A Framework for the Design of Parallel Adaptive Libraries on Hard Computational Problems

Abstract. In this work, we present the Adaptive Multi-Selection Framework (AMF). The AMF is an API developed for helping designers to develop optimized combinations of multiple algorithms solving a same problem in function of the physical architecture and algorithm behavior. AMF offers a simple and generic model for developing automatic combination of algorithms. In this model, the user needs to specify the set of algorithms to be combined and a representative benchmark of instances of the problem solved by the algorithms. This generic solution has advantages over many existing solutions for making automatic combination that are specific to a fixed set of algorithms or computational problems. Automatic combinations of algorithms are made in AMF with the multi-selection technique. For each instance of a computational problem, its resolution under multi-selection comprises a selection of a subset of candidate algorithms followed by a concurrent run of the selected algorithms with a smart resource sharing. The resource sharing is decided according to the physical architecture, the problem instance and the time allowed to compute it. The multi-selection strategy provides excellent results when there is a large variance of execution time per instance. The actual implementation of AMF is built for shared memory architectures. However, it can be extended to distributed ones. The AMF principles have been validated in particular on the Constraint Satisfaction Problem and the Satisfiability Problem.

Keywords: adaptive algorithms, automatic algorithm combination, resource sharing

1 Introduction

The continuous evolution of algorithmics is leading to a huge amount of algorithms available for each computational problem. For the same problem, the performance of these algorithms might vary depending on many aspects as the problem instance considered or the machine architecture [1, 19]. In order to obtain good performances, there is the need for solution that can combine smartly various algorithms solving the same problem. The huge variety of computational problems for which such solutions are required, the large amount of algorithms and machine architectures suggest that generic combinations that can be automated must be prioritized. In this paper, we focus on the automatic combination of multiple algorithms solving the same problem, specially those related to hard computational problems. Our main objective is to provide a framework that eases the implementation of automatic combination of algorithms.

1.1 Contributions

We propose the Adaptive Multi-Selection Architecture for the design of adaptive libraries. AMF has been designed for easing the task of developing automatic combination of algorithms. For this ends, AMF typically needs in input a set of candidate algorithms and a benchmark for tuning algorithms performances. It includes a set of components that can then generate optimized combinations of the provided input algorithms in the machine architectures. AMF works also like a problem solver with a simple interface that gives the possibility to solve a problem instance through a specific combination of algorithms that has been created internally in AMF. The resolution of each problem instance using the AMF interface is based on the multi-selection technique. It comprises a selection of a set of candidate algorithms for the instance and the execution of the algorithm combination related to this selection. These combinations are defined in AMF as algorithms portfolio [11] that are a set of algorithms to be run each with a predefined number of resources an stopped as soon as one algorithm ends its execution. The actual version of AMF is designed for shared memory architectures and support parallelism based on threads on process. Moreover, we provide some validations of its utilization for defining automatic combination of algorithms for the Constraint Satisfaction Problem (with sequential and parallel algorithms).

1.2 Text organization

The rest of the paper is organized as follows: Section 2 presents the multi-selection technique used in AMF for solving instances. The architecture of AMF in a component point of view is presented in Section 3. Section 4 gives some example of utilization of AMF on the Constraint Satisfaction Problem. Related works are presented in Section 5 and we conclude in Section 6.

2 The Multi-selection technique

We suppose that we have a parallel machine architecture and a finite set of parallel algorithms solving a same problem P. Within multi-selection, each instance of P is solved in three phases. The first phase consists in a selection of candidate algorithms for the instance. The second phase consists in sharing resources of the parallel algorithm between the selected algorithms. The third phase is a concurrent execution of the selected algorithms under the adopted resource sharing. Each phase is described in the following.

2.1 Selection of the candidate algorithms

With multi-selection, the resolution of the instance can be done in the *online* or *offline* mode. In the offline mode, all candidate algorithms are selected in this step of the multi-selection. In the online mode, a subset of candidate algorithms is selected. The motivation of the online mode is that it might be possible depending on some instance properties to detect algorithms that can solve it quickly.

The online and offline modes lead to different types of overhead on instance resolution. We illustrate this as follows: lets denote the execution time of an instance I as $t(I)$. In multi-selection, we have $t(I) = t_s(I) + t_{rs}(I) + t_{ep}(I)$ where, t_s, t_{rs} and t_{ep} are, respectively, the time for selecting a subset of algorithms, the time for computing a resource sharing, and the time of executing selected algorithms with the computed resource sharing. Since the selection of algorithms is the same for each problem instance in the offline mode, one can pre-compute the optimal resource sharing that will be re-used for all instances. Typically, $t_s(I) + t_{rs}(I)$ is negligible in the offline mode. The algorithms selected at this phase of the execution will then be executed concurrently. However, it is only the result of one execution that will be exploited at the end. This means that the more selected algorithms, the more the overhead in the resolution. The philosophy of the online mode is to try to reduce $t_{ep}(I)$ even if it will lead to more important values of $t_s(I) + t_{rs}(I)$.

2.2 Computation of the optimal resource sharing

For the determination of resource sharing (second phase in multi-selection), AMF uses the *dRSSP* model [5]. Giving a computational problem $\mathcal{P}$, the inputs of this model are : a finite set of algorithms $\mathcal{A} = \{A_1, \dots A_k\}$ (algorithms selected in the first phase of multi-selection), a finite set of homogeneous computation units $P = \{0, \dots, m\}$, a finite set $\mathcal{I} = \{I_1, \dots, I_n\}$ of representative instances of $\mathcal{P}$, cost values $C(A_i, I_j, p)$ giving for each algorithm $A_i \in \mathcal{A}$ and instance $I_j \in \mathcal{I}$ its execution time $C(A_i, I_j, p)$ when executed on $p \in P$ resources.

