TDD25
Distributed Systems

Distributed Stream Processing

Christoph Kessler

IDA
Linköping University
Sweden

LINKÖPING
UNIVERSITY

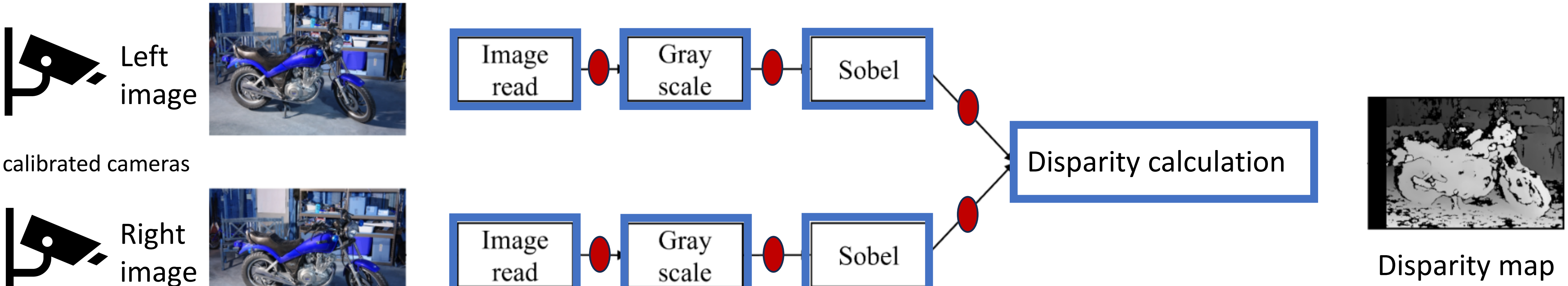
Data Stream

- Possibly infinite sequence of stream elements
- Elements can be values of base type, structs, strings, images, video frames, …,
  key-value pairs,
  time-stamped or indexed
- Encoding: can be
  - binary-native (platform-specific, not portable – use at best within same computer),
  - binary-portable (framework-specific), or
  - text-based (e.g. CSV, JSON, YAML, XML) – wasteful in space and time, but universally portable and even human-readable
- Streams may have a fixed or varying **data rate** (*velocity*, #elements per second)
- In certain stream processing applications, some stream elements might even be **malformed** due to errors in preceding stages – needs be handled at processing

Stream Processing Application

- **Stream sources** (input streams)
  - E.g., from sensors, network connection, (continuous) database query, file read position
- **Stream sinks** (output streams)
  - E.g., to display, file, actuators
- **Stream processing tasks / stream operators**
  - Transform input to output streams by elementwise computation or aggregate input stream elements (e.g., elements with same key)
  - executed whenever new data has arrived
    - execution of a task instance can be input-triggered or time-triggered)
    - may thus exist as long as the overall application itself.
  - Can be stateless or stateful
  - Have input and/or output **ports** that connect to streams from stream sources, to stream sinks or to/from other stream operators' ports
- **Internal streams** connect (ports of) multiple dependent stream processing tasks
  - Stream-**producer** tasks/ports forward data to stream-**consumer** tasks/ports

Example Application: Stereo Depth Estimation

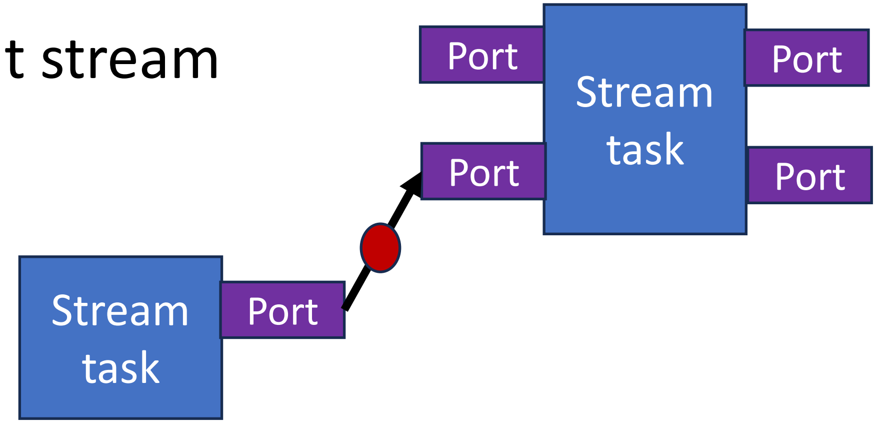


Workflow describing a complex stream processing pipeline

- **Tasks**: stream-processing tasks, stream operators
  - Input and output **ports** – one per input and per output stream
- **Streams**: FIFO-queue-buffered dataflow

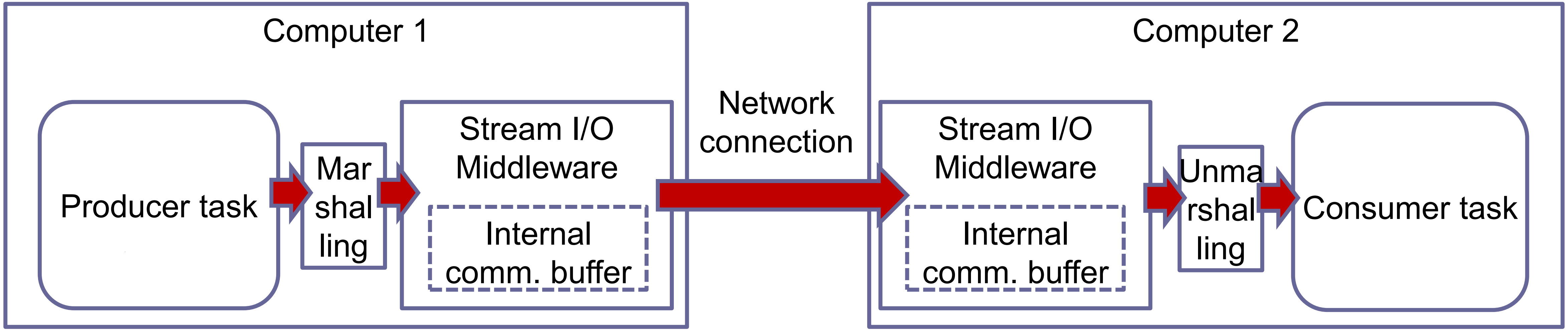
Formally: A bipartite directed acyclic graph (DAG)

G = (Tasks, Streams, Data-flow edges)



Stream Processing Systems

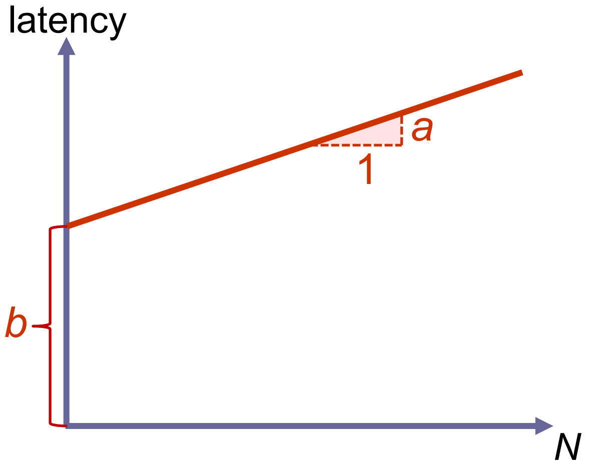
- Where a stream's producer and consumer are mapped to different computers, we need **middleware** and **glue code** (cf. CORBA for RPC/RMI) to mask heterogeneity.



- Optionally, windowing (batching) techniques can be used to aggregate multiple subsequent stream elements to coarser-grained units of further processing and forwarding
  - Often done for input streams from stream sources

Background: Work Agglomeration by Batching

Goal: Reduce average per-element overheads in communication and processing



- **Time** (latency) for **communicating** a message of *N* bytes is usually linear in *N*:

  $T_{msg}(N) = a*N + b$ for constants* $a, b > 0$

  (* a, b still depend e.g. on network distance between sender and receiver)

  - *a*: **per-byte transfer time**, network throughput - typically *a* ~ few nanoseconds
    - limited by network connection bandwidth
  - *b*: **per-message transfer overhead** (usually, ***b* >> *a***) - typically *b* ~ microseconds
    - due to network software overheads, message headers (metadata, e.g. addresses, checksums), routing, physical network delays

- **Time** for elementwise **processing** (loading, parsing, transforming, storing) a batch of *M* elements is often linear in *M*:

  $T_{transf}(M) = c*M + d$ for constants $c, d > 0$

  - *c*: **per-element processing time**
    - Limited by memory bandwidth and computational bandwidth of the processor
  - *d*: **batch-processing startup time**
    - Due to memory latency, parsing overheads, launching processes/threads/kernels, etc.

→ Communicating messages and processing batches of **1** element size is possible, but very **inefficient**.
The larger the batch size, the lower the amortized per-element overheads (*b* / #elements in message, *d* / *M*).

