DISS. ETH NO. 21114

TOWARD MASSIVELY PARALLEL MULTI-OBJECTIVE OPTIMIZATION WITH APPLICATION TO PARTICLE ACCELERATORS

A dissertation submitted to
ETH ZURICH
for the degree of
Doctor of Sciences

presented by
YVES INEICHEN,
Master of Science ETH in Computer Science, ETH Zürich
born on April 5th, 1982
citizen of Eglisau, Switzerland

accepted on the recommendation of
Prof. Dr. Peter Arbenz, examiner
Prof. Dr. Lothar Thiele, co-examiner
Dr. Andreas Adelmann, co-examiner
Dr. Costas Bekas, co-examiner

2013

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 successful design, commission, and operation of accelerator facilities is a non-trivial problem. Today, tuning machine parameters, i.e., 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 automate the process of reliably identifying optimal configurations of accelerators we propose to formulate the investigation for an optimal set of tuning parameters as a large-scale multi-objective design optimization problem.

This involves significant computer modeling using accelerator simulation codes such as PSI's OPAL (Object Oriented Parallel Accelerator Library) framework. Despite the fact that these codes are parallel, typical simulation parameters (e.g. a small number of macroparticles and mesh size) for individual runs, limit their scalability to several hundred or a few thousand processors. This represents a strong impediment in view of the petascale regime, therefore parallelization on multiple levels, e.g. running multiple parallel simulations in parallel, becomes a necessity. On the other hand, introducing a low-dimensional scalable model enables multi-resolution simulation runs.

We present a general-purpose framework for simulation-based multi-objective optimization methods that allows the automatic investigation of optimal sets of machine parameters. The implementation is based on a master/slave paradigm, employing several masters and groups of workers to prevent communication hot-spots at master processes. In addition, we exploit information about the underlying network topology when placing master processes and assigning roles. Solution states are exchanged between masters in a rumor routing fashion on a social network graph using one-sided communication.

Using evolutionary algorithms and OPAL simulations as optimizer and forward solver in our framework, we demonstrate the feasibility and scalability of our approach on real applications in the domain of particle accelerators.

Zusammenfassung

Neben ihrer fundamentalen Bedeutung innerhalb der theoretischen Grundlagenforschung besitzten Teilchenbeschleuniger eine gleichwertige Relevanz für die angwandten Wissenschaften. Das Design, die Inbetriebnahme sowie der permanente Betrieb eines Teilchenbeschleunigers stellen einen komplexen Problembereich dar. Die Feineinstellung der Parameter eines Teilchenbeschleunigers, wie zum Beispiel die Teilchenladung, die Emissionszeit sowie zahlreiche andere Parameter der Beamline Elemente, werden heutzutage meist durch manuelles Absuchen des Parameterraumes mit Hilfe von Simulationen erreicht. Dieser Vorgang ist oft sehr zeitraubend sowie fehleranfällig. In dieser Arbeit formulieren wir diesen Prozess als multikriterielle Design- Optimierungsprobleme, welche anschliessend mit Hilfe von multikriteriellen Optimierungsverfahren gelöst werden können. Mit diesem Ansatz schaffen wir eine automatisierte und zuverlässige Grundlage für das Bestimmen von optimalen Konfigurationen von Teilchenbeschleuniger.

Die Realisierung solcher Lösungsansätze bedingt die Verfügbarkeit von komplexen Rechenmodellen, wie beispielsweise die am PSI entwickelte Teilchensimulation OPAL (Object Oriented Parallel Accelerator Library). Trotz der schon vorhandenen Parallelisierung der Implementierung wird die Skalierung von typischen Simulationsparametern (zum Beispiel eine geringe Anzahl Makropartikel und eine kleine Gittergrösse) auf einige tausend Prozessoren limitiert. Dies ist ein grosses Hindernis im Zeitalter der Petascale-Rechenzentren und eine Parallelisierung auf mehreren Ebenen wird zur Notwendigkeit. Das Einführen von zusätzlichen und parallel effizienten Modellen verschiedener Detailauflösung erlaubt es uns im Laufe des Optimierungsprozesses, Teilchensimulationen mit verschiedener Auflösung einzusetzen.

In dieser Arbeit präsentieren wir die Implementierung eines universellen Software-frameworks für die automatisierte Suche nach optimalen Konfigurationen für Teilchenbeschleuniger. Diese Implementierung basiert auf dem Master/Slave Prinzip. Da die Netzwerkverbindungen zum Master-Knoten durch viele Nachrichten der zahlreichen Slaves saturiert werden, teilen wir die verfügbaren Knoten in Master/Slave-Gruppen ein. Für die Vergabe der Rollen nützen wir zusätzliche Informationen über den Netzwerkgraph für eine optimalere Verteilung. Um globale Synchronisationspunkte zu vermeiden, werden die Zustandsräume während des Optimierungsvorganges mittels rumor routing auf einem Graphen eines sozialen Netzwerks mit Hilfe von einseitiger Kommunikation propagiert.

Die Machbarkeit und Skalierung unseres Ansatzes, unter Verwendung eines evolutionären Optimierungsverfahrens und OPAL als Simulations-komponente, wird anhand von Beispielen aus der Teilchenphysik präsentiert.

Acknowledgments

First and foremost I would like to thank my supervisors Andreas Adelmann, Peter Arbenz and Costas Bekas for their continuous support and scientific advise. I am sure that this dissertation would not have progressed so well without their support, understanding and encouragement, I received when tackling my research problems.

For the stimulating and informative working environment I valued and enjoyed immensely, I thank the members of all the research groups I could be part of, in particular Alessandro Curioni, Cyril Flaig, Hua Goa, Christof Kraus, Stefan Pauli, Erhan Turan and Markus Wittberger for many interesting and helpful discussions on various topics.

For enriching my life and for moral support I would like to thank all the central nodes in my social network graph, most notably Simon, Andrea, Andy, Björn, Boris, Chrissy, Daniel, Daniela, Marc, Mark, Martin, Robi, Roby, Roman, Tom C. and Tom D.

Last but not least I would like to show my gratitude to my parents, Beat and Eva and my brother Christian for their rich and plentiful support over the years.

January 2013

Contents

1 Introduction 1
  1.1 Motivation . . . 1
  1.2 State-of-the-Art Multi-Objective Optimization Approaches . . 4
  1.3 Related Work . . . 8
  1.4 Contributions . . . 9
  1.5 Thesis Outline . . . 10

2 Multi-Objective Optimization 13
  2.1 Definition . . . 13
  2.2 Pareto Optimality . . . 14
  2.3 Solution Approaches: An Overview . . . 18
  2.4 Evolutionary Algorithms . . . 19
    2.4.1 The PISA Framework . . . 21
    2.4.2 Selector . . . 21
    2.4.3 Variator . . . 23
    2.4.4 Parallelization . . . 30
  2.5 Validation . . . 32

3 The Framework 35
  3.1 Related Work . . . 36
  3.2 Components and Plugins . . . 37
  3.3 Inter-Component Communication . . . 40
    3.3.1 Event-Driven MPI State Machine . . . 41
    3.3.2 Job Handling . . . 42
    3.3.3 Optimizer API . . . 43
    3.3.4 Simulation API . . . 44
  3.4 Optimization Problem Parser . . . 45
  3.5 Parallelization . . . 50
    3.5.1 Theoretical Network Model . . . 51
    3.5.2 Communicators . . . 55
    3.5.3 Solution State propagation . . . 56
  3.6 Visualization . . . 59

4 Forward Solvers for Beam Dynamics 63
  4.1 Short Introduction to Particle Accelerator Modeling . . . 64
    4.1.1 Objectives in Beam Dynamic Optimization Problems . 66

4.2 Iterative Space-Charge Solver 68
    4.2.1 The Discretization 70
    4.2.2 The Solution Method 72
    4.2.3 Implementation Details 76
    4.2.4 Numerical Experiments 77
    4.2.5 Improving Domain Decomposition 84
4.3 Envelope Tracker 86
    4.3.1 The Envelope Model 87
    4.3.2 Theoretical Complexity 91
    4.3.3 Parallelization 92

5 Results & Applications 95
5.1 Parallel Performance 95
    5.1.1 Iterative Space-Charge Solver 96
    5.1.2 Envelope Tracker 105
    5.1.3 The Master/Slave Framework 114
5.2 Application: Ferrario Matching 119

6 Conclusions 127

A The opt-pilot Framework 131
A.1 Building 131
    A.1.1 Homotopic Optimizer 132
    A.1.2 Configuring 132
A.2 Environment and Command Line Arguments 132
    A.2.1 The Drivers 134
    A.2.2 Installing a Selector 134
A.3 Documentation 134

B Visualization Tools 135
B.1 Python Visualization Script 135
    B.1.1 Installation 135
    B.1.2 Command Line Arguments 135
    B.1.3 Values of Design Variables 136
B.2 Pareto Explorer 136
    B.2.1 Installation 137
    B.2.2 File Format 137

Bibliography 139

Curriculum Vitae 139

Chapter

Introduction 1

In an age of widespread availablity of massive computational resources one might wonder, why solving huge and complex decision problems has not become more accessible and feasible. A major portion of these decision problems can be formulated as optimization problems with mutiple objectives, providing a mathematical description of the problem. Once formalized, we can apply one of the many possible algorithms to solve the resulting multi-objective optimization problem. Depending on the problem size and equally due to the exessive computational cost of the underlaying models, we fall short of an accessible and feasible solution. We believe that with the developement of an efficient and flexible massively parallel multi-objective optimization framework we take a major step towards rendering solving multi-objective optimization problems more accessible and feasible on large clusters. Here, we will turn our attention to decision problems arising in the domain of particle accelerator design and commissioning. The result of our work can be immediate utilized by machine operators and physicists. In combination with their experience, they can explore the impact of decisions to the design and the commissioning of these complex machines. In contrast to other related approaches, we envision a framework that is completely decoupled from the employed optimization approach and forward solver. Therefore, this adaptive charateristic will empower us to apply the framework directly to multi-objective optimization problems arising in different fields and deploy it efficiently on state of the art compatible machines.

1.1 Motivation

When we consider optimization problems arising in real world problems in engineering and research, we notice that a major portion shares the trait of possessing multiple objective functions. In general, where no additional *a priori* knowledge of the importance of the objective functions is available, we are forced to deal with all these objectives simultaneously. This applies for optimization problems related to particle accelerators likewise. For the most part, we require more than one criterion in order to describe particle bunch properties, subject to optimization, i.e., bunch emittance and matching a target energy. In this thesis, we target simulation-based parallel optimization

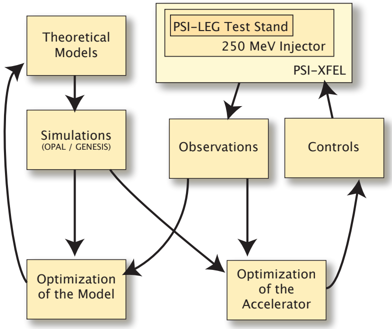


Figure 1.1: Model (Simulation), Measurement and Control. This thesis addresses the improvement of the model depicted in the left column.

techniques, to determine *an optimal set* of accelerator beam parameters, i.e., improve the theoretical model by taking into account observations (measurements) as illustrated in Figure 1.1. We emphasize that amongst various design goals we aim at a solution that can be readily deployed in an on-line control room setting.

In general, but in particle acceleration simulation particularly, the complexity of the underlaying models can be a decisive performance factor. In order to accurately simulate the low emittance electron beam, self-consistent models based on particle-in-cell methods have been successfully developed at PSI in the past. However, running on parallel computers, a single run can still take many hours on a medium number of processors (e.g. 64-256). Unfortunately, even implementing parallel efficient simulation codes is not the end of the story. Due to the nature of the optimization algorithms we are coerced to run thousands of simulations to explore a wide range of the parameter and design space. Depending on the level of detail, each simulation typically has between 1 and 100 million degrees of freedom. Clearly, without efficient massively parallel approaches, solving large multi-objective optimization problems will never be feasible. Additionally, scheduling, distributing and coordinating efforts between simulation runs and the optimizer becomes crucial for the performance and must be handled delicately to achieve an acceptable overall performance. In a nutshell, simulation-based optimization problems pose three main challenges:

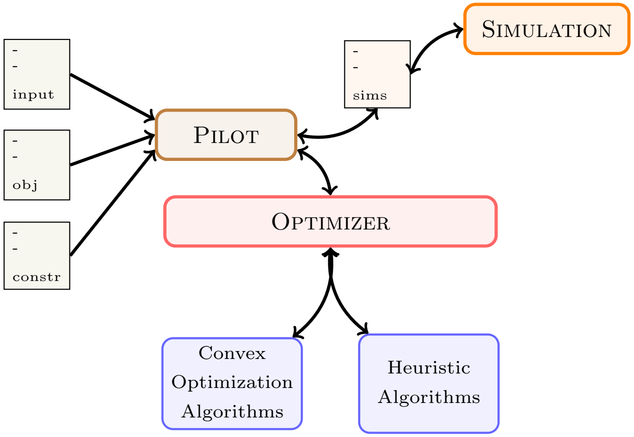


Figure 1.2: Multi-objective framework: the pilot (master) solves the optimization problem specified in the input file by coordinating optimizer algorithm and workers running forward solves.

1. computing cost functions in a parallel efficient manner,
2. applying non-linear constraints, and
3. efficient and accurate evaluation of the multi-objective function i.e. run the particle accelerator simulation.

We aimed at a flexible and modular framework. To achieve this, we divided the framework implementation in three components (see Figure 1.2). Each of the three components can be easily interchanged and benchmarked. However, in practice in many cases only the optimizer or the simulation component are replaced and the pilot seldom has to be adapted.

The core, denoted Pilot, provides a general interface for both the optimizer and the forward solver component and coordinates the computational effort. This is achieved by modeling a finite state machine with MPI point to point as well as one sided communication in order to provide a completely asynchronous execution environment, avoiding global collectives. We decided to use evolutionary algorithms (from the multifaceted choices shown in Figure 1.4) as a first concrete optimization strategy. A wrapper to OPAL serves as an obvious choice for the simulation component. In order to provide a more fine grained level of detail we introduced additional models.

Finally, we use annotated simulation input files to specify and connect the optimization problem with the simulation code. A parser, based on Boost Spirit[1], allows to specify complex expressions or to define custom functions in constraints and objectives.

Given the nature of these multi-objective optimization problems, we have to pre- and post-process large amounts of data. To that end, we introduced two simple yet effective ways to visualize and explore solutions.

Our framework will help accelerator scientists to explore a wide range of the operational space for optimal operation of current and future accelerators. We envision this tool to empower and guide accelerator scientists to render the design of next generation accelerators more efficiently by shortening the design cycle needed to determine optimal parameters. Similar methods, as the one we propose, have been shown to be successful in previous simulation-based optimization problems [**?**].

In this thesis, we will focus on PSI's SwissFEL [**?**] activities, which have an immediate and a long-range impact, by optimizing important parameters of PSI's 250 MeV injector (and later for the full PSI-SwissFEL facility). However, the methodology is much more general and will be *beneficial for all other particle accelerators* at PSI, and elsewhere. Moreover, by means of interchangeable components, the proposed multi-objective framework can be applied in other fields and other optimization algorithms.

1.2 State-of-the-Art Multi-Objective Optimization Approaches

In contrast to single-objective optimization, there is no precise concept of a global single solution in multi-objective optimization. We seek a set of points fulfilling a predefined description of optimality, the Pareto optimality criterion. This criterion states that a set of points is Pareto optimal if non of the points in the set can be further improved without leaving at least one other point worse off. Essentially, the set of Pareto optimal points describes an equilibrium state, where all points in the set are "equally" optimal in all objectives. This is known as the notion of Pareto efficiency or optimality, named after Vilfredo Pareto[2].

We illustrate this in Figure 1.3 with an example where the goal is to simultaneously minimize the price and maximize the performance. We are interested

---

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

[2] 5 July 1848 – 19 August 1923, an Italian engineer and economist who introduced the concept of *Pareto efficiency*. While first applied in economics the notion of Pareto efficiency was later applied to engineering.

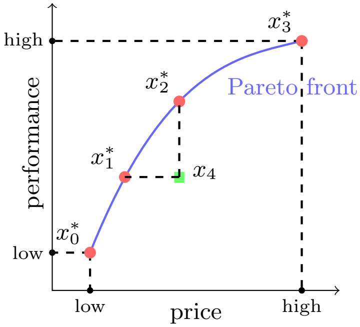
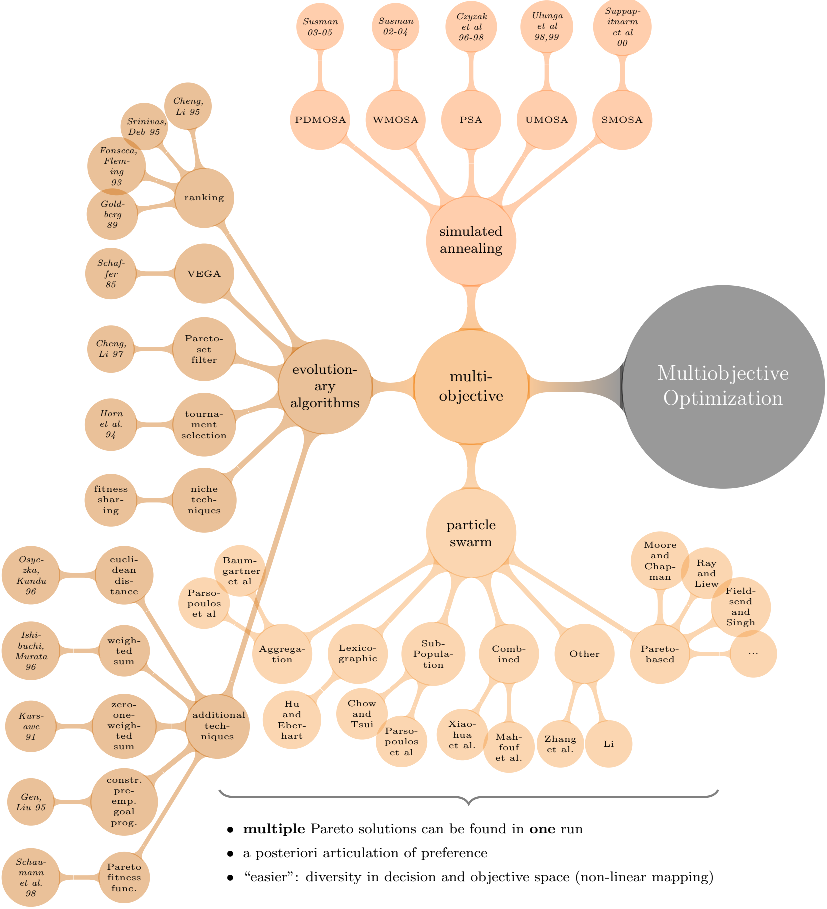


Figure 1.3: Two competing objectives can not be optimal at the same time. Red points represent Pareto optimal points, while x_4 is dominated (exhibits a worse price per performance ratio than e.g. x_2^*) by all points on the blue curve (Pareto front).

in the four red trade-off points (denoted by x_i^*) that are, in a Pareto optimal fashion, "equally" optimal in both objectives. The green square x_4 can still be improved without derogating any of the red points and therefore cannot be part of the Pareto optimal set. In contrast, the red points cannot be improve without hurting the optimality criteria of at least one other solution forming the sought after Pareto optimal set.