The resolution of instances in the *dRSSP* model is based on algorithm portfolio. Lets define a resource sharing as a vector $S = (S_1, \dots, S_k)$ such that $S_i \in P$ and $\sum S_i \leq m$. Here, S_i is the number of resources in which A_i is executed. We can define the resolution time of any instance under this resource sharing as $C(S, I_j) = \min\limits_{1 \leq i \leq k} \{C(A_i, I_j, S_i) | S_i > 0)\}$.

$$\boxed{\begin{array}{l} \text{Minimize } \sum_{j=1}^{n} C(S, I_j) \\ 1.\ S_i \in \{0, \dots, m\} \\ 2.\ \sum S_i \leq m \end{array}}$$

(a) MinSum optimization

$$\boxed{\begin{array}{l} \text{Minimize } \max\limits_{1 \leq j \leq n} C(S, I_j) \\ 1.\ S_i \in \{0, \dots, m\} \\ 2.\ \sum S_i \leq m \end{array}}$$

(b) MinMax optimization

Giving these inputs, we suppose in *dRSSP* that for $\mathcal{P}$ problem to solve, any of its instance will behave like one instance in $\mathcal{I}$. Therefore, a global approach to minimize the resolution time of problem instance can consist in finding the resource sharing S minimizing $\sum_{j=1}^{n} C(S, I_j)$. We will denote this as the **MinSum** optimization function. With the MinSum, one targets a good mean execution time for the resolution of $\mathcal{P}$.

With MinSum, variations can be observed between instances resolution time. In a competition settings where we have a finite set of instances to solve in a maximal amount of time, this might not be a problem. However, in a contexte where instances are not solved in block, variations between execution times of instances are sensitive. In this case for example, a good optimization goal is to minimize the maximal time we can wait for having the solution of an instance. This will be taken as the **MinMax** objective given by the function: minimize $\max_{1\leq j\leq n} C(S, I_j)$. Under the MinMax or MinSum objectives, one can easily show (in using results provided in [5]) that the problem of computing the optimal resource sharing is NP complete. Thus, heuristics must be used in the online mode in order to have an acceptable overhead on instance resolution.

2.3 Execution of algorithms with resource sharing

The last stage for an instance resolution under multi-selection is the concurrent execution of algorithms. As we have said previously we use the algorithm portfolio model of the execution. The advantage of this model is its applicability to a large class of algorithms.

2.4 Advantages of multi-selection

In this Setion, we discuss here the advantage of multi-selection as done in AMF with respect to the importance of selecting more than one algorithm for solving a problem instance, and algorithm ranking [20, 3], that is an alternative model of execution with multiple algorithms solving the same problem.

a)Multiple selection vs unique selection: Lets suppose that for solving an instance with some candidate algorithms $\mathcal{A} = \{A_1, \dots, A_k\}$ we selected a single algorithm. Lets also suppose that we have n instance to solve. Given a technique T_x, it will lead to a mean expected time denoted $E[T_x(n)]$ for solving the n instances. We consider that the risk of this technique is $E[T_x(n)] - t_{opt}$ where $t_{opt} = \sum_{i=1}^{n} t_i^*$.

When selecting of a unique algorithm, we might have a probability of p for selecting the right algorithm. For each instance i, lets denote by $t_i^1, \dots, t_i^k$ the time required to solve it by one algorithm of $\mathcal{A}$. Lets also assume that we have an equiprobability of having any algorithm as the wrong selected ones. The mean time for solving n instances by selection of a single algorithm is $E[S(n)] = \sum_{i=1}^{n} p.t_i^* + (1-p)(t_i^* + \frac{1}{k}\sum_{u=1}^{k}(t_i^u - t_i^*))$. For solving all instances with a multi-selection of k algorithms, we can expect a time of $E[M(n)] \leq \sum_{i=1}^{n} \alpha_i t_i^*$ (since we execute all algorithms concurrently) where α_i depends on the resource sharing. When isolating the optimal resolution time $t_{opt} = \sum_{i=1}^{n} t_i^*$, we have $E[S(n)] = t_{opt} + \frac{(1-p)}{k}\sum_{i=1}^{n}[\sum_{u=1}^{k}(t_i^u - t_i^*)]$ and $E[M(n)] \leq t_{opt} + \sum_{i=1}^{n}(\alpha_i - 1)t_i^*$.

It is reasonable to bound the value of α_i with, for example, the number of resources if there is a linear parallelism, and we have less algorithms than resources $k \leq m$. We will then have $\alpha_i \leq m$. Thus the risk in offline multi-selection can be bounded at a fixed distance factor to the optimal solution while the quantity $\frac{(1-p)}{k} \sum_{i=1}^{n} [\sum_{u=1}^{k} (t_i^u - t_i^*)]$ can be arbitrarily large. This means that the selection of a unique algorithm is more risky than the multiple selection in the offline mode.

Smart values of α_i could be proposed to minimize $M(n)$. In particular, we can share resources to algorithms in order to tolerate, an important overhead on instances whose execution time is small for all algorithms. This is the key point of heuristics for optimizing resource sharing in AMF.

The bigger the value of p, the smaller the risk in unique selection. This is the main interest for an online multi-selection. Indeed, if it is possible to detect with high probability what is the optimal algorithm, then, it might be possible to have a process that can choose for each instance i a subset of k_i algorithms ($k_i \leq k$ and $\sum_{i=1}^{n} k_i < nk$) such that the best algorithms on the instance is included on the subset with a probability of 1. Thus, the expected time for the portfolio will be $E[M(n)] \leq \sum_{i=1}^{n} \beta_i t_i^*$ and since $k_i \leq k$, we have less algorithms executed concurrently and we could expect that $\sum_{i=1}^{n} (\beta_i - 1) t_i^* \leq \sum_{i=1}^{n} (\alpha_i - 1) t_i^*$.

b)Algorithm portfolio vs algorithm ranking: In algorithm ranking, the selected algorithms are not executed concurrently. A fixed amount of time or cutoff and a ranking between algorithms is decided. Then each algorithm is executed on the instance to solve during the cutoff time decided and following the decided ranking. The executions is stopped when a solution is found.

