A MASSIVELY PARALLEL GENERAL PURPOSE MULTI-OBJECTIVE OPTIMIZATION FRAMEWORK, APPLIED TO BEAM DYNAMIC STUDIES

Y. Ineichen, A. Adelmann*, PSI, Villigen, Switzerland
C. Bekas, A. Curioni, IBM Research – Zurich, Switzerland
P. Arbenz, Department of Computer Science, ETH Zurich, Switzerland

Abstract

Particle accelerators are invaluable tools for research in the basic and applied sciences, in fields such as materials science, chemistry, the biosciences, particle physics, nuclear physics and medicine. The design, commissioning, and operation of accelerator facilities is a non-trivial task, due to the large number of control parameters and the complex interplay of several conflicting design goals.

We propose to tackle this problem by means of multi-objective optimization algorithms which also facilitate massively parallel deployment. In order to compute solutions in a meaningful time frame, that can even admit online optimization, we require a fast and scalable software framework.

In this paper, we present an implementation of such a framework and report first results of multi-objective optimization problems in the domain of beam dynamics.

INTRODUCTION

In contemporary scientific research, particle accelerators play a significant role. Fields, such as material science, chemistry, the biosciences, particle physics, nuclear physics and medicine rely on reliable and effective particle accelerators as research tools. Achieving the required performance is a complex and multifaceted problem in the design, commissioning, and operation of accelerator facilities. Today, tuning machine parameters, e.g., bunch charge, emission time and various parameters of beamline elements, is most commonly done manually by running simulation codes to scan the parameter space. This approach is tedious, time consuming and can be error prone. In order to be able to reliably identify optimal configurations of accelerators we propose to solve large multi-objective design optimization problems to automate the investigation for an optimal set of tuning parameters. Observe that multiple and conflicting optimality criteria call for a multi-objective approach.

We developed a modular multi-objective software framework (see Fig. 1) where the core functionality is decoupled from the "forward solver" and optimizer (master/slave). This allows to easily interchange optimizer algorithms, forward solvers and optimization problems. A "pilot" coordinates all efforts between the optimization algorithm and the forward solver. This forms a robust and general framework

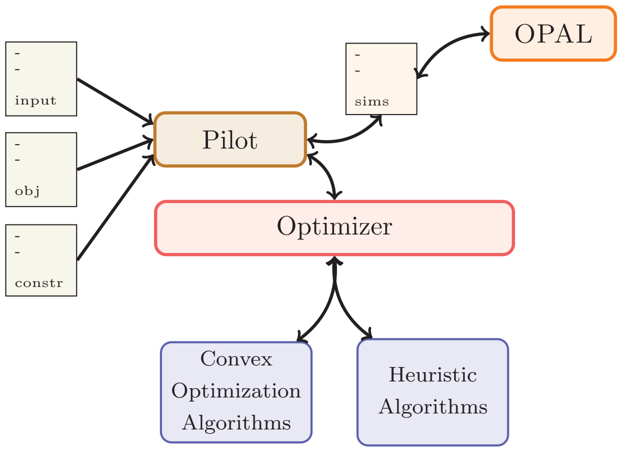


Figure 1: Multi-objective framework: the pilot (master) solves the optimization problem specified in the input file by coordinating optimizer algorithm and workers running forward solves.

for massively parallel multi-objective optimization. Currently the framework offers one concrete optimization algorithm, an evolutionary algorithm employing a NSGAII selector [1]. Normally simulation based approaches are plagued by the trade-of between level of detail and time to solution. We address this problem by using forward solvers with different time and detail complexity.

The first section covers a brief introduction to multi-objective optimization theory and describes the available optimizer. Next we discuss the implementation of the framework and present a proof of concept application of a beam dynamics problem.

MULTI-OBJECTIVE OPTIMIZATION

Optimization problems deal with finding one or more feasible solutions corresponding to extreme values of objectives. If more than one objective is present in the optimization problem we call this a multi-objective optimization problems (MOOP). A MOOP is defined as

$$\begin{aligned} \min \quad & f_m(\mathbf{x}), & m = 1 \dots M \quad (1)\\ \text{s.t.} \quad & g_j(\mathbf{x}) \geq 0, & j = 0 \dots J \quad (2)\\ & x_i^L \leq \mathbf{x} = x_i \leq x_i^U, & i = 0 \dots n, \quad (3) \end{aligned}$$

where we denote f as the objectives (1), g the constraints (2) and x the design variables (3).