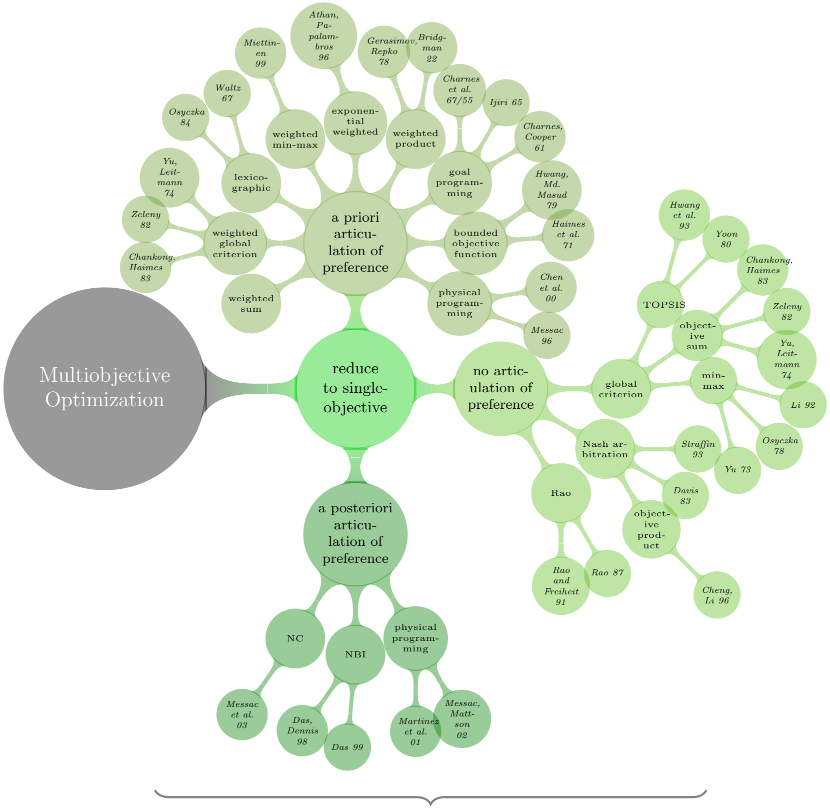
Multi-objective optimization methods are concerned with maintaining valid Pareto sets throughout the entire optimization process. For an overview on multi-objective optimization methods, surveys [**?**, **?**, **?**, **?**, **?**] are summarized in Figure 1.4 and Figure 1.5.

Prior to the discovery of algorithms that could handle more than one objective simultaneously, multi-objective optimization problems were tackled by specifying a preference vector corresponding to the weights for individual objectives (green in Figure 1.4). With the help of some *a priori* specified weights the objectives could be combined in a single objective and solved with standard techniques. However, in real world problems, knowledge about preference might not be available prior to solving the optimization problem and with this achieving and understanding of how our preference affects the optimal solutions. This motivated a search for novel algorithms, able to handle multiple objectives simultaneously, without reducing the problem to a single-objective optimization problem. As a result, many, mostly stochastic, methods were proposed (orange and brown in Figure 1.4). Typically, this methods are fast



- **multiple** Pareto solutions can be found in **one** run
- a posteriori articulation of preference
- “easier”: diversity in decision and objective space (non-linear mapping)

Figure 1.4: Multi-objective state of the art paper survey.



- only **one** Pareto solution can be found in **one** run
- preference-based (specify preference for trade-off solution)
- not all can be found in non-convex MOOPS
- all algorithms require a prior knowledge (weights, ε, targets)

Figure 1.5: Reducing multi-objective to single-objective state of the art paper survey.

and relatively easy to implement and parallelize but it is a delicate process to tune the model parameters. Most prominently there is no guarantee that the generated solution is indeed a global optimum. For most of the heuristic methods we can articulate preference of the objectives *a posteriori*.

In 1983 Kirkpatrick et al. proposed simulated annealing [**?**]. This optimization strategy is motivated by the annealing process of metal where alternating heating to force atoms out of local minima and cooling phases to settle atoms in more stable global minima again. Analogous the optimization approach uses a temperature parameter to control "step size" of random movements through the search space. In every iteration a new candidate solution is chosen from some distribution and is accepted with a probability depending on the current temperature and the old solution. By choosing the temperature to decrease slowly, the algorithm is likely to accept big moves in the beginning (temperature is high) and as the system cools down only solutions close to the old solution will be accepted. This algorithm is often used when the set of states is discrete, as in combinatorial multi-objective optimization problems.

A derivative-free particle swarm optimization approach was proposed by Kennedy et al. [**?**] in 1995. It has its roots in swarm intelligent behavior, like the flocking behavior of birds. Like for example in evolutionary algorithms, a population of individuals is used to sample the search space. In particle swarm methods each individual has a velocity and a position in the search space. Iteratively the individuals venture in the direction of its velocity and updates its position accordingly. The new velocity is determined by incorporating information about the individuals best known position and the best position of individuals in its neighborhood (or the swarms best known position).

Numerous other approaches, such as multi-objective ant colony optimization [**?**], have been proposed in the early nineties. Interestingly, almost all multi-objective meta-heuristic algorithms model stochastic processes found in nature [**?**, **?**, **?**].

1.3 Related Work

In the Domain of Particle Accelerators

A group at the Laboratory for Elementary Particle Physics at Cornell University has conducted a multi-objective computational optimization of a high brightness dc photoinjector[3] [**?**] (the very first section of the SwissFEL facility). Using evolutionary algorithms combined with parallel computing resources, the multivariate parameter space of the photoinjector was explored

---

[3] photocathode gun emitting an electron beam

for optimal values such as emittance and acceleration gradient. This computational tool allows an extensive study of complex and nonlinear systems such as the space-charge dominated regions of an accelerator, and has broad areas of potential application to accelerator physics and engineering problems. The most relevant physical effect that they demonstrated is the optimization of an injector based on drift or velocity bunching where the beam is emitted from a photoemission cathode in a dc gun. This is relevant to our base design for the SwissFEL facility in order to achieve a very high brightness beam with a compact (short) accelerator. Motivated by this work a number of papers [**?**, **?**, **?**] were published, making use of evolutionary algorithms for multi-objective optimization problems in design of particle accelerators. Besides discussing employed strategies trade-offs are analyzed and discussed.

The authors in [**?**] use a continuous adjoint method to solve an electromagnetic shape optimization problem where the actual optimization is performed by a sequential quadratic programming algorithm. The shape optimization problem resembles our problem with the major difference that it involves an eigenvalue problem. In case of handling objectives by combing them into one, a similar approach could be used to optimize our problem. Depending on the final optimization problem we can consider using sequential convex programming (or other more favorable approaches).

Multi-Objective Optimization Frameworks

In contrast to similar frameworks (see Chapter 3 for details), we aimed at a more flexible and modular framework. Instead of pairing the framework to a fixed optimization algorithm, we implemented a more general approach, providing a user-friendly way to introduce new or use existing built-in multi-objective optimization algorithms.

1.4 Contributions

In this thesis we are trying to answer the question for the feasibility of a massively parallel multi-objective optimization framework. Even though these huge problems might seem distressing complex and expensive at a first glance, a closer look reveals that the right combination of techniques reduce the computational complexity and render them practicable. Moreover, most of the nifty details can be carefully hidden away and the scientist can apply the framework as a black box.

In order to be able to handle such large multi-objective optimization problems, we require highly scalable algorithms for both, the simulation and the optimizer component. To that end, we based the implementation of our framework on the master/slave parallelization paradigm: a master generates and distributes computational tasks to slaves that in turn compute the requested information and return the result back to the master. Here, we use masters to coordinate the efforts between the optimizer and simulation component. To avoid common performance pitfalls of master/slave implementations, e.g. too many slaves per master inducing a performance bottleneck at a master, we implemented special parallelization strategies (see Section 3.5) splitting the available computational power into multiple small master/slave groups.

In collaboration with the machine physicists, we applied the novel method to optimize beam dynamic problems arising in the design and commissioning of the 250 MeV INJECTOR, i.e., find new and better working points improving the performance of the 250 MeV INJECTOR with respect to some measure.

By means of the following contributions we show a feasible and efficient approach for solving multi-objective optimization problems:

- a novel general purpose parallel framework for solving multi-objective optimization problems (Chapter 3),
- scalable simulation (Chapter 4) and optimization components (Chapter 2),
- application to a real particle accelerator 250 MeV INJECTOR(Chapter 5), and
- broad range of applicability of the developed methods.

We consider this a first step towards a predictive on-line model, running simultaneously with the machine.

1.5 Thesis Outline

This thesis is organized as follows. We start with a short introduction to multi-objective optimization in Chapter 2 and discuss the implementation details of our evolutionary algorithm.

Chapter 3 addresses the issue of the interplay and parallelization of the framework components. We will take a closer look at the protocols of the MPI finite state machine and how the network topology is employed to improve parallel efficiency.

Concrete implementations of accelerator physics related simulation codes are discussed in detail in Chapter 4. Aside from a short introduction to particle accelerator modeling we present two new models: a fast low dimensional solver and an iterative space charge solver for the macro particle tracker (OPAL-T).

We present parallel performance measurements in Chapter 5 and apply the framework in a real world application.

Finally we draw our conclusions in Chapter 6 and provide a vision of further work.

Multi-Objective Optimization

2

> *"If you optimize everything, you will always be unhappy."*
> – Donald Knuth

Optimization methods deal with finding a feasible set of solutions corresponding to extreme values of some specific criteria. Problems consisting of more than one criterion are called *multi-objective optimization problems*. Multiple objectives arise naturally in many real world optimization problems, such as portfolio optimization, design, planning and many more [**?**, **?**, **?**, **?**, **?**]. It is important to stress that multi-objective problems are in general harder and more expensive to solve than single-objective optimization problems.

In this chapter we introduce multi-objective optimization problems and discuss techniques for their solution with an emphasis on evolutionary algorithms.

2.1 Definition

As with single-objective optimization problems, multi-object optimization problems consist of a solution vector and optionally a number of equality and inequality constraints. Formally, a general multi-objective optimization problem has the form

$$
\begin{aligned}
\text{min} \quad & f_m(\mathbf{x}), & m = \{1, \ldots, M\} \qquad (2.1)\\
\text{s.t.} \quad & g_j(\mathbf{x}) \geq 0, & j = \{1, \ldots, J\} \qquad (2.2)\\
& -\infty \leq x_i^L \leq \mathbf{x} = x_i \leq x_i^U \leq \infty, & i = \{0, \ldots, n\}. \qquad (2.3)
\end{aligned}
$$

The M objectives (2.1) are minimized, subject to J inequality constraints (2.2). An n-vector (2.3) contains all the design variables with appropriate lower and upper bounds, constraining the design space.

In contrast to single-objective optimization the objective functions span a multi-dimensional space in addition to the design variable space – for each point in design space there exists a point in objective space. The mapping from the n dimensional design space to the M dimensional objective space visualized in Figure 2.1 is often non-linear. This impedes the search for optimal solutions

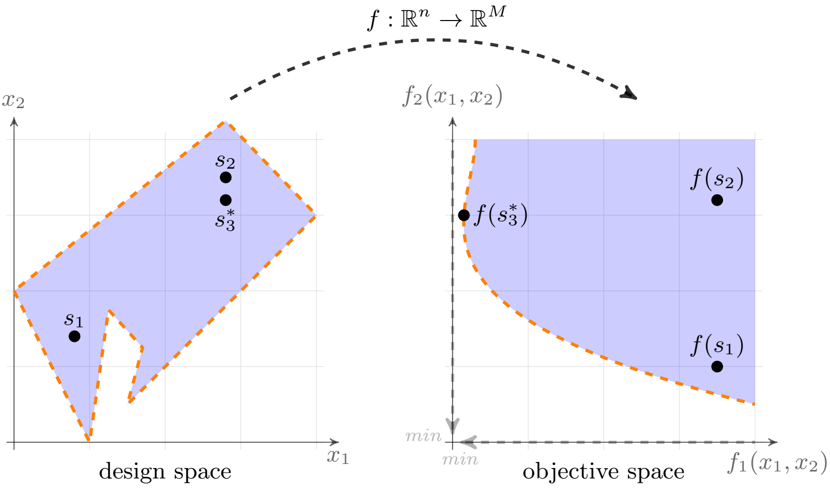


Figure 2.1: The (often non-linear) mapping $f : \mathbb{R}^n \to \mathbb{R}^M$ from design to objective space. The dashed lines represent the constraints in design space.

and increases the computational cost as a result of expensive objective function evaluation. Additionally, depending in which of the two spaces the algorithm uses to determine the next step, it can be difficult to assure an even sampling of both spaces simultaneously.

A special subset of multi-objective optimization problems where all objectives and constraints are linear, called *Multi-objective linear programs*, exhibit formidable theoretical properties that facilitate convergence proofs. In this thesis we strive to address arbitrary multi-objective optimization problems with non-linear constraints and objectives. No general convergence proofs are readily available for these cases.

2.2 Pareto Optimality

In most multi-objective optimization problems we have to deal with conflicting objectives. Two objectives are conflicting if they possess different minima. If all the mimima of all objectives coincide the multi-objective optimization problem has only one solution. To facilitate comparing solutions we define a partial ordering relation on candidate solutions based on the concept of dominance. A solution is said to dominate another solution if it is no worse than the other solution in all objectives and if it is strictly better in at least one objective. A more formal description of the dominance relation is given in Definition 1 [?].

Definition 1. *A point* $\mathbf{x}_1$ *is dominating* $\mathbf{x}_2$*, if both properties*

- $f_m(\mathbf{x}_1) \geq f_m(\mathbf{x}_2),\ \forall m \in \{1, \ldots, M\}$
- $f_m(\mathbf{x}_1) > f_m(\mathbf{x}_2),\ \exists m \in \{1, \ldots, M\}$

hold. We denote this as $\mathbf{x}_1 \preceq \mathbf{x}_2$*.*

The properties of the dominance relation include transitivity

$$x_1 \preceq x_2 \wedge x_2 \preceq x_3 \Rightarrow x_1 \preceq x_3,$$

and asymmetricity, which is necessary for an unambiguous order relation

$$x_1 \preceq x_2 \Rightarrow x_2 \npreceq x_1.$$

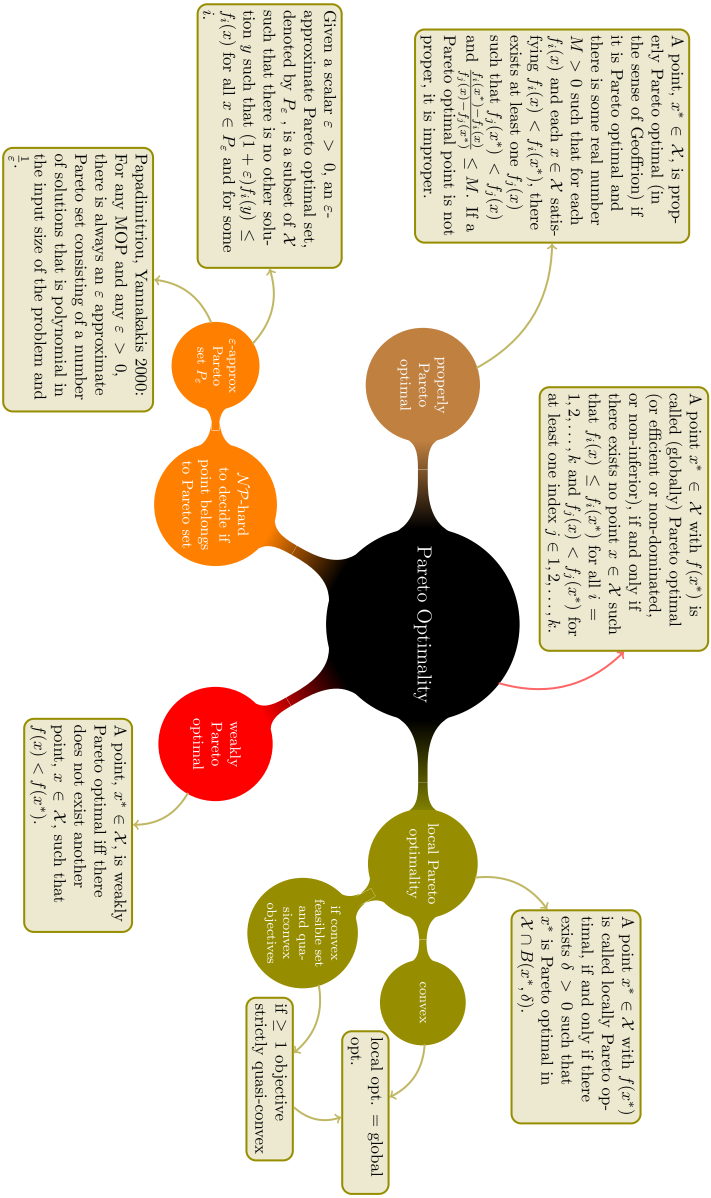
Using the concept of dominance, the sought-after set of Pareto optimal solution points can be approximated iteratively as the set of non-dominated solutions.

The problem of deciding if a point truly belongs to the Pareto set is NP-hard. As shown in Figure 2.2 there exist “weaker” formulations of Pareto optimality. Of special interest is the result shown in [?], where the authors present a polynomial (in the input size of the problem and $1/\varepsilon$) algorithm for finding an approximation, with accuracy ε, of the Pareto set for database queries.

An Illustrative 2 Objective Example

To illustrate the concept of dominance, let us consider the decision problem of buying a new car. When facing the decision of buying a new car, a diverse set of choices has to be made. Most of these choices correspond to subjective trade-off decisions. For the sake of simplicity we focus on only two criteria: price and performance. In this context, performance can represent different properties, e.g., horsepower or distance traveled with x liter of gas. Understandably, we want to get the best performance at a cheap price. This can be formulated as the multi-objective optimization problem shown in (2.4):

$$\begin{array}{ll} \min & [\text{price}, -\text{performance}]^T \\ \text{s.t.} & \text{low}_{\text{pr}} \leq \text{price} \leq \text{high}_{\text{pr}} \\ & \text{low}_{\text{pe}} \leq \text{performance} \leq \text{high}_{\text{pe}} \end{array} \tag{2.4}$$


Pareto Optimality
properly Pareto optimal
$\mathcal{NP}$-hard to decide if point belongs to Pareto set
ε-approx Pareto set P_ε
weakly Pareto optimal
local Pareto optimality
if convex feasible set and quasiconvex objectives
convex

A point $x^* \in \mathcal{X}$ with $f(x^*)$ is called (globally) Pareto optimal (or efficient or non-dominated, or non-inferior), if and only if there exists no point $x \in \mathcal{X}$ such that $f_i(x) \leq f_i(x^*)$ for all $i = 1, 2, \ldots, k$ and $f_j(x) < f_j(x^*)$ for at least one index $j \in 1, 2, \ldots, k$.

A point, $x^* \in \mathcal{X}$, is properly Pareto optimal (in the sense of Geoffrion) if it is Pareto optimal and there is some real number $M > 0$ such that for each $f_i(x)$ and each $x \in \mathcal{X}$ satisfying $f_i(x) < f_i(x^*)$, there exists at least one $f_j(x)$ such that $f_j(x^*) < f_j(x)$ and $\frac{f_i(x^*)-f_i(x)}{f_j(x)-f_j(x^*)} \leq M$. If a Pareto optimal point is not proper, it is improper.

Given a scalar $\varepsilon > 0$, an ε-approximate Pareto optimal set, denoted by P_ε, is a subset of $\mathcal{X}$ such that there is no other solution y such that $(1+\varepsilon)f_i(y) \leq f_i(x)$ for all $x \in P_\varepsilon$ and for some i.

Papadimitriou, Yannakakis 2000: For any MOP and any $\varepsilon > 0$, there is always an ε approximate Pareto set consisting of a number of solutions that is polynomial in the input size of the problem and $\frac{1}{\varepsilon}$.

A point, $x^* \in \mathcal{X}$, is weakly Pareto optimal iff there does not exist another point, $x \in \mathcal{X}$, such that $f(x) < f(x^*)$.