This model of execution is certainly a good alternative to algorithm portfolio. One advantage is that there is no need to compute a resource sharing since each algorithms is executed with all resources. Algorithm ranking has been used with interesting results in [20].

The non-computation of resource sharing in algorithm ranking is repaced by the determination of the cutoff time. A problem for this is that there might be impossible for all algorithms to provide a solution under the chosen cutoff. This means that we will not solve this instance without modifying the cutoff in the execution. Lets suppose that the cutoff has a value of t and guarantee that there is at most one instance solve under this cutoff. Given a ranking of heuristics, an instance will be solved by the first algorithms or if not, the second and if not the third etc. We suppose that when an instance has an equiprobability p to be solved at each rank. Thus, the time for solving n instances will be $E[R(n)] \geq \sum_{i=1}^{n} (pt_i^* + p(t + t_i^*) + p(2t + t_i^*) + \cdots + p((k-1)t + t_i^*))$. In isolating the optimal resolution time, we have $E[R(n)] \geq t_{opt} + p \sum_{i=1}^{n} \frac{k(k-1)}{2} (t_i^* + (t - t_i^*))$. The risk again depends on the cutoff factor. In order to guarantee that each instance will be solved under the cutoff, this value must be in general big, we find the algorithm portfolio approach less risky.

The multi-selection technique is a less risky approach when there is an important difference between execution of algorithm. This is in particular of heuristics solving hard computational problem. In the next section, we illustrate how multi-selection is applied with AMF.

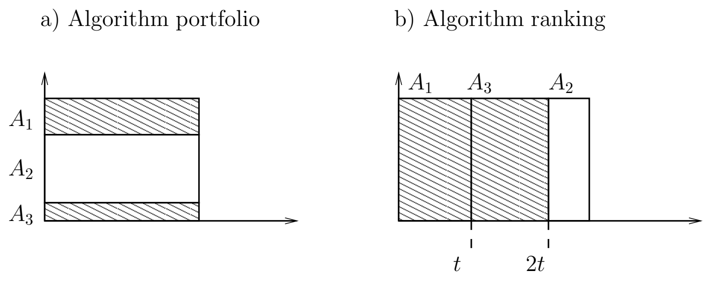


Fig. 1.: Example of execution pattern in algorithm portfolio and algorithm ranking. In the case of portfolio, all algorithms are executed concurrently for solved the instance while they are executed given a rank and under a time limit (here t). In both cases, we have useless executions (dashed in the figure).

2.5 Example of instance resolution with AMF

The AMF framework is a C++ API for automatic combination of algorithms. In Figure 2, we give some examples of its utilization. In Figure 2a), we describe the insertion of a new application in AMF. In AMF, the description of a computational problem is made through applications. The application in the example is related to the CSP (the constraint Satisfaction Problem) and has three sequential algorithms (max-deg, min-dom, max-dom/deg) that are CSP heuristics. Some requirements must be fullfilled previously for a successful application insertion. In Figure 2b) in particular, we show that the algorithms solving the computational problem of the applications must be declared in the file AMF_AMA_ALG.h. Figures 2c) and 2d) address the insertion of a new benchmark for CSP. Similarly to the creation of an application, the insertion is made by invoking a function of the class *Learner*. Finally in Figures 2e) and 2f), we show how to execute an instance and update information about an inserted application.

Applications and benchmark in AMF are manipulated through block of data of type *AMF_Application* and *AMF_Benchmark*. Each application is associated to a benchmark and has a set of algorithms. The application can be sequential as in Figure 2 (its type is AMF_SEQ) or parallel.

All problem instances in AMF are specified using an **AMF_ Instances**. Such instance is basically a structure with fields describing the input argument of the problem instances or the computational problem (in the example "CSP") that we are referring to. In the example, we defined a instance of the application CSP and invoked an AMF solver (object of the class **AMF_ Solver**) for its resolution. After the resolution of the instance, the result is given as a void pointer that from a transtyping operation we can re-structure.

This example gives a tour of operations and objects in AMF. The internal structure of the framework is described in the next section.

```
#include<iostream>
#include "AMF.h"
AMF_Application A;
char[3][10] Alg = {"max-deg",
"min-dom", "max-dom/deg"};
A.name = new char[3];
strcpy(A.name, "CSP");
A.type = AMF_SEQ;
A.alg_number = 3;
A.Alg_name = Alg;
AMF_Learner *L; L = new
AMF_Learner(); L->add_app(A);
```

(a) Creation of an application

```
//....others headers
void* max-deg(void *){//code
};
void* min-dom(void *){//code
};
void* max-dom/deg(void
*){//code };
//....others headers
```

(b) Required content of AMF_AMA_ALG.cpp

```
#include<iostream>
#include "AMF.h"
AMF_Benchmark Bench;
Bench.Benchfilename = new
char[30];
strcpy(Bench.Benchfilename,
"CSPBenchmark");
Bench.BenchReaderName = new
char[30];
strcpy(Bench.Benchfilename,
"CSPReader");
Bench.size = 150; AMF_Learner
*L; L = new AMF_Learner();
Bench.app_id =
L->getId("CSP");
L->add_bench(Bench);
```

