SUPSI
Scuola Universitaria Professionale
della Svizzera Italiana

HyperSpark

A Software Engineering Approach to Parallel Metaheuristics

- **Michele Ciavotta**, University of Applied Sciences of Southern Switzerland
- **Srdan Krstic**, ETH Zurich
- **Damian A. Tamburri**, Politecnico di Milano

Agenda

- Motivations
- Goals
- HyperSpark programming model
- Validation case study
- Conclusions and future work

Motivations

Some MOFs (metaheuristic optimization frameworks) are available

- achieve generality
- streamlining creation of new metaheuristics
- organize the knowledge

[Parejo et al, 2012] identified the following drawbacks

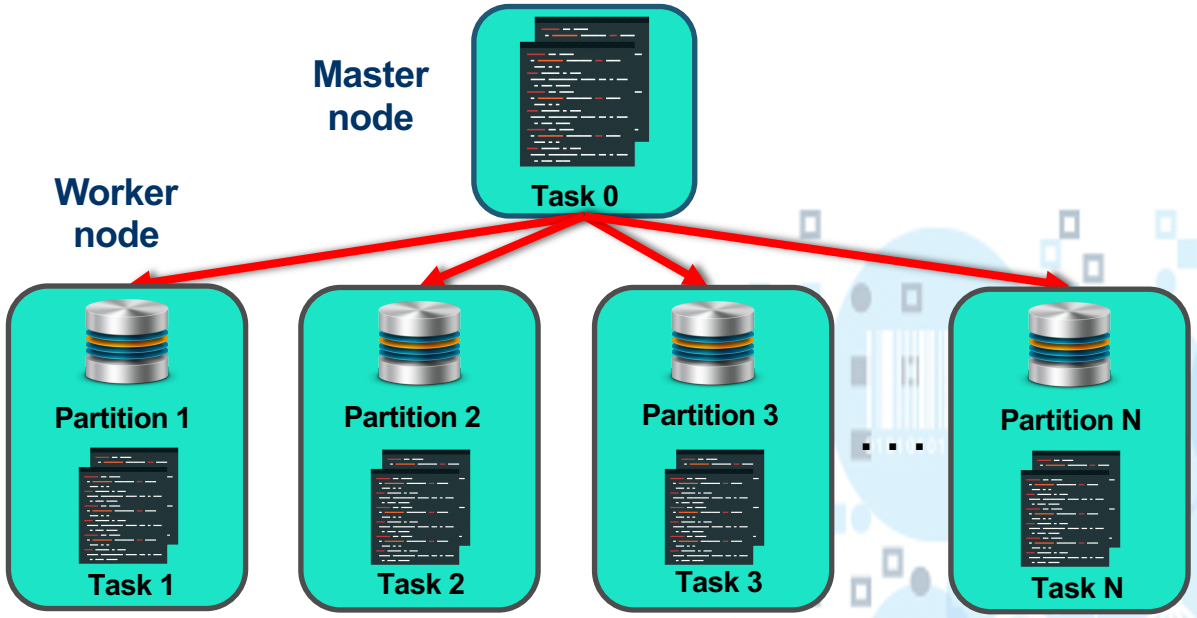
- limited support for parallel and distributed execution,
- lack of support for automated tuning and reactive,
- SE best practices not always followed (extensibility, configurability, portability…)

Goals

- Study the feasibility and challenges of using a distributed computing platform for data-intensive computations to support distributed optimization
- Develop a framework for parallel and distributed execution of meta-heuristic and (more-general) optimization algorithms
- Make it general, extensible, fault-tolerant, portable,
- Motto: *Write locally, distribute painlessly*

