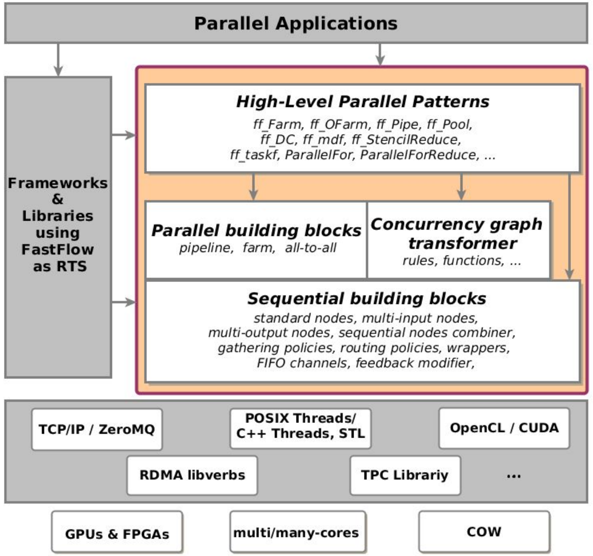
Introduction to FastFlow programming

Massimo Torquati
<massimo.torquati@unipi.it>
SPM lecture 5

Master Degree in Computer Science
Master Degree in Computer Science & Networking
University of Pisa

High-Level Parallel Patterns

- ff_Pipe, ff_Farm, ff_OFarm, ParallelFor*
- **Macro Data-Flow** (MDF)
- **Divide & Conquer** (D&C)
- PoolEvolution, StencilReduce, WindowFarm, PaneFarm, ….etc ...



✓ All High-Level Parallel Patterns implemented on top of Building Blocks

  - mainly: sequential nodes, pipeline, and farm/master-worker

Macro Data-Flow

- The **Data-Flow** programming model is a general approach to parallelism based upon data dependencies among the program's operations expressed in the data-flow graph
- In the data-flow graph, nodes are instructions while edges are true data dependencies (**read-after write** dependencies)
- **Macro** Data-Flow (MDF): the same concepts as Data-Flow but instructions are "fat" instructions, i.e. entire function(s) or block(s) of code

Macro Data-Flow (on shared-memory systems)

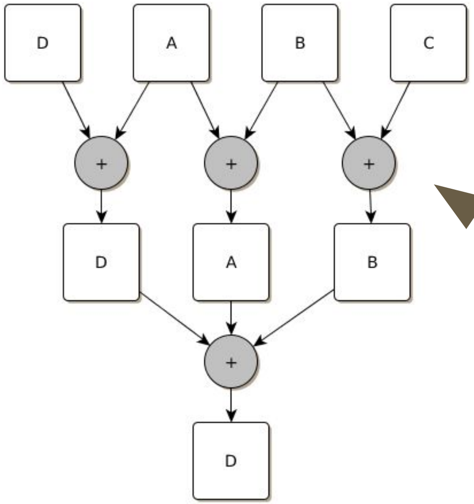
- To reduce memory consumption, in-place computation is generally used (i.e. **A**= F(**A**, B, ….))
- In this way, **anti-dependencies** are possible/more-frequent (i.e. **write-after-read**)
- To solve anti-dependencies either instruction reordering or extra synchronizations or data copies are needed
- Example

D= A+D;

A= A+ B;

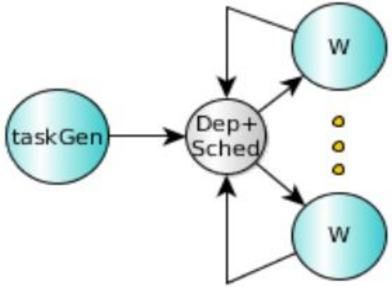
B= B+C;

D= D+A+B



Cannot be executed in parallel

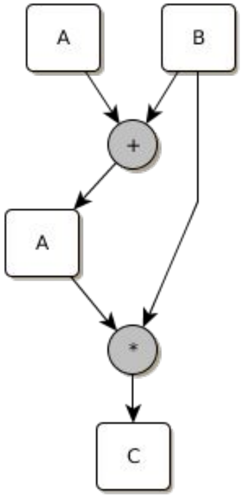
Macro Data-Flow in FastFlow



- In FastFlow, MDF is implemented by the *ff_mdf* pattern.
- The pattern interface is defined in the *mdf.hpp* file
- Currently, it is implemented as a 2-stage pipeline whose second stage is a master-worker.
- The user has to explicitly declare INPUT and OUTPUT data dependencies for each macro task and provides ***pointers*** to input and output data (pointers are used as unique identifier)
- The *AddTask* method creates a macro task
- The run-time takes care of the data dependencies and of the scheduling of the ready tasks

Macro Data-Flow in FastFlow

A = A + B;
C = A * B;




```
// X = X+Y
void SUM(long *X, long *Y, size_t size);
// Z = X*Y
void MUL(long *X, long *Y, long *Z, size_t size);

{ // A = A+B
  const param_info _1={&A, INPUT};
  const param_info _2={&B, INPUT};
  const param_info _3={&A, OUTPUT};
  std::vector<param_info> P={_1,_2,_3};
  mdf->AddTask(P, SUM, &A, &B, size);
}
{ // C = A*B
  const param_info _1={&A, INPUT};
  const param_info _2={&B, INPUT};
  const param_info _3={&C, OUTPUT};
  std::vector<param_info> P={_1,_2,_3};
  mdf->AddTask(P, MUL, &A, &B, &C, size);
}
```

Macro Data-Flow example

- Example (mdf_example.cpp)

