Efficient Dynamic Memory Allocation in Data Stream Processing Programs

Marco Danelutto, Gabriele Mencagli and Massimo Torquati
Department of Computer Science, University of Pisa, Italy
Largo B. Pontecorvo 3, I-56127, Pisa, Italy
Email: {danelutto, mencagli, torquati}@di.unipi.it

***Abstract*—*Data Stream Processing* is a paradigm enabling the real-time processing of live data streams coming from sources like sensors, financial tickers and social media. The history of the stream is often maintained in *sliding windows* and analyzed to produce timely notifications to the users. A challenging issue in the development of parallel implementations of such computations is efficient dynamic memory allocation. In this paper we study two *parallel patterns* for sliding-window computations and we discuss different implementation variants related to how dynamic memory is managed. The results show that the combined use of an efficient general-purpose memory allocator, and of a custom allocator for the pattern considered, results in significant performance optimizations.**

***Index Terms*—Data Stream Processing, Dynamic Memory Allocation**

I. Introduction

Dynamic memory allocation is one of the most basic features in standard sequential programming. In parallel programming it may become a critical issue when looking for high scalability and performance, because the bookkeeping of standard allocation/deallocation procedures (e.g., `libc` malloc/free) introduces global synchronization [1]. This paved the way to *general-purpose* allocators optimized for concurrency such as `Hoard` [2] and `TBB` scalable allocator [3]. Complementary, there is a widespread tendency to develop *custom* allocators built on top of general-purpose ones, with the goal to take further advantage of the application-specific allocation pattern to increase performance.

This work shows how to provide an efficient dynamic memory management in the design of a parallel runtime for *Data Stream Processing* programs (DaSP) [4] on multicores. 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. Typical DaSP computations inspect the recent stream history to detect critical patterns and trigger notifications/alerts to the users. Instead of buffering the entire stream, which is often impractical, tuples are temporarily buffered in *sliding windows* [5] defined as a fixed period of time (*time-based*) or as a fixed number of tuples (*count-based*), e.g., the computation is applied on the tuples received in the last 5 seconds and the results updated every 0.1 seconds.

To target performance- and memory-efficient parallel DaSP computations, the use of dynamic memory allocation/deallocation operations is a critical task for various reasons: *i)* windows are dynamic structures whose size can arbitrarily grow or shrink at run-time, thus they need sophisticated dynamic memory mechanisms; *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.

In this work our standpoint is that of the runtime system developer, i.e. our goal is to offer to the high-level programmer a framework in which parallel patterns for sliding-window streaming computations can be easily instantiated by hiding a set of important low-level details like tuple distribution and window management policies, and dynamic memory management. The latter is critical for performance, because each input tuple can be used concurrently by a different subset of threads. Therefore, the pattern implementation must be able to efficiently detect when the allocated areas for tuples can be safely released or recycled to increase performance and reduce synchronization. Our goal is to relieve the programmer from this burden by solving it within the runtime, leveraging the detailed knowledge of the specific allocation scheme directly available from the pattern.

To the best of our knowledge, no similar work exists in the domain of Data Stream Processing. Our specific contributions are the following:

- we describe two parallel patterns developed in FastFlow [6][1], a `C++` parallel framework for multi-/many-cores. The description focuses on the differences between patterns and how the programmer can instantiate them;
- we go on the details of memory allocation by describing optimizations within the patterns and the use of the custom FastFlow allocator, specialized for the *producer-consumer* allocation scheme;
- we evaluate our implementation choices in a real-world high-frequency trading application fed by synthetic and real datasets.

The results confirm our intuitions and show that the use of the custom allocator on top of a scalable general-purpose allocator actually improves performance without increasing memory occupancy.

The rest of the paper is organized as follows. Sect. II provides an overview of similar works. Sect. III describes the parallel patterns. Sect. IV introduces the dynamic memory

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

allocation issues and presents optimizations. Sect. V evaluates the implementations on a real-world application, and Sect. VI concludes the paper.

II. Related Work

Over the years, several dynamic memory allocators have been written to speedup performance of parallel programs, especially in those cases where frequent requests for small objects are generated by the application threads, i.e. when the standard allocation calls are notoriously inefficient. Widely used allocators are `Hoard` [2] and `TBB` scalable allocator [3]. Others are `LFMalloc` [7], `Streamflow` [8], `TCMalloc` [9] and `XMalloc` [10]. Their rationale is to enable better scalability by using thread-private heaps and free lists in order to significantly cut down synchronization costs. Only when a request cannot be served by accessing private data, global data structures are inspected by using fine-grained locks. All of them 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 special use. Notable examples are the ones used in standard benchmarks for parallel architectures, like the `197.parser` application of the SPEC suite. Interesting is the work published in [11], in which the authors give a brief review about custom allocators developed for *ad hoc* purposes, and present and optimization methodology based on genetic algorithms to optimize allocator policies in order to accommodate the needs of each specific application.

