Transparent Caching for RMA Systems

Salvatore Di Girolamo
Dept. of Computer Science
ETH Zurich
digirols@inf.ethz.ch

Flavio Vella
Dept. of Computer Science
Sapienza, Università di Roma
vella@di.uniroma1.it

Torsten Hoefler
Dept. of Computer Science
ETH Zurich
htor@inf.ethz.ch

***Abstract*—The constantly increasing gap between communication and computation performance emphasizes the importance of communication-avoidance techniques. Caching is a well-known concept used to reduce accesses to slow local memories. In this work, we extend the caching idea to MPI-3 Remote Memory Access (RMA) operations. Here, caching can avoid inter-node communications and achieve similar benefits for irregular applications as communication-avoiding algorithms for structured applications. We propose CLaMPI, a caching library layered on top of MPI-3 RMA, to automatically optimize code with minimum user intervention. We demonstrate how cached RMA improves the performance of a Barnes Hut simulation and a Local Clustering Coefficient computation up to a factor of 1.8x and 5x, respectively. Due to the low overheads in the cache miss case and the potential benefits, we expect that our ideas around transparent RMA caching will soon be an integral part of many MPI libraries.**

I. Motivation

Data caches are among the most performance-critical components in computer systems. Caches exploit the temporal and spatial locality that is inherent to many applications. They are an integral component of hardware architectures such as CPUs and software systems such as databases or filesystems. Most of the existing caches are meant to accelerate accesses between a local slower medium (e.g., a hard disk or DRAM memory) and a faster medium (e.g., on-chip memory).

We propose a Caching Layer for MPI (CLaMPI) that extends the MPI-3 remote-memory access parallel programming environment [14], which has been optimized for systems with remote-direct memory (RDMA) hardware [2, 21]. Differently from traditional caching systems, CLaMPI caches *remote* or *horizontal* accesses instead of the above-mentioned traditional *vertical* accesses. To illustrate the potential gains, we show the latency for various distances in a hierarchical Cray Cascade architecture in Fig. 1. Here, access latencies range from less than 100ns for a local DRAM access (less than ten nanoseconds if the access is cached on the CPU chip) up to 2-3 microseconds for remote accesses, spanning three orders of magnitude. This demonstrates the potential usefulness of a system that caches remote data in local DRAM.

The analysis of temporal and spatial locality in horizontal communications has not received much attention so far. To motivate our work, we analyzed the locality of a representative Barnes-Hut N-Body simulation in Fig. 2. This

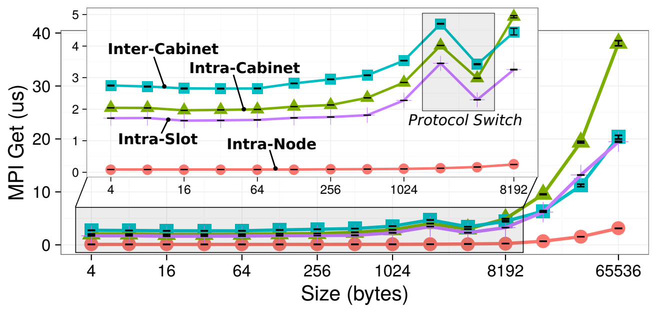

Figure 1: Latency per message size and processes/nodes mappings.

simulation accesses the same remote data up to 3,500 times. We argue that repeated remote accesses are typical in many irregular applications and that a caching layer can reduce the communication for such data-dependent accesses similarly to communication-avoiding algorithms that exploit the fixed access structure of the computation.

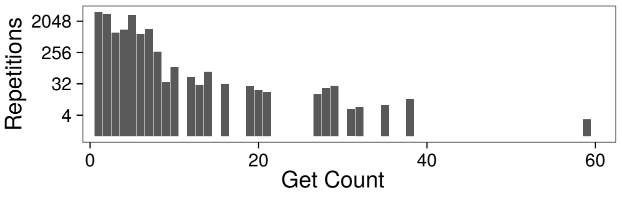

Figure 2: N-Body simulation on 4 processes and $4,000$ bodies. The histogram shows how many gets (x-axis) are repeated y times (y-axis).

These two observations demonstrate that caching with CLaMPI has a high potential to improve the performance of irregular RMA programs in today's supercomputers. Software caching in the RMA context is challenging due to the single-digit latencies of remote accesses. For example, a generic software caching scheme will require coherence messages, that alone will be more expensive than issuing the remote access directly. Thus, we design CLaMPI to specifically fit into the MPI-3 RMA programming model: we cache only get calls and we use MPI's epoch semantics to guarantee light-weight consistency. Furthermore, we allow insert operations to fail to guarantee fixed overheads that never slow down communications. To keep overheads lowest, we utilize variable-size cache entries and we adapt parameters of the cache during the runtime. All-in-all, CLaMPI innovates along several fronts to implement a super-light-weight online-adaptive caching system that almost seamlessly fits on top of existing MPI-3 RMA implementations.

A. *MPI-3 One Sided*

The MPI-3 standard [10] defines the One-Sided communication interface, also known as Remote Memory Access (RMA). A set of processes in a specific *communicator* (e.g., `MPI_COMM_WORLD`) can expose a memory region over the network by creating a *window* with the `MPI_Win_create` or `MPI_Win_allocate`. One-sided operations `MPI_Put` and `MPI_Get` are used to write to or read from a *window* exposed by a remote processes, respectively. Both the operations are blocking. One-sided operations are often supported by remote direct memory access (RDMA) hardware.

RMA operations on a specific window can be issued only during access epochs. Synchronization calls are used to start/terminate an access epoch. The MPI standard defines two target synchronization modes: active and passive. The passive mode does not require the participation of the target process: the initiator (i.e., the process that issues the RMA operation) can access the window of a specific process during the section delimited by the `MPI_Win_lock` and `MPI_Win_unlock` calls. Any process sharing the same window can be targeted if the epoch is delimited by the `MPI_Win_lock_all`/`MPI_Win_unlock_all` calls. On the contrary, active target synchronization requires the participation of both initiator and target processes. Without loss of generality, in the following we assume that the passive mode used. In fact, CLaMPI does not depend on a specific target synchronization mode but on the epoch closure event, that is present in both active and passive modes. All the RMA operations issued during an epoch are completed when the synchronization call concluding the epoch returns. In the passive synchronization mode, `MPI_Win_flush` can be used to complete preceding RMA communications without concluding the current epoch.

II. Caching RMA

Caching RMA operations is different from traditional cache designs. Other caches (e.g., CPU or filesystem) usually accelerate synchronous (blocking) accesses that need to be consumed immediately. MPI enables asynchronous communication arranging accesses in epochs. Thus, while it is important for traditional caches to quickly consume writes, MPI does not benefit from write access caching because the MPI epoch model forbids conflicting `MPI_Put` and `MPI_Get` operations in the same epoch [14]. As a consequence, `MPI_Puts` issued in the same epoch must target different memory areas - i.e., local caching can thus not prevent network accesses. Moreover, `MPI_Put` and `MPI_Get` operations cannot target the same memory area in the same epoch - i.e., read after write patterns cannot be exploited. Thus, **we focus on caching gets**.

