State Access Patterns in Stream Parallel Computations

M. Danelutto°, P. Kilpatrick†, G. Mencagli° M. Torquati°

°Dept. of Computer Science – Univ. of Pisa
†Dept. of Computer Science – Queen's Univ. Belfast

Abstract

We introduce a set of state access patterns suitable for managing accesses to state in parallel computations operating on streams. The state access patterns are useful for modelling typical stream parallel applications. We present a classification of the patterns according to the extent and way in which the state can be structured and accessed. We define precisely the state access patterns and discuss possible implementation schemas, performances and possibilities to manage adaptivity (parallelism degree) in the patterns. We present experimental results relative to implementations built on top of the structured parallel programming framework FastFlow that demonstrate the feasibility and efficiency of the proposed access patterns.

Keywords: structured parallel programming, parallel design patterns, stateful computations.

1 Introduction

Structured parallel programming models have been developed to support the design and implementation of parallel applications. They provide the parallel application programmer with a set of pre-defined, ready to use parallel pattern abstractions that may be directly instantiated, alone or in composition, to model the complete parallel behaviour of the application at hand. This raises the level of abstraction by ensuring that the application programmer need not be concerned with architectural and parallelism exploitation issues during application development. Rather, these issues are dealt with efficiently, using state-of-the-art techniques, by the framework programmer. Algorithmic skeletons, first introduced in the early '90s in the field of High Performance Computing [3], led to the development of several structured parallel programming frameworks including Muesli [12], SKEPU [11] and FastFlow [8]. Meanwhile, the software engineering community extended the classic design pattern concept [14] into the *parallel design pattern* concept [16] inheriting much of the algorithmic skeleton experience, and advantages deriving from structured parallel programming approaches have been clearly identified as a viable solution to the development of efficient parallel applications [2, 17].

In the context of parallel design patterns/algorithmic skeletons, stream parallel computations have typically been modelled via pipeline [10], farm [18] or other kinds of pattern/skeleton. However, they have traditionally been studied, designed and implemented as *stateless* patterns, i.e. as patterns where the stage (in pipeline) or the worker (in farm) processes/threads do not support internal state nor support access to some more generalized notion of "pattern" global state. This despite the fact that there are several well-know classes of application requiring the maintenance of either a "per pattern" or a "per component" state. Consequently, application programmers themselves must program state declaration and access within patterns, which is notably error prone and negates the benefits of the pattern philosophy. In this work we focus on task farm computation and discuss *stateful* pattern variations of the most general stream parallel pattern provided by the task farm. In particular we identify a range of cases from read-only state to the case where every computation requires access to the complete global state and in turn updates it, which is essentially a sequential pattern. We highlight as a key point the fact that there exist intermediate cases where there are clearly defined state updates and yet parallelism may be exploited because of the restricted nature of these updates.

The specific contribution of this paper consists therefore in the introduction of a *classification schema for stateful stream parallel computations and identification of the conditions under which*

Partially supported by EU FP7-ICT-2013-10 project REPARA (No. 609666) and EU H2020-ICT-2014-1 project RePhrase (No. 644235)

meaningful scalability may be obtained on state-of-the-art, off-the-shelf shared memory architectures for each of the classes identified, along with some preliminary experimental results assessing the FastFlow implementation of the patterns.

2 Stream parallel computations

A number of different stream parallel computations may be defined. In particular, here we are interested in those computations defined by providing a function f mapping input data stream items of type α to output data stream items of type β (*map* over streams). Thus the function f will have type $f : \alpha \to \beta$ and the result of the computation over an input stream $\ldots x_3, x_2, x_1, x_0$ will be $\ldots f(x_3), f(x_2), f(x_1), f(x_0)$. The ordering of the output items w.r.t. the input ones is optionally preserved. Input data items are available at different times: if item x_i is available at time t_i, item x_{i+k} will be available at time $t_i + \delta t, \quad \delta t > 0$. Ideally, if input stream item x_i is available for computation at time t_i, then the output stream item $f(x_i)$ will be delivered to the output stream at time $t_i + t_f$, t_f being the time to compute function f. Suppose input items appear on the input stream every t_a, and assuming use of n_w parallel activities (threads, processes) to compute f over different input stream items. The (ideal) service time of the parallel computation $T_s(n_w)$ and the time spent to compute m input tasks $T_c(n_w, m)$ may, respectively, be approximated as:

$$T_s(n_w) = max\{t_a, \frac{t_f}{n_w}\} \qquad \text{and} \qquad T_c(n_w, m) \approx mT_s$$

Implementation Stream parallel computations are usually implemented according to well-know parallel design patterns. Using a **master/worker** pattern, a *master* concurrent activity distributes input tasks and collects output results to/from a set of concurrent activities called *workers*. Each worker executes a loop waiting for a task to be computed, computing f and returning the result. Using a **farm pattern**, an *emitter* concurrent activity schedules input tasks to a set of *workers* computing f. Workers direct the output to a *collector* concurrent activity which, in turn, delivers the results onto the output stream. In this case the emitter and collector activities are called "helper" activities. If the computation is not required to enforce input/output ordering of tasks and results (i.e. if $f(x_i)$ may be delivered onto the output stream in any order w.r.t. $f(x_{i-1})$), the collector activity may be suppressed and worker activities may deliver directly to the output stream. In both cases, the master (emitter) concurrent activity may be programmed to implement different scheduling strategies and the master (collector) concurrent activity may be programmed to post-process the $f(x_i)$ items computed by the workers.