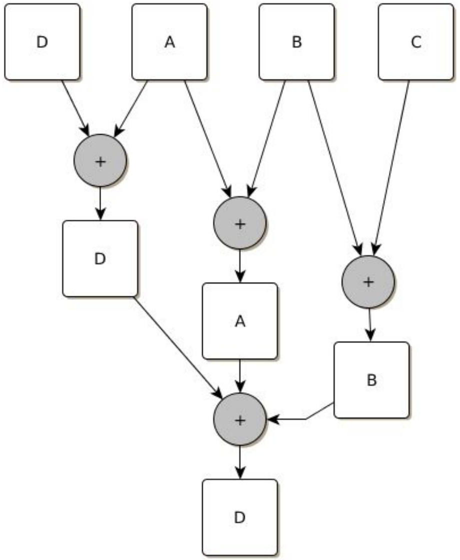
D= A+D; // 1s

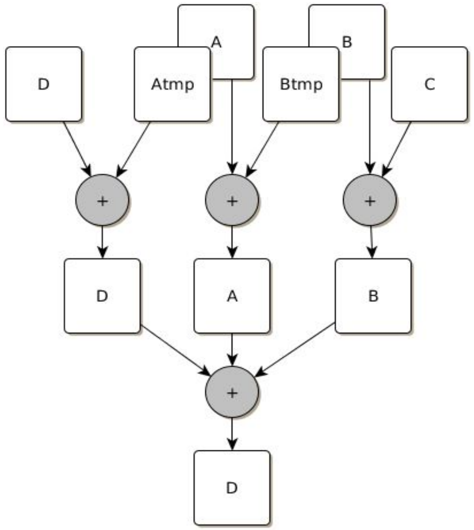
A= A+ B; // 1s

B= B+C; // 1s

D= D+A+B // 1s

Total time 4s





Resolving the anti-dependencies with extra copies

D= Atmp+D; // 1s

A= A+ Btmp; // 1s

B= B+C; // 1s

D= D+A+B // 1s

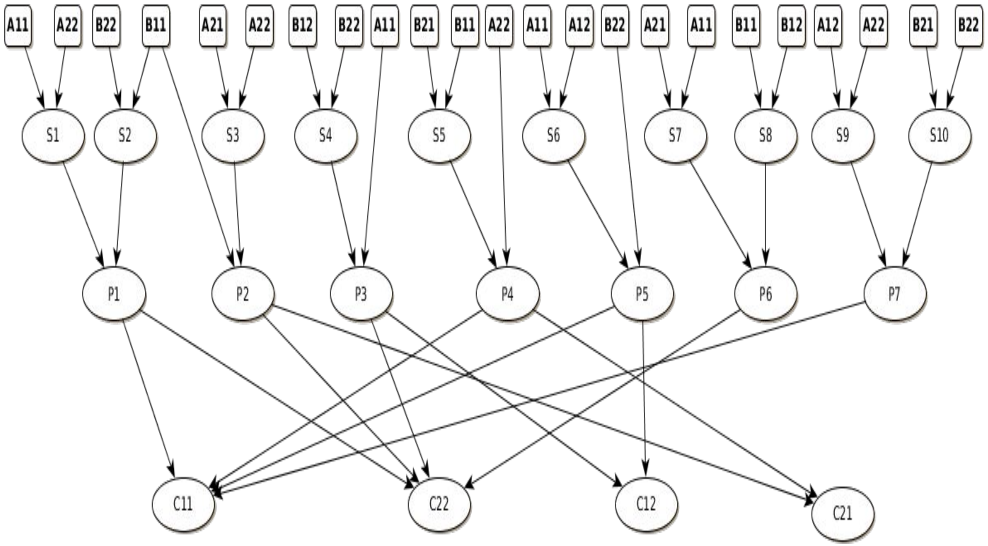
Total time ~2s

(with 3 Workers and assuming ~0s for the copies)

Matrix multiplication, Strassen algorithm

$$\begin{bmatrix} A11 & A12 \\ A21 & A22 \end{bmatrix} X \begin{bmatrix} B11 & B12 \\ B21 & B22 \end{bmatrix} = \begin{bmatrix} C11 & C12 \\ C21 & C22 \end{bmatrix}$$

S1 = A11 + A22 S2 = B11 + B22 P1 = S1 * S2
S3 = A21 + A22 P2 = S3 * B11
S4 = B12 – B22 P3 = A11 * S4
S5 = B21 – B11 P4 = A22 * S5
S6 = A11 + A12 P5 = S6 * B22
S7 = A21 – A11 S8 = B11 + B12 P6 = S7 * S8
S9 = A12 – A22 S10 = B21 + B22 P7 = S9*S10
C11 = P1 + P4 - P5 + P7
C12 = P3 + P5
C21 = P2 + P4
C22 = P1 - P2 + P3 + P6