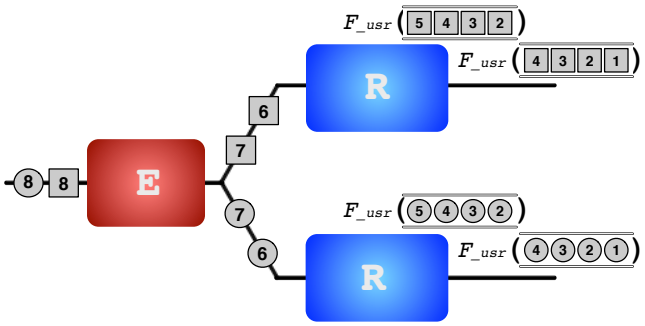
No paper before this one has tried to understand the performance issues of dynamic memory allocation in DaSP programs. To the best of our knowledge, the closest work to ours is [12], where task scheduling strategies in DaSP have been studied and their performance analyzed in highly NUMA machines, to allocate tasks optimizing memory bandwidth. Although this aspect can be studied in a future extension of our work, here we focus more on *pattern-oriented optimizations*, which are independent from the underlying architecture used.

III. Parallel Patterns

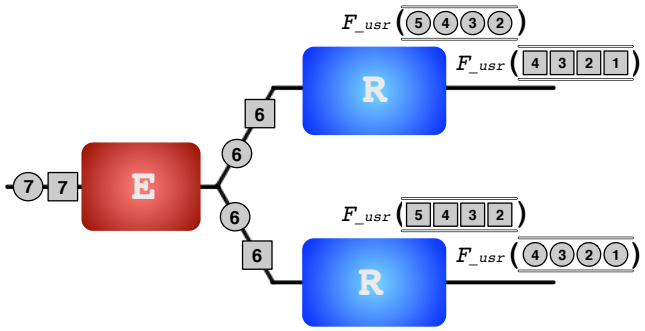
Data stream processing applications [4] are represented as data-flow graphs of operators receiving input streams and producing output sequences. Non-trivial parallelism is often needed by stateful operators that maintain an internal knowledge updated each time a new tuple is received and processed. Usually, the notion of internal state consists in a temporary sliding window of the most recent tuples. In this paper we will focus on two parallel patterns for window-based streaming computations: the *Key Partitioning* (`KP`) and *Window Farming* patterns (`WF`) introduced in [5].

The first pattern is used to parallelize partitioned-stateful operators [4], i.e. operators that work on input streams that convey tuples logically belonging to different substreams according to the value of a partitioning attribute of the tuples (as in a group-by relational operator). The operator maintains an independent state for each substream, updated each time we receive a new tuple belonging to such substream. For simplicity, we use the term *group* as a synonym of substream. Parallelism can be easily exploited for such class of operators, because the computation on tuples belonging to different groups is independent and can be executed in parallel. Therefore, the parallel pattern consists in `nw` ≥ 1 *replicas* (denoted by $\mathtt{R}_1, \ldots \mathtt{R}_n$) also called *workers* that perform the internal processing logic (window update, and triggering of a user function). An *emitter* entity (`E`) distributes tuples to the replicas in such a way as to exploit the parallel semantics of the pattern. In the `KP` pattern, *windows belonging to different groups can be executed in parallel, while all the windows in the same group are processed sequentially by the same replica*. Therefore, the emitter is responsible to schedule all the tuples within the same group to the same replica, i.e. by assigning to each group a worker using a hash function, see Fig. 1a.

The second pattern is based on the observation that consecutive windows in the same group may have some tuples in common. It is possible to identify the windows (numbered starting from 1) to which a tuple belongs. Let $w \geq 1$ and $s \geq 1$ be the *window length* and the *slide* parameters both expressed in number of tuples. As stated in [13], a tuple with identifier $i \geq 1$ belongs to all the windows with identifier j such that $j \geq \lceil (i+w)/s \rceil + 1$ and $j \leq \lceil i/s \rceil$. This can



(a) Key partitioning.



(b) Window farming.

Fig. 1: Scheduling behavior of the patterns with count-based windows, $w = 4$ and $s = 1$. The tuples of the first group are identified with a sequence identifier inside a square box, the ones in the second group with circles.

be generalized to time-based windows, where the identifier is a timestamp attribute. Therefore, by finding a way to assign windows to replicas (e.g., in a round-robin fashion), it is possible to distribute tuples to the replicas in order *to execute in parallel also windows belonging to the same group*. The emitter now multicasts each tuple to a specifically identified subset of the replicas, i.e. the ones assigned to the windows to which the tuple belongs (see Fig. 1b). Each replica uses an inner slide parameter equal to $\texttt{nw} \times s$.

