality of each vertex's neighbours, where the consecutive version results in the neighbour list of each vertex containing consecutive vertex identifiers. For instance, given a vertex i, in this configuration, its neighbours would be $i+1, i+2, ..., i+n$, where n is the number of neighbours. By contrast, the scattered version inserts a gap between any two consecutive neighbours, such that the identifiers of two consecutive neighbours are widely separated. For example, in this configuration given a vertex i, its neighbours would be $i+a, i+2a, ..., i+na$, with n the number of neighbours and a the gap length. These two configurations have been designed to represent extreme cases of memory locality, consecutive and non-consecutive access, which enables us to explore the impact of cache and page friendliness in the context of the NVRAM. These graphs are denoted as C-V for the consecutive version, and S-V for the scattered version, where V is the number of vertices in millions. The degree remains 500 in all cases and the gap a is set to 100,000 for the scattered versions. It can be seen that four out of our five graphs are larger than the largest experiments conducted in both previous studies of graph processing on NVRAM [8] and [6].

Application selected

The results presented in this paper are obtained by running ten iterations of the vertex-centric implementation of PageRank, whose code is illustrated in Fig. 2. PageRank is at the core of vertex-centric programming and has become a de-facto benchmark in the graph community. There are other commonly used graph processing applications, such as the Shortest Single-Source Path or the Connected Components, but PageRank provides a stable workload across iterations. Crucially for our purposes, this means that it minimises load-imbalance in the sense that every vertex participates towards the calculation at every superstep, whereas other graph applications deactivate vertices as the calculation progresses. This enables our experiments to remain focused on evaluating the performance of the NVRAM hardware and software, without being potentially biased by application or configuration specific logical behaviour. Moreover, PageRank places the most pressure on the memory subsystem as

```
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) {
    if(me->out_neighbour_count > 0) {
      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;
}
```

Fig. 2 PageRank implemented in iPregel

every vertex broadcasts a message to all its neighbours at every superstep. Therefore the number of messages emitted at every single superstep is equal to the total number of edges, placing a high degree of pressure on the memory system thus making it a challenging test of NVRAM performance. As such, whilst it might seem somewhat narrow to focus on one specific graph application only, no additional applications were selected in our experiments because PageRank inherently exposes the characteristics that most accurately explore the role of NVRAM, with the conclusions then applicable to a wide variety of other graph applications.

The experiments presented in this paper were conducted on two versions of the iPregel vertex-centric framework. These are *push* and *pull*, where the versions are alternative implementations of iPregel, triggering specific optimisations by redesigning certain part of the vertex-centric features and tuning them for specific situations. The push version of iPregel consists of each sender manually writing into the recipient's memory, where the thread that processes a vertex will write into the memory of each neighbouring vertex, typically held at random locations in memory. Of most interest to us, this version generates memory writes at multiple memory locations, in addition to the locks required to prevent potential data race. By contrast, the pull version consists in the recipient fetching messages from senders. The thread processing a vertex will therefore read from the memory locations of the sender vertices, before writing into the recipient vertex only. In addition to being lock-free, the pull version therefore generates writes that take place at a single memory location. These two versions thus make for two configurations that stress the memory in different ways and provide additional information to aid analysing the performance of NVRAM under pressure.

Results

This section presents and analyses the results collected during our experiments. We explore multiple data placement configurations to assess the different performance overheads related to the use of NVRAM, and by leveraging the memory modes presented in "Memory mode", we control the placement of vertices and edges.

Experiment 1: Staying in DRAM

The first experiment presented in this paper compares processing a graph which is stored exclusively in DRAM, and then exclusively in NVRAM. By comparing these runtimes we can then highlight any overhead imposed by the use of NVRAM.

To place a graph entirely in DRAM we use the app-direct mode presented in "App-direct mode", where dynamic allocations are by default placed on the DRAM. By contrast, to place the graph in NVRAM we use the *libvmmalloc* library (see "App-direct mode"), where dynamic allocations are automatically intercepted and their NVRAM-equivalent are instead issued.

The graph selected is the Kronecker 25 500; a graph small enough to fit within the 192GB of DRAM available on a single node. Nonetheless, it remains a graph that has over 30 billion edges which is larger than most graphs processed by shared memory frameworks or publicly available [7, 10].

The results gathered from this experiment are illustrated in Fig. 3, where it can be seen that there is always an overhead observable between the DRAM and NVRAM placement, as expected. Irrespective of the parallel configuration until 16 OpenMP threads included, NVRAM placement of the graph is approximately 2.5 times slower than its DRAM counterpart for pull version, and three times slower for the push version.

