COMPSAC 2019, Workshop on Data Flow Models and Extreme-Scale Computing

A Functional Programming Model for Embedded Dataflow Applications

Christoph Kühbacher, Christian Mellwig, Florian Haas, Theo Ungerer

Systems and Networking
Department of Computer Science
University of Augsburg

July 19, 2019

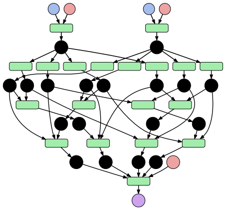
Motivation

- Distributed computing frameworks simplify parallel programming
- Growing interest in big data and machine learning for embedded devices
- Safety-critical embedded devices require fault-tolerance and timing analyzability

Most frameworks are not suitable for embedded systems or are tuned to a specific use-case

Our approach

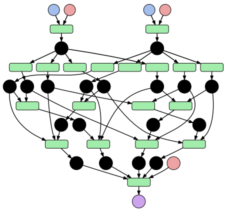
- Programming model similar to Apache Spark
- Dataflow execution based on directed acyclic graph (DAG)
- Easy timing analysis of dataflow execution
- Fault tolerance through actor duplication



Our approach

- Programming model similar to Apache Spark
- Dataflow execution based on directed acyclic graph (DAG)
- Easy timing analysis of dataflow execution
- Fault tolerance through actor duplication

future work



Overview

Overview

Essential Data Collection – RAPID

- **Resilient Analyzable Partitioned Immutable Data Structure**

- Collection consisting of fixed-size data elements
- Divided into partitions of variable size
- Similar to RDDs in Apache Spark
- Based on arrays rather than sets

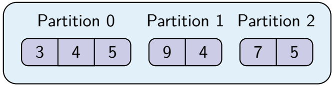


Figure 1: Integer RAPID with three partitions

RAPID Operations

- Create new RAPIDs from existing RAPIDs or standard collections
- RAPID operations build a dataflow graph
- Dataflow execution starts when result is requested

- Functional programming style
- Reduced set of operations compared to Apache Spark
- Suitable for safety-critical embedded systems

RAPID Operations Overview

Initial Operations

- Parallelize
- Distribute

Finalization Operations

- Collect
- Finalize

Transformations

- Map
- Combine
- Reduce
- Zipmap
- Repartition
- Reorder
- Map_Partitions
- Zipmap_Partitions
- Reorder_Partitions
- Append
- Split

Example: High-Level Functionalty of Zipmap

- Element-wise function application on tuples of elements
- Inputs:
  - Multiple RAPIDs with the same element count
  - Zipmap function
- May change partitionings if the partition sizes of inputs do not match

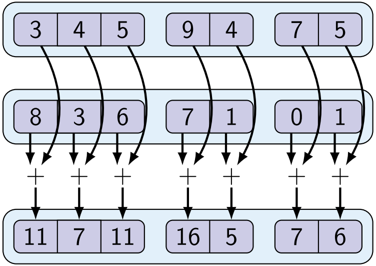


Figure 2: Using Zipmap to add two integer RAPIDs

Example: RAPID program in C++

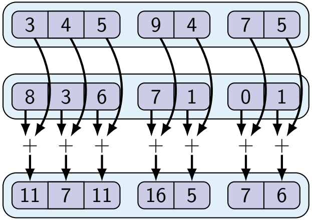
```
rapid_function(add_function, zipmap_t,
     [](int& out, std::tuple<const int&, const int&> in) {
  out = std::get<0>(in) + std::get<1>(in);
});
```

```
std::vector<int> v1 = {3,4,5,9,4,7,5};
std::vector<int> v2 = {8,3,6,7,1,0,1};

auto r1 = parallelize(v1,3);
auto r2 = parallelize(v2,3);

auto r_sum = zipmap({r1,r2},add_function);

auto result = finalize(r_sum);
```



Listing 1: Element-wise addition of integer vectors using RAPID operations

Dataflow Execution

- Directed acyclic dataflow graph
- Separation of graph construction and dataflow execution
- Graph construction at system initialization
- Multiple dataflow executions per graph
- Only one graph execution at a time

