Noname manuscript No.
(will be inserted by the editor)

On Dynamic Memory Allocation in Sliding-Window Parallel Patterns for Streaming Analytics

M. Torquati · G. Mencagli · M. Drocco · M. Aldinucci · T. De Matteis · M. Danelutto

Received: date / Accepted: date

Abstract This work studies the issues related to dynamic memory management in *Data Stream Processing*, an emerging paradigm enabling the real-time processing of live data streams. In this paper we consider two streaming *parallel patterns* and we discuss different implementation variants related on how dynamic memory is managed. The results show that the standard mechanisms provided by modern C++ are not entirely adequate for maximizing the performance. Instead, the combined use of an efficient general-purpose memory allocator, a custom allocator optimized for the pattern considered and a custom variant of the C++ shared pointer mechanism, provides a performance improvement up to 16% on the best case.

Keywords Data Stream Processing, Modern C++, Dynamic Memory Allocation, Multicores, Stream Analytics, Parallel Patterns

1 Introduction

The C++ language is one of the most widely used languages for HPC and financial computations [5]. A consequence of using C++ is that memory management has to be managed explicitly even though, starting from C++11 standard, the language has been extended with a set of mechanisms and features (e.g., *unique and shared pointers*) that have considerably alleviated the burden of memory management. While from one side explicit memory management is desirable when predictable performance is needed, it does make programming much more challenging and error-prone. Moreover, operations

M. Torquati, G. Mencagli, T. De Matteis and M. Danelutto
Department of Computer Science, University of Pisa, Italy
E-mail: {torquati, mencagli, dematteis, marcod}@di.unipi.it

M. Drocco and M. Aldinucci
Department of Computer Science, University of Turin, Italy
E-mail: {drocco, aldinuc}@di.unito.it

such as memory allocation/deallocation are critical when high scalability and performance are required because they introduce extra synchronization points in the applications [12]. This paved the way to *general-purpose* allocators optimized for concurrency such as `Hoard` [3] and `TBB` scalable allocator [14]. Conversely, there is a widespread tendency to develop *custom* allocators built on top of general-purpose ones, with the goal of taking further advantage of the application-specific allocation pattern to increase performance.

This work studies how to provide both efficient and simplified dynamic memory management in the design of a parallel runtime for *Data Stream Processing* programs (DaSP) [2] on multicore platforms. The DaSP paradigm is characterized by the online processing of data streams that convey tuples of attributes (records, sensor readings) representing numerical or categorical information. The main goal of such streaming computations is to extract hidden knowledge from the most recent data (*sliding-window analytics* [9]) in order to assist decision-making processes.

Parallel patterns, expressly designed for real-time streaming analytics, can help in lowering the complexity of memory management in C++ programs [5] [6]. Our primary objective is to offer to the programmer a framework in which parallel patterns for DaSP computations can be easily instantiated by hiding all low-level implementation details such as tuple distribution and dynamic memory management. The pattern user must be relieved from dealing with explicit memory management (unless it is strictly needed by the business logic application code). All the issues related to efficient dynamic memory management should be handled at runtime level, leveraging the detailed knowledge of the particular allocation scheme directly available from the pattern.

To target performance- and memory-efficient parallel DaSP computations, the use of dynamic memory operations represents a critical task for several reasons: *i)* typical data structures employed in the DaSP domain (e.g., sliding windows [2]) are dynamic structures whose size can arbitrarily grow or shrink at run-time; *ii)* no realistic assumption on the stream length and speed can be made, and *iii)* each input tuple has a different lifetime and may be used by a different set of threads according to the parallel semantics of the pattern.

This work extends [6]. We studied the use of the standard C++ shared pointers for dealing with memory management in DaSP applications. This required to deeply change the internals of the reference runtime we have used (FastFlow [7][1]). The results show that this mechanism is not fully appropriate if the primary target is obtaining the maximum performance. Moreover, we study in more details the advantages of using general purpose allocators by extending the number of allocators tested. The results obtained confirm the ones obtained in our preliminary work: combining a pattern-tailored custom allocator with a scalable general-purpose allocator improves performance without increasing memory occupancy significantly. More precisely, at the best case our implementations perform about 16% better than the implementation of the patterns based exclusively on the standard `C++` allocator. Furthermore, the

[1] `http://mc-fastflow.sourceforge.net`.

best performance is very close (about 11% smaller) to the one achieved by a manually configured implementation, where memory is statically preallocated for each experimental scenario. Generally, this confirms that our approach is effective in reducing the run-time overhead by still leveraging dynamic memory to respond to the needs of each particular streaming workload.

The rest of the paper is organized as follows. Sect. 2 provides an overview of similar works. Sect. 3 describes the parallel patterns. Sect. 4 discusses the implementations issues and possible solutions. Sect. 5 evaluates the implementations, and Sect. 6 concludes the paper.

2 Related work

Stream processing engines have evolved into established solutions suitable to program applications that receive huge amount of transient data at great velocity. However, such solutions lack of high-level programming interfaces to easily instantiate parallel versions of compute-intensive functionalities in the streaming applications. Typically, applications are programmed by expressing arbitrary topologies of interconnected operators that perform common stream processing tasks [2] (e.g., filtering, aggregation, sliding-window analytics).

Some recent papers have investigated the definition of recurrent stream processing patterns. In [15], the authors proposed an interesting C++ library providing parallel patterns such as filtering, pipeline, stream-reduce and farm. The patterns are very general and suitable for many common streaming computations. Conversely, it does not provide (in its current version) specific patterns for the Data Stream Processing domain. A fundamental concept like *sliding windows* has to be introduced and programmed by hand. Although written in C++, the library does not provide any specific support for dynamic memory management. This from one hand is a valuable feature that simplifies the programming effort (e.g., avoiding memory leaks), from the other hand pattern specific memory optimizations is left to the programmer. From this viewpoint, our approach represents a clear departure from this vision.