A point $x^* \in \mathcal{X}$ with $f(x^*)$ is called locally Pareto optimal, if and only if there exists $\delta > 0$ such that x^* is Pareto optimal in $\mathcal{X} \cap B(x^*, \delta)$.

if ≥ 1 objective strictly quasi-convex

local opt. = global opt.

Figure 2.2: Various definitions regarding Pareto optimality.

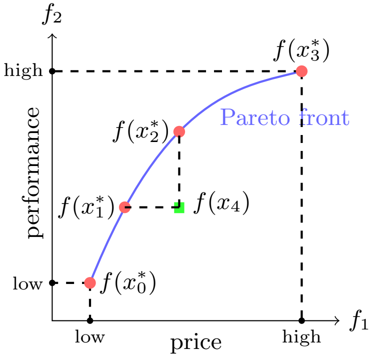


Figure 2.3: Two competing objectives can not be optimal at the same time. Red points represent Pareto optimal points, while x_4 is dominated (exhibits a worse price per performance ratio than x_1^* and x_2^*). The Pareto front is illustrated by the blue curve.

In general, it is not possible to satisfy both objectives at the same time and a trade-off decision between performance and price has to be reached (see Figure 2.3). Since not every choice is equally profitable, we can use Definition 1 to sort all candidate solutions according to dominance. For instance the car x_4 is clearly dominated by x_2^* (same price for less performance). We are interested in the red trade-offs points that are essentially “equally” optimal in both objectives. Using Definition 1 the set of all optimal points (blue curve) corresponds to the set of all non-dominated solutions, called Pareto front or surface. In other words we cannot improve any point on the Pareto front without hurting at least one other solution. This is known as Pareto efficiency or optimality, named after Vilfredo Pareto (cf. Page 4).

Once the shape of the Pareto front has been determined we can *specify preference*, balancing the features by observing the effect on the optimality criteria, converging to the preferred solution. This is called *a posteriori* preference specification since we select a solution after all possible trade-offs have been presented to us. In contrast, for *a priori* preference specification we have to have a precise idea of the importance of each objective before we can combine all objectives into a single objective optimization problem and solve that. In many situations preference is not known *a priori* and exploring the trade-offs on the Pareto front helps to convey a deeper understanding of the solution space and the impact of trade-off decisions becomes observable.

Unfortunately, gathering a representative set of non-dominated points approximating a Pareto front is far from trivial. This is due to various factors: no gradient information, high dimensional search spaces, the non-linear mapping and infeasible regions inside the search space. A number of different approaches have been proposed, e.g. evolutionary algorithms, sampling, simulated annealing, swarm methods and many more. The next section presents a brief overview of the various approaches.

2.3 Solution Approaches: An Overview

Prior to the discovery of algorithms that could handle more than one objective simultaneously, multi-objective optimization problems were tackled by specifying a preference vector corresponding to weights for individual objectives. With the help of some *a priori* specified weights the objectives could be combined in a single objective and solved with standard techniques. However, in real world problems, knowledge about preference might not be available prior to solving the problem. This motivated a search for novel algorithms, able to handle multiple objectives simultaneously, without reducing the problem to a single-objective optimization problem. As a result, many, mostly stochastic, methods were proposed (see Figure 1.4).

In 1983 Kirkpatrick et al. proposed simulated annealing [**?**]. This optimization strategy is motivated by the annealing process of metal where alternating heating to force atoms out of local minima and cooling phases to settle atoms in more stable global minima again. Analogously the optimization approach uses a temperature parameter to control “step size” of random movements through the search space. In every iteration a new candidate solution is chosen from some distribution and is accepted with a probability depending on the current temperature and the old solution. By choosing the temperature to decrease slowly, the algorithm is likely to accept big moves in the beginning (temperature is high) and as the system cools down only solutions close to the old solution will be accepted. This algorithm is often used when the set of states is discrete, as in combinatorial multi-objective optimization problems.

A derivative-free particle swarm optimization approach was proposed by Kennedy et al. [**?**] in 1995. It has its roots in swarm intelligent behavior, like the flocking behavior of birds. Like for example in evolutionary algorithms, a population of individuals is used to sample the search space. In particle swarm methods each individual has a velocity and a position in the search space. Iteratively the individuals venture in the direction of its velocity and updates its position accordingly. The new velocity is determined by incorporating information about the individuals best known position and the best position of individuals in its neighborhood (or the swarms best known position).

Numerous other approaches, such as multi-objective ant colony optimization [**?**], have been proposed in the early nineties. Interestingly, almost all multi-objective meta-heuristic algorithms model stochastic processes found in nature [**?**, **?**, **?**].

In the next section we will take a closer look at another derivative-free algorithm mimicking nature's famous evolutionary pressure ("survival of the fittest") to solve multi-objective problems. For a more profound coverage to evolutionary algorithms we refer to [**?**].

2.4 Evolutionary Algorithms

Evolutionary algorithms (EA) are loosely based on nature's evolutionary principles to guide a population of individuals towards an improved solution by honoring the "survival of the fittest" practice. This "simulated" evolutionary process preserves entropy (or diversity in biological terms) by applying genetic operators, such as mutation and crossover, to remix the fittest individuals in a population. Maintaining diversity is a crucial feature for the success of all evolutionary algorithms.

In general, a generic evolutionary algorithm consists of the following components:

- *Genes*: traits defining an individual,
- *Fitness*: a mapping from genes to a set of numeric values describing the fitness of an individual,
- *Selector*: selecting the k fittest individuals of a population based on some sort of ordering,
- *Variator*: recombination (mutations and crossover) operators for offspring generation.

Applied to multi-objective optimization problems, genes correspond to design variables. The fitness of an individual is computed by evaluating each objective function for the gene set of that individual. Figure 2.4 schematically depicts the connection of the components introduced above. The process starts with an initially random population of individuals I_i, each individual with a unique set of genes and corresponding fitness, representing one location in the search space. In a next step the population is processed by the SELECTOR determining the k fittest individuals. While the k fittest individuals are

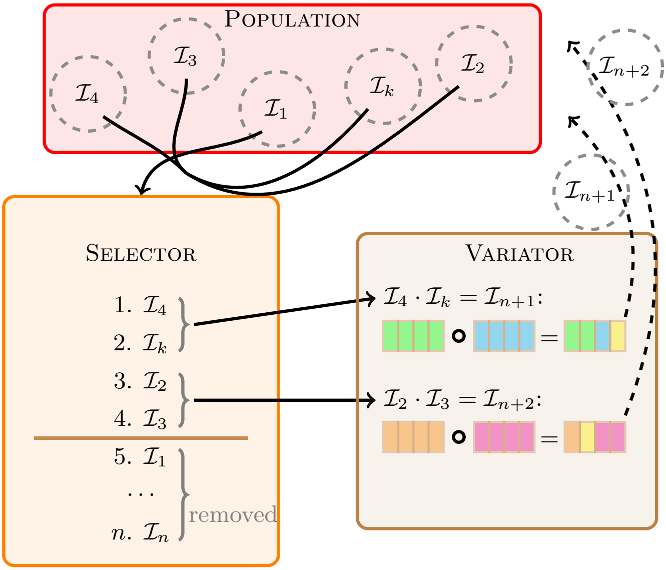


Figure 2.4: Schematic view of interplay between selector and variator. The selector ranks all individuals in the population according to fitness and subsequently the variator creates offspring using the genes of the fittest parents. Finally the new children are reintroduced in the population.

passed to the Variator, the remaining $n - k$ individuals are eliminated from the population. The Variator mates the k fittest individuals to generate new offsprings and applies the recombination operators. After evaluating the fitness of all the freshly born individuals a *generation* cycle has completed and the process can start anew.

The choice, to start with an implementation of an evolutionary algorithm as optimizer algorithm, was mainly motivated by good results [**?**, **?**] attained in applications in the domain of accelerator physics.

2.4.1 The PISA Framework

Since there already exist plenty of good implementations of evolutionary algorithms, we benefit here from the PISA library [**?**]. One of the distinct advantages of PISA is its approach to separate the variator from the selector, rendering the library easy to expand and configure. This is achieved by implementing the library as a finite state machine, as shown in Figure 2.5, where each component is in charge of its own states. With this isolation and a precise definition of a "communication protocol" both sides can be exchanged and implemented independently. The PISA library uses files to communicate between the variator and selector.

In the rest of this chapter we briefly introduce one of the selectors available in PISA, namely the Nsga-II selector, and cover our implementation of the variator.

2.4.2 Selector

Selection algorithms can be subdivided into two categories: either they are *elite-preserving* or not. In elite-preserving algorithms elite individuals (a fixed number of the top individuals with the highest fitness value) have a chance to be directly carried over into the next generation. To what degree elitism should be considered is vague but its presence plays an important role in preventing the fitness of the population from deteriorating. When parents cannot be carried over to the next generation it is possible that the recombination and the mutation operators generation only inferior offspring and all the good parent solutions are "lost". Therefore it is beneficial to have at least a simple mechanism to ensure that some of the most elite individuals are carried over into the next generation. In most elitism conserving approaches the newly generated offspring is compared with the parents and keep only the fitter solution. Aside from preventing fitness deterioration the selection algorithm has to be careful not to allow a buildup of niches. Niching strategies counteract this effect to some degree by breaking ties between two equally fit individuals.

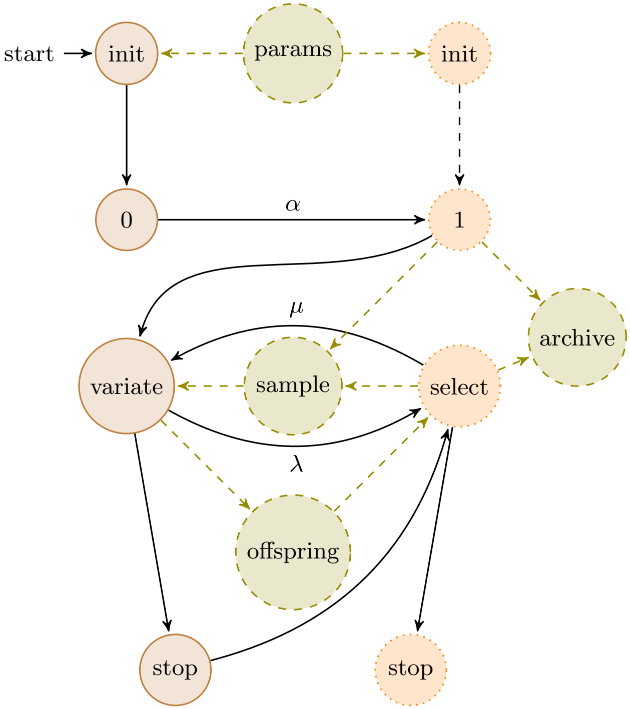


Figure 2.5: PISA finite state machine: Solid arrows represent state changes, dashed arrows show data flow and brown (left side) shows the variator states, selector states are dotted and orange (right side). The size of the initial population is denoted by α. The selector selects the μ fittest individuals and the variator passes λ freshly evaluated individuals to the selector.

Most commonly this is done by computing the density of equally fit solutions in an area around each of the two individuals under consideration and rejecting the better represented solution.

Unfortunately there is no “globally” best suited selection algorithm, in fact it has been shown [**?**] that the performance of a selector depends on the number of objectives and the surface of the search space. In all our tests and real world applications we merely used the NSGA-II selector [**?**]. So far it proved to deliver satisfactory convergence performance. In the remainder of this section we illustrate this specific selector in more detail.

The non-dominated sorting genetic algorithm (NSGA-II) preserves elitism by combining offspring and parent individuals in a new set, twice the size of the initial population N, and sorts this set according to different levels of dominance (see Definition 1). In a next step the set is reduced to the population size N by adding each dominance frontier set iteratively, beginning with the set of non-dominated individuals, continuing with individuals dominated by one other individual and so on. If the algorithm has to break ties between two equally fit individuals it consults a crowding distance measure to prevent niches.

In order to efficiently compute the dominance frontiers, as shown in Algorithm 1 on lines 2 – 14, an initialization phase helps building the required auxiliary data structures. This is done by checking the dominance relation of each individual with all the other individuals in the population. In the second phase (lines 15 – 23) the auxiliary data structure is used to build dominance fronts $\mathcal{F}_k$, where the first front contains all non-dominated individuals. The computation of all these dominance fronts $\mathcal{F}_k$ takes a total of $\mathcal{O}(MN^2)$ time, where M denotes the number of genes and N the population size.

The crowding distance used to break ties is computed by the side-length of the cuboid spanned by the closest left and right neighbor solution (I_{i-1} and I_{i+1}) relative to maximal and minimal value of the values for each objective. We visualize this for $i = 1$ in Figure 2.6. In order to compute neighbors, maximal and minimal value efficiently the fitness values are sorted with respect to their m-th objective value. Since the side-length is computed for all M dimensions of the multi-objective problem we get a weighted average of the cube side-length in all dimensions. Therefore the running time of these two algorithms is at most $\mathcal{O}(MN \log N)$ for sorting M arrays. The total time to compute this non-dominated sorting NSGA-II therefore amounts to $\mathcal{O}(MN^2)$.

2.4.3 Variator

The central tasks of a variator is to apply gene recombination operators and evaluate the fitness of the individuals in the population. This is reflected in

Algorithm 1 NSGA-II computation for a population $\mathcal{P}$ to assign each individual to its proper k-th front $\mathcal{F}_k$.

```
procedure NonDominatedSorting($\mathcal{P}$)
    for all $p \in \mathcal{P}$ do
        $\mathcal{S}_p \leftarrow \emptyset$                    ▷ holds all solutions $p$ dominates
        $n_p \leftarrow 0$          ▷ counter for number of solutions dominating $p$

        for all $j \subset \mathcal{P}$ do
            if $j \preceq p$ then
                $n_p \leftarrow n_p + 1$
            end if
            if $p \preceq j$ then
                $\mathcal{S}_p \leftarrow \mathcal{S}_p \cup j$
            end if
        end for
    end for
    $k \leftarrow 0$

    while $\exists$ solution $p$ with $n_p \geq 0$ do
        for all $p$ with $n_p = 0$ do
            $\mathcal{F}_k \leftarrow \mathcal{F}_k \cup p$                          ▷ $p$ in $k$-th front
            $n_p \leftarrow -1$
            for all $j \subset \mathcal{S}_p$ do
                $n_j \leftarrow n_j - 1$
            end for
        end for
        $k \leftarrow k + 1$
    end while
end procedure
```

the implementation of our variator sketched in Algorithm 2.

First, the set of selected individuals is cloned. In a subsequent step the crossover operator is applied pairwise, followed by the mutation operator. Finally, the fitness of each individual is evaluated.

The concrete implementation of our variator expects template policies for the two major operations: mutation and recombination (see Listing 1). This design allows the user to provide small functors[1] implementing their own algorithms for these operations or choose any of the library provided operators

---

[1] a function object

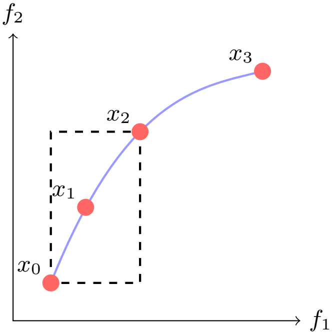


Figure 2.6: The crowding distance metric for solution x_1 is the area of the cuboid spanned between x_0 and x_2 relative to the maximal and the minimal value for each objective f_1 and f_2.

Algorithm 2 Variator implementation for a set of selected individuals $\mathcal{F}$ (new parents) $I_1, \ldots, I_k$.

```
procedure VARIATE($\mathcal{F}$)
    $\mathcal{F}' \leftarrow \emptyset$
    for all p doarent pairs $(I_i, I_{i+1}) \in \mathcal{F}$
        recombination($I_i$, $I_{i+1}$)
        mutate($I_i$)
        mutate($I_{i+1}$)
        $\mathcal{F}' \leftarrow \mathcal{F}' \cup I_i$
        $\mathcal{F}' \leftarrow \mathcal{F}' \cup I_{i+1}$
    end for

    for all $I_i \in \mathcal{F}'$ do
        evaluate($I_i$)
    end for
end procedure
```

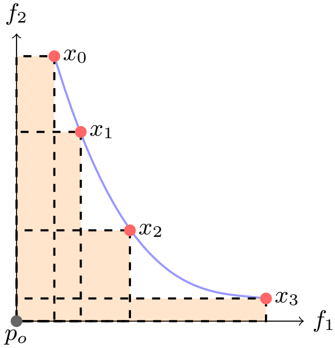


Figure 2.7: The hypervolume for a two-objective optimization problem corresponds to the accumulated area of all dashed rectangles spanned by all points in the Pareto set and arbitrary origin p_o.

as discussed in the following. The discussion of evaluating the fitness of an individual is deferred to the next chapter.

CODE LISTING 1: TEMPLATE POLICIES FOR VARIATOR

```
template<
       class T
     , template <class> class CrossoverPolicy
     , template <class> class MutationPolicy
>
class Variator {
  /* ... */
};
```

Convergence can be determined with respect to various measures, e. g. the fitness threshold (best fitness value) or smoothness of front. Naturally, convergence can also be measured with respect to the gene values (average of genes) or the population fitness (average of population). Instead of averaging values, we can compute the hypervolume of the dominated subset in objective space (relative to a point) as a measure of the populations performance. Given a point in the Pareto set, we compute the m dimensional volume (for m objectives) of the dominated space, relative a chosen origin. We visualize this for 2 objectives in Figure 2.7. Unfortunately, we cannot compute the hypervolume exactly due to its NP-complete nature, but fast approximation approaches exist [**?**]. To that end, we use the hypervolume approximation routines [**?**], implemented in the **`wfgHypervolume`** library[2], in our variator to feature the following three convergence criteria:

- max generations: "converge" once a maximal number of generations is reached,
- hypervolume value: converge once the hypervolume gets ε-close to the specified value, and
- hypervolume change: converge once the hypervolume reduces by ε compared to the hypervolume of the previous generation.

Recombination Operators

To ensure diversity in the population evolutionary algorithms make use of mutation and crossover operators. Algorithms describing these operators are closely related to their biological counterparts as we will see below.

Code Listing 2: Independent bit mutation implementation

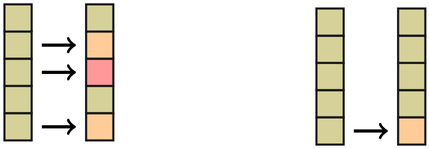
```
#include "boost/smart_ptr.hpp"

template <class T> struct IndependentBitMutation
{
    // type of bounds for each gene
    typedef std::vector< std::pair<double, double> > bounds_t;

    static void apply(boost::shared_ptr<T> ind, bounds_t dVarBounds) {

        double probability = 0.5;

        for(size_t i = 0; i < ind->genes.size(); i++) {
            double rval = static_cast<double>(rand() / (RAND_MAX + 1.0));
            if(rval < probability) {
                double max = std::max(dVarBounds[i].first,
                                      dVarBounds[i].second);
                double min = std::min(dVarBounds[i].first,
                                      dVarBounds[i].second);
                double delta = fabs(max - min);
                ind->genes[i] = rand() / (RAND_MAX + 1.0) * delta + min;
            }
        }
    }
};
```

Mutation operators

The mutation operator models random accidental changes in the set of genes. Compared to discrete (biological) gene "values" (the four nucleobases) our mu-

---

[2] http://www.wfg.csse.uwa.edu.au/hypervolume/



independent bit mutation

one bit mutation

Figure 2.8: Available mutation operators on single individuals.

tation operators have to deal with floating point numbers. Instead of flipping bits in design variables, as in binary coded evolutionary algorithms, we change one design variable to a new value, e.g., inside the bounds of the design variables as shown in Listing 2. Other approaches create mutations non-uniformly or uniformly in the immediate vicinity of the parent, modeling the binary or biological mutation operator more precisely. We visualize two mutation operations in Figure 2.8. The independent bit mutation operator changes each design variable with some probability p_{ind}, e.g. $p_{\text{ind}} = \frac{1}{2}$ mutates half of the design variables in expectation. In contrast, the one bit mutation operator we select only one gene uniform at random and change the value of the corresponding design variable.