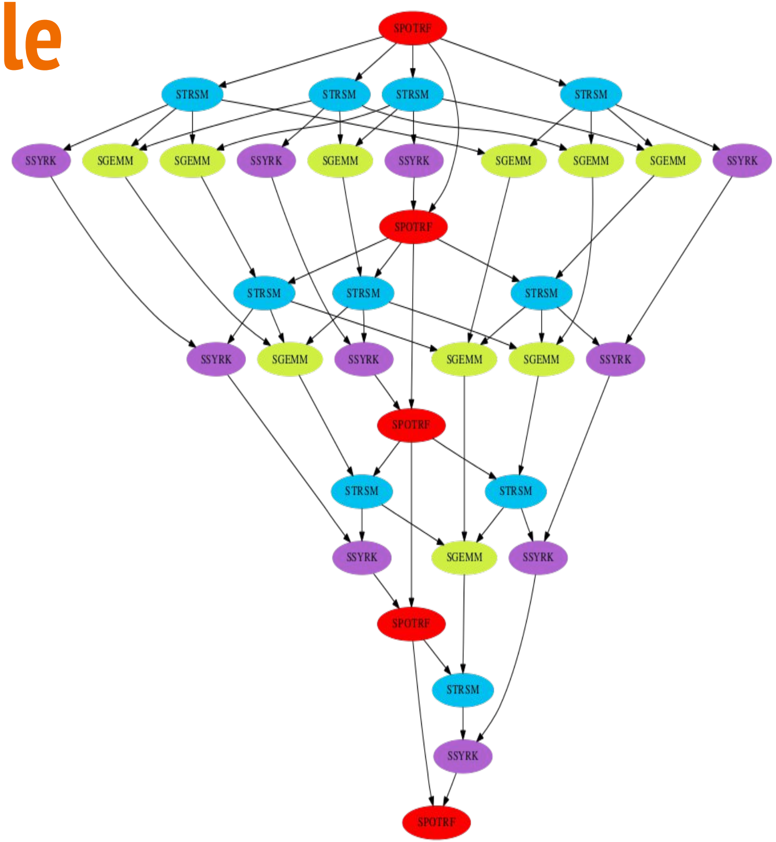


- The function TaskGen generates the instructions S1, S2, P1, S3, …..in any order specifying the INPUT and OUTPUT dependencies (see the *strassen_mdf.cpp* implementation)
- Each macro instructions can be computed in parallel by using a ParallelFor* pattern or can be computed by calling optimized linear algebra routines (e.g., those provided by PLASMA, Lapack, Armadillo, ...)

Cholesky factorization example

- Block-based algorithm
- The DAG is generated automatically during the computation
- Macro instructions are BLAS kernels working on portions (blocks) of the matrix
- Scheduling of macro instructions is very important for performance

Have a look at the code ff_*chol_mdf.cpp*



The DAG represents a 5 tiles left-looking version of the Cholesky algorithm

Divide & Conquer

- Simple API to implement parallel Divide & Conquer computations (DC)
- The programmer that uses the DC pattern has to provide:
    - Two data types as template parameter:
        - ProblemType: input data type
        - ResultType: type of the result
    - The input data and the result data as two objects
    - The following functions:
        - The divide function
        - The combine function
        - The base case function
        - The conditional function

Divide & Conquer interface

```
// functions aliases
using divide_f_t=std::function<void(const ProblemType&,
                                    std::vector<ProblemType>&)>;
using combine_f_t=std::function<void(std::vector<ResultType>&,
                                     ResultType&)>;
using base_f_t=std::function<void(const ProblemType&, ResultType&)>;
using cond_f_t=std::function<bool(const ProblemType&)>;
// D&C pattern constructor
template <typename ProblemType, typename ResultType>
ff_DC(const divide_f_t& divide, const combine_f_t& combine,
      const base_f_t& base, const cond_f_t& cond,
      const ProblemType& p, ResultType& res, int par_degree)
```

Divide & Conquer algorithm

- In FastFlow the D&C pattern is implemented using a master-worker (farm with feedback)
- Each worker executes the DC algorithm in parallel
- Data dependencies are managed by the Emitter
- The Emitter takes care of the scheduling of the macro instructions

```
void DC(const ProblemType &p,ResultType &ret) {
 if(!cond(op)) { //not the base case
   //divide
   std::vector<ProblemType> ps;
   divide(p,ps);
   std::vector<ResultType> res(ps.size());
   //conquer, recursive phase
   for(size_t i=0;i<ps.size();i++)
     DC(ps[i],res[i]);
   combine(res,ret);     //combine results
   return;
 }
 seq(p,ret); //base case
}
```

Divide & Conquer examples

- Fibonacci number example: *fib_dac.cpp* file
- Merge-Sort Algorithm
    - Input parameters: std::vector of integer values, the parallelism degree
    - mergesort_dac_ptr.cpp file

Do parallel patterns really works?

- We have seen that they are OK for single use-case and toy-examples….
- What about real-world applications?
- Is it true that they reduce time-to-solution (and therefore the programming effort)?
- Are performances acceptables? How about performance w.r.t. other programming models?

The PARSEC benchmark suite

- 13 real-world applications from several different application domains
- Initially thought to test new multi-core platforms
- Also used to test programming models
- The NATIVE input datasets, provides realistic input workloads to the applications

Benchmark	Domain	Parallelism	
		Model	*Grain*
blackscholes	Financial Analysis	data parallelism	coarse
bodytrack	Computer Vision	data parallelism	medium
canneal	Engineering	unstructured	fine
dedup	Enterprise Storage	stream	medium
facesim	Animation	data parallelism	coarse
ferret	Similarity Search	stream	medium
fluidanimate	Animation	data parallelism	fine
freqmine	Data Mining	data parallelism	medium
raytrace	Computer Vision	data parallelism	medium
streamcluster	Data Mining	data parallelism	medium
swaptions	Financial	data parallelism	coarse
vips	Media Processing	data parallelism	coarse
x264	Media Processing	stream	coarse

The P³ARSEC benchmarks

Benchmark	Domain	Parallelism	
		Model	*Grain*
blackscholes	Financial Analysis	data parallelism	coarse
bodytrack	Computer Vision	data parallelism	medium
canneal	Engineering	unstructured	fine
dedup	Enterprise Storage	stream	medium
facesim	Animation	data parallelism	coarse
ferret	Similarity Search	stream	medium
fluidanimate	Animation	data parallelism	fine
freqmine	Data Mining	data parallelism	medium
raytrace	Computer Vision	data parallelism	medium
streamcluster	Data Mining	data parallelism	medium
swaptions	Financial	data parallelism	coarse
vips	Media Processing	data parallelism	coarse
~~x264~~	~~Media Processing~~	~~stream~~	~~coarse~~

- Implementation of the PARSEC benchmarks by using parallel patterns
- Code available: https://github.com/paragroup/p3arsec

The P³ARSEC benchmarks

Patterns used:

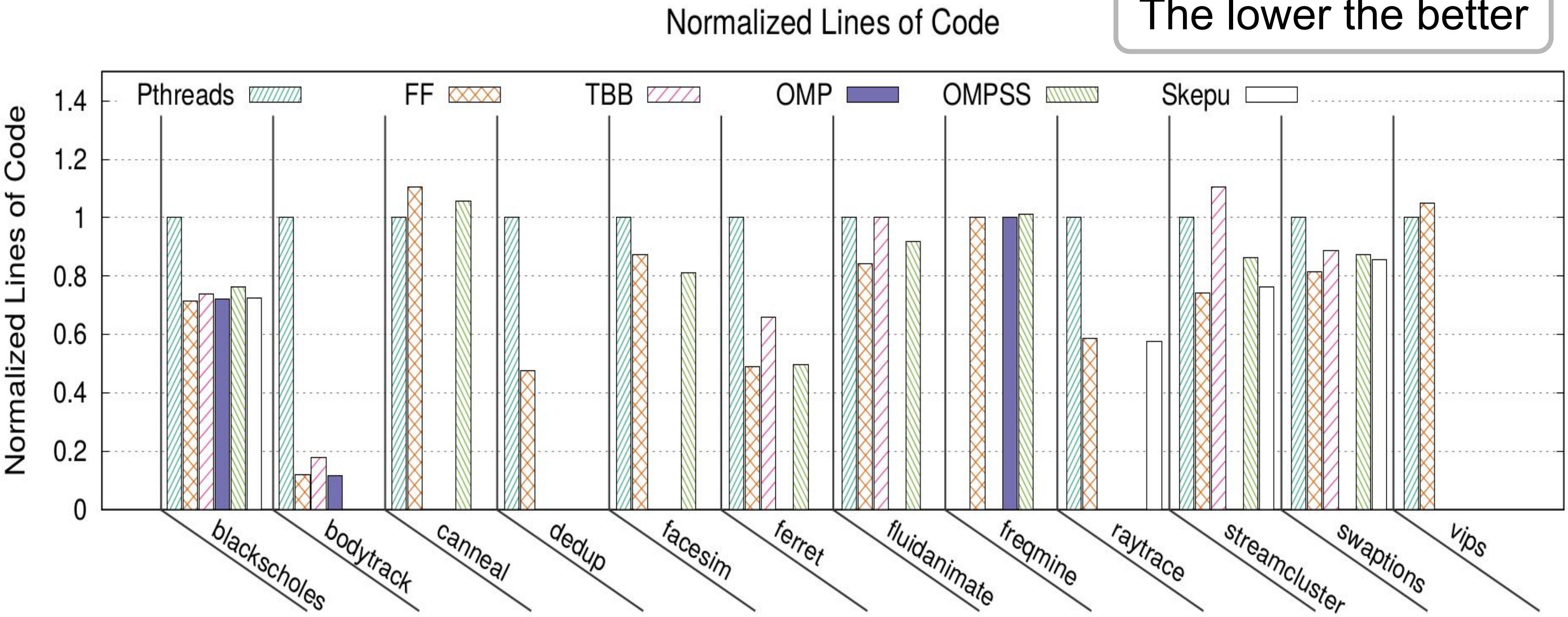
- ✓ iterator
- ✓ map
- ✓ map+reduce
- ✓ pipeline
- ✓ farm
- ✓ ordered farm
- ✓ master-worker
- ✓ composition

blackscholes	iterator(map)
bodytrack	iterator(iterator(map1; map2; map3); iterator(map4; map5)
canneal	master-worker
dedup	4 versions based on pipeline and farm compositions
facesim	iterator(map1; map2; map1; map3; map4; map2; iterator(map5; map6; map7); map1; map4; map2)
ferret	4 versions based on pipeline and farm compositions
fluidanimate	iterator(map1; map2; .... ; map9)
freqmine	map1; map2; ... ; map6; iterator(map7)
raytrace	iterator(map)
streamcluster	iterator(map+reduce; map1; iterator(map2;map3;map4); iterator(map5;map6;map7)); the same as before but repeated only 1 time
swaptions	map
vips	pipe(farm(seq1), seq2)

D. De Sensi, T. De Matteis, M. Torquati, G. Mencagli, M. Danelutto “**Bringing Parallel Patterns Out of the Corner: the P3ARSEC Benchmark Suite**” ACM TACO, 2017, DOI: 10.1145/3132710

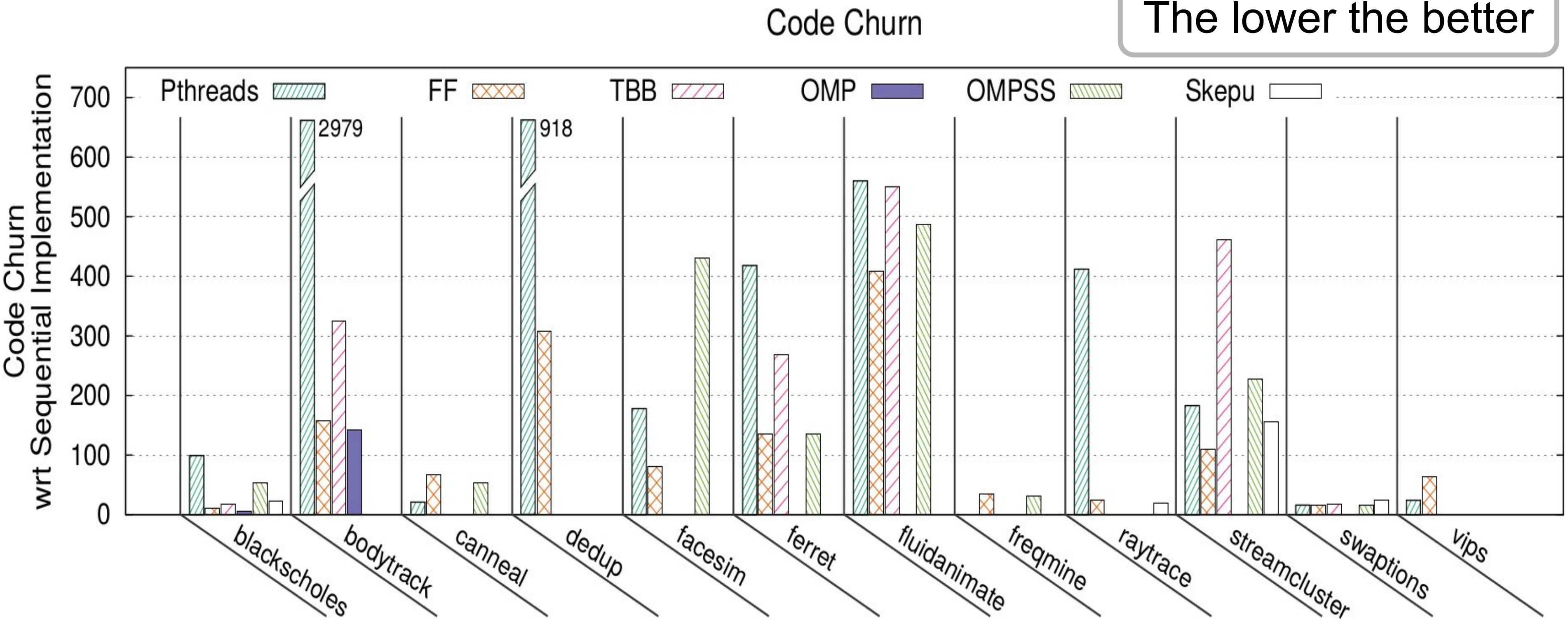
Expressiveness (Lines of Code -- LOC)

