A Caching Approach to Reduce Communication in Graph Search Algorithms

Pietro Cicotti
San Diego Supercomputer Center
University of California, San Diego
pcicotti@sdsc.edu

Laura Carrington
San Diego Supercomputer Center
University of California, San Diego
lcarring@sdsc.edu

***Abstract*—In many scientific and computational domains, graphs are used to represent and analyze data. Such graphs often exhibit the characteristics of small-world networks: few high-degree vertexes connect many low-degree vertexes. Despite the randomness in a graph search, it is possible to capitalize on this characteristic and cache relevant information in high-degree vertexes. We applied this idea by caching remote vertex ids in a parallel breadth-first search implementation, and demonstrated 1.6x to 2.4x speedup over the reference implementation on 64 to 1024 cores. We proposed a system design in which resources are dedicated exclusively to caching, and shared among a set of nodes. Our evaluation demonstrates that this design has the potential to reduce communication and improve performance over large scale systems. Finally, we used a memcached system as the cache server finding that a generic protocol that does not match the usage semantics may hinder the potential performance improvements.**

Keywords—computer architecture, accelerator architectures; parallel processing, parallel algorithm; data system, data analysis.

I. INTRODUCTION

The exponential growth of data observed in recent years has fueled the interest in systems and methodologies to analyze large quantities of data. This new need for systems that can efficiently solve data-intensive problems has challenged common design practices and evaluation metrics.

By design, most HPC benchmarks represent classical HPC workloads and numerical methods. As an example, the LINPACK benchmark has been used since 1993 to rank the supercomputer in the Top500 list [1, 2]; the performance achieved in LINPACK has been used to compare the capability and performance of machines using this simple FLOPS-oriented metric. However, this class of benchmarks provides little insight into the capability of a system to perform data-intensive computations that characterize emerging workloads (e.g. cybersecurity, bio and medical informatics, social networks).

The Graph500 benchmarks address this gap by defining kernels that are a core part of analytics workloads [3, 4]. In this study we consider an algorithm and system design for the breadth-first search (BFS) kernel. The BFS kernel consists in traversing a graph and constructing a BFS tree rooted in a randomly chosen vertex. The graph is a scale-free graph randomly generated by a Kronecker generator [5]. The graph generated presents the characteristics of a *small-world network*, which is a type of graph in which most nodes are not adjacent, but many have a common adjacent vertex. Essentially, there is a small subset of high-degree vertexes (the degree of the vertexes follows a power law distribution). This kind of graphs occurs in many domains, and numerous examples can be found in applications including social networks, computer networks, and gene networks.

In this paper, we propose to use a caching mechanism to reduce the computation and communication in data-intensive computations. By taking advantage of the distribution of edges in the graph, we show that caching can significantly reduce communication, and consequently the computation required to process the exchanged information. Despite the inherent randomness in the data access pattern, the search repeatedly traverses edges that are incident on a small number of vertexes (the high-degree vertexes), which in the case of remotely stored vertexes is an operation that requires communication. Our solution involves caching the remote vertexes that have been previously visited, and checking the cache before initiating any communication. In addition, we propose to use a dedicated server for the caching, much as a co-processor shared between different computing nodes.

The primary contributions of this work are:

- a novel BFS implementation that reduces the cost of communication by using caching,
- an evaluation of the new algorithm,
- a proposed system design in which resources are dedicated to caching and shared among a localized set of nodes, and
- an evaluation of the proposed design and a discussion of the challenges encountered.

The paper is organized as follows. Section II provides a brief overview of related work. Section III describes the BFS algorithms that are evaluated in this study. Section III.C provides more details on the algorithms and all the variants

implemented. In Section V we present the results and compare the different variants. Finally, conclusions and future work are presented in Section VI.

II. Related Work

Previous research characterized the performance of the BFS kernel [6], and explored different optimizations. Several studies focused on the performance on shared-memory nodes, for example minimizing the memory footprint of frequently accessed data (e.g. using bitmaps) [7, 8], or reducing intra-socket communication [9]. Other work has been done on distributed BFS managed partitioning as a way to control load balance and communication [10, 11], or adopting sparse linear algebra representations to reduce the storage requirements [12].

Many studies targeted specific architectures, such as general purpose shared-memory architectures [7, 13-16], heavily multithreaded shared-memory architectures [17, 18], accelerators [19-23], and specialized processors [24-26]. In this work we propose a specialized system design although the approach is not specific to BFS and generalizes to other data-intensive algorithms that exhibit similar locality observed in searching small-world graphs.

Among other optimizations to the BFS algorithm, it has been proposed to change the search to find new vertexes adjacent to the frontier, rather than edges crossing the frontier, to take advantage of the performance characteristics of the search on small-world graphs and reduce the number of memory accesses [27]. Similarly, our solution leverages the characteristics of the computation on small-world graphs but reduces primarily the inter-node communication.