FastFlow In the remainder of this paper, we will discuss possible implementations, while highlighting advantages and issues, of state access patterns in stream parallel computations on top of FastFlow, which is a structured parallel programming framework available as an open source header-only library [1]. FastFlow natively provides a number of different stream and data parallel algorithmic skeletons implementing various parallel design patterns [8] on shared memory multicores. FastFlow provides the `ff_farm` class which implements both the master/worker and the farm pattern with fully programmable (if required) master/emitter and collector. For more details, the interested reader may refer to the FastFlow tutorial [8].

3 State patterns

When state is taken into account in a task farm pattern, various situations arise which may lead to different parallelization opportunities. The generic situation is that the computation of the result produced for input item x_i depends on both the value of x_i and on the value of the state

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

at the moment x_i is received. However, more specific possibilities can be identified according to the way in which the state is logically used and the access pattern of the sequential computation semantics, which must be respected by any parallization schema. To explain clearly from where different parallel patterns originate, we introduce two orthogonal dimensions that characterize the behavior of stateful streaming computations:

- ***non-partitionable vs. partitionable state***: the state is *non-partitionable* when for every input item it is not possible to establish *a-priori* (i.e. before executing the task item) which parts of the state data structures are accessed and possibly modified by the item execution. Instead, if for each item we can precisely state which parts of the state are used (e.g., by inspecting specific properties of the item) we say that the state is logically *partitionable*.
- ***ordered state access vs. relaxed state access***: in the case of *ordered* access semantics, the computation on every input item x_i must use the state obtained *after* the execution of the immediately preceding item x_{i-1}. In contrast, if the semantics allows the item x_i to be processed using the state obtained after the execution of any input item (received *before* or *after* x_i), we say that the computation allows a *relaxed* access to the state.

These two orthogonal dimensions can be used to define a precise classification of parallel patterns for stateful computations on streams which, in combination with further properties of the computation to be parallelized, turn out a) to be useful for modelling common real-world parallel applications and b) to support non-serial implementations of the state concept or at least provide upper/lower bounds on the scalability eventually achieved in the parallel computation. In the following sections we will present different parallel patterns by providing for each of them a description with its functional semantics, implementation directives and theoretical performance (scalability).

3.1 Serial state access pattern

Definition This pattern (briefly, SSAP) models the most conservative situation of computations with *ordered* access to *partitionable/non-partitionable* state when we have no further information about the properties of the sequential computation and of the input items. The pattern computes the result relative to input task $x_i : \alpha$ as a function ($\mathcal{F} : \alpha \to \gamma \to \beta$) of the value of the task and of the current value of a state $s_i : \gamma$. The new state s_{i+1} to be used to compute the next stream item will be computed using another function $\mathcal{S} : \alpha \times \gamma \to \gamma$. Given an initial state $s_0 : \gamma$ and an input stream $\ldots, x_2, x_1, x_0$ the result of the computation is $\ldots, \mathcal{F}(x_2, \mathcal{S}(x_1, \mathcal{S}(x_0, s_0))), \mathcal{F}(x_1, \mathcal{S}(x_0, s_0)), \mathcal{F}(x_0, s_0)$ which obviously implies sequential computation of the items appearing on the input stream.
Implementation To provide both consistent access to the state and to process input items in the arrival order, the serial state access pattern may be implemented using a FastFlow farm with a single worker (simplified to a sequential wrapper pattern) accessing a single global state variable.
Performance Serial state access pattern obviously implies serial execution of the worker code and, as a consequence, no scalability can be achieved.

3.2 All-or-none state access pattern

Definition This pattern (briefly, ANSAP) represents a modification of the pattern described in Sect. 3.1 in which, in the case of computations with *ordered* state access, we use the fact that not all of the input items modify the internal state (*partitionable* or *non-partitionable*). In addition to the function $\mathcal{F}$ defined as in the previous pattern, we introduce a boolean predicate $\mathcal{P} : \alpha \to \texttt{bool}$ which, evaluated on an input item, returns true if the input item may modify the state, or false if the execution on the input item reads but does not modify the state. Therefore, the next state x_{i+1} produced after the computation on the input item x_i is defined as $\mathcal{S}(x_i, s_i)$ if $\mathcal{P}(x_i) = \texttt{true}$ or s_i otherwise. The basic intuition of this pattern is that parallelism can be exploited among input items that do not modify the state. Let x_k be the last processed input item at a certain

time instant such that $\mathcal{P}(x_k)$ evaluates to true. All the consecutive input items x_i with $i > k$ such that $\mathcal{P}(x_i)$ is false can be executed in any order, because they do not modify the internal state, i.e. their execution is idempotent.

Implementation The pattern can be implemented in FastFlow as a farm with n_w workers, each having a *copy* of the state. The emitter schedules items according to the value of the predicate. For input items x_i such that $\mathcal{P}(x_i)$ is true, they are broadcast to all the workers which process them keeping their state local copy updated. Instead, each x_i such that $\mathcal{P}(x_i)$ is false is scheduled to a worker according to any scheduling strategy (e.g., round-robin, on-demand or any other). Smaller memory occupation can be achieved using a single shared copy of the state, and placing the emitter in charge of executing all the tasks that can modify the state in their arrival order. In both implementations the scheduler has to ensure that all the previous items have already been computed by the workers before scheduling (or executing) an item that modifies the state. No locks/mutexes/semaphores are in general required to protect the state in this pattern.

