Memory Management Techniques for Exploiting RDMA in PGAS Languages

Barnaby Dalton[1], Gabriel Tanase[2], Michail Alvanos[1](✉), Gheorghe Almási[2], and Ettore Tiotto[1]

[1] IBM Software Group, Toronto, Canada
{bdalton,malvanos,etiotto}@ca.ibm.com
[2] IBM TJ Watson Research Center, Yorktown Heights, NY, USA
{igtanase,gheorghe}@us.ibm.com

Abstract. Partitioned Global Address Space (PGAS) languages are a popular alternative when building applications to run on large scale parallel machines. Unified Parallel C (UPC) is a well known PGAS language that is available on most high performance computing systems. Good performance of UPC applications is often one important requirement for a system acquisition. This paper presents the memory management techniques employed by the IBM XL UPC compiler to achieve optimal performance on systems with Remote Direct Memory Access (RDMA). Additionally we describe a novel technique employed by the UPC runtime for transforming remote memory accesses on a same shared memory node into local memory accesses, to further improve performance. We evaluate the proposed memory allocation policies for various UPC benchmarks and using the IBM® Power® 775 supercomputer [1].

1 Introduction

Partitioned Global Address Space languages (PGAS) [2–6] have been proposed as viable alternatives for improving programmer productivity in distributed memory architectures. A PGAS program executes as one or more processes, distributed across one or more physical computers (*nodes*) connected by a network. Each process has an independent virtual address space called a partition. The collection of all partitions in a program is called the Partitioned Global Address Space or PGAS. A PGAS process can access both data from the local partition as well as remote partitions. Accessing remote data employs the network via a transport Application Interface (API). The network read and write operations, are typically several orders of magnitude slower than local read and write operations to memory.

The global shared array is a data abstraction supported by most PGAS languages that allows users to specify arrays physically distributed across all

Researchers are supported by the IBM Center for Advanced Studies, and by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.

J. Brodman and P. Tu (Eds.): LCPC 2014, LNCS 8967, pp. 193–207, 2015.
DOI: 10.1007/978-3-319-17473-0_13

processes. Compiler and runtime support are subsequently employed to map from the high-level, index-based, access to either local or remote memory accesses. The PGAS runtime or the compiler translates the shared memory addresses to a process identifier and to a virtual address on the remote process. There are three different methods commonly used for computing remote memory addresses:

Sender Side Lookup: For every shared array instantiated the sender maintains a translation table with a list of virtual base addresses, one for each process partition, pointing to the memory block used in that partition. The downside of this approach is the non-scalable memory requirement for every shared array declared. Assuming there are N shared arrays and P processes in a computation, this solution requires $O(N \times P^2)$ storage.

Receiver Side Lookup: In this case, each process only maintains a local translation table with an entry for every shared array instantiated. The table maps from a shared array identifier to the address of the local memory block used for storage. The downside of this approach is the difficulty of exploiting remote direct memory access (RDMA) hardware features where the memory address must be known on the process initiating the communication.

Identical Virtual Address Space on all Processes: A third solution requires all processes to maintain symmetric virtual address spaces and the runtime allocates each shared array at the same virtual address on all processes. This technique is often more complex, but it provides the benefits of the other two methods we already introduced: efficient address inference on the sender side for RDMA exploitation and minimal additional storage for shared arrays tracking.

The XLUPC compiler and its runtime implementation [7] fits into the third category. The performance of accessing a shared array is a function of two main factors: the latency of address translation and the latency of remote accesses. RDMA and the large shared memory nodes present on a modern HPC system are two hardware features that need to be carefully exploited in order to reduce the latency of these operations. This work presents a symmetric memory allocator that allows easy inference of the remote address for shared array data and subsequently low latency access using either RDMA or direct memory access depending on the remote element location. More specifically, the paper makes the following novel contributions:

- Describe the symmetric heap, a memory allocator that guarantees same virtual memory address for shared arrays on all processes of a PGAS computation, enables more efficient address translation and it allows efficient RDMA exploitation. The allocator does not require kernel support for allocating symmetric partitions.
- Describe the symmetric heap mirroring, a novel solution for transforming remote array accesses into local memory accesses, for processes collocated within same shared memory address space.

2 Unified Parallel C Background

The UPC language follows the PGAS programming model. It is an extension of the C programming language designed for high performance computing on large-scale parallel machines. UPC uses a Single Program Multiple Data (SPMD) model of computation in which the amount of parallelism is fixed at program startup time.