Our approach complements the optimizations proposed in previous research by providing a solution that is widely applicable in reducing the communication cost. By avoiding unnecessary communication, the proposed algorithm reduces the cost of communication, both in terms of performance and energy efficiency. Our idea is also applicable to other search algorithms and data-intensive workloads exhibiting similar locality properties. In addition, we propose to implement a system that couples the general purpose nodes in a system with reconfigurable resources to accelerate and improve the energy efficiency of data intensive workloads that exhibit similar locality properties observed in searching small-world graphs.

III. Vertex-Caching Algorithm

In this section we first describe a simple Breadth-First Search (BFS) algorithm [28], then we detail the parallel variant of the algorithm, and finally we cover the parallel variant that uses caching to reduce communication. In the latter, no assumption is made about the implementation of the caching mechanism.

A. Breadth-First Search Algorithm (BFS)

Given a graph *G*, represented by a set of vertexes *V* and a set of edges *E*, and given a source vertex *s*, a BFS algorithm searches G starting from s, trying to find all the vertexes that can be reached. At the same time, it computes the distance of each vertex *v* from *s*, defined as the fewest number of edges traversed to reach *v* from *s*. In practice, by tracking the previous vertex on the path for each vertex reached, a BFS also produces a tree rooted in *s* that contains all the reachable vertexes and their shortest paths.

Algorithm 1 illustrates a BFS formulation that computes the BFS tree for a given graph G, but that does not compute the distance from the source. The tree is stored as a *parent relationship*, where each vertex reached has a parent vertex, and the source *s* is the root (the root has itself as a parent). Starting from the source (line 2), a FIFO queue (Q) contains the vertexes discovered that may have uncovered adjacent vertexes. Then, as long as there are vertexes in the queue, a vertex *v* is removed from the queue (line 4), and then all its adjacent vertexes *w* that have not been reached yet (and therefore have no parent) are added to the queue (line 7); in addition, *v* is recorded in the *parent* vector for each of the newly reached vertexes (line 8).

The algorithm ends when there are no more vertexes in the queue. At that point, the *parent* vector contains a representation of the discovered BFS tree.

```
BFS(V,E,s)
1   parent[s]=s
2   Q=(s)
3   while(Q≠∅)
4       v=pop(Q)
5       for w ∈ {w|(v,w) ∈ E}
6           if(parent[w] is null)
7               Q=(Q,w)
8               parent[w]=v
```

Algorithm 1: Sequential BFS algorithm

B. Parallel Breadth-First Search (P-BFS)

The BFS algorithm presented in Sec. III.A can be parallelized with a message passing strategy, as shown in Algorithm 2. Initially, the set of vertexes *V* are randomly generated, distributed to processes, and stored in compressed Row Storage format (CSR); the *parent* vector is partitioned across all the processes according to the vertexes' distribution. Then, each process runs a slightly modified instance of the BFS algorithm where first, whenever a remote vertex is reached, the owner process is notified via a message that contains the vertex reached and its predecessor (potentially its parent, line 13). Second, the depth level of the BFS tree is explicitly marked via a global synchronization (line 19) to prevent a shorter path via remote vertex to be ignored in favor of a longer local path that is discovered first (this scenario can occur if the remote parent notification is late with respect to the discovery of the local path). The global synchronization ensures that the processes advance through depth levels of the tree synchronously avoiding the scenario described. The global synchronization also detects global quiescence (all queues are empty) and triggers termination (lines 19 and 3).

```
P-BFS(V,E,s)
1   parent[s]=s
2   Q=(s)
3   while(!alldone)
4       Q'=∅
5       while(Q≠∅)
6           v=pop(Q)
7           for w ∈ {w|(v,w) ∈ E}
8               if(is_local(w))
9                   if(parent[w] is null)
10                      Q'=(Q',w)
11                  parent[w]=v
12              else
13                  send (v,w) to owner(w)
14      Q=Q'
15      for (v,w) in received
16          if(parent[w] is null)
17              Q=(Q,v)
18              parent[w]=v
19      alldone = global_check(Q is ∅)
```

Algorithm 2: Parallel BFS

```
C-BFS(V,E,s)
1   parent[s]=s
2   Q=(s)
3   while(!alldone)
4       Q'=∅
5       while(Q≠∅)
6           v=pop(Q)
7           for w ∈ {w|(v,w) ∈ E}
8               if(is_local(w))
9                   if(parent[w] is null)
10                      Q'=(Q',w)
11                  parent[w]=v
12              else if(cache_miss(w))
13                  send (v,w) to owner(w)
14      Q=Q'
15      for (v,w) in received
16          if(parent[w] is null)
17              Q=(Q,v)
18              parent[w]=v
19      alldone = global_check(Q is ∅)
```

Algorithm 3: Caching BFS

C. *Vertex-Caching Algorithm (VC-BFS)*

In the parallel BFS algorithm, many messages may be sent to notify a process of different candidate parents for the same vertex, even from the same process. While the first message received may effectively add an edge to the BFS tree, all following messages are redundant and have no effect.