A *get* operation can target arbitrarily large data segments, opposed to the CPU or disk caches that limit the read/write operations to the cache-line or block size, respectively. A block-based software cache for RMA would require to fix a block size. This introduces an internal fragmentation problem, and raises the question of how to handle requests not fitting in a block. Possible answers to this question are: 1) not caching requests targeting data larger than the block size; 2) distribute such requests on multiple blocks. In the following discussion we assume that the second solution is adopted.

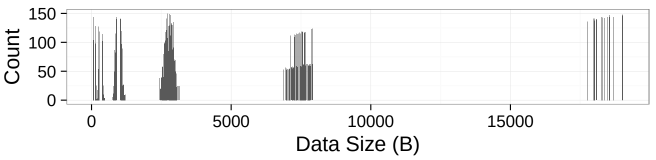

Figure 3: Data Size distribution of a Local Clustering Coefficient (LCC) instance, averaged on 32 nodes. R-MAT input graph: 2^{16} vertices, 2^{20} edges.

As an example, Fig. 3 reports the distribution of the data segment sizes accessed by an instance of the Local Clustering Coefficient computation (described in Sec. IV-C). A block size of $5\,\text{KB}$ would allow to store the 82% of all the requests in a single block. However, the average size of such requests is $\sim 1\,\text{KB}$, hence the 80% of the block space will be wasted in average for all these entries. The remaining 18% of the entries would require 2 blocks on average. A smaller block size would reduce the internal fragmentation but increase the average number of blocks needed to store the cached entries. **Our caching system handles variable-size cache entries** in order to avoid the internal fragmentation and minimize the CPU cache misses during the data copy phase.

According to the MPI epoch model, the destination buffer of a *get* issued in an epoch i can be considered ready - i.e., the requested data is available - after the end of epoch i. No assumptions on the destination buffer can be made by the MPI layer after that moment. This requires us to keep a separate storage area for the cached *get*s. Moreover, RDMA does not allow to copy the same payload into more than one destination buffers, so **data has to be explicitly copied into the cache memory at the epoch closure time or after a synchronization call (i.e., `MPI_Win_flush`).**

A. *Notation*

An RMA operation transfers data between the process from which the operation originates (i.e., the initiator) and a target process. We define a get as an operation transferring data from the target to the initiator. A get x initiated by a process p is uniquely identified (with respect to p) by a tuple:

$$x = (win, eph, trg, dsp, dtype, count)$$

Where win is an MPI window, eph is the epoch in which x is issued, trg is the target rank, dsp is the displacement in win, $dtype$ is an MPI datatype, and $count$ is the number of entries of type $dtype$ to transfer. A counter $w.eph$ is associated with each window w, counting the number of concluded epochs since the window creation. A get issued on w takes the current $w.eph$ as epoch identifier: $x.eph = w.eph$.

A caching-enabled window w is associated with a caching layer $C_w = (I_w, S_w)$, where I_w and S_w are the data

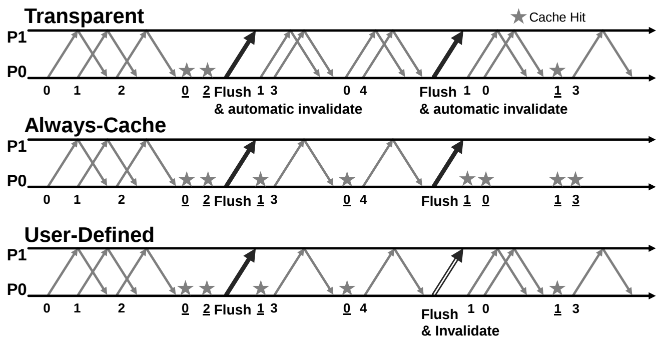

Figure 4: Effects of the different operational modes with respect to the issuing epochs. The ⋆ indicates a cache hit.

structures used for indexing and storing cache entries, respectively. The *get* requests targeting a caching-enabled window w are processed by C_w and are referred to as get_c. The number of entries that can be cached in C_w is $|I_w|$, while the size of the memory buffer used to store the cache entries is $|S_w|$. An entry i of I_w is identified by the tuple: $i = (trg, dsp, dtype, count, ptr)$, where ptr is a pointer to the i's data that is stored in S_w. We define $C_w.G$ as the sequence of get_c operations on C_w issued in program order.

B. Handling Datatypes

The proposed caching layer uses the MPI Datatype Library [19] in order to support arbitrary datatypes. It allows us to flatten the datatype d to a list of data blocks $d_i = (s_i, o_i)$ where s_i is the size of the data block and o_i is its offset in the data buffer. Let x be a *get* or an entry of I_w, we define $size(x)$ as the sum of the sizes of all the data block composing $x.dtype$ multiplied by $x.count$.

III. CLaMPI

One of our main goals is to minimize the cost of the cache hit while introducing a minimal overhead with respect to the non-cached *get* operation in the cache-miss case. According to the considerations expressed in Sec. II, an optimal handling of the cache-hit case would consist of only the local data-copy from the cache to the local destination buffer. Similarly, an optimal cache-miss would perform the get operation for retrieving the data plus one additional memory copy. However, additional overheads stemming from cache managing activities - i.e., renaming, lookup, allocation, and replacement - have to be taken into account.

A. Caching-Enabled Windows

CLaMPI offers three different strategies (i.e., operational modes) to enable the caching of RMA accesses: *transparent*, *always-cache*, and *user-defined*. Examples of all these operational modes are reported in Fig. 4.

Transparent: It allows applications to enable caching without any code change. All the MPI windows are caching-enabled. Since no assumptions can be made on the data access pattern, the cache is invalidated at each epoch closure.

Always-Cache: If the memory area identified by the window is read-only for the entire window lifespan then there is no need to perform any cache invalidation. Examples of such applications are the ones applying graph-processing algorithms: if the graph structure is not modified, then the window representing it can be set in the always-cache mode. Such information can be communicated to CLaMPI as an `MPI_INFO` key passed at window creation time.

User-Defined: This strategy let the user define epochs, or sets of consecutive epochs, in which the memory area identified by a window is in a read-only state. Use cases that can take advantage of this operational mode are, for example, BSP-like (Bulk Synchronous Parallel) applications presenting steps where no write accesses are performed towards the specific window. In Sec. IV-B we discuss the Barnes-Hut algorithm, which falls in this category. With this strategy, the user creates the window with the *always-cache* option. The cache can be explicitly invalidated using the `CLAMPI_Invalidate(MPI_Win win)` call when the sequence of read-only epochs terminates.

```
MPI_Win_lock(MPI_LOCK_SHARED, peer, 0, win);
while (!terminate){
  MPI_Get(lbuf1, ..., peer, off1, ..., win);
  MPI_Get(lbuf2, ..., peer, off2, ..., win);
  MPI_Win_flush(peer, win); //closes epoch
  terminate = computation(lbuf1, lbuf2);
}
CLAMPI_Invalidate(win);
MPI_Win_unlock(peer, win);
```

Listing 1: Example of User-Defined Caching Strategy

Listing 1 shows an usage example of the *user-defined* mode: a set of read-only epochs - i.e., where no write accesses are issued - is delimited by the `MPI_Win_lock` and `MPI_Win_unlock` calls. All the gets performed in these epochs are cached. The cache is explicitly invalidated with a `CLAMPI_Invalidate` before the last epoch termination.