* andreas.adelmann@psi.ch

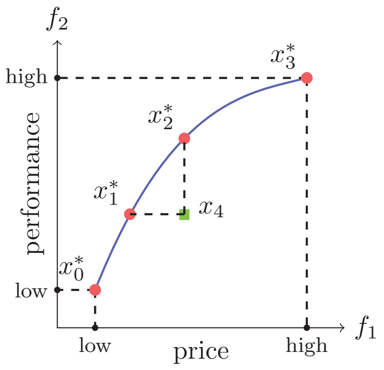


Figure 2: Two competing objectives can not be optimal at the same time. Red points represent Pareto optimal points, while x_4 is dominated (exhibits a worse price per performance ratio than e.g. x_2^*) by all points on the blue curve (Pareto front).

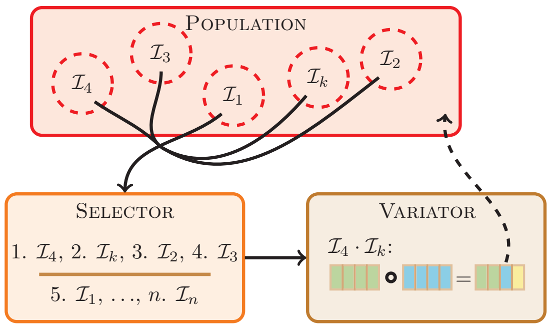
Often, we encounter conflicting objectives complicating the concept of optimality. To illustrate this, let us consider the problem of buying a car. Naturally, we want to get the best performance for the lowest price. This can be formulated as MOOP (4).

$$\begin{array}{ll} \text{min} & \text{price} \\ \text{max} & \text{performance} \\ \text{s.t.} & \ldots \end{array} \tag{4}$$

Obviously it is not possible to get the maximal performance for the lowest price and a trade-off decision between performance and price has to be reached (see Figure 2). Since not every choice is equally profitable for the buyer (car x_4 costs as much as x_2^* but offers less performance), we pick trade-offs (red points) that are essentially "equally optimal" in both conflicting objectives, meaning we cannot improve one point without hurting at least one other solution. This is known as the notion of Pareto optimality.The set of Pareto optimal points (blue curve) form the Pareto front or surface. All points on this surface are optimal.

Once the shape of the Pareto front has been determined the buyer can *specify preference*, balancing which features are more important. This is called *a-posteriori* preference specification since we select a solution after all possible trade-offs have been presented to us. The other alternative is to specify preference *a-priori*, e.g., by weighting (specifying preference before solving the problem) and combining all objectives into one and applying a single-objective method to solve the problem (yielding only one solution). In many situations preference is not known *a-priori* and an *a-posteriori* preference specification helps conveying a deeper understanding of the solution space. The Pareto front can be explored and the impact of trade-off decision become visible.

Sampling Pareto fronts is far from trivial. A number of different approaches have been proposed, e.g. evolutionary algorithms, sampling, simulated annealing, swarm methods and many more. The next section describes how evolutionary algorithms (available in our framework) are used to approximate the Pareto front of an multi-optimization problem.



POPULATION
$\mathcal{I}_4$
$\mathcal{I}_3$
$\mathcal{I}_1$
$\mathcal{I}_k$
$\mathcal{I}_2$
SELECTOR
1. $\mathcal{I}_4$, 2. $\mathcal{I}_k$, 3. $\mathcal{I}_2$, 4. $\mathcal{I}_3$
5. $\mathcal{I}_1$, …, n. $\mathcal{I}_n$
VARIATOR
$\mathcal{I}_4 \cdot \mathcal{I}_k$:

Figure 3: Schematic view of interplay between selector and variator.

Evolutionary Algorithms

Evolutionary algorithms (EA) are loosely based on nature's evolutionary principles to guide individuals towards an optimal solution (survival of the fittest). This "simulated" evolutionary process preserves entropy/diversity by using mutation and crossover to remix the fittest individuals in a population. Maintaining diversity in a population is critical for the success of all evolutionary algorithms.

In general, any evolutionary algorithm consists of the following components:

- *Genes*: properties/traits of an individual
- *Fitness*: a mapping from genes to fitness of individuals
- *Selector*: ordering relation to select k fittest individuals
- *Variator*: recombination (mutations and crossover) schemes for offspring generation.

In the context of our framework, genes correspond to specified design variables and individuals are ranked by their objective values (fitness). Evolutionary algorithms schematically work as depicted in Figure 3. Since there already exist plenty of implementations of evolutionary algorithms, we decided to incorporated the existing PISA library [1] into our framework. One of the advantages of PISA is that it separates variator from selector, rendering the library expendable and configurable. Implementing a variator was enough to use PISA in our framework and retain access to all available PISA selectors. As shown in Figure 3, the selector is in charge of ordering a set of d-dimensional vectors and selecting the k fittest individuals currently in the population. The performance of a selector depends on the number of objectives and the surface of the search space. So far, we only used an NSGA-II selector [2] exhibiting satisfactory convergence performance.