In addition, in the case of small-world graphs, the edges are not uniformly distributed across vertexes, but follow a power law distribution resulting in few popular vertexes, and many loosely connected vertexes. As a consequence, the majority of the messages are in fact redundant. Most vertexes will connect to a small set of vertexes, which with a great likelihood have already been reached.

To take advantage of this characteristic, we propose to *cache* remote vertexes that have been reached, and before sending a message check and update the cache. If the remote vertex is in the cache, the message is redundant and there is no need to send it, otherwise the cache is updated and the message is sent. As long as the hit rate is high, the communication traffic is reduced accordingly.

Algorithm 3 illustrates the small difference between the two algorithms, P-BFS and C-BFS. The added condition that determines whether or not a message is sent, which is emphasized in line 12, is dependent on whether the vertex *w* is present in the cache; otherwise, if *w* is not present the cache is updated with *w*.

IV. IMPLEMENTATION

In this section we describe the implementation of the reference, which is a parallel MPI implementation of Algorithm 2, and three variants that extend Algorithm 2 with caching, as described in III.C.

A. *Reference Implementation*

As a reference implementation, we used the MPI implementation made available by Graph 500 [4]. The reference implementation closely resembles Algorithm 2, except that communication is asynchronous, and edges sent to other processes are coalesced into fewer messages. When a remote vertex *v* is reached, the corresponding edge is added to the coalescing buffer designated to the owner of *v*; if the buffer reaches maximum capacity, the message is sent asynchronously. Incoming messages are often checked for the following: when an edge is added to a buffer, if a send to the corresponding owner is pending, and all pending send and receive operations are also checked for completion. In addition, before the global synchronization, all the pending send and receive operations are checked again to ensure no message is in transit before moving to the next depth level.

B. *Local Caching (LC-BFS)*

In the local caching variant, each process maintains a vertex cache in memory. The cache is implemented as a simple direct mapped cache and does not store any value, other than the vertex number (i.e. the vertex number is both tag and value). The cache is implemented using an array of configurable size, and directly maps a vertex *v* to an entry in the array. When checking for a vertex *v*, if the entry it maps

to contains the value *v,* then it is a hit, otherwise it is a miss and the current entry is replaced by *v*. While this implementation is very simplistic and could be improved by employing ad-hoc policies and heuristics, it is extremely fast and a check is completed in only a few instructions.

C. Remote Caching (RC-BFS)

In the remote caching variant, a different server process maintains the cache. Each MPI process connects via a TCP socket to a cache server, which can be shared, and queries the server to check whether a vertex is cached or not. Figure 1 shows a system configuration in which several nodes connected to the same switch (e.g. all the nodes of a rack) share a cache server. This design has several benefits: the cost of maintaining the cache is shared, for example the memory to allocate the cache is reduced by avoiding replicating the cache in each process and competing with the memory required to store the graph; the caching system may be optimized for that single task; the cache is populated sooner which would likely result in higher hit rates; and finally, by physically localizing communication, low-latency messages are exchanged in place of long-latency messages going across multiple hops in the interconnect (similarly the energy cost of communication should be reduced).

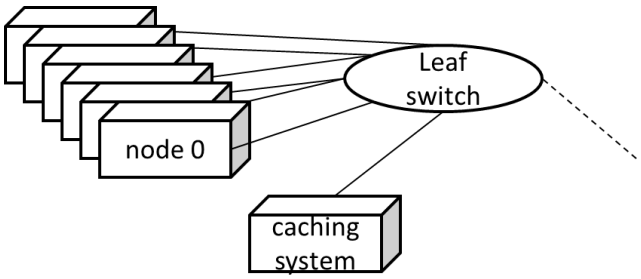

Figure 1: System configuration with remote caching.

The queries are packed in a message according to a simple protocol. Each message sent to the server contains a counter indicating the number of edges that follow, to immediately determine the size of the message, and a sequence of edges, as shown in Figure 2. On the server, the destination vertex of each edge received is checked against the cache (the cache is implemented as in the LC-BFS variant). For each message received, a response with the same format is sent back to the client: a counter precedes the list of edges received whose destination vertex missed in the cache (in case of no misses, a counter value of zero is sent as acknowledgment). Upon receipt of the response, a process scans the message and sends each edge contained to the owner of the destination vertex.

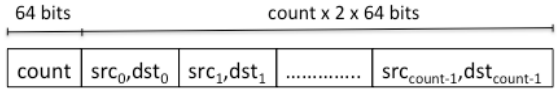

Figure 2: Message format. The counter field is a 64bit, and it is followed by pairs of vertex ids (each pair represents an edge). Vertex ids are also 64bit values.

D. Memcached Caching (MC-BFS)

Memcached is a distributed object caching system widely used to speedup database queries in web applications [29]. Memcached supports web applications that exhibit *temporal locality,* as in web searches, social networks, or media sharing services with significant *popularity/ viral* effects, and by providing quick responses to cached queries.

A memcached server maintains a key-value store typically accessed via TCP connections. The protocol is generic and supports keys and values of arbitrary size, and variants of basic *get* and *set* operations (plus other miscellaneous operations). In the context of this study, the memcached server is used in a somewhat unusual way because no value needs to be effectively retrieved, and the cache is used only to check the presence of a key.