The work presented in [17] proposed a set of patterns for elastic stream processing, where the patterns provide a uniform way to solve elasticity and fault tolerance by overcoming the distinction usually applied of traditional frameworks. Although interesting and complementary to our work, the proposed patterns are not oriented to the high-level programmer, but they mainly represent solutions for the design of the run-time support of streaming frameworks in order to design elasticity and fault-tolerance mechanisms easily within the Mesos cluster infrastructure.

Patterns for real-time streaming analytics have been studied in [13]. As in this paper, the authors argue that designing applications *from scratch* is an approach neither viable nor effective to develop systems with reduced time-to-maket. The patterns proposed are recurrent sub-topologies of operators implemented in Apache Storm. Since the framework has a front-end implemented in Java, memory allocation issues are outside the responsibility of the pattern

implementation and are trasparently solved within the JVM by relying on the standard garbage collector.

The research in [18] described a window-oblivious implementation of stream join operators. The problem studied by the authors is to dynamically configure the window size according to the data stream characteristics in such a way as to obtain acceptable query results. To this end, the data structures supporting the operator internal state may be dynamically resized based on real-time monitored data. The approach lacks of generality, since it is designed for window joins and assumes only sequential processing of complex joins.

Architecture-oriented optimizations for stream processing have been developed for task scheduling strategies in [11], where each task is properly allocated in the memory closer to the assigned processing core in charge of computing it. This approach, is beneficial only for highly NUMA machines and not of general applicability.

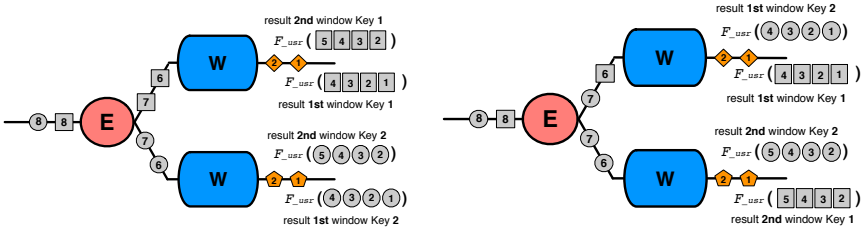
Outside the stream processing domain, there has been a long research endevour for designing efficient general-purpose dynamic memory allocators. Examples are `Hoard` [3], `TBB` scalable allocator [14] and `Jemalloc` [10]. Their goal is to enable better scalability by using thread-private heaps and free lists in order to significantly cut down synchronization costs. They are *general-purpose* allocators aimed at intercepting the use of standard allocation routines in the code by replacing them without needing to recompile the program.

In addition, the programmer can develop *custom allocators* optimized for a particular use. Notable examples are the ones used in standard benchmarks for parallel architectures, as the `197.parser` application of the SPEC suite. Interestingly, in the work published in [16] the authors gave a brief review about custom allocators developed for *ad hoc* purposes, and presented and optimization methodology based on genetic algorithms to optimize allocation policies in order to accommodate the needs of each specific application. Following this rationale, our work leverages on the approach of using custom allocators to lower the dynamic memory overhead in specific parallel patterns for sliding-window computations, a representative class of streaming queries.

3 Parallel patterns for streaming analytics

Most of the existing stream processing frameworks [2] allow the programmer to express computations using abstractions modeling directed acyclic graphs, where nodes are operators processing data tuples exchanged through the arcs of the graph. Trivial parallelism can be expressed in case of *stateless* computations, by replicating the same operator multiple times (*task-farm*) or by processing distinct data tuples in parallel by multiple sequential stages (*parallel pipeline*). Meanwhile, *stateful* operators represent a more challenging case from the parallelization standpoint. The notion of state is a sort of internal knowledge consisting in the history of the past data tuples seen by the operator, and can be represented as a *sliding window* of the last received tuples [2].

Parallel patterns for sliding-window queries have been presented in [9]. Two of them will be studied in this paper. The first, called *Key Partitioning* (KP), can be applied in case of multiplexed data streams, where each data tuple includes a *key* attribute. All the tuples with the same key logically belong to the same group. For example, a stateful operator processing financial transactions (tuples) computes the average price of the last 100 transactions having the same stock symbol (an identifier of a publicly traded stock) and produces a new update every new 10 tuples trading the same stock (sliding factor). The KP pattern exploits parallelism between window computations on different stock symbols. The schema is the one of a standard task-farm, see Fig. 1a, where an emitter (denoted by `E`) receives the input tuples and routes them to the worker (an operator replica denoted by `W`) assigned to the corresponding key attribute. Each worker is responsible to maintain the last 100 tuples for each of its assigned stock symbols, and to start the processing each time a new window is complete.



(a) Key partitioning. (b) Window farming.

Fig. 1: Parallel patterns with window length of 4 tuples and slide of 1 tuple. Tuples of the first and second key are represented by a square and a circle.

Parallelism in the KP pattern is limited by the number of keys, since all the windows of the same key are computed serially by the same worker. A pattern overcoming this limitation is called *Window Farming* (WF) [9]. In most of the window semantics, it is possible to statically identify the set of consecutive windows that will contain a certain data tuple. Given $w \geq 1$ and $s \geq 1$ the window length and the sliding factor, the tuple x_i will belong to the windows with identifier j such that $j \geq \lceil (i+w)/s \rceil + 1$ and $j \leq \lceil i/s \rceil$. This can be generalized to time-based windows, where the tuple identifier is a timestamp and w and s are expressed in time units. The idea of the WF pattern is to pre-assign consecutive windows to the workers, e.g., according to a round-robin assignment. For each tuple, the emitter determines the windows containing that tuple, and schedules it to the workers assigned to the corresponding windows. Fig. 1b exemplifies this concept with windows with $w = 4$ and $s = 1$. The tuple x_2 belongs to the first and the second window that are assigned to the first and the second worker. Therefore, the tuple is routed to both the workers that execute the assigned windows using an internal sliding factor of $s \times n_w$, where n_w is the number of workers (the parallelism degree).