Performance The scalability depends on the probability of receiving input items that modify the state. The higher the probability the lower the scalability achieved by this parallelization. Formally, if t_f and t_s denote the time spent by the workers in executing the functions $\mathcal{F}$ and $\mathcal{S}$, respectively, and p the probability that the predicate returns true, the scalability with $n_w > 0$ workers is:

$$sc(n_w) = \frac{t_f + p\,t_s}{p\,(t_f + t_s) + (1-p)\,\dfrac{t_f}{n_w}} \qquad \lim_{p \to 0} sc(n_w) = n_w \qquad \lim_{n_w \to \infty} sc(n_w) = \frac{t_f + p\,t_s}{p\,(t_f + t_s)}$$

This result is valid if the evaluation of the predicate $\mathcal{P}$ by the emitter introduces a negligible overhead. Otherwise, the maximum scalability is limited by $1/T_{\mathcal{P}}$, where $T_{\mathcal{P}}$ is the average computation time of the predicate on an input task.

3.3 Fully partitioned state access pattern

Definition Preconditions to apply this pattern (briefly called FPSAP) are: *i)* the state must be *partitionable*, and *ii)* we have an *ordered* access to the state. The state can be modelled as a vector v of values of type γ of length N ($v : \gamma$ vector). A hash function: $\mathcal{H} : \alpha \to [0, N-1]$ exists mapping each of the input items to a state vector position (entry). The state vector is initialized before starting the computation with some initial value $s_{init} : \gamma$. Functions $\mathcal{F}$ and $\mathcal{S}$ are defined as stated in Sect. 3.1 and the computation of the farm is defined such that for each item of the input stream x_i, the output result on the output stream is computed as $\mathcal{F}(x_i, v[\mathcal{H}(x_i)])$ and the state is updated as $v[\mathcal{H}(x_i)] = s(x_i, v[\mathcal{H}(x_i)])$.

We can observe that while the pattern in Sect. 3.2 exploits the information that some of the items do not modify the state, here we exploit a different property: state entries other than $\mathcal{H}(x_i)$ are not needed to compute stream item x_i. This makes it possible to extract parallelism and an ideal scalability in the best case.

Implementation Given a task farm skeleton with n_w workers, the N state entries will be partitioned among the workers. In the simplest situation, the workers are assigned to a subset of consecutive state entries, i.e. by giving item v_i to worker $\lceil i/n_w \rceil$. The farm emitter will schedule task x_k to worker $\lceil h(x_k)/n_w \rceil$ using the hash function $\mathcal{H}$. Furthermore, that worker will be the one hosting the current, updated value of the state entry necessary to compute both the output result and the state update. In general, other assignment strategies of state entries to the workers can be applied in order to keep the load balanced. Since all the input items modifying the same state partition are executed serially and in arrival order by the same worker, no locks/mutexes/semaphores are in general required to protect the state partitions.

Performance Load balancing, and therefore scalability, depends on the effectiveness of the hash function to spread incoming tasks (more or less) equally across the full range of workers. In the case of a fair implementation of function $\mathcal{H}$, we can get close to the ideal scalability. If the function $\mathcal{H}$ schedules more items to a subset of the available workers, the scalability achieved will

Pattern	*Motivating example/Application*
SSAP	An online auction system processes a stream of requests, each representing a buy proposal for a set of goods with a proposed price and quantity. The system maintains a global state with all the available goods, their quantity and a minimum price. The received bids are processed *serially*, as they may modify the internal state, and in their *strict arrival order* in order to provide a fair bidding environment.
ANSAP	A financial auto-quoting system receives high-volume data streams of operations from either human or automated users. Some operations (typically less frequent) may modify the market state, i.e. the current set of bids and asks (buy and sell proposals) both active and suspended, which must be executed in their *arrival order*. As an example, an operation can create/delete a proposal or modify (edit) an existing one in the market. In contrast, more frequent operations are queries that do not modify the market state (e.g., search for the best proposals for a given stock symbol).
FPSAP	A recurrent pattern in Data Stream Processing [1] consists in the computation of data streams that convey input items belonging to a collection of sub-streams, each corresponding to all the items with the same partitioning attribute. The state is separated into *independent partitions*, each associated with a sub-stream. The computation of an input item reads and modifies only the state partition of its sub-stream. An example is a deep packet inspection system [7] that maintains a state partition relative to each individual connection to be analyzed.
SFSAP	The Dedup application of the PARSEC benchmark suite [4] consists in a pipeline of several stages processing a stream of video frames. The third stage is executed in parallel by several threads sharing a global hash table. Each input of that stage is a stream chunk which is uniquely identified by a SHA-1 checksum. The stage checks the presence of the chunk in the global table (indexed by the checksum) and, if present, the chunk is directly forwarded to the last stage of the computation. Otherwise, the hash table is *atomically* updated and the chunk transmitted to a compression stage.
ASAP	Searching for the number of occurrences of a string in a text (or of DNA sequences in a genome) is a typical application implementing this state access pattern. The state maintains the number of occurrences of the target strings that will be accessed and modified for each input item (a string in the text).
SASAP	An application that dynamically generates a space of solutions by looking for the one with the best "fitness" value is an example of this state access pattern. The global state is represented by the best solution candidate. Both solution and fitness values are stored in the state. Local approximations of the currently available "best" solution may be maintained and updated to hasten convergence of the overall computation. Solutions "worse" than the current "best" solution are simply discarded. This pattern is naturally able to model Branch and Bound algorithms for finding the best solution in a combinatorial search space.

Figure 1: Motivating examples and applications relative to the different state access patterns

be impaired. Let p_i the probability to schedule an input item to the i-th worker. The maximum scalability is: $sc(n_w) = \frac{1}{p_{max}}$ where $p_{max} = \max\{p_1, p_2, \ldots, p_{n_w}\}$, i.e. the probability of the worker receiving more input items. In the case of a uniform distribution (fair scheduling), the scalability is ideal.