The `WF` pattern has a wider applicability than `KP`. In fact, its scalability is independent from the frequency distribution of the groups, while `KP` is unable to achieve good performance if we have few groups (e.g., less than the number of workers) or when some groups are much more frequent than the others (the replicas assigned to the most frequent groups receive more tuples than the other replicas). However, the `WF` implementation is more challenging due to the more complex distribution (multicast). Moreover, while `KP` produces results in the same group in order, this is not generally true for `WF`. If the ordering of results in the same group is needed, the `WF` implementation must take care of it.

In this paper we will study specific solutions for the issues that arise in the implementation of the two patterns, with special attention to `WF` which, as studied in our past work [5], [14], is potentially able to achieve good performance in a broader range of application configurations than `KP` (e.g., number of groups and their frequency distributions).

A. FastFlow Implementation

The two patterns have been implemented in FastFlow [6] as high-level parallel patterns. High-level parallel patterns solve specific yet recurrent problems in an application domain, and their implementation in FastFlow is usually developed on top of more general core patterns like *farm* and *pipeline*, or their compositions. The `C++` interface of the patterns allows the programmer to:

- indicate as template parameters the type of the input tuple (`input_t`), the output result (`output_t`), and the type of the window (`window_t`);
- provide the user function to be applied at each window activation as an input parameter of the pattern constructor. Let $\texttt{F}_{\texttt{usr}}$ be the function provided by the user;
- 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 identifier of the group; the second $\texttt{F}_{\texttt{out}}$ does the same for an output result. Essentially, these functions are in charge of extracting specific fields from tuples/results whose types are specified as template parameters of the pattern.

Fig. 2 exemplifies the instantiation of the `WF` pattern, which has been used inside a *pipeline* pattern of three stages. The first and the last stages (*Generator* and *Consumer*) are objects extending the `ff_node_t` class, i.e. the FastFlow single-threaded node abstraction. The *Generator* receives the input tuples from a TCP/IP socket and forwards them to the second stage. The *Consumer* writes the results into a file. The second stage is the `WF` pattern, which internally has an emitter and a collector thread and $\texttt{nw} \geq 1$ *worker* threads for the replicas.

This example instantiates the pattern with a count-based sliding-window data type (`CB_Win`) based on a circular buffer. This type has to define a set of pure virtual methods declared in the `I_Window` class: *i)* an *insert* method to add a tuple to the window container; *ii)* an *expire* method to evict all the expired tuples; *iii)* a *reset* method to empty the window.

The `Win_WF` class implements the pattern. The emitter and collector threads are defined by two inner classes that extend `ff_node_t`. The emitter implements the distribution logic shown in Fig. 1b by calling the $\texttt{F}_{in}$ function to extract the group and the identifier of each input tuple. The collector can be executed in two modes: 1) by preserving the ordering of results in the same group; 2) by producing results as soon as they arrive at the collector. The workers are implemented by the `Win_Seq` class which extends `ff_node_t`. Its logic is the following: each tuple received by the emitter is inserted into the corresponding window; if the window is triggered, the worker allocates the result data structure, calls the function $\texttt{F}_{\texttt{usr}}$ and passes the result pointer to the collector.

IV. Issues and Solutions

The implementation of our parallel patterns has been designed in order to deal with two aspects:

- the implementation must encapsulate and abstract the way in which tuples and results are internally allocated. The same tuple can be used by more than one worker, and it must be safely deallocated when it is no longer necessary without hampering performance by introducing rigid synchronization points;
- the user provides the functions $\texttt{F}_{\texttt{usr}}$, $\texttt{F}_{\texttt{in}}$ and $\texttt{F}_{\texttt{out}}$. He can also use a customized window implementation, e.g., that exploits some indexes like trees.

Input tuples are inserted into the corresponding window by calling the insert method, which takes the tuple as an input argument passed by a constant reference. Therefore, the designer of the window data structure (he can be the user himself) can copy the input tuple into the window container without knowing how it has been originally allocated by the previous stages of the application, thus without dealing with its deallocation. In contrast, the programmer has a complete visibility of the window data structure used (it can be a user-defined implementation or a more general one provided by our pattern library), where any sort of allocator can be used to manage the internal container according to the window specifications (e.g., time-based and count-based windows have different memory requirements). 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 the content by reading the actual window container passed as input parameter.