In this paper we are not interested in comparing the two patterns and understanding their applicability. For example, it is known that the KP pattern suffers from load imbalance [8,9,5] in case of skewed distributions of keys (i.e. when few keys are much more frequent than the others). Instead, in this work we will focus on how to implement efficiently the two patterns of multicores by focusing on dynamic memory allocation mechanisms and strategies.

3.1 High-level pattern interface

The two patterns have been implemented in FastFlow [7] as high-level parallel patterns. High-level parallel patterns solve specific yet recurrent problems in an application domain, and their implementation is developed on top of more general core patterns like *task-farm* and *parallel pipeline* and their compositions. The pattern `C++` interface allows the programmer to:

- indicate as a template parameter the type of the input tuple (`input_t`), the output result type (`output_t`), and the type of the window (`window_t`);
- provide the user function $\texttt{F}_{\texttt{usr}}$ to be applied at each window activation as an input parameter of the pattern constructor;
- provide two other functions: $\texttt{F}_{\texttt{in}}$ takes a reference to an input tuple and returns a pair of values in which the first one is the unique identifier of the tuple (or its timestamp) and the second is the value of the key attribute; $\texttt{F}_{\texttt{out}}$ does the same for an output result.

Fig. 2 exemplifies the instantiation of the `WF` pattern in FastFlow. The application is a parallel pipeline of three stages. The first and the last one are sequential operators in charge of producing the data stream by reading the tuples from a TCP/IP socket (*Generator*) and to collect the results (*Consumer*). The second stage is the `WF` pattern, which is internally implemented as a task-farm with two threads for the emitter and the collector and $\texttt{nw} \geq 1$ identical worker threads.

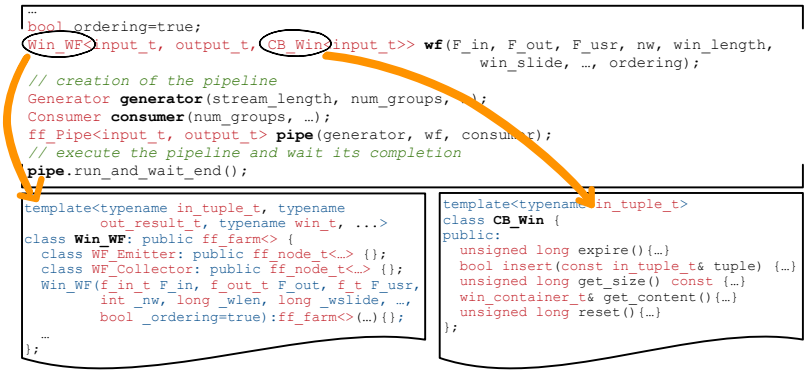


Fig. 2: Example of pattern instantiation in FastFlow using `C++` objects.

This example instantiates the pattern for *count-based* sliding windows (implemented by the `CB_Win` data type), where the w and s parameters are expressed in number of tuples. The window type is passed as a template parameter to the WF pattern, and must implement proper methods: *i) insert*, to add a tuple to the window container (actually, it is copied in the WF pattern like shown in Fig. 2, or either copied or moved in the KP pattern); *ii) expire*, to evict all the expired tuples; *iii) reset*, to empty the window (denoted by `C`).

The pattern is implemented by the `Win_WF` class. The emitter and collector threads are defined by two inner classes that extend the FastFlow single-thread operator abstraction `ff_node_t`. The emitter implements the distribution logic shown in Fig. 1b by calling the $\texttt{F}_{in}$ function to extract the key and the identifier from each input tuple. The workers are implemented by the `Win_Seq` class extending `ff_node_t`. Each worker maintains a window data structure for each key, adds the received tuples to the corresponding window, checks whether the window is triggered and starts the computation of the function $\texttt{F}_{\texttt{usr}}$ that reads all the tuples in the window container. The window results are finally transmitted to the collector which takes care of the result ordering.

4 Patterns implementation and optimizations

In the FastFlow runtime the threads implementing high-level parallel patterns cooperate by exchanging memory pointers to shared data structures through push/pop operations on lock-free queues [1]. A critical aspect to be addressed from the performance viewpoint is the dynamic memory allocation of the tuples received at a high speed from the stream. To reduce the programming complexity, our solution is designed with the goal of hiding the way in which tuples and results are allocated within the pattern.

Once obtained a memory pointer from the input queue, a worker copies (or move) the tuple into the corresponding window by calling the insert method. The designer of the window data structure (e.g., it can be the user himself) does not need to know how the tuple has been originally allocated by the previous stages of the application. Similarly, the user function $\texttt{F}_{\texttt{usr}}$ takes the result as an input argument passed by reference. Therefore, the user is unaware of how the result data structure has been allocated, and his function code only fills its content by reading the actual window container.

The runtime system is responsible for dealing with the allocation/deallocation of input tuples and output results using strategies enabling fine-grained scalable parallelism. The general picture of our approach is sketched in Fig. 3 by referring to the example previously shown in Fig. 2.

The programmer is in charge of writing the code of the Generator operator, and to allocate the input tuples received from the socket. Typically, since a stream is a large sequence of elements, tuples are stored in contiguous virtual memory areas or allocated in batches of several tuples. Similarly, the user expects that the results produced by the pattern are allocated using a desired allocator, which will be used to free the memory space in the final stage (the

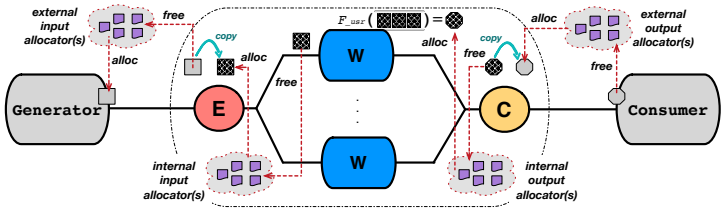


Fig. 3: General picture of a parallel pattern working on sliding-window computations.