The lower the better



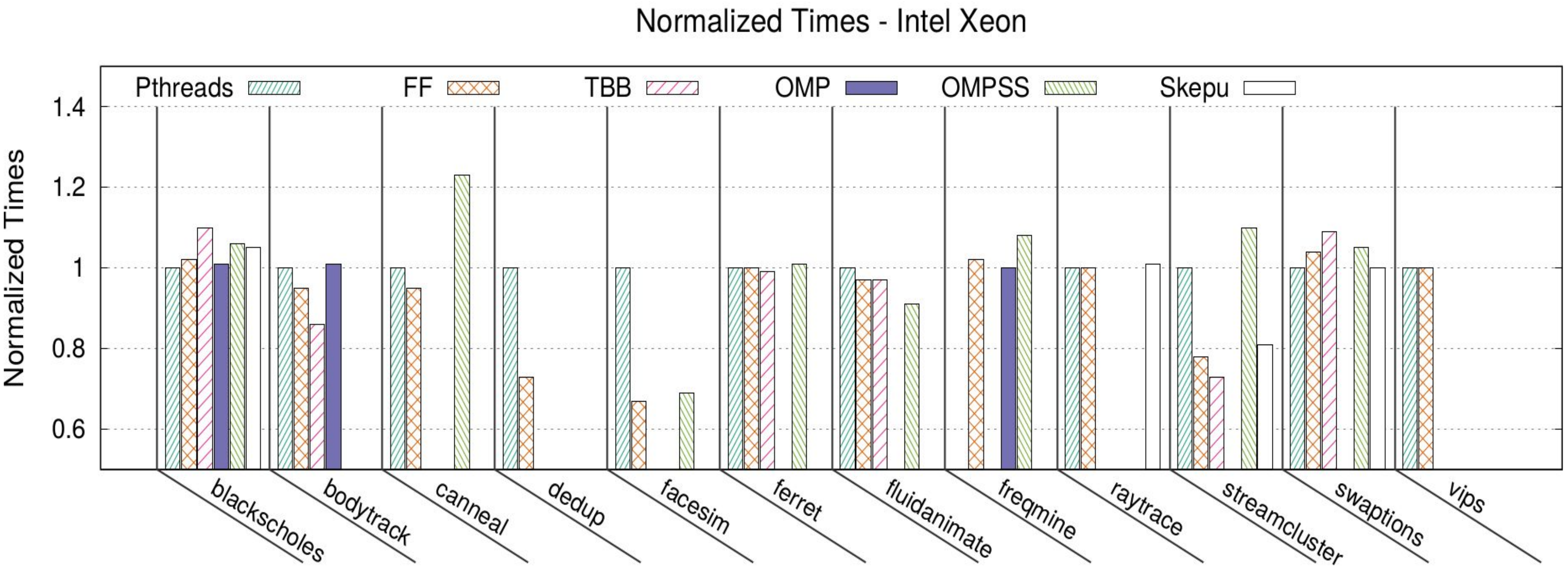
Expressiveness (Code Churn)

The lower the better



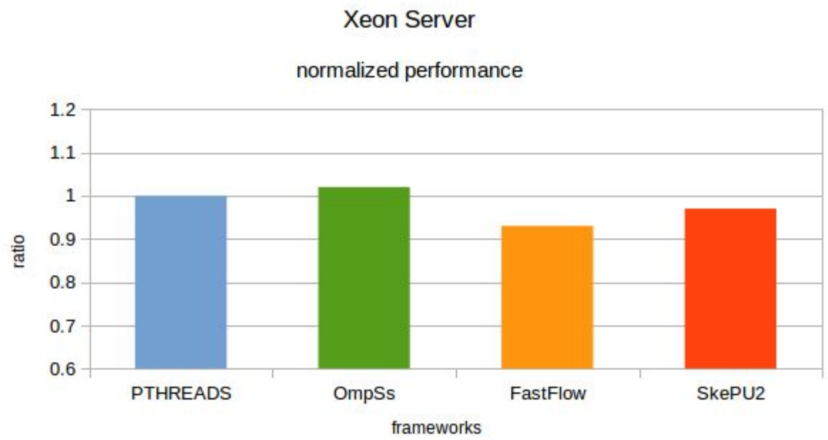
Performance (Intel Xeon server)

