Run-time resource allocation

Re-allocation of resources in case of heavy load

Davide Burba

Tutor: Prof. Danilo Ardagna

Politecnico di Milano
Mathematical Engineering a.y. 2017/2018

17^{th} January 2018

The Context

Goal

Compute an optimal configuration of resources in a cloud infrastructure in case of heavy load

- Inserted in *Big Data Application Performance models and run-time Optimization Policies* project
- Goal of the overall project: provide Quality of Service guarantees for big data applications execution while minimising resource usage costs

The Problem

Premise:

- Applications have a deadline
- Applications have a weight according to their priority
- Soft deadlines:= applications for which a delay is allowed
- A simulator (dagSim) is used to predict the time of execution of one application

Objective function:

$$\sum_{i \in \mathcal{A}^d} w_i (T_i - D_i) \mathbb{1}_{T_i > D_i}$$

where:

- $\mathcal{A}^d$: set of soft deadline applications
- w_i: weight of application i
- T_i: estimate of execution-time obtained from dagSim simulator
- D_i: deadline of application i

Local Search Policies

There are two policies for the search of an optimal solution (described in next section).

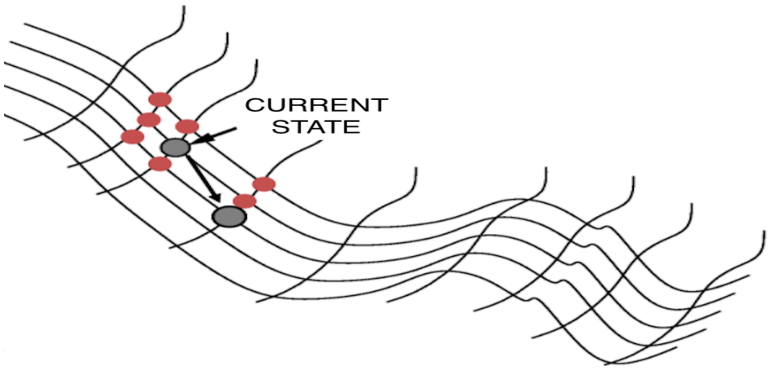
For both:

- the search is local and iterative
- at each iteration a cores exchange between a pair of applications is done

The difference between policies lies in how the profitability of the exchanges is evaluated.

Example

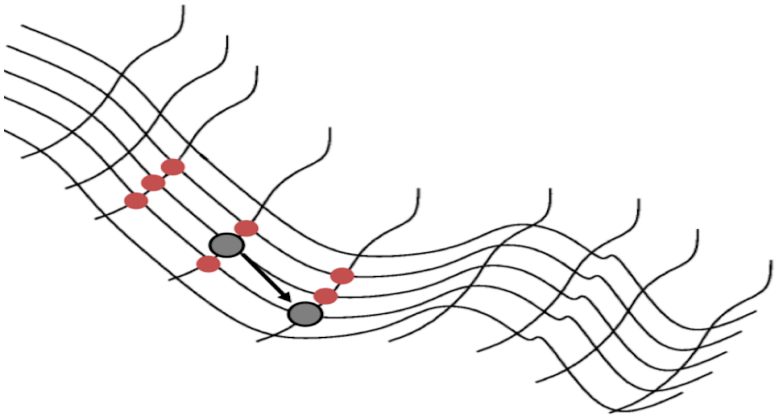
Iteration 1:



- App 1: 16 cores
- App 2: 16 cores
- App 3: 16 cores
- App 4: 16 cores

O.F.= 100.000

Iteration 2:



- App 1: 16 cores
- App 2: 20 cores
- App 3: 12 cores
- App 4: 16 cores

O.F.= 50.000

Premise:

- **Applications data** is stored in a file, say *application.csv*
- A **database** is used to store the results; since dagSim is rather slow (2.5 minutes for a single call), each time it is invoked it looks first if the result has already been cached, otherwise it starts the computations.
- **Nu indices**: indices used to approximate the initial solution through machine learning techniques
- **Bound**: minimum number of cores necessary for an application in order to meet deadline

Workflow:

1. Read and save informations from configuration file
2. Read and save execution parameters from command line
3. Connect to the Database
4. Open *applications.csv* file and store applications data
5. Compute bounds for each application loaded
6. Compute nu indices for each application and initialize the number of cores for each application
7. Fix the initial solution
8. Initialize objective function evaluation for each application
9. Find an optimal solution invoking a *local_ search()* method