Consumer in the figure). We call these allocators *input/output external allocators* from the pattern viewpoint (they are provided as input parameters of the pattern constructor).

The tricky part of the pattern implementation is that input tuples, allocated outside the pattern, must be used within the pattern and deallocated by the run-time support when they have been used by all the workers in charge of processing them. This raises two important problems that need to be solved from the performance perspective:

- *ownership passing*: in our implementation an entity having a pointer to a shared data structure in its active scope has the ownership of that data. Both for the KP and WF patterns, the ownership of the input tuples must be transferred from the emitter to the workers, and this must be efficiently implemented. Only the entity owning the data can destroy it and deallocate the utilized dynamic memory;
- *memory allocation*: input streams of high-frequency data streaming applications usually have very fast rates (hundreds of thousands or even millions of tuples per second). This implies that dynamic memory allocation must be properly implemented in such a way as to avoid fine-grained synchronizations among threads.

The rest of this section will be devoted to analyzing these two problems and to designing proper countermeasures.

4.1 Ownership passing methods

The KP and WF patterns have different semantics for how the ownership of an input tuple is transferred from the emitter to the workers. In the case of the KP, the unique ownership of the tuple is transferred from the emitter to one worker, the one assigned to the tuple's key attribute. In fact, once routed to the worker, the emitter does not need the tuple anymore, while the worker is responsible to copy (or move) it in the corresponding window data structure and then to destroy it by deallocating the corresponding storage.

The WF pattern represents a more interesting case because each tuple can be forwarded to multiple workers (statically known). The emitter transfers the shared ownership of the tuple to the destination workers that use the

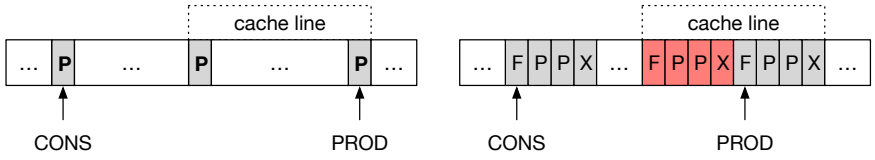
tuple and perform its destruction/deallocation when all of them have done with it. Therefore, some form of reference counting is needed to understand when a worker can destroy a tuple and release its storage. In the current FastFlow release (v2.1.3), lock-free queues work with raw pointers (single machine word, i.e. 4 or 8 bytes). There are two basic ways to implement the necessary semantics:

- the first is a sort of workaround, in which the emitter makes some copies of the input tuple, one for each worker that expects to receive it. A raw pointer to a distinct copy is then passed to each worker by the FastFlow runtime, and the worker can update the corresponding window and safely destroy/deallocate its copy. Despite its simplicity, this approach pays the extra-overhead of making multiple copies serially in the emitter;
- the second approach uses a *custom wrapper* struct built around a tuple and containing: *i)* a raw pointer to the tuple; *ii)* an atomic counter (`atomic<size_t>`) initialized to the number of workers that will receive that tuple. The FastFlow runtime multicasts the pointer to the same wrapper instance to all the workers that should receive the pointed tuple. Each worker uses the tuple and decreases the couter atomically. The worker who last decrements the counter, deallocates both the tuple and its wrapper.

Another possibility is to try to exploit the C++11 features offered by the family of smart pointers and more specifically by *shared pointers* (`shared_ptr`), which implement the same concept of our wrapper but with a more powerful semantics. A shared pointer object maintains a pointer to a shared data (resource) and a pointer to a struct (called control block) containing a reference counter. When a shared pointer is destroyed, the counter is automatically decremented and the resource is destroyed and deleted when the counter reaches zero. When a shared pointer is copied, the counter is incremented.

4.1.1 Passing `C++` shared pointers through lock-free queues

The main technical problem to be solved in the FastFlow runtime is to adapt the lock-free queue implementation to accept shared pointers instead of raw pointers without loosing the lock-free property. In the native implementation, the synchronization between the producer and consumer threads is guaranteed by the write atomicity on single memory words, which is provided by all the relevant architectures. Since queue slots are raw pointers (whose size is one word in most architectures), the consumer can "see" an update by the producer to a queue slot by polling on the slot (until the slot is not null), as shown in Fig. 4a. In contrast, a shared pointer requires two words to be stored, thus the simple polling approach is not correct in this case: if the consumer sees an update to one word of the slot, it does not know if the other words have been updated or not. More precisely, the polling approach would be correct if one could guarantee that a particular word is always the last to be written during a multi-word update; obviously this assumption can not be made in the case of a system-provided implementation.



(a) Queue of raw pointers.

(b) Queue of shared pointers.

Fig. 4: Synchronization flags are denoted by F, whereas P represents a shared pointer word and X are padding words.

Fig. 4b illustrates the solution we adopted to overcome the issue. Each slot of the queue is composed of several fields, not just one word as in the standard implementation. The first field is a word representing a boolean flag, two words are used for the shared pointer, and the rest of the bytes of the slot are used as a padding space to properly align the data to the cache line size. On machines with L1 cache lines of size 64 bytes, two queue slots can be stored per cache line. Each push operation first updates the shared pointer words (in any order), then sets the synchronization flag. It is important to observe that on some architectures a memory fence is needed in order to guarantee such first-then relation. With this protocol, when the producer sees an update to the synchronization flag, it can safely read the respective shared pointer.