To check for a vertex id, a GETKQ operation is issued with the vertex id as the key. In response, the server sends both the value and key if the key is present, or nothing if the key is not present. Using the quiet version of the operation (notice the Q in the GETKQ) helps reducing traffic avoiding failure responses. The requests are then compared to the responses received, and each get failed (no response) is treated as a miss. A miss triggers an additional SETQ request to add the missing vertex id to the cache. A SETQ always succeeds, and using the quiet version of the operation avoids a response.

Since the server processes all the requests as a stream, multiple operations are coalesced within the same message to minimize the number of round-trips (and therefore minimize the latency of each operation). Each sequence of requests is terminated by a NOOP, to ensure that a response is produced even in the case that all the *quiet* GETKQ operations trigger no response. The acknowledgement to the NOOP also signals the end of the response message. If only a NOOP acknowledgement is received, then all the requests missed, and all the vertex ids are added to the cache.

Each SETQ operation has a fixed-size header of 24 bytes, followed by a 24-byte body containing the key, the value, and an unused mandatory field. The header identifies the type of operation, and the size of the key and the value, which in this case are both 64 bits. The GETKQ operation has a 24 bytes header followed by the key (vertex id) to be retrieved. The reply from the server, which is only generated in the case of a hit, contains a response for each operation. The reply has a fixed-size header, a 4B mandatory field, which is ignored, and the retrieved key-value pair. Notice that for our purpose the key (hence the use of the K variant of GET) is used to signal which vertex id was a hit in the cache. The value, however, is irrelevant and it is ignored (the source vertex id was used as value to simplify validation and debugging). The format of all the operations used is illustrated in Figure 3.

We also developed an alternative implementation that uses the *ADD* operation and limits each request to a single round-trip. The ADD operation, with the destination vertex as the key and the source vertex as the value, succeeds if the key is not already present: the server responds to the *ADD*

operation with a success or a failure notification, which is then interpreted as a miss (the key was not present) or a hit (the key was already present). While this implementation requires fewer messages, it performed poorly and was discarded in favor of the GET/SET implementation. There are two reasons for this poor performance: first, as long as there is sufficient reuse of the data in cache the GET/SET implementation rarely requires two round trips, and second the ADD operation is always more expensive than the GET (apparently the ADD involves locking internal data structures regardless of whether the ADD fails or not).

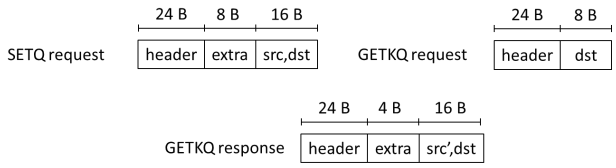

Figure 3: Format of messages, operations and responses.

V. Evaluation

In this section we evaluate the different variants presented. For the evaluation we used two systems, a large system, and a 4-node cluster and a dedicated memcached appliance.

The large system is Stampede, a cluster hosted at the Texas Advanced Computing Center (TACC) [30, 31]. The compute nodes have two 8-core Intel Xeon processors (Sandy Bridge E5-2680), running at 2.7 GHz, and 32GB of DRAM (DDR3-1600). The 6400 nodes of the Stampede are connected via FDR Infiniband, in an oversubscribed 2-level fat-tree topology. Stampede also features Xeon Phi cards, a feature we did not use for this study.

The second test system that we used for the experiments is a 4-node cluster and a memcached system. Each node has two 4-core Intel Xeon processors (Nehalem E-5530), running at 2.4 GHz, and 24GB of DRAM (DDR3-1066). The nodes are connected to a 1Gb Ethernet switch.

The memcached system is a Convey HC2ex [32]. The Convey system is a host+co-processor system featuring 4 FPGAs. The host has two Intel Xeon processors (E5-2663) running at 2.6GHz, and 128GB of DRAM (DDR3-1600); the coprocessor has four FPGA and 32GB of DRAM (SG-DIMM). The FPGAs are programmed using *personalities* that are highly specialized configurations for specific workloads. In this this study, the FPGAs are programmed with the memcached personality [33].

A. Local Caching

First we compare LC-BFS to the reference. We ran both LC-BFS and a reference implementation on Stampede, using from 64 cores and up to 1024 cores, weakly scaling a graph with a ratio of 2^{20} vertexes per core (scale 26 to 30, edge factor of 16), a ratio that is comparable to the that found in several top results on the Graph 500 list. Once a graph is generated, the search is repeated 64 times for different randomly chosen roots and the timing is averaged.

Figure 4 shows the comparison between the reference implementation, labelled no-cache, with the LC-BFS variant and different cache sizes. First, the results show a significant loss of parallel efficiency, which is not unexpected for the workload, and that is a consequence of the increased communication and latency (the distance between nodes also increases) and the congestion generated on the interconnect. The LC-BFS variants are consistently faster, with an average speedup of 2.3 for the cache size of 16M vertexes (128MB).

