An Interval Constrained Memory Allocator for the Givy GAS Runtime

François Gindraud
UJF
francois.gindraud@inria.fr

Fabrice Rastello Albert Cohen
Inria
{fabrice.rastello,albert.cohen}@inria.fr

François Broquedis
Grenoble INP
francois.broquedis@imag.fr

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; E.1 [*Data Structures*]: concurrent slab

Keywords memory allocation, global address space, distributed shared memory

1. Introduction

The *shared memory* model helps parallel programming productivity, but it also has a high hardware cost and imposes scalability constraints. Ultimately, higher performance will use *distributed memories*, which scales better but requires programmers to manually transfer data between local memories, which is a complex task. Distributed memories are also more energy efficient than shared memories, and are used in a family of embedded computing solutions called *multi processor system on chip* (MPSoC).

Many solutions to manage and abstract memory transfers have been proposed. Among them, *distributed shared memories* (DSMs) aim to implement a software-defined virtual shared memory over a distributed architecture. Such virtually shared memory spaces implement offer a *Global Address Space* (GAS) accessible from all nodes of the system. We study the memory allocation considerations of a new DSM implementation, representative of the general challenges in the field, called *Givy*.

2. DSM

Some NUMA architectures implement a DSM with hardware support. In these machines, dedicated hardware intercept the loads and stores, determines whether the target is in local or distant memory, and triggers network memory transfers in the distant case. They offer the same semantics as shared memory machines (GAS pointers are real pointers, transparent load/stores), but performance can degrade a lot if too many accesses are remote. These architecture suffer from low flexibility (memory transfers at fixed granularity which may cause false sharing, simple data movement patterns), due to their generic hardware implementation, although they sometimes provide specific APIs to perform bulk communications (a pattern that might not be easily recognized by the hardware).

On the opposite side, software-defined DSMs, like Stanford Legion [7] or Grappa [5] require heavy programmer intervention (annotating possibly distant load/store, or managing all data through their framework). They are more flexible due to the software implementation, being able to handle application-specific, structured data transfer, and to use programmer hints to place data. However these solutions usually replace C pointers with richer *fat pointers*, or with keys in abstracts spaces (tables indexed by tuples). This makes it difficult to interact with high performance C libraries using raw pointer structures, as the interface with the DSM may need copies. It may be difficult to control the data layout in memory, with respect to alignment and cache properties.

As a middle ground, language/compiler based DSM exists, like UPC [2] or Legion [7] (which has a compiled DSL). In these DSM, the compiler replaces load/stores by more complex calls which will determine whether this is a distant access or not. Most of these strategies will compile the provided language to a runtime target (GASNet library for UPC, Realm for Legion). The compiler will lift some of the programmer's burden, as it automatically generates calls to the runtime API and manages fat pointers, but it relies on fragile static analyses and remains limited in applicability.

3. Givy

The Givy runtime is a software DSM solution, which uses *raw pointers* to index memory. Using raw pointers means that any local distributed memory will contain a subset of existing GAS memory blocks, always placed at the same virtual addresses (see Figure 1).

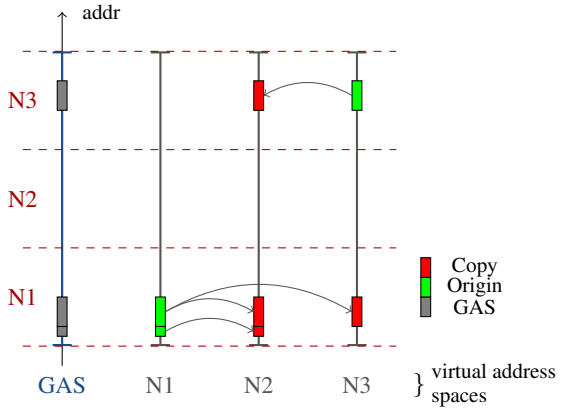

Figure 1. Example of GAS organization with 3 nodes (N1,N2,N3) and some GAS memory blocks (origin is the first copy that was created by `malloc()`)

The goal is to have the flexibility of a software DSM implementation, while allowing programmers to carefully control the memory layout of their data (doesn't change between nodes), and use

PPoPP '16 March 12-16, 2016, Barcelona, Spain

DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2851141.2851195

C libraries without having to convert data at the interface. Using real pointers, and no data layout conversion restricts Givy to distributed systems with the same architecture (types with the same memory layout, size, endianness, ...), but removes network overhead (no conversion needed). Efficient RDMA one-sided communications are a natural match as the remote memory address is always the same as the source address. We also require virtual memory support, as it is needed to freely setup the GAS memory area in each process address space. All of these constraints and properties fit well with MPSoC's high degree of parallelism, high-bandwidth network with RDMA support, and in the case of our target (the Kalray MPPA[1, 3]), virtual memory support.