Crossover operators

The crossover operator models the exchange of genetic material between homologous chromosomes in the final phase of genetic recombination by breaking and reconnecting matching chromosomes. In contrast to random gene mutation, the crossover operator mixes genes of the two parents, e.g., by taking one half of the genes from the mother and the other half from the father. This is visualized in Figure 2.9 where either random genes are used from either parent or a crossover position is chosen uniformly at random.

Note that the simple one-point crossover shown on the left in Figure 2.9 preserves the average value of the genes. On the other hand, the proposed operators shown in Figure 2.10 use interpolating mechanism to create completely new sets of genes. Genes are essentially blended with help of some models, i.e. the *blend crossover* picks a new gene value uniform at random in

$$\left[x_i^1 - \alpha(x_i^2 - x_i^1),\ \ x_i^2 + \alpha(x_i^2 - x_i^1)\right], \tag{2.5}$$

where

$$x^1 < x^2 \text{ and } 0 < \alpha < 1,$$

where x^1 denotes the first and x^2 the second gene in the blend operations and subscripts denote the i-th value of the gene set. According to the author [**?**],

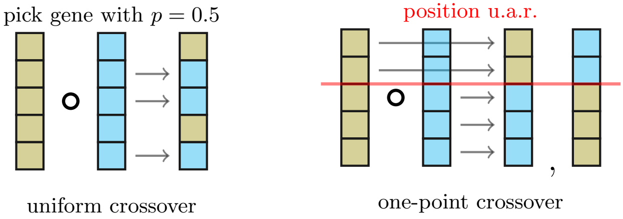


Figure 2.9: Simple crossover operators applied to two parent individuals.

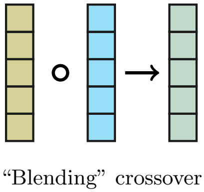


Figure 2.10: Blending crossover operators applied to two parent individuals.

a number of tests seem to point out that $\alpha = 0.5$ has superior performance compared to other α values.

Another blending crossover is the *simulated binary crossover*. This crossover variant models the one-point crossover for binary encoded genes for real valued genes. As before, the operator preserves the gene average value. Using the spread factor

$$\beta_i = \left| \frac{x_i^{2,\text{offspring}} - x_i^{1,\text{offspring}}}{x_i^{2,\text{parent}} - x_i^{1,\text{parent}}} \right|,$$

we can define a probability distribution β_{qi} with a similar search power as the one-point crossover for binary coded genes [?]. With the β_{qi} distribution we choose two new offspring gene sets in the range

$$0.5 \left[(1+\beta_{qi}) x_i^{1,t} + (1-\beta_{qi}) x_i^{2,t}, (1-\beta_{qi}) x_i^{1,t} + (1+\beta_{qi}) x_i^{2,t} \right].$$

This is beneficial when using real-valued genes, because the blending approach better captures the mixing of parent genes. Genes are not just a random composition of the parents genes, but a new solution is generated in-between the parent solutions. An implementation for the α blend crossover is given in Listing 3.

Code Listing 3: Implementation of blend crossover as given in (2.5)

```
template <class T> struct BlendCrossover
{
    static void apply(T* ind1, T* ind2) {

        // BLX-0.5 performs best
        double alpha = 0.5;

        for(int i = 0; i < ind1->genes.size(); i++) {

            double ming   = min(ind1->genes[i], ind2->genes[i]);
            double maxg   = max(ind1->genes[i], ind2->genes[i]);
            double r1     = static_cast<double>(rand() / (RAND_MAX + 1.0));
            double r2     = static_cast<double>(rand() / (RAND_MAX + 1.0));

            double gamma1   = (1 + 2 * alpha) * r1 - alpha;
            double gamma2   = (1 + 2 * alpha) * r2 - alpha;
            ind1->genes[i] = (1 - gamma1) * ming + gamma1 * maxg;
            ind2->genes[i] = (1 - gamma2) * ming + gamma2 * maxg;
        }
    }
};
```

2.4.4 Parallelization

In our master/slave framework introduced in the next chapter, evaluating individuals is embarrassingly parallel. Groups of worker processes take care of any outstanding evaluations and return the computed objectives immediately to the optimizer. From a performance perspective the number of idle workers should be minimized. This can be achieved by ensuring that the optimizer keeps the requests queue filled.

In order to achieve this, we first need to understand the interplay between the variator and the selector in detail. This is shown on the left in Figure 2.11. In standard approaches we wait until all N individuals of a population are evaluated before we pass the N individuals to the selector. After the selector computed the fittest individuals amongst the N freshly evaluated offspring individuals, recombination operators are applied and a new batch of individuals can be evaluated. This explicit synchronization point before the evaluated individuals can be passed to the selector (red box in Figure 2.11) poses as an obvious bottleneck when evaluating evolutionary algorithms in parallel. In the worst case some MPI processes are taking a long time to compute the fitness of the last individual in the pool of individuals to evaluate. During this time all other resources are idle and wait for the result of this one individual in order to continue to generate and evaluate offspring.

To reduce the number of idle workers we remove this barrier by introducing

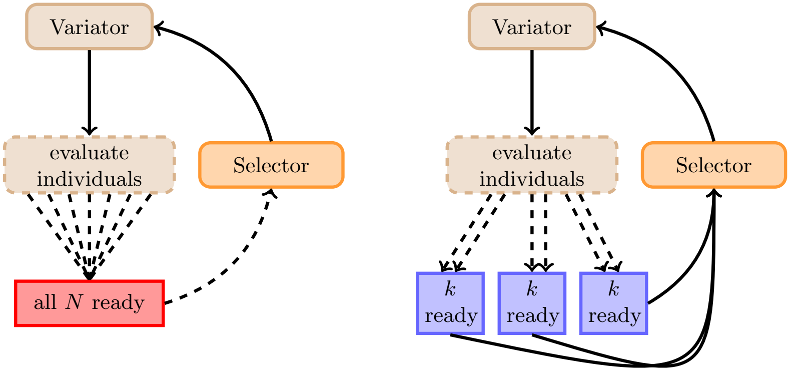


Figure 2.11: Parallel bottleneck and worker starvation with generation "barrier" and "*continuous generations*" as solution (right). N denotes the total number of individuals in the population and $k \in [1, \ldots, N]$ the number of evaluated individuals necessary before calling the selector again.

"continuous" generations. With "continuous" generations the selector is called after k individuals have been evaluated, where $1 \leq k \leq N$. This is visualized on the right side of Figure 2.11. Consequently, with a lower k value we call the selector more often and reducing the chances that we will run out of work. In such a setting, the number of idle workers is $k - 1$ in the worst case. The extreme case, where we choose $k = 1$, lifts the separation between generations and individuals completely. We call the selector every time after only one offspring individual has been evaluated, select new parents and can immediately queue new offspring individuals for evaluation.

Note that the choice of k affects the running time since computing the sorting in the selector is expensive. We introduced a command line argument that allows the user to set a specific k value. The value of k should be chosen with regard to the number of available workers per optimizer. By choosing k such that $k-1$ is a vanishing part of the total number of workers should ensure an acceptable performance. In the next section we will examine the impact of the parameter k on the convergence behavior of evolutionary algorithms.

2.5 Validation

We validated our variator in combination with the NSGA-II selector against the following multi-objective optimization problem presented in [**?**]:

$$
\begin{aligned}
\min \quad & \Bigg[1 - \exp\left(-1\left(\left(x_1 - \frac{1}{\sqrt{3}}\right)^2 + \left(x_2 - \frac{1}{\sqrt{3}}\right)^2 + \left(x_3 - \frac{1}{\sqrt{3}}\right)^2\right)\right), \\
& \quad 1 - \exp\left(-1\left(\left(x_1 + \frac{1}{\sqrt{3}}\right)^2 + \left(x_2 + \frac{1}{\sqrt{3}}\right)^2 + \left(x_3 + \frac{1}{\sqrt{3}}\right)^2\right)\right)\Bigg]^T \\
\text{s.t.} \quad & -1 \leq x_i \leq 1, \quad i = 1, 2, 3.
\end{aligned}
$$

The hypervolume of the reference solution (0.6575) for our benchmark was computed by sampling the solution provided in [**?**]. The optimizer was stopped after 1200 generations using a population of 100 individuals, a blend crossover operator and independent gene mutations. In each generation we evaluate two individuals ($k = 2$) before calling the selector again. This results in a total of 2500 function evaluations for 1200 generations. The upper plot in Figure 2.12 shows the initial random population, the reference solution and the individuals of the 1200th generation, covering the reference solution smoothly.

This results also confirmed, as stated above, that real valued evolutionary algorithms perform much better using interpolation crossover operators such as the blend crossover operator employed here.

To investigate the impact of the parameter k, determining the number of individuals to evaluate before passing them to the selector, we reran the benchmark in the “standard” setting ($k = 100$) calling the selector after the all offspring individuals have been evaluated. The results are shown in the bottom of Figure 2.12. In Table 2.1 we show that no significant differences are visible between the hypervolumes of both solutions after 2500 function evaluations.

However, the convergence seems to be slightly faster when using $k = 2$. By calling the selector more frequently (already after two offsping individuals have been evaluated) results in better populations since bad solutions are rejected faster. On the other hand, calling the selector more often is computationally more expensive. Another factor influencing the comparison is the “fitness” of the initial random population. As shown in the first row in Table 2.1 the initial population in the $k = 100$ case slightly worse compared to the $k = 2$ case.

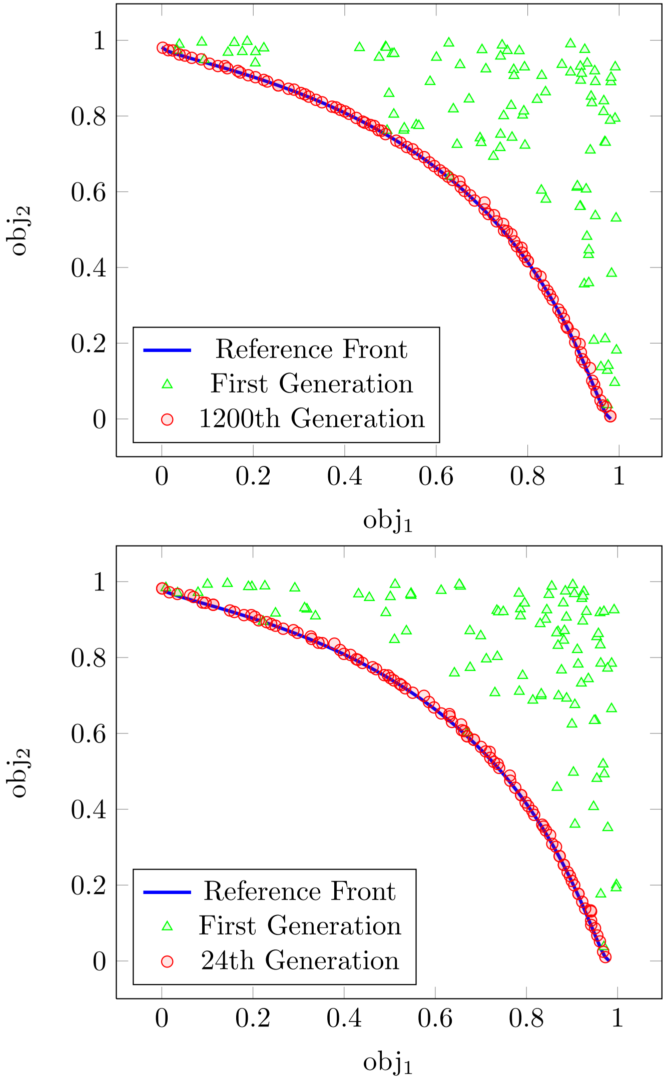


Figure 2.12: Variator benchmark after 2500 function evaluations using binary crossover and independent gene mutations (each gene mutates with probability $p = \frac{1}{2}$) on a population of 100 individuals. Top uses $k = 2$, Bottom $k = 100$.

function evaluations	$k = 2$ hypervol	$k = 2$ relative error	$k = 100$ hypervol	$k = 100$ relative error
100	0.859753	3.076×10^{-1}	0.868817	3.214×10^{-1}
900	0.661898	6.689×10^{-3}	0.740168	1.257×10^{-1}
1100	0.657615	1.749×10^{-4}	0.705240	7.261×10^{-2}
2500	0.656891	9.262×10^{-4}	0.657973	7.194×10^{-4}

Table 2.1: Comparison of hypervolume and errors relative to hypervolume of sampled reference solution after some function evaluations for two k values.

Chapter

The Framework

3

"As I would not be a slave, so I would not be a master. This expresses my idea of democracy."

– Abraham Lincoln

After introducing multi-objective optimization problems in Chapter 2, we turn to the implementation of a general framework for solving simulation based multi-objective problems. The main task of such a framework consists of handling all interactions between the optimization algorithm and simulation runs.

Such simulation based multi-objective optimization problems are omnipresent in research and industry. The simulation and optimization problems arising in such problems are in general very big and computationally demanding. All the more this motivated us to design a massively parallel general purpose framework. The key traits of such a design can be summarized as:

- support any multi-objective optimization method,
- support any function evaluator: simulation code or measurements,
- offer a general description/specification of objectives, constraints and design variables,
- run efficiently in parallel on current large-scale high-end clusters.

This chapter addresses and illustrates our solution to each of the above stated features.

Our framework is divided into three components: the optimizer, the simulation and the pilot. In a first step, the available MPI processes are distributed amongst the components. With this, every MPI process is assigned a specific role that executes the code determined by the component. We visualize this in Figure 3.1. Once every MPI process has determined its role, the framework operates under the master/slave paradigm. The pilot roles correspond to the masters and each gets assigned some simulation and optimization roles in order to solve the specified multi-objective optimization problem. The total number

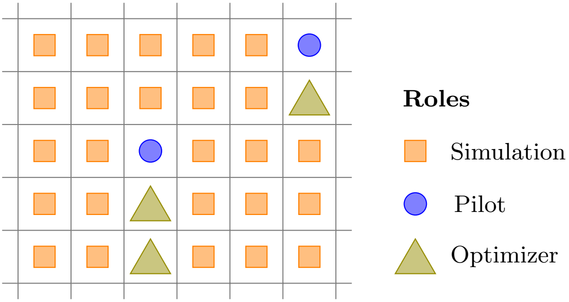


Figure 3.1: Assigning MPI processes roles of the three components.

of master nodes depends on the optimization algorithm and the total number of available cores. If multiple master roles are employed, they will cooperate to solve the problem.

Components, described in Section 3.2, are implemented as exchangable modules in order to cover the first two requirements introduced above. The third requirement, dealing with problem specification and parsing, is addressed in Section 3.4. Finally, Section 3.5 discusses the parallelization of the framework. Visualization of high dimensional solution data is a complex endeavor. In Section 3.6 we briefly introduce two basic visualization tools for inspecting Pareto fronts that we implemeted for the purposes of the present thesis.

3.1 Related Work

Several similar frameworks, e.g. [**?**, **?**, **?**, **?**], have been proposed. Commonly these frameworks are tightly coupled to an optimization algorithm, e.g. only providing evolutionary algorithms as optimizers. Users can merely specify optimization problems, but cannot change the optimization algorithm. Our framework follows a more general approach, providing a user-friendly way to introduce new or choose from existing built-in multi-objective optimization algorithms. Tailoring the optimization algorithm to the optimization problem at hand is an important feature due to the many different characteristics of optimization problems that should be handled by such a general framework. As an example, we show how PISA [**?**], an existing evolutionary algorithm library, was integrated with ease. Similarly, other multi-objective algorithms could be incorporated and used to solve optimization problems.

The framework presented in [**?**] resembles our implementation the most, aside from their tight coupling with an evolutionary algorithm optimization strategy. The authors propose a plug-in based framework employing an island parallelization model, where multiple populations are evaluated concurrently and independently up to a point where some number of individuals of the population are exchanged. This is especially useful to prevent the search algorithm to get stuck in a local minimum. A set of default plug-ins for genetic operators, selectors and other components of the algorithms are provided by their framework. User-based plug-ins can be incorporated into the framework by implementing a simple set of functions.

Unfortunately, the island parallelization approach is not well suited for running search algorithms in the massively parallel environment. The decision how many MPI processes are assigned to an island is non-trivial. Parallel performance is sensitive to placement and assignment of roles to MPI processes of such frameworks and therefore should be considered carefully. Especially the master process on an island (used to exchange individuals between the islands and aggregate solutions) can be overwhelmed with messages from slaves evaluating individuals. This creates hotspots on the network and as a result the network performance suffers. To counteract this effect we propose an adaptable graph model as underlaying communication graph between masters. As we report in Section 5.1 this leads to a better parallel performance.

Additionally, as with simulation based multi-objective optimization, we can exploit the fact that both the optimizer and simulation part of the process possess different computational complexity and therefore require a certain fraction of the available computational resources. An assignment of the number of MPI processes to each component, depending on the implementation of the components, determines this relationship. This not only provides users with great flexibility in using any simulation or optimizer, but as well renders influencing the role assignment easy.

3.2 Components and Plugins

The basic assumption in simulation-based optimization is that we need to compute an expensive simulation software component present in the constraints or objectives. We divide the framework in three exchangeable components, as shown in Figure 3.2, to encapsulate the major behavioral patterns of the framework. Pilot components act as a bridge between optimizer and forward solvers, providing the necessary functionality to handle passing requests and results between Optimizer and Simulation.

Using template parameters, the composition of the framework can be specified as shown in Listing 4.

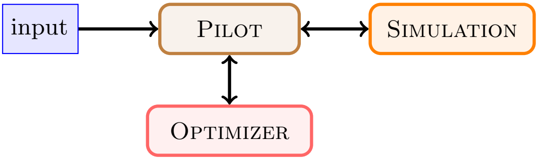


Figure 3.2: The optimization problem specified in the input is solved by a multi-objective algorithm using workers to evaluate simulations.

Code Listing 4: Assembling the optimizer from components

```
typedef OpalInputFileParser  Input_t;
typedef PisaVariator         Opt_t;
typedef OpalSimulation       Sim_t;

typedef Pilot< Input_t, Opt_t, Sim_t > Pilot_t;
```

The framework provides "default" implementations that can be controlled via command line options. Due to its modular design, all components can be completely customized. This topic will resurface through-out the rest of this chapter as we discuss the individual components in detail.

The Optimizer Component

This component implements an algorithm to solve a multi-objective optimization problem. Besides solving the optimization problem the Optimizer should be able to request information, such as function evaluations or derivatives from the Pilot.

The Simulation Component

This component is in charge of running simulations corresponding to information requests issued by an optimizer. It listens to requests issued by the Pilot, computes the requested values, and returns results back to the Pilot.

The Pilot Component

Partially depending on the total number of cores and the optimization algorithm, but mainly due to the bottleneck at a master node, we allow the framework to assign multiple pilot roles. This leads to two important tasks for each present pilot: It

- coordinates the efforts between optimizer and simulation workers using an *inter-component* communication, and it
- coordinates results with other Pilots (if any) using an *intra-component*

communication.

Plugins

Compared to the coarser partition in three major components, a fair number of details of a components implementation can be steered on a lower level with the help of plugins. We realized framework plugins by means of policy design pattern [**?**] (also known as strategy pattern) which are related to the *mixin* [**?**] design pattern. Both patterns provide a way to satisfy the open/close principle [**?**].

Plugins are not intended to be instantiated as stand alone objects, but rather have to be attached to another class (e.g. by inheriting from a plugin) contributing (or mixing in) additional functionality. Here, the inheritance is not used as a mean of specialization but rather as an accumulator of functionality.