Software Choices

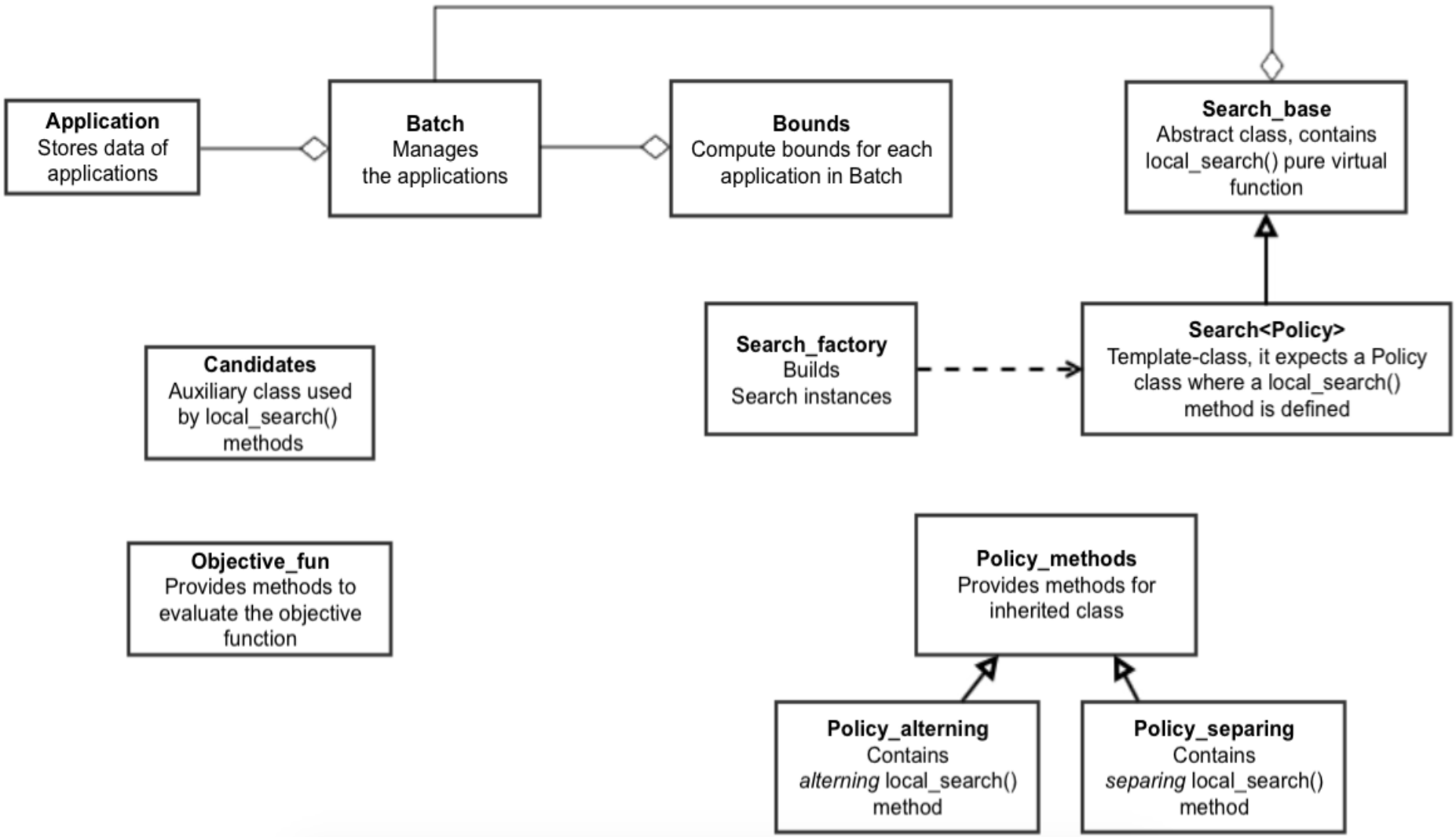
The program has been devoloped in C++.

Time efficiency is the main goal ⇒ all the parts in which dagSim is involved have been parallelized with openMP:

- initialization of objective function
- bounds evaluation
- local search

To provide **easy extendibility** and **dynamic polymorphism**, the local search has been developed with the Policy-based design and an object factory.

Abstract UML Diagram



Configuration Class

A class *Configuration* has been used to store data from configuration file.

- main member: *unordered_map<string,string>*
- operator [] has been defined for the class in order to provide easy access to the map

Batch Class

Applications are managed by **Batch** class which stores a *vector<Application>*

Main member functions:

- **initialize()**: computes a base value of the objective function for each *Application*. If parallelization is activated, each *Application* is assigned to a thread, a database connection is opened for each thread and the objective function is evaluated from each thread.
- **fix_initial_solution()**: fixes the initial solution (computed earlier) by reallocating the residual cores to the applications with highers weights. It uses an auxiliary list of pointers to *Application* ordered by weight.

Bounds Class

Bounds class provides a parallel method to evaluate the bounds for the application.

- Application are stored in a *Batch* member
- Parallelization is done with the same logic as before
- Bounds evaluation is done through a hill climbing: dagSim is invoked until an upper bound is found, then a rollback is performed

