Parallel Patterns for Window-based Stateful Operators on Data Streams: an Algorithmic Skeleton Approach

Tiziano De Matteis, Gabriele Mencagli

University of Pisa
Italy

INTRODUCTION

The recent years have been characterized by an explosion of data streams generated by a variety of sources

Over the Web in 60 seconds we have (2014):

- 2.6 M searches on Google
- 430K new tweets
- 25K purchases on Amazon
- 290K FB status updates
- 5M videos views on YouTube
- ...

Same story for Stock Market Feed, Sensor Networks and much more

Translating this information into decision in real-time has become a valuable opportunity

DATA STREAM PROCESSING

Data Stream Processing (DaSP): a new paradigm characterized by:

- unbounded input and no control on how elements (tuples) arrive;
- stringent performance requirements in terms of throughput and latency.

Various DaSP architectures have been proposed from *Data Stream Management Systems* (Aurora, Borealis, …) to modern *Stream Processing Engine* (Storm, Spark Streaming, Infosphere, …)

Applications are expressed as compositions of core functionalities in directed flow graphs (*continuous queries*):

- arcs represent streams;
- vertices are operators;
- **stateless** or **stateful** operators.

WHAT ABOUT PARALLELISM?

Parallelism in existing SPEs is expressed at different levels:

- *query* level: **inter-query** parallelism;
- *operator* level: **inter-/intra-operator** parallelism.

Stateful operators deserve special attentions from a parallelization p.o.v. but a *methodology* for intra-operator parallelism is still lacking.

Goal: show how parallelization issues of DaSP operators can be dealt with the *algorithmic skeletons*:

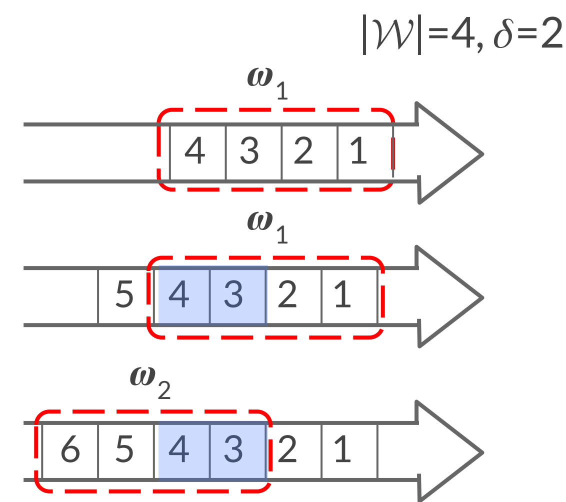
- it is a well known methodology;
- simplifies the reasoning on properties of the parallel solution;
- may be integrated in existing SPE.

WINDOW-BASED OPERATORS

Windows are the predominant *state abstraction* in DaSP. State is maintained as a buffer in which only the most recent tuples are kept. Useful also for focus the attention to more recent data

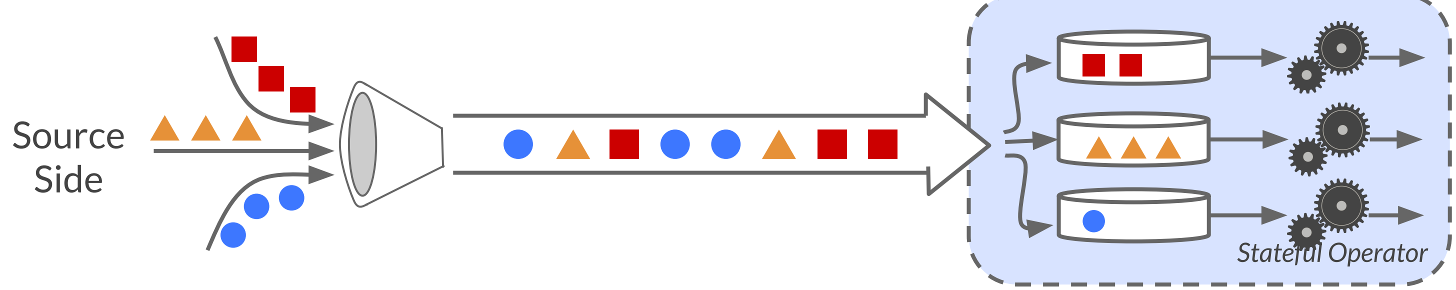
Different windowing methods can be characterized by specifying the eviction and triggering policies. Two parameters are important:

- the window size $|\mathcal{W}|$: in time (time-based windows) or #tuples (count- based):
  - *"The tweets received on the last 10 min"*;
  - *"The last 1000 quotes per stock symbol"*;
- the sliding factor δ: how window moves and its content is valid for being processed. If $\delta = |\mathcal{W}|$ we talk of *tumbling windows.*



MULTI-KEYED OPERATORS

In many contexts, the physical input stream conveys tuples belonging to **multiple logical substreams**. Examples from network monitoring, financial applications, social networks, ...