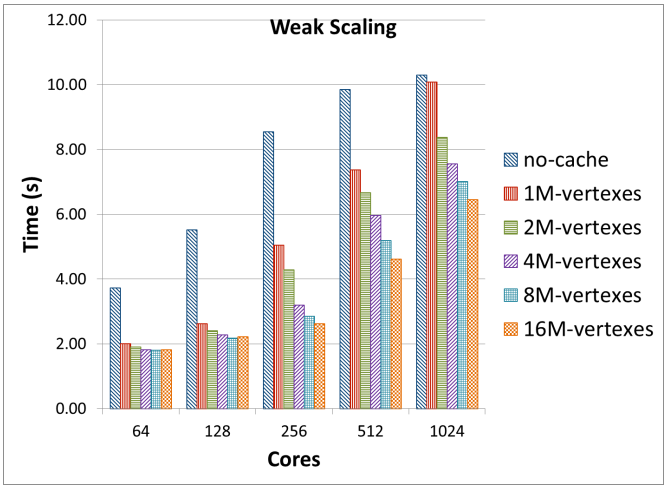

Figure 4: Weak Scaling at 2^{20} vertexes per core and edge factor 16.

In addition the figure illustrates how the larger the cache is the larger the speedup; however, it is apparent that as the problem size grows, the cache needs to be scaled to preserve the performance benefit. On 1024 cores and 2^{30} vertexes, many high-degree vertexes are not found in the local cache. In such cases the performance benefit may be very little. Nevertheless, even the 2M cache achieves a 1.5 speedup on 1024 cores, indicating that the performance benefit increases rapidly with the cache size.

Table 1: Cache hit rates and message statistics. The benchmarks runs on 1024 cores, scale 30, and edge factor 16. The column *repeated vertexes* indicates how may vertexes received already have a parent in the BFS tree. The column *normalized MPI messages* shows the reduction in total messages exchanged between MPI processes.

Cache Size (vertexes)	Cache Size (MB)	Cache Hit Rate	Repeated Vertexes (%)	Normalized MPI Messages	Speedup
0	0	0.0	98.7	1.00	1.00
2^{20}	8	22.3	98.3	0.78	1.02
2^{21}	16	29.7	98.2	0.71	1.23
2^{22}	32	37.4	97.9	0.63	1.36
2^{23}	64	44.2	97.7	0.57	1.47
2^{24}	128	49.2	97.4	0.52	1.60

Table 1 shows the hit rate on the local cache, and the resulting reduction in repeated vertexes and MPI messages on 1024 cores. Surprisingly, there are a large number of unnecessary vertexes sent even with caching. Since each process has its own cache, and because the cache hit rate is relatively low, several vertexes are still visited multiple times (e.g. a vertex has been visited either by the owner process or by another remote process). However, the overall number of MPI messages is significantly reduced leading to the

performance improvements observed. The metric on the normalized MPI messages shows the number of MPI messages in LC-BFS normalized with respect to the reference.

We also compared LC-BFS to the reference in the case of strong scaling. For these experiments we ran from 128 cores up to 1024 cores, strong scaling a graph with 2^{28} vertexes (scale 28, edge factor of 16). As before, the search is repeated 64 times for different randomly chosen roots and the timing is averaged.

Even in this case the caching implementation is significantly faster than the reference, with an average speedup ranging from 1.6 to 2.5 for cache size respectively from 1M vertexes to 16M vertexes. Nevertheless, the reference implementation scales better.

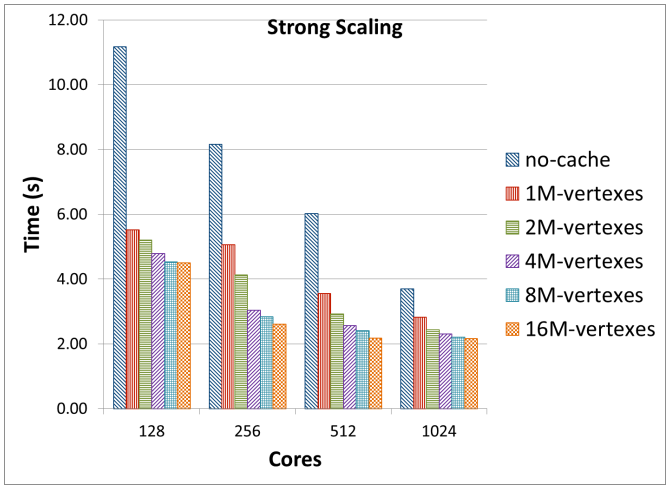

Figure 5: Strong scaling at 2^{28} vertexes and edge factor 16.

Table 2 lists the speedup achieved by the reference and by LC-BFS with different cache sizes. The reference (column N-C) improves more than the caching variants as core count increases. The caching variant seem to reach a performance plateau once it achieves a 2.1 speedup relative to 128 cores, as observed for the 16M-vertexes cache configuration (16M-V).

Table 2: Relative speedup to 128 cores on a graph of class 28 and edge factor 16.