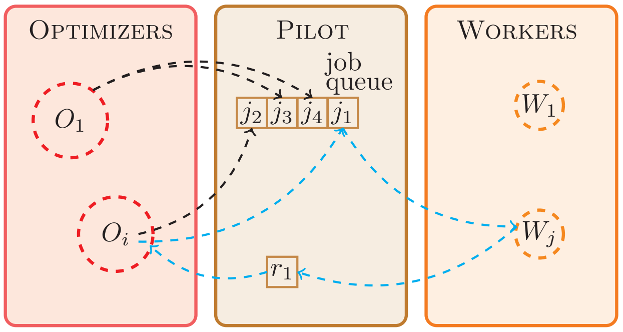


Figure 4: Schematic view of messages passed within the network. The cyan path describes a request sent from O_i to the PILOT being handled by W_j. Subsequently the result is returned to the requesting OPTIMIZER (O_i).

The task of the variator is to generate offspring and ensure diversity in the population. The variator can start generating offspring once the fitness of every individual of the population has been evaluated. This explicit synchronization point defines an obvious parallel bottleneck of evolutionary algorithms. In the worst case one group of processors is taking a long time to compute the fitness of the last individual in the population. During this time all other processors are idle and wait for the result of this one individual in order to continue to generate offspring. To counteract this effect we call the selector already when 2 individuals have finished evaluating their fitness, lifting the boundaries between generations and evaluating the performance of individuals. New offspring will be generated and processors can immediately go back to work on the next fitness evaluation.

Our variator implementation uses the master/slave architecture, presented in the next section, to run as many function evaluations as possible in parallel. Additionally various crossover and mutation policies are available for tuning the algorithm to the optimization problem.

THE FRAMEWORK

The fact that multi-objective optimization problems are omnipresent in research and industry calls for a general purpose framework. The following requirements are necessary to be able to apply the framework independently of field and optimization problem:

- supporting any multi-objective optimization method,
- supporting any forward solver (simulation code or measurements),
- general description/specification of objectives, constraints and design variables.

The reminder of this section discusses each requirement and show how this is implemented in the OPT-PILOT framework.

Every processor will take up one of three available roles (see Figure 1): one core will act as PILOT, the remaining cores are divided amongst WORKERS and OPTIMIZERS. As shown in Figure 4 the PILOT is used to coordinate all "information requests" between OPTIMIZER and WORKER. An information request job consists of a set of design variables (e.g. the genes of an individual) and a type of information it requests (e.g. function evaluation or derivative). The PILOT keeps checking for idle WORKERS and assigns jobs in the queue to any free WORKER groups. Once the WORKER has computed and evaluated the request it is routed back to the OPTIMIZER that originally requested the information.

Using template parameters the composition of the optimizer can be specified:

CODE LISTING 1: CALLING THE PILOT

```
typedef OpalInputFileParser Input_t;
typedef PisaVariator Opt_t;
typedef OpalSimulation Sim_t;
typedef Pilot< Input_t, Opt_t, Sim_t,
               /* ... */ > pilot_t;
scoped_ptr< pilot_t > pi(new pilot_t(args, comm));
```

After a processor gets appointed a role it starts a polling loop asynchronously listening for incoming requests. To that end a POLLER base class has been introduced. Classes implementing the POLLER interface enter an infinite loop and upon receiving MPI messages the appropriate handler is called. A concrete implementation of the POLLER is allowed to implement special methods acting as hooks in the polling process, e.g. for actions that need to be taken after a message has been handled. The core of the POLLER is the `onMessage()` method. The method is called with the `MPI_Status` of the received message and a `size_t` value specifying different values depending on the value of the `MPI_Tag`. Every POLLER terminates the loop upon receiving a special MPI tag.

Implementing an Optimizer

All OPTIMIZER implementations have to respect the API shown in Listing 2.

CODE LISTING 2: OPTIMIZER API

```
virtual void initialize() = 0;

// Poller hooks
virtual void setupPoll() = 0;
virtual void prePoll() = 0;
virtual void postPoll() = 0;
virtual void onStop() = 0;
virtual bool onMessage(MPI_Status status,
                       size_t length) = 0;
```

The optimizer entry point corresponds to the `initialize()` method. Since an optimizer derives from the `Poller` interface, predefined hooks can be used to influence the polling procedure. Using a special "external" communicator group (passed to the OPTIMIZER in the constructor) serves as interface to the PILOT to queue job requests. Every set of optimizers working on one optimization problem have their own "internal" MPI

ISBN 978-3-95450-116-8

communicator for handling the optimization part of the framework.