The *transparent* and *always-cache* strategies do not require any modification to the MPI standard, while an explicit cache invalidation call is required by the *user-defined* operational mode. Alternatively, the MPI standard could be extended in order to offer a special *get* call, allowing the user to use/bypass the caching on a per-operation basis. In the current MPI-compliant model, the user could achieve the same by creating two windows, with the same local memory, and enabling just one of the two for caching. At this point the user can issue operations on the two different windows according with the need to cache such operations or not.

B. Processing Gets

When a get_c is issued on a window w, the index I_w is queried to check if the entry is already in cache: the query result is the state of the searched cache entry. A cache entry can be in one of the following states: `MISSING`, `PENDING`, or `CACHED`. Fig. 5 sketches the possible state transitions.

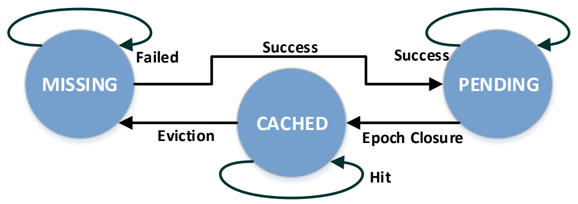

Figure 5: Cache entry state diagram. The initial state of any cache entry is `MISSING`. Accesses of type *direct*, *capacity* or *conflicting* are reported as *success*.

1) Hitting Access: Given a $get_c\ x$ targeting a window w, we have a cache hit if there exists an entry i in I_w such that $x.trg = x_c.trg \wedge x.dsp = i.dsp$. If $size(x) \leq size(i)$ we have full hit, it is a partial hit otherwise. This definition allows us to implement I_w as a constant lookup time data structure (see Sec. III-C1). A different approach would look up for overlapping gets with different displacements. However, this would lead to a $O(\log N)$ lookup cost, where N is the number of cached entries (e.g., interval trees [23]). A lookup returning an entry in the `CACHED` or `PENDING` state is defined as *hitting* access. An entry in the `CACHED` state can be directly copied into the destination buffer. If the entry is `PENDING`, the same data has already been requested from a different *get* in the same epoch: it will be copied into the destination buffer at the end of the current epoch. If the hit is only partial, a remote *get* is issued to acquire the missing data. The entry is extended only if S_w contains enough space to store it. A `CACHED` entry that is selected as victim by the eviction procedure is set as `MISSING`.

2) Direct/Conflicting/Capacity Access: If the lookup returns a `MISSING` entry, a remote *get* is issued to acquire the requested data. The cost of the remote *get* can be overlapped with the caching overheads that are required to check if I_w and S_w are able to respectively index and store the new data and to execute the eviction procedure if this check fails. An access is *direct* if it does not require any eviction. We have a *conflicting* access if an eviction is required due to conflicts in I_w or a *capacity* access if the eviction is due to missing space in S_w and enough space to store the new data is freed after it. The access is defined as *failing* otherwise. A *direct*, *conflicting*, or *capacity* access move the targeted cache entry from the `MISSING` to the `PENDING` state.

C. Data Structures

The core of CLaMPI consists of the index I_w and storage S_w data structures. Now we discuss a their possible implementation, assuming a caching layer C_w on a window w.

1) Naming - Indexing Entries: In CLaMPI, cache entries are indexed using a hash table. We employ the Cuckoo scheme [11, 17] for resolving hash collisions. It uses p hash functions $h_0(e)...h_{p-1}(e)$ to identify the possible locations of an entry e in the hash table, leading to a lookup cost linear in p. Universal hashing [5] can be used in order to derive the p hash functions. We use $p = 4$, that showed to achieve up to 97% space utilization of the hash table [11].

As described by Fotakis et al. [11], the insertion procedure tries to insert a new element x in $h_i(x)$ where i is randomly chosen in $[0, p)$. If $h_i(x)$ already contains an element y (i.e., $h_i(x) = h_i(y)$) then x is inserted in $h_i(x)$ and a new location for y is searched. The procedure continues iterating trying to insert y in $h_j(y)$ with j randomly selected among the $p-1$ remaining positions. We define the *insertion path* as the sequence of hash table entries visited during the insertion procedure. The procedure stops when either an empty position is found or a maximum number of iterations is reached. This threshold helps to detect cycles in the Cuckoo graph. Normally, the second case (*insertion failure*) is handled by selecting a new set of hash functions and re-hashing all the entries. In our approach we handle the *insertion failure* case as a *conflicting* access, triggering the eviction procedure in order to evict one of the entries in the insertion path.

2) Storage - Allocating Space: Cache entries are contiguously stored in a memory buffer of size M. The memory buffer size can be fixed or dynamically adjusted at runtime (see Sec. III-E). We allocate memory regions of size as multiple of the CPU cache line size, in order to maintain CPU cache data alignment in S_w. Free memory regions are indexed with an AVL tree [1], using their sizes as indexes: the search of a free region requires $O(logN)$ time, where N is the number of free regions. This solution implies that new allocations are served with a best-fit policy. A successful search returns an AVL node representing the free region: the node is removed from the tree and, if the free region is not fully occupied by the requested data, a new node is inserted for indexing the remaining free space.

Storing cache entries in a contiguous space allows us to exploit hardware prefetching during the memory copy in the case of a *hitting* access. However, this layout may cause external fragmentation of the memory buffer. We have external fragmentation when free memory is distributed in a way such that it is not possible to satisfy new allocation requests, even if the total free memory is enough to satisfy them. In order to tackle this problem, we associate a *positional* score with each entry to give an indication on how they contribute to the current fragmentation of S_w.

Let us define $C_w.ags(i)$ as the average get size of the entries processed by C_w after the i-th get_c in $C_w.G$. Let c be a cache entry in C_w, we define d_c as the total free memory in S_w that is adjacent to c. The positional score $R_P^i(c)$ of c after processing the i-th get_c is:

$$R_P^i(c) = \min\left\{\frac{|C_w.ags(i) - d_c|}{C_w.ags(i)}, 1\right\}$$

After the i-th get_c has been issued, an entry c gets a position score proportional to the difference between the total free memory adjacent to it (i.e., d_c) and $C_w.ags(i)$. The lower the positional score of c, the higher the probability that evicting c will make enough space to store new data of size $C_w.ags(i)$. Fig. 6 reports a sketch of the memory buffer

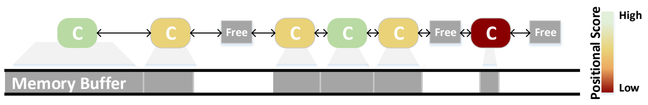

Figure 6: The memory buffer is represented by a list of cache entry (*C*) and free region (*Free*) descriptors.

management structures (described in Sec. III-C3). Darker colors of the cache entry descriptors - marked with *C* - indicate a lower positional score of the associated entry.

3) Cache Entries Descriptors: To compute the positional score of a cache entry c, we need to keep the value of d_c up to date. This value has to be updated at every eviction or allocation targeting a memory region adjacent to c. Now we briefly explain how this update can be efficiently performed.