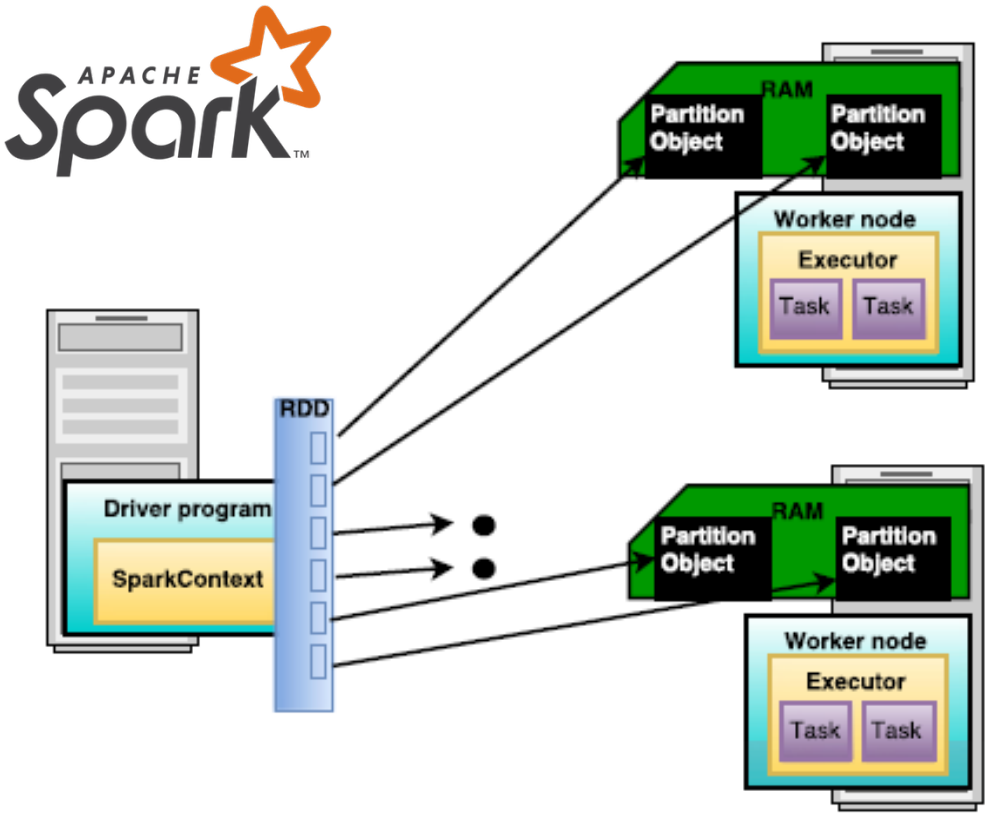
Data parallelism (MapReduce and locality)

- Data are pre-divided
- Data resides on computational nodes
- Functional codes is moved around an executed
- Results are gathered and the process is repeated



Apache Spark: Runtime model


```
val data = Array(1, 2, 3, 4, 5, 6)
val distData = sc.parallelize(data)
```



Spark's secret?

- Efficient use of Memory (100x faster than Hadoop)
- High-level API based on RDD

Why not use Spark to handle code for long running jobs?

Policy: *Write locally, distribute painlessly*

- In HyperSpark's core there is no reference to parallelization. It is a transparent feature.

- The developer is encouraged to write plain ***single-threaded methods*** to tackle the considered problem, as it were to be executed on one single core machine.

- The framework takes care of autonomously and transparently distributing the code, running it in parallel, and collecting results following user specifications.

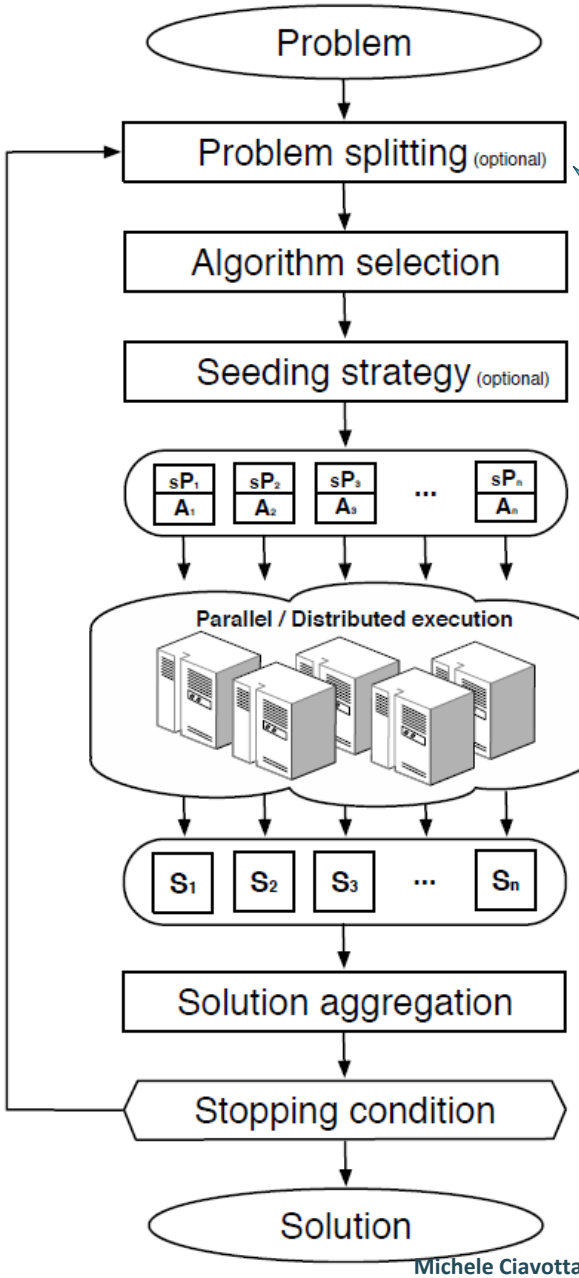
Driving principles

- Easy-to-use – introduce abstractions to make a simple, general. Transformation to the final Spark program is hidden
- Extensibility – abstractions for solution, problem, algorithm and stopping condition concepts make the framework extensible to different domains
- Scalability – enable setting the number of parallel algorithm instances
- Flexibility:
  - User-defined solution-space split strategy
  - User-defined results aggregation operator
  - Virtual topology for cooperation between parallel algorithms

