A Dynamic Memory Allocator for heterogeneous platforms

Maurizio Drocco*, Marco Aldinucci*,[1],
Massimo Torquati†,[2]

* *University of Torino, corso Svizzera 185, 10149 Torino, Italy*
† *University of Pisa, largo Pontecorvo 3, 56127 Pisa, Italy*

ABSTRACT

Modern computers are built upon heterogeneous multi-core/many cores architectures (e.g. GPGPU connected to multi-core CPU). Achieving peak performance on these architectures is hard and may require a substantial programming effort. High-level programming patterns, coupled with efficient low-level runtime supports, have been proposed to relieve the programmer from worrying about low-level details such as synchronisation of racing processes as well as those fine tunings needed to improve the overall performance. Among them are (parallel) dynamic memory allocation and effective exploitation of the memory hierarchy. The memory allocator is often a bottleneck that severely limits program scalability, robustness and portability on parallel systems.

In this work we introduce a novel memory allocator, based on the FastFlow's allocator and the recently proposed CUDA Unified Memory, which aims to efficiently integrate host and device memories into a unique dynamic-allocable memory space, accessible transparently by both host and device code.

KEYWORDS: Concurrency; multiprocessors; non-blocking synchronisation; skeleton programming; dynamic memory allocation

1 Introduction

Dynamic memory allocation enables programs to determine and request memory for its data structures at runtime. It is a necessary commodity for expressive parallel programming, typically used for implementing complex data structures (arrays, lists, trees) that need constant restructuring at runtime. Graph analysis and in-memory MapReduce problems are only two examples of large classes of problems that require a parallel application to work with such dynamic and irregular data structures. Unfortunately, it is a matter of fact that dynamic memory allocation is a relevant source of performance poorness in many parallel systems, since the most popular allocators (i.e. components that manages dynamic memory allocation) are not designed for supporting concurrent requests in an efficient way, thus resulting

[1]E-mail: {drocco,aldinuc}@di.unito.it
[2]E-mail: torquati@di.unipi.it

in very poor scalability with respect to e.g. the number of cores of a multi-core platform. This leads to flaws in terms of both execution time and memory wastage.

Moreover, hardware industry (in particular in the HPC world) is moving towards heterogeneous parallel platforms that rely on so-called *many-cores* accelerator devices (e.g. GPGPUs), supporting offloading of the execution of parallel *kernels* from the host (i.e. the CPU) to such dedicated devices. This trend poses many challenges for parallel programming models that should provide performance portability and seamless exploitation of such massively parallel platforms (e.g. NVidia K40 GPGPU has 2880 cores). In this scenario, efficient allocators for hybrid multi-core/many-cores platforms must deal with extremely high traffic of concurrent requests coming from a possibly huge number of processing elements. Moreover they should provide seamless integration of heterogeneous host/devices memories into a unified address space, equally accessible by both host and device code, in order to ease the code portability.

2 Dynamic Memory Allocators in the Many-core Era

The design of dynamic memory allocators has been extensively studied for both sequential and parallel systems. A dynamic memory allocator is a generic name given to the system that manages a heap of memory and handles, in a centralised manner, the memory allocation and deallocation requests coming from applications. Generally, such a system (1) keeps track of the memory blocks on its heap, including the allocation state of each block, (2) handles requests to allocate or free memory while constantly updating the state of the heap accordingly, and (3) communicates with the (main) operating system to alter the size of the heap when required.

Multi-core Allocators Doug Lea's DLmalloc is one of the most popular allocator and it is used in the GNU libc library. Although it features some nice features (e.g. coalescence of adjacent free blocks for minimising internal fragmentation), it is designed for a strict single-thread model, thus it scales poorly as cores are added to the system. Hoard allocator tackles this by using two types of heaps: a global heap and a per-process local heap. This approach limits the amount of conflicts that require locking, augmenting the concurrency, but local heaps lead to severe over-provisioning and low overall memory utilisation [SHGV14].

Many-cores Allocators Traditional parallel allocators fall short on many-cores SIMD/SIMT machines, since they tend to somehow serialise concurrent allocations. Xmalloc allocator for CUDA GPGPUs aims to work around this issue by gathering requests coming from threads of the same block. ScatterAlloc brings to GPGPUs the concept of local (per-block) heap from Hoard allocator. KMA is a OpenCL allocator based on lock-free accesses to a centralised queue of superblocks [SHGV14]. They are all efficient, but none of them deals with integration of host/device memories.

3 FastFlow

The FastFlow parallel programming environment was originally designed to support efficient streaming on cache-coherent multi-core platforms [ADKT14]. It is realised as a C++

(header-only) pattern-based parallel programming framework aimed at simplifying the development of applications for (shared-memory) multi-core and GPGPUs platforms. The key vision of FastFlow is that ease-of-development and runtime efficiency can both be achieved by raising the abstraction level of the design phase. It provides developers with a set of parallel programming patterns (aka *algorithmic skeletons*), in particular data-parallel patterns (such as *map*, *stencil*, *reduce* and their composition). High-level patterns are implemented on top of the Core patterns level, consisting of the composition/nesting of basic stream-parallel patterns (*farm*, *pipeline* and *feedback*).

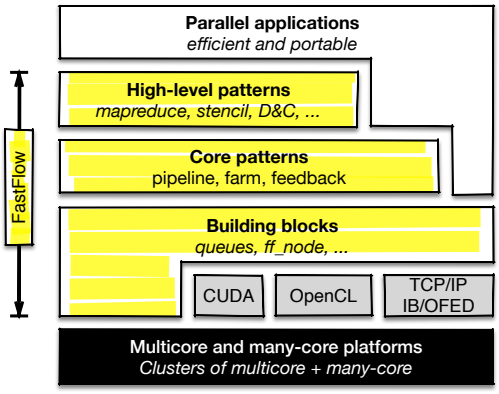

Figure 1: Architecture of FastFlow.