We can use this mechanism like multiple inheritance for extending functionality of a class by a set of methods without predetermining what actually can be extended. With mixins we avoid some of the pitfalls introduced when using multiple inheritance.

With help of this construct we can create simple and small plugins covering just one specific functionality of the core components which is in turn easy to interchange. To illustrate this with a concrete example we consider the implementation of genetic mutation operator, as introduced previously, mutating the genes of an individual. The implementation we present in Listing 5 can be used to contribute the mutation operator to the variator.

Code Listing 5: Implementation of one bit mutation operator

```
struct MyBitMutation {

    void mutate(Individual* ind) {
      int range = ind->genes.size();
      int position = static_cast<int>((rand() / (RAND_MAX + 1.0)) * range);
      ind->new_gene(position);
    }
};
```

Now that we have defined a simple plugin, we can use it to extend the optimizer with the functionality of mutation genes as shown in Listing 6.

Code Listing 6: Extending the variator with the mutation operator

```
template< class MutationOperator >
class Variator : public MutationOperator {

    void createOffspring(/*...*/) {
        /* ... */
        this->mutate(individual);
    }
};
```

We use this policy pattern throughout the framework to determine the fine granular behavior of the major components. The user can specify the employed plugins by changing the driver or by exposing the selection as command line arguments.

Summarizing, in order to integrate a new plugin the user has to implement its functionality in a header file, include the header in the drivers and replace the template name of the old plugin with the newly written one. This is shown in Listing 7.

Code Listing 7: Selecting plugins

```
#include "MyBitMutation.h"

/* ... */

typedef PisaVariator< BlendCrossover, MyBitMutation > Opt_t;

/* ... */
```

This leads to a framework where top-level components can be changed or interchanged (by implementing new classes respecting the abstract interfaces of the components) and the low-level configuration and behavior of subpart of the components can be steered or changed using either mixins or policies. The detailed implementation of these components is exposed in the remainder of this chapter.

3.3 Inter-Component Communication

In order to minimize the idle waiting time of the components, they have to behave “asynchronously”. We implemented a mechanism that is related to *futures* [?] (or synonymously *delay* or *promises*), acting as synchronizers in asynchronous systems. *Futures* act as proxy objects for a yet to be accomplished computation. In this context, *futures* are used to allow components to request some information and immediately return to their own local computational task, without waiting for the computation to deliver the result first.

At some point in the future, the result is delivered and the component reacts upon receiving the result, e.g., by integrating the result in the computation.

This mechanism is implemented via an event-driven state machine. `MPI_Tag`'s are used to label different events (or states). A common protocol, defining sequence of information exchanges ensure that components can "talk" to each other in a uniform way. This is necessary to allow users to extend or implement their own components interacting with the framework.

3.3.1 Event-Driven MPI State Machine

All components share a common part of the state machine. This common behavior is captured by an abstract class, named `Poller`. This class has a set of virtual functions (hooks) and one concrete implementation of a polling loop, as shown in Algorithm 3. Components deriving from the `Poller` class are

Algorithm 3 Default polling behavior of the MPI state machine.

```
procedure RUN
    setupPoll()                                   ▷ hook for setup

    MPI_Irecv()
    running ← $\top$

    repeat
        prePoll()                                 ▷ hook for setup of loop pass
        if new request $r$ then
            if $r$.TAG = STOP_TAG then
                running ← $\bot$
                onStop()                          ▷ on stop hook
            else
                if onMessage($r$) then            ▷ handle request
                    MPI_Irecv()                   ▷ continue polling
                else
                    running ← $\bot$
                end if
            end if
        end if
        postPoll()                                ▷ hook for reactions after loop pass
    until ¬ running
end procedure
```

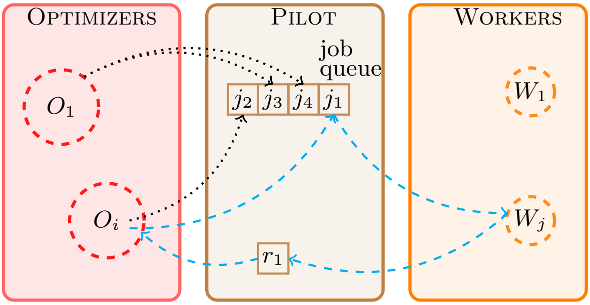


Figure 3.3: Schematic view of messages passed within the network. The dashed blue path describes a request (job j_1) sent from O_i to the PILOT being handled by W_j. Subsequently the result r_k is returned to the requesting OPTIMIZER (O_i).

extended by a simple poll loop that can be customized via hooks or completely overridden. The poll loop implemented in the `Poller` posts asynchronous MPI receives and continuously checks if a new message has arrived. If a new message has been received, we first check if the MPI tag corresponds to the predefined stop tag and exit the poll loop if this is the case. Otherwise, the received message is passed to the `onMessage` message handler that implements the component specific behavior. As arguments, the `onMessage` method receives the `MPI_Status` of the received message and an unsigned value whose meaning depends on the event or `MPI_Tag`. In most cases this number is either a dummy value or holds a message size of some sort.

Additionally there are four special method calls to `setupPoll`, `prePoll`, `onStop` and `postPoll`. Implementing any of these lets the component influence the polling mechanism in crucial phases, e.g., the component can process information after one iteration of the poll loop has finished.

After an MPI process gets appointed a role, it starts a polling loop asynchronously listening for incoming requests.

3.3.2 Job Handling

The protocol underlaying the above introduced state machine is mainly depending on the way requests are “routed” between components. Figure 3.3 visualizes the request flow from OPTIMIZER over PILOT to SIMULATION workers.

An information request origins at an Optimizer component and is enqueued in the work queue of the Pilot. The Pilot keeps track of busy and idle workers and assigns jobs in its work queue to any free worker. Once the worker has computed and evaluated the request it is routed back to the Optimizer that originally requested the information.

Information requests are implemented to consist of a set of design variables (e.g. the genes of an individual) and an information type they request (function evaluation or derivative).

To exchange information between the components (Optimizer ⇔ Pilot ⇔ Worker), we implemented wrappers around MPI point to point communication routines. These wrappers use Boost Archive[1] for serialization and deserialization of the request data structures. After serialization the resulting bytestring is sent to the receiver using primitive point to point communication. The new point to point methods `MPI_Send_*` and `MPI_Recv_*`, where the asterisk denotes type (design variables or results) depending implementations, require a receiver (respectively sender) MPI process id and a communicator. The implementation of these methods actually requires two primitive MPI send and receive point to point calls. First we have to inform the receiving MPI process of the size of the serialized data and in a second step we can send the actual data. Another advantage of using the serializer is its compression property.

For convenience, a serialized broadcast was implemented as well, sending design variables or results to all MPI processes of the participating communicator. This is mostly used to inform MPI processes working together as a component of newly received data.

3.3.3 Optimizer API

All Optimizer component implementations have to derive from the `Optimizer` class, which in turn derives the `Poller` class. The `Optimizer` class, defining the API for all Opimizer implementations, is shown in Listing 8.

---

[1] www.boost.org/libs/serialization/

Code Listing 8: Optimizer API

```
class Optimizer : protected Poller {

    /* ... */

    virtual void initialize() = 0;

private:
    // 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;
};
```

All MPI processes running an Optimizer component will call the `initialize` entry point after role assignment in the Pilot. The implementation of `initialize` should set up and start poller and optimization code. Hooks can be implemented as empty methods, but the `onMessage` implementation should reflect the optimization part of the protocol for handling events from the Pilot. To facilitate request exchanges with the Pilot the protocol shown in Figure 3.4 has been implemented. An Optimizer sends a new request using the `OPT_NEW_JOB` tag, e.g., for a function evaluation, where input values correspond to the values for the design variables. Once a request has finished the Optimizer receives a message with the tag `REQUEST_FINISHED`. The accompanying unsigned number (`id`) specifies the job id the Optimizer has assigned to the task before shipping it of to the Pilot. We make use of the introduced serialized point to point communication to exchange parameters and results.

Once the optimizer converged it notifies its Pilot with an `OPT_CONVERGED_TAG`. Depending on user settings the Pilot either propagates the converged status to all other Pilots to stop all ongoing optimization processes or simply waits on the other Pilots to converge.

3.3.4 Simulation API

In most cases Simulation implementations are simple wrappers to run a preexisting “external” simulation code using the set of received design variables. As for the Optimizer component there exists a base class, labeled `Simulation` as common basis for all Simulation implementations. In addition, this component also inherits from the `Worker` class, already implementing the polling protocol shown in Figure 3.5 for default worker types. As shown in the API in Listing 9 the `Worker` class expects an implementation to provide implemen-

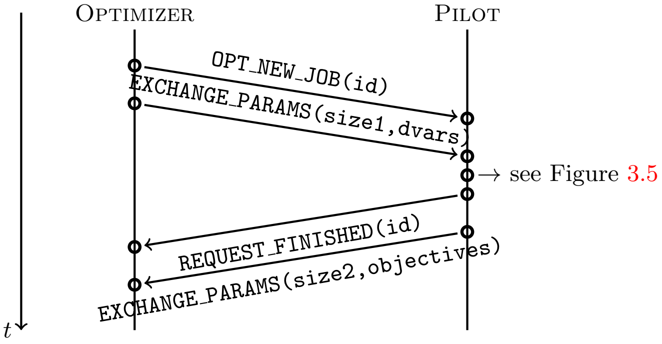


Figure 3.4: Protocol for information requests between OPTIMIZER and PILOT.

tations for those three methods.

CODE LISTING 9: SIMULATION API

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

First, upon receiving a new job, the `Worker` class will call the `run` method on the SIMULATION implementation. This expects the SIMULATION implementation to run the simulation in a *blocking* fashion, meaning the method call blocks and does not return until the simulation has terminated. Subsequently, the `Worker` calls `collectResults`, where the SIMULATION prepares the result data, e.g., parsing output files and stores the requested information in a `reqVarContainer_t` data structure. Finally, the results obtained with `getResults` are sent to the PILOT. As before, data is exchanged using the implemented MPI serialized point to point functions.

It goes without saying that an OPTIMIZER can implement its own polling mechanism respecting the protocol, but in most situations the `Worker` implementation is all that is needed.

3.4 Optimization Problem Parser

We aimed at an easy and expressive way for users to specify multi-objective optimization problems. Following the principle of keeping metadata (optimization and simulation input data) together, we decided to embed the opti-

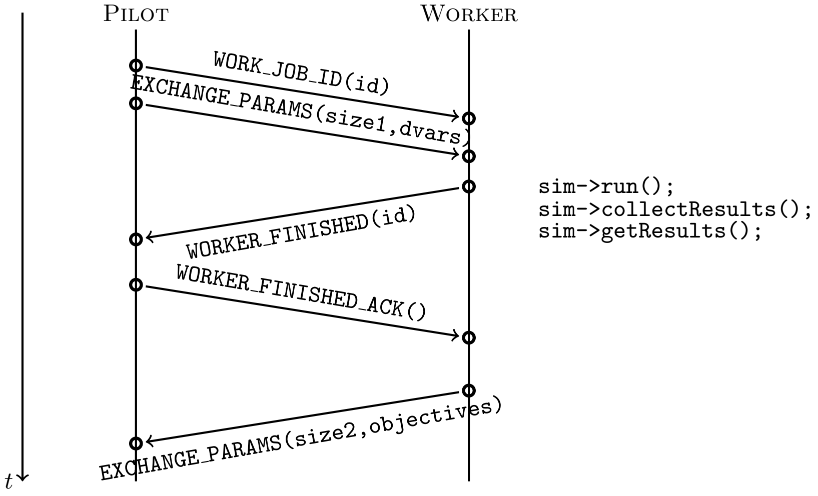


Figure 3.5: Protocol for job requests between the Pilot and a Worker.

mization problem specification in the simulation input file by prefixing it with special characters, e.g., as annotations prefixed with a special character. In some cases it might not be possible to annotate the simulation input file. By providing an extra input file parser, optimization problems can be read from stand-alone files.

To allow arbitrary constraints and objective expressions, such as

```
name: OBJECTIVE,
      EXPR="5 * average(42.0, "measurement.dat") + ENERGY";
```

we implemented an expression parser using Boost Spirit[2].

In this setting any combination of simulation output variables (during any point of the simulation) and user defined operator can be used to compute objectives or constraints. Using the `C++` operators the parsers EBNF (in Boost Spirit syntax) is shown in Listing 10.

Code Listing 10: Boost Spirit Parser EBNF

```
expr =
```

---

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

```
    logical_or_expr.alias()
    ;

logical_or_expr =
        logical_and_expr
    >> *(logical_or_op > logical_and_expr)
    ;

logical_and_expr =
        equality_expr
    >> *(logical_and_op > equality_expr)
    ;

equality_expr =
        relational_expr
    >> *(equality_op > relational_expr)
    ;

relational_expr =
        additive_expr
    >> *(relational_op > additive_expr)
    ;

additive_expr =
        multiplicative_expr
    >> *(additive_op > multiplicative_expr)
    ;

multiplicative_expr =
        unary_expr
    >> *(multiplicative_op > unary_expr)
    ;

unary_expr =
        primary_expr
    |   (unary_op > unary_expr)
    ;

primary_expr =
        constant_expr
    |   function_call
    |   identifier
    |   bool_
    |   '(' > expr > ')'
    ;

constant_expr =
        double_
    |   uint_
    ;

function_call =
        (identifier >> '(')
    >   argument_list
    >   ')'
    ;

quoted_string %=
        '"'
    >   *(char_ - '"')
    >   '"'
    ;

argument_list = -((expr | quoted_string) % ',');

identifier =
        !lexeme[keywords >> !(alnum | '_')]
    >>  raw[lexeme[(alpha | '_') >> *(alnum | '_')]]
    ;
```

In addition to the parser, we need an evaluator able to evaluate an expression, given a parse tree and variable assignments to an actual value. Expressions arising in multi-objective optimization problems usually evaluate to booleans or floating point values. The parse tree, also denoted abstract syntax tree (AST), is constructed recursively while an expression is parsed. Upon evaluation, all unknown variables are replaced with values, either obtained from simulation results or provided by other subtrees in the AST. In this stage, the AST can be evaluated bottom-up and the desired result is returned after processing the root of the tree.

To improve the expressive power of objectives and constraints we introduced a simple mechanism to define and call custom functions in expressions. Using simple functors, e.g., as the one shown in Listing 11 to compute an average over a set of data points, enriches expressions with custom functions. Custom function implementations overload the `()` parenthesis operator. The function arguments specified in the corresponding expression are stored in a `std::vector` of Boost Variants[3] that can be booleans, strings or floating point values.

---

[3] www.boost.org/doc/html/variant.html

Code Listing 11: Simple Average Functor

```
struct average {

    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;
    }
};
```

All custom functions are registered with expression objects. This is necessary to ensure that expressions know how they can resolve function calls in their AST. As shown in Listing 12 this is done by creating a collection of Boost Functions[4] corresponding to the available custom functions in expressions and passing this to the Pilot.

Code Listing 12: Creating function pointer for registering functor

```
functionDictionary_t funcs;
client::function::type ff;
ff = average();
funcs.insert(std::pair<std::string, client::function::type>
        ("my_average_name", ff));
```

A set of default operators, corresponding to a mapping to `C` math functions, is included in the dictionary by default. This enables an out of source description of the optimization problem. Considering the benchmark problem introduced at the end of the last chapter, the following input file describes the problem completely:

```
d1: DVAR, VARIABLE="x", LOWERBOUND="-1.0", UPPERBOUND="1.0";
d2: DVAR, VARIABLE="y", LOWERBOUND="-1.0", UPPERBOUND="1.0";
d3: DVAR, VARIABLE="z", LOWERBOUND="-1.0", UPPERBOUND="1.0";

obj1: OBJECTIVE,
  EXPR="1- exp(-1 * (sq(x - 1/sqrt(3)) + sq(y - 1/sqrt(3)) + sq(z - 1/sqrt(3))))";
obj2: OBJECTIVE,
  EXPR="1- exp(-1 * (sq(x + 1/sqrt(3)) + sq(y + 1/sqrt(3)) + sq(z + 1/sqrt(3))))";

objs:    OBJECTIVES = (obj1, obj2);
dvars:   DVARS = (d1, d2, d3);
```

---

[4] www.boost.org/libs/function/

```
constrs: CONSTRAINTS = ();
opt: OPTIMIZE, OBJECTIVES=objs, DVARS=dvars, CONSTRAINTS=constrs;
```

3.5 Parallelization

One major disadvantage the of master/slave implementation model is the fast saturation of the network links surrounding the master node. In [**?**] authors observe an exponential increase in hot-spot latency with increasing number of workers that are attached to one master process. Clearly, the limiting factor is the number of outgoing links of a node in the network topology and already for a few workers the links surrounding a master process are subject to congestions. The effect is amplified further by large message size.

These hot-spots severely limit the maximal attainable parallel performance, especially in the case of light workload for workers. By increasing the number of master processes, e.g., by splitting the search space or exploiting specific characteristics of the search algorithm, the problem can be (at least partially) alleviated. Unfortunately, this introduces additional complexity – the performance depends on the placement and number of master processes. In this setting the question arises as to whether an optimal mapping of master processes onto the network topology exists. As shown for example in [**?**] additional information, e.g., the interconnection network topology and the application communication graph to determine the mapping can improve the parallel performance. Using architecture dependent libraries the hardware network connection graph can be obtained easily and the application communication graph for master/slave applications is known prior to execution once the number of master processes has been fixed.

In this paper we utilize the network connectivity graph in the context of a large-scale multi-objective optimization framework to:

- determine the number and location of master nodes in order to minimize hot-spots and congestion in the network,
- improve parallel efficiency of master/slave based applications by implementing general and easy to extend mapping strategies for roles in a network topology-aware fashion,
- define a communication grid of master nodes, modeled after a social rumor network, to exchange solution states using one-sided communication to avoid global synchronization points.

We stress that the proposed approach can be applied to to any randomized search algorithm.

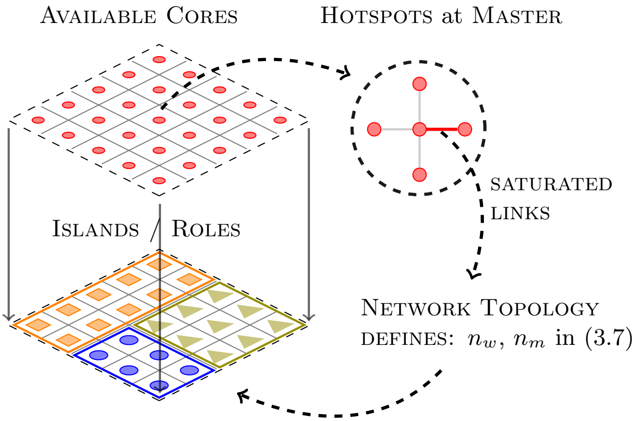


Figure 3.6: Size and shape of master/slave islands are determined by the network topology minimizing hotspots.

We start by discussing the underlaying network model, simplifications for master/slave applications and details of our implementation using one sided communication to exchange solutions.

In Chapter 5 we evaluate the efficiency of the proposed approach using a real world optimization problem.

3.5.1 Theoretical Network Model

Our theoretical considerations are based on Hoefler's and Snir's network model [?]. This model allows us to analyze properties, such as network congestion or traffic of the network under different role distributions. The goal is to find a mapping between vertices of the weighted communication graph $V_{\mathcal{G}}$ of an application and the hardware network connection graph (network wiring) $V_{\mathcal{H}}$

$$\Gamma : V_{\mathcal{G}} \to V_{\mathcal{H}}, \tag{3.1}$$

respecting certain optimality conditions. Of particular interest is the minimization of the network congestion to improve the parallel performance in a master/slave framework. The weighted communication graph $\mathcal{G}$ is defined by vertices and edge weights w

$$\mathcal{G}(V_{\mathcal{G}}, w_{\mathcal{G}}), \tag{3.2}$$

and the network connection graph $\mathcal{H}$ describes the hardware of a cluster

$$\mathcal{H}(V_{\mathcal{H}}, C_{\mathcal{H}}, c_{\mathcal{H}}, R_{\mathcal{H}}), \tag{3.3}$$

where C denotes the number of cores per node, c the capability of links and R the routing algorithm.