Benefits of using Spark

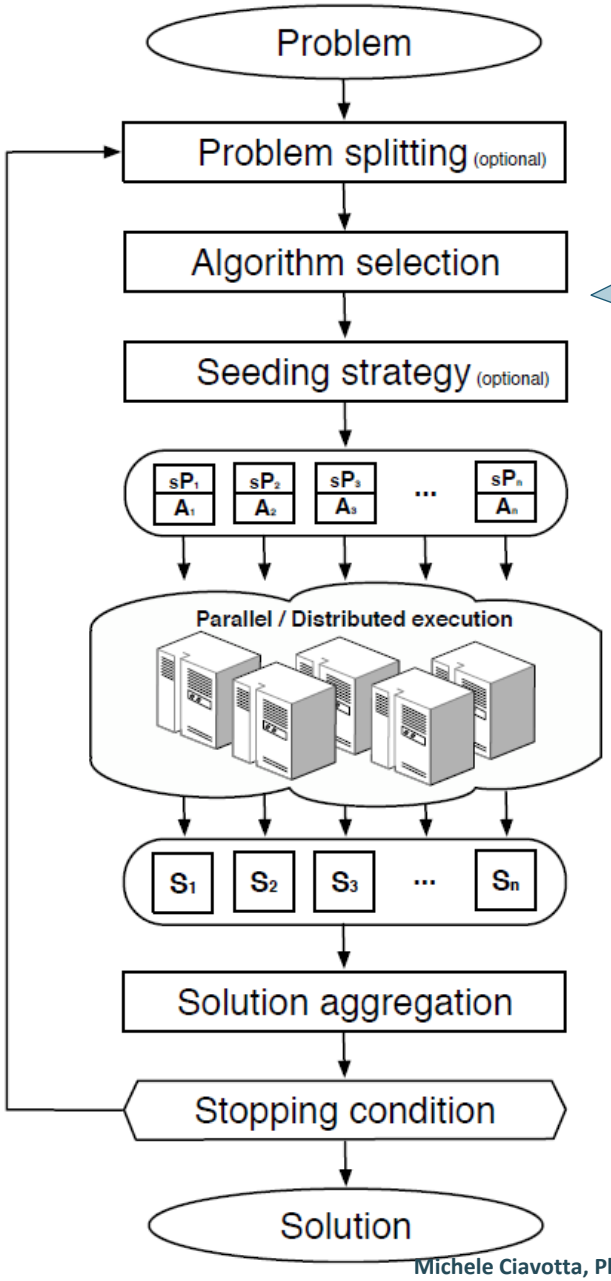
- Parallel and distributed execution – supported by Spark
- Portability – Scala inherits Java portability
- Fault tolerance – supported by Spark
- High performance – in-memory computation of Spark

Programming model



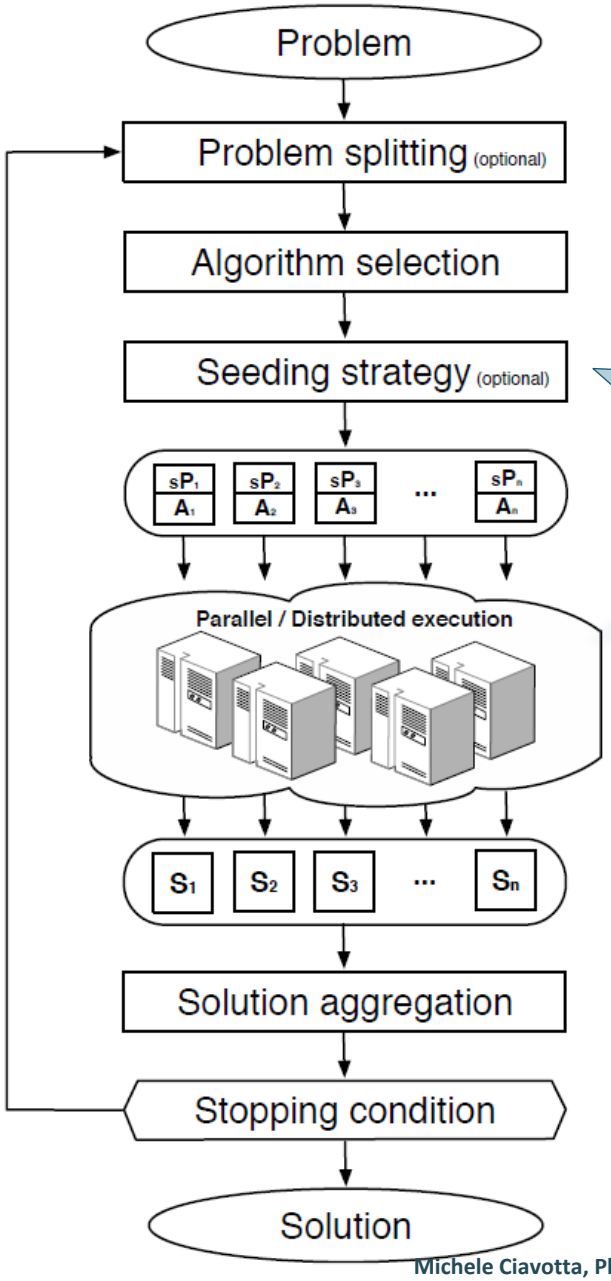
- Split the problem into different sub-problems
- Parallelize the algorithm and assign to each parallel instance a different region of the solution space to explore, or a different objective function