(c) Creation of a benchmark

```
//....others headers
void CSPReader(FILE *F, void
*argInstance){//code};
/* This function given a file
descriptor F towards data of
CSP returns the next CSP
instance on which the pointer
of F is*/
//....others headers
```

(d) Required content of *reader.cpp* for creating the benchmark

```
#include<iostream>
#include "AMF.h"
AMF_Instances I; AMF_Solver
*SOL;
int **tab;
// Init tab with values of the
CSP problem void *result, int
*value;
I.app_name = "CSP";
I.arg = (void *) tab;
I.mode = AMF_OFFLINE;
I.objective = AMF_MINSUM;
SOL = new AMF_Solver();
SOL->solve(I, result);
value = (int *)result;
cout<< "the result is "
>>*value ;
```

(e) Execution of an instance

```
#include<iostream>
#include "AMF.h"
AMF_Application A;
char[4][10] Alg =
{"min-deg","max-deg",
"min-dom", "max-dom/deg"};
A.name = new char[3];
strcpy(A.name, "CSP");
A.type = AMF_SEQ;
A.alg_number = 4;
A.Alg_name = Alg;
AMF_Learner *L; L = new
AMF_Learner();
L->update_app(A);
```

(f) Update of an application

Fig. 2.: Possible usages of AMF

3 The AMF component structure

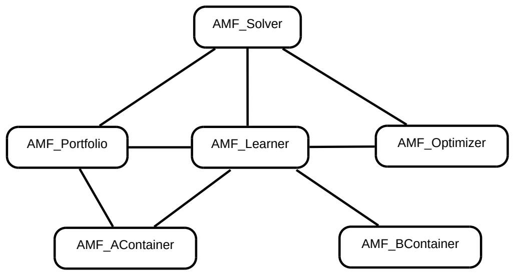


Fig. 3.: AMF Components and relations between them

Figure 3 describes AMF internal components and dependencies among them. The key component of this architecture is the **AMF_ Learner** that centralizes information about manipulated applications , available benchmarks, platform settings. The AMF_ Learner works as a tuning engine that learns from the architecture and generates automatic combination of algorithms for registered applications and as a knowledge base given for other components information about automatic combination supported in AMF. The **AMF_ Portfolio** engine role is responsible for the portfolio execution of defined combinations of algorithms. It communicates with the **AMF_ AContainer** that contains all algorithms solving an application defined in AMF. The **AMF_ BContainer** contains multiple source files of instances used for tuning application. Ideally, instances must capture the difficulty of the related computational problem. The Benchmark and algorithms containers can be modified by the user. The user submits a request for the resolution of an instance in AMF through the **AMF_ Solver**. It calls the **AMF_ Optimizer** for computing an adequate resource sharing and then run the a porfolio engine with the appropriate resource sharing. In the next section we give details about these components.

3.1 The Solver

The solver component is constituted mainly by the class AMF_ Solver. In this part we will discuss about the following methods of this class:

```
void solve(AMF_Instance I, void *argout);
void set_MultiSelector(char * app_name, char *method_name)
```

The *solve* method that takes in inputs an instance (I) and output a pointer toward a void (*argout*) containing the solution of the instance. The main elements of AMF_ Instance are:

- the application name to which the instance refers,
- a void pointer toward the input data describing the instance,
- the mode of resolution chosen (online or offline),
- the timelimit for instance resolution (this time is significant only if the chosen mode is online),
- the type of optimization (MinSum or MinMax).
- a proportion field $p \in [0, 1]$.

The method *solve* will then communicate with the learner to have information about the applications (in particular, the benchmark file tuned for its). Using this information, it will ask a resource sharing to the optimizer and will finally call a portfolio engine for its execution.

For each application, AMF gives the possibility to define an appropriate method for selecting algorithms to execute in portfolio (a *selector*). In particular, if the value of the proportion field $p = 0$, then it is the selection method defined by the user that will be invoked on the instance. If not, the *solve* method will randomly select a propotion of p algorithms on which a resource sharing will be defined. The definition of a personalized method for algorithm the multiple selection of algorithms in the online mode is done through the method: *set_MultiSelector* of the class Solver.

The inclusion can only be effective if in the file *MultiSelector.cpp* a method with the signature *void method_name(AMF_Instance I, int tab[], int k)* must be defined. The implementation of this signature must ensure that *method_name* modifies the array *tab* for indicating among the k algorithms available for the application, the ones selected. The modification must set $tab[i] = 1$ if the algorithm i is selected.

In AMF, a constant array of functions defined in the file *MultiSelector.h* is kept towards selector functions. This pointer is used by the solver to have for each application the selector defined for it. When a new selector is defined, the content of *MultiSelector.h* is re-generated in order to update the pointer of selectors.

For defining proper function that operates multiple selections of algorithms, works done in [14, 2, 7] can be used.

3.2 The Optimizer

The optimizer in AMF is used in two main scenarios:

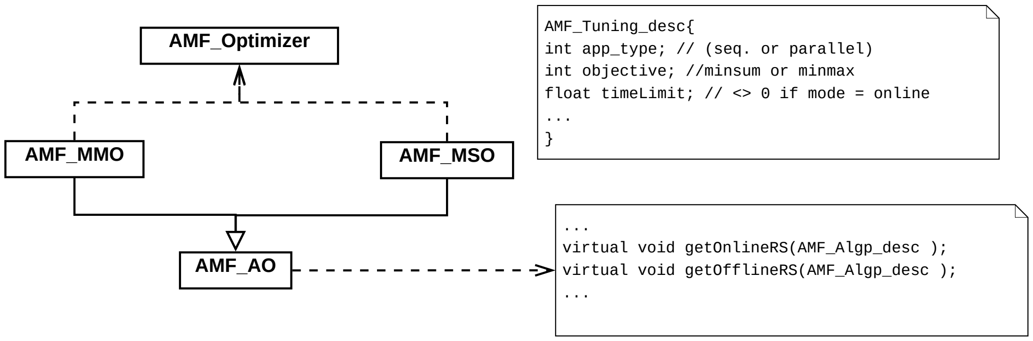
- At the installation time or when a new benchmark data is provided for an application.
- For the resolution of an instance in the online mode, it is called by the Solver component to a good resource sharing within time limit.