Cache entries and free regions are associated with descriptors. These descriptors store intervals endpoints and they are organized in a doubly linked list that reflects the order in which they appear in S_w (see Fig. 6). We use $r.prev$ and $r.next$ to indicate the previous and the next element of the r descriptor in this list, respectively. When a new entry is cached, a free region is searched as described in Sec. III-C2. Since the found free regions is linked with its descriptor f and the new cache entry descriptor can be inserted between f and $f.next$, we conclude that the insertion in this sorted list can be performed in constant time. During the insertion, the values of $d_{f.prev}$ and $d_{f.next}$ are updated by subtracting the size of the new allocated region. The eviction procedure returns the descriptor of the victim entry c: the values of $d_{c.prev}$ and $d_{c.next}$ can be adjusted starting from it. The removal from the sorted list takes constant time because we already know the descriptor to remove. If c is adjacent to a free region f, then f is enlarged and respective node is updated in the AVL tree indexing the free regions.

D. Eviction Procedure

In the case of a *conflicting* or *capacity* access, the eviction procedure has to select and evict a cached entry. On a *conflicting* access, the victim is selected among the entries in its *insertion path* (see Sec. III-C1). Now we describe the victim selection scheme adopted if a *capacity* access happens.

The idea is to select the victim among a random sample of M cache entries, where M is a configurable parameter. Let us model I_w as a circular array: $I_w[i..j]$ indicates the set of entries stored from position i to j in I_w. Assuming that the cached entries are uniformly distributed in I_w[1], we define our random sample as $I_w[i..(i+M-1)]$, where i is randomly chosen. If the sample is empty, the procedure keeps scanning until at least one non-empty entry is found. Hence, in order to select a victim, the actual number of visited entries starting from i is $v_i = \max(M, k_i)$, where k_i is the number of consecutive empty entries starting from i. The value of k_i depends on the sparsity of the I_w.

[1]It relies on the hash functions used by the Cuckoo scheme, see Sec. III-C1

1) Victim Selection: For each cache entry x we define $x.last$ as the index in $C_w.G$ of the last get_c that matched it. The temporal score of a cache entry x after the i-th *get* in C_w is the ratio between $x.last$ and i: $R^i_T(x) = \frac{x.last}{i}$.

We define the score of a cache entry as function of its temporal and positional score:

$$R^i(x) = R^i_P(x) \times R^i_T(x) \quad 0 \le R^i(x) \le 1$$

The aim of $R^i(x)$ is to estimate how x contributes to the fragmentation of S_w and its reuse probability. The eviction procedure selects the victim as the entry with the lowest score among the visited ones.

2) Discussion: The missing guarantee to be always able to store a new cache entry make our proposed caching layer following a weak-caching approach. We motivate our design choice with the following points:

- Cache entries have variable size: multiple evictions could be required in order to make room in S_w to store the incoming data. This would lead to an overhead proportional to the current number of cached entries in the worst case.
- If a get_c is targeting highly-reused data, then it will be issued multiple times, leading to multiple evictions, hence increasing its own probability of being successfully cached.

The proposed solution, that evicts a constant number of entries, also avoids the case in which multiple entries are evicted due to a sporadically accessed big data segment.

E. Parameter Tuning

We define the working set $W_w(t, \tau)$ [8] as the set of gets issued to a caching layer C_w over the interval $[t-\tau, t]$, where t and $t-\tau$ are indices in the $C_w.G$ sequence. The set of gets belonging to the working set and that are stored in cache at time t is defined as $\gamma(t,\tau)$. The CLaMPI parameters $|I_w|$ and $|S_w|$ introduce the following constraints on the $\gamma(t,\tau)$:

$$|\gamma(t,\tau)| \le |I_w| \qquad \sum_{g\in\gamma(t,\tau)} size(g) \le |S_w|$$

The index size $|I_w|$ limits the total number of cache entries that can be indexed $|\gamma(t,\tau)|$, while the memory buffer size $|S_w|$ limits the total space that the entries in $\gamma(t,\tau)$ can occupy. Hence, $|I_w|$ and $|S_w|$ have a direct impact on the number of conflicting and capacity accesses, respectively.

1) Adaptive Parameter Selection: Tuning the above described parameters can be a challenging task for the user, especially if we consider their direct impact on application performance. We propose a strategy that allows CLaMPI to adjust such parameters at runtime by itself. The idea is that the starting values of $|I_w|$ and $|S_w|$ are predefined. The caching layer will then increase or decrease those values keeping track of some statistics about the cache usage at runtime. It is worth noting that changing the value of any parameter requires the invalidation of the cache.

A high number of conflicting accesses is a signal that I_w has to be extended. When the ratio $\frac{conflicting}{total\_gets}$ is greater than a *conflict threshold*, the hash table size $|I_w|$ is increased by a *index_increase_factor*. Let us define q as the ratio between the number of non-empty and total entries visited by the eviction procedure. We identify the event of this value getting lower than a certain threshold as a signal of a highly sparse I_w. In this case, to improve the quality of the victim selection (i.e., increasing q), we decrease $|I_w|$ by a *index_decrease_factor*.

The memory buffer size is adjusted according with the total number of capacity and failed accesses. When we observe that the number of *capacity* and *failing* accesses ($\frac{capacity+failed}{total\_gets}$) becomes greater than a *capacity threshold*, we increase $|S_w|$ by a *memory_increase_factor*. On the other side, if the working set is stable (i.e., $\frac{hits}{total\_gets} > stable\_threshold$) and the free space in S_w is above the respective threshold, then $|S_w|$ is decreased by a *memory_decrease_factor*.

In the rest of the paper we use the terms *fixed* and *adaptive* to refer the two strategies where the discussed parameters are fixed or dynamically adjusted, respectively.

IV. Experiments

The benchmarks presented in this section are executed on Piz Daint@CSCS, a hybrid Cray XC50/XC40 system. Each compute node is equipped with an eight-core Intel Xeon E5-2670 clocked at 2.60GHz. The system is interconnected with Cray's Aries network arranged in a Dragonfly topology. We use the optimized, open-source foMPI [12] implementation of the MPI standard in order to enable a comparison with respect to the fastest available RMA implementation on the targeted system. All libraries and benchmarks discussed in this section are compiled using the Cray Programming Environment 5.2.82. We use the Berkeley UPC compiler 2.22 for UPC benchmarks and GNU gcc 4.3.4 for applications requiring features not supported by the the Cray C compiler. Time measurements are taken using the LibLSB timing library [13]. The number of repetions per experiment is selected such that the 95% confidence interval is no larger than the 5% of the reported median. When not differently stated, we map one MPI rank - also referred to as processing element - per node.

A. Micro-Benchmarks

In this section we use a micro-benchmark consisting of two processes mapped on different physical nodes, namely *initiator* and *target*. The initiator creates a sequence of *get*s towards the area of memory exposed by the target. This sequence is created in the following way: 1) we create a set of $N = 1$K *get*s targeting different data (i.e., no cache hits are produced if this sequence is performed on an ideal cache system). The size (in bytes) of each *get* in such sequence is randomly chosen in the set $\{2^i | i = 0..16\}$ according to an uniform distribution; 2) a sequence of $Z \geq N$ *get*s is created by sampling from the set of step 1. The sampling is done according to a normal distribution $\mathcal{N}(\frac{N}{2}, \frac{N}{4})$. We use a normal distribution to create a sequence in which a subset of *get*s is more frequent than the others.