Another factor that may hamper scalability of this pattern is the computational weight of the function $\mathcal{H}$ computed by the emitter on each input item. Let t_h be the processing time of this function, the maximum number of input items processed by the farm per time unit is bounded by $1/t_h$ and this may impair the overall performance if t_h is not negligible with respect to the computation on the input items.

3.4 Separate task/state function state access pattern

Definition The separate task/state function access pattern (briefly, SFSAP) can be used in scenarios characterized by a *relaxed* access to the state (any order is acceptable) in the case of both *non-partitionable* and *partitionable* state. Let $s : \gamma$ be the state. The computation relative to the input item $x_i : \alpha$ is performed in two steps: first a function $\mathcal{F} : \alpha \to \beta$ (not depending on state values) is applied to the input item to obtain $y_i = \mathcal{F}(x_i)$. Then, a new state value s_{new} is computed out

of y_i and of the current value of the state s_{curr}, i.e. $s_{new} = \mathcal{S}(y_i, s_{curr})$. It is worth noting that, according to the relaxed access to the state, s_{curr} is the current state value which is not necessarily the result of the computation on the previous input item x_{i-1} in the arrival order.

The computation on a generic item x_i will therefore require some time (t_f) to compute $\mathcal{F}$ and *then* some time to fetch the current state value and to compute and commit the state update (t_s). The pattern can output all modifications applied to the state onto the output stream. A variant worth consideration is the one which outputs only the value updates to the global state s such that $cond(s)$ holds true for some $c : \gamma \to \mathsf{bool}$.

Overall, this pattern is similar to that discussed in Sect. 3.1, the main difference being the way in which the state is accessed. In the `SSAP`, the state is read at the beginning and written at the end of the item computation, and the state needed to compute item x_i is the one after computing the previous item x_{i-1}. In this pattern instead, the state must be accessed atomically during the computation of the function $\mathcal{S}$ on any item. However, input items can be processed in any order. Furthermore, in the `SFSAP` the state is accessed only while computing $\mathcal{S}$. For the whole period needed to compute $\mathcal{F}$ there is no need to access the state.

Implementation This pattern is implemented on top of a FastFlow farm. A global state is allocated in shared memory before actually starting the farm, along with all the appropriate synchronization mutexes/locks/semaphores needed to ensure mutually exclusive access to the state. Pointers to the shared data and to all the required synchronization mechanism variables are passed to all the parallel components composing the task farm pattern. A generic farm worker therefore computes $\mathcal{F}$ relative to the received x_i item and then: *a)* accesses shared global state using the synchronization mechanisms provided along with the shared state pointer; *b)* computes the state update; *c)* updates the global state; and *d)* eventually releases the locks over the global state.

This pattern can be also applied to partitionable states. In this case it is possible to increase concurrency among threads by using a separate lock/mutex/semaphore protecting each state partition instead of a lock for the whole state. This solution can be useful in cases where it is very costly to have the emitter in charge of determining the partition of the state accessed by each input item (as in the pattern in Sect. 3.3) provided that the computation semantics allows a relaxed access to the state (any sequential order of input items is admissible).

Performance Scalability of the separate task/state function state access pattern is obviously impacted by the ratio of the time spent in a worker to compute $\mathcal{F}$ (denoted by t_f) to the time spent to update the state (t_s), the latter contributing to the "serial fraction" of the farm. The time taken to compute n_w items sequentially will be $n_w(t_f + t_s)$. The time spent computing the same items in parallel, using n_w workers will be (at best) $n_w t_s + t_f$ and therefore the maximum scalability will be limited by:

$$\lim_{n_w \to \infty} sc(n_w) = \lim_{n_w \to \infty} \frac{n_w(t_f + t_s)}{n_w t_s + t_f} = 1 + \frac{t_f}{t_s} \quad (1)$$

3.5 Accumulator state access pattern

Definition In the accumulator state pattern (briefly, `ASAP`) the state is a value $s : \gamma$. Functions $\mathcal{F}$ and $\mathcal{S}$ are defined that compute the result and the state update out of the current state and of the current input item. Function $\mathcal{S}$ is restricted to be of the form: $\mathcal{S}(x_i, s_{curr}) = \mathcal{G}(x_i) \oplus s_{curr}$ where $\oplus$ is an *associative* and *commutative* binary operator and $\mathcal{G}$ is any function $\mathcal{G} : \alpha \to \gamma$. We denote by s_{zero} the initial value of the state.

Implementation Owing to the associativity and commutativity of the binary operator, we can maintain a local state variable per worker, without needing mutual exclusion (locks, semaphores) to access the global state directly by the workers as in the previous, more general, pattern. The local state value s^w of worker with identifier $w = 1, 2, \ldots, n_w$ is initialized to the identity value with respect to the function $\oplus$ (s_{zero}). The worker w processing item x_i computes $y_i = \mathcal{F}(x_i, s^w_{curr})$. Then it either sends y_i immediately to the farm collector, and then computes the new state value $s^w_{new} = \mathcal{G}(x_i) \oplus s^w_{curr}$ and periodically sends the value s^w_{new} to the collector, re-initializing the local state to s_{zero}; or delivers y_i and $\mathcal{G}(x_i)$ to the collector, which will update the global state value accordingly, item by item.