With this slight modification of the FastFlow queue, shared pointers can be used to exchange the ownership of a shared data structure between threads. In order to use the new implementation, we have adapted the FastFlow API in order to accept shared pointers in addition to standard raw pointers in the definition of the sequential FastFlow nodes (extending the `ff_node_t` class) like the emitter, worker and collector of our pattern implementation.

4.2 Custom memory allocation of streaming data

The second problem mentioned at the beginning of this section concerns the efficiency of the dynamic memory allocation mechanism, which is affected by two main issues:

– the pattern implementation must be aware of how the input tuples have been allocated in the previous stages of the application (i.e. by the use of external allocators as shown in Fig. 3). For example, if the storage of each tuple has been created separately, single tuples must be deallocated in the workers when they can be destroyed. Instead, if tuples are allocated in larger blocks, the pattern runtime should keep track of the destroyed tuples and deallocate the block storage when all its tuples have been destroyed. This task depends on the external allocation strategy outside the pattern control, and it is difficult to be managed inside the pattern in a general way. Moreover, the external allocator could be not efficient to achieve acceptable performance;

– in streaming patterns the tuples allocation/deallocation activities are executed very frequently in general, according to the input rate of the stream. The same problem affects the allocation of output results within the pattern, though less severely because just one output result is produced per window activation (one every s tuples in case of count-based windows).

To solve both the issues, the pattern runtime uses *internal input/output allocators* optimized to manage small objects exchanged according to the producer-consumer paradigm representative of the streaming context. Once a tuple is received, the emitter copies it by requesting a memory area to the internal allocator, and passes a pointer/shared pointer to the worker(s). The original tuple is deallocated by the emitter using the external input allocator provided during the pattern construction. Symmetrically, the worker requests a memory area for allocating a result object to the internal output allocator, executes the $\texttt{F}_{\texttt{usr}}$ function and passes a pointer/shared pointer to the result to the collector, which in turn transmits it to the final stage by eventually copying it using the external allocator that the user expects to use. This whole idea is depicted in Fig. 3.

In the next section we will describe the FastFlow custom allocator that targets the use cases of our patterns

4.2.1 FastFlow allocator

The FastFlow framework provides a custom allocator optimized for the allocation of small objects used in a producer-consumer way. It is based on the idea of *slab* allocator [4]. A slab is a contiguous region of memory split into equal-size chunks plus a header containing information about how many of those chunks are in use. Virtual memory is acquired and released per slab using a general-purpose allocator (by default `libc` malloc/free calls). The allocator is implemented as a `C++` class that provides malloc-like and free-like methods.

A set of slabs, for a given object size, are pre-allocated in a *local cache*, so that when a request to allocate memory for an object of that size is received, it can be immediately served by using a free chunk. A request to release an object just produces a new item in the free chunk list without really releasing virtual memory. Only when all the chunks of a slab have been released, the slab memory is returned to the general-purpose allocator. This simple process eliminates the need to search for suitable memory space thus increasing the performance, reduces memory fragmentation and increases memory re-use [4].

The base FastFlow allocator has been implemented with the idea that only one thread can allocate memory (*mem-producer*) and one or more threads can release memory (*mem-consumer*(s)). This is the typical scenario of *task-farm* and *pipeline* computations. For implementing these simple scenarios, the FastFlow allocator internally uses lock-free Single-Producer Single-Consumer queues [1] (i.e. the same data structure used in FastFlow to implement the memory channels between pipeline stages). In particular, there is a queue for each *mem-consumer*, where the *mem-consumer* is actually the only producer

for the queue. The generic *mem-consumer* thread notifies the presence of a new free object to the *mem-producer* thread by inserting the memory pointer to be released into its own queue. When the *mem-producer* threads needs a chunk of memory, the allocator first checks the presence of a free chunk in its internal cache, if no chunk is available it tries to pop a new chunk from its input queues, otherwise it allocates a new slab for that object size and initializes it. The cost of initializing a new slab is the most expensive operation for the allocator, fortunately, this cost may be paid only until the system reaches the steady state. After that point, no new virtual memory is allocated or reclaimed.

In FastFlow, this implementation has been used as a building block for a more general allocator that has no constraint in the number of producers/-consumers. This notwithstanding, in this work we considered the base version which is the most efficient although it is the less user-friendly. This last point is not a real issue in our case, because the base FastFlow allocator can be easily adapted to our `KP` and `WF` patterns, and most of all, the memory management of tuples/results is completely transparent to the end-user.

5 Experiments

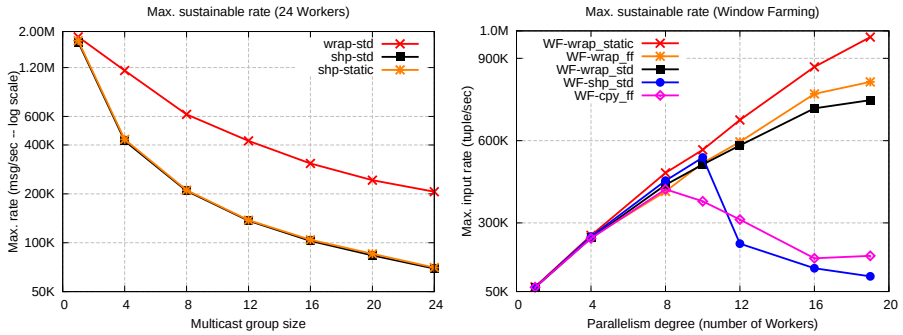
The architecture used for the experiments is a dual-socket Intel Xeon Ivy Bridge running at 2.40GHz with 24 cores (12 per socket). Each core has 32KB private L1, 256KB private L2 and 30MB shared L3. The OS is Linux 3.14.49 x86_64 (CentOS 7.1). We use `gcc 4.8.5` with the optimization flag `-O3`.