In conclusion, the runtime system of parallel patterns is responsible to deal with the allocation/deallocation of input

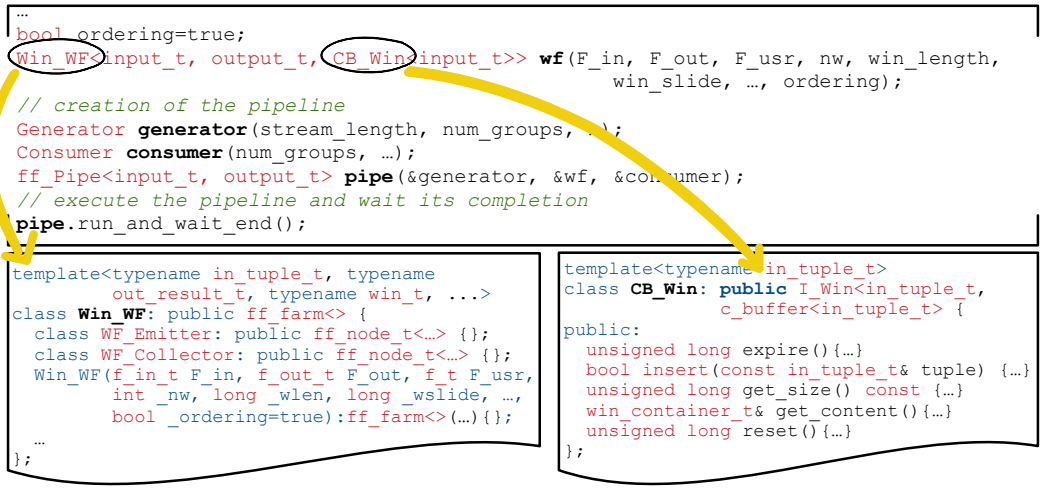


Fig. 2: Example of pattern instantiation in FastFlow using C++ objects.

tuples and output results using strategies suitable for parallelism. To this end, we use an intra-pattern allocation scheme transparent to the user, exemplified in Fig. 3. In this example the tuples are received from the network by a Generator stage, where other parts of the application may have used any allocator to dynamically create the memory space for the input tuples. Typically, since a stream is a sequence of elements, tuples are stored in contiguous memory areas and allocated in batches of several tuples. Similarly, the user expects that the results produced by the pattern are allocated using a desired allocator instance, which will be used to free the memory space in the afterward stages (the 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).

As outlined in Fig. 3, the emitter thread (both in the KP and WF patterns) is in charge of: *i)* receiving the input tuples, *ii)* distributing them (by passing a pointer according to the FastFlow model) to a subset of workers according to the pattern semantics, and *iii)* deallocating the memory of the original tuples when they can safely deallocated. This task is not straightforward because a tuple can be destroyed only when all the receiving workers have used it for their processing. Then, the memory area can be released when all the tuples in the same batch have been destroyed. We perform this task in a very efficient way by using proper *internal input/output allocators* optimized to manage small objects like single tuples. Once a tuple is received, the emitter thread copies it by requesting a memory area to the internal allocator, and passes a pointer to the worker(s). The original tuples are released by the emitter (using the external input allocator) when a consecutive batch has been received and all the tuples have been distributed to the workers.

Symmetrically, after a window triggering, the worker requests a memory area for a new result to the internal output allocator, executes the F_{usr} function and passes the result object (through a pointer) to the collector thread, which in turn transmits it to the backward stage by eventually copying it using the external allocator that the user expects to use.

This scheme allows the patterns to exploit internal and possibly lock-free allocators specialized for the specific inter-thread interaction scheme. This is a typical *producer-consumer paradigm*, in which one thread allocates memory areas for data structures of the same size (tuples and results), and one or more threads deallocate them. In the next section we will describe the FastFlow special-purpose allocator that targets this use case.

A. The 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 [15]. 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 [15].

The base FastFlow allocator has been implemented with the idea that only one thread can allocate memory (*mem-*

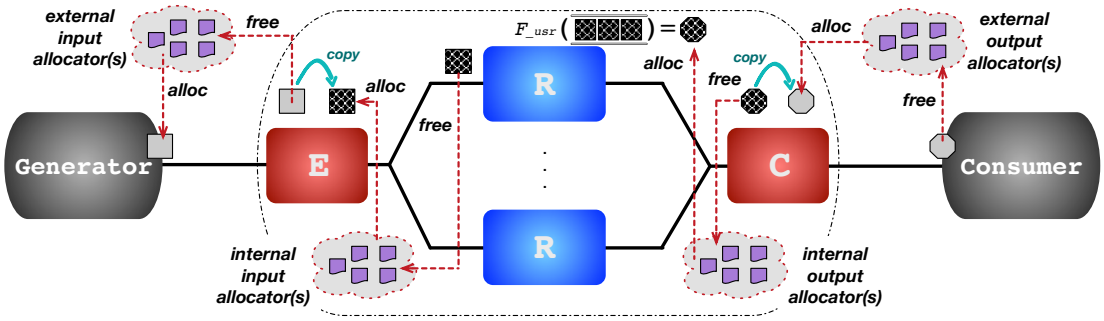


Fig. 3: Allocation scheme of a parallel pattern for sliding-window streaming operators.