Stateful operators can require to maintain separated state (e.g. window) for each substream and apply computation on a substream basis. The association between tuples and substreams is made by a **key** attribute.

We refer to:

- ***multi-keyed* stateful operator** if $|\mathcal{K}|>1$;
- ***single-keyed* stateful operator** otherwise.

PARALLEL PATTERNS FOR WINDOW OPERATORS

In the following we assume a generic *window based stateful operator* working on a single physical input stream and producing one output stream.

A **task** is a *segment of the input stream corresponding to all the tuples belonging to the same window* on which the operator applies a computation $\mathcal{F}$.

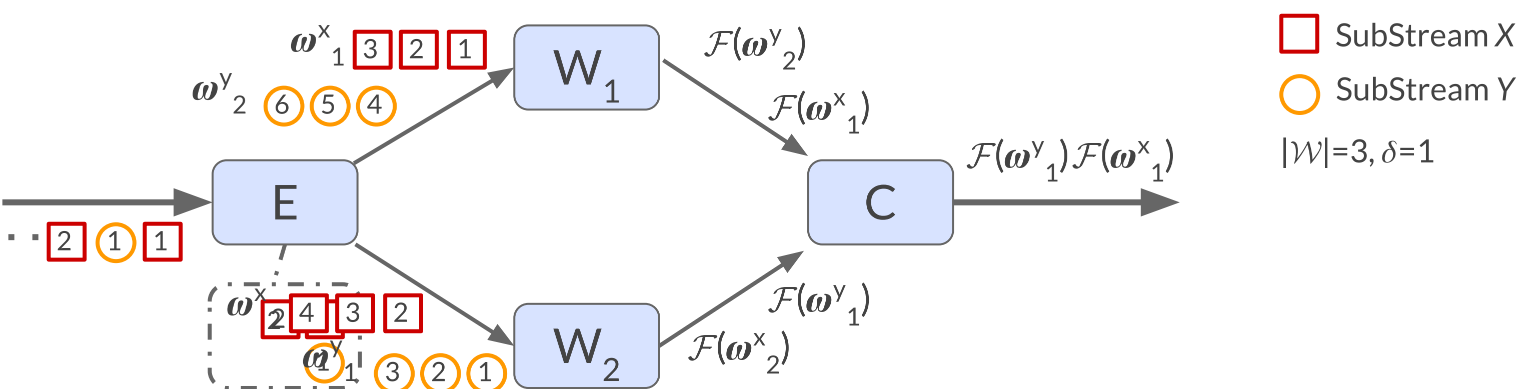
Patterns can be categorized in various ways:

- depending on how compute tasks: **window parallel** or **data parallel**
- on the basis of task distributions: the **granularity** of the distribution to the parallel executors (Workers) and the **assignment policy**, of windows to the Workers;
- considering window management: **active** vs **agnostic** Workers

4 patterns exemplified on count-based windows

WINDOW FARMING

Intuition: each window can be processed independently. We can adopt a classical farm skeleton, in which full windows are distributed to Workers



- Emitter: buffer tuples, update windows and send them to Workers;
- Workers: receive data, apply $\mathcal{F}$, transmit results and discard data. They are agnostic of the window management;
- Collector: receive results, (re-order them), forward.

WINDOW FARMING

Optimizations:

- tuples distribution on the fly (active Workers);
- assign batches of windows to Workers.

Summary:

- applicable to any window-based operator and any $\mathcal{F}$, single or multi-keyed. It is a window-parallel pattern;
- optimizes throughput, but not latency;
- Emitter can become bottleneck and data is replicated in several Workers. Optimizations mitigate this problems;
- load balancing can be easily obtained.