We first study the performance of using the C++ shared pointer mechanism for passing tuple pointers to workers, as we discussed in Sect. 4.1.1. For this purpose, we used a simple FastFlow synthetic benchmark in which a generator thread generates a stream of shared pointers towards a farm pattern having 24 workers. Each shared pointer points to a data token of 64 bytes. The farm scheduler multicasts each input stream element to a subset of the workers; we call the cardinality of the subset *multicast group size*.

Then, we study the high-frequency trading application described in [8]. The generator receives a stream of financial *quotes* represented as a tuple of 64 bytes, which are processed by a parallel operator *algotrader*. The operator maintains a window of size $w = 1,000$ and slide $s = 25$ for each stock symbol[2]. At each window activation the user function aggregates quotes with a resolution interval of 1 ms and applies a model aimed at estimating the future price of the stock symbol. The kernel uses the Levenberg-Marquardt regression algorithm implemented by the `C++` library `lmfit`[3]. This application is fine-grained (the fitting procedure takes about 300 μsec), therefore it is suited to bring out the differences between the implementation variants. With respect to the tests presented in [6], we have globally optimized the number of allocations/deallocations performed by just allocating both the tuple and its wrapper with a single operation.

[2] These parameters can be changed, the values used are typical ones [8].

[3] `http://apps.jcns.fz-juelich.de/lmfit`.



(a) C++ shared pointer benchmark. (b) WF pattern

Fig. 5: a) Comparing custom wrappers and C++ shared pointers performance by varying the number of destination workers. b) Maximum input rate sustained by different variants of the `WF` pattern.

To avoid memory explosion, we forced the FastFlow runtime to use bounded queues for all patterns. In particular, the size of the farm's queues is set to 264K slots so that they are big enough to sustain high input rate avoiding to block too frequently the emitter thread for lack of queue space. All experiments have been repeated 20 times. They exhibit a small variance, therefore error bars are not reported.

5.1 Custom wrappers vs. C++ shared pointers

Figure 5a reports the maximum rate measured running the benchmark test. The farm pattern has a fixed number of worker threads (24) and the test is executed by varying the *multicast group size*. We studied three different scenarios: a) `wrap-std` where the standard FastFlow API based on raw pointer queues is used and each message pointer is *wrapped* in a structure containing the data pointer, the allocator that has been used to allocate the entire message (wrapper and data), and an atomic counter initialized to the multicast group size; b) `shp-std` in which a standard C++ shared pointer is used together with the FastFlow API based on shared pointer queues, the shared pointer is allocated by using the call `make_shared` which uses the standard C++ allocator; c) `shp-static` that is the same of case b) but the shared pointer is allocated by using the call `allocate_shared` which allows to provide a custom allocator for allocating the shared pointer. For this test, we implemented a custom allocator that pre-allocates upfront all memory needed without releasing it till the end of the execution. We call this kind of allocator `static allocator`.

Not surprisingly, the performance of the API based on shared pointers are worse than those obtained with the raw pointers API. The extra overhead is not due to the dynamic memory allocator used, in fact by using the static allocator (`shp-static`), the performance gain is minimal ranging between 1%

and 2%. Instead, we have identified at least two major factors introducing extra overhead in substituting raw with shared pointers: 1) cache efficiency, 2) shared pointer copies. The former can be easily visualized in Fig. 4. A raw queue slot occupies one memory word (i.e., 64 bits), therefore a cache line fits 8 slots assuming the widespread 64 bytes L1 cache lines. Conversely, a cache line hosts 2 shared pointer queue slots, thus resulting in more cache traffic for a given memory working set. The second factor emerges when considering a pervasive operation in the FastFlow runtime: copying pointers. For instance, each push/pop operation requires copying a pointer. Copying a raw pointer has negligible cost since it amounts to a plain read/write operation in the worst case scenario (i.e., non-cached memory). On the other hand, copying a shared pointer requires an *atomic* update of the memory region holding pointer meta-data, that are shared by all the threads accessing the pointer.

This notwithstanding, the use of shared pointer significantly simplifies the code avoiding the explicit management of memory. In this work, our standpoint is that of the runtime system developer, therefore our main aim is to offer the most efficient solution without exposing low-level implementation details. From now on, the custom wrapper solution will be our default one.

5.2 Evaluating the implementation variants

We want to measure the maximum input rate that a pattern implementation sustains without being a bottleneck. Input tuples belong to $2,836$ uniformly distributed stock symbols. In all experiments, each thread is pinned on a dedicated core. Hyperthreading is not effective in this application, and considering the presence of the additional threads (i.e., generator, filter, emitter, collector and consumer stages), the maximum number of workers that can be used is 19. We do not report here the sustained input rate for the `KP` pattern because, as discussed in [6], all different `KP` parallel variants have very similar behavior with a maximum scalability close to the ideal one (about 18 with 19 workers). For this pattern, the memory management is less challenging and the gain derived from the usage of the internal FastFlow allocators with respect to the standard one is minimal and about 1.5%.

Fig. 5b shows the highest input rate sustained with a different number of workers by the `WF` patterns. We considered five variants for the patter: `WF-wrap_static` is a hand-made version in which we have statically preallocated memory space to avoid dynamic allocations; `WF-wrap_ff` uses the internal allocation scheme depicted in Fig. 3 using a custom wrapper for the tuple; `WF-wrap_std` uses the custom wrapper and `libC++` new/delete operations; `WF-shp_std` uses the shared pointer version and the standard allocator (`make_shared`); and `WF-cpy_ff` in which the farm emitter executes a copy of each tuple for each destination worker and uses the FastFlow allocator. The `WF-wrap_static` versions assume the length of the input stream is known, which is unrealistic in general. It will be used as a baseline for the comparison.