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 [16] (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 by the system.

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.

B. Alternative Design Choices

Several optimizations can be designed for enhancing the patterns. They focus on: *a)* the way in which the emitter distributes input tuples; *b)* which internal allocators are used and how. Concerning point *a)*, we study two ways to distribute the tuples that also affect the allocation scheme: (`CPY`) the emitter makes several copies of the input tuple, one for each worker that needs it; (`WRAP`) the emitter distributes a pointer to the same tuple to all the workers that require it.

These approaches coincide in the `KP` pattern because each tuple is *always* transmitted to one worker. In contrast, in the `WF` pattern each tuple can be transmitted to more than one destination. While in the `CPY` version each worker can safely deallocate the received tuples once inserted into the corresponding window (they are private copies), in the `WRAP` version *each input tuple must be deallocated by the last worker that operates on it*. Without assumptions on the relative speeds of the workers it is impossible to know statically that worker. Our solution consists in wrapping each tuple in a `wrapper_t<input_t>` structure with three fields: *i)* a pointer to the input tuple; *ii)* a pointer to the allocator that has been used to allocate the tuple (and its wrapper); *iii)* an atomic counter initialized to the number of workers that receive that tuple (*reference counter*).

This solution allows to make the worker threads independent from which internal allocators are used by the emitter. For a similar reason this solution is adopted also to wrap the output results transmitted to the collector. The wrapper contains the counter to free the tuple's space. While in the `KP` pattern this field is not really important, in the `WF` pattern each worker atomically decreases the counter, and the worker that finds the counter equal to 1 deallocates the tuple/wrapper pair using the allocator whose pointer is in the wrapper itself.

Furthermore, the distribution in the `WF` pattern can be optimized. As originally presented in [13], [5], workers can be assigned to a set of consecutive windows grouped in a *bundle*, i.e. each tuple is transmitted to all the workers assigned to the bundles that the tuple belongs. *The goal is to reduce the number of workers than need the same tuple*. Let $b \geq 1$ be the bundle size in terms of windows, each tuple belongs to at most $\lceil n_b \rceil$ bundles, where $n_b = (w + (b-1) \times s)/(b \times s)$. As expected, the bigger the bundle the smaller the number of bundles that contain the same tuple. This option can be enabled by passing a proper flag to the pattern constructor, and the potential benefits will be discussed in Sect. V.

Point *b)* is related to how the internal allocators are used. Two opposite possibilities can be identified: (`STD`) all the allocation/deallocation operations are calls to `libc++` new/delete primitives; (`FF`) the implementation uses several instances of the FastFlow allocator. For the inputs, the emitter can use a single FastFlow allocator instance to allocate the input tuples; alternatively, a different instance can be used for each pair emitter-worker. For the output results, each worker has its own allocator that will be used by the collector to free the corresponding memory. In the rest of this paper we will

evaluate these design choices with a real-world data stream processing application.

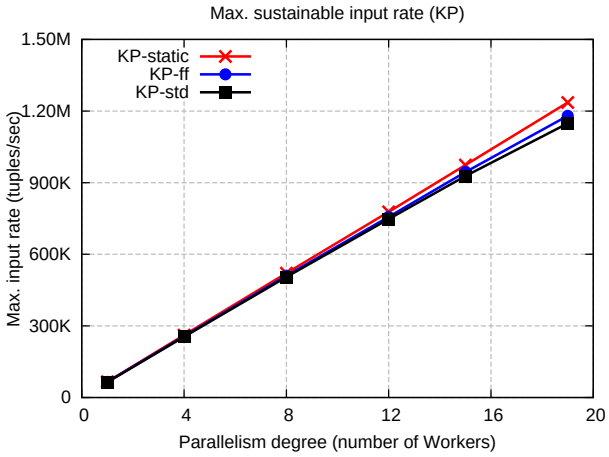
V. Experiments

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 shipped with CentOS 7.1. We use `gcc 4.8.3` with the optimization flag `-O3`.

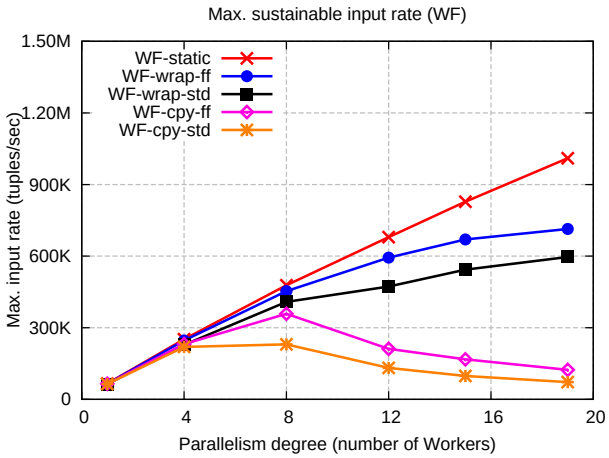
We study the high-frequency trading application described in [17]. The generator receives a stream of financial *quotes* represented as a tuple of 64 bytes, which are processed by a parallel operator *algotrader*. The operator maintains a window of size $w = 1,000$ and slide $s = 25$ for each stock symbol[2]. 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. The experiments have been repeated 20 times. They exhibit a small variance, therefore error bars are not reported.