Listing 1.1 presents the computation kernel of a parallel vector addition. The benchmark adds the content of three vectors (A, B, and D) to the vector C. The programmer declares all vectors as `shared` arrays. Shared arrays data can be accessed from all UPC threads using an index or shared pointer interface. In this example, the programmer does not specify the layout qualifier (blocking factor). Thus, the compiler assumes that the blocking factor is one. The construct `upc_forall` distributes loop iterations among UPC threads. The fourth expression in the `upc_forall` construct is the affinity expression, that specifies that the owner thread of the specified element will execute the *i*th loop iteration.

```
#define N 16384
shared int A[N], B[N], C[N], D[N]

upc_forall(i=0; i<N-1; i++; i)
   C[i] = A[i+1] + B[i+1] + D[i];
```

Listing 1.1. A parallel `upc_forall` loop.

The compiler transforms the `upc_forall` loop in a simple for loop and the shared accesses to runtime calls to fetch and store data (Listing 1.2). Each runtime call may imply communication, creating fine-grained communication that leads to poor performance. In this example, the compiler privatizes [8] accesses *C[i]* and *D[i]* (Listing 1.2). The compiler does not privatize the *A[i+1]* and *B[i+1]* accesses because it is possible that these elements belong to other UPC threads. Before accessing shared pointers, the compiler also creates calls for shared pointer arithmetic (*__xlupc_ptr_arithmetic*).

```
#define N 16384
shared int A[N], B[N], C[N], D[N]

local_ptr_C = __xlupc_local_addr(C); local_ptr_D = __xlupc_local_addr(D);
for (i=MYTHREAD; i < N; i+= THREADS){
   tmp0 = __xlupc_deref( __xlupc_ptr_arithmetic(&A[i+1]) );
   tmp1 = __xlupc_deref( __xlupc_ptr_arithmetic(&B[i+1]) );
   *(local_ptr_C + OFFSET(i)) = tmp0 + tmp1 + *(local_ptr_D + OFFSET(i));
}
```

Listing 1.2. Transformed `upc_forall` loop.

In Unified Parallel C Language shared arrays can be allocated either *statically* or *dynamically.* In the *static* scheme, the programmer declares the memory on the heap using the keyword `shared`, as presented in Listing 1.1. Alternatively, the programmer can use the runtime for shared memory allocation:

– `upc_all_alloc(size_t blocks,size_t bytes)`: Allocates *blocks* $\times$ *bytes* of shared space with blocking factor *bytes.* This is a collective call and it must be invoked from all UPC threads.
– `upc_global_alloc(size_t blocks,size_t bytes)`: Allocates *blocks* $\times$ *bytes* of shared space with blocking factor *bytes.* The call uses one-side communication and it must be invoked from only one UPC thread.

- `upc_alloc(size_t bytes)`: Allocates *nbytes* of shared space with blocking factor bytes with affinity to the calling thread.
- `upc_free(shared void *ptr)`: Frees dynamically allocated shared storage.

3 Symmetric Heap Allocation

Modern 64-bit systems have a virtual address space that is several orders of magnitude larger than available physical memory. In a virtual memory system, physical memory may be mapped to any virtual address. Two independent processes on the same computer may have distinct physical memory with the same virtual address in their respective virtual address spaces. Since most of the virtual address space is unused, and virtual addresses are reserved systemically in well-known regions on a system, a region of memory to reserve for symmetric partition use is typically available.

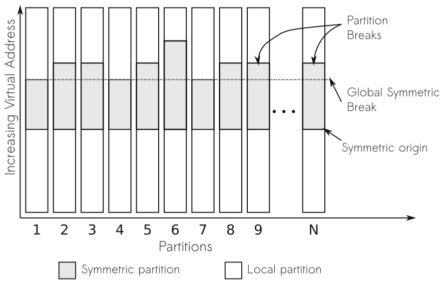


Fig. 1. Layout of the symmetric heap.

Figure 1 depicts the main concepts we use throughout this section. Each process has its own heap (memory partition), and the union of all individual partitions is called *global heap*. A section, called the *symmetric partition*, is reserved for storing distributed data structures within each partition. Each symmetric partition is contiguous in virtual memory and begins at the same virtual address. The collection of all symmetric partitions is called the *symmetric heap*. The starting address in each partition is, by design, chosen to be common across all partitions and is called the symmetric origin. The lowest unmapped address greater than the origin is called the *partition break*. Unlike the origin, we do not require the break to be identical across all symmetric partitions.

3.1 Allocation

Allocation is a distinct process from mapping memory in the methodology we propose. One process, labeled the allocating process, maintains all book-keeping information within its partition. All other partitions maintain a mirror of the memory regions as defined by the allocating process. A span of bytes that is unused in one symmetric partition is unused in all symmetric partitions.

The implementation uses the Two-Level Segregated Fit memory allocator (TLSF) [9], adapted to enable symmetric allocation. The symmetric partition of the allocating process is fragmented in one or more contiguous blocks. Two blocks can't overlap and the collection of all blocks covers the entire address space of the symmetric partition. Each block is either free or used. Each block exists isomorphically in every symmetric partition. A block marked as used (resp. free) in one partition is used (resp. free) in every other partition. There is a collection of all unused blocks called the *freelist*.