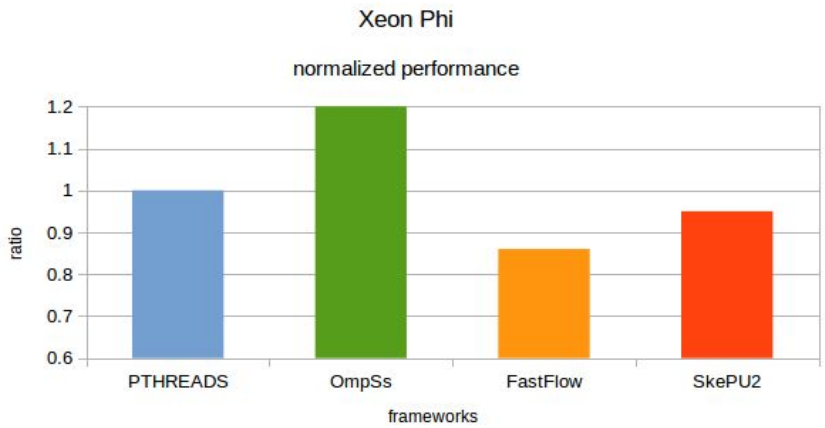
The lower the better

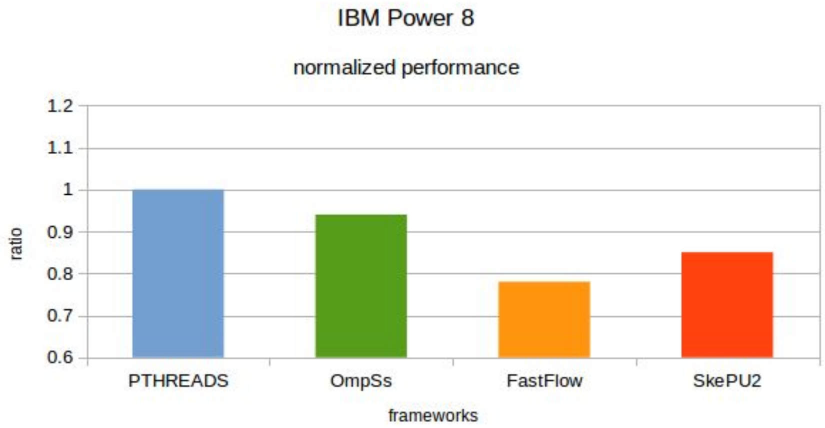


Average performance on different platforms

The lower the better

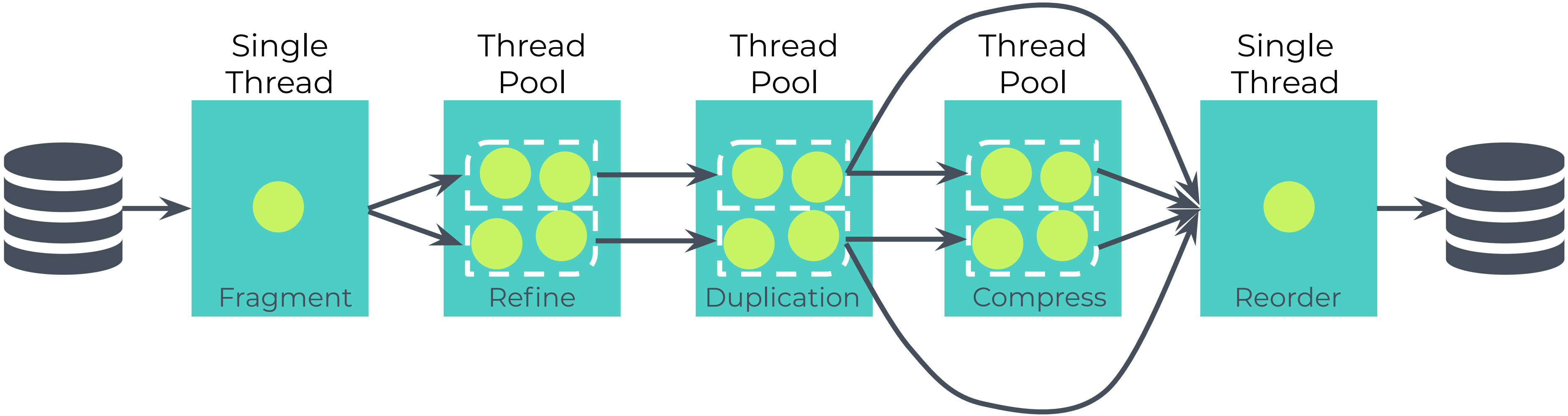




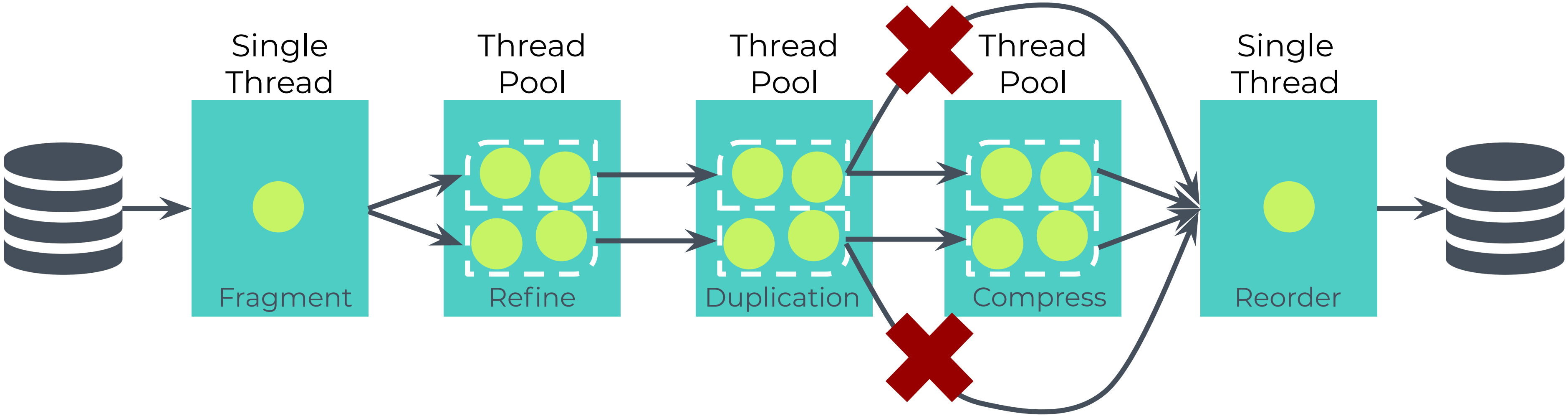


Results available here:
https://github.com/ParaGroup/p3arsec/tree/master/results_TACO