The main functions used in the Optimizer are :

```
void getOnlineRS(AMF_Algp_desc)
void getOfflineRS(AMF_Algp_desc)
```

When the *solve* function is called in the Solver, a request is submitted to the optimizer in calling one function of these functions (*getOnlineRS* if the instance resolution mode is online). The input data of type *AMF_ Algp_ desc* comprises most information about the related AMF Instance but also, the list of algorithms, the number of resources or the time limit for the computation of the resource sharing. The Optimizer then computes an appropriate resource sharing and returns it.

The Optimizer structure The Optimizer components is organized in 4 main classes: AMF_ Optimizer, AMF_ AO, AMF_ MS, AMF_MMO. Dependencies between these classes are presented in Figure 4. The two classes AMF_ MSO,



```
AMF_Tuning_desc{
int app_type; // (seq. or parallel)
int objective; //minsum or minmax
float timeLimit; // <> 0 if mode = online
...
}
```

```
...
virtual void getOnlineRS(AMF_Algp_desc );
virtual void getOfflineRS(AMF_Algp_desc );
...
```

Fig. 4.: Classes of the optimizer component. The function *getOnlineRS* and *getOfflineRS* are designed to compute a resource sharing in an online or offline mode.

AMF_MMO are specialized on the computation of resource sharing under respectively the MinSum and MinMax objectives. These two classes are derivated from an abstract optimizer class (AMF_ AO). Finally, the class AMF_ Optimizer works as interface of communication for other components.

For computing resource sharing of selected algorithms, we have for each optimizer type, 5 heuristics for optimization when the application is parallel or when it is sequential. We listed these heuristics in table 1. Detailed explanations on the heuristics MAG, MA, RAND, WTA can be found in [5]. For HIF, we refer details can be found in [**?**]. The heuristic for the case of sequential algorithms is just adaptation of the parallel case where we limited the number of possible resources for each algorithm to 1.

In the heuristics implementations, we added small changes for MAG. The idea of its original implementation consists of selecting a number g of algorithms on which all possible assignments of resources are explored. If for the g selected heuristics, we explore an assignement of resources that use a total of m_g re-

Optimization of parallel algorithms ($\lvert\mathcal{A}\rvert = k$, $\lvert\mathcal{I}\rvert = n$ and m resources)		
Heuristics	Approx. ratio	Time complexity
HIF	arbitrary	$O(\min(k,m).(n^2k^2 + km^2))$
MAG	$k - g + 1$	$O(n2^{k-g}.(m+1)^g.(nk))$
MA	$2k - 1$	$O(k)$
RAND	arbitrary	$O(k)$
WTA	arbitrary	$O(nk)$
Optimization of sequential algorithms		
HIF^s	arbitrary	$O(m.n^2k^2)$
WTA^s	arbitrary	$O(nk)$
RAND^s	arbitrary	$O(k)$
OPT^s	1	$O\binom{k}{m}$

Table 1.: Heuristics used for optimization, guaranteed approximations ratio and complexity

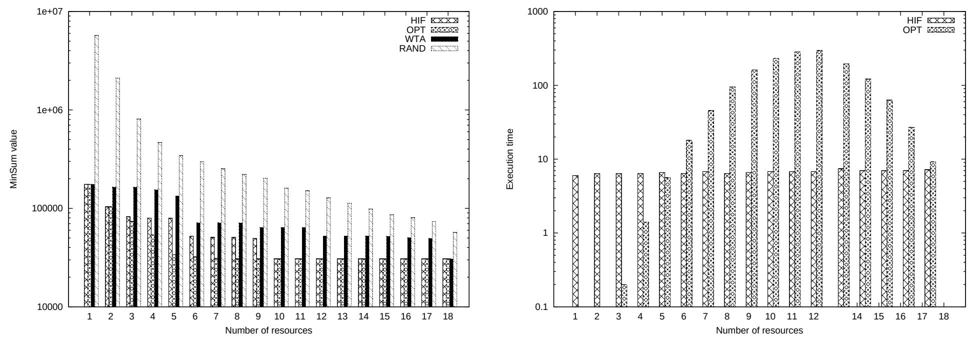
sources, we share fairly the $m - m_g$ resources to the remaining algorithms (each algorithms then have approximately $\frac{(m-m_g)}{k-g}$ resources). One can easily notice that when $g = k$, the MAG heuristic will give the optimal solution. For $g < k$, it has an approximation ratio of $k - g + 1$ when in the optimal solution all algorithms have one resource. We modified these heuristics in assuming that beyong the remaining algorithms, some might not have any resources in the exact solution. So, we selected any possible subsets of algorithms $k' < k - g$ algorithms besides the remaining ones to which we fairly share resources. This modification keeps the guarantee of $k - g + 1$ in all cases.

One can notice that when $g = k$, MAG gives the optimal solution. In the online case, multiple choices are possible because there are multiple heuritics, and the MAG heuristics can be executed with differents g values. Since we have a time limit function, we have a tradeoff to ensure between the quality of the solutions proposed and the time given for the optimization. In the next section we explain how optimization is done in for the online mode.