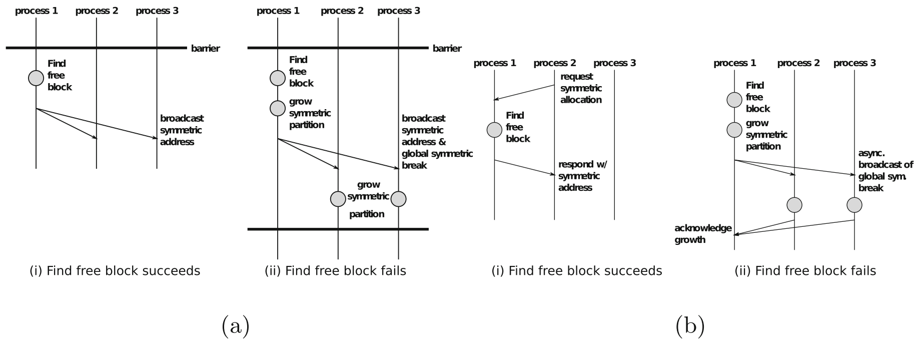

process 1
process 2
process 3
barrier
Find free block
broadcast symmetric address
(i) Find free block succeeds
process 1
process 2
process 3
barrier
Find free block
grow symmetric partition
broadcast symmetric address & global symmetric break
grow symmetric partition
(ii) Find free block fails
(a)
process 1
process 2
process 3
request symmetric allocation
Find free block
respond w/ symmetric address
(i) Find free block succeeds
process 1
process 2
process 3
Find free block
grow symmetric partition
async. broadcast of global sym. break
acknowledge growth
(ii) Find free block fails
(b)

Fig. 2. Operations for collective (left) and independent allocation (right).

3.2 Collective Allocation

An allocation that is executed by all processes concurrently is called collective allocation (e.g., `upc_all_alloc`). The process receives a pointer to the newly created data structure upon completion. Each process begins the allocation request concurrently for an unambiguous count of bytes per partition and waits at a barrier until all processes have started. The allocator process searches for an unallocated block of memory that is at least as large as the requested size. If a sufficiently large block can not be found, then the allocating process increases the global symmetric break and issues a grow operation on the local symmetric partition. The extended bytes are added to the free store guaranteeing the existence of a block to accommodate the request.

The memory block is then removed from the symmetric free store. If it exceeds the original request it is split and the residue is returned to the free store. The address for the block is broadcasted to all other processes together with the new global symmetric break. The remaining processes issue a grow operation if the global symmetric break exceeds their local partition breaks. At this point, they all agree upon an address that is within the range of their symmetric partition. As a last step, the runtime registers the memory of the shared array with the operating system (memory is pinned, as required by the RDMA protocol).

Figure 2(a) presents two scenarios for the collective allocation. When successful (i), the process issues a `Remove-block` operation to mark the block as used and remove it from the freelist. The allocation process issues a `Split-block` operation to create a block and possibly a residue block. It issues an `Insert-block` operation to return the block to the freelist. Finally, the allocating process sends the virtual address of the block to all processes with a broadcast collective operation. In the second example, Fig. 2(a)(ii), the allocating process is unable to find any free blocks in the freelist. The allocating process increases the global symmetric break by the requested size and issues a `Grow` operation on its local symmetric partition. The newly mapped memory is inserted into the freelist with a `Create-block` operation. Allocating process broadcasts the virtual address of the block together with the global symmetric break and the remaining processes issue `Grow` operations on their local partitions.

3.3 Independent Allocation

If a single process needs to issue a global allocation request without collaboration of other processes, the process of allocation is different from collective allocation. The allocating process maintains all book-keeping information within its partition. If the requesting process is not the allocating process, a request is sent to the allocating process. In either case, all requests for independent allocation are serialized by the allocating process. The allocator process searches for an unallocated block that accommodates the request. If a block is found, it is removed from the free-store and split as necessary. The address is returned to the requester in an acknowledgment message. If no block is found that accommodates the request, then the global symmetric break is increased to guarantee sufficient space in the free store. A local grow operation is executed to map memory into the partition. When all processes acknowledge the updated global symmetric break, the address is returned to the requester in an acknowledgment message.

Figure 2(b) presents two scenarios of independent allocation. In Fig. 2(b)(i), the allocation success and the allocating process returns the address to the requested process. However, if the allocation fails, the allocating process exchanges messages to globally grow the symmetric partitions (Fig. 2(b)(i)). In order to maintain the global symmetric break invariant, process 1 issues interrupting requests to each other process to issue a grow to their local symmetric partitions.

4 Heap Mirroring for Shared Memory Optimizations

In this section we present an extension to the symmetric heap introduced in Sect. 3 we call *heap mirroring*. This extension addresses the need to access quickly the memory which is collocated on the same node, but in a different process. It should be stressed that improving intranode communication will not show improvements in UPC programs that uses both intra and inter-node communication as the later will often dominate the overall communication time.