The performance exhibited by the NVRAM-only placement at 32 and 48 threads deviates from the scaling pattern seen with smaller thread counts and there is also a noticeable performance drop too. This is explained by how the NVRAM-only placement is implemented, where dynamic memory allocations are supplied from a memory pool built upon a memory-mapped file. This memory-mapped file can only be created on either the first or second socket, meaning that only the NUMA region local to that socket will be local to that memory pool. A socket contains 24 physical cores on this NextGenIO cluster and utilising fewer than 24

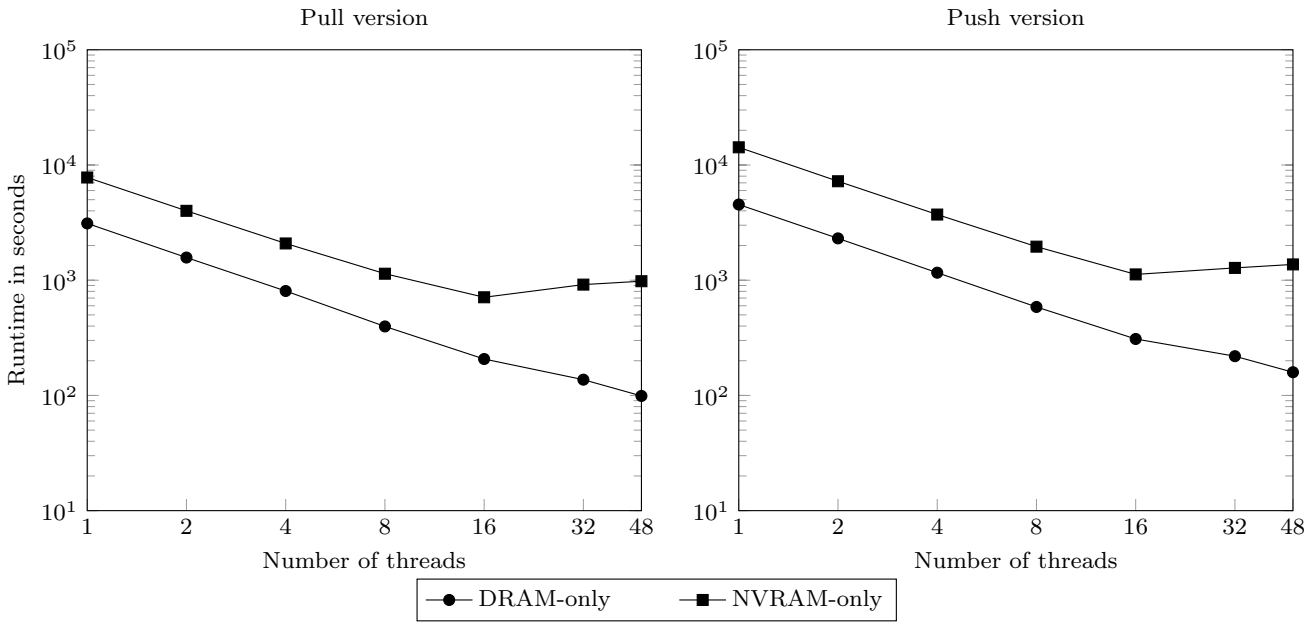

Fig. 3 Evolution of the iPregel runtime against the number of threads, for the Kronecker 25 500 graph using different graph memory placements

OpenMP threads, given the OpenMP placement configuration adopted, results in those OpenMP threads being pinned to physical cores on the socket which is local to the memory-mapped file. However, 32 and 48 OpenMP thread configurations result in threads also being mapped to physical cores of the other socket, and hence accessing the memory-mapped file in a cross-NUMA fashion which impacts performance. Until this cross-NUMA configuration is reached, we can see that the use of NVRAM memory mode does not hinder parallel scaling, either for the push version, or the pull version.

Experiment 2: Scaling Up Graphs

Unlike the experiment presented in "Experiment 1: staying in DRAM", our next experiment aimed to explore the use of DRAM and NVRAM working together. To achieve this, we use DCPMM's *memory mode* (see "Memory mode"), which automatically places data in NVRAM and uses DRAM as a last level cache.

Our three Kronecker graphs (see Table 2) are used for this experiment, designed to gradually increase the pressure on the non-volatile memory as their size grows. The first graph is the Kronecker 25 500, the 30 billion edge graph used in experiment one and small enough to fit entirely in DRAM. The second graph selected is Kronecker 28 500, and with 270 billion edges requires more than 5 times the memory available in DRAM. The third graph is Kronecker 33 16 which also contains approximately 270 billion edges, however, this graph holds 30 times more vertices; exceeding 2^{32}. Such a number of vertices requires vertex identifiers to be encoded using 64-bit integers instead of 32-bit, effectively doubling the amount of memory required to store the edges. Moreover, the number of edges per vertex on this last graph is 30 times lower than that on the Kronecker 28 500 where vertices are more densely interconnected, enabling us to evaluate the performance of the automatic cache feature on widely different graphs.

The timings collected during this experiment are reported in Fig. 4 and, as expected, runtime increases as the graph size grows. The Kronecker 25 500, which can fit entirely in DRAM, only requires a single movement of data from NVRAM to cache it in DRAM. The two other graphs, however, cannot fit in DRAM alone and therefore require multiple data movements between NVRAM and DRAM as data is evicted from this last level cache.