Windowing Techniques

- Application-specific **windowing** techniques can be applied (e.g. to stream sources) to **aggregate** multiple subsequent stream elements into coarser-grained units of work for further processing
  - A generalization of **batching**
  - Trades higher throughput (lower per-element overhead) for longer latency for some stream elements
    - Granularity of work becomes large enough to reduce per-element overhead (e.g. for data transfers in larger packets) and possibly use parallel computing or accelerators in tasks to speed up processing.
    - But waiting for more input elements to arrive in order to form larger batches also increases end-to-end latency for the earlier ones – linearly with the batch/window size
  - Whenever the current window on an input stream is considered **full**,
    it will be **dispatched** as a packet for further stream processing,
    and the window on the input stream is moved (slided or tumbled).

Windowing Techniques

When is a window considered **full** for the application?

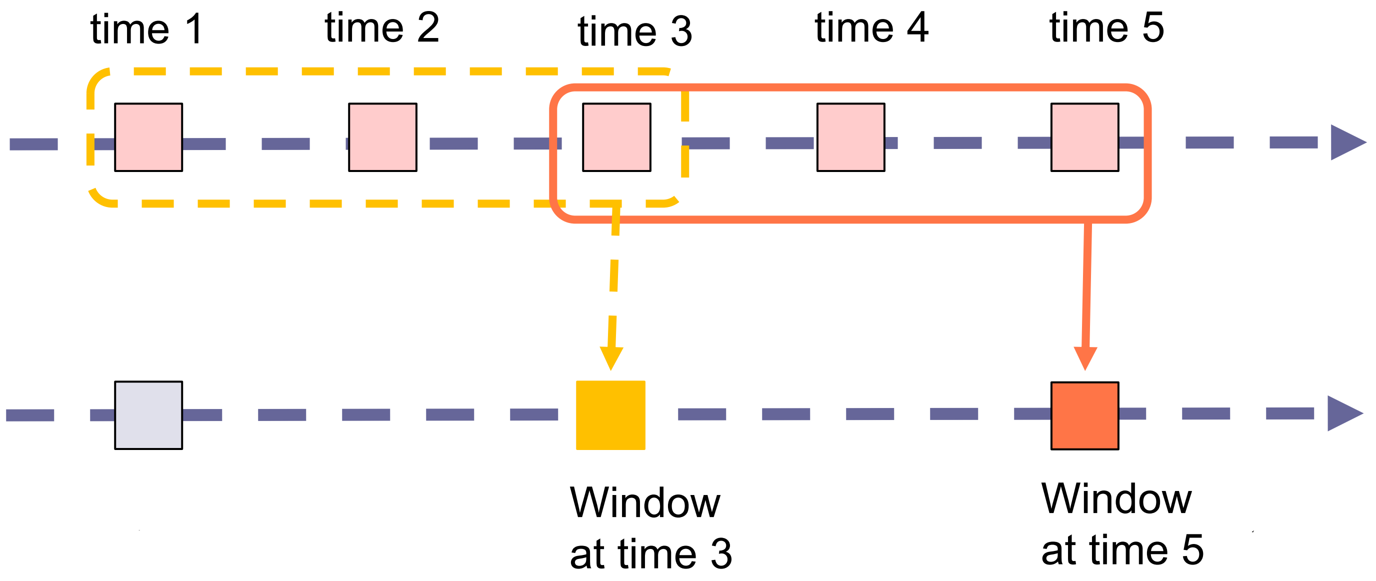
- **Capacity based**: if it contains *K* elements or no further elements can be stored
  - Corresponds to traditional batching (as used e.g. in ANN training and inference)
- **Wallclock time based**: whenever *S* seconds have passed since the window was moved last (also if there is some capacity left)
- **Timestamp based**: if an element with a timestamp newer than some threshold *S*' since the timestamp of the oldest element in the window has arrived
  - Time-based windowing may produce window instances that contain varying amounts of elements.

Subsequent window instances may overlap or not

- **Tumbling window**: The window is moved so the next window instance contains only new elements. Different window instances contain different stream elements.
- **Sliding window**: Subsequent window instances overlap partially so that recent stream history is considered together with the elements arrived since the last move.

Windowing Example: Spark Streaming

- Can define a sliding window over a source DStream



Window length (here 3)
Slide length (here 2)
→ Overlap size (here 1)

Every time the window slides over a source DStream, the source elements that fall within the window are forwarded as input to a Spark batch-processing computation (as an “RDD”)

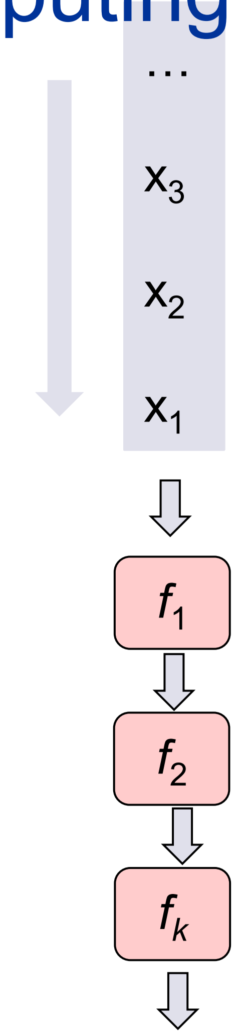
```
# Example:  Reduce last 30 seconds of data, every 10 seconds:
windowedWordCounts =    \
  pairs.reduceByKeyAndWindow( lambda x, y: x + y, lambda x, y: x - y, 30, 10 )
```

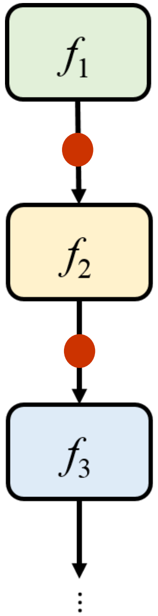
Pipelining (algorithmic design pattern in parallel computing)

Applicable if a sequence of dependent computations/tasks (f_1, f_2, ..., f_k) is elementwise applied to data sequence $\boldsymbol{x} = (x_1, x_2, x_3, ..., x_n)$

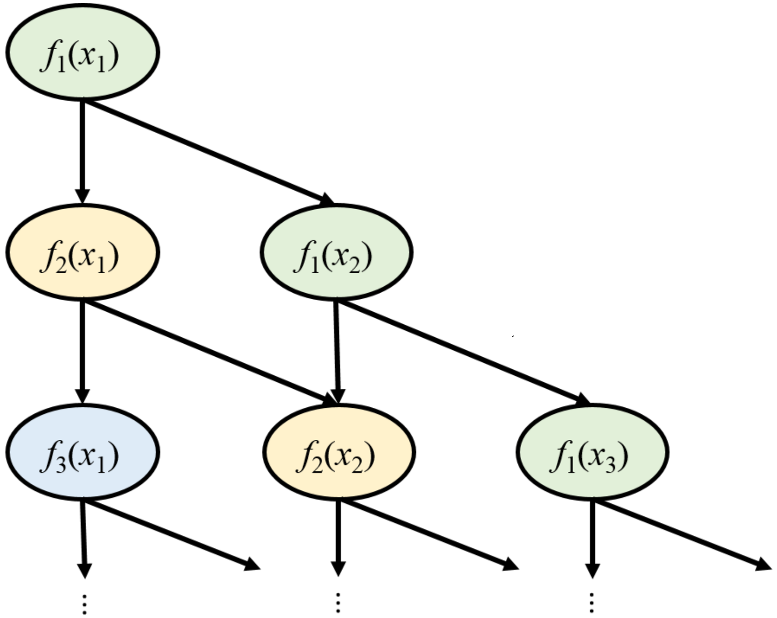
For fixed x_j, must compute $f_i(x_j)$ before $f_{i+1}(x_j)$

… and $f_i(x_j)$ before $f_i(x_{j+1})$ if the tasks f_i have a *run-time state* (*stateful tasks*)





Pipeline stage/task dependence graph



Task instances dependence graph (trace)

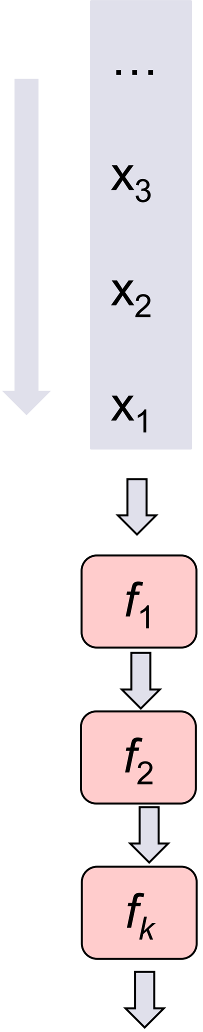
Remark: It is possible to explicitly create a new one-shot task for each instantiation (input) $f_i(x_j)$ of a pipeline task f_i. However, this involves high runtime overhead for dynamic task creation, resource allocation/mapping and scheduling. For long-running pipelines with statically predictable task workloads it is often better to have each pipeline task assigned to a fixed resource for executing all its instances.