Pattern	*Adaptivity hints*
SSAP	This is *de facto* a serial pattern, therefore no adaptivity policies in this case.
ANSAP	Adaptivity can be handled quite efficiently. In the solution with replicated state, a new worker can be instantiated by assigning to it a copy of the updated version of the state (all the other workers must have consumed their pending items). A worker can be removed from the pool of workers without the need to modify/update the local state of the other workers.
FPSAP	In case of an increase in the parallelism degree, some of the state partitions held by some workers must be migrated to the new worker which, having acquired them, can start processing new input items. The state migration must be performed by respecting the computation semantics (state partitions must be accessed atomically) and must not violate the strict sequential order in which items of the same group must be processed. Techniques to solve this issue in the case of autonomic/elastic solutions have been proposed in some past research works [9, 15]. Symmetric actions are taken in the case of parallelism degree reduction.
SFSAP	Increasing or decreasing the number of workers used does not pose any particular issue. Adding a new worker simply requires addition of the worker to the emitter worker queues. Taking away one worker simply requires stopping it while it is waiting for a new input item.
ASAP	When increasing the number of workers the new workers should be instantiated with a local state value initialized with s_{zero}. When decreasing the number of workers, before stopping any worker thread, the locally stored state values should be directed to the collector. If workers have to be "merged" (e.g., to reduce the worker number but not imposing unexpected update messages on the collector) the resulting worker should be given the "sum" of the merged workers' local state values ($s_i \oplus s_j$ where workers i and j are merged).
SASAP	When the number of workers in the farm is increased, the new worker(s) should be given the current value of the global state maintained in the collector. This can also be implemented by allowing the worker(s) to be started with a proper s_{init} and then leaving the new workers to get regular update values from the collector. This obviously slows down the convergence of the overall computation, as the new workers will initially only provide "wrong" approximations of the global state. When the number of workers in the farm is decreased, the candidate workers to be removed may simply be stopped immediately before attempting to get a new task on their input stream from the emitter.

Figure 2: Adaptivity issues in different state access patterns

Performance Load balancing is not affected by the state updates, apart from an increased load on the collector. Depending on the computational weight of $\oplus$, the implementation with periodic updates to the collector will be preferred to the one continuously sending the updates to the collector. Scalability is theoretically ideal.

3.6 Successive approximation state access pattern

Definition The pattern (briefly, SASAP) manages a state which is a value $s : \gamma$. For an input stream with items x_i, a stream of successive approximations of the global state s is output by the pattern. Each computation relative to the task x_i updates state if and only if a given condition $\mathcal{C} : \alpha \times \gamma \rightarrow \mathsf{bool}$ holds true. In that case, the new state value will be computed as $\mathcal{S}(x_i, s_{curr})$. Therefore, in this state access pattern we have:

$$s_{new} = \begin{cases} s_{curr} & \text{if } \mathcal{C}(x_i, s_{curr}) = \mathsf{false} \\ \mathcal{S}(x_i, s_{curr}) & \text{otherwise} \end{cases}$$

The state access pattern is defined *if and only if* $\mathcal{S}$ is *monotone* in the state parameter, that is $\mathcal{S}(x_i, s_{curr}) \leq s_{curr}$ for any x_i, and the computation converges even in the case of inexact state updates, that is, where different updates read a state value and decide to update the state with distinct values at the same time (global state updates are anyway executed in mutual exclusion).
Implementation The pattern is implemented with a task farm, where global state value is maintained by the collector. Any update to the state is broadcast to the workers via a feedback channel from

collector to the emitter. Each worker w maintains a properly initialized[2] local copy of the global state $s^w : \gamma$. Workers processing an input stream item x_i send update messages to the collector. Updates are computed on the local value of the state, and so this may turn out to be misaligned with respect to the global state value maintained by the collector and to the local copies maintained by the other workers. The collector only accepts state updates satisfying the monotonic property of $\mathcal{S}$, that is if a worker sends an update that would change the state in a non-monotonic way, that update is discarded on the basis that a better update has already been found. At any update of its local "global" state value, the updated value is output over the pattern output stream, and therefore the pattern output stream hosts all the subsequent successive approximations computed for the global state.

Performance Again, by maintaining local copies and by relying on the monotonic property of the state updates (when executed), we can avoid having a unique global state updated atomically by the workers (as in the general pattern in Sect. 3.4). However, we can identify three possible additional sources of overhead in the pattern compared with the stateless task farm pattern:

- a first performance penalty is paid to update the global state at the farm collector every time a worker decides to send a state update. As this just requires the comparison between the state currently computed as the "best" one in the collector and the update value obtained from the worker, this may be considered negligible;
- a second penalty is paid to send back the global state update to the workers, through the farm feedback channel. This requires an additional communication from collector to emitter and a broadcast communication from emitter to workers. FastFlow implements both communications very efficiently and so the associated overhead is negligible (in the range of fewer than some hundred clock cycles on state-of-the-art multicore architectures);
- a third penalty is paid for the extra update messages directed by workers not having available (as local state copy) an updated state value. This happens in the case that the collector has already propagated the new state value but the message has not yet reached the worker. This performance penalty comes in two components: *a)* the worker w may compute an extra $\mathcal{S}(x_i, s^w_{curr})$ as a consequence of having a wrong local s^w_{curr} value in the computation of $\mathcal{C}(x_i, s^w_{curr})$, and *b)* the worker directs an extra state update message to the collector.

3.7 Pattern summary

Here we briefly summarize the patterns described above with respect to their requirements in terms of the nature of the state (partitionable vs. non-partitionable) and the type of state access (ordered or relaxed). This summary is provided in Tab. 3.