As with the symmetric heap, each process maintains a local partition at a fixed origin which is common across all processes. In addition, each process maintains a view of every partition from collocated nodes. We call these views, mirrored partitions or mirrored heaps. The mirrored partitions are not memory replicas that need to be maintained; rather they maintain the same physical memory pages mapped into multiple processes virtual address spaces.

At this point we must clarify that symmetric allocation maps distinct memory at identical virtual addresses between distinct processes. In contrast, mirroring maps identical memory at distinct virtual addresses within each process. Both are used for accessing shared memory of a remote process or UPC thread.

4.1 Arrangement of Mirrors

If N UPC threads are collocated on the same node, each thread has a symmetric partition and $N-1$ mirrored partitions each separated by *sym_gap*. We always place the process's symmetric heap at the symmetric origin.

For each thread i, we introduce a function, $mirror_i(j)$, mapping a UPC thread j, to a value called a mirror-index with the following properties:

- $mirror_i(j) = -1$ if threads i and j are not collocated,
- $mirror_i(i) = 0$,
- $mirror_i(j) \in \{1, \ldots, N-1\}$ if i and j are collocated,

We build this function dynamically into a hash table at startup time. This guarantees fast lookup by checking at most two locations of an array. The function is used to test whether threads are collocated, and if so, the location of the mirrored partition. The mirrored partition of process j within process i is located at $sym\_origin + mirror_i(j) \times sym\_gap$.

4.2 Implementation Challenges Using SystemV Shared Memory

Most common solutions for implementing shared memory across processes are Unix System V, BSD anonymous mapped memory or kernel extensions [10]. Due to portability and availability reasons we selected System V shared memory and next we discuss some key challenges for the implementation.

In System V, for multiple processes to allocate the same physical memory in their address spaces, they need a shared secret called the key. Inside XLUPC memory allocator, the key is a 16-bit integer, alive only during the brief time period when same node processes are mapping the shared memory.

If N collocated processes need to extend the symmetric heap, they each map the new memory in via `shmget`, and additionally they map the memory of the other $N-1$ processes into their heap. Each of the N memory leases has a shared memory key. They are all valid at the same time, and none of them may collide. To further complicate things, a second instance of the same or other UPC program could be running concurrently also allocating another M segments and they should not collide either. So we need to make collisions impossible between threads in the same program, unlikely between instances of UPC programs running on the same node, but have a resolution scheme that is fast if collisions happen anyway.

The algorithm employed uses the parallel job unique identifier (PID) as a seed from which the keys are inferred. On IBM systems this is the PID of the parallel job launcher (e.g., poe or mpirun) that starts the parallel computation. The algorithm uses the following formula: $keyspace = 0 \times 10000u + atoi(env(PID)) << 14u$. PID's are a 16 bit value and the keyspace is a 32 bit value. This spreads keyspaces with distinct values of PID by at least 16k and avoids the most commonly used first 64k of keys.

Next a scratch space is created in all collocated processes using key $keyspace+0$. This is used for communication and synchronization between collocated processes. If there are N collocated processes, each of them is given an index from $0..N-1$ called their hub_index. Each process will use $keyspace + 1 + hub\_index$ to attempt allocation. In the event of a key collision among the N keys in the scratch pad, detected during allocation of the symmetric heap, we increment the key used by

N (the number of collocated processes) and try again. We record the key used in the scratch space. During mirroring, the runtime looks in the scratch space for the key used by the owner of the symheap. A failure to find the key here is an unrecoverable error.

4.3 Collective and Independent Allocation

The main difference from initial allocation algorithm introduced in Sect. 3.2 is mirroring the collocated heaps. First, while all processes enter the call, only one process will look for a free block. If found its address is broadcasted to all other processes and the function returns. If a block large enough to accommodate this request is not found we calculate the new symmetric heap size that will guarantee a free block large enough and we broadcast the size.

Independent allocation requires the allocator process to asynchronously interrupt all other UPC processes to enter a state where they can collectively work to extend the symmetric and mirrored heaps. When all threads reach the common allocation function than we employ the same mechanism as for collective allocation. Special care is taken to ensure the node-barriers do not deadlock with system-wide barriers.

5 Shared Address Translation

UPC shared objects, such as shared arrays, reside in the shared memory section local to the thread. Shared pointers are actually fat-pointers: a structure that represents the shared address which allows the program to reference shared objects anywhere in the partitioned global address space.