The latest extensions of the FastFlow framework, aimed at supporting heterogeneous platforms, make it possible to easily port the application to hybrid multicore/GPGPU systems by embedding CUDA or OpenCL business code. Data-parallel patterns can be run both on multi-cores and offloaded onto GPGPUs. In the latter case, the business code can include GPGPU-specific statements (i.e. CUDA or OpenCL statements).

At the bottom level (i.e. the Building blocks level) FastFlow CPU implementation of patterns is realised via non-blocking graphs of threads connected by way of lock-free channels [ADKT14], while the GPU implementation is realised by way of the OpenCL or CUDA bindings and offloading techniques. The framework also takes care of memory transfers between CPU host and GPGPU device. In general, different patterns can be mapped onto different sets of cores or accelerators, thus, in principle, using the full available power of the heterogeneous platform. The architecture of FastFlow framework is reported in Fig. 1.

FastFlow Allocator FastFlow's allocator, loosely based on the idea of the *slab* allocator. With slab allocation, memory chunks suitable to fit data objects of certain type or size are pre-allocated. The slab allocator keeps track of these chunks, known as caches, so that when a request to allocate memory for a data object of a certain type is received it can instantly satisfy the request with an already allocated slot. Destruction of the object, however, does not free up the memory, but only opens a slot which is put in the list of free slots by the slab allocator. The next call to allocate memory of the same size will return the now unused memory slot. This process eliminates the need to search for suitable memory space and alleviates memory fragmentation. In this context a slab is one or more contiguous pages in the memory containing pre-allocated memory chunks. Also, slab relies on the concept of *object re-use*, which is meant for keeping commonly used object in an initialised state, since the initialisation task might be the most expensive part of the process.

Under the performance perspective, FastFlow's allocator minimises concurrency by distributing the global state over per-thread local objects (the cloud-shaped FFalloc items in Fig. 2). Moreover it is *lock-less*, as it reduces the use of locks and mutual exclusion to the minimum required – basically when memory management is committed to the main underlying allocator (e.g. *malloc/free* operators in GNU libc). For every other contention it uses FastFlow's lock-free SPSC queues, which is a guarantee of efficiency and scalability.

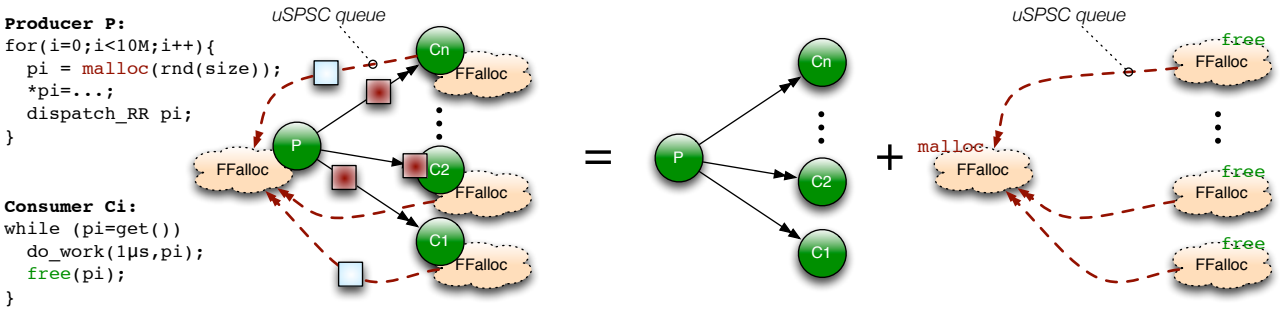

Figure 2: A possible allocation/deallocation workflow with FastFlow's allocator.

4 Future work

Let us consider, as a paradigmatic scenario, a FastFlow graph deployed onto a heterogeneous multi-core CPU host connected to a NVidia CUDA-capable GPGPU device. The recently proposed CUDA Unified Memory introduces automatic memory management, providing unified address space for both host and device code. However it works in one direction, in the sense it only allows device code to access host-allocated memory by managing all the memory transfers needed to keep the two memory copies synchronised.

Going further along the same direction, we propose a novel dynamic memory allocator, based on FastFlow's allocator and Unified Memory, which allows both host and device code to access memory (dynamically) allocated from a global shared space. In the initialisation phase, a global heap of memory is allocated via CUDA Unified Memory allocator. On top of FastFlow's allocator, each GPGPU-mapped node is coupled to n local objects – where n is the number of kernel threads for that node – that reside on the device and manage memory requests for single kernel threads, thus actually avoiding inter-thread concurrency on memory operations. Moreover, the object re-using mechanism discussed above limits requests for effective memory (de)allocation to the global heap, which is the only shared structure that must be protected (in a efficient lock-free fashion) from conflicting accesses. We remark that the proposed allocator fits particularly well in the pattern-based FastFlow programming framework, allowing mapping of different patterns onto different devices (e.g. GPGPU-Map CPU-Reduce) and seamless portability to different heterogeneous platforms.

References

[ADKT14] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Fastflow: high-level and efficient streaming on multi-core. In *Programming Multi-core and Many-core Computing Systems*, Parallel and Distributed Computing, chapter 13. Wiley, March 2014.

[SHGV14] Roy Spliet, Lee Howes, Benedict R. Gaster, and Ana Lucia Varbanescu. KMA: A dynamic memory manager for OpenCL. In *Proc. of Workshop on General Purpose Processing Using GPUs*, GPGPU-7, New York, NY, USA, 2014. ACM.