It can be observed that whilst both the Kronecker 33 16 and Kronecker 28 500 graphs contain a similar number of edges, processing of the former performs worse than the later. The crucial difference here is in the number of vertices, with Kronecker 33 16 containing 30 times more vertices than Kronecker 28 500. Storing the 280 million vertices of the Kronecker 28 500 graph requires approximately 10GB in iPregel, which can fit in DRAM. By contrast, storing 30 times more vertices consumes over 300 GB of memory, exceeding the total amount of DRAM available. As a result, not all vertices can be held in DRAM at once and as a superstep processes vertices must be evicted from DRAM to NVRAM.

This experiment has allowed us to evaluate the performance of using DRAM and NVRAM together. However,

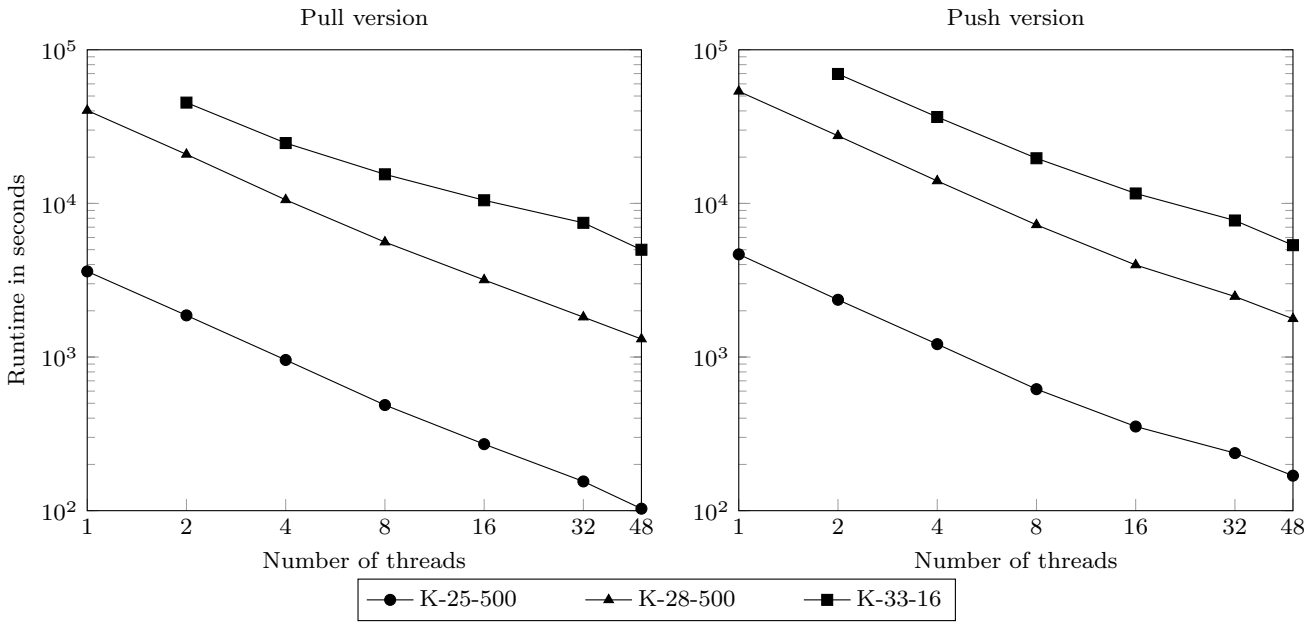

Fig. 4 Evolution of the iPregel runtime against the number of threads used, for different graph configurations, using NVRAM memory mode

whilst DCPMM's memory mode enables the use of a much larger memory pool without requiring application rewriting, the initial placement of all data on NVRAM regardless of their access pattern is likely sub-optimal. This can have serious implications for performance when edges evict vertices from the DRAM cache, since the NVRAM overhead of the latter is three to four times bigger.

Experiment 3: Read/Write Schism

The second experiment does not take into account the esoteric property of NVRAM, where the overhead involved in read and write operations is asymmetric (write operations being over twice as slow as read). Therefore, it was our hypothesis that to most optimally leverage NVRAM, one should understand these differences and tune their application accordingly.

To minimise the penalty of NVRAM's write overhead, data should be placed on DRAM or NVRAM based on its access pattern. It follows that DRAM should, therefore, be privileged for data that is read-write, while NVRAM should be ideally kept for read-only data. In the case of iPregel for instance, vertices are writable while edges are read-only. Therefore, this experiment relies on vertices being placed in DRAM and edges in NVRAM. Furthermore, since NVRAM modules are plugged to DRAM slots, they are subject to Non-Uniform Memory Access (NUMA) effects. Therefore, to maximise performance, the placement of edges on NVRAM should be NUMA-aware; placing edges on the NVRAM NUMA region corresponding to that of the vertex from which they are outgoing.

App-direct mode (see "App-direct mode") is used to manually place data on the DRAM or NVRAM, where a file system is mounted on each NVRAM NUMA region which is then accessed via PMDK's *libvmmem* library. The first step involves allocating a memory space on the NVRAM, from which a pointer is returned. Subsequently this pointer is then passed to a decorated set of functions equivalent to classic malloc functions, which perform the actual allocation on the NVRAM area pointed to.