The time taken to execute the application can be approximated as

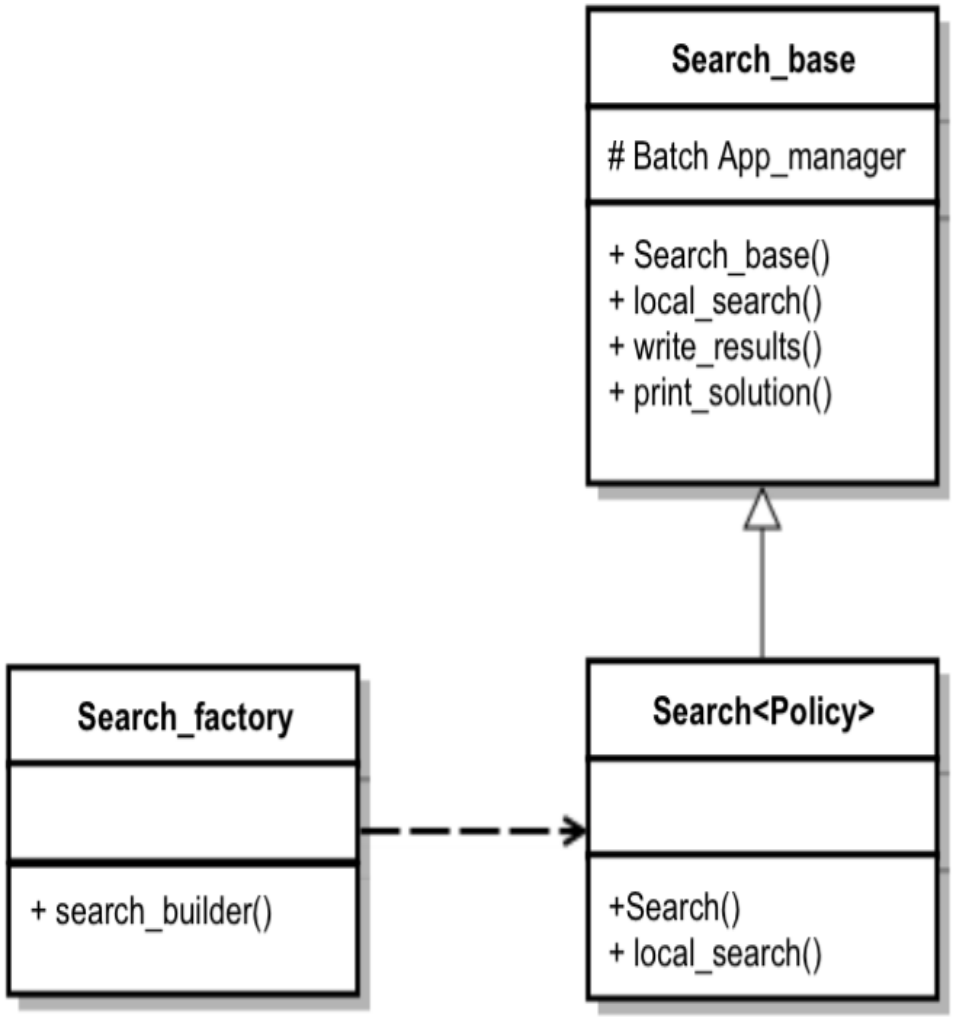
$$t = \frac{\alpha}{n\_cores} + \beta$$

where:

- t is the time taken to execute the application
- n_cores is the assigned number of cores

An estimate of α and β coefficients is done at each iteration.

Search Template Class



- in *Search_base* a pure virtual *local_search()* method is declared.
- in *Search* template class the *local_search()* method in base class is overridden; it uses the *local_search()* defined in Policy class.
- A factory is used to allow dynamic polymorphism; *search_builder()* returns an unique pointer to *Search_base* initialized with *Search<Policy>*

Candidates Class

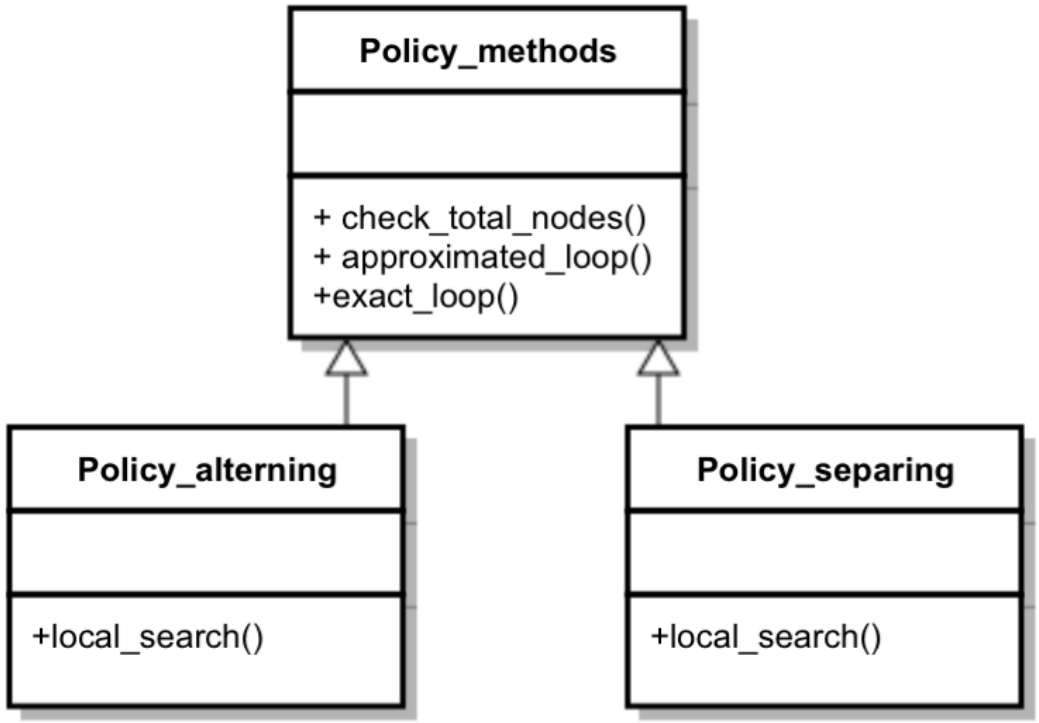
Candidates is an auxiliary class used by *local_search()* methods; data about potential cores exchanges is stored in a *list<Candidate_pair>* ordered by Δ(approximated objective function).

Main methods:

- **invoke_predictor_openMP()**: evaluates the objective function for each pair of applications. Since some pairs could share the same values an auxiliary *vector<Application>* with unique copies of the applications is used. Then the same parallelization logic as before is applied on the auxiliary vector.
- **invoke_predictor_seq()**: the same as above but sequentially.

Policy_methods Class

Policy_methods class provides useful methods to find a solution minimizing the objective function; those methods are used by children classes



Main methods:

- **approximated_loop()**: given a *Batch&*, approximates objective function for each possible cores exchange using estimated α and β coefficients. Profitable moves are stored in a *Candidates* object (which is returned).

- **exact_loop()**: given a Candidates& and a Batch&, it executes an "invoke predictor" method of *Candidates* (*invoke_predictor_openMP()* or *invoke_predictor_seq()*). If there are pairs of Applications for which the exchange is profitable, the best pair is selected and the exchange is done in the Batch object.

Policy_alterning Class

Policy_alterning is a policy-class.

The *local_search(Batch&,...)* method performs a single loop. At each iteration:

- *approximated_loop()* is invoked and returns a *Candidates* object (say cand) containing potential profitable moves;
- *exact_loop(cand, ...)* is invoked and evaluates the objective function for selected pairs using dagSim and does the best move.

This is done until no more profitable exchanges are found or the maximum number of iterations is reached.

Policy_separing Class

Policy_separing is a policy-class.
The *local_search(Batch&,...)* method performs two separates loop.
First loop, at each iteration:

- *approximated_loop()* is invoked and returns a *Candidates* containing potential profitable moves; then the most profitable move (the first) is done in the Batch object.