Cores	N-C	1M-V	2M-V	4M-V	8M-V	16M-V
256	1.4	1.1	1.3	1.6	1.6	1.7
512	1.9	1.6	1.8	1.9	1.9	2.1
1024	3.0	2.0	2.1	2.1	2.1	2.1

Strong scaling shows that there are two competing effects: the local subset of vertexes continues to shrink and the amount of work per process continues to shrink, but at the same time the cache is less effective as there are fewer opportunities to fill the cache and successively avoid communication. Table 3 illustrates this effect in the cache hit rates and communication statistics in that the cache hit rate decreases, and so does the effect on communication when the number of cores increases, as indicated by the fact that the normalized MPI messages count increases. Despite this effect of diminishing return, the caching variant is significantly faster than the reference in every case, and even when using fewer cores; as an example, with a cache of 16M vertexes on 256 cores it is 1.4x faster than the reference on 1024 cores.

Table 3: Cache hit rates and message statistics. The benchmarks runs scaling from 128 cores to 1024 cores, with a graph of scale 28, and edge factor 16. The column repeated vertexes indicates how may vertexes received already have a parent in the BFS tree. The column normalized MPI messages shows the reduction in total messages exchanged between MPI processes. The speedup is relative to the reference (no-cache).

Cores	Cache Hit Rate	Repeated Vertexes (%)	Normalized MPI Messages	Speedup
128	69.4	95.4	0.31	2.5
256	64.5	96.0	0.36	3.1
512	58.0	96.7	0.43	2.8
1024	50.4	97.2	0.53	1.7

B. *Remote Caching*

The second set of experiments aims at evaluating a different system design in which a dedicated server maintains the cache. In this case, checking the cache involves communication with the server.

We compare 5 different configurations on the 4-node test cluster: the reference implementation, the local caching variant (LC-BFS) with a cache size of 8M vertexes per process, the remote caching variant (RC-BFS) with a cache size of 8M vertexes, and a memcached variant with a cache size of 128MB; the latter is run against both the standard software memcached distribution (MC-BFS) and the accelerated memcached (CY-BFS) that uses the HC2ex coprocessor.

In this evaluation, a graph of scale 23 is weakly scaled from 8 cores to 32 cores, with a ratio of 2^{20} vertexes per core. Figure 6 shows the comparison between the reference implementation, LC-BFS, and RC-BFS. On 8 cores, for the reference implementation all the communication is intra-node, whereas with remote caching an equivalent number of messages are directed to the cache. The latency gap between intra-node and inter-node communication is evidently reflected in the performance gap between the reference and RC-BFS. LC-BFS improves on the reference but only marginally, on 8 cores, because the cost of communication is relatively low. However, as messages cross the node boundaries the increase in latency is directly reflected in the performance of the reference. On 16 cores LC-BFS is 3.8x faster, and on 32 cores is 4.1x faster than the reference. RC-BFS is slower than the reference, although the gap reduces rapidly as the portion of off-node communication increases. The more cores that are utilized, the more messages are directed off node and the overhead of accessing the cache is compensated by the reduction in MPI messages. Scaling further to multi-level network topologies, the latency of the MPI messages increases, due to the increase in distance between nodes, and in that case the cost of communication could be much higher than the cost of accessing the cache; at large scale, the performance advantage in reducing global communication with RC-BFS should outweigh the overhead of accessing the cache.

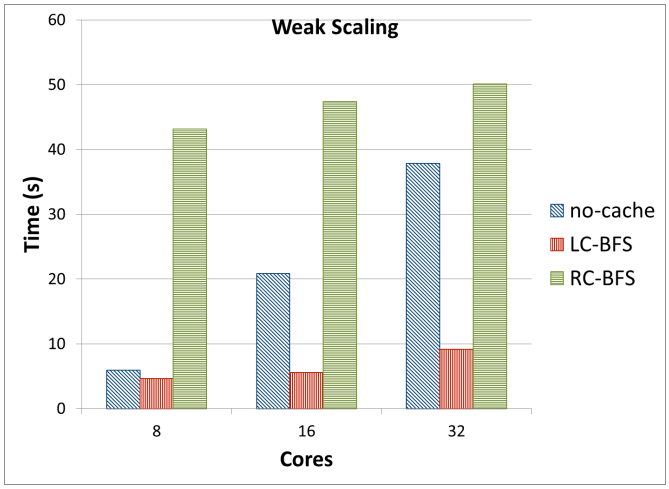

Figure 6: Weak scaling comparison between the reference (no-cache), local caching with caches size of 8M vertexes (LC-BFS), and remote caching with a cache size of 8M vertexes (RC-BFS). The graph has 2^{20} vertexes per core and edge factor 16.

Table 4 further supports this hypothesis by showing that remote caching achieves greater reduction in communication than local caching. By sharing the cache, each process avoids communication by also considering the vertexes discovered by other processes. In addition, as the problem size grows the local cache is less effective in reducing communication, because the chances of traversing a vertex visited by another process increases, but that is not the case for a shared cache.