Figure 5 reports the runtimes obtained by applying the read/write split technique on the Kronecker 25 500 graph (30 billion edges). We compare this runtime against those collected in previous experiments (DRAM-only, Memory mode, and NVRAM-only). The performance observed is bounded by the DRAM-only and NVRAM-only configurations, where for both the push and pull versions the DRAM-only configuration is the fastest, and NVRAM-only the slowest. It can be seen that the RW-split and memory mode experiments exhibit similar performance, with the memory mode approach being slightly slower in both cases. Therefore, having vertices pinned in DRAM, by contrast to the memory mode that may evict them and flush them back to NVRAM, helps the RW-split configuration to offer improved performance. It should be highlighted that the fact that this graph fits in DRAM does smooth the data movements performed by memory mode.

As expected, placing data while taking into the asymmetric overhead of NVRAM allows us to obtain a performance

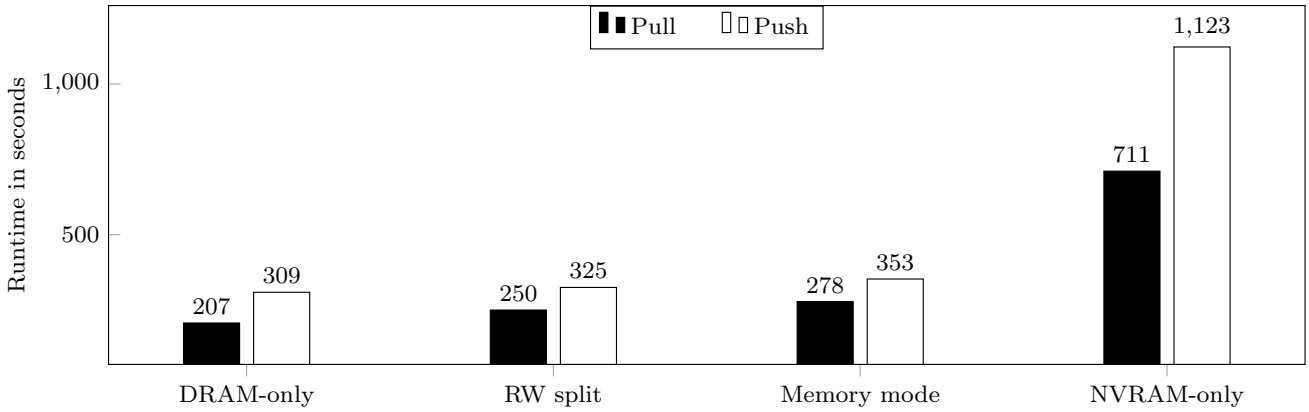

Fig. 5 Evolution of the iPregel runtime in seconds on the Kronecker 25 500, using multiple data placement configurations, for both push and pull versions at 16 threads

improvement. Nonetheless, the performance presented in the three experiments so far rely on Kronecker graphs made of random connections between vertices. As a result, the impact that memory locality has on performance must not be ignored.

Experiment 4: data locality and paging

To complement the experiments presented in "Experiment 1: staying in DRAM", "Experiment 2: scaling up graphs" and "Experiment 3: read/write schism", a fourth experiment was designed which evaluates the impact of memory locality. The experiment presented in this subsection involves two versions of each graph which share the same size, both in terms of vertices and edges but differ in terms of how they are connected. This is an important property as it influences the efficiency of caching during graph processing.

As described in "Experimental environment", two graphs have been designed, each with a contiguous and scattered version. Contiguous versions contain vertices whose neighbours are consecutive identifiers, whereas the scattered versions contain vertices with neighbours that are widely apart from each other. The NVRAM memory mode presented in "Memory mode" was selected so that its ability to automatically paging data between NVRAM and DRAM can be explored. Furthermore, the graphs generated consisting of 250 and 750 billion edges respectively, enabling us to evaluate the performance of NVRAM under significant memory pressure.

Figure 6 depicts the runtimes collected when processing synthetic graphs C/S-250 and C/S-750 respectively. Whilst, as would be expected, absolute runtime is less for the 250 billion edge graphs, scalability remains as high when tripling the graph size to 750 billion edges. The graphs with scattered memory access patterns exhibit poorer performance than those with consecutive accesses. This is no great surprise and is due to the extra paging operations required when recipient vertices are not already in DRAM, which is much more likely to occur on the scattered configurations. The overhead witnessed varies between 3.19 and 6.85 times, with 4.99 times being the average, depending on the graph size and iPregel version used, albeit then remaining constant as the number of threads increases.