Second loop, at each iteration:

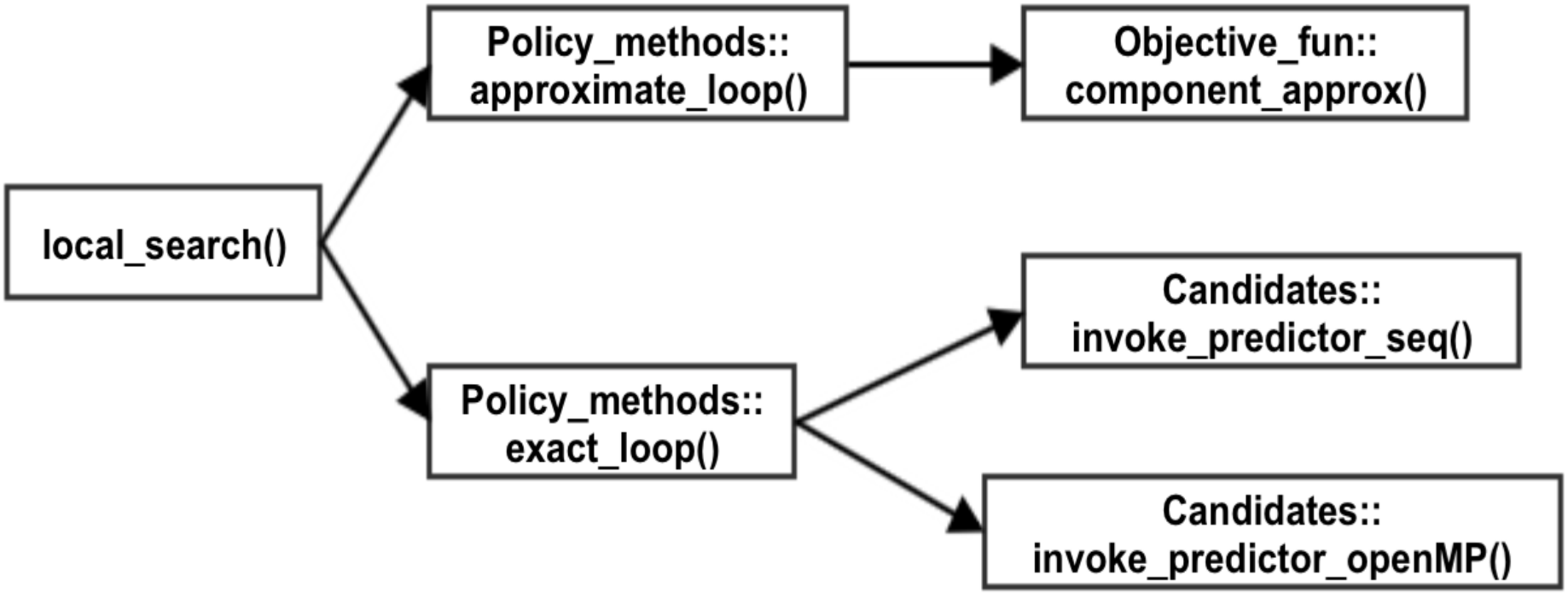
- first a *Candidates* with **all** possible exchanges is built (say all_cand), then *exact_loop(all cand, ...)* is invoked and evaluates the objective function for all possible pairs (and does the best move).

The loop ends when there are no more profitable moves or the maximum number of iterations is reached.

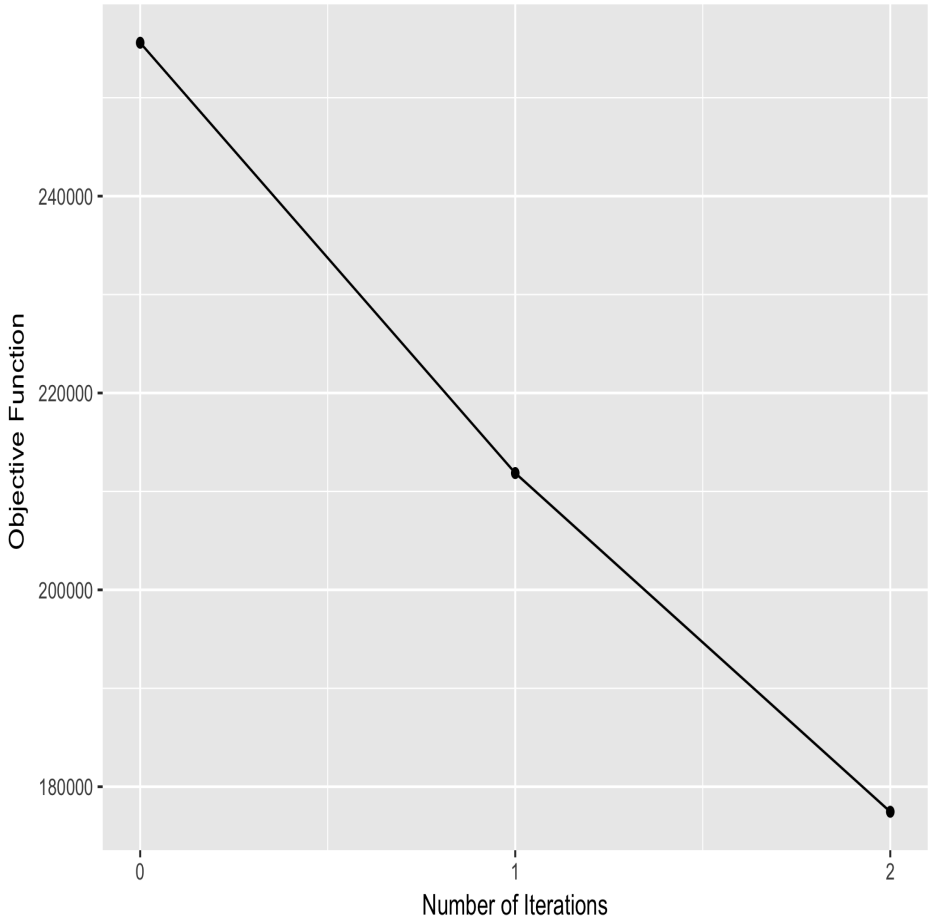
Note: here the auxiliary vector used in *Candidates::invoke predictor openMP()* is fundamental since almost surely there are repetitions of applications in the *Candidates* object.