The average *dilation* D expresses the average number of hops a message travels until it reaches its destination and is defined as

$$\begin{aligned} D(u,v) &= \sum_{p \in P(\Gamma(u)\Gamma(v))} R_{\mathcal{H}}(\Gamma(u)\Gamma(v))(p) \cdot |p| \\ D(\Gamma) &= \sum_{u,v \in V_{\mathcal{G}}} w_{\mathcal{G}}(uv) \cdot D(u,v), \end{aligned} \tag{3.4}$$

where $R_{\mathcal{H}}(uv)(p)$ is the fraction of traffic from u to v routed through path p.

The average *traffic* τ over an edge e is defined as

$$\tau(e) = \sum_{u,v \in V_{\mathcal{G}}} \left(\sum_{p \in P(\Gamma(u)\Gamma(v)), e \in p} R_{\mathcal{H}}(\Gamma(u)\Gamma(v))(p) \right). \tag{3.5}$$

Finally the *congestion* χ can be defined by traffic and capacity over edges e and mapping Γ respectively as

$$\begin{aligned} \chi(e) &= \frac{\tau(e)}{c_{\mathcal{H}}(e)} \quad \text{and} \\ \chi(\Gamma) &= \max_{e} \chi(e). \end{aligned} \tag{3.6}$$

Due to the relation between the congestion and the communication time, the congestion of the mapping $\chi(\Gamma)$ can be used as a lower bound for the total communication time.

Simplifications for the master/slave model

The problem of finding an optimal mapping Γ has been shown to be NP-complete. Using several simplifications of the model, e.g., assuming the traffic is evenly distributed on all routing paths (commonly the shortest paths), good approximations can be computed. For master/slave applications the problem complexity can be further reduced because the network topology and the communication graph are known a-priori. Most notably the special star-like shape of the communication graph helps in predicting congestions and defining a model to evaluate various choices and placements.

For the remainder of this chapter we denote the total number of processors by n_p and the number of masters "islands" running concurrently by n_m. Each of the n_m masters employs n_{mp} processors for running the optimizer component and has access to n_{wpm} worker groups, each using n_w processors to run simulations. With this definitions the following properties hold in master/slave networks:

- In the worst case $\mathcal{G}$ is a set of n_m star graphs (transformation of grid to star) with
$$|E| = \frac{n_p - n_m}{n_m}. \tag{3.7}$$
The centers of the stars are connected in a grid-like fashion.
- All paths are similar as they are single bidirectional paths from slaves to a master.
- In a load balanced setting the fraction of traffic on a path p is assumed to be uniformly distributed (data sent from master to slaves and back have roughly the same size).

Under the assumption that the computation is reasonably well balanced, the traffic arriving through edge e at a master node is uniformly distributed over the incoming links. Therefore, the fraction of traffic arriving through edge e at the master node is proportional to the total number of workers divided onto the number of incoming links to the master node. The traffic increases as we approach master nodes and we can conclude that edges surrounding a master have the most traffic. This leads to the worst-case congestion

$$\chi(\Gamma) \approx \frac{\tau_m(e)}{c_{\mathcal{H}}(e)}. \tag{3.8}$$

The definition of τ_m depends on the underlaying network, e.g., on a d dimensional Torus we distribute the non-master cores equally amongst the masters

$$\tau_m(e) \propto \frac{n_p - n_m}{n_m}. \tag{3.9}$$

The definition of τ_m varies with the network topology but in every case a fraction of the traffic arrives through links surrounding a master node.

This enables us to express the congestion depending on the number of specified masters independently of any other values. On a d dimensional torus, the fraction of traffic routed through an edge surrounding a master is 0 for edges not part of the sub-cube spawned at the master with length l and 1 otherwise. The inner most sum in (3.5) therefore reduces to

$$l = \frac{1}{2}\sqrt[d]{\frac{n_p - n_m}{n_m} + 1} = \frac{1}{2}\sqrt[d]{\frac{n_p}{n_m}}, \tag{3.10}$$

assuming all edges have the same weight we get

$$\chi(n_m, \mathcal{H}) \le \frac{w_{\mathcal{G}}}{2c_{\mathcal{H}}(e)}\sqrt[d]{\frac{n_p}{n_m}}. \tag{3.11}$$

This means that with increasing number of masters the congestion shrinks (traffic over master links decreases).

For systems with fat nodes the optimal configuration with respect to congestion on links surrounding a master node is attained when a master/slave subsystem fits completely on a single node. In general, we can specify a maximal saturation and the network-aware implementation will determine the number of masters required to attain the specified saturation.

This leads to a parallelization as depicted in Figure 3.7.

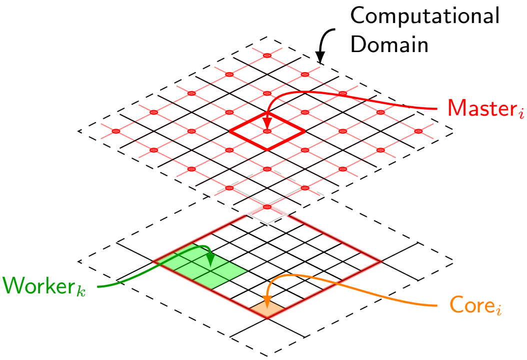


Figure 3.7: Hierarchical layout, top-level represents the master grid (network of all master nodes) using Cartesian neighbors and random links to propagate solution and bottom-level the master assigns cores to workers.

3.5.2 Communicators

The assignment of MPI processes to roles is implemented with strategy policies making use of the policy design pattern introduced above. This provides a flexible base for testing and evaluating different mappings of roles and placements as they can easily be replaced with other implementations. Once the strategy has been "executed" every MPI process has access to a bundle of communicators to send messages according to the protocol introduced above. This bundle varies with the role of the MPI process. The optimizer and simulation roles have an internal communicator and the corresponding communicator with the master node. The pilot role has access to all three communicators:

- INTERNAL: MPI processes working on the same subproblem, e.g., all MPI processes assigned to one simulation run. This can be `MPI_COMM_SELF` if a component consist only of a single core.
- WORKER-COMM: The master MPI process and all leaders of it's assigned workers solving the forward problem. This communicator is used to distribute jobs on available worker groups.
- OPT-COMM: The master MPI process and its assigned optimizer leader MPI process executing the optimization algorithm.

We specify the mapping of roles to communicator groups with means of a coloring. Every strategy policy defines a coloring for every communicator group (see Figure 3.9). For the sake of illustration let us look at one MPI process that was assigned the simulation role. This MPI process does not participate in the OPT-COMM (no color). It uses its own color in the INTERNAL communicator because it runs the simulation on a single MPI process. Finally it specifies the same color as all other simulation roles working with the same pilot as master for the WORKER-COMM. Colorings are fairly straightforward to generate (every core simply has to decide in which communicator groups it participates) and application specific role assignment becomes trivial. As shown in Figure 3.8 each role uses its processor id to assign each of the three communicators a color (number) and `MPI_UNDEFINED` if it does not participate in this communicator. After performing an `MPI_Comm_split` the necessary communicator bundle is created and each role can use the different communicators in the bundle to implement the protocol introduced above.

As a default the master consists of one core and is attached to the optimizer (preferably on the same node).

This functionality is implemented in the `CommSplitter` class and after creation each processor is assigned a role and is part of one or more communicator groups.

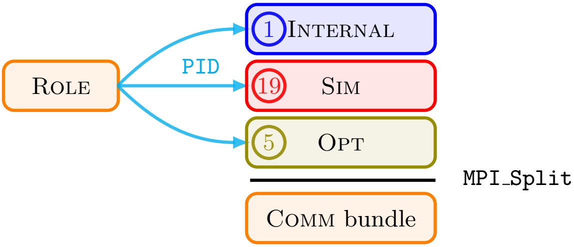


Figure 3.8: Mapping roles to communicator bundles by providing colors (or `MPI_UNDEFINED`) for different communicators and calling `MPI_Comm_Split`.

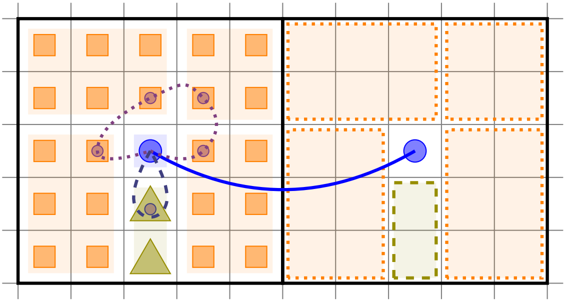


Figure 3.9: INTERNAL communicators are defined by solid lines for different components and dashed lines represent communicator groups between master and optimizers or workers.

3.5.3 Solution State propagation

As mentioned previously it is important to avoid hot-spots in the network and we are well advised to relieve a master process of any unnecessary communication work. Using the INTERNAL communicator master nodes can exchange solution states among each other. To alleviate huge message sizes and communication times with large number of masters we propose the to follow two concepts to attain good performance when using a large number of masters while retaining a good solution diversity:

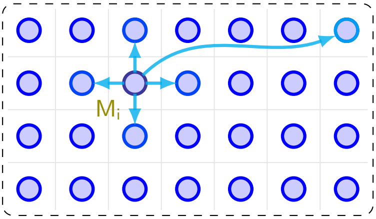


Figure 3.10: On the 2D master grid, the master $\mathsf{M_i}$ is connected to its Cartesian neighbors and one additional random link.

- *only local communication:* exchange solution states with a small subset of other masters, and
- *prevent global synchronization:* use one-sided communication.

A lot of research has been devoted to study distributed algorithms for building consensus with minimal number and size of messages, e.g. computing the average of n values on n systems each containing exactly one value. It has been shown, e.g. in [**?**, **?**], that these problems can be solved efficiently by employing gossip algorithms. The information (or values) is only sent to neighbors and spreads slowly through the network until a consensus has been reached. In [**?**] techniques for accelerating building consensus are discussed that possibly could be applied for solution propagation as well. In the following we will show how we use the same concept of a rumor propagation to implement our solution exchange on the large network of masters.

We start by placing all masters on a grid with the same topology as the underlaying hardware network wiring, e.g. a 2 dimensional torus. Subsequently every master chooses a set of "neighbors" (arbitrary set of masters) that participates in a solution exchange. We incorporate the "real" network topological neighbors of a master directly into the neighbor set because of their short distance. Exchanging solutions only with close neighbors has an unsatisfactory slow gossip behavior, that is, it takes a long time for the solutions to spread. By adding long range masters we hope to attain a better rumor behavior. We visualized the concept of such a long range "neighbor" in Figure 3.10. A graphs of this form is called *augmented grid* and its definition is given in Definition 2.

Definition 2. *An augmented grid $\mathcal{A}_n(E_a, V_g)$ is a regular grid $\mathcal{G}_n(E_g, V_g)$ extended with a set of additional edges E_e*

$$E_a = E_g \cup E_e \,.$$

Both graphs $\mathcal{A}_n$ and $\mathcal{G}_n$ have the same set of vertices V_g.

In the following we will use a special instance of the augmented grid, the social network, to model the connections between the masters used to exchange solutions. Social networks can be modeled using augmented grids, where the set E_e consists of an additional long range neighbor (as shown in Figure 3.10) following a specific distance distribution. As shown by Kleinberg [**?**] in a quantitative study this long range neighbor needs to be drawn from a power law distribution

$$P\left[\exists \text{ edge between } (u,v)\right] \propto \frac{1}{d(u,v)^{-\alpha}},$$

where u and v are two nodes on $\mathcal{G}_n$ and $d(u,v)$ denotes the Manhattan distance between u and v. The long range neighbor essentially models a friend living in a distant environment. In our setting this ensures that the solution states propagate to a remote location on the social graph, accelerating the rate good solutions propagate through the network.

To our advantage a fair amount of research has been spent on analyzing the theoretical properties of social networks, e.g. social network graphs using a power law distribution of random neighbors possess a diameter of $\mathcal{O}(\log n)$ in expectation. Since the distribution of the long range neighbor can be steered by the parameter α we are interested in the α value producing the graph where rumors are spreading the quickest. In [**?**] it has been shown that the choice $\alpha = 2$ results in a graph where the path (using greedy routing) between any two nodes a and b has length $\mathcal{O}(log^2 n)$ in expectation. This bounds the number of rumor steps we have to perform to propagate solutions through master networks before the best solution states have reached all running optimizers with high probability. As shown in the results in Chapter 5 this indeed helps further reducing the communication at master hot-spots further.

In order to achieve satisfactory parallel performance, we aim to prevent the use of global synchronization primitives if possible (depends on the search algorithm). To that avail one-sided MPI communication can be used, creating the illusion of a shared address space where processes can read from and write to a "memory windows". Each Pilot possesses such a window where its current solution state is stored. Using appropriate MPI calls other processes can read the content of the window to retrieve solution states of other Pilots. The algorithm used by the Pilot p_i is stated in Algorithm 4.

In addition to the memory window holding the solution state an extra window holding a revision number of the solution state was introduced. This shared "revision counter" (see line 2) is used to enumerate versions of the solution space. With help of the revision number other Pilots can check if a

Algorithm 4 Propagate solution states in rumor network. The solution state variable $\mathcal{S}$ and its revision v are *shared variables* (`MPI_Win`) in the master neighborhood.

```
 1: procedure propagateSolutionState($k$)
 2:     $v \leftarrow 0$                                  ▷ revision of stored solution state
 3:     $v\,[] \leftarrow 0$                              ▷ revisions of neighbors $\mathcal{S}$

 4:     repeat
 5:         $\mathcal{S} \leftarrow$ perform $k$ steps of search algorithm
 6:         $v \leftarrow v + 1$
 7:         for all $p \in$ neighboring masters do
 8:             if $v_p > v\,[p]$ then
 9:                 $v\,[p] \leftarrow v_p$
10:                 $\mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{S}_p$
11:             end if
12:         end for
13:     until converged
14: end procedure
```

new solution has been stored since the last check (see line 8) and needs to be collected for incorporation in its search efforts. After running some step of the search algorithm the Pilot is instructed to exchange solution spaces. First it stores the current solution space of the Optimizer in the shared variable $\mathcal{S}$, increases the revision counter and starts a *collect* operation. Upon completion, the gathered new solution spaces are send to the Optimizer which will in turn merge them with its solution space. This process is repeated until the search algorithm converges. Aside from providing an asynchronous way to exchange solution states, this also introduces a certain degree of redundancy.

In practice, the optimizer pushes its solution state, using the `EXCHANGE_SOL_STATE` tag, at any algorithm specific rate to the Pilot. The Pilot executes a store and collect operations and returns the collected solution states of specific neighboring Pilot nodes. The `MasterNode` component is responsible for the implementations of the store and collect operations and also encodes knowledge about neighbors to exchange solution states with. Each implementation of a `MasteNode` implements a `store` and `collect` operation that ensure that the solution states are exchanged between the correct neighbors. The Pilot keeps such a `MasterNode` object and relies on its `store` and `collect` methods to exchange solution states when receiving an order from the Optimizer.

3.6 Visualization

Visualizing high dimensional surfaces in a human comprehensible and explorable way is difficult but often crucial for multi-objective optimization frameworks. Without elaborate visualization it becomes complex to interpret and explore high dimensional Pareto surfaces and grasp the extent of trade-off decisions. In these cases there are two options: dimensionality reduction (e.g. principal component analysis) or 2D "multivariate data" plots such as star plots or parallel coordinates. The latter is especially powerful in visualizing clusters in the data, but is not able to convey a spatial picture of the actual Pareto surface.

In order to visualize Pareto fronts during the development and application phases of our framework we implemented a simple Python script. To handle more elaborate analysis and potential for on-line visualization of the search process a web server based solution was implemented. This enables end users, e.g. operators in control rooms of particle accelerators, to navigate the search process (and possibly steer in the future) while the optimization framework is running on a cluster. In addition, this allows the visualization of results without prior transfer to another system – the data analysis can be performed directly on the cluster.

Both visualization tools are able to work with data dumped in the JSON[5] file format. Solution states can be dumped at arbitrary frequency to JSON files.

The Python visualizer A simple Python script relying on PyLab, matplotlib and SciPy to display Pareto fronts as 2D plots, where the user can select and visualize up to 4 objectives. The visualization uses x, y, square marker area size and a colormap to encode 4 dimensions. By clicking on solution points users can see the corresponding design variables and interactively experience the effect of design parameters to objective values. This is shown in Figure 3.11.

The Pareto Explorer A web server based visualization implementation using Node.js[6] and other Javascript libraries to bring Pareto front exploring to the browser.

Currently, this tool offers a 3D view using Three.js[7] where 3 objectives can be selected. The view can be rotated, zoomed and panned to inspect

---

[5] http://www.json.org/

[6] http://nodejs.org/

[7] http://threejs.org

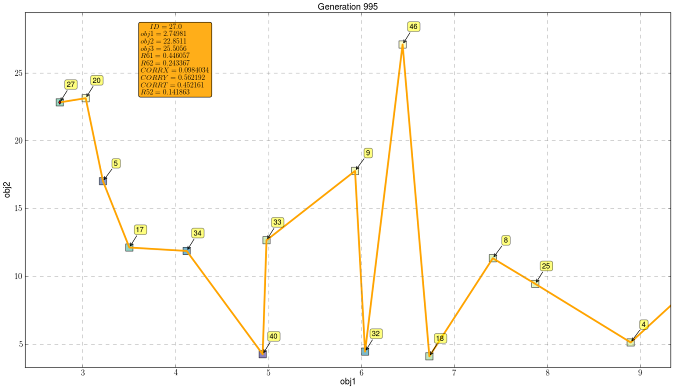


Figure 3.11: Python visualization.

the solution set. All solution points are rendered as particles using shaders. This allows us to render solution points in various ways, simply by applying different shader plugins. We provide a default shader to render the solution as point clouds as shown in Figure 3.12. Further extensions, e.g. using an EWA splatting [**?**] shader to render the solution as a point based surface, are in development.

Additionally we incorporated parallel coordinates visualization based on the D3[8] Javascript library. For every objective we introduce a vertical axis and each solution is displayed as a line with a unique color connecting all its objective values. This is shown in Figure 3.13. The user can highlight individual solutions by hovering over the corresponding row in the data table. The table contains design variable data as well, providing a compact view of the mapping from design space to objective space.

Possible extensions include dimensionality reduction and view plane splitting to increase the number of possibilities to explore the solution set.