The performance observed on NVRAM for these graphs is reported in Table 3 in terms of billion edges traversed per second (GTEPS). As observed above, scattered neighbours result in a significant performance overhead compared to the contiguous graph versions, and to our knowledge, the 750 billion edge graph runs are a new world record for the size of graph processed within a single node without the use of out-of-core computation. For comparison, PageRank with similarly sized synthetic large graphs when processed using the latest out-of-core solutions typically ranges between 0.07 and 0.83 GTEPS [12], and distributed GraM over 64 servers results in 8.6 GTEPS [12]. GraM was the fastest reported PageRank implementation for large graphs until the ShenTu implementation which significantly out performs all of these approaches by two to three orders of magnitude over 38,656 compute nodes. Whilst the performance of ShenTu is hugely impressive, realistically even for large companies such as Facebook or Google, such a specialist and expensive system is likely a difficult proposition.

Performance Summary

Across the four experimental configurations tested, certain performance patterns and overheads were observed. Table 3 reports the performance in billion edges traversed per second (GTEPS) observed in our experiments conducted in this section.

As expected, the highest performance is delivered for the contiguous version of the graph made of 250 billion edges

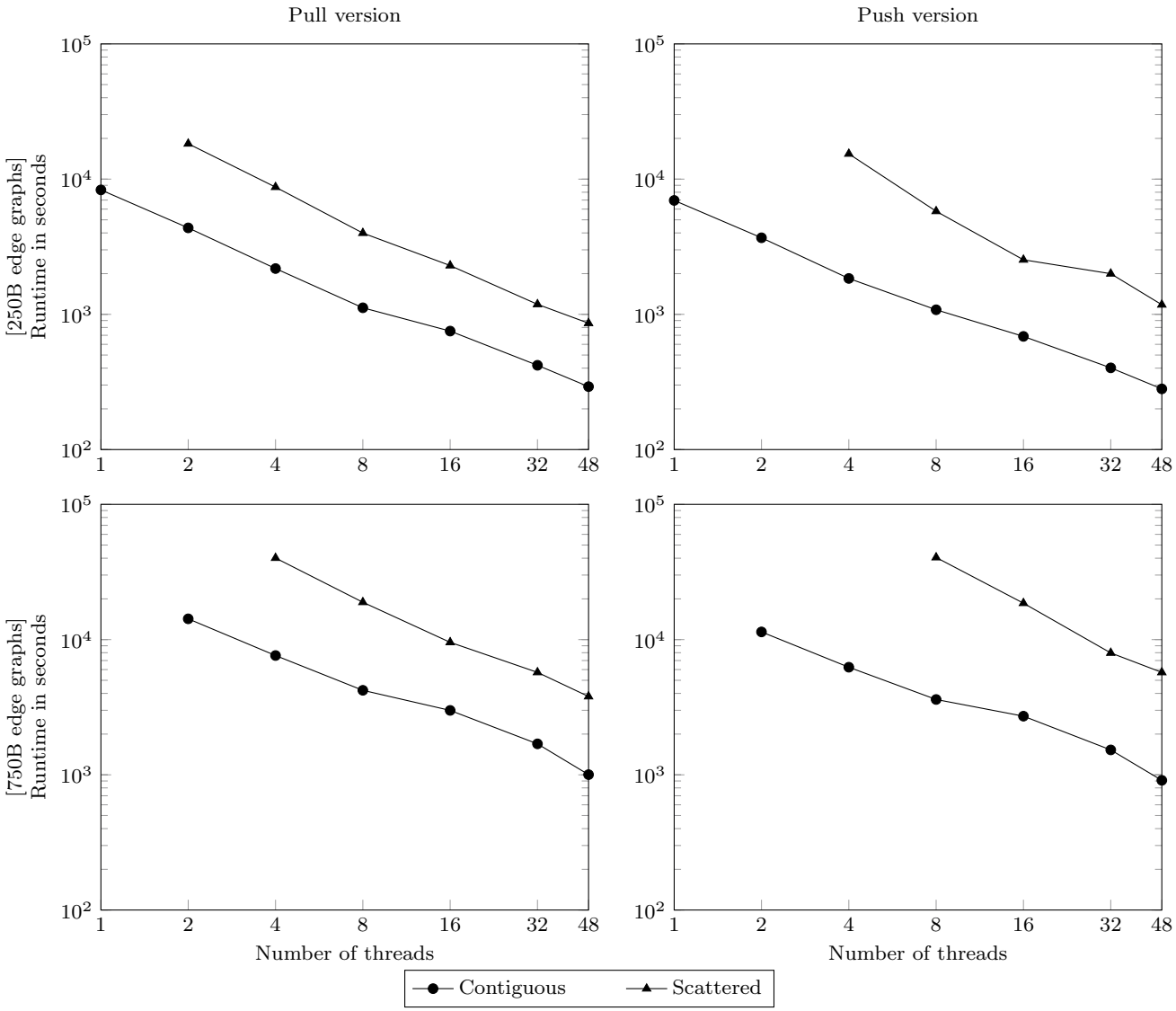

Fig. 6 Evolution of the iPregel runtime against the number of threads used, on the contiguous and scattered versions of the 250 and 750 billion edge graphs

Table 3 Maximum number of billions of edges traversed per second (GTEPS) by the pull and push versions of iPregel, on all graphs considered in this section, running over 48 threads

Graph	Pull version	Push version
C-250	10.12	9.56
S-250	3.26	2.13
C-750	8.61	8.63
S-750	2.35	1.35
Kronecker 25 500	3.35	2.10
Kronecker 28 500	2.31	1.55
Kronecker 33 16	0.61	0.54

(C-250). This is due to the optimal data-locality and load-balancing, and enables both iPregel versions to achieve approximately ten GTEPS. Between 85 and 90% of this performance was preserved when moving to the C-750 graph which contains three times as many edges, resulting in approximately 8.6 GTEPS for both the pull and push iPregel versions.