Dataflow Graph

- Consists of actor and partition nodes
- Input and output partition nodes (multiple inputs and outputs possible)
- Input nodes can be provided with data for multiple executions

- Constructed from dependencies between RAPID operations
- One-way transformation, retrieval of RAPID program from dataflow graph not always possible

Example: RAPID program and Datenflow graph

```
std::vector<int> v1 =
    {3,4,5,9,4,7,5};
std::vector<int> v2 =
    {8,3,6,7,1,0,1};

auto r1 = parallelize(v1,3);
auto r2 = parallelize(v2,3);

auto r_sum = zipmap({r1,r2},
    add_function);

auto result = finalize(r_sum);
```

Listing 2: RAPID program

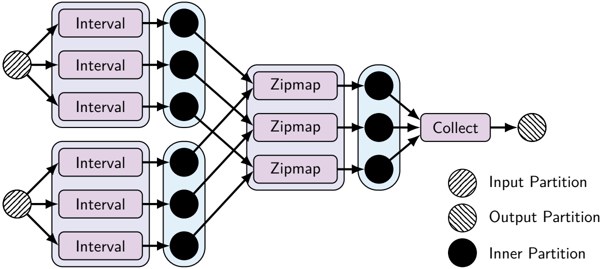


Figure 3: Corresponding dataflow graph

Implementation

- Implemented in C++14
- Optimized for small code size
- One thread per core
- Construct static schedule with list scheduling algorithm (HEFT)

- Shared dataflow graph and schedule
  - Graph structure does not change during dataflow execution
  - Threads can check if actor is ready for execution

Example: Schedule

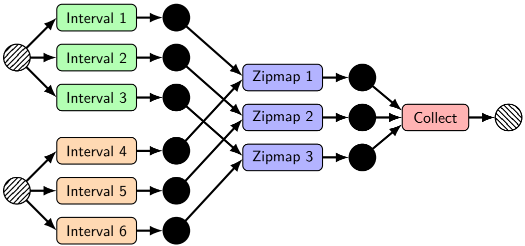


Figure 4: Dataflow graph

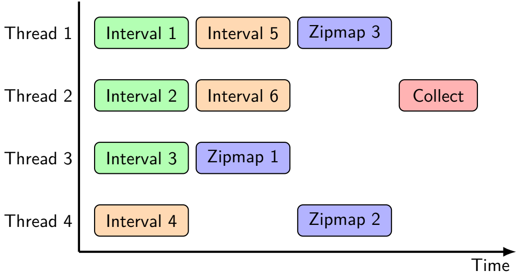


Figure 5: Possible schedule with four threads

Analyzability

- Avoidance of timing anomalies
  - Static distribution of actors on threads
  - Threads execute actors in a fixed order
- Simple timing analysis of single actors
  - Sequential actor execution
  - Output of actor only depends on its input
- Avoidance of dynamic memory allocation
  - Known partition sizes
  - Memory allocation only at graph construction

Overview

Evaluation Setup

- Comparison with OpenMP 4.5
- Small benchmark algorithms
  - Matrix multiplication (Cannon's algorithm / triple-loop algorithm)
  - Fast Fourier transform (iterative Cooley-Tukey algorithm)
  - Bitonic sort (iterative algorithm)
- Hardware: Intel Core i7-7700, 32GB RAM
- Software: Linux Kernel 4.18, GCC 8.2 (with optimization level O3)