Call Graph

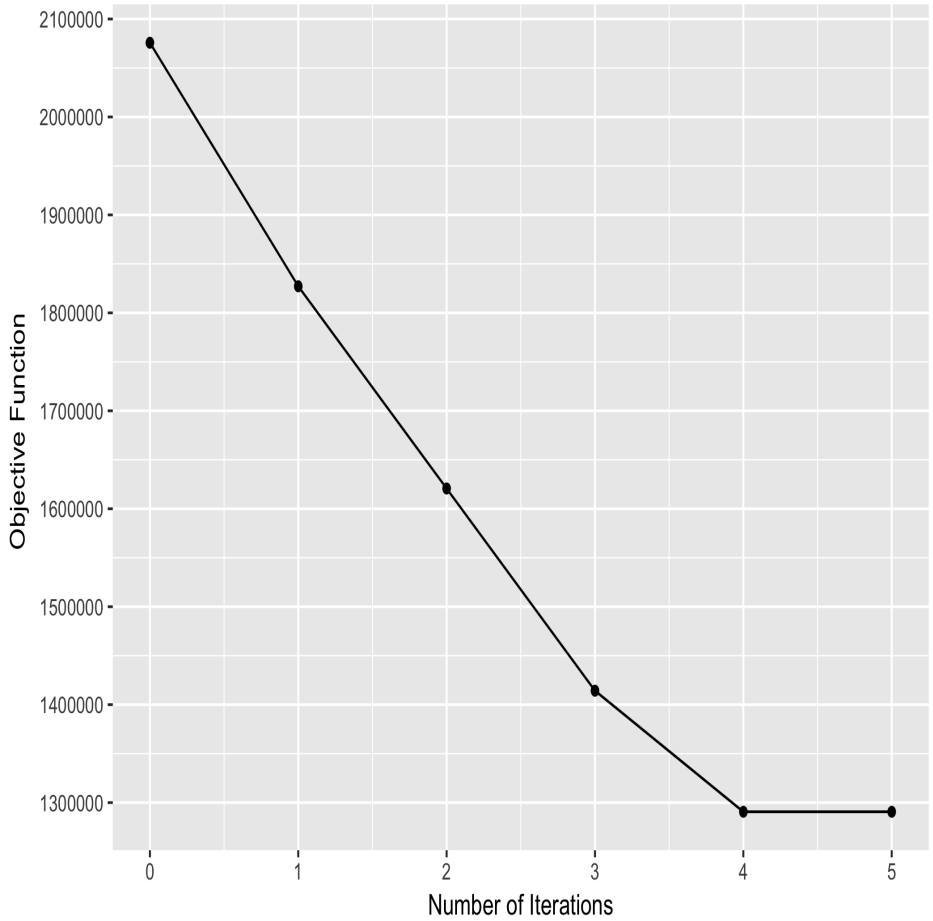
A (simplified) call graph valid for both *local_search()* methods is reported.