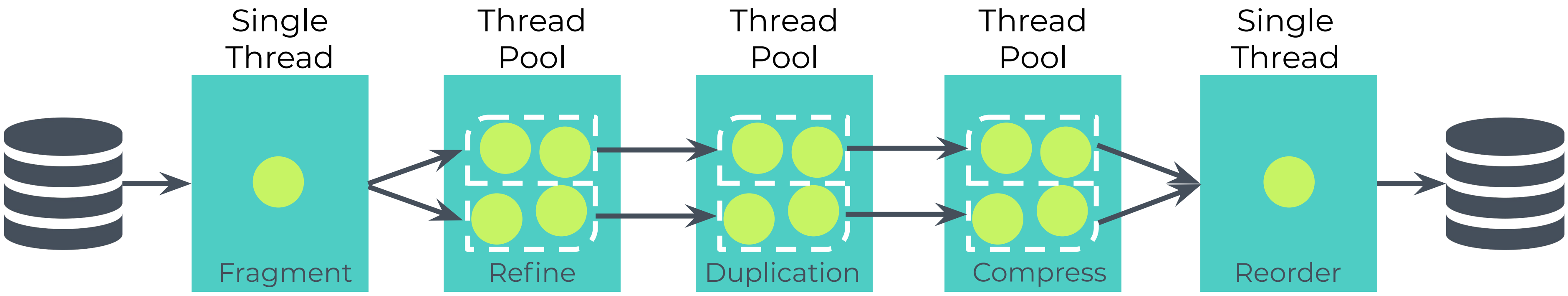
How patterns have been used -- dedup example



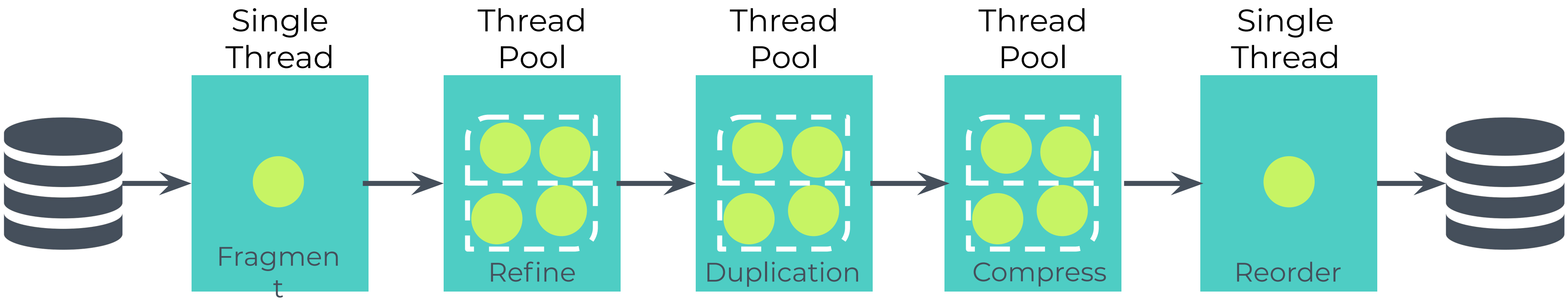
How patterns have been used -- dedup example



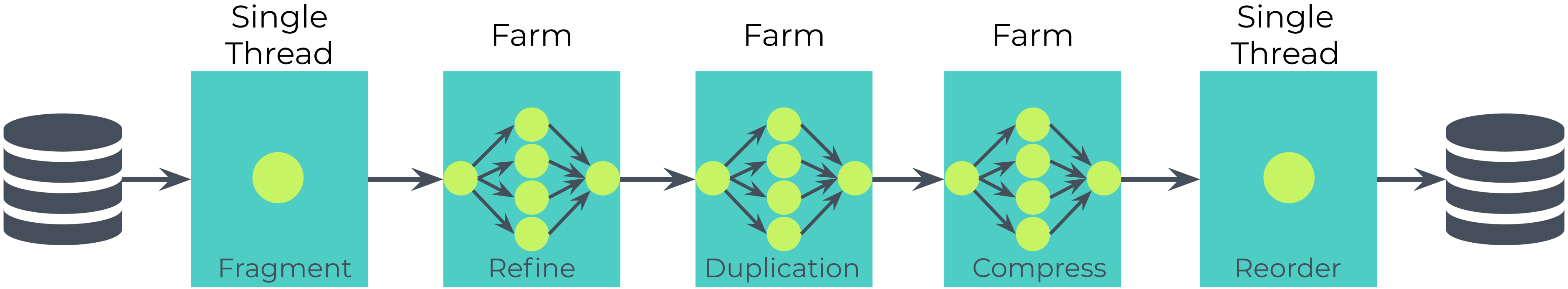
How patterns have been used -- dedup example