Execution Times of Matrix Multiplication and Bitonic Sort

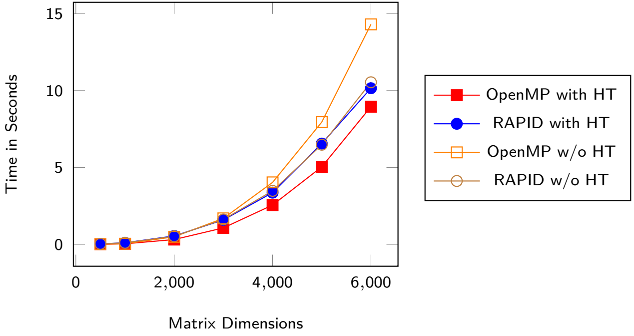


Figure 6: Execution times of matrix multiplication

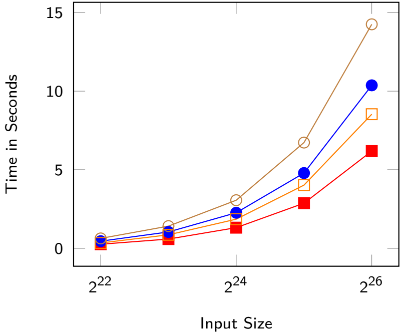


Figure 7: Execution times of bitonic sort

Execution Times of Fast Fourier Transform

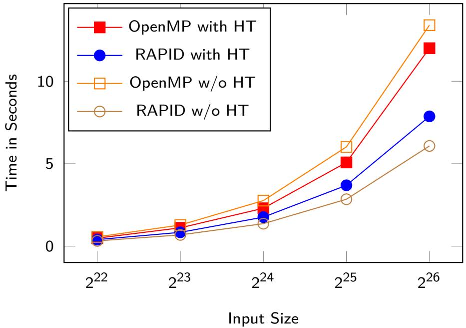
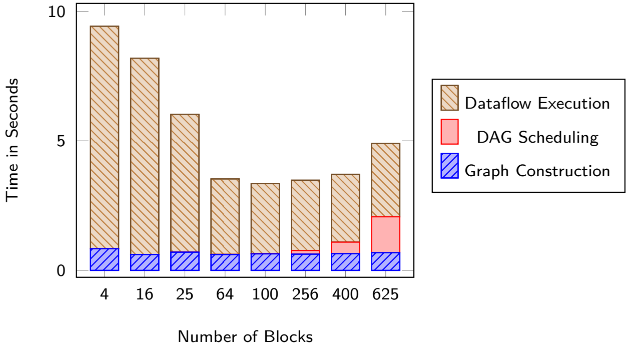


Figure 8: Execution times of fast Fourier transform

Graph Construction, DAG Scheduling and Dataflow Execution



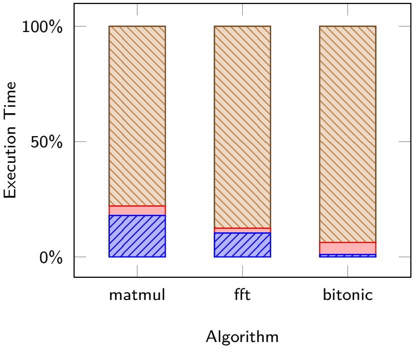


Figure 9: 4000×4000 matrix multiplication with various partitionings

Figure 10: Graph construction and DAG scheduling in various algorithms

Overview

Conclusion and Future Work

- New programming model suitable for safety-critical embedded systems
- Dataflow execution based on directed acyclic graph
- Performance similar to OpenMP

- Investigate timing analyzability
- Utilize dataflow for fault-tolerance through actor duplication
- Expand programming model to support fault tolerance

Questions?

7 Example: Matrix Multiplication

Matrix Multiplication Algorithm

- Cannon's Algorithm
- Inputs: Two square matrices (arrays)
- Algorithm:
  1. Divide input matrices into blocks (Parallelize and Reorder)
  2. Block-wise standard matrix multiplication (Zipmap_Partitions)
  3. Exchange blocks (Reorder_Partitions)
  4. Repeat steps 2 and 3 based on the number of blocks
  5. Restore row-wise element order (Reorder)
  6. Return result matrix (Finalize)

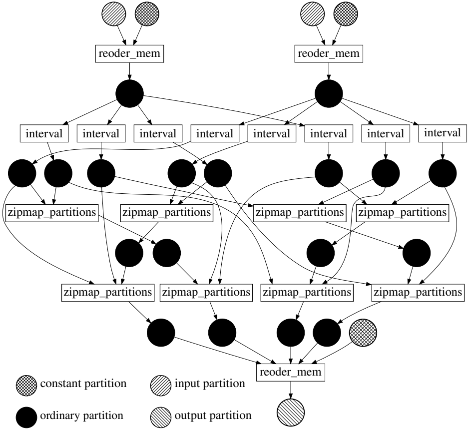


Figure 11: Dataflow graph, four blocks per matrix (Graphviz output)