Programming model



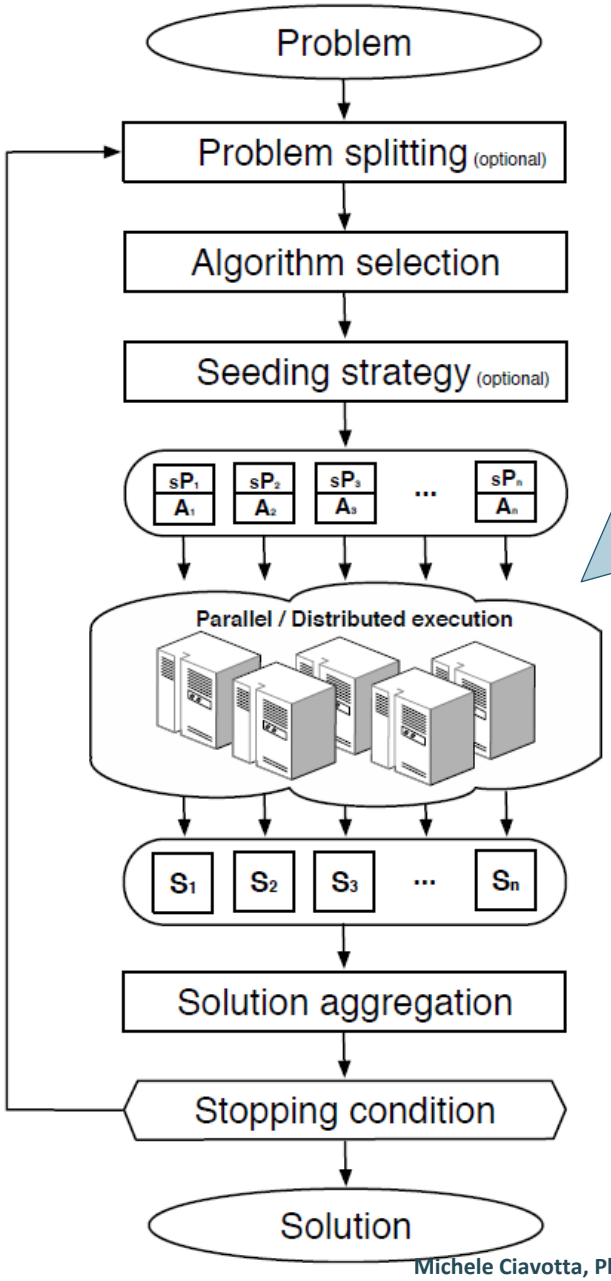
- One or more algorithms can be selected to deal with the split problem
- Algorithms must implement a specific interface