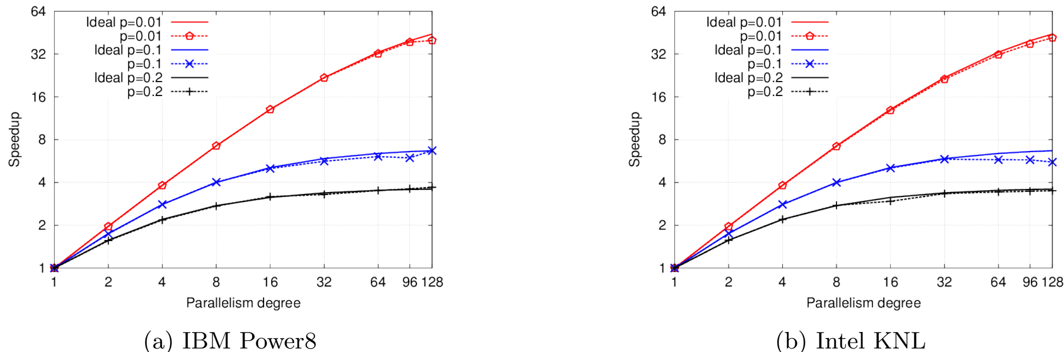
	SSAP	ANSAP	FPSAP	SFSAP	ASAP	SASAP
Relaxed Access	*No*	*No*	*No*	**Yes**	**Yes**	**Yes**
Ordered Access	**Yes**	**Yes**	**Yes**	*No*	*No*	*No*
Partitionable State	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
No-partition. State	**Yes**	**Yes**	*No*	**Yes**	**Yes**	**Yes**

Table 3: Summary of the state access patterns presented in the paper.

It is interesting to observe that the two options in the state access dimension are indeed mutually exclusive because this dimension is related to the computation semantics. Instead, the nature of the state (partitionable or not) is a performance-oriented dimension meaningful mainly for the application of FPSAP.

Motivating examples/applications for each pattern are summarized in the table in Fig. 1, while hints about policies for dynamic adaptivity (i.e. dynamic variations in the number of workers) in

[2] e.g., to some known s_{max} value.



(a) IBM Power8 (b) Intel KNL

Figure 3: `ANSAP` scalability

the FastFlow implementations are outlined in the table in Fig. 2. It is worth pointing out that we target *off-the-shelf* multicore architectures and the applications listed in Fig. 1 represent notable use cases for this class of machine.

4 Experiments

We describe several experiments relating to the state access patterns aimed at demonstrating that the patterns work and that the performance results are as predicted. The majority of the experiments[3] have been performed on a 64 core, 4 way hyperthreading Intel Xeon PHI Knights Landing (KNL) architecture. The experiments use parallelism degrees up to 128, to exploit the double floating point units per core available. For the experiments we used Linux 3.10.0 with `icc` 16.0.3 and FastFlow 2.1.2. All experiments have been run using synthetic applications employing the various state access patterns, where actual computations are dummy floating point kernels. A couple of full size applications have been used to assess `FPSAP` (see below).

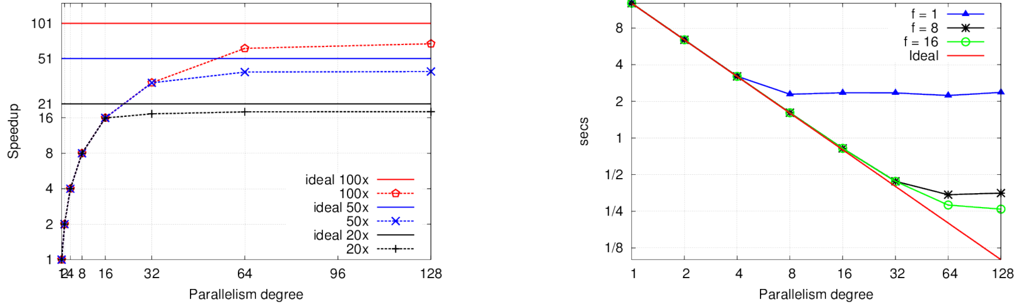
All-or-none state access pattern: here we aimed to verify the scalability limits of this pattern. We considered the case in which: i) $t_f = t_s$; ii) the state is replicated in all farm workers; and iii) the probability of modifying the state varies between 1%, 10% and 20%. The emitter checks if the predicate on the input item is true and if so it broadcasts the item to all workers making sure that all of them are not still computing previous items. Fig. 3 shows that the scalability achieved by the FastFlow implementation is very close to the theoretical bounds.

Fully partitioned state access pattern: we did not run specific synthetic benchmarks to verify the performance of the FPSAP as some of the authors of this work have already developed distinct applications explicitly using the pattern. The Deep packet inspection application in [6] uses the FPSAP FastFlow pattern and i) achieved close to linear scalability on state-of-the-art Intel Sandy Bridge 16 core servers and ii) succeeded to sustain deep packet inspection on a 10Gbit router. Financial applications used in [5] achieved similar results (almost linear scalability on Intel Ivy Bridge 24 core servers) using the FPSAP.

Separate task/state function state access pattern: the second experiment investigates the performance of the separate task/state function state access pattern. In this case we aimed at verifying the limits given by equation (1). We ran the synthetic application incorporating the separate task/state function state access pattern while varying the ratio between the times spent computing f and computing s. As shown in Fig. 3a, scalability behaves as predicted by (1): for each t_f/t_s ratio, there is a corresponding maximum speedup asymptote which is the one approximated by the actual speedup curve.