Pipelining

Applicable if a sequence of dependent computations/tasks ($f_1, f_2, ..., f_k$) is elementwise applied to data sequence $\boldsymbol{x} = (x_1,x_2,x_3, ..., x_n)$

For fixed x_j, must compute $f_i(x_j)$ before $f_{i+1}(x_j)$

… and $f_i(x_j)$ before $f_i(x_{j+1})$ if the tasks f_i have a *run-time state*



Parallelizability: Overlap execution of all f_i for k subsequent x_j

Round 1: compute $f_1(x_1)$

Round 2: compute $f_1(x_2)$ *and* $f_2(x_1)$

Round 3: compute $f_1(x_3)$ *and* $f_2(x_2)$ *and* $f_3(x_1)$

...

At k rounds we reach the steady state of the pipeline (all tasks are active). After round n, the pipeline starts draining (no new elements entering).

Total time*:* $O\ (\ (n+k)\ \mathbf{max}_i\ (\ time(f_i)\ ))$ if we have k processors

Still, requires good mapping of the tasks f_i to a fixed set of processing resources for even load balancing – often, static mapping (done before running)

Leveraging More Parallelism: Stream Farming

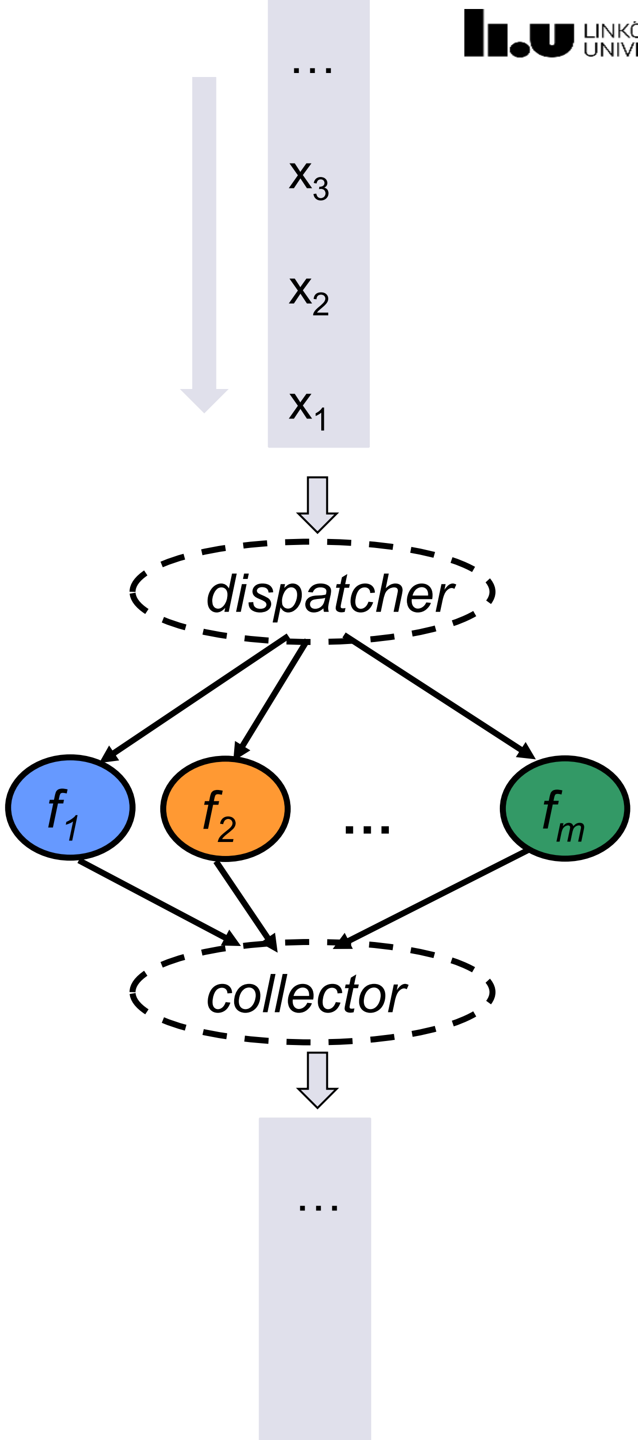
A heavy pipeline stage/task *f* can easily dominate the overall time!

Idea: Speed up that task by parallel processing of multiple subsequent data packets where this is acceptable for the application,
by distributing the computation of *f* across multiple resources
(**stream farming** – also known as “replication parallelism”)

- For **stateless** tasks: Not a problem. E.g., dispatch data round-robbin.
- For **stateful** tasks: must observe application-specific constraints, e.g. if state is key-specific, might be OK to process elements with different keys separately, while computations of elements with same key have to take place in same f_i in their original order.
- Needs a **dispatcher** task (dynamic scheduler) and a **collector** task (putting result stream in right order). These manager tasks will also consume some resources.

Caution: In many cases, the original order of stream elements must be maintained after processing, i.e. in the output stream.

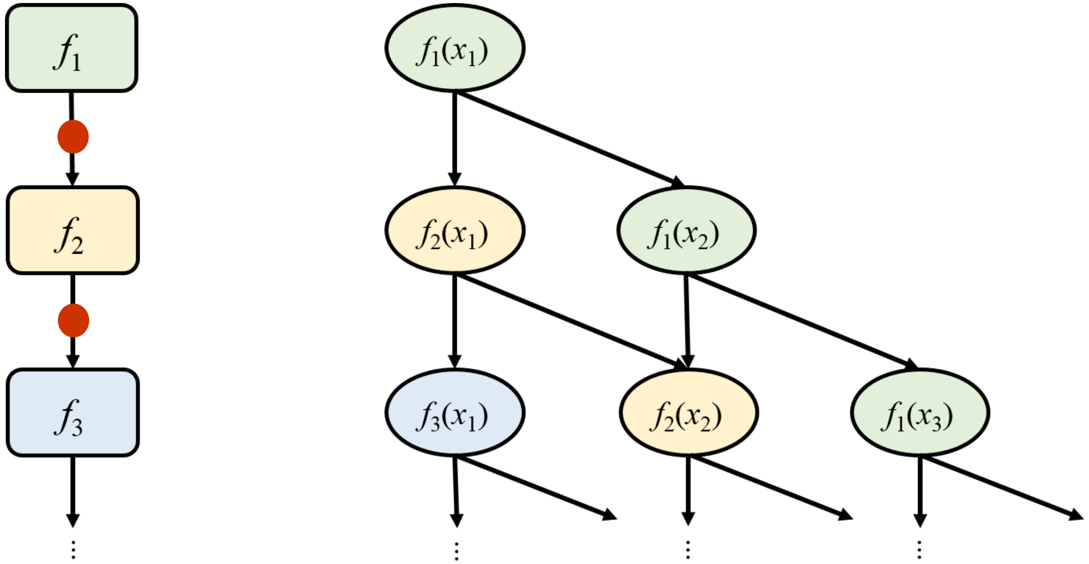
An alternative is to allow **internally parallel** task implementations.



Foundations: Dataflow Programming

Executing stream-processing tasks in a pipeline

→ Trace of *partially dependent* task instances



Well-established **execution models** for such **data flow graphs** of periodic tasks have been defined in the literature, e.g.

- **Kahn Process Networks** (KPN)
  - assumes FIFO channels with **unbounded** buffering capacity (for unknown number of in-flight task instances)
    → any fixed buffering capacity might be insufficient
    (could deadlock if not handling backpressure by slowing down upstream tasks)
- **Synchronous Data Flow** (SDF)
  - assumes **fixed, statically known data production / consumption rates** for each task instance
    → fixed buffering capacities sufficient
- ...

Stream-Processing Frameworks

- API for application writer to specify the stream-processing pipeline DAG structure, other pipeline properties, and the stream-processing tasks' implementations
- Framework provides deployment, resource management, portable communication (middleware), buffering, windowing
- Varying support for distribution, heterogeneous nodes, parallel tasks, accelerator usage, fault tolerance, stream encryption, portability of stream operator specifications, etc.