Table 4: Communication reduction in local and remote caching.

Cores	Normalized MPI Messages in LC-BFS	Normalized MPI Messages in RC-BFS
8	0.07	0.02
16	0.12	0.02
32	0.18	0.02

Using memcached on the server resulted in a performance degradation. The tests with the RC-BFS implementation exposed some performance loss due to the communication with the server. In RC-BFS, the same number of MPI messages that the reference would generate are directed to the server, while the MPI messages are greatly reduced. The overall number of messages is not reduced, and performance gains are possible only if the latency of a query to the server is less than the latency of a point-to-point MPI message. This is not the case on the test cluster, and in fact, a large fraction of the MPI messages are intra-node and the latency is even smaller than that of communicating with the server. Nevertheless, the communication protocol is simple enough that the performance loss observed is contained within 32% on 32 cores. However, with memcached there are several issues that hinder performance. The protocol is much more general than the simple protocol used in RC-BFS, adding complexity to the otherwise minimal implementation of BFS, and that costs some processing time both on the computing processes and the server. More importantly, the protocol does not include operations that match usage model well like: checking whether a key is present involves one or two round-trip messages, and additional processing of the response messages.

The result is that memcached becomes a bottleneck and the performance observed is much worse than that of the reference (up to 2 orders of magnitude for the 32 cores test). We also observed slight differences in the effectiveness in reducing communication between the standard and the accelerated version, likely due to differences in allocating and using the space specified as the cache size. The difference in communication reduction penalizes the accelerated version that on 8 cores is slower than the standard version. Compared to the standard memcached, the accelerated version achieves a 2.5x speedup on 8 cores, and a 1.2x speedup on 16 and 32 cores. Table 5 shows the speedup and the normalized MPI messages for the remote caching variants. As mentioned, differences in the way the cache capacity is utilized are reflected in less reduction in the communication. In memcached, 128MB are used to store the keys, values, and additional extra data. Since both keys and values are 8B ids, the cache can contain less than 8M vertexes, which is the capacity of the cache used in RC-BFS.

Table 5: Communication reduction in remote caching, and with memcached.

Cores	Normalized MPI Messages in RC-BFS	Normalized MPI Messages in MC-BFS	Normalized MPI Messages in CY-BFS	Speedup of CY-BFS over MC-BFS
8	0.02	0.11	0.11	2.5
16	0.02	0.19	0.19	1.2
32	0.02	0.29	0.29	1.2

VI. Conclusions and Future Work

Often, data-intensive applications access data randomly but with some inherent locality. This is the case, for example, in searches on small-world graphs, in which a few high-degree vertexes are frequently traversed. This paper shows that it is possible to leverage this characteristic by caching information about the frequently accessed vertexes and avoid communication, and perhaps also computation.

In this study we implemented this approach in a BFS benchmark used by Graph 500 to rank systems by their ability to execute data-intensive workloads. By caching high-degree vertexes, a parallel BFS implementation can avoid the communication cost otherwise incurred when sending messages that are disregarded by the receiving process.

Results with an in-memory cache implementation have shown great performance improvement when compared to the reference implementation. Both in weak scaling and strong scaling, from 64 cores to 1024 cores, the speedup observed ranges from 1.6x to 2.4x for a cache of 16M vertexes in capacity. Message counters embedded in the code also demonstrate that the number of MPI messages is greatly reduces, with almost a 50% reduction on 1024 cores.

We have shown that a remote-caching system has several advantages over an in-memory implementation, including a much higher hit rate and potentially can be more effective in reducing communication. However, remote caching does not

reduce the overall number of messages, and the overhead of accessing a remote caching system needs to be compensated by reducing more expensive point-to-point communication; this would be the case in large scale systems. In this case, the nodes within a building block (e.g. a rack) would share a cache, and avoid expensive communication across the entire system. In addition, the shared caching may become a bottleneck is servicing a large number of neighbors; how to find a suitable ratio of caching systems per nodes should be investigated.

Using existing caching systems may be challenging. General protocols whose operations do not match the semantics of the intended usage lead to an inefficient implementation and hinder performance. As a consequence, even accelerated implementations that can sustain high throughput workloads suffer a performance penalty when compared to simple ad-hoc caching implementations.

Future work will focus on validating our results with remote caching at scale, and on different search algorithms. In addition, we will address the inefficiencies of the protocol by implementing ad-hoc extensions to memcached and the corresponding personality for the co-processor.

ACKNOWLEDGMENT

This work was supported by the UC Laboratory Fees research program and used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1053575. This work used resources generously provided by Convey Computer.