Listing 1.3 presents the structure containing the information. In contrast with the traditional SVD approaches the structure contains all the necessary information for the program to access the data. Due to symmetric memory allocator, the base address is the same in all nodes. Thus, the runtime calculates the virtual offset using arithmetic operations. Before accessing shared data, the compiler automatically creates runtime calls to modify the thread and the offset fields. The runtime call (*pointer increment*) calculates the thread containing the data and the relative offset inside it. The runtime updates the shared pointer with the calculated information.

```c
typedef struct xlpgas_ptr2shared_t
{
  xlpgas_thread_t thread; /* Thread index */
  size_t offset; /* Offset inside thread */
  xlpgas_local_addr_t base; /* Base address */
  size_t allocsize; /* Allocated bytes */
} xlpgas_ptr2shared_t ;
```

Listing 1.3. Shared pointer structure.

Thus, this approach avoids SVD lookup for local access (less overhead) and the SVD lookup during remote access (guarantees RDMA). Furthermore, this approach increases the possibility of compiler shared pointer arithmetic inlining.

Listing 1.4 presents the naive algorithm for the calculation of the thread and the relative offset. At compile time the compiler linearizes the offset of the shared pointer (*idx*). The blocking factor (BF) and element size (ES) can be calculated at compile time or runtime. The first step of the runtime is the calculation of the phase and the block. The runtime uses the block information to calculate the thread that contains the shared data and the phase to calculate the local offset. Note that this simplified example ignores the case of using local offsets: when accessing a structure fields from a shared array of structures. Furthermore, during the linearization the compiler also take into calculation the element access size that can be different from the array element size. Unfortunately, this naive approach is computation intensive.

```
phase = (idx % BF);
block = (idx / BF);
ptr.thread = block % THREADS;
ptr.addrfield = ES * (phase + block / THREADS);
ptr.base = /* address of A */;
```

Listing 1.4. Naive virtual address calculation.

Runtime Optimizations. A first optimization is for the case where the blocking factor is zero, the shared increment is zero, or the increment is the array element size multiplied by the blocking factor. In this case the runtime makes an addition to the offset and returns to the program. When the number of threads and the blocking factor multiplied by the size of elements are power of two, the runtime uses shifts and masks to calculate the offset and the thread containing the data.

Inlining. The runtime optimizations significantly improve the performance of shared pointer increment. However a large fixed overhead is associated with runtime calls. The PowerPC Application Binary Inerface (ABI) mandates significant cost for a function call due to memory operations. Moreover, the branching code in the runtime is required to check for special cases and inserts additional overhead. To solve this challenge the compiler inlines the shared pointer increment when possible and there is enough information at compile time.

6 Experimental Results

This section analyzes the overhead of memory allocation and presents the performance evaluation of symmetric heap and mirroring optimizations. We first use microbenchmarks to examine the cost of allocation, and six benchmarks to examine the performance of applications with regular and irregular communication patterns. For certain experiments we make a comparison of the latest release with an older version of the compiler employing the Shared Variable Directory (SVD) solution.

The evaluation uses the Power®775 [1] system. Each compute node (*octant*-shared memory address space) has a total of 32 POWER7®cores (3.2 GHz), 128 threads, and up to 512 GB memory. A large P775 system is organized, at a higher level, in *drawers* consisting of 8 octants (256 cores) connected in an all-to-all fashion for a total of 7.86 Tflops/s. The most relevant aspect of the machine for the current paper is the availability of RDMA hardware.

The SVD version uses both active messages and RDMA depending on the data size and available opportunities for symmetric allocation. The active messages use the immediate send mechanism on P775 which allows small packets to be sent by injecting them directly into the network. Thus, the SVD implementation achieves lower latency than the RDMA for small messages. For messages larger than 128 bytes, the SVD implementation uses RDMA if the array happens to be allocated symmetrically. This is an opportunistic optimization whose success depends on the history of previous allocation/deallocations on the system. The main difference relative to the solution we present in this paper is that the symmetric memory implementations always guarantee that shared arrays are allocated symmetrically.

The evaluation uses microbenchmarks and four applications:

Microbenchmarks: A first microbenchmark allocates a large number of small arrays to evaluate the cost of memory allocation. The second microbenchmark contains streaming-like local shared accesses to evaluate the cost of address translation and access latency relative to the SVD solution.

Guppie: The guppie benchmark performs random read/modify/write accesses to a large distributed array. The benchmark uses a temporary buffer to fetch the data, modify it, and write it back. The typical size of this buffer is 512 elements and it is static among different UPC threads.

Sobel: The Sobel benchmark computes an approximation of the gradient of the image intensity function, performing a nine-point stencil operation. In the UPC version [11] the image is represented as a one-dimensional shared array of rows and the outer loop is a parallel *upc_forall* loop.

Bucketsort: The benchmark sorts an array of 16-byte records using bucketsort [12] algorithm. Each node generates its share of the records. Each thread uses a $\frac{17}{16} \times 2\,GB$ buffer to hold records received from other threads, which are destined to be sorted on this thread.

UTS: The Unbalanced Tree Search benchmark [13] belongs in the category of state-space search problems. The Unbalanced Tree Search benchmark measures the rate of traversal of a tree generated on the fly using a splittable random number generator.