- **Examples**:
  - StreamIt (https://groups.csail.mit.edu/cag/streamit)
  - Spark-Streaming
  - Apache Storm (https://storm.apache.org)
  - Apache Flink (https://flink.apache.org)
  - Apache Kafka Streams (https://kafka.apache.org)
  - SkePU-Streaming

SkePU-Streaming

Reference:

Distributed Pipelining of Portable Data-Parallel Skeleton Computations for the Heterogeneous Edge-Cloud Continuum
Int. J. of Parallel Programming, to appear, 2026

August Svensson, Fabio Crugnola, August Ernstsson,
Sajad Khosravi, Sebastian Litzinger, Alexander Lindskog,
Christoph Kessler

Linköping University, Sweden

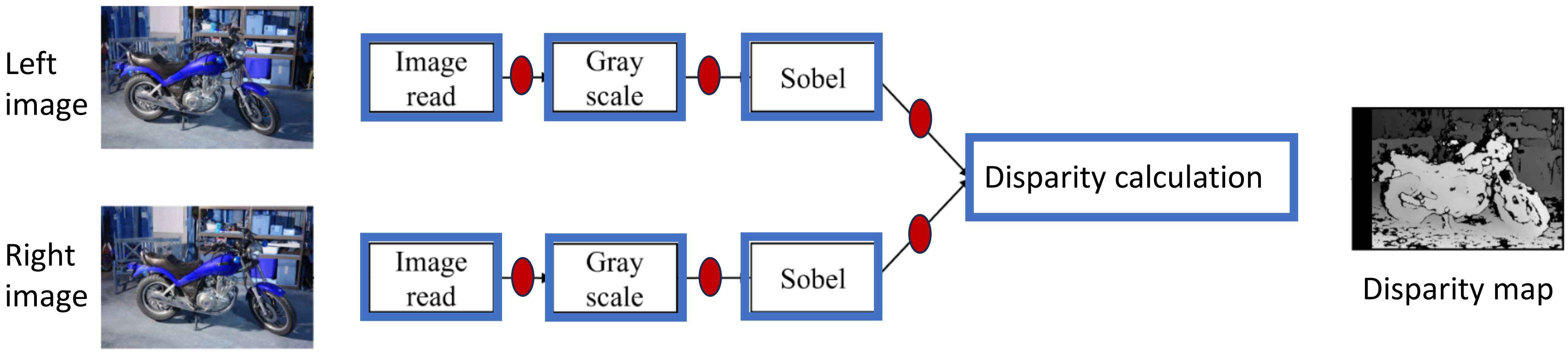
LINKÖPING UNIVERSITY

Adaptive Software for the Heterogeneous Edge-Cloud Continuum

SWEDISH FOUNDATION *for* STRATEGIC RESEARCH

This work has been partly funded by SSF project FUS21-0033 ASTECC (https://astecc.gitlab-pages.liu.se/).

Application Model



Focus here:

Workflows describing complex stream processing pipelines

- Tasks: Multi-variant stream-processing tasks
  - Characterized by workload, intra-task parallelism (**moldable** tasks), algorithmic/platform-specific variants for heterogeneous systems …

Embrace heterogeneity, intra-task parallelism

- Streams: FIFO-buffered dataflow (cf. Kahn Processing Networks)

Encapsulation, explicit inter-task (here, pipeline) parallelism

Formally: A bipartite DAG (Tasks, Streams, Data-flow edges)

Challenges

- Native execution on heterogeneous distributed hardware
- Convenient, portable, high-level specification of
  - Heterogeneous intra-task parallelism
  - Pipeline flow structure and properties to connect the tasks
- Automatic deployment to a distributed target system
- Easy reconfiguration

Skeleton programming, e.g. SkePU: Single source, many backends

Can this be achieved with an (if possible, non-intrusive) extension of SkePU?

SkePU

https://skepu.github.io

- C++-based high-level programming framework for heterogeneous parallel systems
  - Developed since 2010 as an open-source effort at Linköping University, Sweden.
- Provides a set of **multi-variadic skeletons** for common data-parallel patterns such as map, reduce, stencil, scan, ...
  - **Generic in**: operand number (**arity**), operand **shape**, operand **access pattern** (elementwise, random-access, neighborhood region, …), **element type**, and **user function** (per-element operator)
  - Can be parameterized with problem-specific sequential C++ code (**user-functions**), resulting in **skeleton instances** which can be invoked like hand-written C++ functions.

```
#include <skepu>

int main ()
{
 skepu::Matrix<int> m(5, 5);

 auto sum_function = skepu::Reduce(     // Matrix sum reduction
     [](int a, int b) { return a + b; }
 );

 std::cout << "Matrix␣Sum:␣" << sum_function(m) << std::endl;
}
```

Data-container for 2D skeleton operands

Builds a Reduce skeleton instance

User function (here, lambda)

SkePU https://skepu.github.io

- C++-based high-level programming framework for heterogeneous parallel systems
  - Developed since 2010 as an open-source effort at Linköping University, Sweden.
- Provides a set of **multi-variadic skeletons** for common data-parallel patterns such as map, reduce, stencil, scan, ...
  - **Generic in**: operand number (**arity**), operand **shape**, operand **access pattern** (elementwise, random-access, neighborhood region, …), **element type**, and **user function** (per-element operator)
  - Can be parameterized with problem-specific sequential C++ code (**user-functions**), resulting in **skeleton instances** which can be invoked like hand-written C++ functions.
- Each skeleton provides **multiple backends**
  - parallel and accelerator-specific implementations (called backends) for the various native parallel programming models for CPU (sequential C++, OpenMP), single- and multi-GPU execution (OpenCL, CUDA), …
  - Static or dynamic backend selection
  - Separately for each skeleton instance call, or use global default settings, or use SkePU's auto-tuning backend selection
- STL-like array-based generic **data-container** abstractions
  - Vector<>, Matrix<>, Tensor3<>, Tensor4<>
  - Wrapping C/C++ array operands to be used in skeleton instance calls
  - Memory management, coherent software caching, transfers, and some other run-time optimizations

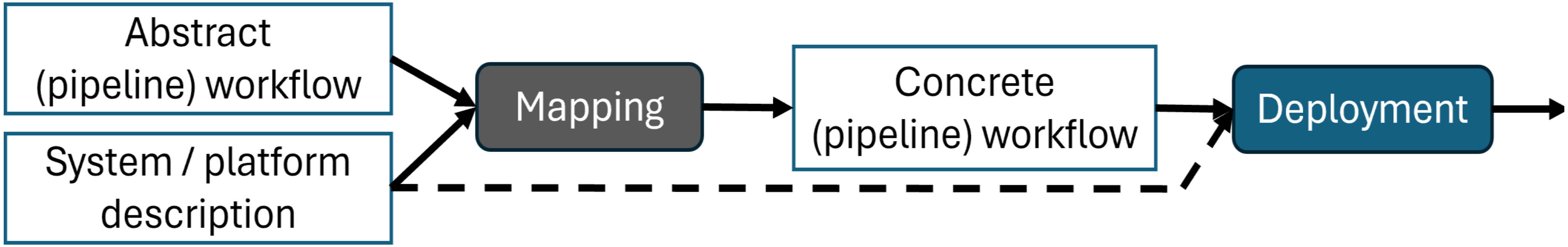
Abstract and Concrete Workflows

Abstract workflow:

- Directed bipartite graph (tasks, data-flow nodes, data-flow edges)

Concrete workflow:

- Abstract workflow + Mapping/deployment information for target system


Abstract (pipeline) workflow
System / platform description
Mapping
Concrete (pipeline) workflow
Deployment

SkePU-Streaming Design (1)

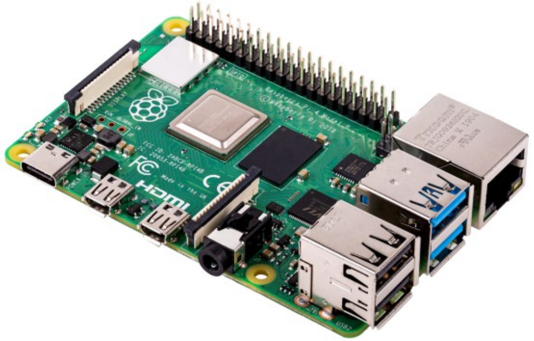
- **Global pipeline workflow and properties** are defined in a pipeline workflow description in (customized) JSON format

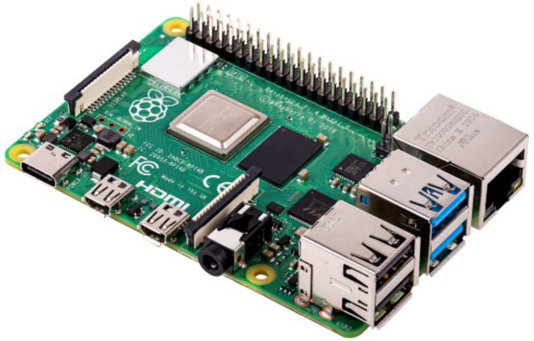
  Main structure:

  - List of named **hosts** (possible deployment targets)
    - each host with properties (usually included from separate specification file)

Hosts specification

```
resources {
    hosts: [
        { id: rbpi_2
          executor_affinity: "0"
          crosscompile: True
          sysroot: "/home/fabcr/Documents/rbpisysroot"
          host: "192.168.0.102"
          cxx: "aarch64-linux-gnu-g++"
          arch: "aarch64-linux-gnu"
        }
        { id: rbpi_3
          crosscompile: True
          …
        }
       …
    ]
}
```