Programming model



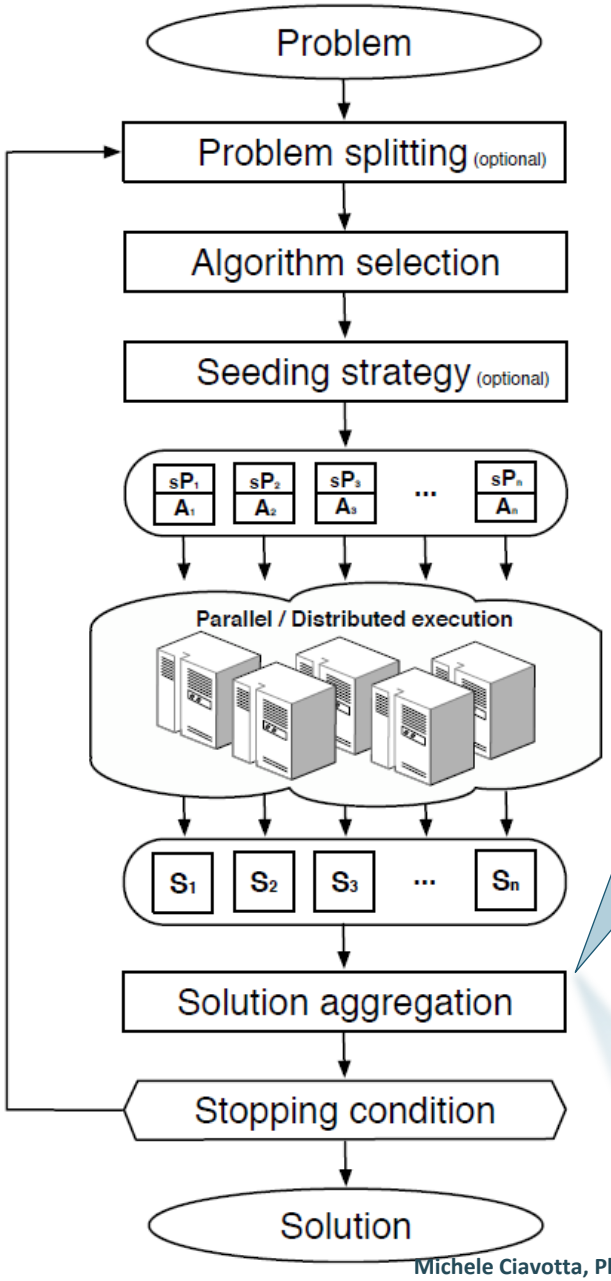
- This strategy defines the way an initial solution is generated at each iteration of HyperSpark (also referred to as stage)
- E.g. this is a way to implement elitism

Programming model



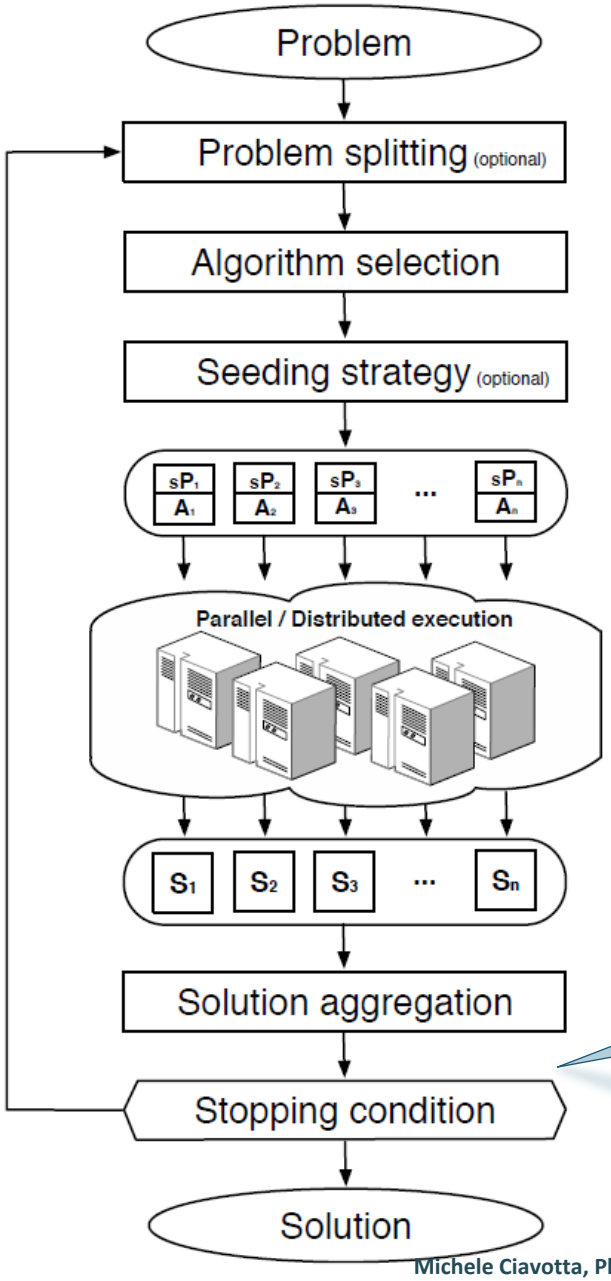
- HyperSpark distributes and runs the algorithms
- To avoid high synchronization times the user is encouraged to implement algorithms that stops after the same amount of time
- HyperSpark has to wait that all the algorithms complete their execution to collect the solutions generated and combine suitably

Programming model



- Aggregation function that combines solutions from different algorithms.
- HyperSpark provides by design an aggregation function that returns the solution with the minimal value obtained from the evaluation of the objective function.