Test Objective Function - Alterning

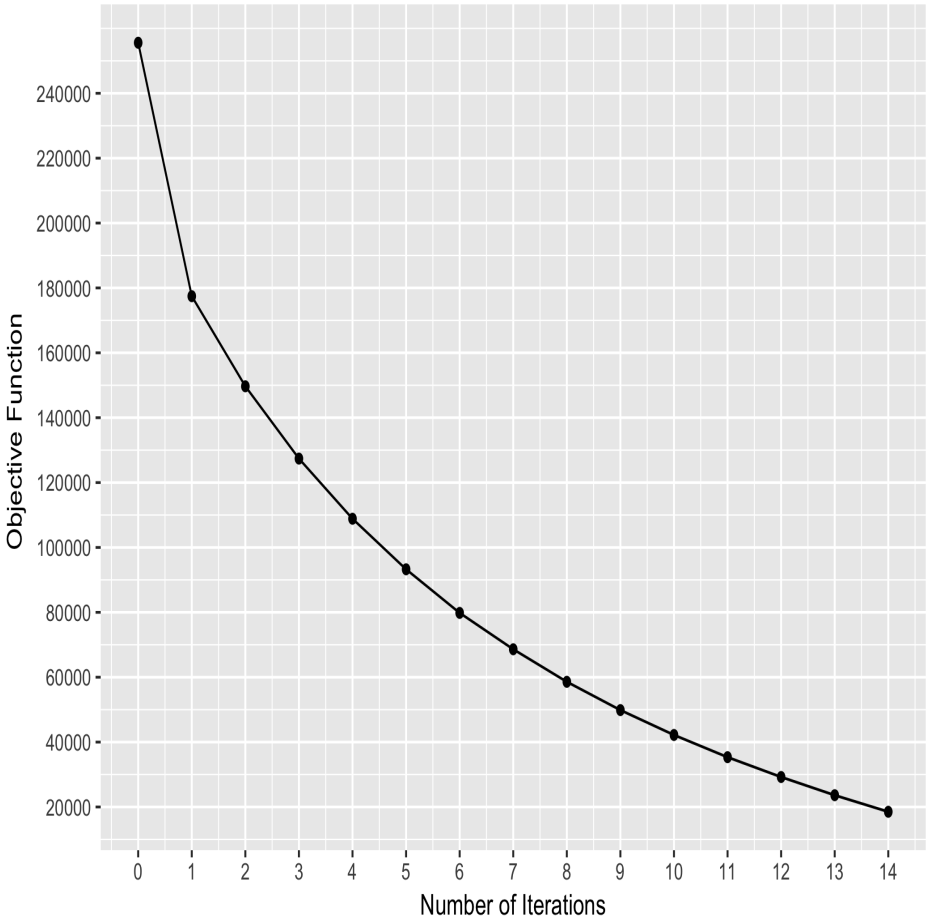


4 Applications

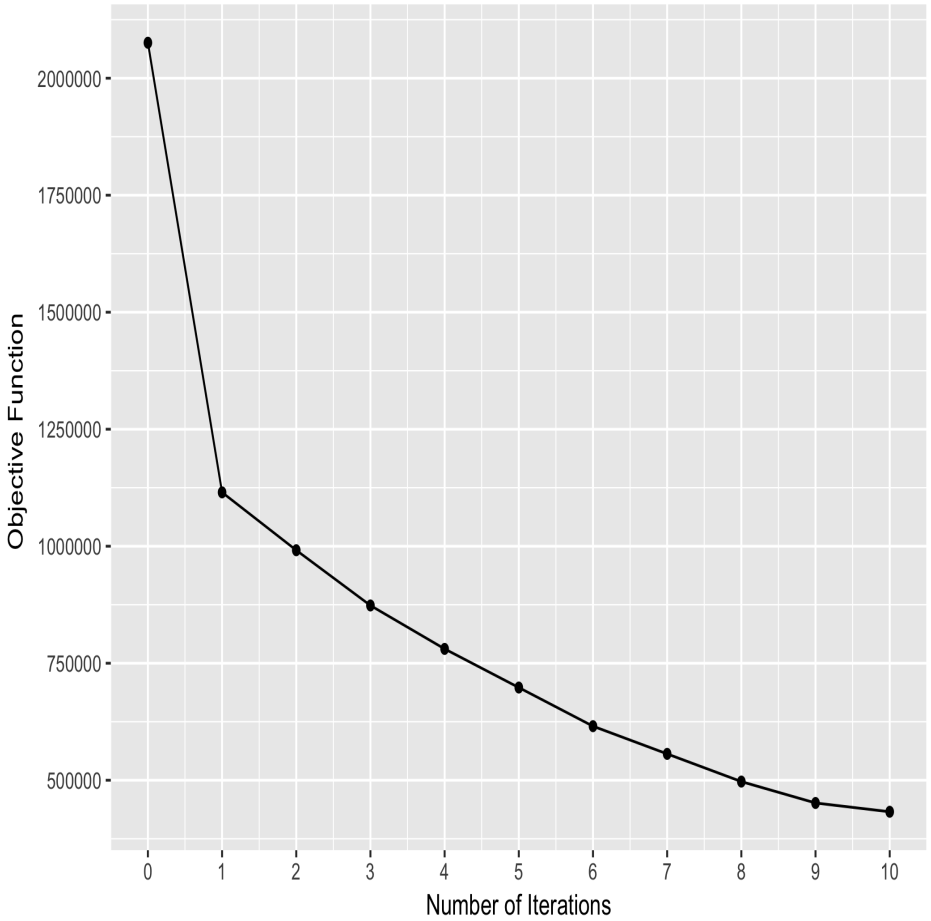


8 Applications

Test Objective Function - Separing



4 Applications



8 Applications

$\Rightarrow$ **Separing:** better results on objective function

Time Performances

All time performances have been measured on a 8-cores machine (DEIB).
Tests have been done with:

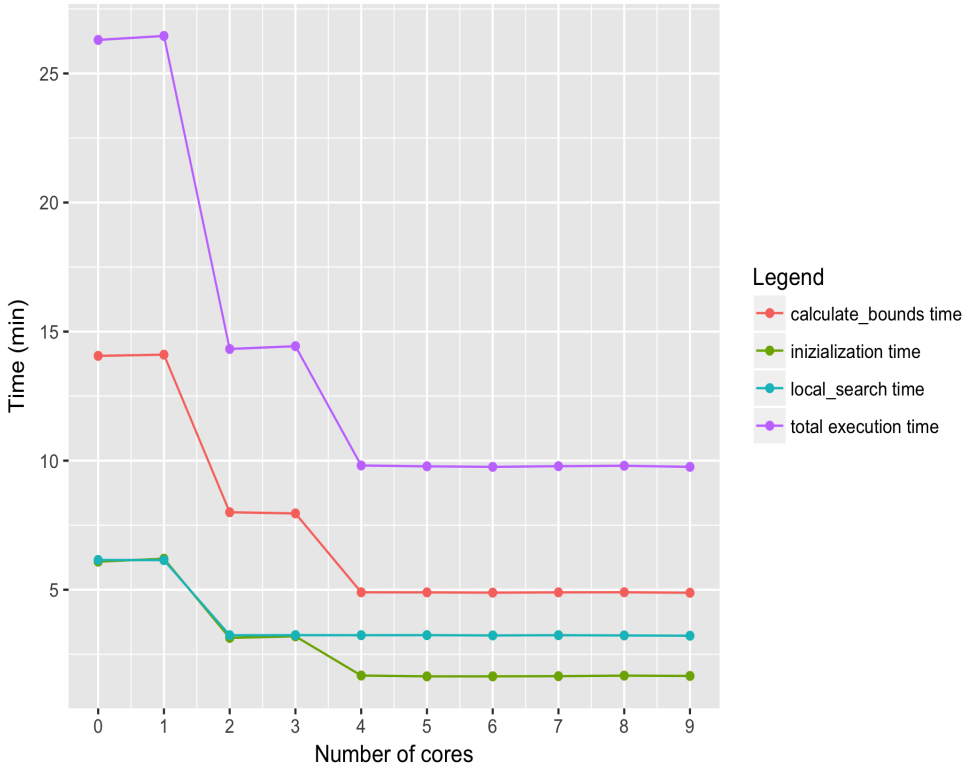
- 10 maximum iterations
- 4/8 applications
- no limits to the maximum number of considered candidates
- changing the cache status (on/off, varying percentage of already computed values)
- changing the number of threads
- changing the policy