The performance observed on their scattered counterparts however is noticeably lower due to poor data locality. Performance achieved for the scattered graphs are 3.1 to 3.6 times lower than those seen for contiguous graphs with the pull version, and 4.5 to 6.4 times lower for the push version. This difference is explained by the access pattern paired with the asymmetric read-write overhead of NVRAM. The pull version of iPregel fetches broadcast messages

from neighbours (read) and combines them on the vertex being processed (write). Thus all writes are located on the vertex which is being processed by the executing thread. Conversely, with the push version of iPregel, when a thread processes a vertex then as messages are produced by that vertex these are immediately placed into the recipient's mailbox (or combined with an already present message). Therefore, writes are located on every neighbour of the vertex being processed, and therefore no longer local but are instead remote. Although both iPregel versions encounter scattered memory accesses when processing the S-250 and S-750 graphs, only for the push version does the location of write accesses change. Therefore this version of iPregel is more impacted by the write-specific additional overhead imposed by NVRAM.

When considering the Kronecker graphs the pattern of performance for each iPregel version is initially similar. When moving from the K 25 500 to the K 28 500 graph, the later comprising 8 times as many vertices and edges, this results in a performance reduction of approximately 30% for both iPregel versions. However, the performance reduces significantly, by approximately 70%, when moving from the K 28 500 to the K 33 16 graph. This performance decrease is explained by the increased number of vertices, over 8 billions, for the K 33 16 graph. Whilst such an increase enlarges the workload, the main issue is that at this scale a change in the type used to identify vertices is required. Eight billion is beyond what can be encoded by a 32-bit unsigned integer type and therefore this number of vertices requires the use of a 64-bit type instead. Whilst the consequence for the vertex data structure is fairly negligible, requiring only an additional 4 bytes per vertex structure, it is far more significant for the edges. Such an increase in data-type size results in each edge requiring double the amount of memory, resulting in longer loading times as well as worse data locality since the cache now contains 50% fewer edges.

It is also instructive to compare the performance for our approach on NVRAM reported in Table 3 with other popular graph processing frameworks which do not use NVRAM. Table 4 is reproduced from [12] and illustrates the performance achieved by these other graph processing frameworks with similar-sized graphs. Whilst the exact configuration of these graphs is not made explicit in [12], and likely a number of different configurations are used between frameworks, a number of broad comparisons can still be made. Firstly frameworks including G-Store, Graphene, and Mosaic utilise an out-of-core approach for processing these large graphs within a single node. Irrespective, this results in very poor performance which is typically significantly lower than all our performance figures for graph processing with NVRAM, apart from when we run the Kronecker 33 16 graph due to the issues highlighted in this section. Whilst each of these frameworks is utilising a server with SSDs rather than spinning hard disks, clearly the performance delivered by such hardware for large-scale graph processing falls significantly short of that delivered by NVRAM.

Table 4 Performance of other graph processing frameworks running with similar sized graphs to ours, data reported in [12] and reproduced here for comparison against NVRAM results

Framework	In-memory or out-of-core	Configuration	Performance (GTEPS)
Giraph	In-memory	200 nodes	5.6
GraM	In-memory	64 nodes	8.6
ShenTu	In-memory	1024 nodes	72.8
Chaos	Out-of-core	32 nodes(480GB SSD each)	0.07
G-Store	Out-of-core	1 node(8x512GB SSDs)	0.23
Graphene	Out-of-core	1 node(16x500GB SSDs)	0.83
Mosaic	Out-of-core	1 node(6 NVMe SSDs)	0.82

Whilst the parallelised in-memory frameworks Giraph and GraM deliver much greater performance than the out-of-core solutions in Table 3, our results demonstrate that a single node NVRAM approach to graph processing is still competitive for a number of graph types. Moreover, Giraph and GraM deliver only 0.028 GTEPS and 0.134 GTEPS per node respectively. Performance wise the stand out framework in Table 3 is that of ShenTu which, for a similar sized graph achieved 72.8 GTEPS over 1024 nodes. However, as we explore in "Additional metrics", this is equivalent to only 0.07 GTEPS per node.