Implementing a Forward Solver

Forward solvers can easily be added by implementing a wrapper to run the simulation using a set of design variables. Listing 3 describes the API a forward solver has to implement.

CODE LISTING 3: SIMULATION API

```
virtual void run() = 0;
virtual void collectResults() = 0;
virtual reqVarContainer_t getResults() = 0;
```

The `run()` method will be called by workers and should execute the simulation in a *blocking* fashion. Subsequently, the worker will call `collectResults` to parse data from output files to the result data structures and ultimately use `getResults()` to get the resulting objectives and pass the data back to the optimizer. As before, every set of workers handling the same problem has access to a shared MPI communicator.

Optimization Problem Specification

Generally, any combination of simulation output variables (during any point of the simulation) can be used to compute objectives, constraints or act as design variables. Motivated by the principle of keeping meta-data (optimization and simulation input data) together, we decided to embed the optimization problem in the simulation input file (e.g. using comments). In some cases it might not be possible to annotate the simulation input file. By using another input file parser the optimization problems from stand-alone files.

Expression strings are parsed using Boost Spirit[1]. In the process an annotated expression tree is constructed and upon evaluation, all unknown variables are replaced with values from simulation results. To improve the expressive power of objectives and constraints we added a simple mechanism to define and call custom functions in expressions. This enables the user to define custom functions (e.g. over data produced by a simulation or measurement files) with ease using functors as shown in Listing 4.

CODE LISTING 4: SIMPLE AVERAGE FUNCTOR

```
struct avg {

    double operator()(
      client::function::arguments_t args) const {

        double limit = boost::get<double>(args[0]);
        std::string filename =
          boost::get<std::string>(args[1]);

        double sum = 0.0;
        for(int i = 0; i < limit; i++)
            sum += getDataFromFile(filename, i);

        return sum / limit;
    }
};
```

[1]http://boost-spirit.com/

All custom functions have to be registered with the expression to ensure the expression knows how to resolve function calls in its tree.

Parallelization

Parallelization is defined by a mapping of roles to available cores. Command-line options allow the user to steer the number of processors used in worker and optimizer groups. Here, we mainly use the command-line options to steer the number of processors running a forward solver.

FORWARD SOLVER

The framework contains a wrapper implementing the API mentioned in Listing 3 for OPAL [3]. OPAL provides different trackers for cyclotrons and linear accelerators with satisfactory parallel performance [4]. Recently we introduced a reduced envelope model [5] into OPAL reducing time to solution by several orders of magnitude.

Access to the OPAL forward solver enables the optimizer to solve a multitude of optimization problems arising in the domain of particle accelerators.

EXPERIMENTS

Experiments were executed on the PSI FELSIM cluster, running the framework using the components described above. The FELSIM cluster consists of 8 dual quad-core Intel Xeon processors at 3.0 GHz and has 2 GB memory per core with a total of 128 cores. The nodes are connected via Infiniband network with a total bandwidth of 16 GB/s.

A first benchmark tries to reproduce the Ferrario matching point [6] using the optimization problem given in equations (5) to (13).

$$
\begin{aligned}
\min\quad & \varepsilon_x && (5)\\
& \Delta\mathrm{rms}_{x,\mathrm{peak}} && (6)\\
& \Delta\varepsilon_{x,\mathrm{peak}} && (7)\\
\text{s.t.}\quad & q = 200\,[\mathrm{pC}] && (8)\\
& \mathrm{Volt}_{\mathrm{RF}} = 100\,[\mathrm{MV/m}] && (9)\\
& \sigma_L \le \sigma_x = \sigma_y \le \sigma_U && (10)\\
& \mathrm{KS}_L \le \mathrm{KS}_{\mathrm{RF}} \le \mathrm{KS}_U && (11)\\
& \mathrm{LAG}_L \le \mathrm{LAG}_{\mathrm{RF}} \le \mathrm{LAG}_U && (12)\\
& \Delta z_{L\mathrm{KS}} \le \Delta z_{\mathrm{KS}} \le \Delta z_{U\mathrm{KS}} && (13)\\
& && (14)
\end{aligned}
$$

The first objective minimizes the emittance at the end of the first traveling wave structure. The remaining two objectives minimize the distance from the position of the current minimum peak to the expected peak location at 3.025 m for transverse bunch size (beam waist) and emittance. Equations (8) and (9) define constraints for initial conditions and design variables given in (10) to (13) correspond to fieldstrength, and displacement of the solenoid.