GUPS: The GUPS benchmark contains accesses in the form of read-modify-write, distributed across a shared array in random fashion. The compiler optimizes the benchmark using the remote update optimization [8].

6.1 Allocator Performance

To evaluate the overhead of allocating shared memory dynamically we created a simple kernel that repeatedly allocates shared arrays using either the collective `upc_all_alloc()` or the one sided `upc_global_alloc()`. For the one sided case only UPC thread zero is performing the allocation while all other threads are waiting in a barrier. We allocate a total of 100 shared arrays and report the average execution time

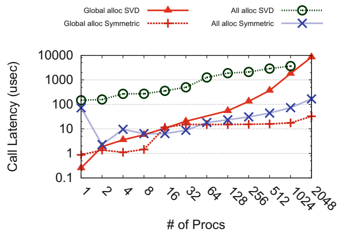


Fig. 3. Performance of global allocation benchmark in average execution time of the call.

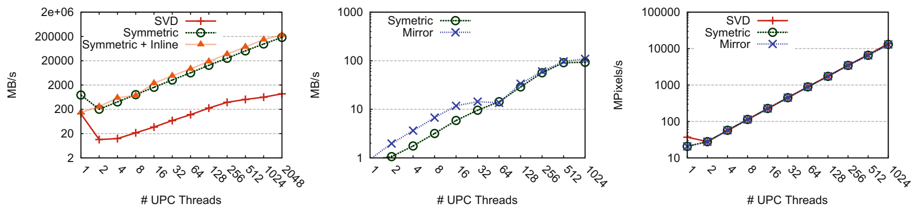
per allocation. Figure 3 presents the performance of the `global_alloc` and `all_alloc` runtime calls without the mirroring optimization and the performance of the same allocations but using a previous version of the runtime that uses SVD. As expected the `all_ alloc` runtime call incurs higher overhead than the `global_alloc` call, due to global synchronization. The `all_ alloc` has an additional internal barrier in the beginning. The latency of `global_alloc` increases in two cases: one when the number of UPC increases from 32 to 64 and one when the number of UPC threads increases from 1024 to 2048. These "cliffs" are the result of higher latency communication. Finally, the overhead of using the mirroring optimization is less than 1 %, and for this reason we excluded it from the plot. For reference only, we also include the allocation performance when SVD is used for address translation. However the results can not be compared directly as more optimizations and changes were added it to the latest version of the compiler framework.

6.2 Local Shared Accesses Performance

Many PGAS loops work almost entirely on local data but the compiler is unable to prove this. The SVD implementation has a heavy penalty on these type of loops. Figure 4(a) and (b) presents the results in aggregated MBytes/s. The privatization is a compiler optimization that could interfere with this experiment and for this reason we disable it. The results show that the symmetric implementation is an order of magnitude better than the SVD version. The improvements are mainly due to the fact that symmetric heap solution avoids the SVD table lookup.

6.3 Regular Applications

The Bucketsort and Sobel benchmarks contain regular communication patterns and usually coarse grain messages. The Sobel implementation includes the static coalescing optimization that aggregates the data statically. The mirroring optimizations gives a significant performance improvement on the Bucketsort benchmark, up to +90 %, when running within one node. On the other hand, the Sobel benchmark benefits less than then Bucketsort with the mirroring optimization because most of the shared accesses are local and are privatized by the



(a) (b) (c)

Fig. 4. Performance of local shared read microbenchmark (a), bucketsort (b), and Sobel (c), using weak scaling.

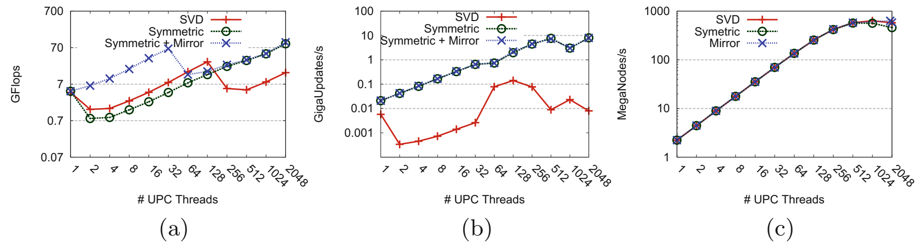


Fig. 5. Performance of Guppie, GUPS, and UTS using strong scaling.

compiler. In regular applications we keep constant the computation per UPC thread.

6.4 Irregular Applications

Figure 5(a) presents the results of the Guppie benchmark. The performance bottleneck is mainly the network latency. The mirroring optimization has the best results for intra node communication. On the other hand, the SVD achieves slightly better performance than the symmetric version for small number of cores. This is because the SVD uses active messages that are optimized to use shared memory within an octant. Symmetric always uses RDMA which is slower than direct shared memory access. The GUPS benchmark uses a different communication mechanism: the remote update packets. In GUPS benchmark, the message creation rate and the address resolution burdens the performance. Thus, the symmetric memory approach is an order of magnitude better than the SVD implementation. There are not any significant differences in the cases of the UTS benchmark. The good shared data locality and in combination with the privatization optimization of the compiler provide good performance, independent from the implementation approach.