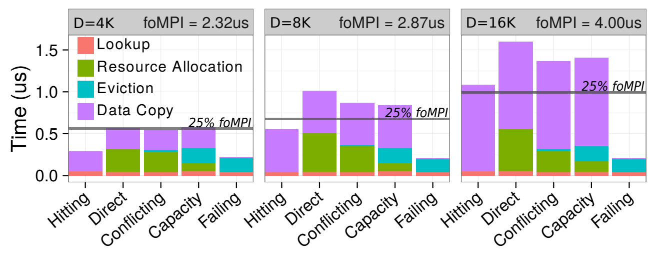

Figure 7: CLaMPI caching costs for different access types and data sizes (D). The horizontal line is the 25% of the foMPI latency. The 95% CIs are always within the 5% of the reported medians.

1) Caching Costs Characterization: We define the latency of a *get* operation as the time interval between the issuing of the operation (i.e., `MPI_Get`) and the time in which the requested data has been copied in the destination buffer (e.g., `MPI_Win_Flush`). Fig. 7 shows the overheads introduced by the different access types and data sizes. In this benchmark we set $Z = 20$K. As expected, the lookup cost is constant for all the access types. The *hitting* access consists of only the lookup phase and the data copy from the cache to the user-provided buffer, being up to 9.3x and 3.7x faster than *foMPI* for the 4 KB and 16 KB data sizes, respectively.

An eviction is required in the *conflicting*, *capacity*, and *failing* accesses. The eviction cost, that accounts also for the victim selection, is constant for *conflicting* accesses. In this case, the victim is selected among the ones in the *insertion path* of the new entry. In the other cases, this cost depends on the sample size (see Sec. III-D1). The data copy phase is not present in the *failing* access, since in this case no resources are available to store the missing entry.

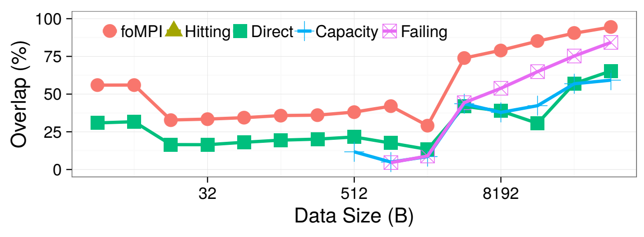

Figure 8: Portion of the communication that can be overlapped with computation as function of the data size.

Fig. 8 shows the results of a communication/communication overlap study. Our reference curve is *foMPI*, that is able to overlap up to the 85% of the communication to computation in the 64 KB case. It provides an upper-bound for the overlap that can be reached by CLaMPI. The *direct* and *capacity* accesses show a similar behaviour, because they are both dominated by the data copy phase. The *failing* access, instead, is able to provide a higher overlap for larger data sizes, because it does not require any additional data copies. CLaMPI was always able to directly cache *get* operations targeting data of size smaller than 512 B, explaining the missing data points for the *capacity* and *failing* accesses.

2) Adaptive Parameter Selection Evaluation: In Sec. III-E1 we discuss an adaptive strategy in order to adjust the sizes of I_w and S_w at runtime. This adjustment is made according to a set of indicators about the performance provided by the caching layer. In Fig. 9, we show the evaluation of the *adaptive* strategy reporting the completion time of the micro-benchmark as function of different hash table sizes. In the *adaptive* case, the value of the hash table size is the starting value that could be dynamically adjusted. The *fixed* strategy presents poor performance when $|I_w|$ is small w.r.t. the number of distinct gets N (i.e., $200 \leq |I_w| < N = 1K$) due to high number of *conflicting* accesses. This does not happen with the *adaptive* strategy, which is able to adjust the hash table size at runtime, minimizing the number of *conflicting* accesses.

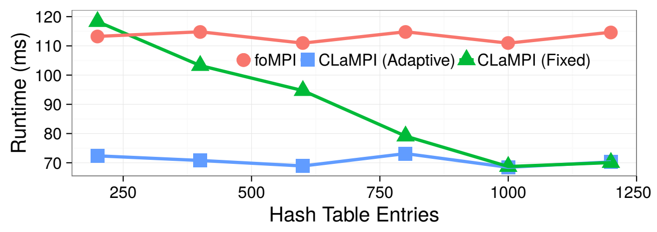

Figure 9: Completion time as function of hash table entries. The number of hash table entries is the starting value for the *adaptive* strategy.

3) Victim Selection Algorithm Evaluation: In Sec. III-D1 we propose an LRU-based victim selection scheme that also aims to reduce the external fragmentation in S_w, referred to as *Full*. In this section we evaluate this scheme, comparing it to the cases in which the entry score is composed only from the *Temporal* (i.e., LRU-like), or the *Positional* score.

In this experiment, the total number of issued gets is set to $Z = 100$K in order to observe the effects of external fragmentation on a longer term. We choose to not include the *conflicting* accesses in this analysis because it is a special case of the *capacity* access. We verified that the number of *conflicting* accesses becomes negligible (i.e., $< 5\%$) when the hash table size is set to values equal or greater than N. In the following analysis we consider only values of $|I_w|$ satisfying this condition.

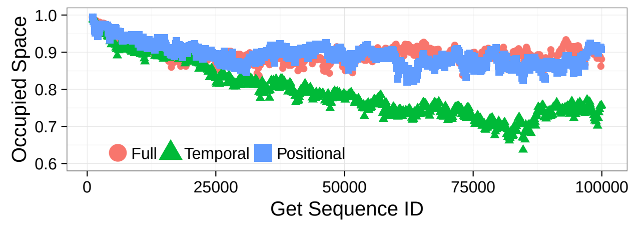

Figure 10: Space occupation per *Get Sequence ID* and the victim selection scheme. $|I_w| = 1.5$K entries. The y-axis is normalized with respect to $|S_w|$.

Fig. 10 reports the fraction of occupied space in the memory buffer as function of the *get* identifier in $C_w.G$, that is how the memory buffer occupation state changes with the issuing of subsequent *gets*. We start reporting measurements once the buffer is completely filled up for the first time (i.e., the first *capacity/failed* access takes place). At this point the buffer is saturated, so the space occupancy gives us an estimation of the external fragmentation: the lower the space occupation, the higher the external fragmentation. The *Temporal* scheme does not take into account external fragmentation, leading to make it increasing with the issuing of subsequent gets. Instead, the *Full* and *Positional* estimate how entries are contributing to the external fragmentation: they keep the space occupation around 90% of S_w capacity.