REFERENCES

[1] "Top 500," www.top500.org.
[2] J. J. Dongarra, and H. D. Simon, *High Performance Computing in the U.S. 1995- An Analysis on the Basis of the TOP500 List*, University of Tennessee, 1995.
[3] R. C. Murphy, K. B. Wheeler, B. W. Barret, and J. A. Ang, "Introducing the Graph 500."
[4] "Graph 500," www.graph500.org.
[5] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker Graphs: An Approach to Modeling Networks," *J. Mach. Learn. Res.,* vol. 11, pp. 985-1042, 2010.
[6] T. Suzumura, K. Ueno, H. Sato, K. Fujisawa, and S. Matsuoka, "Performance characteristics of Graph500 on large-scale distributed environment," in Proceedings of the 2011 IEEE International Symposium on Workload Characterization, 2011, pp. 149-158.
[7] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader, "Scalable Graph Exploration on Multicore Processors," in Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, 2010, pp. 1-11.
[8] F. Checconi, F. Petrini, J. Willcock, A. Lumsdaine, A. R. Choudhury, and Y. Sabharwal, "Breaking the speed and scalability barriers for graph exploration on distributed-memory machines," in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, Salt Lake City, Utah, 2012, pp. 1-12.
[9] Z. Cui, L. Chen, M. Chen, Y. Bao, Y. Huang, and H. Lv, "Evaluation and Optimization of Breadth-First Search on NUMA Cluster," in Proceedings of the 2012 IEEE International Conference on Cluster Computing, 2012, pp. 438-448.
[10] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek, "A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L," in Proceedings of the 2005 ACM/IEEE conference on Supercomputing, 2005, pp. 25.
[11] E. Chow, K. Henderson, and A. Yoo, *Distributed breadth-First Search with 2-D Partitioning*, LLNL, 2005.
[12] J. Gilbert, S. Reinhardt, and V. Shah, "High-Performance Graph Algorithms from Parallel Sparse Matrices," *Applied Parallel Computing. State of the Art in Scientific Computing*, Lecture Notes in Computer Science B. Kågström, E. Elmroth, J. Dongarra and J. Waśniewski, eds., pp. 260-269: Springer Berlin Heidelberg, 2007.
[13] Y. Xia, and V. K. Prasanna, "Topologically Adaptive Parallel Breadth-First Search on Multicore Processors."
[14] G. Cong, and D. A. Bader, "Designing irregular parallel algorithms with mutual exclusion and lock-free protocols," *J. Parallel Distrib. Comput.,* vol. 66, no. 6, pp. 854-866, 2006.
[15] G. Cong, G. Almasi, and V. Saraswat, "Fast PGAS Implementation of Distributed Graph Algorithms," in Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, 2010, pp. 1-11.
[16] D. A. Bader, G. Cong, and J. Feo, "On the Architectural Requirements for Efficient Execution of Graph Algorithms," in Proceedings of the 2005 International Conference on Parallel Processing, 2005, pp. 547-556.
[17] D. Mizell, and K. Maschhoff, "Early experiences with large-scale Cray XMT systems," in Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing, 2009, pp. 1-9.
[18] D. A. Bader, and K. Madduri, "Designing Multithreaded Algorithms for Breadth-First Search and st-connectivity on the Cray MTA-2," in Proceedings of the 2006 International Conference on Parallel Processing, 2006, pp. 523-530.
[19] D. P. Scarpazza, O. Villa, and F. Petrini, "Efficient Breadth-First Search on the Cell/BE Processor," *IEEE Trans. Parallel Distrib. Syst.,* vol. 19, no. 10, pp. 1381-1395, 2008.
[20] E. Saule, and Ü. V. Catalyurek, "An Early Evaluation of the Scalability of Graph Algorithms on the Intel MIC Architecture," in Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, 2012, pp. 1629-1639.
[21] L. Luo, M. Wong, and W.-m. Hwu, "An effective GPU implementation of breadth-first search," in Proceedings of the 47th Design Automation Conference, Anaheim, California, 2010, pp. 52-55.
[22] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient Parallel Graph Exploration on Multi-Core CPU and GPU," in Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques, 2011, pp. 78-88.
[23] P. Harish, and P. J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in Proceedings of the 14th international conference on High performance computing, Goa, India, 2007, pp. 197-208.
[24] O. Mencer, Z. Huang, and L. Huelsbergen, "HAGAR: Efficient Multi-context Graph Processors," in Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications, 2002, pp. 915-924.
[25] M. deLorimier, N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T. E. Uribe, T. F. J. Knight, and A. DeHon, "GraphStep: A System Architecture for Sparse-Graph Algorithms," in Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2006, pp. 143-151.
[26] C. Computer, "Convey Computer Announces Graph 500 Performance," 2011.
[27] S. Beamer, K. Asanovic, and D. Patterson, "Direction-optimizing breadth-first search," in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, Salt Lake City, Utah, 2012, pp. 1-10.
[28] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*: MIT Press, 2001.
[29] B. Fitzpatrick, "Distributed caching with memcached," *Linux J.,* vol. 2004, no. 124, pp. 5, 2004.
[30] "Texas Advanced Computing Center," https://www.tacc.utexas.edu/.
[31] TACC, "Stampede, Dell PowerEdge C8220 Cluster with Intel Xeon Phi coprocessors."
[32] C. Computer, "The HC series."
[33] C. Computer, "Convey launches Hybrid-Core Memcached Appliance," 2013.