Additional metrics

In addition to raw performance, other metrics such as purchase cost and energy efficiency are also important when one is considering a specific hardware solution such as NVRAM. A node from the NextGenIO cluster contains hardware totalling £20,000. Comparatively, taking the example of the Sunway TaihuLight supercomputer, 96 nodes are needed to obtain the same amount of memory. With the total cost of the 40,960-node supercomputer estimated at 273 million dollars [21], we estimate a per node price of 6665 dollars (approximately £5000). Processing our graphs which occupy 3072 GB of memory therefore either require a single £20,000 NextGenIO node, or 96 × £5000 Sunway TaihuLight nodes totalling £480,000. From a cost perspective alone there would be an approximate saving of £460,000 by adopting a non-volatile memory approach. Of course in such a scenario, the amount of computational processing power that can be leveraged by 96 nodes of the Sunway TaihuLight supercomputer far exceeds that of a single NextGenIO node, however as described in "Related work", typically vertex-centric graph processing is not computationally bound and

there is a communication overhead involved in a distributed memory approach.

Energy and power usage is another metric that has become increasingly important over recent years. The NextGenIO and Sunway TaihuLight nodes share the same power consumption (approximately 400 W), therefore in terms of power draw, which can be a major limit for data-centre machine rooms, NextGenIO will draw 96 times less power when at full load at any one point in time. It is also instructive to compare the overall energy consumption in terms of energy to solution, which also requires taking into account the execution time. The fact that NextGenIO and Sunway TaihuLight nodes share the same power consumption allows the calculations to be simplified to runtime alone. When processing one iteration of PageRank on Kronecker 34 16, which is twice as many vertices and edges of the Kronecker 33 16, the ShenTu framework using 1024 nodes from the Sunway TaihuLight supercomputer reaches 72.8 GTEPS, equivalent to 0.07 GTEPS per node. By contrast, when processing one iteration of PageRank on Kronecker 33 16, iPregel using a single node from the NextGenIO cluster reaches 0.60 GTEPS. This is 8.5 times more GTEPS for a graph twice as small. Unless the runtime grows as a quartic of the graph size, the NextGenIO node proves to be more efficient when considering the energy to solution.

Conclusion and Further Work

In this paper, we have discussed and analysed multiple experiments covering numerous scenarios in testing the performance of NVRAM for processing large-scale graphs. These experiments have been designed to evaluate the performance of NVRAM when used in isolation, or in conjunction with DRAM either implicitly or directly by the programmer. Two versions of the iPregel vertex-centric framework which exhibit different memory access patterns have been used as a vehicle to drive our experiments, with five different graphs including two specifically designed to provide fine control over data locality.

Whilst the focus of this paper has been on the vertex-centric methodology, and such a study is highly interesting in itself, we believe that the results and conclusions drawn can be more widely applied to both other graph processing technologies and codes with similar irregular memory access patterns. We found that NVRAM permits, without code rewriting, a shared memory framework such as iPregel to seamlessly scale to a graph of 750 billion edges, equivalent to 75% the Facebook graph [14]. To our knowledge, this is a new world record for the largest graph ever processed by a shared memory system without the use of out-of-core computation, and makes NVRAM a crucial enabler in reaching new horizons in shared memory graph processing within reasonable purchase cost and energy usage. Therefore for technology companies, such as Facebook and Google, who have large graph processing requirements and vast data centres, it is our opinion that NVRAM is a technology that should be considered a serious contender as having a role in their overarching hardware strategy. This is not least because such a step change in single-node graph processing capability can be delivered at a very reasonable price and power cost compared with other options.

We observed the impact of data placement and confirmed that, if one is willing to invest time in tuning their code for NVRAM, then manually placing data based upon the access pattern provides a better performance than that of automatic placement. Nonetheless, the multiple modes available to make use of NVRAM allow non-experts to leverage this technology without having to rewrite their application or their framework, and experts to rewrite parts of their software to make the most of this technology.

Moving forwards, we believe that in addition to being an enabler for shared memory graph processing systems, NVRAM could very well prove revolutionary in distributed memory graph processing systems too. In contrast to the Sunway TaihuLight supercomputer, where the full Sogou graph (270 billion vertices and 12 trillion edges) required at least 10,000 nodes [12], holding the Sogou graph in NVRAM memory could be achieved with fewer than 50 nodes based upon the NextGenIO cluster specifications. Furthermore, it was demonstrated in "Performance summary" that on a node by node basis the performance of existing distributed in-memory graph processing frameworks tends to be poor. As a next step, we believe it is important to explore how the emergence of clusters made of low-numbered large-memory nodes could also lead to the design of a new generation of distributed memory graph systems and algorithms. This will potentially deliver next generation extremely large graph processing performance by combining the single-node performance enabled by NVRAM with the memory capabilities of multi-node NVRAM.

Acknowledgements The authors would like to thank the Edinburgh Parallel Computing Centre (EPCC) for providing computing time on the NextGenIO cluster.

Funding This research is supported by the UK Engineering and Physical Sciences Research Council under grant number EP/L01503X/1, CDT in Pervasive Parallelism.

Data availability All data generated or analysed during this study are included in this published article. Requests for material should be made to the corresponding author.

Code availability The code used in this paper is publicly accessible at https://github.com/capellil/iPregel.

Declarations

Conflict of interest The authors have no relevant financial or non-financial interests to disclose.