The WF pattern stresses the memory hierarchy more with respect to the KP one. With the used window length and slide, each tuple is transmitted on average to all the workers, and all of them perform simultaneously a copy in the corresponding window. With an input rate of λ tuples/sec, WF performs on average $n\lambda$ copies per second while KP requires only λ copies. Furthermore, the WF pattern requires an order preserving collector, which is not needed in KP. This introduces extra overhead to maintain the priority queues with additional results copies. This justifies why the scalability is not optimal in this case (14.5 with 19 workers) even with the static allocator.

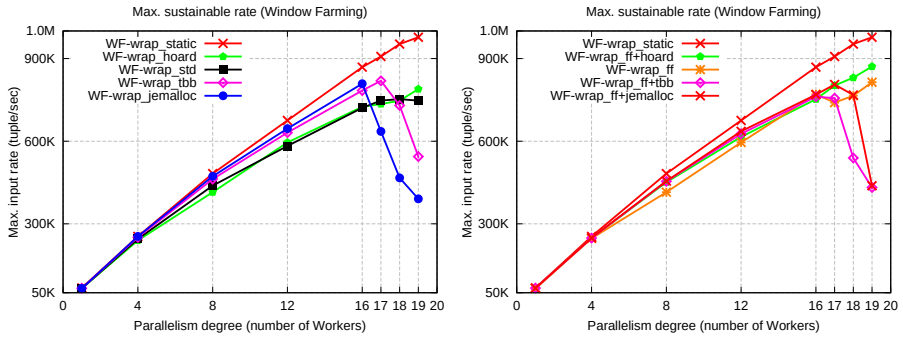
The worst variant is the one where the farm emitter executes a copy of each tuple to the destination workers. While this keeps the deallocation easier and less costly, it makes the emitter service time proportional to the number of copies. This is the reason why the curve in the figure drops for parallelism degrees greater than 8 (WF-cpy_ff). The shared pointers version scales better up to 10 (WF-shp_std) workers and then drops down more quickly due to the higher costs of managing the shared pointers. The custom wrapper solution, used with the FastFlow internal allocator, provides the best results. The distance with the static version is about 17% and the gain derived from the usage of the FastFlow allocators on the standard allocator is about 10%.

5.3 Use of existing allocators

The idea to use custom allocators for the producer-consumer scheme is effective in optimizing the scheduling/release of input tuples and output results, and in recycling their memory areas. In contrast, general-purpose allocators are application-wise and can further speedup the execution of dynamic memory allocations everywhere in the application code (e.g., inside the function F_{usr}). To cover the possible optimization of dynamic memory allocations in all the parts of the application, we studied the combined use of the custom allocator on top of general-purpose ones specifically targeted for parallel processing. In particular, we considered the use of Hoard [3] (version 3.11), TBB scalable allocator [14] (version shipped with Intel Parallel Studio 2017 – 2017.0.098) and Jemalloc allocator [10] (version 4.4.0), which are widely used in multicore-based parallel programming. All of them can be used as drop-in replacement for standard allocation calls [4].

Figure 6a shows the results only for the WF pattern for space reasons. By enabling Jemalloc and TBB allocators we obtain a performance increase of about 10% up to 16 and 17 workers, but then the performance drop down. Instead, the Hoard allocator provides only a marginal improvements with respect to the standard allocator, but it keeps scaling up to 19 workers providing a 6% better performance. All external allocators tested intercept and replace all the dynamic memory allocation calls in the pattern, i.e., also the ones in the lmfit function called by the workers. Thus, the idea is to use together

[4] By setting the LD_PRELOAD environment variable to force loading the allocator library before libc.



(a) Rate with different external allocators. (b) Combining FF with other allocators.

Fig. 6: Effect of using different external allocators in the `WF` implementation.

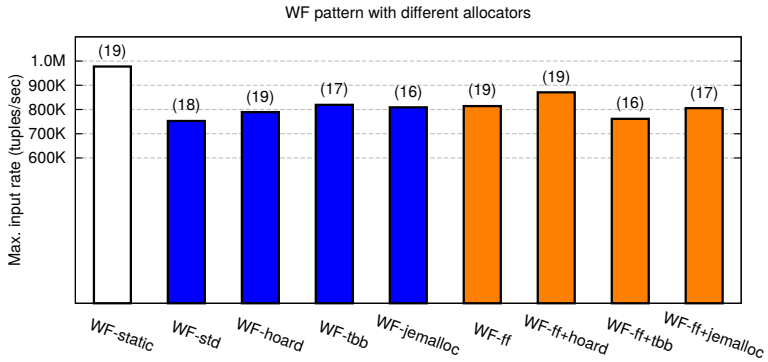


Fig. 7: Maximum rate sustained by the `WF` pattern with different memory allocators. On top of the bar is reported the number of workers needed.

both an internal allocator tailored for the inter-thread interaction scheme of the pattern and an external general purpose allocators for optimizing user's functions. Figure 6b shows the results obtained combining the FastFlow allocator with some external allocators. The outcome of this approach is a 16% improvement on the standard version obtained combining the `Hoard` allocator and the FastFlow allocator. The loss of this solution with respect to the reference static one is only 11%. The `Jemalloc` and `TBB` allocators do not produce the same effects as the `Hoard` allocator. Although they provide very good performance up to 16 and 17 workers respectively, when used together with the FastFlow allocator the performance decreases though less quickly. This issue is probably due to the interferences between the two allocators that are particularly relevant when the number of worker threads is high.

To allow a direct comparison of the best results obtained on the reference static version, Fig. 7 summarizes the maximum performance achieved by the WF pattern when using different memory allocators and their combined use with the FastFlow allocator.

Finally, we have evaluated the different versions also regarding memory occupancy. Table 1 shows an experiment in which the application is fed by a real financial dataset from the NASDAQ market[5], composed of about 50M quotes generated with an accelerated (50×) variable rate. We tested the `WF` pattern in different variants, executed with 8 workers (the minimum to sustain the peak rate). The data reported in the table are the memory consumption in the steady-state phase. As can be seen, the differences among allocators are minimal and not significant on modern multicore platforms. The performance benefits are achieved without increasing memory consumption significantly.