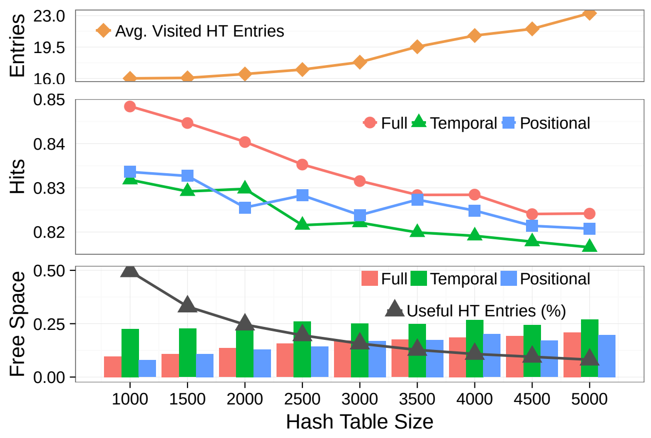

Figure 11: Top: average number of visited entries per eviction. Middle: number of hits per different victim selection scheme. Bottom: free space per victim selection scheme; average number of non-empty entries visited per eviction. All the measures are reported as function of $|I_w|$.

Fig. 11 reports a study as function of the hash table size. On the top we show the average number of visited entries per *capacity/failed* accesses. Here we set $M = 16$, that is the number of I_w entries that are visited to build the sample among which the victim is selected. Increasing the hash table size, we increase the sparsity of I_w, hence we need to visit more than M entries in order to find at least one non-empty entry in I_w. On the middle we show the hit ratio presented by different victim selection strategies. The *Full* scheme provides the best hit ratio for all the considered hash table sizes: By reducing the external fragmentation, it can store more cache entries than the *Temporal* scheme. Moreover, it selects the victim also according with the estimated reuse probability of the cached entries, keeping in cache the most well-positioned and reused ones. On the bottom we show the average free space per different victim selection scheme. As expected, the *Temporal* scheme is the one presenting the higher free space (hence higher external fragmentation). We also show the portion of non-empty entries that are visited during the victim selection. The higher the hash table size, the higher the sparsity in I_w, the lower the number of non-empty entries among which the victim is selected.

B. *N-Body Simulation*

The *N-Body* problem consists of simulating the evolution of a system composed by N bodies. The system evolves as the bodies apply their own force on the others. The algorithm operates on discrete time intervals: at each time step the forces applied from one body on all the others are computed, updating the position and velocity of the bodies.

The Barnes-Hut algorithm [3] presents $O(N \cdot logN)$ time complexity. The idea behind this approach is that if the distance between a body and a different group of bodies is greater than a certain threshold ϕ, then the forces applied by such group can be approximated as if all the bodies in the group are positioned in a point that coincides with the center of mass of the group. The quality of the approximation is controlled by the ϕ parameter. The bodies are organized into an octree: each non-leaf node contains information about the center of mass of the bodies laying in the descendant subtree. The force computation phase consists in a top-down visit of the tree: if a cell is "far enough" or is made by a single body, then the force is computed using its center of mass. Otherwise, the cell is accessed and all the contained cells/leaves are recursively visited.

Different works on this problem point out the potential advantage of exploiting the high locality presented by this irregular application [16, 24]. In fact, during the force computation phase the octree is not modified and a process can access the same node of the tree multiple times. We modified the UPC implementation by Larkins et al. [16] introducing MPI One Sided operations for retrieving/storing cells and leaves of the tree. We enable CLaMPI with the *user-defined* mode (see Sec. III-A), that allows us to invalidate the cache after the termination of the force computation phase.

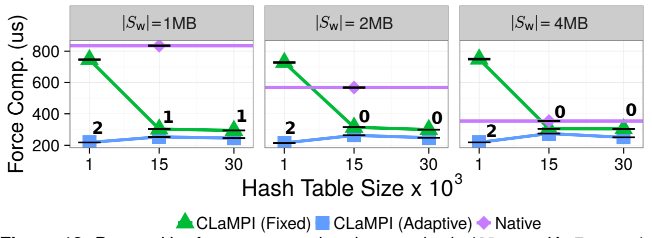

Figure 12: Barnes-Hut force computation time per body (N = 20K, P = 16). The non-caching enabled body force computation needs $1.53\,\mathrm{ms}$ to complete.

In Fig. 12, we compare the two CLaMPI strategies, *adaptive* and *fixed*, with an ad-hoc caching system, referred to as *native*, that was included in our reference UPC implementation. The *adaptive* strategy is annotated with the number of performed invalidations/adjustments. The *Force Computation* time required by foMPI is $1.53\,\mathrm{ms}$. The experiment fixes the number of processing elements to P = 16 and the total number of bodies to n = 20K. The CLaMPI parameters such as $|I_w|$ and $|S_w|$ are varied in order to find the best setting. The memory size of the *native* caching solution is set to the same value of $|S_w|$. The *adaptive* strategy is the one presenting the best performance: in all the cases it converges to $1\,\mathrm{MB}$ as the value of $|S_w|$, while it sets $|I_w|$ = 20K (averaged on all the processes). When $|I_w| = 1K$, the performance of the *fixed* strategy is limited by the high number of *conflicting accesses*, as also confirmed by Fig. 13 for the $|S_w| = 1\,\mathrm{MB}$ case. The *native*

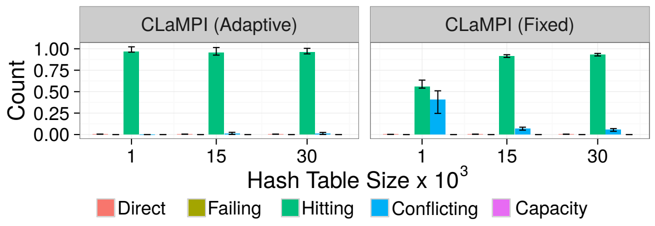

Figure 13: Barnes-Hut body force computation stats. $|S_w| = 1\,\mathrm{MB}$, $N = 20K$ and $P = 16$. The y-axis is normalized w.r.t. the total number of *gets*

solution performance heavily depends on its memory size: it ranges from $\sim 820\,\mathrm{\mu s}$ with $1\,\mathrm{MB}$ to $\sim 400\,\mathrm{\mu s}$ with $4\,\mathrm{MB}$. This behaviour can be explained by the fact that this system is a block-based software cache with direct mapping, hence the number of conflicts is strictly related to the available memory size.

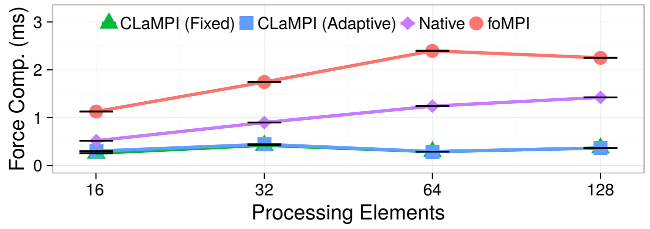

Figure 14: Barnes-Hut weak scaling. Force computation time per body as function of the number of processing elements (PEs). Bodies per process: 1.5K.

Fig. 14 shows the results of a weak scaling experiment. The CLaMPI parameters are set to $|S_w| = 2\,\mathrm{MB}$ and $|I_w| = 30$K. These are the initial parameters of the *adaptive* strategy. The number of bodies is set to $N = P \cdot 8K$ and P, that is the number of processing elements, varies between 16 and 128. Both the CLaMPI strategies outperform the *native* and *foMPI* solutions up to a factor of $\sim$ 3x and $\sim$ 5x, respectively. With this settings, the *adaptive* strategy does not perform any adjustment of the initial parameters.