A. Evaluating the Implementation Variants

We measure the maximum input rate that a pattern implementation sustains without being a bottleneck. Input tuples belong to $2,836$ uniformly distributed stock symbols. This scenario allows a fair comparison, otherwise the `KP` suffers from load unbalance with skewed distributions. Fig. 4 shows the highest input rate sustained with a different number of workers. We pin each thread on a dedicated core. Hyper-threading is not effective in this application, i.e. the maximum number of workers is 19 (the application uses other 5 threads).



(a) Key partitioning.



(b) Window farming.

Fig. 4: Maximum input rate sustained by different variants of the parallel patterns.

Fig. 4a compares three variants of the `KP` pattern: `KP-ff` uses the internal allocation scheme depicted in Fig. 3; `KP-std` uses `libc++` new/delete; `KP-static` is a hand-made version in which we have statically preallocated memory space to avoid dynamic allocations. The latter assumes to know the length of the input stream, which is unrealistic in general. It will be used as a baseline for the comparison. The results show that the `KP` pattern has a near ideal behavior in all the versions, with `KP-std` slightly worse than the others. The scalability with 19 workers is 18.8 with `KP-static` and 17.5 with `KP-std`. The gain derived from usage of internal FastFlow allocators is about 7%.

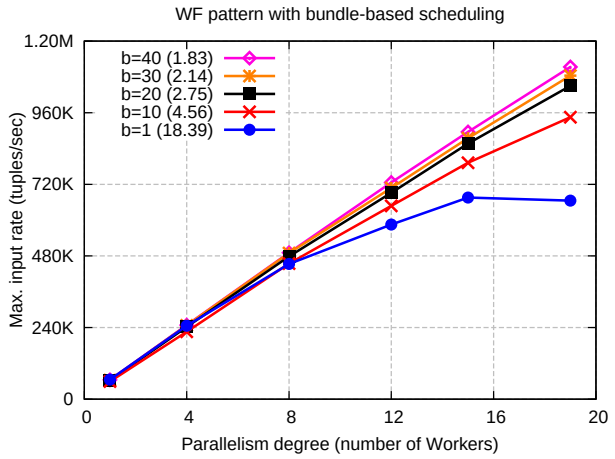
The results with the `WF` pattern are lower (Fig. 4b). The reason is that `WF` stresses more the memory hierarchy. With the used window length and slide, each tuple is transmitted to all the workers on average, and all of them perform simultaneously a copy in the corresponding window. With an input rate 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 implementation, which is not needed in `KP`. This introduces extra overhead to maintain the priority queues with additional results copies. This justifies why, also with the static version, the scalability is not optimal (15.35 with 19 workers).

The worst variants use the `CPY` emitter that executes a copy of each tuple to the destination workers. While this keeps the deallocation easier, 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 4 (8) in the `WF-cpy-std` (`WF-cpy-ff`) version. The use of FastFlow allocators alleviates the problem. The `WRAP` solution, used with internal allocators, provides the best results. The distance with the static version is about 20%. From now on, the `WRAP` solution will be our default one.

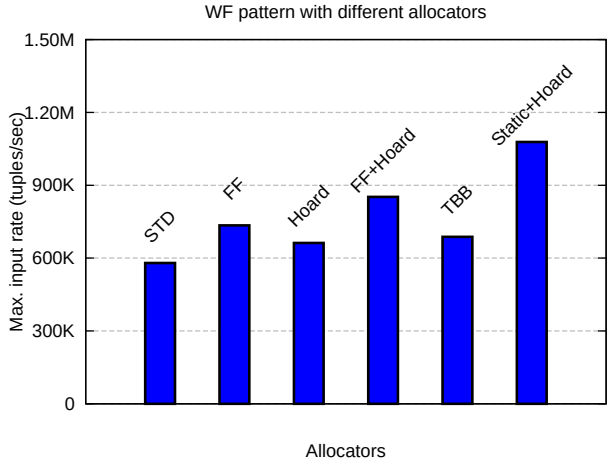
Fig. 5a shows the effect of the bundle-based distribution described in Sect. IV-B. This optimization allows to reduce the threads synchronizing on the atomic counters for deallocating

[2]These parameters can be changed, the values used are typical ones [17].
[3]http://apps.jcns.fz-juelich.de/lmfit.

tuples. Furthermore, the emitter service time is lower since the pointer to the input tuple must be forwarded to a smaller set of workers. We show the maximum rate sustained with different bundle sizes (in terms of consecutive windows). Between parenthesis in the legend we report the average number of workers operating on the same tuple. As expected, the greater the bundle the better the performance. However, too large bundles can be not always acceptable because they increase the latency between results [5].