Optimization in the online mode For this, the online optimization mode comprises two steps: The construction of a *plan* that indicates which is a successive list of optimization heuristics that we will call to compute a resource sharing and the execution of the optimization heuristics following the plan. The best resource sharing issued of the plan is the one returned by the optimizer. Thus, the total time for deriving a resource sharing with the time limit t is $t_{rs}(I|t) = t_{cp}(I|t) + t_{ep}(I|t)$. There, $t_{cp}(I|t)$ is the time required for computing the plan and $t_{ep}(I|t)$ is the time required for executing the plan. The construction of the plan must guarantee that $t_{ep}(I|t) \leq t$ and $t_{cp}(I|t)$ must ideally be small.

For having small values of $t_{cp}(I|t)$, we built the plan using a model of the execution time of heuristics (we explain later how they are obtained). We also classify the optimization heuristics in three classes: the *polynomial fast* heuristics (MA, RAND, WTA), the *polynomial slow* heuristics (HIF), and the *exponential* heuristics (MAG, OPT).

Given a time limit t, the construction of the plan starts with an estimation of the time required for executing and selecting the best resource sharing from the first class of heuristics. If the estimation suggests that this part of the plan will not exceed t, one evaluates the possibility of including heuristics of *polynomial slow* with the remaining time limit estimated. Finally, *exponential* heuristics are considered. For this special case, we search for the best value of g that will lead to an optimization under the remaining time limit.



(a) MinSum value of resource sharing computed by the heuristics

(b) Execution time required for computing the resource sharing

Fig. 5.: MinSum cost and execution time of heuristics for sequential optimization on a benchmark of SAT solvers

We based this choice on some experimental results observed on these heuristics which show that the exponential heuristic has, in general, a better quality than the remaining heuritic but will be more time consumming. In Figure 5, we depict an experimentation made on a benchmark of sequential SAT solvers with the sequential heuristics for optimization. The results concerned the MinSum objective. In the experiments, there is a total of 23 sequential SAT solvers. It has been used for building a resource sharing with our heuritics, assuming that we have $1, 2, 3..$ resources. Details about the experiments can be found in [**?**]. About the execution time, the time of the *polynomial fast* heuristic is not reported because it is under 0.1. This Figure clearly exhibits the tradeoffs between the quality of the heuristic and the execution time required.

3.3 The Learner

The learner is the central component of the AMF architecture. Its is mainly involved in the following scenarios:

– It learns platform settings (mainly at the installation of AMF) and tunes the analytical perfomance model of the optimizer.
– It is the main component for application and benchmark registration;
– The learner is also invoked by other components when they need information about applications (e.g. the optimizer needs to know if an application is parallel or sequential)

The basic functions used for these operations are:

```
void tune_bench(int app_id)
void add_app(AMF_Application )
void add_bench(AMF_Benchmark)
AMF_Application getData_app(int app_id)
AMF_Benchmark getData_bench(int app_id)
```

We will discuss them in what follows.

Tuning of optimizer functions For tuning optimizer function, the learner supposes that the execution time of each of these functions can be as a real function written in the general form $f(m, n, k, g)$. This choice is motivated by the complexity result obtained in table 1. At the installation, the learner explores the data base of application and benchmarks for searching for possible value for (n, k) (m is the number of cores of the architectures).

For all valid points (n, k), the learner considers all values of $g \in \{1, \ldots, k\}$ and makes multiple executions (actually 50) of available optimzation heuristics (HIF, MA etc). It retains the mean execution time obtained from the executions and save it. It is important to notice that the space of valid points (n, k) may be modified when information related benchmarks in AMF are modified. In this case, the learner will re-consider all the new valid entries and tune the optimization functions on them.

Applications and benchmarks registration Benchmarks and applications registration are made in AMF through the learner functions: *add_ app* and *void add_ bench*. An application to add is described through a block of data of type AMF_ Application. This block is mainly characterized by:

– An application ID that is an integer unique for each application
– A name which is supposed to be the computational problem to which we refer (e.g. SAT for Satisfiability)
– A list of algorithms that are given through pointer toward algorithms implemented and available for the resolution of computational problem
– A type that can be **Sequential** if all algorithms available for the application are sequential or **Parallel** otherwise. This information is important for the computation of resource sharing.

An AMF_ Benchmark comprises mainly:

- An application ID that is the application referred by the benchmark
- A benchmark source file that are brute instances representative of the computational problem
- A benchmark reader that is a pointer towards a function that can extract an instance from the benchmark source file.

The registration of a new application will create automatically a unique identifier for its. It also initiates an action from the learner to inform the portfolio engine of this new registration. This is possible because the portfolio engine works with a pointer of functions towards the set of algorithhms available for each application. This pointer is defined in the file *AlgorithmPointers.h*. At the insertion or update of each application, the content of this file is re-generated by the learner to include new algorithms.

To any application, there is associated a unique AMF_ Benchmark block of informations. For a benchmark registration, the learner automatically generates a benchmark performance profile for it, and informs the benchmark container (by code generation as for the portfolio engine) of the presence of a new identified reader function, and tunes eventually the optimization heuristic if a new couple *(number of benchmark instances, number of algorithms)* is introduced. The benchmark performance profile will be used mainly by the optimizer. It defines for each benchmark instance and number of resources the mean execution time of all registered algorithms that can solve the instance. Finally, we recall that information about registered applications or benchmarks can be updated in AMF.

Communications with other components The learner is involved in multiple operations by other components when they need information about applications and benchmarks. To do so, they invoke its functions *getData_app* and *getData_bench*. To ease the access to this information, the learner maintains a table of applications and a table of benchmark. In Figure 6, we give a description of information related to applications and benchmarks that are manipulated by the learner.

3.4 The portfolio engine

The portfolio engine is mainly invoked by the Solver when a new instance is to be solved or by the Learner to generate benchmark performance profile.