⋮

SkePU-Streaming Design (1)

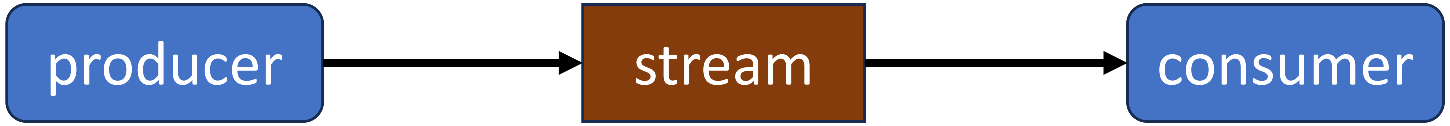
- **Global pipeline workflow and properties** are defined in a pipeline workflow description in (customized) JSON format

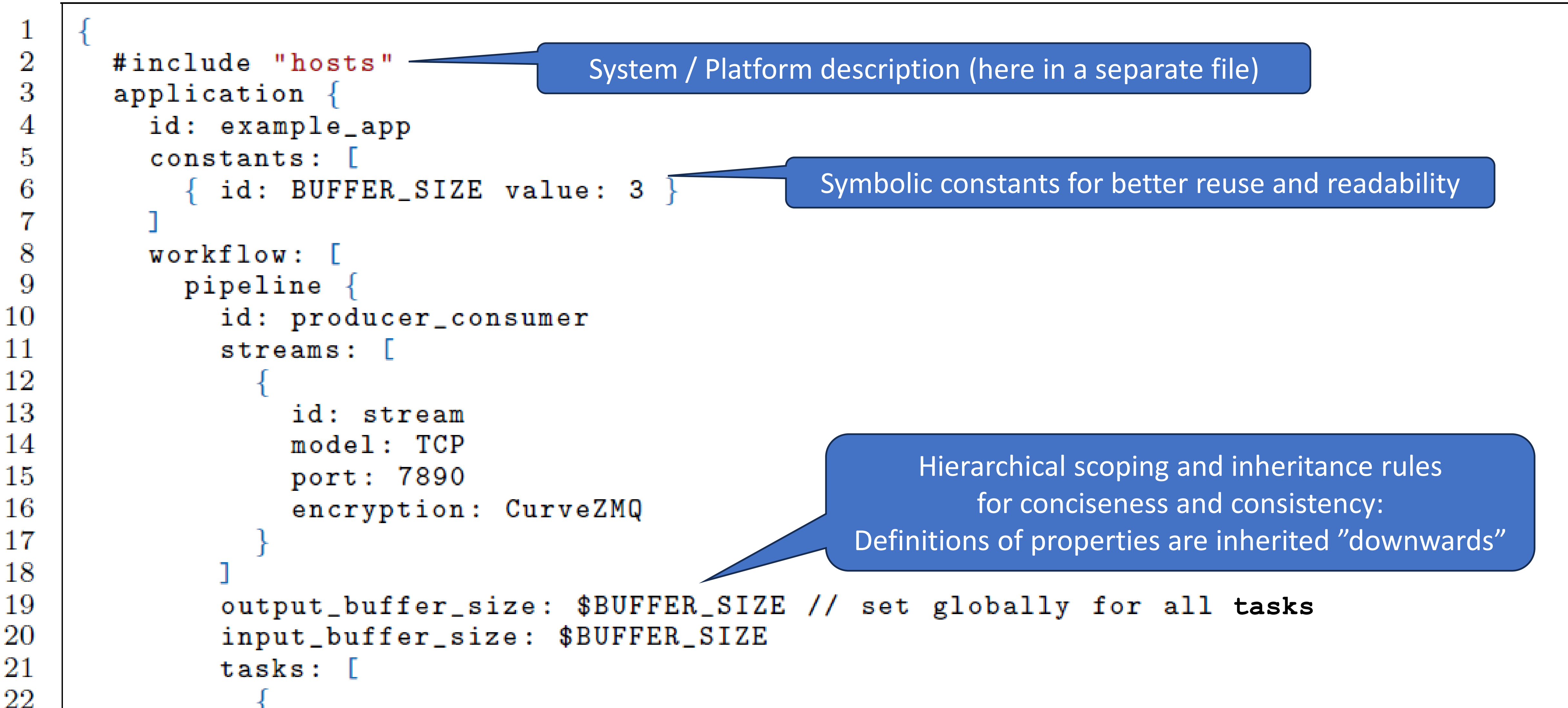
  Main structure:

  - List of named **hosts** (possible deployment targets)
    - each host with properties (usually included from separate specification file)
  - List of named **streams**
    - each stream with properties (e.g., buffering capacity, encryption scheme, transfer method)

Example

Producer-Consumer pipeline





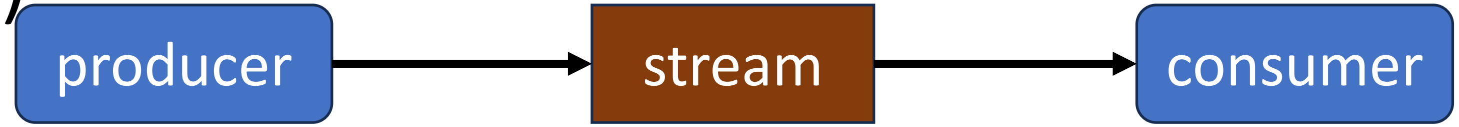
```
{
  #include "hosts"
  application {
    id: example_app
    constants: [
      { id: BUFFER_SIZE value: 3 }
    ]
    workflow: [
      pipeline {
        id: producer_consumer
        streams: [
          {
            id: stream
            model: TCP
            port: 7890
            encryption: CurveZMQ
          }
        ]
        output_buffer_size: $BUFFER_SIZE // set globally for all tasks
        input_buffer_size: $BUFFER_SIZE
        tasks: [
          {
```

System / Platform description (here in a separate file)

Symbolic constants for better reuse and readability

Hierarchical scoping and inheritance rules
for conciseness and consistency:
Definitions of properties are inherited ”downwards”

Example (cont.)



```
tasks: [
  {
    id: producer
    host: $rbpi_2
    output_streams: [$stream]
    variants: [
      skepu {
        id: producer_variant_1
        backend: openmp
        build: {
          type: make
          source: "examples/producer-consumer/producer"
          output: "producer"
        }
      }
    ]
  }
  {
    id: consumer
    host: $rbpi_3
    input_streams: [$stream]
    variants: [
      skepu {
```

Producer task is a stream source: no input stream operand

External backend selection

Consumer task is a stream sink: no output stream operand

SkePU-Streaming Design (1)

- **Global pipeline workflow and properties** are defined in a pipeline workflow description in (customized) JSON format

  Main structure:

  - List of named **hosts** (possible deployment targets)
    - each host with properties (usually included from separate specification file)
  - List of named **streams**
    - each stream with properties (e.g., buffering capacity, encryption scheme, transfer method)
  - List of named **stream-processing tasks**, each with
    - Ingoing and outgoing streams (ports, operands) – variadic
    - Task properties and mapping information (resource assignment, core pinning)
    - **Implementation variants** (current options: SkePU with backends; sequential C++, OpenMP)
      - Each with deployment information: source directory/files, compilation flags etc.
      - One of them set as default variant (usually, SkePU)

  Extensions over JSON:

  #include, expressions, symbolic constants, C/C++ comments, simplified syntax { } ,

  Implementation: Custom JSON parser based on pyparsing module.

SkePU-Streaming Design (2)

- Each **stream-processing task** is defined as a separate SkePU program
  - Source code and local makefile located in its own subdirectory
  - Portable multi-backend data-parallelism
    - Skeleton instance parallelism limited to resources within one node
  - **Access to stream operands**
    by new skepu::io standard library extensions overloading operators <<, >>
    and specifying the accessed input or output port number

Producer Task, Source Code

```
#include <skepu>
#include <skepu-streaming>

int main ()
{
 skepu::io::streams::init();

 skepu::Matrix<int> m(5, 5);

 skepu::io::streams::endless ( []() {
     … // produce m
     skepu::io::output_stream[0] << m;
 });
}
```

The stream-processing task body

endless() construct iterates until all input streams see *EndOfStream* (default)

Flushes and serializes matrix container m contents to first output stream (output port 0)

Consumer Task, Source Code

```
#include <skepu>
#include <skepu-streaming>

int main ()
{
 skepu::io::streams::init();
 skepu::Matrix<int> m(5, 5);

 auto sum_function = skepu::Reduce(      // Matrix sum reduction
     [](int a, int b) { return a + b; }
 );

 skepu::io::streams::endless( [&]() {
     if (!(skepu::io::input_stream[0] >> m)) continue;
     std::cout << "Matrix␣Sum:␣" << sum_function(m) << std::endl;
 });
}
```