(a) Rate with different bundle sizes.



(b) Rate with different allocators.

Fig. 5: Effect of the bundle-based distribution in the WF pattern, and impact of different allocators in the WF implementation (without bundling, i.e. $b = 1$).

B. 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 study the combined used of the custom allocator on top of general-purpose ones specifically targeted for parallel processing. In particular, we focus on the use of Hoard [2] and the TBB scalable allocator [3], which are widely used in multicore-based parallel programming.

Hoard is used to reduce the heap contention in multi-threaded applications. It is a drop-in replacement for standard allocation calls [4]. Fig. 5b shows the results only for the WF pattern for space reasons. By enabling Hoard we achieve a performance increase of 13% with respect to using standard libc++ new/delete. This result is lower than the one using internal FastFlow allocators, which are specialized for the inter-thread interaction scheme of the pattern. However, Hoard intercepts and replaces all the dynamic memory allocation calls, i.e. also the ones in the lmfit function called by the workers. Thus, an idea is to use both the allocators in conjunction. The outcome of this solution is a 47% improvement with respect to the standard version, and only a 18% loss of performance than the static one. The reason for this is twofold: first Hoard takes advantage of the NUMA management of memory areas to exploit better memory bandwidth, second without the FastFlow allocator Hoard manages a large set of small allocation/deallocation requests (tuples, results) introducing more overhead; in the FF+Hoard case instead, Hoard manages only requests from the FastFlow allocator, which are less and for bigger memory chunks.

We have also evaluated the use of the TBB scalable allocator [3] (version 4.4 release 2). We obtain a performance benefit of 18% compared with the standard new/delete. The performance is 3.7% better than using Hoard, however lower than using Hoard combined with the FastFlow allocator. We have tried to combine TBB with FastFlow on top of it. Nevertheless, no performance improvement has been achieved. This aspect needs further investigations in our future work.

In the KP pattern the difference between allocators is marginal. By using the bundling optimization ($b = 40$) and the configuration FF+Hoard, the performance of the WF pattern is close to the best results of KP (only 11% lower). This result is valuable because WF can be efficiently used also in cases with very skewed distributions of groups (stock symbols in this case), where KP fails.

Finally, we have evaluated the different versions in terms of memory occupancy. Fig. 6 shows an experiment in which the application is fed by a real financial dataset from the NASDAQ market[5], composed of 50M of quotes generated with an accelerated (50×) variable rate. We study the WF pattern in different variants, executed with 8 workers (the minimum to sustain the peak rate).

The virtual memory in use (VmSize) increases over time because the workers allocate a new window each time they encounter a stock symbol for the first time, and in the dataset

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

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

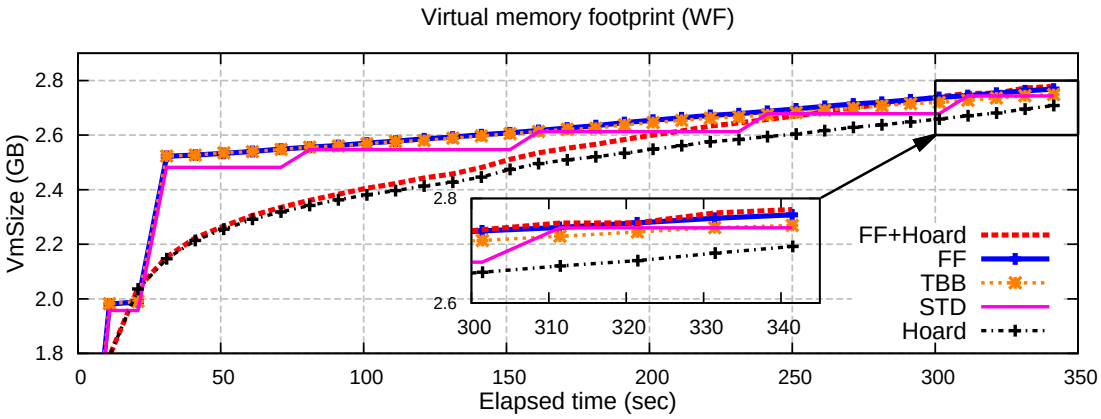


Fig. 6: Virtual memory footprint (`VmSize`) of the `WF` pattern with different allocators.

new stock symbols are received over the entire execution. The `VmSize` is almost equal using the general-purpose `TBB` allocator and the custom FastFlow allocator. The version with `libc++` new/delete (`STD`) exhibits a step behavior. Interesting is the case of `Hoard`, which is more conservative in terms of memory use, and the memory footprint grows more slowly and remains lower with respect to the other solutions. This behavior is inherited by using FastFlow custom allocators on top of `Hoard`, though the `VmSize` becomes almost equal with the others at the end. The `VmSize` of the static version is one order of magnitude greater. In conclusion, this test confirms that the performance benefits of our optimizations are achieved without increasing memory consumption.