6.5 Summary

This benchmark-based performance study shows that he overhead of allocating memory on 2048 UPC thread is less than 35 usec with one side allocation and less than 180 usec for the global allocation. The benchmarks evaluation shows significant improvements in local and shared memory node accesses when using shared pointers due to RDMA exploitation and improved address translation.

7 Related Work

PGAS Allocators. Berkeley UPC compiler [14] uses the GASNet [15] runtime to implement a collective `upc_all_alloc` call. In high performance machines, the GASNet runtime uses the Firehose [16] technique, an explicit DMA registration. Furthermore, the Berkeley framework supports also collective deallocation. At the program startup, each UPC thread reserves a fixed portion of the address space for shared memory using the `mmap()` system call. This address range is the maximum value on the amount of shared memory per-thread that the program can use. In contrast our implementation does not contain any restriction on dynamic memory allocation. The Berkeley UPC framework also uses fat pointers for accessing shared structures. Michigan UPC runtime (MuPC) [17] and CRAY compilers [18,19] use a symmetric heap way to allocate memory for shared arrays. The main differences compared to our work is the fact that the size of the symmetric heap is fixed and controlled using an environment variable and to the best of our knowledge no in-depth details on how memory is managed are provided.

Cray SHMEM [20] introduces another popular PGAS paradigm that provides the notion of shared arrays. The IBM implementation of SHMEM library employs a symmetric heap allocator similar to the one presented in Sect. 3 to efficiently exploit RDMA. It doesn't however employ the mirroring capability of the allocator presented here.

PGAS Shared Pointer Translation. Researchers also use Memory Model Translation Buffer (MMTB), conceptual similar to the Translation Look Aside Buffers (TLBs). The idea is to use a caching that contains the shared pointer values along with the corresponding virtual addresses [21]. In other approaches, that use a distributed shared variable directory (SVD), an address cache is implemented. The caching of remote addresses reduces the shared access overhead and allows better overlap of communication and computation, by avoiding the SVD remote access [8,22]. Another approach is to simplify the shared pointer arithmetic by removing some fields of the shared pointer structure. Experimental results [23] show that cyclic and indefinite pointers simplification improves the performance of pointer-to-shared arithmetic up to 50 %. Machine specific implementations, such as Cray X1 [18] use this approach. Other researchers focus on techniques for minimized the address translation overhead for multi-dimensional arrays [24]. The authors use an additional space per array to simplify the translation for multi-dimensional arrays. In contrast, our approach does not allocate additional space for the translation of multi-dimensional arrays.

Allocators for Efficient Migration. The techniques presented in this paper are similar with the thread or process data migration used from different runtimes. For example, Charm++ [25] implements "isomalloc" stacks and heaps, which are similar technique with ours. However, they focus on the migration of stack and heap to a different node and not for effectively translating virtual addresses to remote addresses. PM2 runtime system [26] implements a similar technique that guarantees the same virtual address that simplifies the migration. Thus, there is no need to keep pointers in a directory. However, the focus of the runtime is the efficient migration of the working unit rather the exploitation of the RDMA.

Shared Memory with MPI. Shared memory exploitation is also a well know technique for MPI programs that are written for distributed memory systems. For example, MPI-3 interface for RMA can be efficiently implemented on networks supporting RDMA as shown in [10]. The key mechanism the authors of [10] employ for this is a symmetric heap and remote address space mirroring similar to the ones presented in this work. However, this approaches differs in the language targeted, and the protocols used for mirroring. The XPMEM Linux kernel module was not available on the particular machine targeted in our work [1] so we relied on a novel protocol built around the more portable SystemV calls as described in Sect. 4. Other approaches requiring kernel support are addressed in the literature [27] with the drawback that they require a slightly different MPI interface.

8 Conclusion

This paper demonstrates the importance of proper design of memory allocator in PGAS languages. The architecture of the allocator and the remote addresses translation to virtual addresses play an important role on application performance. The evaluation shows that both versions can scale with high number of UPC threads. However, the performance of the symmetric memory allocation is better than the SVD in local shared accesses and guarantees the RDMA usage. Furthermore, the mirroring optimization provides an order of magnitude better performance than the simple symmetric version when running in one node. The current implementation is integrated on the latest version of the XLUPC compiler.