The portfolio engine keeps a pointer towards algorithms for application registered in AMF. The code of these algorithms is available in the Algorithms container. Giving a defined resource sharing for an application, it can start on algorithms accessible from its pointer a concurrent execution of algorithms.

In the concurrent execution, if the application is sequential, the resource sharing indicates the algorithms that will be run. In the parallel cases, it gives the number of resources for the execution of each algorithms of the applications. In the case of sequential application, the engine has two options:

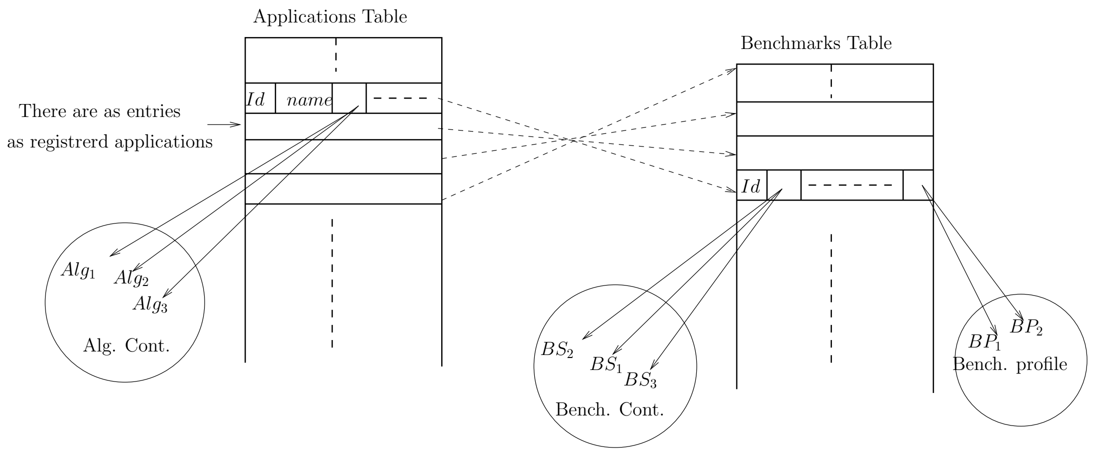


Fig. 6.: Data dependencies in the learner. Each entry of the application table is related to at most one entry in the benchmark table (when a benchmark is specified for the application). These entries also point on benchmark source files (in the benchmark container), benchmark performance profile files, and algorithms (in the algorithm container)

– It can execute the algorithms of the applications referred inside threads. The threads execution is based on the pthread API in C.
– It can execute the algorithms of the applications referred inside unix processes.

The choice between these options is made by the user in the description of the instance.

When the application is parallel, the portfolio engine only calls the different algorithms for which a not null number of resources is assigned in passing to them the number of resources required plus a Monitor object. These parallel algorithms must create at most the number of threads or processes authorized by the resource sharing and set at the end of their execution the result in the monitor object.

3.5 Algorithms and Benchmark container

The algorithms and benchmark containers comprises implementation of algorithms and benchmark readers. Source implementations must be located in the file *_AMA_ALG.cpp*. The signature of each algorithm in AMF must have the generic form *void* algorithmName(void *)*. Despite the fact that the input argument is of type *void **, its internal organization has the structure of type AMF_Argument. This structure comprises:

– A pointer towards the input argument that are data of the instance (of type void *).
– A pointer towards a monitor object used for the synchronization and collection of results.
– The number of resources for the execution

At this stage, it is interesting to recall the different level of algorithms choices that are done in AMF for an instance resolution. We give a description of this in Figure 7.

The benchmark reader is used to read benchmark source file related to an application. For this, it needs the pointer towards the file to be read (pointer of type FILE *) and the application to which we refer. It also implements a function *void * readInstance(int app_number)* that at each call returns one instance from the benchmark source file that given by the descriptor *filedesc*. It is important to notice that this function simply call the reader that must be provided by the user. The signature of all readers must be defined in the file *reader.h* and has the form *void readerName(FILE *F, void *arg)*. The implementation must ensure that on the file pointed by *F*, a call of the reader function returns an instance in the void pointer.

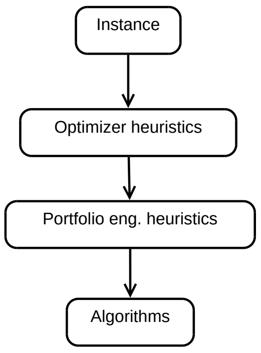


Fig. 7.: Levels of algorithmic choices between algorithms (A revoir)

4 Example of Constraint Satisfaction

5 Related Work

The main philosophy that has been considered in the automation of algorithms combination is the problem specific approach. In the problem specific approach, an adaptive algorithm that can adjust its execution depending on the machine architecture is designed for each computational problem. Given a computational problem P, the adaptive algorithms can manage a pool of candidate algorithms solving the same problem. Depending on the machine architectures and instance of P to solve, it selects the most appropriate algorithm to obtain good performances. To be able to make these choices, the adaptive algorithm will learn how to proceed during its installation on each platform from a finite benchmark of P instances. The problem specific approach has been applied successfully on many computational problem as Matrix multiplication [17, 9], Sorting [1, 19, 4] or Fast Fourrier Transform [19, 13].

Lets now consider the problem specific approach philosophy in the evolution of algorithms and machine architectures. We can say that if only the machine architectures change, the philosophy of adaptive algorithms suggests that we might not necessarily need to re-design the problem specific approach (since the algorithm adapts itself to the platform). If however, the set of known algorithms for the problem changes, a design of adaptive algorithms is required in order to include this new algorithm (otherwise, one can have an external algorithm more performant than the adaptive algorithm). However, one cannot anticipate the algorithmics evolution on a computational problem. Moreover, depending on the utilization context, the set of algorithms required for solving a problem can change. For example, there is no advantage of having the quicksort algorithms in a context where we have just small number of data to sort [12]. To deal with this, one can observe how parallelization is done in parallel computation. Parallel programming proposes both problem specific library (on sorting, searching etc.) and general API as MPI and pthreads for simplifying the implementation of parallel program. Considering this example, we can say that the design of automatic combination of algorithms also requires general API that eases the implementation task without being specific to a particular computational problem. This point of view has received some interests over the last decade.