---

[8] `http://d3js.org`

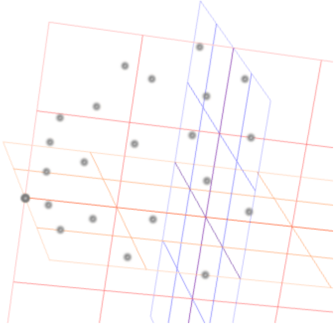
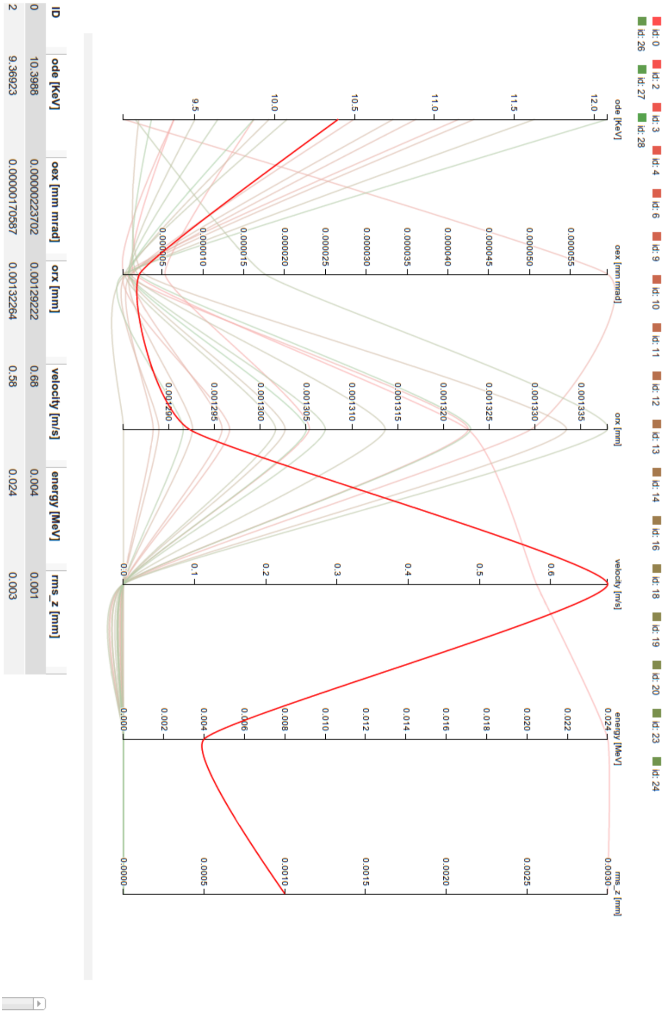


Figure 3.12: Pareto front surface rendered as points.



ID	ode [KeV]	oex [mm mrad]	orx [mm]	velocity [m/s]	energy [MeV]	rms_z [mm]
0	10.3988	0.00000223702	0.00129222	0.68	0.004	0.001
2	9.36923	0.00000170587	0.00132264	0.58	0.024	0.003

Figure 3.13: Pareto points of particle accelerator multi-objective problem displayed as parallel coordinate plot. Hovering with the mouse over data rows (here row 1) highlights the solution in the plot.

Chapter

Forward Solvers for Beam Dynamics 4

> *"Science is what we understand well enough to explain to a computer. Art is everything else we do."*
>
> – Donald Knuth

As a basis for solving the "forward problem" arising in mutli-objective optimization problems of particle acclerators, we make use of OPAL (Object Oriented Parallel Accelerator Library) [**?**]. OPAL is used to tackle the most challenging problems in the field of high precision particle accelerator modeling, including the simulation of high power hadron accelerators and of next generation light sources.

With respect to the optimization framework, we are first and foremost intrested in a low time-to-solution and secondly in simulation accuracy. Clearly these two goals are conflicting and we are interessted in a low accuracy model with fast time-to-solution mimicking the behavior of higher dimensional models well enough to retain optimization relevant characteristics.

Some of the effects can be studied by using a low dimensional model, i.e., envelope equations [**?**, **?**, **?**, **?**, **?**]. These are a set of ordinary differential equations for the second-order moments of a time-dependent particle distribution. In most cases a priori knowledge of critical beam parameters, such as the emittance, is required with the consequence that the envelope equations cannot be used as a self-consistent method. In Section 4.3 we introduce the envelope tracker embedded in OPAL [**?**] and discuss its implementation and parallel efficiency.

Low dimensional simulation codes require a small time to solution, however in some cases the level of detail is insufficient. To that end, OPAL provides a 3D macro particle tracking facility using an FFT-based solver for particle-particle interactions neglecting the influece of beamline boundaries. We extended OPAL with an iterative solver for particle-particle interactions [**?**] described in Section 4.2. This enables more precise simulation of particle accerelators by incorporating analytic or real treatement of the beam pipe boundaries.

4.1 Short Introduction to Particle Accelerator Modeling

In recent years particle accelerators have become 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. Here we provide the fundamental knowledge of the basic principles of particle accelerators to understand the two forward solvers proposed in the next two sections. Of course, this is by far not a complete picture and the interested reader is referred to [**?**].

Described on a very high level, a particle accelerator is a collection of beam line elements emitting electromagnetic fields to accelerate and shape a collection of charged particles. Such a collection of particles is often denoted as a *particle bunch*. Since the particles can reach speeds close to the speed of light, all formulations need to take into account relativistic behavior. The particles speed v is usually expressed by the fraction of the speed of light

$$\beta = \frac{v}{c}, \tag{4.1}$$

it attains. With this the Lorentz factor becomes

$$\gamma = \frac{1}{\sqrt{1-\beta^2}}. \tag{4.2}$$

While particles travel down in a particle accelerator they are subject to two forces. First the force resulting from electromagnetic fields imposed from beam line elements denoted here as *external forces*. Second, the charges particles impose a repulsion force onto each other which we will call *self forces* or *space charge forces*. The external electromagnetic forces can be described by Maxwell's equations

$$\begin{aligned}
\nabla \cdot \mathbf{E} &= \frac{q}{\varepsilon_0} \\
\nabla \cdot \mathbf{B} &= 0 \\
\nabla \times \mathbf{E} &= -\frac{\partial \mathbf{B}}{\partial t} \\
\nabla \times \mathbf{B} &= \mu_0 \mathbf{J} + v_0 \varepsilon_0 \frac{\partial \mathbf{E}}{\partial t},
\end{aligned} \tag{4.3}$$

subject to conformal boundary conditions in the domain of interest $\Omega \subset \mathcal{R}^3$. By chaining beam elements in the correct order and position, the desired beam characteristics can be achieved (see Section 4.1.1). Most of these beam line elements fall in one of two categories: beam guidance and acceleration. Radio frequency cavities can be used to accelerate particles and magnets to focus the beam.

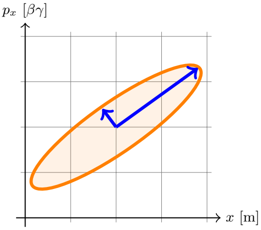


Figure 4.1: 2D phase space of the x position and momenta of particles.

The time evolution of the particles in a plasma with long-range interactions can be described by the collisionless *Vlasov equation*

$$\partial_t f(\mathbf{x}, \mathbf{v}, t) + \mathbf{v} \cdot \nabla_x f(\mathbf{x}, \mathbf{v}, t) + \frac{q}{m_0} (\mathbf{E} + \mathbf{v} \times \mathbf{B}) \cdot \nabla_v f(\mathbf{x}, \mathbf{v}, t) = 0,$$

where $f \in \mathcal{R}^3 \times \mathcal{R}^3$ denotes the phase space density at time t. The phase space of a particle bunch $f(\mathbf{x}, \mathbf{v}, t)$ refers to the density of the particles in the phase space, i.e., the position-velocity $(\mathbf{x}, \mathbf{v})$ space. The phase space of a particle bunch refers to the space spawned by direction and momenta pairs in all space extensions. A simple two dimensional phase space is illustrated in Figure 4.1.

Solving this equations self-consistently in 6 dimensions is a very hard problem. Simulation models use simplifications use e.g. particle in cell (PIC) methods to reduce the computational complexity while retaining self-consistency in the electrostatic approximation.

Using the separability of Hamiltonians the electric and the magnetic fields $\mathbf{E}$ and $\mathbf{B}$ can be computed as superpositions of external fields and self-fields,

$$\mathbf{E} = \mathbf{E}_{\text{ext}} + \mathbf{E}_{\text{self}}, \quad \mathbf{B} = \mathbf{B}_{\text{ext}} + \mathbf{B}_{\text{self}}, \tag{4.4}$$

as shown in Figure 4.2. If $\mathbf{E}$ and $\mathbf{B}$ are known, then each particle can be propagated according to the equation of motion for charged particles in an electromagnetic field,

$$\frac{d\mathbf{x}(t)}{dt} = \mathbf{v}, \quad \frac{d\mathbf{v}(t)}{dt} = \frac{q}{m_0} (\mathbf{E} + \mathbf{v} \times \mathbf{B}).$$

These equations can be integrated in time with any numerical integration schema providing the required accuracy, i.e. leap-frog or RK-4 [?].

Then, the particle push $\mathbf{E}_{\text{self}}$ and $\mathbf{B}_{\text{self}}$ need to be updated. To that end, we change the coordinate system into the one moving with the particles. By means of the appropriate *Lorentz transformation* [?] we arrive at a (quasi-)

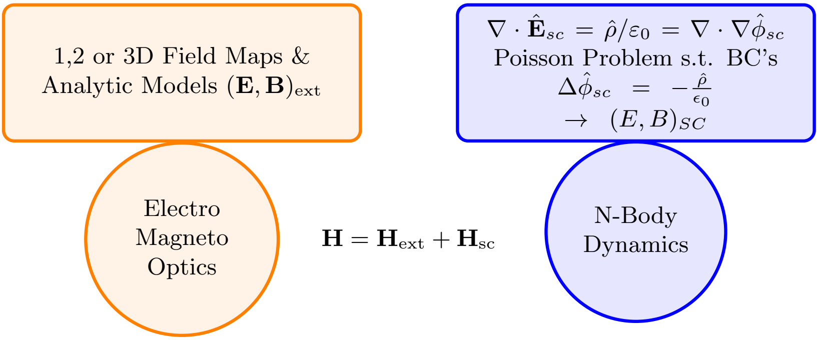

1,2 or 3D Field Maps &
Analytic Models $(\mathbf{E},\mathbf{B})_{\mathrm{ext}}$
$\nabla\cdot\hat{\mathbf{E}}_{sc}=\hat{\rho}/\varepsilon_0=\nabla\cdot\nabla\hat{\phi}_{sc}$
Poisson Problem s.t. BC's
$\Delta\hat{\phi}_{sc} = -\frac{\hat{\rho}}{\epsilon_0}$
$\rightarrow (E,B)_{SC}$
Electro
Magneto
Optics
$\mathbf{H}=\mathbf{H}_{\mathrm{ext}}+\mathbf{H}_{\mathrm{sc}}$
N-Body
Dynamics

Figure 4.2: Separability of the Hamiltonians allows disjoint computation of external and space-charge fields. The Poisson problem is solved in the co-moving frame.

static approximation of the system in which the transformed magnetic field becomes negligible, $\hat{\mathbf{B}}\approx\mathbf{0}$. The transformed electric field is obtained from

$$\hat{\mathbf{E}}=\hat{\mathbf{E}}_{\mathrm{self}}=-\nabla\hat{\phi}, \tag{4.5}$$

where the electrostatic potential $\hat{\phi}$ is the solution of the *Poisson problem*

$$-\Delta\hat{\phi}(\mathbf{x})=\frac{\hat{\rho}(\mathbf{x})}{\varepsilon_0}, \tag{4.6}$$

equipped with appropriate boundary conditions. Here, $\hat{\rho}$ denotes the spatial charge density and ε_0 is the dielectric constant. By means of the inverse Lorentz transformation the electric field $\hat{\mathbf{E}}$ can then be transformed back to yield both the electric and the magnetic fields in (4.4).

4.1.1 Objectives in Beam Dynamic Optimization Problems

Multi-objective optimization problems described here will mainly employ a combination of the following objectives:

- transverse and longitudinal bunch size (see Figure 4.3),
- 6D phase space,
- energy spread (left in Figure 4.4),

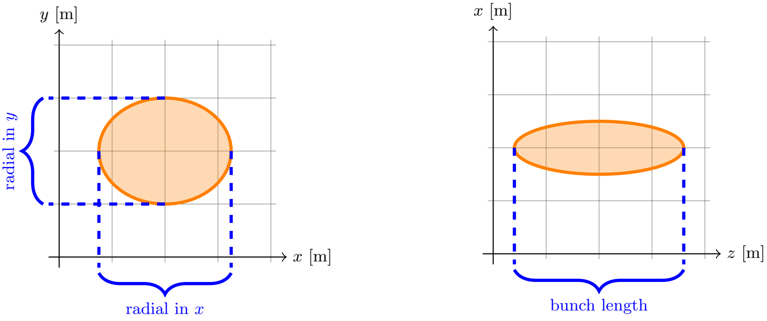


Figure 4.3: Transversal and longitudinal particle bunch size.

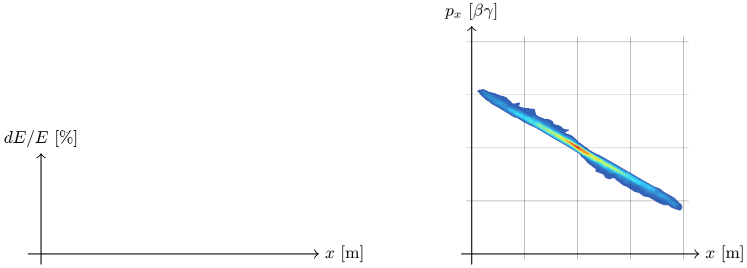


Figure 4.4: LEFT energy spread, and RIGHT beam emittance.

- emittance (right in Figure 4.4).

The bunch size as shown in Figure 4.3 describes the expansion of the particle bunch (minimal to maximal size) in all coordinates. Usually, we are interested in a minimal bunch size, effectively producing more focused beams. These extensions are typically compared using the root-mean-square (rms) size along a given direction over all n particles, i.e.,

$$\frac{1}{n}\sqrt{\sum_{i=1}^{n}|p_{x,i}-\overline{p}_x|}, \tag{4.7}$$

where $\overline{p}_x$ is the mean of all particle positions in x direction.

The energy spread

$$\delta E = \delta p_z m_0 \beta \tag{4.8}$$

is a measure of the energy distribution in the particle bunch. The difference in energy between particles is proportional to the energy spread. Therefore, by minimizing the energy spread we can ensure that the particles have roughly equal energy.

Similarly to the energy spread, the particle bunch extension in phase space can be expressed using the notation of the beam emittance. Minimizing the emittance enforces the particles to be confined to a small volume in configuration space and particles to have small momenta spread.

4.2 Iterative Space-Charge Solver

The solver described in this section is based on [**?**] and its improvements [**?**].

The Poisson problem (4.6) discretized by finite differences can be efficiently solved on a rectangular grid by a Particle-In-Cell (PIC) approach [**?**]. The right hand side in (4.6) is discretized by sampling the particles at the grid points. In (4.5), $\hat{\phi}$ is interpolated at the particle positions from its values at the grid points. We also note that the FFT-based Poisson solvers and similar approaches [**?**, **?**] are restricted to box-shaped or open domains.

Serafini et al. [**?**] report on a state-of-the-art conventional FFT-based algorithm for solving the Poisson equation with 'infinite-domain', i.e., open boundary conditions for large problems in accelerator modeling. The authors show improvements in both accuracy and performance, by combining several techniques: the method of local corrections, the James algorithm, and adaptive mesh refinement.

However with the quest of high intensity, high brightness beams together with ultra low particle losses, there is a high demand to consider the true geometry of the beam-pipe in the numerical model. This assures that the image charge components are taken properly into account. This results in a more exact modeling of the non-linear beam dynamics which is indispensable for the design of next generation of particle accelerators.

In recent years, precise beam dynamics simulations in the design of high-current low-energy hadron machines as well as of 4th generation light sources have become a very important research topic. Hadron machines are characterized by high currents and hence require excellent control of beam losses and/or keeping the emittance (a measure of the phase space) of the beam in narrow ranges. This is a challenging problem which requires the accurate modeling of the dynamics of a large ensemble of macro or real particles subject to complicated external focusing, accelerating and wake-fields, as well as the self-fields caused by the Coulomb interaction of the particles. In general, the geometries of particle accelerators are large and complicated. The discretization of the computational domain is time dependent due to the relativistic nature of the problem. Both phenomena have a direct impact on the numerical method used to solve the problem.

In a related paper by Pöplau et al. [**?**], an iterative solver, preconditioned by geometric multigrid, is used to calculate space-charge forces. The authors employ a mesh with adaptive spacings to reduce the workload of the BiCGStab solver used to solve the nonsymmetric system arising from quadratic extrapolation at the boundary. The geometric multigrid solver used in their approach is much more sensitive to anisotropic grids arising in beam dynamic simulations (e.g. special coarsening operators have to be defined). With smoothed aggregation-based algebraic multigrid (AMG) preconditioning as used in here the aggregation smoother takes care of anisotropies and related issues and leads to a robustness superior to geometric multigrid, see [**?**] for a discussion. The preconditioner easily adapts to the elongation of the computational domain that happens during our simulation.

In Section 4.2.1 we describe how the Poisson equation on a 'general' domain $\Omega \subset \mathbb{R}^3$ can be solved by finite differences and the PIC approach. We treat the boundary in three different ways, by constant, by linear, and by quadratic extrapolation, the latter being similar to the approach of McCorquodale *et al.* [**?**]. The system of equations is solved by the conjugate gradient algorithm preconditioned by smoothed aggregation-based algebraic multigrid (SA-AMG) [**?**, **?**], see Section 4.2.2. The preconditioned conjugate gradient (PCG) algorithm is also used if the system is 'mildly' nonsymmetric. In Section 4.3.1 we deal with details of the implementation, in particular its parallelization. In Section 4.2.4 we report on numerical experiments including a physical application from beam dynamics and in 5.1.1 the parallel performance of the proposed methods.

A very attractive approach to solve the Poisson equation is by a fast elliptic direct solver based on FFT's [**?**]. However, this approach is limited to simple geometries such as boxes and cylinders. Other fast and accurate methods are known [**?**] but they are also limited to simple geometries. Because ρ is varying slowly in time (4.6), it is desirable to use solutions from previous time step(s). In the sequel of this section we consider iterative solvers for (4.6) applies on general but simply connected geometries $\Omega \subset \mathbb{R}^d$.

4.2.1 The Discretization

In this section we discuss the solution of the Poisson equation in a domain $\Omega \subset \mathbb{R}^3$ as indicated in Figure 4.5. The boundary of the domain is composed of two parts, a curved, smooth surface Γ_1 and two planar portions at $z = -d$ and $z = +d$ that form together Γ_2. In physical terms Γ_1 forms the casing of the pipe, while Γ_2 is the open boundary at the inlet and outlet of the beam pipe, respectively. The centroid of the particle bunch is at the origin of the coordinate system. The Poisson problem that we are going to solve is given by

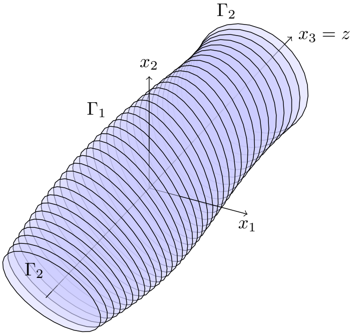


Figure 4.5: Sketch of a typical domain

$$\begin{aligned} -\Delta\phi &= \frac{\rho}{\epsilon_0} \text{ in } \Omega, \\ \phi &= g \equiv 0 \text{ on } \Gamma_1, \\ \frac{\partial\phi}{\partial\mathbf{n}} + \frac{1}{d}\phi &= 0 \text{ on } \Gamma_2. \end{aligned} \tag{4.9}$$

The parameter d in the Robin boundary condition (third equation in (4.9)) denotes the distance of the charged particles to the boundary [**?**]. It is half the extent of Ω in z-direction. Notice that the Robin boundary condition applies only on the planar paraxial portions of the boundary.