ISBN 978-3-95450-116-8

Visualization

For visualization purposes we implemented a simple Pareto front explorer in Python. This simple tool provides a simple mapping from solutions on the Pareto front to the correct design variable values and helps investigate effects of the trade-off decision on the design variables.

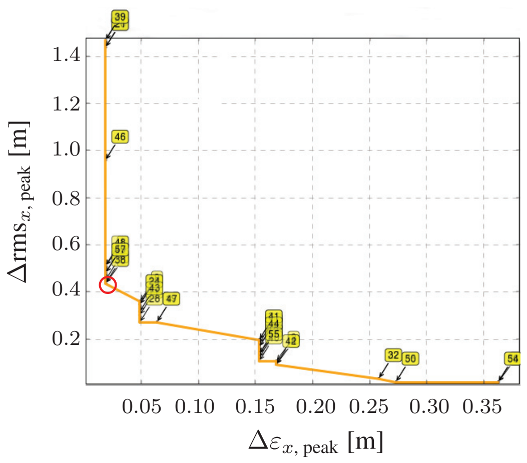


Figure 5: Approximation of Pareto front for the Ferrario matching point optimization problem. Red circle marks selected trade-off solution (38).

The result of a 1000 generation run (taking approximately 20 minutes with 16 cores) for the Ferrario matching problem mentioned above is show in Figure 5. For the trade-off solution 38 we show the complete time evolution of the slice beam in Figure 6. The minimum of beam waist and emittance peak are coinciding and this point falls close to the expected 3.025 m.

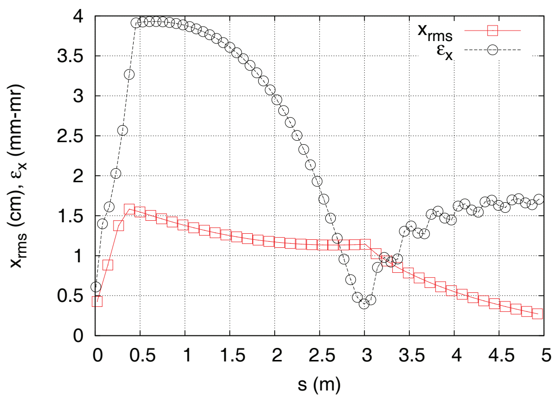


Figure 6: Simulation results for individual 38 of Pareto front shown in Figure 5.

CONCLUSIONS

We presented a framework to solve general multi-objective optimization framework. Its modularity allows to cover simulation-based optimization of a wide range of problems.

A first study on a simple benchmark shows that the framework is ready to tackle problems arising in the domain of beam dynamics. Even tough the stability analysis of the presented results is still work in progress the results are very convincing. With help of the master/slave parallelization the framework and using only a small amount of processors shows that a good approximation of the Pareto front can be computed in a matter of minutes.

Improving parallel efficiency on massively parallel systems is currently work in progress.

ACKNOWLEDGMENT

The authors thank the SwissFEL team for contributing to the formulation of optimization problems.

REFERENCES

[1] PISA — a platform and programming language independent interface for search algorithms. In Carlos M. Fonseca, Peter J. Fleming, Eckart Zitzler, Kalyanmoy Deb, and Lothar Thiele, editors, *Evolutionary Multi-Criterion Optimization (EMO 2003)*, Lecture Notes in Computer Science, pages 494 – 508, Berlin, 2003. Springer.

[2] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, April 2002.

[3] A. Adelmann, Ch. Kraus, Y. Ineichen, and J. J. Yang. The OPAL (Object Oriented Parallel Accelerator Library) Framework. Technical Report PSI-PR-08-02, Paul Scherrer Institut, 2008-2010. `http://amas.web.psi.ch/docs/opal/opal_user_guide.pdf`.

[4] A. Adelmann, Ch. Kraus, Y. Ineichen, S. Russell, Y. Bi, and J.J. Yang. The object oriented parallel accelerator library (opal), design, implementation and application. In *Proceedings ICAP09*, 2009.

[5] Yves Ineichen, Andreas Adelmann, Costas Bekas, Alessandro Curioni, and Peter Arbenz. A fast and scalable low dimensional solver for charged particle dynamics in large particle accelerators. *Computer Science - Research and Development*, pages 1–8, May 2012.

[6] M. Ferrario, J.E. Clendenin, D.T. Palmer, J.B. Rosenzweig, and L. Serafini. HOMDYN study for the LCLS RF photoinjector. pages 534–563, 2000.