References

1. Capelli LAR, Hu Z, Zakian TAK. ipregel: a combiner-based in-memory shared memory vertex-centric framework. In: Proceedings of the 47th international conference on parallel processing companion—ICPP '18; 2018.
2. Capelli LAR, Brown N, Bull JM. ipregel: strategies to deal with an extreme form of irregularity in vertex-centric graph processing. In: Proceedings of the the international conference for high performance computing, networking, storage, and analysis—SC '19; 2019.
3. Capelli LAR, Hu Z, Zakian TAK, Brown N, Bull JM. ipregel: vertex-centric programmability vs memory efficiency and performance, why choose? Parallel Comput. 2019;86:45–56.
4. Caulfield AM, Coburn J, Mollov T, De A, Akel A, He J, Jagatheesan A, Gupta RK, Snavely A, Swanson S. Understanding the impact of emerging non-volatile memories on high-performance, io-intensive computing. In: SC '10: proceedings of the 2010 ACM/IEEE international conference for high performance computing, networking, storage and analysis; 2010. p. 1–11.
5. Ching A, Edunov S, Kabiljo M, Logothetis D, Muthukrishnan S. One trillion edges: graph processing at Facebook-scale. Proc VLDB Endowment. 2015;8(12):1804–15.
6. Dhulipala L, McGuffey C, Kang H, Gu Y, Blelloch GE, Gibbons PB, Shun J. Semi-asymmetric parallel graph algorithms for nvrams. arXiv preprint arXiv:1910.12310; 2019.
7. For Discrete Mathematics TC (DIMACS) TCS. 9th DIMACS implementation challenge. http://www.dis.uniroma1.it/challenge9/download.shtml; 2006.
8. Gill G, Dathathri R, Hoang L, Peri R, Pingali K. Single machine graph analytics on massive datasets using Intel Optane DC persistent memory. arXiv preprint arXiv:1904.07162; 2019.
9. Intel announces broadest product portfolio for moving, storing and processing data. https://newsroom.intel.com/news-releases/intel-data-centric-launch/#gs.no8yic. Accessed 10 Dec 2020.
10. Kunegis J. Konect: the Koblenz network collection. In: Proceedings of the 22nd international conference on world wide web. WWW '13 companion. New York: ACM. . p. 1343–1350; 2013. https://doi.org/10.1145/2487788.2488173.
11. Kyrola A, Blelloch G, Guestrin C. Graphchi: large-scale graph computation on just a pc. In: Proceedings of the 10th USENIX conference on operating systems design and implementation. OSDI'12. Berkeley: USENIX Association. p. 31–46; 2012. http://dl.acm.org/citation.cfm?id=2387880.2387884.
12. Lin H, Zhu X, Yu B, Tang X, Xue W, Chen W, Zhang L, Hoefler T, Ma X, Liu X, Zheng W, Xu J. Shentu: processing multi-trillion edge graphs on millions of cores in seconds. In: Proceedings of the international conference for high performance computing, networking, storage, and analysis. SC '18. Piscataway: IEEE Press. p. 56:1–56:11; 2018.
13. Malewicz G, Austern MH, Bik AJ, Dehnert JC, Horn I, Leiser N, Czajkowski G. Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD international conference on management of data. SIGMOD '10. New York: ACM. p. 135–146; 2010.
14. Martella C, Logothetis D, Loukas A, Siganos G. Spinner: scalable graph partitioning in the cloud. In: 2017 IEEE 33rd international conference on data engineering (ICDE). p. 1083–1094; 2017.
15. Persistent memory development kit. https://pmem.io/pmdk/. Accessed 11 Dec 2020.
16. Tian Y, Balmin A, Corsten SA, Tatikonda S, McPherson J. From "think like a vertex'' to "think like a graph''. Proc VLDB Endow. 2013;7(3):193–204.
17. Valiant LG. A bridging model for parallel computation. Commun ACM. 1990;33(8):103–11.
18. Weiland M. Evaluation of Intel Optane DCPMM for memory and i/o intensive HPC applications. In: iXPUG workshop at HPC Asia 2020. https://www.ixpug.org/resources/download/michelewieland-hpcasia2020. Accessed 21 Dec 2020; 2020.
19. Weiland M, Brunst H, Quintino T, Johnson N, Iffrig O, Smart S, Herold C, Bonanni A, Jackson A, Parsons M. An early evaluation of Intel's Optane DC persistent memory module and its impact on high-performance scientific applications. In: Proceedings of the international conference for high performance computing, networking, storage and analysis. p. 1–19; 2019.
20. Wu M, Yang F, Xue J, Xiao W, Miao Y, Wei L, Lin H, Dai Y, Zhou L. Gram: scaling graph computation to the trillions. In: Proceedings of the sixth ACM symposium on cloud computing. SoCC '15. New York: ACM. p. 408–421; 2015.
21. Zhe G. Sunway taihulight: things you may not know about China's supercomputer. https://news.cgtn.com/news/3d517a4d324d444e/share_p.html (2017)