The Givy DSM handles memory transfers between nodes by copying data at the exact same position every time. The copies are managed by a software *cache coherence protocol* (named *OWM*), which is similar to protocols used in hardware caches.

4. Allocator

Due to the choice of raw pointers as GAS keys, every addressable block of memory is technically in the GAS and could be copied to other nodes. As we need to be able to provide coherence metadata to every block, we restrict GAS behavior to memory blocks allocated by `malloc()` (it excludes stack and static variables). It is then the job of the `malloc()` implementation to provide the following properties:

1. avoid any address collision in the GAS;
2. retrieve coherence metadata of any block from its pointer.

The first property is a consequence of the raw pointer choice: in Givy, `malloc()` allocates in the GAS space. Thus we must guarantee that any `malloc()` call in one node will never return a block overlapping with any alive malloc'ed block from any node. In addition to that, we do not want `malloc()` to generate network traffic if possible. The solution is to split the GAS into logical node intervals, like in figure 1 (red lines are limits). A node can access any interval; however only the interval owner can allocate in its interval. This strategy does not waste memory because of the virtual memory support; the area just need to be big enough to fit any node memory, but in practice will be sparsely used. This also avoids any communication for allocation, and any node-local `malloc()`/`free()` generates no network communication.

All `malloc()` implementation act as a cache for memory blocks; they ask memory from the system in chunks called *pages*, and deliver blocks carved into these pages to achieve a small memory usage. So controlling the placement of memory to allocate in our interval means asking the system for pages in that interval. The only way to choose a placement is to use the `mmap(addr,size,MAP_FIXED)` system call; it asks the system to make available the $[addr, addr + size[$ memory block.

The current Givy `malloc()` implementation relies on this system call to only allocate memory in the right interval. As there is no fast way to get the state of virtual memory from the system (in order to drive the system call), we have to store this information ourselves in a structure called *page mapper*. The page mapper contains a list of unmapped sequence of pages of each size in the node interval; this allow to serve page request in the interval easily, by taking one of these sequence and giving it to `mmap()`.

The second required property is to find a block coherence metadata as quickly as possible (or to detect the absence of it in the case of a fully local block). As meta-data is accessed through page headers, we need to check if the page can be accessed before trying to access the header and metadata: it would cause a segmentation fault on failure. The page mapper also contains a concurrent hash-table which is used to quickly test if a page is mapped, solving the problem.

To finally carve allocated blocks from VM pages, the allocator uses standard high performance allocator techniques [6], including *sizeclass bins*, *thread local heaps*, *highly concurrent malloc/free*.

5. Performance

To the best of our knowledge, no allocator provides this specific interval property in addition to the fast testing. A GAS runtime name Myrmics [4] contains an allocator, which uses network traffic to allocate areas to nodes, allowing more flexibility at the cost of bandwidth.

We decided to test the performance of our allocator against current high performance allocators (Hoard, TcMalloc, JeMalloc, Streamflow, SFMalloc, SSMalloc). As all other allocator only supports shared memory, we compared HPC allocators to one node instance of the Givy allocator. The goal was to measure the possible computation and memory cost of providing the GAS properties. The tests were run on shared memory x86 hardware as porting other allocators to the MPPA would have been too complicated. Results confirmed that adding our property has no significant impact on performance. It also confirms our choice to replace the default `malloc()` implementation with the Givy allocator for every situation, simplifying the user interface to Givy.

6. Conclusion

We designed an allocator for a GAS runtime. It provides the required address space overlap prevention and highly efficient access to coherence meta-data, while still having node-local performance similar to other state-of-the-art memory allocators, both in memory usage and speed.

The Givy runtime is designed to have a task graph data-flow execution model in the future; it would allow to hide latency from the GAS transfers and load balance both computation and memory usage by making the scheduler aware of the allocator status.

References

[1] Kalray. http://www.kalrayinc.com.

[2] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. *Introduction to UPC and language specification*. Center for Computing Sciences, Institute for Defense Analyses, 1999.

[3] B. D. de Dinechin, P. G. de Massas, G. Lager, C. Léger, B. Orgogozo, J. Reybert, and T. Strudel. A distributed run-time environment for the kalray mppa®-256 integrated manycore processor. *Procedia Computer Science*, 18:1654–1663, 2013.

[4] S. Lyberis, P. Pratikakis, D. S. Nikolopoulos, M. Schulz, T. Gamblin, and B. R. de Supinski. The myrmics memory allocator: hierarchical, message-passing allocation for global address spaces. *ACM SIGPLAN Notices*, 47(11):15–24, 2013.

[5] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-tolerant software distributed shared memory. Technical report, Technical Report UW-CSE-14-05-03, Univeristy of Washington, 2014.

[6] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *Proceedings of the 5th international symposium on Memory management*, pages 84–94. ACM, 2006.

[7] S. Treichler, M. Bauer, and A. Aiken. Realm: An event-based low-level runtime for distributed memory architectures. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 263–276. ACM, 2014.