References

1. Rajamony, R., Arimilli, L., Gildea, K.: PERCS: The IBM POWER7-IH high-performance computing system. IBM J. Res. Dev. **55**(3), 1–3 (2011)
2. U. Consortium, UPC Specifications, v1.2, Lawrence Berkeley National Lab LBNL-59208, Technical report (2005)
3. Numwich, R., Reid, J.: Co-array fortran for parallel programming, Technical report (1998)
4. Cray Inc., Chapel Language Specification Version 0.8, April 2011. http://chapel.cray.com/spec/spec-0.8.pdf
5. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an Object-oriented Approach to Non-Uniform Cluster Computing. In: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, vol. 40, no. 10. Oct 2005
6. Yelick, K.A., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P.N., Graham, S.L., Gay, D., Colella, P., Aiken, A.: Titanium: a high-performance java dialect. Concurrency Pract. Experience **10**(11–13), 825–836 (1998)
7. Tanase, G., Almási, G., Tiotto, E., Alvanos, M., Ly, A., Daltonn, B.: Performance Analysis of the IBM XL UPC on the PERCS Architecture, Technical report (2013). RC25360

8. Barton, C., Cascaval, C., Almasi, G., Zheng, Y., Farreras, M., Chatterje, S., Amaral, J.N.: Shared memory programming for large scale machines. In: Programming Language Design and Implementation (PLDI 2006) (2006)
9. Masmano, M., Ripoll, I., Crespo, A., Real, J.: Tlsf: a new dynamic memory allocator for real-time systems. In: Proceedings of the 16th Euromicro Conference on Real-Time Systems, ECRTS 2004, pp. 79–88. IEEE (2004)
10. Friedley, A., Bronevetsky, G., Hoefler, T., Lumsdaine, A.: Hybrid MPI: efficient message passing for multi-core systems. In: SC, p. 18. ACM (2013)
11. El-Ghazawi, T., Cantonnet, F.: UPC performance and potential: a NPB experimental study. In: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing, Supercomputing 2002, pp. 1–26 (2002)
12. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms. MIT Press, Cambridge (2001)
13. Olivier, S., Huan, J., Liu, J., Prins, J.F., Dinan, J., Sadayappan, P., Tseng, C.-W.: UTS: An unbalanced tree search benchmark. In: Almási, G.S., Caşcaval, C., Wu, P. (eds.) KSEM 2006. LNCS, vol. 4382, pp. 235–250. Springer, Heidelberg (2007)
14. The Berkeley UPC Compiler. http://upc.lbl.gov
15. Bonachea, D.: Gasnet specification, v1.1. Technical report, Berkeley, CA, USA (2002)
16. Bell, C., Bonachea, D.: A New DMA Registration Strategy for Pinning-Based High Performance Networks. In: Proceedings of the International Parallel and Distributed Processing Symposium, pp. 198–208. IEEE (2003)
17. Michigan Technological University, UPC Projects (2011). http://www.upc.mtu.edu
18. Bell, C., Chen, W.-Y., Bonachea, D., Yelick, K.: Evaluating support for global address space languages on the Cray X1. In: Proceedings of the 18th Annual International Conference on Supercomputing, pp. 184–195. ACM (2004)
19. ten Bruggencate, M., Roweth, D.: Dmapp-an api for one-sided program models on baker systems. In: Cray User Group Conference (2010)
20. Barriuso, R., Knies, A.: SHMEM user's guide for C. Technical report (1994)
21. Cantonnet, F., El-Ghazawi, T.A., Lorenz, P., Gaber, J.: Fast address translation techniques for distributed shared memory compilers. In: Proceedings of 19th IEEE International Parallel and Distributed Processing Symposium, p. 52b. IEEE (2005)
22. Farreras, M., Almasi, G., Cascaval, C., Cortes, T.:Scalable RDMA performance in PGAS languages. In: IEEE International Symposium on Parallel & Distributed Processing, IPDPS 2009, pp. 1–12. IEEE (2009)
23. Husbands, P., Iancu, C., Yelick, K.: A performance analysis of the berkeley upc compiler. In: Proceedings of the 17th Annual International Conference on Supercomputing, pp. 63–73. ACM (2003)
24. Serres, O., Anbar, A., Merchant, S.G., Kayi, A., El-Ghazawi, T.: Address translation optimization for unified parallel c multi-dimensional arrays. In: Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), pp. 1191–1198. IEEE (2011)
25. Huang, C., Lawlor, O.S., Kalé, L.V.: Adaptive MPI. In: Rauchwerger, L. (ed.) LCPC 2003. LNCS, vol. 2958, pp. 306–322. Springer, Heidelberg (2004)
26. Antoniu, G., Bougé, L., Namyst, R.: An efficient and transparent thread migration scheme in the PM2 runtime system. In: Rolim, J.D.P. (ed.) IPPS-WS 1999 and SPDP-WS 1999. LNCS, vol. 1586, pp. 496–510. Springer, Heidelberg (1999)
27. Jin, H.-W., Sur, S., Chai, L., Panda, D.: LiMIC: support for high-performance MPI intra-node communication on Linux cluster. In: International Conference on Parallel Processing: ICPP 2005, pp. 184–191 (2005)