KEY PARTITIONING

It is a variant of Window Farming, with a constrained assignment policy

With *n* Workers, $\mathcal{K}$ is splitted into *n* partitions. Tuples having key in partition *i* are sent to Worker *i*

Optimization: on the fly distribution

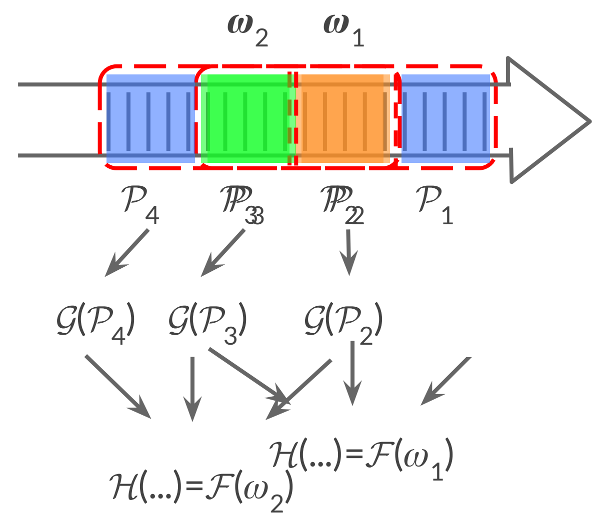
Summary:

- Applicable to any stateful-operator (not only windows). No single key;
- Improves throughput, but no latency;
- No data replication. Results with the same key are ordered;
- Only windows of different substreams may be computed in parallel;
- No load balancing if keys distribution is skewed (limited scalability);
- Widely diffused in literature and modern SPE.

PANE FARMING

Given $|\mathcal{W}|$ and δ>1, the idea is to divide each window into r non-overlapping partitions, called **panes**, of size:

$$\boldsymbol{\sigma}_p = gcd(|\mathcal{W}|, \delta) \quad r = |\mathcal{W}|/\boldsymbol{\sigma}_p$$



If $\mathcal{F}$ can be decomposed into two functions $\mathcal{G}$ and $\mathcal{H}$, used as:

$$\mathcal{F}(\omega) = \mathcal{H}(\mathcal{G}(\mathcal{P}_1), \mathcal{G}(\mathcal{P}_2), \ldots, \mathcal{G}(\mathcal{P}_r))$$

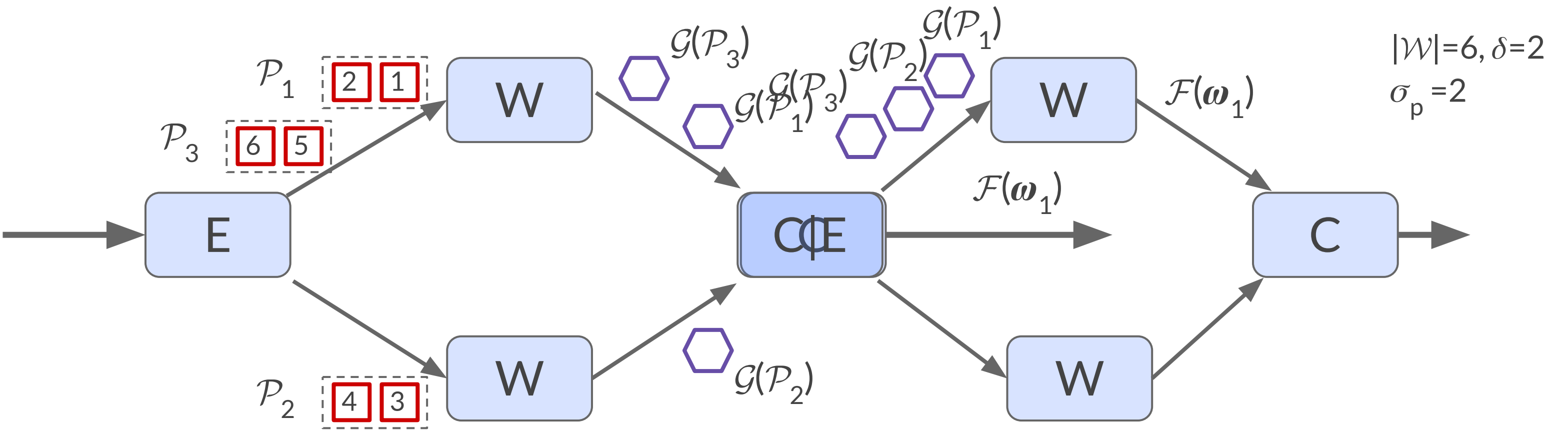
we can:

- apply $\mathcal{G}$ on each pane independently;
- obtain the final result by combining the pane results with $\mathcal{H}$;
- reuse: consecutive windows share overlapping panes.