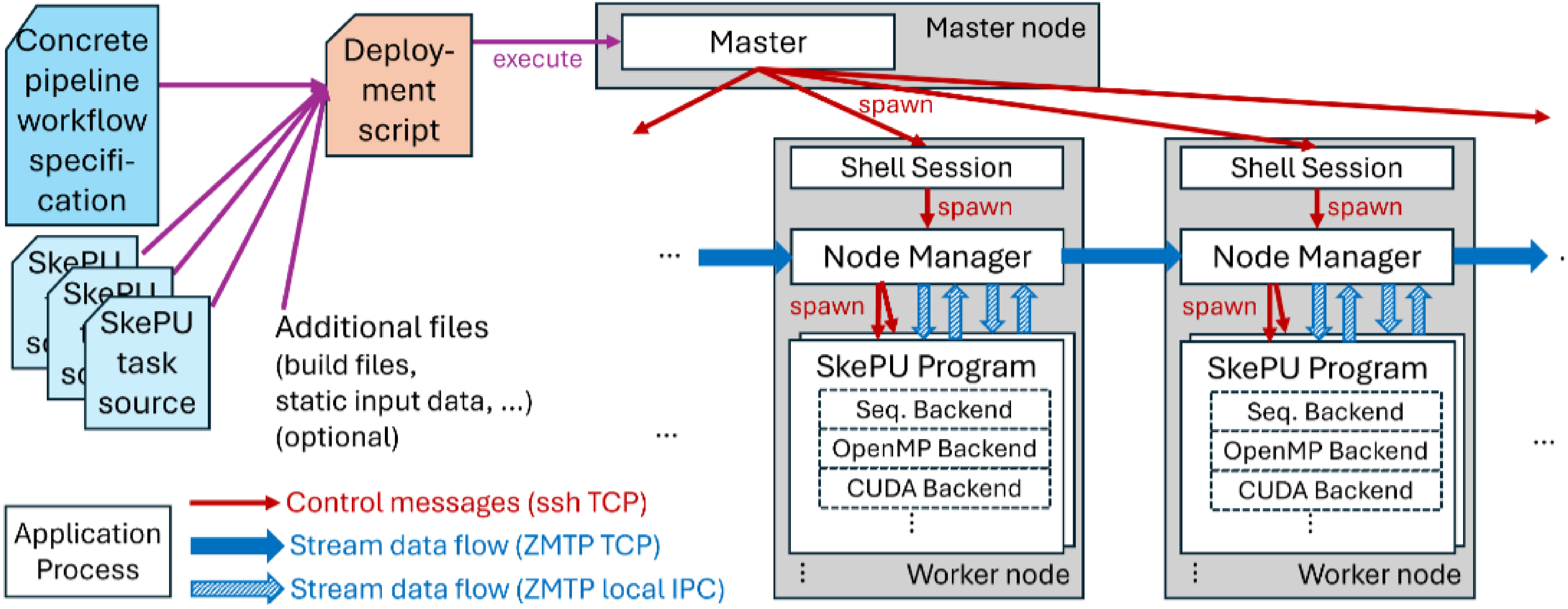
No backends chosen here
→ external global settings apply

Deserialize to matrix container m from first input stream (inp. port 0)

The stream-processing task body

Implementation

Deployment of a SkePU-Streaming workflow and runtime management



Implementation Details (1)

NodeManager component handles external routing
and feeds SkePU application with stream elements

Communication library based on ZeroMQ (https://zeromq.org)

- ZeroMQ Message Transport Protocol (ZMTP)
- For node-manager – node-manager communication (TCP)
  and node-manager – SkePU communication (Unix domain sockets)
- **Serializers** implemented for skepu::Vector and skepu::Matrix containers
  - Before serialization, the container contents are flushed to main memory
- We use binary encoding for performance reasons,
  and because SkePU seldom is used for text processing.
- For TCP stream **encryption and decryption** as requested by the workflow, we use for now the CurveZMQ mechanism provided by ZeroMQ

Implementation Details (2)

Deployment script

- turns into the **master process**
  - Parses the pipeline description
  - Resolves eventual IP addresses of hostnames, and for each task opens up an SSH connection to the worker node specified in the concrete workflow for that task
  - Generates a global Makefile
  - Builds and transfers task binary to respective target node (alternatively, local compilation)
  - Launches execution of node manager and SkePU task processes
- Implemented in Python, as not considered performance-critical
- Concurrency needed e.g. for transferring multiple files concurrently

Experiments: ASTECC Testbed

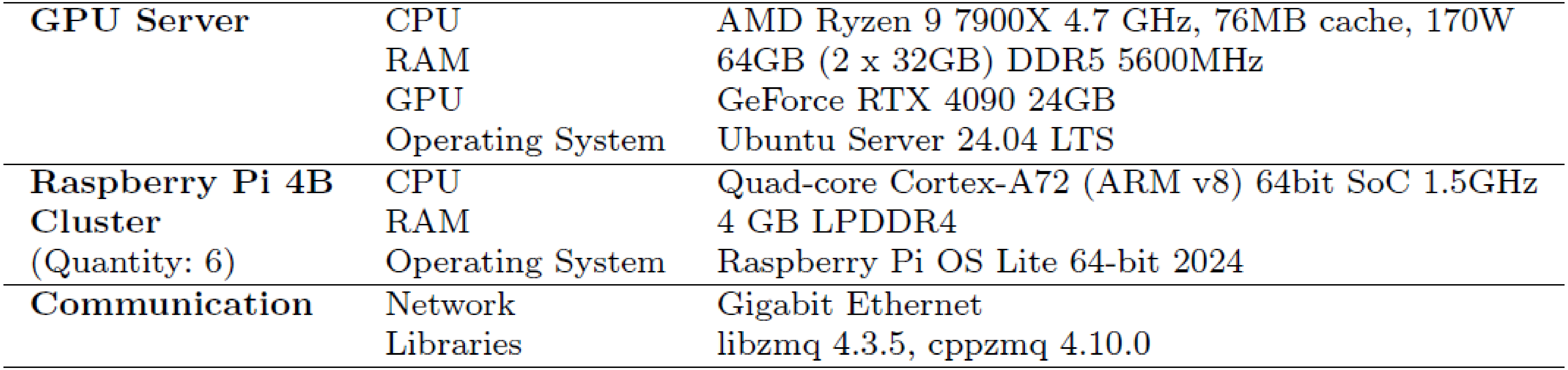
Mapping targets:

”Cloud”:

- GPU server
  - AMD Ryzen 9 7900X, 12 cores, 5.6 GHz
  - Nvidia RTX 4090

”Device, Edge resources”:

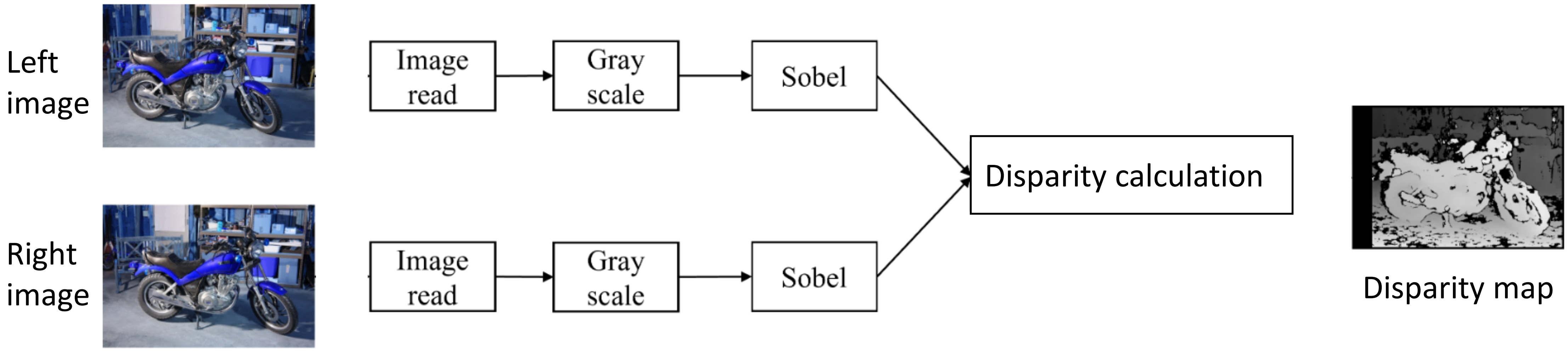
- Raspberry Pi 4B
  - Quad-core ARM Cortex A72, 1.5 GHz



GPU Server	CPU	AMD Ryzen 9 7900X 4.7 GHz, 76MB cache, 170W
	RAM	64GB (2 x 32GB) DDR5 5600MHz
	GPU	GeForce RTX 4090 24GB
	Operating System	Ubuntu Server 24.04 LTS
Raspberry Pi 4B Cluster (Quantity: 6)	CPU	Quad-core Cortex-A72 (ARM v8) 64bit SoC 1.5GHz
	RAM	4 GB LPDDR4
	Operating System	Raspberry Pi OS Lite 64-bit 2024
Communication	Network	Gigabit Ethernet
	Libraries	libzmq 4.3.5, cppzmq 4.10.0