VI. Conclusions and Future Work

This paper describes how dynamic memory management affects the implementation complexity of DaSP programs. We implement our design choices and optimizations using parallel patterns and the FastFlow runtime. The overhead of dynamic memory management has been reduced by using a custom allocator tailored for the patterns. The experiments show that the combined use of it with a scalable general-purpose allocator allows to achieve up to 47% performance improvement with respect to the version using standard malloc/free or new/delete. In our future work we plan to extend our approach to other patterns for DaSP as well as to evaluate the potential advantage of using other lock-free allocators.

Acknowledgments

This work has been partially supported by the EU FP7 REPARA (ICT-609666) and H2020 RePhrase (ICT-2014-1) projects.

References

[1] M. M. Michael, "Scalable lock-free dynamic memory allocation," *SIGPLAN Not.*, vol. 39, no. 6, pp. 35–46, Jun. 2004.

[2] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: A scalable memory allocator for multithreaded applications," *SIGOPS Oper. Syst. Rev.*, vol. 34, no. 5, pp. 117–128, Nov. 2000.

[3] C. Pheatt, "Intel® threading building blocks," *J. Comput. Sci. Coll.*, vol. 23, no. 4, pp. 298–298, Apr. 2008.

[4] H. Andrade, B. Gedik, and D. Turaga, *Fundamentals of Stream Processing*. Cambridge University Press, 2014, cambridge Books Online.

[5] T. De Matteis and G. Mencagli, "Parallel patterns for window-based stateful operators on data streams: an algorithmic skeleton approach," *International Journal of Parallel Programming (to appear)*, 2016.

[6] M. Danelutto and M. Torquati, "Structured parallel programming with "core" fastflow," in *Central European Functional Programming School*, ser. LNCS, V. Zsók, Z. Horváth, and L. Csató, Eds. Springer, 2015, vol. 8606, pp. 29–75.

[7] D. Dice and A. Garthwaite, "Mostly lock-free malloc," *SIGPLAN Not.*, vol. 38, no. 2 supplement, pp. 163–174, Jun. 2002.

[8] 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*, ser. ISMM '06. New York, NY, USA: ACM, 2006, pp. 84–94.

[9] S. Lee, T. Johnson, and E. Raman, "Feedback directed optimization of tcmalloc," in *Proceedings of the Workshop on Memory Systems Performance and Correctness*, ser. MSPC '14. New York, NY, USA: ACM, 2014, pp. 3:1–3:8.

[10] X. Huang, C. I. Rodrigues, S. Jones, I. Buck, and W.-m. Hwu, "Xmalloc: A scalable lock-free dynamic memory allocator for many-core machines," in *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, ser. CIT '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1134–1139. [Online]. Available: http://dx.doi.org/10.1109/CIT.2010.206

[11] J. L. Risco-Martn, J. M. Colmenar, J. I. Hidalgo, J. Lanchares, and J. Daz, "A methodology to automatically optimize dynamic memory managers applying grammatical evolution," *Journal of Systems and Software*, vol. 91, pp. 109 – 123, 2014.

[12] Z. Falt, M. Kruli, D. Bednrek, J. Yaghob, and F. Zavoral, "Towards efficient locality aware parallel data stream processing," vol. 21, no. 6, pp. 816–841, jun 2015.

[13] C. Balkesen and N. Tatbul, "Scalable Data Partitioning Techniques for Parallel Sliding Window Processing over Data Streams," in *VLDB International Workshop on Data Management for Sensor Networks (DMSN'11)*, Seattle, WA, USA, August 2011.

[14] M. Danelutto, T. D. Matteis, G. Mencagli, and M. Torquati, "Parallelizing high-frequency trading applications by using c++11 attributes," in *Trustcom/BigDataSE/ISPA, 2015 IEEE*, vol. 3, Aug 2015, pp. 140–147.

[15] J. Bonwick, "The slab allocator: An object-caching kernel memory allocator." in *USENIX summer*, vol. 16. Boston, MA, USA, 1994.

[16] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati, "An efficient unbounded lock-free queue for multi-core systems," in *Proc. of 18th Intl. Euro-Par 2012 Parallel Processing*, ser. LNCS, vol. 7484. Rhodes Island, Greece: Springer, Aug. 2012, pp. 662–673.

[17] T. De Matteis and G. Mencagli, "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*, ser. PPoPP 2016. New York, NY, USA: ACM, 2016.