Applicability: aggregates, associative functions,...

PANE FARMING

The application of $\mathcal{G}$ and $\mathcal{H}$ can be seen as a two-staged pipeline where each stage can be parallelized using WF:



- Emitter: schedules panes to Workers. They are tumbling windows;
- Workers: are agnostic: apply $\mathcal{G}$ and send results to Collector;
- Collector: gets pane results and applies $\mathcal{H}$ on windows of pane results. If needed, it can be further parallelized using WF

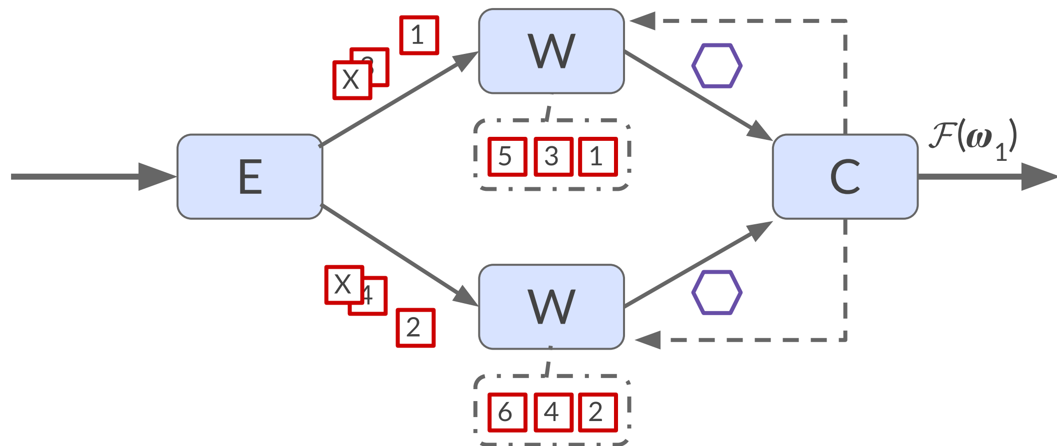
PANE FARMING

Summary:

- Can be applied on windowed operators, if $\mathcal{F}$ can be expressed as $\mathcal{G}$+$\mathcal{H}$. It is a window parallel pattern;
- Applicable to multi-keyed operators;
- Optimize throughput but also latency by sharing overlapping results between consecutive windows. Reduction is proportional at most to r the number of panes;
- No data replication, load balancing easy to achieve;
- Not useful is slide δ is equal to one.

WINDOW PARTITIONING

It is an adaptation of map-reduce skeleton on data streams: current window is partitioned among the Workers



- Emitter: distributes tuples to Workers (or scatter entire window);
- Workers: they are active: receive tuples, update window partition and, apply the map phase, transmit results and discard tuples;
- Collector: receive results, apply $\oplus_r$,forward.

WINDOW PARTITIONING

Summary:

- This is a data-parallel pattern;
- Applicable on $\mathcal{F}$ decomposable as map-reduce. Single and multi-keyed;
- Optimize throughput and latency. Latency reduction is proportional to partition size (hence degree of parallelism);
- No data replication. Can benefit of distribution on the fly;
- Load balancing easy to achieve if the computation has low variance processing time.

SUMMARY

We have introduced 4 different patterns

Pattern	Paradigm	Keyes	Win. Man.	Optimizes	Load Balancing
Window Farming	Window Parallelism	Single-/Multi-Keyed	Agnostic/Active	Throughput	✓
Key Partitioning	Window Parallelism	Multi-Keyed	Agnostic/Active	Throughput	⚠
Pane Farming	Window Parallelism	Single-/Multi-Keyed	Agnostic	Throughput and Latency	✓
Window Partitioning	Data Parallelism	Single-/Multi-Keyed	Agnostic/Active	Throughput and latency	✓

Nesting can be useful for augmenting the coverage

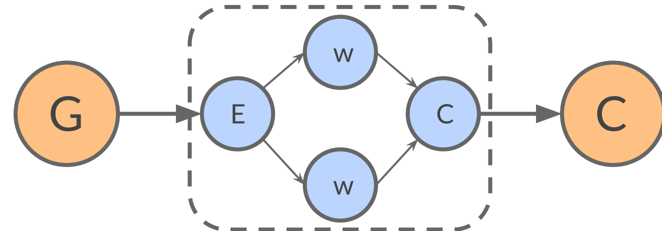
EXPERIMENTS

A prototypal implementation of the 4 patterns have been done in **Fastflow**, a C++ framework for skeleton-based parallel patterns:

- parallel entities have been implemented as *pthreads*, pinned on cores;
- they interact through *non-blocking lock-free queues*

Target architecture: dual CPU Intel Sandy Bridge Xeon E5-2650
16 cores (32 with SMT) running at 2GHz. 32 GB of Ram

In addition to the entities required by the patterns, we have a *Generator* and a *Consumer* threads. Therefore we can have up to 12 Workers



EXPERIMENTS

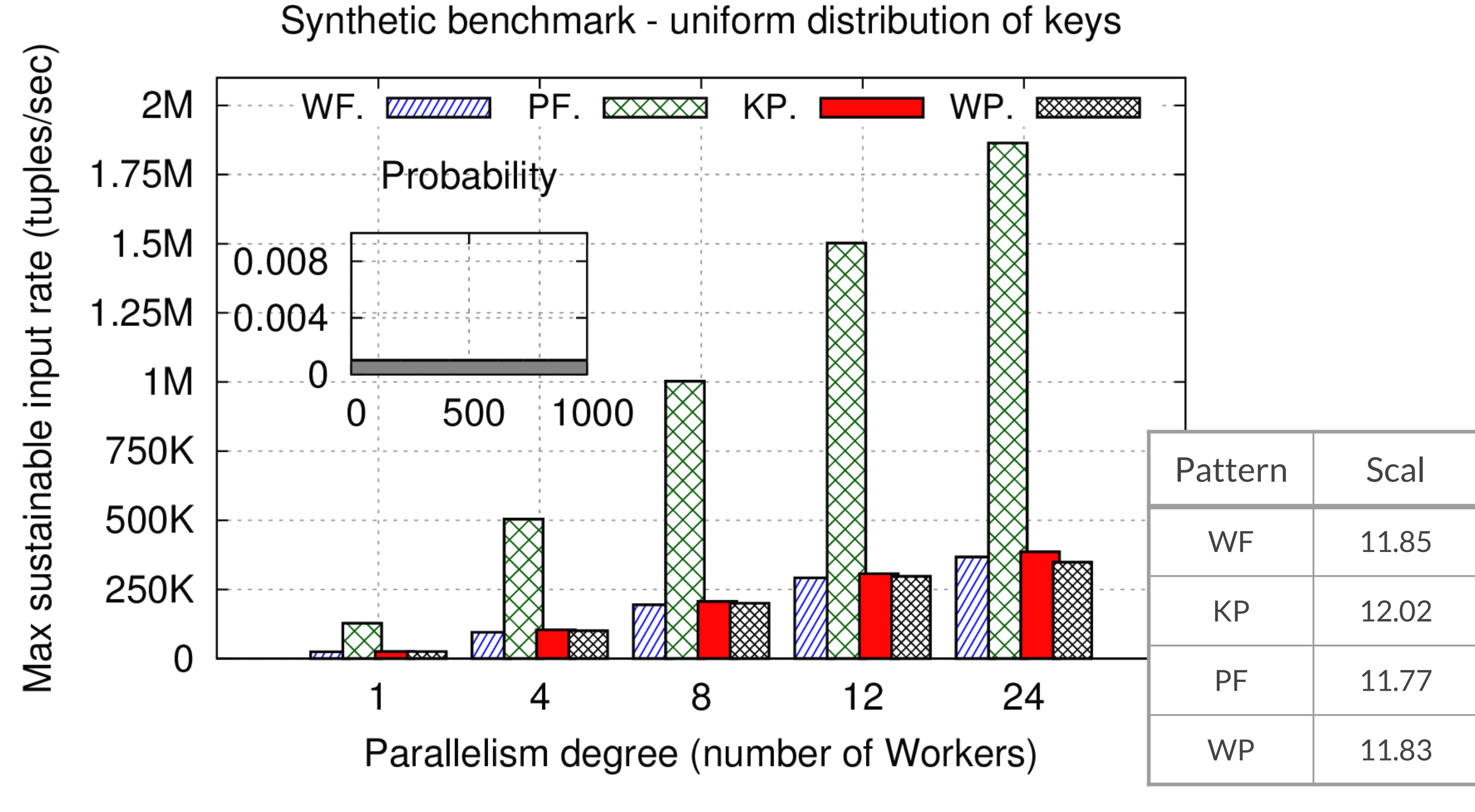
Synthetic benchmark that mimic a suite of statistical aggregates over quotes coming from stock market:

- tuples are records of 64 bytes containing numeric fields;
- $|\mathcal{W}|$=1000 tuples, δ=200 tuples, $|\mathcal{K}|$=1000;
- to exploit all patterns, the computations is a function $\mathcal{F}$ decomposable in a function $\mathcal{G}$ and $\mathcal{H}$ (the latter is very light);
- three different probability distribution: uniform, skew ($p^{max}=3\%$) and very skew ($p^{max}=16\%$).

We will see results on:

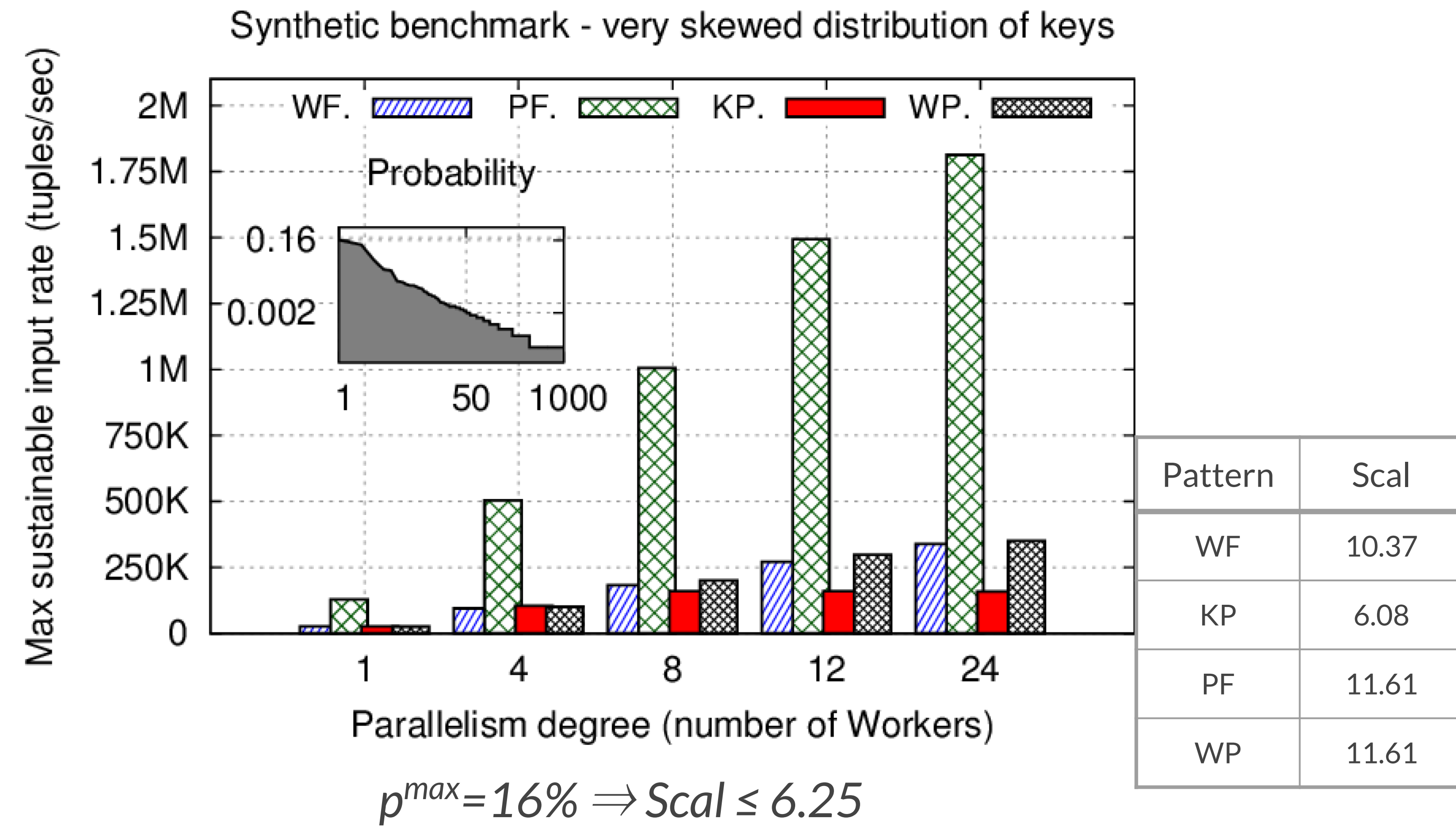
- maximum sustainable rate for uniform and very skew distributions;
- latency.