	S	H	T	J	FF+S	FF+H	FF+T	FF+J
Memory (MB)	2498	2524	2253	2320	2643	3093	2390	2480

Table 1: Memory used by the `WF` pattern with different allocators (S:standard, H:Hoard, T:TBB, J:Jemalloc, FF:FastFlow). The `WF` pattern uses 8 workers.

6 Conclusions and future work

This paper discusses dynamic memory management effects in C++ DaSP applications for computing sliding-window analytics in real-time. We studied different design choices proposing optimizations techniques for stream parallel patterns implemented by using the FastFlow runtime framework. The experiments show that the combined use of a custom allocator tailored for the patterns together with a scalable general-purpose allocator and a custom variant of the C++ shared pointer mechanism allows achieving a performance improvement on the version using just the standard allocator and the C++ shared pointer mechanism.

Our work deserves future extensions. The approach and the presented custom allocator are tailored for two parallel patterns that represent suitable ways to parallelize a wide class of streaming problems. However, other parallel patterns can be identified, typically with less general applicability but potentially able to achieve better performance: e.g., the patterns based on the *pane-based* model enabling reuse of partial computations between subsequent windows [9]. We plan to study how our custom allocation scheme can be adapted in order to be beneficial in those cases, also by studying other streaming workload scenarios in addition to the financial application described in this paper.

Acknowledgements This work has been partially supported by the H2020 RePhrase (ICT-2014-1) project.

References

1. Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., Torquati, M.: An efficient unbounded lock-free queue for multi-core systems. In: Proc. of 18th Intl. Euro-Par 2012 Parallel Processing, *LNCS*, vol. 7484, pp. 662–673. Springer (2012)

[5] Daily trades and quotes of 30 Oct 2014 downloadable at `http://www.nyxdata.com`.

2. Andrade, H., Gedik, B., Turaga, D.: Fundamentals of Stream Processing. Cambridge University Press (2014). Cambridge Books Online
3. Berger, E.D., McKinley, K.S., Blumofe, R.D., Wilson, P.R.: Hoard: A scalable memory allocator for multithreaded applications. SIGOPS Oper. Syst. Rev. **34**(5), 117–128 (2000)
4. Bonwick, J.: The slab allocator: An object-caching kernel memory allocator. In: USENIX summer, vol. 16. Boston, MA, USA (1994)
5. Danelutto, M., Matteis, T.D., Mencagli, G., Torquati, M.: Parallelizing high-frequency trading applications by using c++11 attributes. In: Trustcom/BigDataSE/ISPA, 2015 IEEE, vol. 3, pp. 140–147 (2015)
6. Danelutto, M., Mencagli, G., Torquati, M.: Efficient dynamic memory allocation in data stream processing programs. In: UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld, 2016 IEEE, pp. 1181–1188 (2016)
7. Danelutto, M., Torquati, M.: Structured parallel programming with "core" fastflow. In: V. Zsók, Z. Horváth, L. Csató (eds.) Central European Functional Programming School, *LNCS*, vol. 8606, pp. 29–75. Springer (2015)
8. De Matteis, T., Mencagli, G.: Keep calm and react with foresight: Strategies for low-latency and energy-efficient elastic data stream processing. In: Proceedings of the 21th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016. ACM, New York, NY, USA (2016)
9. De Matteis, T., Mencagli, G.: Parallel patterns for window-based stateful operators on data streams: an algorithmic skeleton approach. International Journal of Parallel Programming (to appear) (2016)
10. Evans, J.: Scalable memory allocation using jemalloc (2011). Facebook notes
11. Falt, Z., Kruliš, M., Bednárek, D., Yaghob, J., Zavoral, F.: Towards efficient locality aware parallel data stream processing **21**(6), 816–841 (2015)
12. Michael, M.M.: Scalable lock-free dynamic memory allocation. SIGPLAN Not. **39**(6), 35–46 (2004)
13. Perera, S., Suhothayan, S.: Solution patterns for realtime streaming analytics. In: Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, DEBS '15, pp. 247–255. ACM, New York, NY, USA (2015). DOI 10.1145/2675743.2774214. URL `http://doi.acm.org/10.1145/2675743.2774214`
14. Reinders, J.: Intel Threading Building Blocks, first edn. O'Reilly & Associates, Inc., Sebastopol, CA, USA (2007)
15. del Rio Astorga, D., Dolz, M.F., Sanchez, L.M., Blas, J.G., García, J.D.: A C++ Generic Parallel Pattern Interface for Stream Processing, pp. 74–87. Springer International Publishing, Cham (2016). DOI 10.1007/978-3-319-49583-5_5. URL `http://dx.doi.org/10.1007/978-3-319-49583-5_5`
16. Risco-Martín, J.L., Colmenar, J.M., Hidalgo, J.I., Lanchares, J., Díaz, J.: A methodology to automatically optimize dynamic memory managers applying grammatical evolution. Journal of Systems and Software **91**, 109 – 123 (2014)
17. Sattler, K.U., Beier, F.: Towards elastic stream processing: Patterns and infrastructure. In: G. Cormode, K. Yi, A. Deligiannakis, M.N. Garofalakis (eds.) BD3@VLDB, *CEUR Workshop Proceedings*, vol. 1018, pp. 49–54. CEUR-WS.org (2013). URL `http://dblp.uni-trier.de/db/conf/vldb/bd32013.html#SattlerB13`
18. Wu, J., Tan, K.L., Zhou, Y.: Window-oblivious join: A data-driven memory management scheme for stream join. In: 19th International Conference on Scientific and Statistical Database Management (SSDBM 2007), pp. 21–21 (2007). DOI 10.1109/SSDBM.2007.43