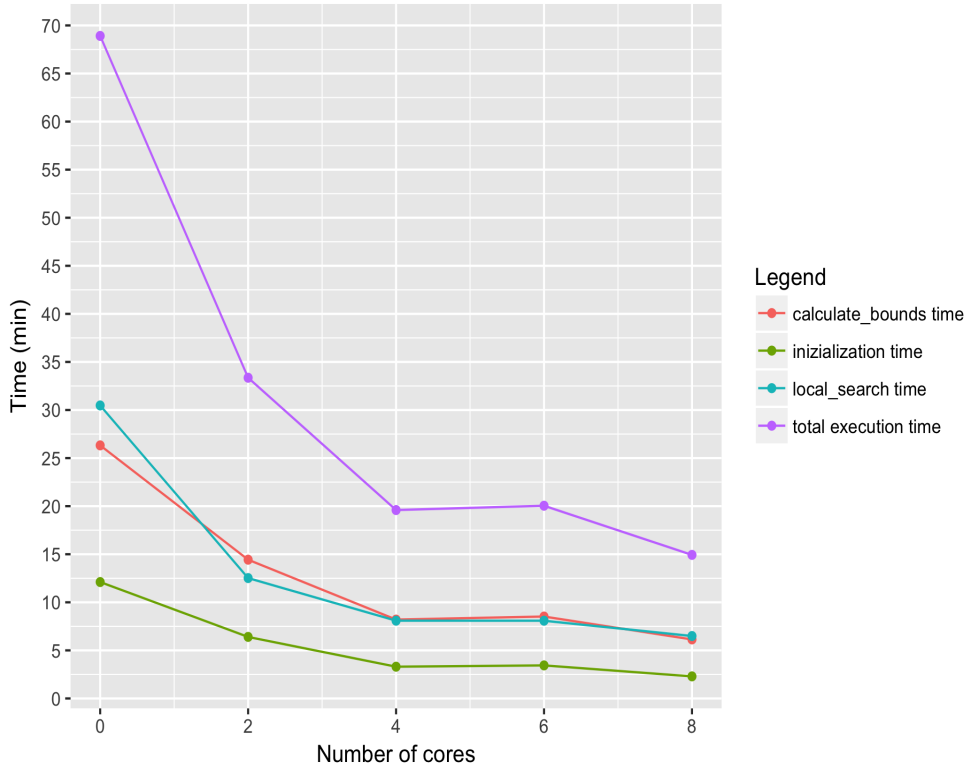
Cache 100%: total execution time < 0.1 seconds
Indeed if all the results are already stored dagSim is never invoked.

$\Rightarrow$ In this case parallelization is useless

Time Performances - Alterning

Cache off

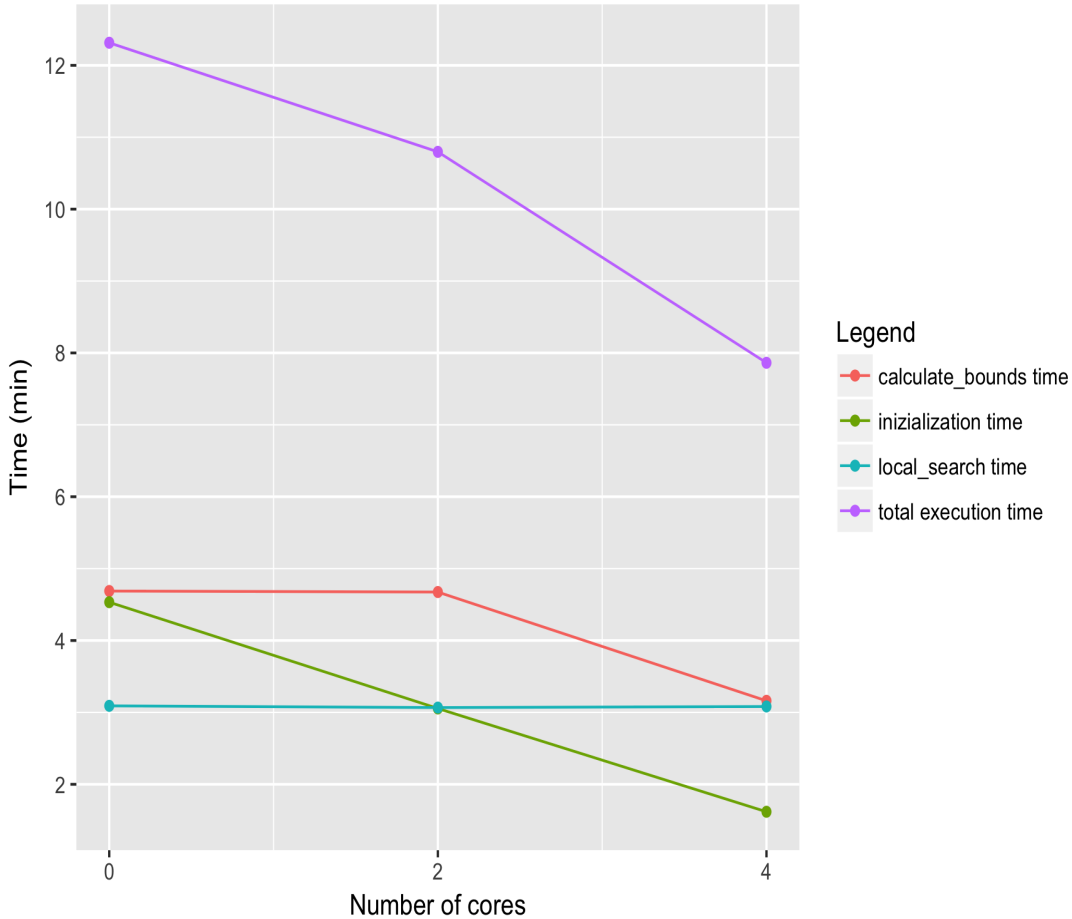




$\Rightarrow$ total execution time reduced by:

- 4 applications: 63%
- 8 applications: 78%

A realistic scenario: **50% cache**

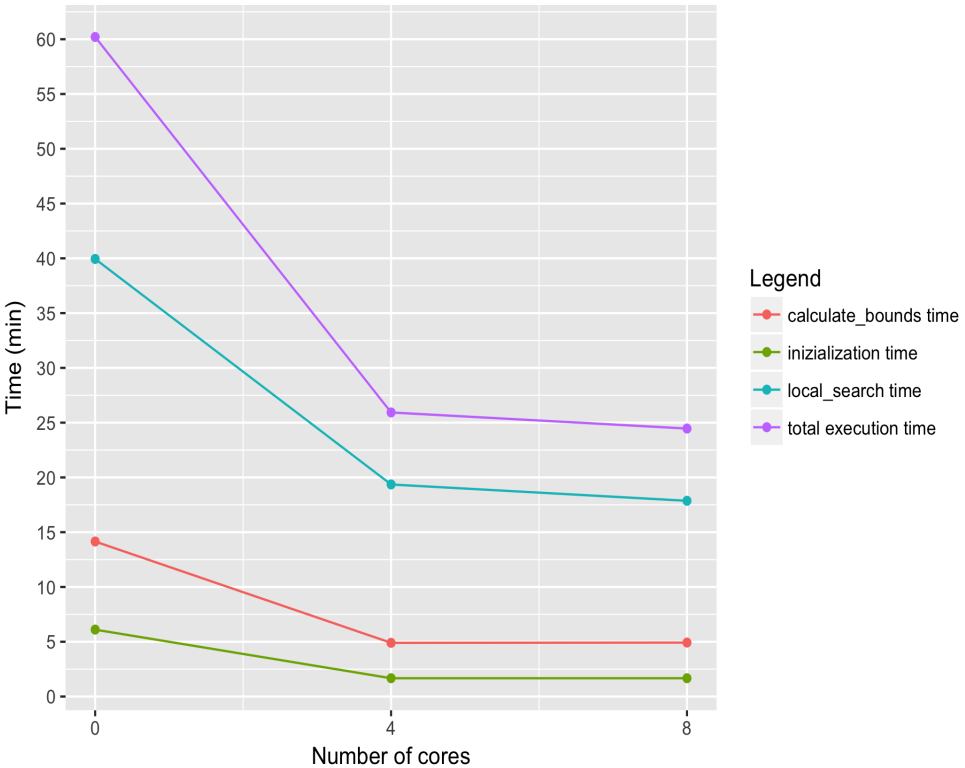


$\Rightarrow$ total execution time reduced by 33% (4 applications)

Time Performances - Separing

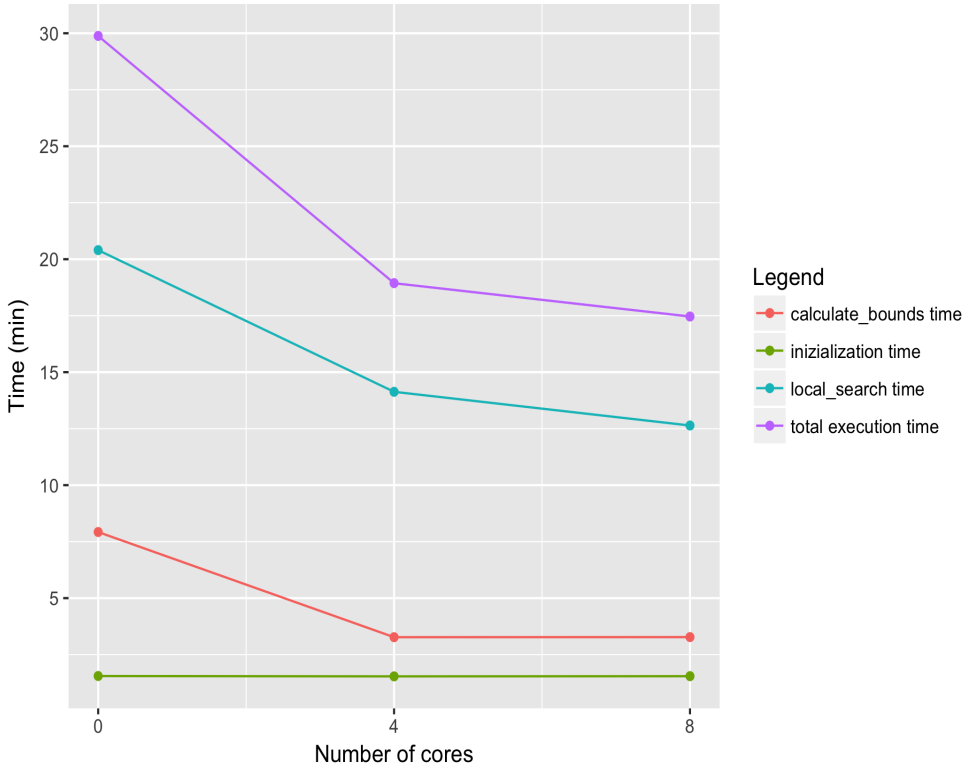
Many repetitions⇒ very long times with cache off ($\simeq$ 41 minutes with 4 cores and 4 applications)
To represent the worst scenario is more meaningful to turn on the cache and start from an empty database.

0% cache



⇒ total execution time reduced by 58% (4 applications)

A realistic scenario: **50% cache**



$\Rightarrow$ total execution time reduced by 41% (4 applications)

Conclusions

Both policies lead to remarkable improvements of the initial solution; improvements are greater for Separing policy.

The parallelization leads to a big decrease on execution time:

- **Alterning policy:** from 33% to 78%
- **Separing policy:** from 41% to 58%
- time improvement decreases as the percentage of already computed predictions increases

Alterning policy is quite fast; with:

- parallelization
- small amount of already computed values
- appropriate maximum number of iterations
- appropriate maximum number of considered candidates

it should meet the ideal deadline of 10 minutes.

Possible extensions

The class structure allows easy reuse of the code and the policy structure allows easy extension of local search methods.
Indeed if another type of local search has to be implemented it could be done with a policy class implementing a *local_search()* method.

Variants of local search could:

- do multiple virtual machines exhanges
- explore the neighborood more deeply allowing cores exchange between more than two applications at each iteration

References

Danilo Ardagna, Enrico Barbierato (Polimi), Jussara M. Almeida, Ana Paula Couto Silva (UFMG) (2016)
D3.2 - Big Data Application Performance models and run-time Optimization Policies
www.eubra-bigsea.eu