C. Local Clustering Coefficient

Let $G = (V, E)$ be an undirected graph. Given a vertex v, we define $deg(v)$ as the number of incident edges to v and $adj(v)$ (adjacency of v) as the subset of vertices $u \in V$ such that $\exists(u, v) \in E$. Without loss of generality, we assume that G is partitioned among P processes by using a one-dimensional scheme [4]: each partition $V_i \subseteq V$ is assigned to a process p_i. The process p_i owns all the vertices $v \in V_i$ and all the edges (v, u) such that $v \in V_i$, $u \in V$.

Given a vertex v, the number of possible edges among the nodes adjacent to v is $deg(v) \times (deg(v) - 1)$. The Local Clustering Coefficient (LCC) [22] of v is the fraction of these edges that are actually defined in E. For undirected graphs, it can be computed as:

$$LCC(v) = \frac{2 \cdot |\{(u, w) :\ u, w \in adj(v),\ (u, w) \in E\}|}{deg(v) \cdot (deg(v) - 1)}$$

We introduce CLaMPI in an LCC algorithm leveraging one-sided operations. To compute the LCC of a local vertex

$v_j \in V_i$, the process p_i has to retrieve the adjacency list of every incident vertex u. If the owner of u is not p_i, the retrieve operation can be performed as a *one-sided* communication. The number of *get*s issued by a single process depends on the size of its own partition and on the degree of the vertices in such partition. The size of the each issued *get* depends on the degree of each neighbour of the vertex v.

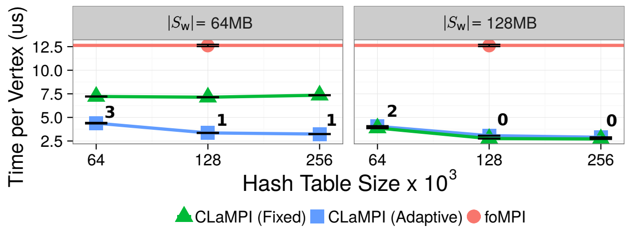

Figure 15: LCC communication time. Input graph: 2^{20} vertices and 2^{24} edges. Number of processes: 32.

The LCC computation exposes data reuse since the adjacency list of the same vertex u can be accessed several times by a single process: every time u appears in the adjacency list of an owned node. In the following we evaluate the described algorithm with and without the CLaMPI support. We use the *always-cache* operational mode of CLaMPI because the graph is never modified. Input instances are created with the R-MAT random graph generation algorithm [6]. We generate scale-free graphs which are used to model real-world networks.

1) Parameter Selection: As discussed in Sec. III-E, ClaMPI exposes two performance critical parameters that are $|I_w|$ and $|S_w|$. In order to evaluate the effects of such parameters on the LCC computation we use an R-MAT graph with 2^{20} vertices and 2^{24} edges distributed over $P = 32$ processes. Fig. 15 compares the vertex processing time (i.e., the time to compute $LCC(v \in V_i)$) of the CLaMPI *adaptive* and *fixed* configurations with respect to *foMPI*. The *adaptive* strategy is annotated with the number of performed invalidations/adjustments. The CLaMPI *fixed* strategy with $|S_w| = 64\,\text{MB}$ is limited by the significant number of *capacity/failed* accesses (i.e., $\sim 60\%$ of the total *get*s). Increasing the memory buffer size to $128\,\text{MB}$ the number of *capacity/failed* accesses decreases to less the 5% of the total total *gets*. In both cases the number of *conflicting* accesses becomes lower than 1% when the $|I_w|$ is set to $256K$ entries.

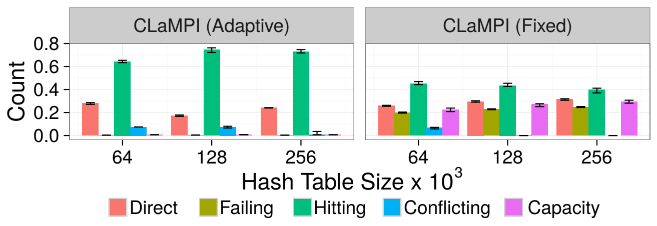

Figure 16: LCC CLaMPI statistics for an R-MAT graph with 2^{20} vertices and 2^{24} edges, distributed on $P = 32$ processes, $|S_w| = 64MB$. The y-axis is normalized with respect to the total number of issued *get*s.

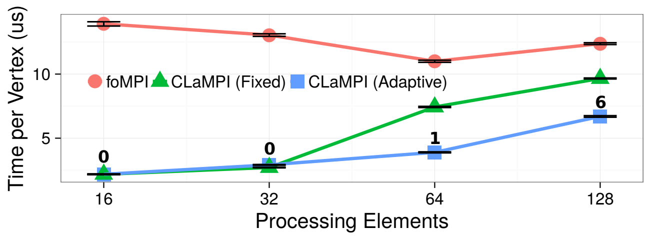

Figure 17: LCC weak scaling experiment starting with R-MAT graph ranging from $S = 19$ to $S = 22$ and $EF = 16$.

This explains the 5x speedup presented by the *fixed* strategy with respect to *foMPI* when $|S_w| = 128M$ and $|I_w| = 256K$.

The *adaptive* strategy achieves a speedup similar to the one presented by the best fixed configuration, independently from the starting parameters. Fig. 16 shows how the *adaptive* strategy is able to keep the number of *hitting* accesses always above the 60% of the issued *get*s already with $|S_w| = 64\,\text{MB}$. The different completion times achieved by this strategy starting from different values of I_w and S_w is explained by the number of adjustments (with consequent cache invalidation) needed to keep the *capacity/failing* and *conflicting* accesses under the specified threshold. In all the cases the adaptive strategy converges to the values of $144K$ entries and $128\,\text{MB}$ for I_w and S_w, respectively.

2) Weak Scaling: In the weak scaling experiment, the problem size per processing element stays constant. We set $|I_w| = 128K$ entries and $|S_w| = 128\,\text{MB}$. The input graph is an R-MAT with $|V| = P \cdot 2^{15}$ vertices, and $|E| = 2^4 \cdot |V|$ edges. We vary P, that is the number of processing elements, between 16 and 128. The experiment results are reported in Fig. 17, where the *adaptive* strategy is annotated with the total number of automatically performed invalidations/adjustments. Increasing the graph scale with the number of processes leads to a situation in which the number of *get*s per process stays constant but the average *get* size increases. As a consequence, the fixed strategy suffers of an higher number of *capacity/failed* accesses when increasing the number of processing elements, as showed in Fig. 18. This explains the gap between the *fixed* and the *adaptive* strategy when $P > 32$. The *adaptive* strategy is able to resize S_w in order to accommodate the larger *get*s, converging to $|S_w| = 256\,\text{MB}$ with 1 adjustment and $|S_w| = 512\,\text{MB}$ with 6 adjustments for the $P = 64$ and $P = 128$ case, respectively. $|I_w|$ is not modified for any value of P.

Overall, the performance of the two CLaMPI strategies starts converging to the *foMPI* one as the number of processes increases. This is due to the nature of the weak scaling experiment with respect to this particular application: the problem size stays constants per process, but the graph is distributed among an increasing the number of processes. As a consequence, the data reuse decreases with P. This is also confirmed by Fig. 18: in the *adaptive* strategy statistics, the number of *direct* accesses increases with P, while other access types stay below the 8% of the total *get*s.