Accumulator state access pattern: we measured the execution time of our prototype synthetic application incorporating the accumulator state access pattern while varying the amount of time (t_f) spent in the computation of the task to be output on the output stream ($f(x_i, s_w)$) and the

[3] apart from those aimed at assessing performance portability – see paragraph "Different architectures" below



(a) `SFSAP` speedup ($x\times$ means $t_f = x \times t_s$) (b) `ASAP` compl. time vs. grain and update frequency

Figure 4: `SFSAP` and `ASAP` experimental results (KNL)

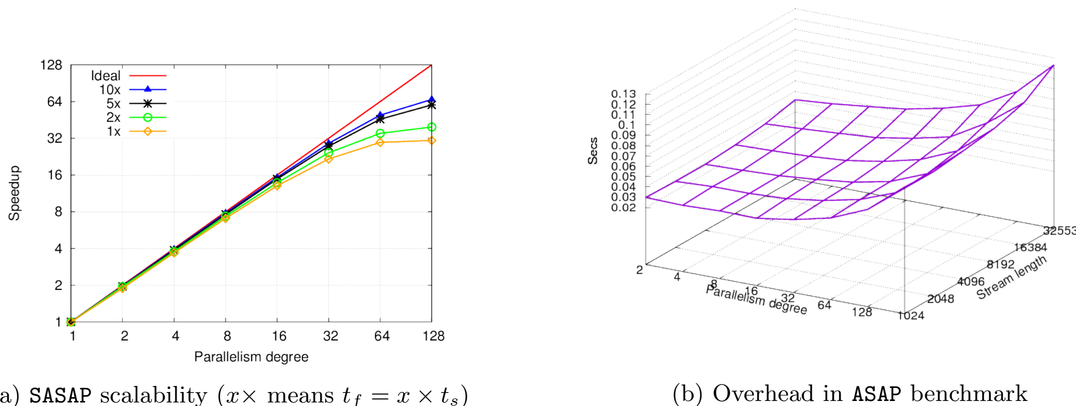
time (t_s) spent in the computation of the new state value/update ($g(x_i) \oplus s_{i-1}$)). When $t_f >> t_s$ we verified that the completion time is close to the ideal, that is $\frac{m(t_f+t_s)}{n_w}$. We also investigated the effect of the update frequency (f, the number of items processed by a worker before sending the state update of the collect). Ideally, the frequency should be chosen so that the collector does not become a bottleneck due to the state updates. This means that the number of workers should not be larger than $\frac{f(t_f+t_s)}{t_s}$. Fig. 4b shows completion times for a run where $t_f = 5t_s$. Ideally, scalability should stop at 6 workers when $f = 1$, at 48 when $f = 8$ and at 96 when $f = 16$, which is in fact what we can observe in the plot.

When $t_f = t_s$ and the update frequency is 1 the scalability is limited to 2. Experiments (not shown for space reasons) on the KNL have demonstrated that in fact the completion time halves when moving from parallelism degree 1 to 2 and then stays constant for larger parallelism degree.

Successive approximation: Fig. 5a shows results obtained with the implementation of the successive approximation state access pattern on the Sandy Bridge architecture. Several curves are plotted against the ideal completion time (the one computing according to formula in Sec. 3.6), while varying the amount of time spent computing the condition $c(x_i, s_{i-1})$ (t_f in the legend) and the time spent computing the state update $s'(x_i, s_{i-1})$ (t_s in the legend). As expected, the larger the time spent in the (worker local) computation of the condition, the better the results achieved.

Overhead We measured the overhead in the execution of our FastFlow implementation of the state access patterns. Fig. 5b shows the overhead measured with different parallelism degree and stream lengths in the execution of an `ASAP` pattern benchmark on a KNL architecture. The overhead increases more significantly with the parallelism degree and less significantly with the stream length, as expected in a stream parallel pattern. Specific experiments measuring the overhead introduced by the FastFlow *farm* pattern used to implement all of the state access patterns showed that the overhead related to the setup of the threads ranges from 30 to close to 80 msecs on KNL while the global overhead related to the management of a single stream item across farm components ranges from 0.1 to 0.5 μsecs on the KNL.

Different architectures The experimental results discussed so far have been obtained on the same Intel multicore architecture. However, very similar results have been achieved also when running our synthetic applications on different types of state-of-the-art architectures. In particular, we used Intel Sandy and Ivy bridge architectures with 16 and 24 2-way hyperthreading cores, respectively, and a 20 core, 8-way hyperthreading IBM Power8 architecture. Fig. 3 compares results achieved running the same experiment on IBM Power8 and Intel KNL.



(a) `SASAP` scalability ($x\times$ means $t_f = x \times t_s$)

(b) Overhead in `ASAP` benchmark

Figure 5: `SASAP` scalability and `ASAP` overhead evaluation (KNL).

5 Related work & Conclusions

A number of authors have considered various aspects of state in the context of stream processing. Typically, they employ less overtly structured approaches than the pattern-based concept presented here. Perhaps the closest to our work is that of Wu et al. [20] who introduce a framework for parallelizing stateful operators in a stream processing system. Their *split-(process*)-merge* assembly is very similar to the task farm presented here. They divide each stateful instance of *process* into non-critical access and critical access segments and present a more comprehensive theoretical model to determine scalability (based on shared lock access times, queue lengths, etc.) than is attempted here. However, they do not attempt the sort of classification scheme given in this work.