RESULTS



Pattern	Scal
WF	11.85
KP	12.02
PF	11.77
WP	11.83

RESULTS

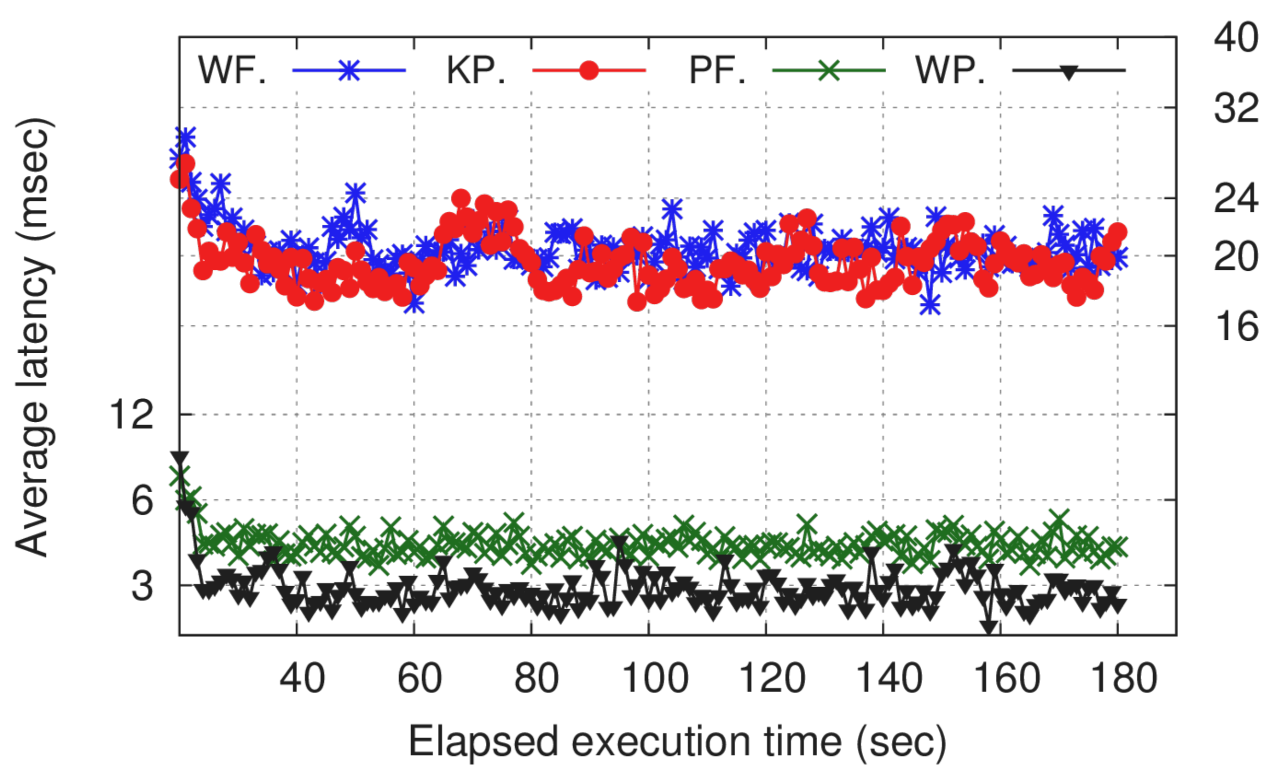


Pattern	Scal
WF	10.37
KP	6.08
PF	11.61
WP	11.61

$p^{max} = 16\% \Rightarrow Scal \leq 6.25$

RESULTS

Input rate: 200KT/s, 10 Workers for WF,KP,WP; 2 Workers for PF



CONCLUSIONS

DaSP is a paradigm focusing on real-time processing of flows of data. Intra-operator parallelism is necessary but still not completely exploited:

- we have characterized 4 different patterns, that cover a wide variety of situations;
- they can be implemented on top of existing skeleton framework or SPE;
- proof-of-concept implementation in Fastflow.

Extensions:

- distributed implementation;
- autonomic management: scaling and load balancing.

Thank you!

Questions?