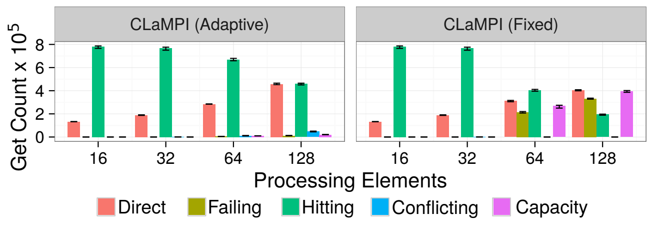

Figure 18: LCC weak scaling experiment statistics.

V. Related Work

There are many studies on software caching schemes for vertical local communications [18, 20] and their interaction with hardware caches [15]. These systems, optimized for large block requests, are designed for blocking interfaces where the application waits for the completion of a request before starting the next request. For horizontal communications, two caching solutions exist for PGAS languages: a software caching system for UPC [7] and Chapel [9]. Both follow traditional approaches from vertical caching and implement a standard block-based read/write caching scheme.

To the best of our knowledge, CLaMPI is the first caching scheme aiming at the asynchronous epoch-based MPI-3 RMA system. Here, we innovate on multiple fronts: we devise a variable-block-size scheme with minimal cache-management overheads and propose a victim selection strategy that also aims to reduce the external fragmentation of the memory buffer. Furthermore, we focus on reads because MPI's semantics support write accesses well at the user-level. CLaMPI integrates with the MPI-3 RMA epoch consistency model and thus simplifies consistency management significantly. Our performance analysis shows that CLaMPI can provide better performances than hand-crafted algorithm-specific implementations (see Sec. IV-B).

VI. Summary

In this work we present a caching system of RMA get operations. The idea is to exploit the data reuse that is typical of irregular applications (e.g., graph processing, N-body simulation), introducing a transparent caching layer between the application and the network. CLaMPI can be easily integrated into the MPI standard requiring minimal modifications. In particular, the proposed design enables a fully associative caching system where cache entries are indexed with a hash table with Cuckoo hashing. To tackle the external fragmentation induced by the variable-size cache entries, the eviction procedure takes also into account the *positional* score of cached entries during the victim selection.

To evaluate the performance of the proposed caching system, we present a set of micro-benchmarks to show the overheads introduced by the various access types. Moreover, we show the effects of employing CLaMPI in two applications such as the N-Body simulation and the Local Clustering Coefficient computation. In both cases we observe how the introduction of the caching layer is able to provide a speedup up to a factor of ~ 5x in the N-Body simulation and ~ 1.8x in the Local Clustering Coefficient computation. In both cases, no invasive interventions on the application code are required to enable RMA caching.

Acknowledgemts

This work was supported by the Swiss National Science Foundation under Sinergia grant CRSII2_154486/1 and by Intel under the OCoOl project. The authors want to thank the Swiss National Supercomputing Centre (CSCS) for the provided computational resources and support, and the entire SPCL research group for the useful discussions.

References

[1] G. Adelson-Velskii and Y. Landis. An algorithm for the organization of information. *Soviet Math. Dokl.*, page 12591262, Mar. 1962.
[2] B. Arimilli, R. Arimilli, V. Chung, S. Clark, W. Denzel, B. Drerup, T. Hoefler, J. Joyner, J. Lewis, J. Li, N. Ni, and R. Rajamony. The PERCS high-performance interconnect. In *Proc. of the IEEE Symp. on High Performance Interconnects (HOTI'10)*, pages 75–82, 2010.
[3] J. Barnes and P. Hut. A hierarchical O(NlogN) force-calculation algorithm. *Nature*, 324(6096):446–449, 1986.
[4] A. Buluc and K. Madduri. Graph partitioning for scalable distributed graph computations. *Graph Partitioning and Graph Clustering*, 588:83, 2013.
[5] J. L. Carter and M. N. Wegman. Universal classes of hash functions. In *Proc. of the ninth annual ACM symposium on Theory of computing*, pages 106–112, 1977.
[6] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *SDM*, volume 4, pages 442–446, 2004.
[7] W. Chen, J. Duell, and J. Su. A software caching system for UPC. *Memory*, 1:P2, 2003.
[8] P. J. Denning. The working set model for program behavior. *Commun. ACM*, 11(5):323–333, May 1968.
[9] M. P. Ferguson and D. Buettner. Caching Puts and Gets in a PGAS language runtime. In *International Conference on Partitioned Global Address Space Programming Models (PGAS)*, pages 13–24, 2015.
[10] M. Forum. MPI. A Message-Passing Interface standard. Version 3.0. 2012.
[11] D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis. Space efficient hash tables with worst case constant access time. *Theory of Computing Systems*, 38(2):229–248, 2005.
[12] R. Gerstenberger, M. Besta, and T. Hoefler. Enabling Highly-Scalable Remote Memory Access Programming with MPI-3 One Sided. In *Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 53:1–53:12, 11 2013.
[13] T. Hoefler and R. Belli. Scientific Benchmarking of Parallel Computing Systems. In *Proc. of the IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis*, 11 2015.
[14] T. Hoefler, J. Dinan, R. Thakur, B. Barrett, P. Balaji, W. Gropp, and K. Underwood. Remote Memory Access Programming in MPI-3. *ACM Trans. Parallel Comput.*, 2(2):9:1–9:26, June 2015.
[15] R. Huggahalli, R. Iyer, and S. Tetrick. Direct cache access for high bandwidth network I/O. In *Proc. of the 32Nd Annual International Symp. on Computer Architecture*, ISCA '05, pages 50–59, 2005.
[16] D. B. Larkins, J. Dinan, S. Krishnamoorthy, S. Parthasarathy, A. Rountev, and P. Sadayappan. Global trees: a framework for linked data structures on distributed memory parallel systems. In *Proc. of the 2008 ACM/IEEE conference on Supercomputing*, page 57, 2008.
[17] R. Pagh and F. F. Rodler. Cuckoo hashing. In *Proc. of the 9th Annual European Symp. on Algorithms*, pages 121–133, 2001.
[18] V. Pai, P. Druschel, and W. Zwaenepoel. IO-lite: A unified I/O buffering and caching system. *ACM Trans. Comput. Syst.*, 18(1):37–66, Feb. 2000.
[19] R. Ross, R. Latham, W. Gropp, E. Lusk, and R. Thakur. Processing mpi datatypes outside mpi. In *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*, pages 42–53. Springer, 2009.
[20] A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. *SIGOPS Oper. Syst. Rev.*, 35(5):188–201, Oct. 2001.
[21] The InfiniBand Trade Association. *Infiniband Architecture Specification Volume 1, Release 1.2.* InfiniBand Trade Association, 2004.
[22] D. J. Watts and S. H. Strogatz. Collective dynamics of small-world networks. *Nature*, 393(6684):440–442, 1998.
[23] A. P. Witkin. Scale-space filtering: A new approach to multi-scale description. In *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP'84.*, volume 9, pages 150–153. IEEE, 1984.
[24] J. Zhang, B. Behzad, and M. Snir. Optimizing the Barnes-Hut algorithm in UPC. In *Proc. of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 75, 2011.