Example Application

Stereo Depth Estimation (SDE)



Left image

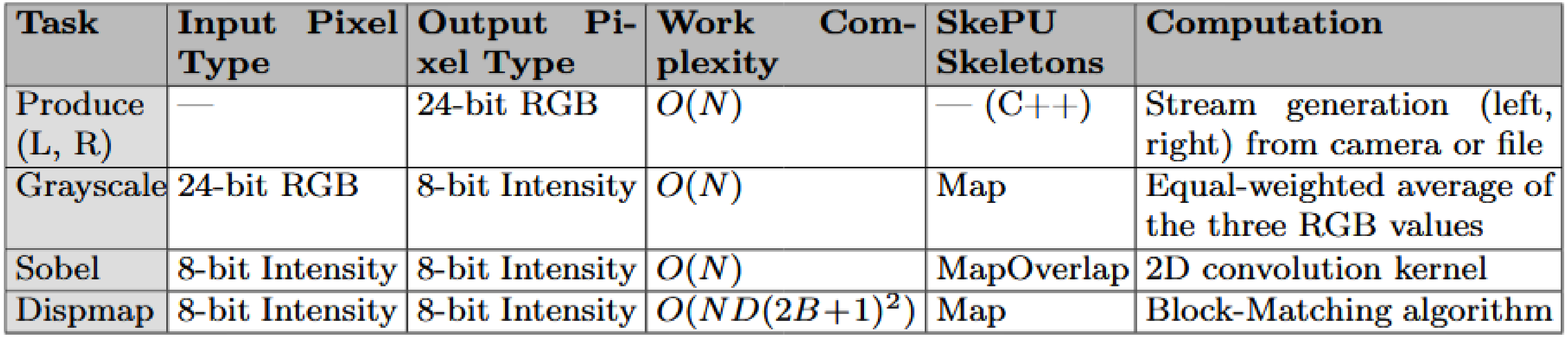
Right image

Image read → Gray scale → Sobel

Image read → Gray scale → Sobel

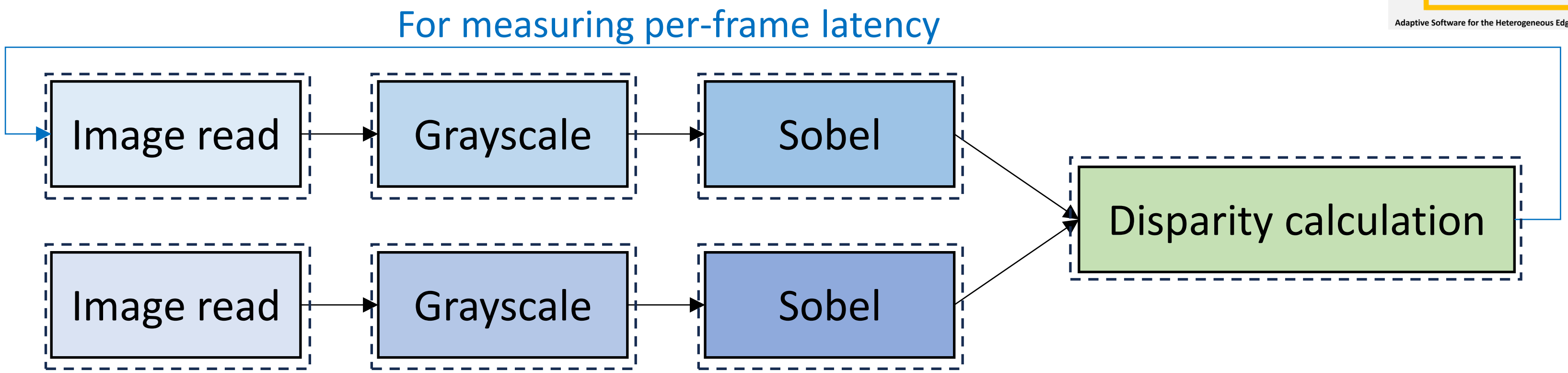
Disparity calculation

Disparity map

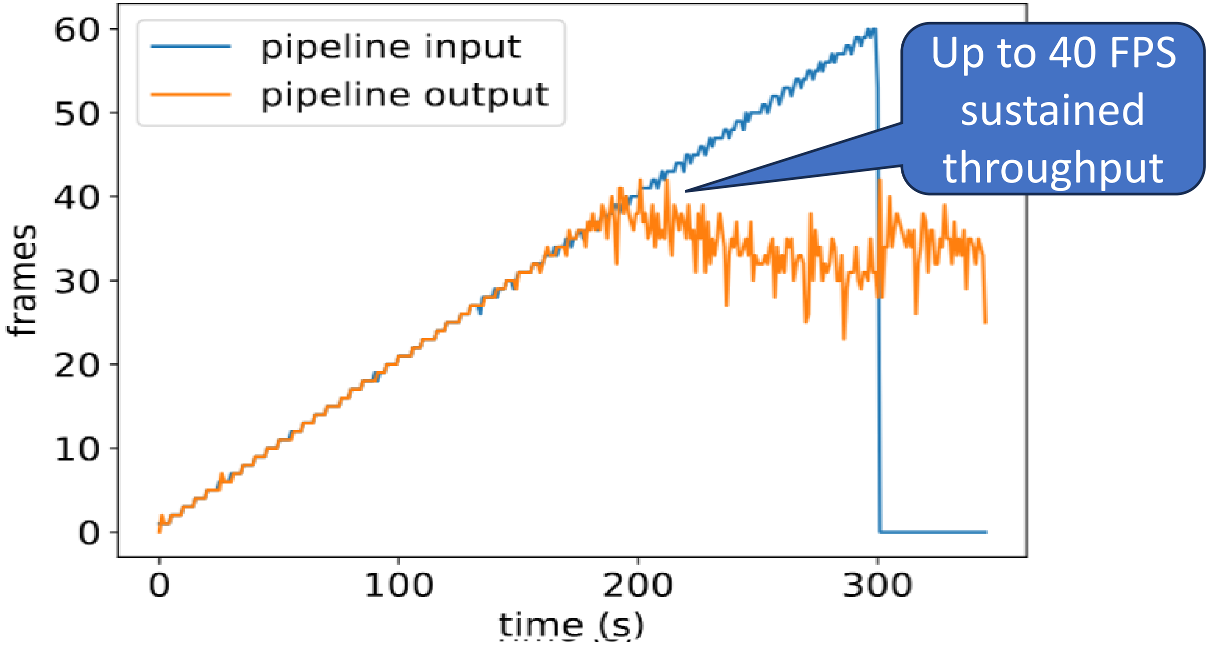


Task	Input Pixel Type	Output Pixel Type	Work Complexity	SkePU Skeletons	Computation
Produce (L, R)	—	24-bit RGB	$O(N)$	— (C++)	Stream generation (left, right) from camera or file
Grayscale	24-bit RGB	8-bit Intensity	$O(N)$	Map	Equal-weighted average of the three RGB values
Sobel	8-bit Intensity	8-bit Intensity	$O(N)$	MapOverlap	2D convolution kernel
Dispmap	8-bit Intensity	8-bit Intensity	$O(ND(2B+1)^2)$	Map	Block-Matching algorithm

Fully distributed mapping



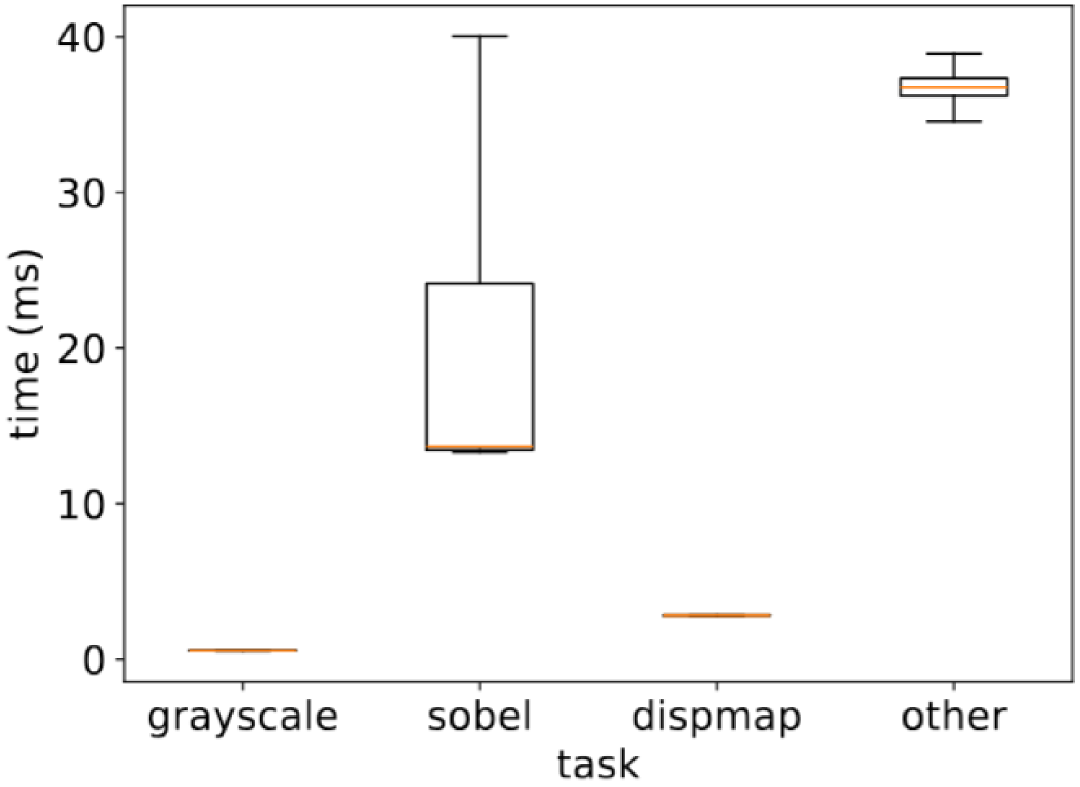
- 741x500px video source starting at 1 FPS and increasing by 1 FPS every 5 seconds
- One pipeline task per node
- Disparity calculation task on GPU server, GPU backend selected; f. other tasks: OpenMP