Among relevant studies, we have the AEOS method [8] used on Self Adaptive Numerical System [10] that deals with automatic selection between multiple implementations of the same algorithm (in changing for example loop orders in the implementation). AEOS has been used in particular as a methodology for tuning and selecting kernels on dense and sparse linear algebra problems. In [18], a framework is proposed for composing a general parallel algorithm and sequential algorithms such as to automatically adjust the load balance in parallel execution. This solution is typically well suited when there are parallel algorithms based on divide and conquer with few communications. In [6], a framework (mainly conceptual) for dynamic adaptation of parallel codes (in a grid context) is proposed. The works that are certainly the most closed to the contribution of this paper are those done on hyper-heuristics [7]. The idea of hyper-heuristics is to develop generic search procedures that work on a space of algorithms solving the same problem. Typically, this search must select the most performant algorithm solving a computational problem. This idea has been validated on many cases like the resolution of time tabling problem [16].

In this paper, we propose a framework for developing adaptive and parallel programs based on automatic combination of algorithms. We differ mainly from existing works on the fact that our strategy for automatic combination of algorithm is based on multi-selection [11].

6 Conclusion

Integrate more advanced multiselector reduce the time required for plan computation in parallelizing optimizer distributed version

Improve the computation of the plan with a knapsack based approach.

References

1. P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: An adaptive, generic parallel C++ library. *Lecture notes in computer science*, pages 193–208, 2003.
2. S. Bhowmick, V.Eijkhout, Y.Freund, E. Fuentes, and D. Keyes. Application of machine learning to the selection of sparse linear solvers. `www.tacc.utexas.edu /~eijkhout/Articles/2006-bhowmick.pdf`.
3. Sanjukta Bhowmick, Lois C. McInnes, Boyana Norris, and Padma Raghavan. The role of multi-method linear solvers in pde-based simulations. In *ICCSA (1)*, pages 828–839, 2003.
4. Eran Bida and Sivan Toledo. An automatically-tuned sorting library. Technical report, School of Computer Science, Tel-Aviv university, 2006.
5. M. Bougeret, P.F. Dutot, A. Goldman, Y. Ngoko, and D. Trystram. Combining multiple heuristics on discrete resources. In *11th Workshop on Advances in Parallel and Distributed Computational Models APDCM, (IPDPS)*, 2009.
6. Jérémy Buisson, Françoise André, and Jean-Louis Pazat. A framework for dynamic adaptation of parallel components. In *PARCO*, pages 65–72, 2005.
7. Edmund K. Burke, Mathew R. Hyde, Graham Kendall, Gabriela Ochoa, Ender Ozcan, and John R. Woodward. Exploring hyper-heuristic methodologies with genetic programming. In Christine L. Mumford and Lakhmi C. Jain, editors, *Computational Intelligence*, volume 1 of *Intelligent Systems Reference Library*, chapter 6, pages 177–201. Springer, 2009.
8. J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R.C. Whaley, and K. Yelick. Self-adapting linear algebra algorithms and software. *Proceedings of the IEEE*, 93(2):293 –312, feb. 2005.
9. J. Dongarra, G. Bosilca, Z. Chen, V. Eijkhout, GE Fagg, E. Fuentes, J. Langou, P. Luszczek, J. Pjesivac-Grbovic, K. Seymour, et al. Self-adapting numerical software (SANS) effort. *IBM Journal of Research and Development*, 50(2-3):223–238, 2006.
10. Victor Eijkhout, Erika Fuentes, Thomas Eidson, and Jack Dongarra. The component structure of a self-adapting numerical software system. 33, June 2005.
11. B.A. Huberman, R.M. Lukose, and T. Hogg. An economics approach to hard computational problems. *Science*, 275(5296):51–54, 1997.
12. Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley second Edition, 1998.
13. Steven G. Johnson Matteo Frigo. FFTW: An adaptive software architecture for the FFT. In *proceedings of the International Conference on Acoustics, Speech and Signal Processing*, Seattle, Washington, May 1998. ACM SIGARC.

14. Y. Ngoko and D. Trystram. Combining numerical iterative solvers. In *Parallel Computing*, pages accepted, to be published, Lyon, France, 2009.
15. Y. Ngoko and D. Trystram. Combining SAT solvers on dicrete resources. In *HPCS, International Conference on high performance computing and simulation*, pages 153–160, Leipzig, Germany, 2009.
16. Rong Qu, Edmund K. Burke, and Barry McCollum. Adaptive automated construction of hybrid heuristics for exam timetabling and graph colouring problems. *European Journal of Operational Research*, 198(2):392–404, 2009.
17. Antoine Petitet R. Clint Whaley and Jack Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.
18. Jean-Louis Roch, Daouda Traoré, and Julien Bernard. On-line adaptive parallel prefix computation. In *Euro-Par*, pages 841–850, 2006.
19. María Jesús Garzarán Xiaoming Li and David A. Padua. A dynamically tuned sorting library. In *proceedings of the 2004 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 111–124, San Jose, California, June 2004. IEEE Computer Society.
20. Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Satzilla: Portfolio-based algorithm selection for sat. *J. Artif. Intell. Res. (JAIR)*, 32:565–606, 2008.

This article was processed using the LaTeX macro package with LLNCS style