Programming model



- the stopping condition is an arbitrary predicate that determines when HyperSpark stops its execution
- the stopping condition is checked after each stage
- Example, a fixed number of iterations, a timeout, or a complex condition that depends on the solution

Example of use

- ***Convention over configuration design*** paradigm: all of the setter methods, excluding the ones defining the problem and the algorithms, are optional

```
val problem = PFSProblem.fromResources("inst_ta054.txt")
val conf = new FrameworkConf()
                .setProblem(problem)
                .setNAlgorithms(new HGAlgorithm(), 100)
                .setSeedingStrategy(new SlidingWindow(sqrt(problem.numOfJobs).toInt))
                .setStoppingCondition(new TimeExpired(100))
                .setStages(5)
                .setDeployment("spark", 20)
                .setProperty(spark.executor.cores,5)
                .setProperty(spark.executor.memory,8g)
val solution = Framework.run(conf)
println(solution)
```

Validation: Research Questions

1. Is the overhead introduced by HyperSpark acceptable in the context of parallel cooperative optimization?

2. Are the algorithms implemented using HyperSpark competitive with respect to the state-of-the-art?

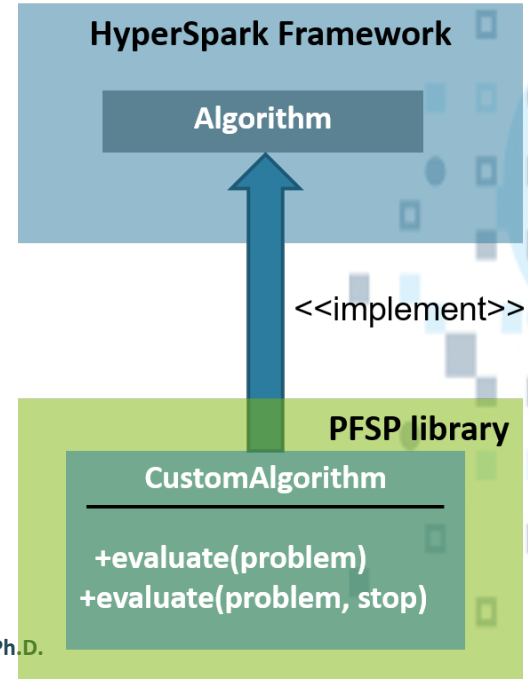
Case study - Permutation Flowshop

- Flowshop is a quite common layout in production and processing lines at industries
- In a Flowshop problem we have a set N = {1,….,n} of jobs to be processed on a set M= {1,….,m} of machines.
- *Permutation flowshop problem*: the same processing sequence for all the machines: n! sequences
- The problem NP-Hard for most objectives (even for *m>2*)
- Objectives: *Cmax*

Extensibility

Implemented Algorithms:

Algorithm	Authors	Year	Ref.	Name
NEH	Nawaz, Enscore and Ham	1983	[15]	*NEH*
Iterated Greedy	Ruiz and Stützle	2007	[23]	*IG*
Genetic Algorithm	Reeves	1995	[22]	*GA*
Hybrid Genetic Algorithm	Zheng and Wang	2003	[35]	*HG*
Simulated Annealing	Osman and Potts's addaption for PFSP	1989	[18]	*SA*
Improved Simulated Annealing	Xu and Oja	1990	[33]	*ISA*
Taboo Search	Taillard	1990	[27]	*TS*
Taboo Search with backjump tracking	Novicki and Smutnicki	1996	[17]	*TSAB*
Ant Colony Optimization	Dorigo and Stützle	2010	[9]	*ACO*
Max Min Ant System	Stützle	1997	[26]	*MMAS*
mMMAS	Rajendran and Ziegler	2004	[21]	*MMMAS*
PACO	Rajendran and Ziegler	2004	[21]	*PACO*



Experiment 1: overhead estimation

Benchmarks:

instance	jobs	machines	execution time (s)
inst_ta001	20	5	3.0
inst_ta031	50	5	7.5
inst_ta061	100	5	15.0
inst_ta091	200	10	60.0
inst_ta111	500	20	300.0

$$executionTime = jobs \times \frac{machines}{2} \times 60 \qquad [ms]$$

One algorithm (IG), 5 runs

Goals:

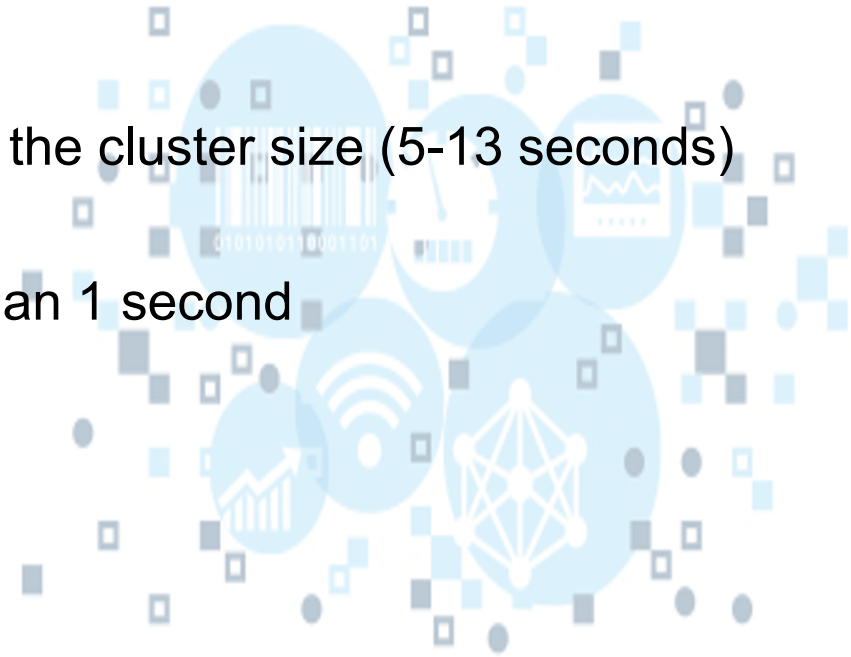
Measure time overhead imposed by Spark with respect to the number of cores used (1-40)

Experiment 1: overhead estimation

	instance	cores	stage_0 (s)	overhead (s)	ovr (%)	init (s)	close (s)
20 jobs, 3 seconds	inst_ta001	1	5.80	2.8	46.69	5.80	0.60
	inst_ta001	8	5.91	2.91	49.00	7.20	0.20
	inst_ta001	16	7.68	4.68	60.79	7.60	0.60
	inst_ta001	24	9.54	6.54	68.50	9.40	0.40
	inst_ta001	32	10.76	7.76	71.94	11.80	0.60
	inst_ta001	40	13.26	10.26	77.33	12.80	0.60
50 jobs, 7.5 seconds	inst_ta031	1	10.13	2.63	25.75	8.20	0.40
	inst_ta031	8	10.56	3.06	28.91	7.60	0.20
	inst_ta031	16	12.38	4.88	39.43	7.60	0.40
	inst_ta031	24	14.29	6.79	47.47	10.80	0.40
	inst_ta031	32	15.84	8.34	52.66	12.00	0.20
	inst_ta031	40	18.26	10.76	58.89	13.60	0.20
100 jobs, 15 seconds	inst_ta061	1	17.37	2.37	13.66	6.40	0.00
	inst_ta061	8	18.22	3.22	17.63	7.60	0.20
	inst_ta061	16	20.14	5.14	25.35	9.00	0.20
	inst_ta061	24	22.21	7.21	32.38	9.20	0.60
	inst_ta061	32	23.47	8.47	36.08	12.60	0.40
	inst_ta061	40	25.89	10.89	42.02	12.80	1.00
200 jobs, 1 minute	inst_ta091	1	63.70	3.70	5.77	6.60	0.00
	inst_ta091	8	66.61	6.61	9.91	4.60	0.40
	inst_ta091	16	69.46	9.46	13.62	4.40	0.20
	inst_ta091	24	70.16	10.16	14.47	7.60	0.20
	inst_ta091	32	73.28	13.28	18.12	7.40	0.40
	inst_ta091	40	76.97	16.97	22.04	8.00	0.80
500 jobs, 5 minutes	inst_ta111	1	318.05	18.05	5.68	5.20	0.20
	inst_ta111	8	320.89	20.89	6.51	4.40	0.20
	inst_ta111	16	326.35	36.35	8.07	4.80	0.40
	inst_ta111	24	332.56	32.56	9.79	6.20	0.40
	inst_ta111	32	337.27	37.27	11.05	7.20	0.60
	inst_ta111	40	346.33	46.33	13.38	11.80	0.80

Experiment 1: overhead estimation

- Overhead for parallelism and synchronization depends on both cluster and instance sizes
- The impact is very high for small instances (up to 80%) acceptable for more time-demanding scenarios (5%-13%)
- The overhead time is much higher for the first stage than for the following ones.
- Initialization time grows only with the cluster size (5-13 seconds)
- Closing time constant and less than 1 second



Experiment 2

Benchmarks:

instance	jobs	machines	execution time (s)
inst_ta{001-010}	20	5	3.0
inst_ta{011-020}	20	10	6.0
inst_ta{021-030}	20	20	12.0
inst_ta{031-040}	50	5	7.5
inst_ta{041-050}	50	10	15.0
inst_ta{051-060}	50	20	30.0
inst_ta{061-070}	100	5	15.0
inst_ta{071-080}	100	10	30.0
inst_ta{081-090}	100	20	60.0
inst_ta{091-100}	200	10	60.0
inst_ta{101-110}	200	20	120.0
inst_ta{110-120}	500	20	300.0

$$executionTime = jobs \times \frac{machines}{2} \times 60 \qquad [ms]$$

Goals:

- Introduce cooperation. Measure time overhead for each iteration (stage)
- Determine the best (algorithm, seeding strategy) combination

Obtained results – Experiment 2a

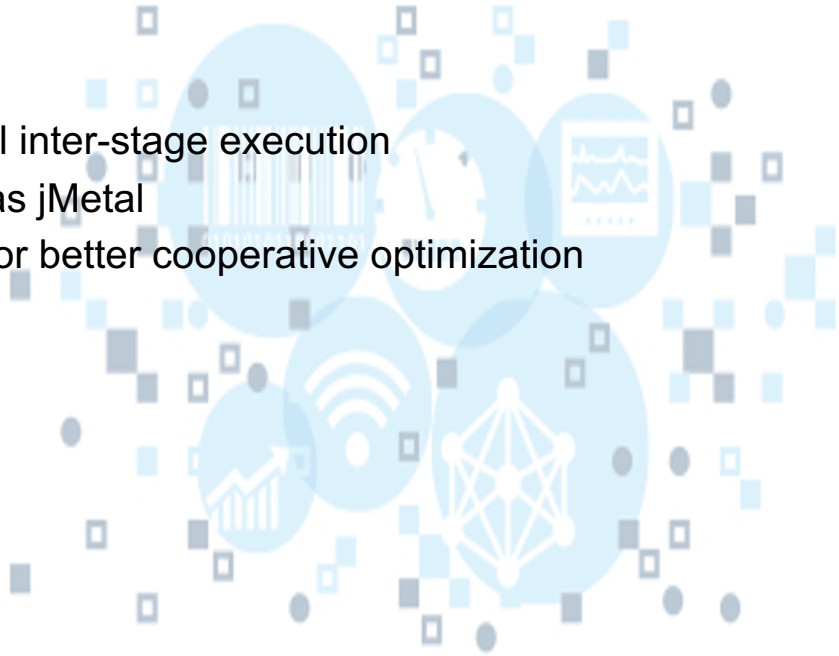
size	exec. time	init time	stage 0 time	stage 0 ovr	stages 1to9 time	stages 1to9 ovr	stages 1to9 avg ovr	close time	compute ovr (%)
50 x 5	7.5	9.64	6.73	5.98	8.94	2.19	0.24	0.48	51.19
50 x 10	15	9.84	7.50	6.00	15.87	2.37	0.26	0.49	35.25
50 x 20	30	6.78	11.41	8.41	29.52	2.52	0.28	0.65	26.55
100 x 5	15	9.06	8.05	6.55	17.04	3.54	0.39	0.48	39.62
100 x 10	30	6.70	11.35	8.35	30.96	3.96	0.44	0.40	28.91
100 x 20	60	6.43	14.63	8.63	59.52	5.52	0.61	0.54	18.91
200 x 10	60	6.24	16.37	10.37	72.56	18.56	2.06	0.62	30.19
200 x 20	120	6.42	22.63	10.63	139.09	31.09	3.45	0.72	23.75
average		7.64						0.55	

Obtained results – Experiment 2b

size	HG_SS	IG_SS	HG_SPSW	IG_SPSW	HG_SPFW	IG_SPFW
50 x 5	0.18	0.10	0.15	0.06	0.15	0.06
50 x 10	2.24	1.74	2.27	1.74	2.28	1.75
50 x 20	3.42	2.87	3.50	2.62	3.52	2.67
100 x 5	0.19	0.10	0.21	0.16	0.21	0.16
100 x 10	1.32	1.11	1.38	1.50	1.38	1.49
100 x 20	3.98	3.58	4.17	3.96	4.17	4.02
200 x 10	0.87	1.05	0.92	1.03	0.90	1.03
200 x 20	3.65	3.76	3.73	3.87	3.76	3.87
average	1.98	1.79	2.04	1.87	2.05	1.88

Conclusions and Future work

- We outlined, evaluated, and discussed a framework for execution of parallel metaheuristics implemented on top of Apache Spark
- We aimed at following sound software engineering principles like, ease-of-use, configurability, flexibility, cooperation, extensibility, and portability
- We realized a promising preliminary experimental evaluation to validate the approach
- We plan:
  - to provide out-of-the-box support for stateful inter-stage execution
  - To integrate with more mature MOFs such as jMetal
  - To facilitate asynchronous communication for better cooperative optimization
  - To support multi-objective optimization



Thank you for your attention!

SUPSI
Scuola Universitaria Professionale
della Svizzera Italiana

Questions?