Up to 40 FPS sustained throughput

For reference:
Executing the entire pipeline with only sequential tasks on a single Raspberry Pi 4 gives a throughput of 0.065 FPS ...

Fully Distributed Mapping: Breakdown of End-to-End Latency

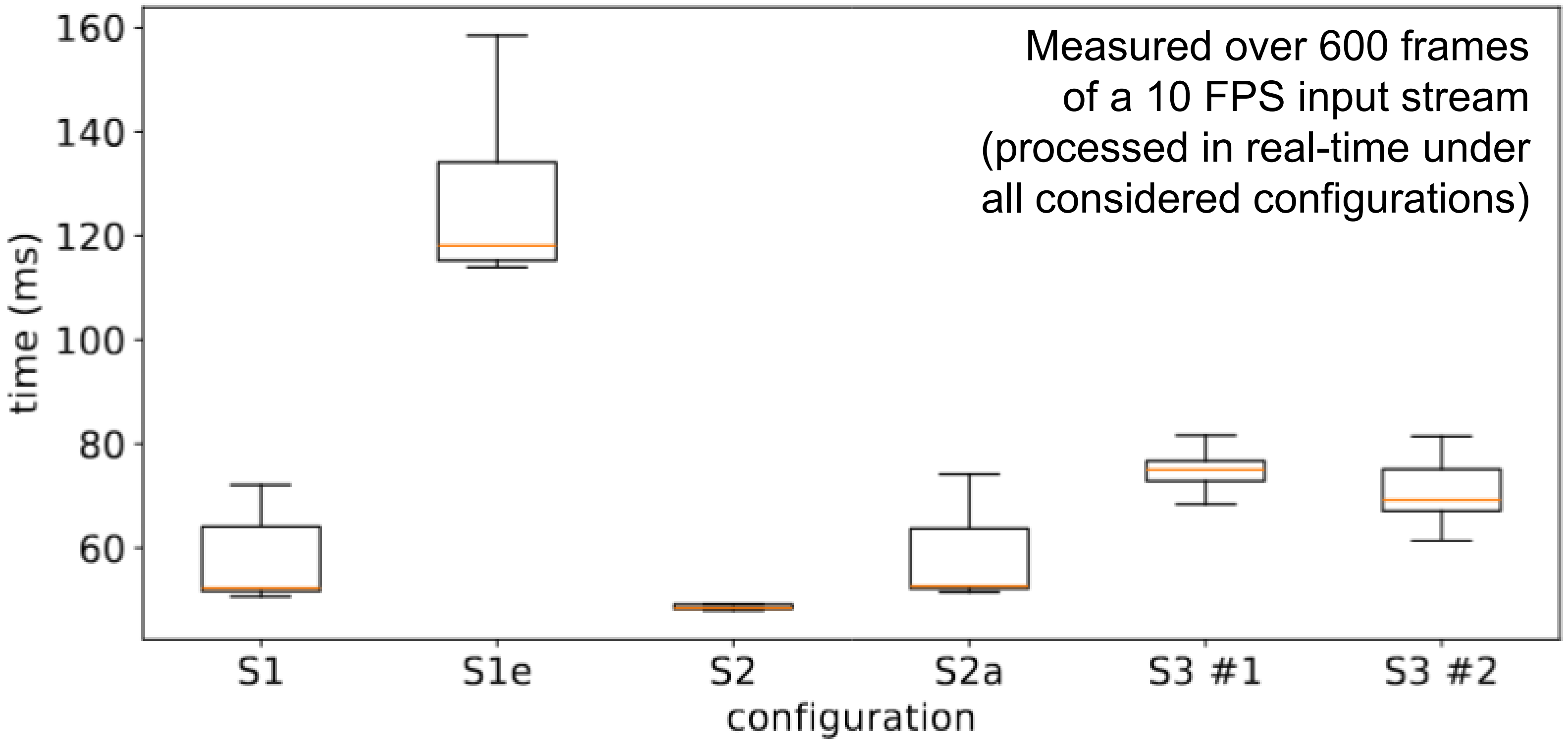


Measured over 600 frames of a 10 FPS input stream (processed in real-time under all considered configurations)

- Disparity calculation (dispmap) is the heaviest task, but executing on the GPU server (RTX 4090) it is not the performance bottleneck.
- “Other” includes 4 network jumps and the intermediate buffering/queuing.

End-to-End Latency Experiments

S1e: Deploying **encrypted** messages on all internal streams with the fully distributed mapping more than doubles end-to-end latency



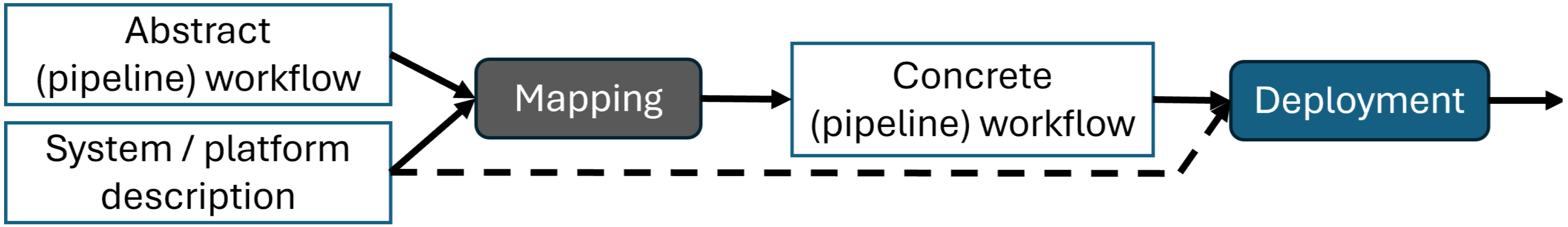
Measured over 600 frames of a 10 FPS input stream (processed in real-time under all considered configurations)

S2: Co-locating Grayscale and Sobel tasks on one node (reduced distribution) saves transfers → shorter latency

S3: Deploying *two* concurrent SDE pipelines time-sharing the same resources only increases latency for each of them a little.

Future Work

- Address some inefficiencies in the prototype
- Further example applications
- More configuration options, e.g. DVFS (in progress)
- Coupling with (crown-scheduling based) optimizer for mapping and resource allocation [Khosravi *et al*. FMEC'24]


Abstract (pipeline) workflow
System / platform description
Mapping
Concrete (pipeline) workflow
Deployment

- Monitoring, tracing, live performance visualization
- Dynamic redeployment for dynamically changing application population
- **Thesis projects available!**

References

SkePU: https://skepu.github.io

SkePU-Streaming:

A. Svensson, F. Crugnola, A. Ernstsson, S. Khosravi, S. Litzinger, A. Lindskog, C. Kessler: ”SkePU-Streaming: Distributed Pipelining of Portable Data-Parallel Skeleton Computations for the Heterogeneous Edge-Cloud Continuum.” Accepted for publication in *International Journal of Parallel Programming*, Feb. 2026, to appear.

General overview:

H. Isah et al.: “A Survey of Distributed Data Stream Processing Frameworks.” *IEEE Access*, 2019.

Foundations (Kahn Process Networks, Synchronous Data Flow):

G. Kahn: ”The semantics of a simple language for parallel programming.” Proc. IFIP Congress on Information Processing. North-Holland, 1974.

E. A. Lee, D. G. Messerschmitt: ”Synchronous Data Flow.” *Proceedings of the IEEE* 75(9), 1987.