How patterns have been used -- dedup example

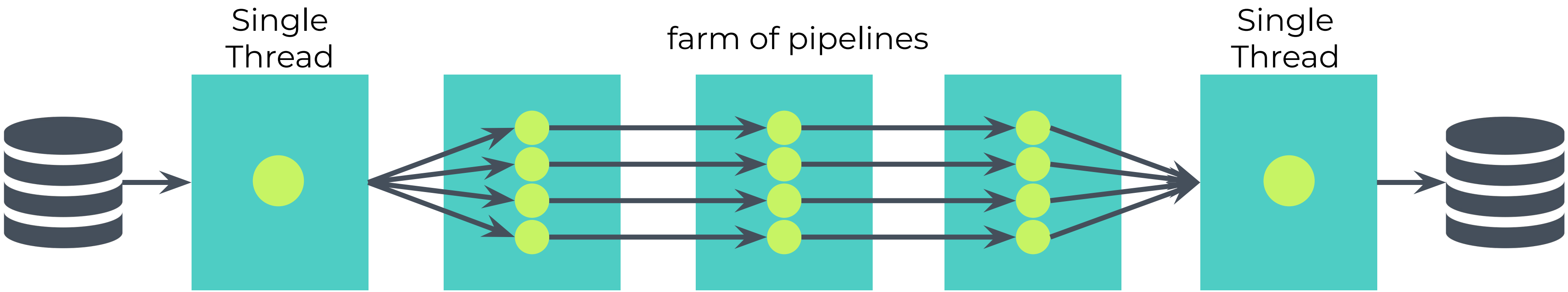


Dedup example -- pipeline of farms



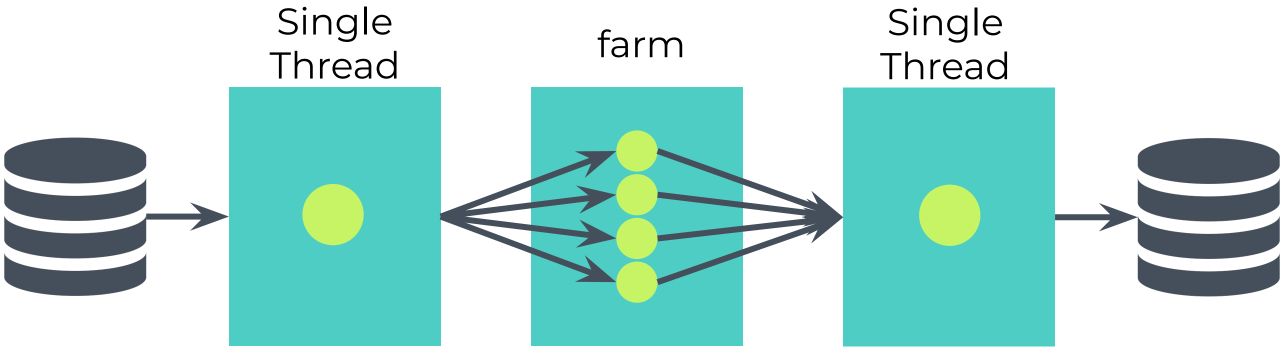

```
pipeline(seq1, farm(seq2), farm(seq3), farm(seq4), seq5)
```

Dedup example -- farm of pipelines



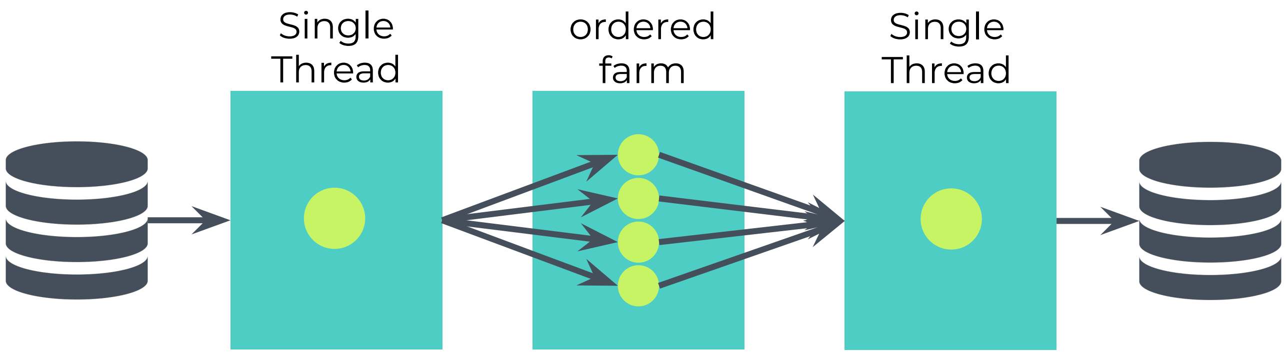

```
pipeline(seq1, farm(pipe(seq2,seq3,seq4)), seq5)
```

Dedup example -- single farm (normal form)




```
pipeline(seq1, farm(seq2;seq3;seq4), seq5)
```

Dedup example -- single ordered farm




```
pipeline(seq1, ofarm(seq2;seq3;seq4), seq5')
```

ordered farm, the seq5' has simplified code

Dedup example -- speedup of the different versions

Arch.	Bench	1.	2.	3.	4.
Intel Xeon Server	dedup	9.23	7.36	8.74	**9.26**
	ferret	25.44	24.48	25.89	**25.89**
Intel Xeon Phi	dedup	6.22	6.54	6.32	**6.6**
	ferret	51.13	52.9	55.69	**92.6**
IBM Power 8	dedup	10.79	12.07	12.61	**13.59**
	ferret	25.53	23.79	25.32	**35.2**

Speedups

1. Pipe of farms; **2.** Farm of pipes; **3.** farm of seqs; **4.** ordered farm/parallel files' reads

Lessons learned from the P^3ARSEC experience

Reduced programming effort (on average) by using patterns in terms of LOC and modifications wrt the sequential code

Simplified design space exploration.
(quickly moving between different parallel alternatives is often paramount for performance)

Comparable performance wrt. state-of-the-art specialized approaches.
(in few cases even better).