Verdu et al [19] focus on implementation issues in relation to parallel processing of stateful deep pack inspection. The propose Multilayer Processing as a model to leverage parallelism in stateful applications. They focus on lower level implementation issues, such as caching and do not explicitly employ structured pattern based parallelism of the kind used here. Gedik [15] examines properties of partitioning functions for distributing streaming data across a number of parallel channels. Thus the author focuses on the equivalent of properties of the hash function in our fully partitioned state access pattern. De Matteis et al. [9] discuss stateful, window based, stream parallel patterns particularly suited to model financial applications. The techniques used to implement the applications fit the design patterns discussed in this paper, but actually somehow mix accumulator, partitioned and separate task/state state access patterns. Fernandez et al. [13] also consider the partitioned state and examine issues related to dynamic scale-out and fault tolerance. As with the others, they do not use a pattern-based approach nor do they attempt a classification scheme of the kind presented here.

Stream processing has become increasing prevalent as a means to address the needs of applications in domains such as network processing, image processing and social media analysis. Such applications, when targeted at multicore systems, may be implemented using task farm and pipeline parallel patterns. We observe that typically such applications employ task farms in stateless fashion as it is here that the implementation is easiest and the return in terms of parallel scalability is greatest. However, we note that, while embracing state can lead to a de facto sequential computation, there are variations which can provide scope for parallel scalability. We have classified these variations, indicating for each the issues that arise in relation to implementation detail, performance and how the pattern may be adapted to vary performance. We have presented experimental evidence that, in terms of performance, the various classes of stateful task farms behave as predicted. We consider that a greater understanding of the extent to which (streaming) parallel patterns may incorporate state will broaden the possibilities for development of multicore applications using parallel pattern based approaches. To this end, our next step is to investigate other traditionally stateless patterns for stateful variants.

References

[1] H. C. M. Andrade, B. Gedik, and D. S. Turaga. *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*. Cambridge University Press, New York, NY, USA, 1st edition, 2014.

[2] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, Oct. 2009.

[3] M. Cole. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, 30(3):389–406, Mar. 2004.

[4] M. Danelutto, T. De Matteis, D. De Sensi, G. Mencagli, and M. Torquati. P^{3}ARSEC: Towards parallel patterns benchmarking. In *Proceedings of the 32th Annual ACM Symposium on Applied Computing*, SAC '17, New York, NY, USA, 2017. ACM.

[5] M. Danelutto, T. De Matteis, G. Mencagli, and M. Torquati. Data stream processing via code annotations. *The Journal of Supercomputing*, pages 1–15, 2016, doi:10.1007/s11227-016-1793-9.

[6] M. Danelutto, D. Deri, D. De Sensi, and M. Torquati. Deep packet inspection on commodity hardware using fastflow. In *Parallel Computing: Accelerating Computational Science and Engineering (CSE) (Proc. of PARCO 2013, Munich, Germany)*, volume 25 of *Advances in Parallel Computing*, pages 92 – 99, Munich, Germany, 2014. IOS Press.

[7] M. Danelutto, L. Deri, and D. D. Sensi. Network monitoring on multicores with algorithmic skeletons. In *Applications, Tools and Techniques on the Road to Exascale Computing, Proceedings of the conference ParCo 2011, 31 August - 3 September 2011, Ghent, Belgium*, volume 22 of *Advances in Parallel Computing*, pages 519–526. IOS Press, 2011.

[8] M. Danelutto and M. Torquati. Structured parallel programming with "core" fastflow. In V. Zsók, Z. Horváth, and L. Csató, editors, *Central European Functional Programming School*, volume 8606 of *Lecture Notes in Computer Science*, pages 29–75. Springer International Publishing, 2015.

[9] 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 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 13:1–13:12, 2016.

[10] D. del Rio Astorga, M. F. Dolz, L. M. Sanchez, and J. D. García. Discovering Pipeline Parallel Patterns in Sequential Legacy C++ Codes. In *Procs of the 7th Int'l Workshop on Progr. Models and Applications for Multicores and Manycores*, PMAM'16, pages 11–19, NY, USA, 2016. ACM.

[11] J. Enmyren and C. W. Kessler. Skepu: A multi-backend skeleton programming library for multi-gpu systems. In *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*, HLPP '10, pages 5–14, New York, NY, USA, 2010. ACM.

[12] S. Ernsting and H. Kuchen. Algorithmic skeletons for multi-core, multi-gpu systems and clusters. *Int. J. High Perform. Comput. Netw.*, 7(2):129–138, Apr. 2012.

[13] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Scalable and Fault-tolerant Stateful Stream Processing. In A. V. Jones and N. Ng, editors, *2013 Imperial College Computing Student Workshop*, volume 35 of *OpenAccess Series in Informatics (OASIcs)*, pages 11–18, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[15] B. Gedik. Partitioning functions for stateful data parallelism in stream processing. *The VLDB Journal*, 23(4):517–539, 2014.

[16] T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming.* Addison-Wesley Professional, first edition, 2004.

[17] M. McCool, J. Reinders, and A. Robison. *Structured Parallel Programming: Patterns for Efficient Computation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[18] M. Poldner and H. Kuchen. On implementing the farm skeleton. *PPL*, 18(1):117–131, 2008.

[19] J. Verdú, M. Nemirovsky, and M. Valero. Multilayer processing - an execution model for parallel stateful packet processing. In *Procs of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '08, pages 79–88, New York, NY, USA, 2008. ACM.

[20] S. Wu, V. Kumar, K.-L. Wu, and B. C. Ooi. Parallelizing stateful operators in a distributed stream processing system: How, should you and how much? In *Procs of the 6th ACM Int'l Conf. on Distributed Event-Based Systems*, DEBS '12, pages 278–289, New York, NY, USA, 2012. ACM.