We discretize (4.9) by a second order finite difference scheme defined on a rectangular lattice (grid)

$$\Omega_h := \{\mathbf{x} \in \Omega \cup \Gamma_2 \mid x_i/h_i \in \mathbb{Z} \text{ for } i = 1, 2, 3\},$$

where h_i is the grid spacing and $\mathbf{e}_i$ the unit vector in the i-th coordinate direction. The grid is arranged in a way that the two portions of Γ_2 lie in grid planes. A lattice point is called an *interior* point if all its direct neighbours are in Ω. All other grid points are called *near-boundary* points. On the interior points $\mathbf{x}$ we approximate $\Delta u(\mathbf{x})$ by the well-known 7-point difference stencil

$$-\Delta_h u(\mathbf{x}) = \sum_{i=1}^{3} \frac{-u(\mathbf{x}-h_i\mathbf{e}_i) + 2u(\mathbf{x}) - u(\mathbf{x}+h_i\mathbf{e}_i)}{h_i^2}. \tag{4.10}$$

At grid points near the boundary we have to take the boundary conditions in (4.9) into account. To explain the schemes on the Dirichlet (or PEC) boundary Γ_1 let $\mathbf{x}$ be a near-boundary point. Let $\mathbf{x}' := \mathbf{x} - h_i\mathbf{e}_i$ for some i be outside Ω and let $\mathbf{x}^* := \mathbf{x} - sh_i\mathbf{e}_i$, $0 < s \leq 1$, be the boundary point between $\mathbf{x}$ and $\mathbf{x}'$ that is closest to $\mathbf{x}$, cf. Figure 4.6. If $s = 1$, i.e., if $\mathbf{x}' \in \partial\Omega$

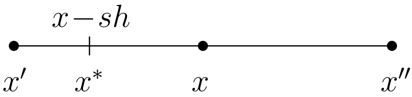


Figure 4.6: 1-dimensional sketch of a near-boundary point x

then $u(\mathbf{x}')$ in (4.10) is replaced by the prescribed boundary value. Otherwise, we proceed in one of the three following ways [**?**, **?**]

1. In *constant extrapolation* the boundary value prescribed at $\mathbf{x}-sh_i\mathbf{e}_i \in \Gamma_1$ is assigned to $u(\mathbf{x}')$,

$$u(\mathbf{x}') = u(\mathbf{x} - h_i\mathbf{e}_i) := g(\mathbf{x}^*). \tag{4.11}$$

2. In *linear extrapolation* the value at $\mathbf{x}'$ is obtained by means of the values u at $\mathbf{x}$ and at $\mathbf{x} - sh_i\mathbf{e}_i$,

$$u(\mathbf{x}') := (1 - \frac{1}{s})\, u(\mathbf{x}) + \frac{1}{s}\, g(\mathbf{x}^*). \tag{4.12}$$

3. *Quadratic extrapolation* amounts to the Shortley-Weller approximation [**?**, **?**,**?**]. If $\mathbf{x}'' := \mathbf{x}+h_i\mathbf{e}_i \in \Omega_h$ then the value $u(\mathbf{x}')$ is obtained by quadratic interpolation of the values of u at $\mathbf{x}$, $\mathbf{x}''$, and the boundary point $\mathbf{x}^*$,

$$u(\mathbf{x}') := \frac{2(s-1)}{s} u(\mathbf{x}) - \frac{s-1}{s+1} u(\mathbf{x}'') + \frac{2}{s(s+1)} g(\mathbf{x}^*). \tag{4.13}$$

If $\mathbf{x}'' \notin \Omega_h$ then let $\mathbf{x}^{**} := \mathbf{x} + t h_i \mathbf{e}_i$, $0 < t \leq 1$, be the boundary point between $\mathbf{x}$ and $\mathbf{x}''$ that is closest to $\mathbf{x}$. Then, similarly as before, we get

$$\begin{aligned} u(\mathbf{x}') &:= \frac{(s-1)(t+1)}{st} u(\mathbf{x}) + \frac{(t+1)}{(s+t)s} g(\mathbf{x}^*) - \frac{(s-1)}{(s+t)t} g(\mathbf{x}^{**}), \\ u(\mathbf{x}'') &:= \frac{(s+1)(t-1)}{st} u(\mathbf{x}) - \frac{(t-1)}{(s+t)s} g(\mathbf{x}^*) + \frac{(s+1)}{(s+t)t} g(\mathbf{x}^{**}). \end{aligned} \tag{4.14}$$

In all extrapolation formulae given above we set $g(\mathbf{x}^*) = g(\mathbf{x}^{**}) = 0$ according to (4.9). The value on the right side of (4.11)–(4.14) substitutes $u(\mathbf{x} \pm h_i \mathbf{e}_i)$ in (4.10).

Let us now look at a grid point $\mathbf{x}$ on the open boundary Γ_2. If $\mathbf{x}$ is located on the inlet of the beam pipe then $\mathbf{x}'' := \mathbf{x} + h_3 \mathbf{e}_3 \in \Omega$ and $\mathbf{x}' := \mathbf{x} - h_3 \mathbf{e}_3 \notin \Omega$. The Robin boundary condition is approximated by a central difference,

$$-\frac{u(\mathbf{x}'') - u(\mathbf{x}')}{2h_3} + \frac{1}{d} u(\mathbf{x}) = 0,$$

or

$$u(\mathbf{x}') = u(\mathbf{x}'') - \frac{2h_3}{d} u(\mathbf{x}). \tag{4.15}$$

The same formula holds on the outlet boundary portion if $\mathbf{x}'$ denotes the virtual grid point outside Ω.

Notice that some lattice points may be close to the boundary with regard to more than one coordinate direction. Then, the procedures (4.11)–(4.15) must be applied to all of them.

The finite difference discretization just described leads to a system of equations

$$A\mathbf{x} = \mathbf{b}, \tag{4.16}$$

where $\mathbf{x}$ is the vector of unknown values of the potential and $\mathbf{b}$ is the vector of the charge density interpolated at the grid points. The Poisson matrix A is an M-matrix irrespective of the boundary treatment [**?**]. Constant and linear extrapolation lead to a *symmetric* positive definite A, while quadratic extrapolation yields a *nonsymmetric* but still positive definite Poisson matrix.

Notice that the boundary extrapolation introduces large diagonal elements in A if s or t gets close to zero. In order to avoid numerical difficulties it is advisable to scale the system matrix. This can be achieved by one step of balancing [**?**].

4.2.2 The Solution Method

In this section we discuss the solution of (4.16), the Poisson problem (4.9) discretized by finite differences as described in the previous section.

The conjugate gradient algorithm

The matrix A in (4.16) is symmetric positive definite if the boundary conditions are treated by constant or linear extrapolation. For symmetric positive definite systems, the conjugate gradient (CG) algorithm [**?**, **?**] provides a fast and memory efficient solver. The CG algorithm minimizes the quadratic functional

$$\varphi(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{x}^T\mathbf{b} \tag{4.17}$$

in the Krylov space that is implicitly constructed in the iteration. In the k-th iteration step the CG algorithm minimizes the quadratic functional φ along a search direction $\mathbf{d}_k$. The search directions turn out to by pairwise conjugate, $\mathbf{d}_k^T A\mathbf{d}_j = 0$ for all $k \neq j$, and $\varphi(\mathbf{x})$ is minimized in the whole k-dimensional Krylov space.

If we use the quadratic extrapolation (4.13) at the boundary then A in (4.16) is not symmetric positive definite anymore. Nevertheless, A is an M-matrix [**?**] and the solution of (4.16) is still a minimizer of $\varphi(\mathbf{x})$. The CG algorithm can be used to solve (4.16). It is known to converge [**?**]. However, the finite termination property of CG is lost as the search directions are not mutually conjugate any more. Only consecutive search directions are conjugate, $\mathbf{d}_k^T A\mathbf{d}_{k-1} = 0$, reflecting the fact that $\varphi(\mathbf{x})$ is minimized only locally. Young & Jea [**?**] investigated generalizations of the conjugate gradient algorithm for nonsymmetric positive definite matrices, in which conjugacy is enforced among $\mathbf{d}_k, \ldots, \mathbf{d}_{k-s}$ for some $s > 1$. We do not pursue these approaches here. As GMRES [**?**], they consume much more memory space than the straightforward CG method that turned out to be extremely efficient for our application. Although A is nonsymmetric it is so only 'mildly', i.e., there are deviations from symmetry only at some of the boundary points. Therefore, one may hope that the conjugate gradient method still performs reasonably well. This is what we actually did observe in our experiments.

Methods that are almost as memory efficient as CG like, e.g., the stabilized biconjugate gradient (BiCGStab) method [**?**] could be used for solving (4.16) as well. However, when considering computational costs we note that BiCGStab requires two matrix-vector products per iteration step, in contrast to CG that requires only one.

Preconditioning

To improve the convergence behavior of the CG methods we precondition (4.16). The preconditioned system has the form

$$M^{-1}A\mathbf{x} = M^{-1}\mathbf{b},$$

where the positive definite matrix M is the preconditioner. A good choice of the preconditioner reduces the condition of the system and thus the number of steps the iterative solver takes until convergence [**?**, **?**]. Using preconditioning is inevitable for systems originating in finite difference discretizations of 2nd order PDE's since their condition number increases as h^{-2} where h is the mesh width [**?**].

Here we are concerned with multilevel preconditioners. Multigrid or multilevel preconditioners are the most effective preconditioners, in particular for the Poisson problem [**?**, **?**]. Multigrid methods make use of the observation that a smooth error on a fine grid can be well approximated on a coarser grid. When this coarser grid is chosen to be a sufficient factor smaller than the fine grid the resulting problem is quite smaller and thus much cheaper to solve. We can continue coarsening the grid until we arrive at a problem size that can be solved cheaply by a direct solver. This observation suggests an algorithm similar to Algorithm 5.

Algorithm 5 Multigrid V-cycle

```
 1: procedure MULTIGRIDSOLVE($A_\ell$, $\mathbf{b}_\ell$, $\mathbf{x}_\ell$, $\ell$)
 2:     if $\ell$ = maxLevel − 1 then
 3:         DirectSolve $A_\ell \mathbf{x}_\ell = \mathbf{b}_\ell$
 4:     else
 5:         $\mathbf{x}_\ell \leftarrow S_\ell^{pre}(A_\ell, \mathbf{b}_\ell, 0)$            ▷ presmoothing
 6:         $\mathbf{r}_\ell \leftarrow \mathbf{b}_\ell - A_\ell \mathbf{x}_\ell$            ▷ calculate residual
 7:         $\mathbf{b}_{\ell+1} \leftarrow R_\ell \mathbf{r}_\ell$            ▷ restriction
 8:         $\mathbf{v}_{\ell+1} \leftarrow \mathbf{0}$
 9:         MultiGridSolve($A_{\ell+1}$, $\mathbf{b}_{\ell+1}$, $\mathbf{v}_{\ell+1}$, $\ell+1$)
10:         $\mathbf{x}_\ell \leftarrow \mathbf{x}_\ell + P_\ell \mathbf{v}_{\ell+1}$            ▷ coarse grid correction
11:         $\mathbf{x}_\ell \leftarrow S_\ell^{post}(A_\ell, \mathbf{b}_\ell, \mathbf{x}_\ell)$
12:     end if
13: end procedure
```

The procedure starts on the finest level ($\ell = 0$) and repeatedly coarsens the grid until the coarsest level is reached (`maxLevel`), on which a direct solver is used to solve the resulting problem. On all other levels ℓ the algorithm starts by presmoothing (S_ℓ^{pre}) the problem to damp high frequency components of the error (line 5). Subsequently, the fine grid on level ℓ can be restricted with the restriction operator R_ℓ to a coarser grid on level $\ell + 1$ (line 7). This essentially "transfers" the low frequency components on the fine grid to high frequency components on the coarse grid. After the recursion has reached the coarsest level and used the direct solver to solve the coarse level problem the solution can be prolongated back to a finer grid. This is achieved with the prolongation operator P_ℓ (line 10). Often a postsmoother S_ℓ^{post} is used to remove artifacts caused by the prolongation operator. Usually these operators (for every level ℓ) are defined in a setup phase preceding the execution of the actual multigrid algorithm. Lastly, A_ℓ denotes the matrix of the discretized system in level ℓ.

The performance of multigrid methods profoundly depends on the choices and interplay of the smoothing and restriction operators. To ensure that the resulting preconditioner is symmetric we use the same pre- and postsmoother S_ℓ and the restriction operator is chosen to be the transpose of the prolongation operator $R_\ell = P_\ell^T$. This leaves us with two operators, P_ℓ and S_ℓ, that have to be defined for every level.

Prolongation Operator P_ℓ Aggregation based methods cluster the fine grid unknowns to aggregates (of a specific form, size, etc.) as representation for the unknowns on the coarse grid. First, each vertex of G_ℓ, the adjacency graph of A_ℓ, is assigned to one of the pairwise disjoint aggregates. Then, a tentative prolongation operator matrix is formed where matrix rows correspond to vertices and matrix columns to aggregates. A matrix entry (i, j) has a value of 1 if the i^{th} vertex is contained in the j^{th} aggregate and 0 otherwise. This prolongation operator basically corresponds to a piecewise constant interpolation operation. To improve robustness one can additionally smooth the tentative prolongation operator. This is mostly done with a damped Jacobi smoother. In general applying a smoother results in better interpolation properties opposed to the piecewise constant polynomials and improves convergence properties. Tuminaro & Tong [**?**] propose various strategies how to parallelize this process. The simplest strategy is to have each processor aggregate its portion of the grid. This method is called “decoupled” since the processors act independently of each other. Usually the aggregates are formed as cubes of 3^d vertices in d dimensions. Since the domains under consideration are close to rectangular the decoupled scheme seems to be an appropriate strategy. In the interior of our domain we get regular cubes covering the specified number of vertices. Only a few aggregates near subgrid interfaces and domain boundary contain fewer vertices resulting in a non-optimal aggregate size. The overhead introduced is small. “Coupled” coarsening strategies, e.g. Parmetis [**?**], introduce interprocessor communication and are often needed in the presence of highly irregular domains. In our context applying uncoupled methods only restrict the size of the coarsest problem. This is due to the fact that on the coarsest level each processor must at least hold one degree of freedom.

Smoothing operator S_ℓ As advised in [?] we choose a Chebyshev polynomial smoother. The choice is motivated by the observation that polynomial smoothers tend to minimize the use of global reductions and therefore perform better in parallel than Gauss-Seidel smoothers. Advantages are, e.g., that polynomial smoothers do not need special matrix kernels and formats for optimal performance and, generally, polynomial methods can profit of architecture optimized matrix vector products. Nevertheless, routines are needed that yield bounds for the spectrum. But these are needed by the prolongator smoother anyway.

Coarse level solver The employed coarse level solver (Amesos-KLU) ships the coarse level problem to node 0 and solves it there by means of an LU factorization. Once the solution has been calculated it is broadcasted to all nodes. To gather and scatter data a substantial amount of communication is required. Moreover the actual solve can be expensive if the matrix possesses a large amount of nonzeros per row.

An alternative is to apply a few steps of an iterative solver (e.g. Gauss-Seidel) at the coarsest level. A small number of iteration steps decreases the quality of the preconditioner and thus increases the PCG iteration count. A large number of iteration steps increases the time for applying the AMG preconditioner. We found three Gauss-Seidel iteration steps to be a good choice for our application.

Cycling method We observed a tendency that timings for the W-cycle are 10% – 20% slower compared with the V-cycle.

4.2.3 Implementation Details

The multigrid preconditioner and iterative solver are implemented with the help of the Trilinos framework [?, ?]. Trilinos provides state-of-the-art tools for numerical computation in various packages. Aztec, for example, provides iterative solvers and ML [?] multilevel preconditioners. By means of ML, we created our smoothed aggregation-based AMG preconditioner. The essential parameters of the preconditioner discussed above are listed in Table 4.1.

To embed the solver in the physical simulation code (OPAL [?]) we utilized the Independent Parallel Particle Layer (IP^{2}L [?]). This library is an object-oriented framework for particle based applications in computational science designed for the use on high-performance parallel computers. In the context of this paper IP^{2}L is only relevant because OPAL uses IP^{2}L to represent and interpolate the particles at grid points with a charge conserving Cloud-in-Cell area weighting scheme. ML requires an `Epetra_Map` handling the parallel

name	value
preconditioner type	MGV
smoother	pre and post
smoother type	Chebyshev
aggregation type	Uncoupled
coarse level solver	Amesos-KLU
maximal coarse level size	1000

Table 4.1: Parameters for multilevel preconditioner ML.

decomposition to create parallel distributed matrices and vectors. To avoid additional communication the `Epetra_Map` and the IP^{2}L field are determined to have the same parallel decomposition. In this special case the task of converting the IP^{2}L field to an `Epetra_Vector` is as simple as looping over local indices and assigning values.

A particle based domain decomposition technique based on recursive coordinate bisection is used (see Section 4.2.4) to parallelize the computation on a distributed memory environment. One, two and three-dimensional decompositions are available. For problems of beam dynamics with highly nonuniform and time dependent particle distributions, a dynamic load balancing is necessary to preserve the parallel efficiency of the particle integration scheme. Here, we rely on the fact that IP^{2}L attains a good load balance of the data.

We use the solution of one time step as the initial guess for the next time step.

4.2.4 Numerical Experiments

In this section we discuss various numerical experiments and results concerning different variants of the preconditioner and comparisons of solvers and boundary extrapolation methods. Unless otherwise stated the measurements are done on a tube embedded in a rectangular equidistant 256 × 256 × 256 grid. This is a common problem size in beam dynamics simulations.

Most of the computations were performed on the Cray XT4 cluster of the Swiss Supercomputing Center (CSCS) in Manno, Switzerland. Computations up to 512 cores were conducted on the XT4 cluster (Buin) with 468 AMD dual core Opteron 2.6 GHz processors and a total of 936 GB DDR RAM on a 7.6 GB/s interconnect bandwidth network. Larger computations were performed on the XT3 cluster (Palu) with 1692 AMD dual core Opteron 2.6 GHz processors and a total of 3552 GB DDR RAM interconnected with a 6.4 GB/s interconnect bandwidth network.

Comparison of Extrapolation Schemes

For validation and comparison purposes we applied our solver to a problem with a known analytical solution. We solved the Poisson problem with homogeneous Dirichlet boundary conditions ($\phi = 0$) on a cylindrical domain $\Omega = \{|r| < \frac{1}{2}\} \times (-\frac{1}{2}, \frac{1}{2})$. The axisymmetric charge density

$$\rho = -\left(\pi^2 r^2 - \frac{\pi^2}{4} - 4\right) \sin(\pi(z - 0.5))$$

gives rise to the potential distribution

$$\phi(r, \theta, z) = \left(\frac{1}{4} - r^2\right) \sin(\pi(z - 0.5)).$$

The charge density in our test problem is smoother than in real problems. Nevertheless, it is very small close to the boundary. This reflects the situation in particle accelerators where most particles are close to the axis of rotation and, thus, charge densities are very small close to the boundary.

We measure the error on the grid Ω_h with mesh spacing h in the discrete norms

$$\|e_h\|_2 = \|\hat{\phi}_h - \phi\|_2 = \sqrt{h^3 \sum_{i \in \Omega_h} |(\hat{\phi}_{i,h} - \phi_i)|^2},$$

$$\|e_h\|_\infty = \|\hat{\phi}_h - \phi\|_\infty = \max_{i \in \Omega_h} |\hat{\phi}_{i,h} - \phi_i|,$$

where $\hat{\phi}_h$ is the approximation of the solution ϕ on Ω_h, and e_h denotes the corresponding error. The convergence rate is approximately

$$r = \log_2\left(\frac{\|e_{2h}\|}{\|e_h\|}\right).$$

We solved the Poisson equation with the boundary extrapolation methods introduced earlier. The errors are listed in Tables 4.2–4.4. The numbers confirm the expected convergence rates, i.e., linear for the constant extrapolation and quadratic for the linear and quadratic extrapolation [**?**]. The results obtained with the linear extrapolation scheme are more accurate than constant extrapolation by two orders of magnitude. Quadratic extrapolation is again more accurate than linear extrapolation, but for both norms by only a factor 3 to 5. It evidently does not make sense to use constant extrapolation as the

h	$\|e_h\|_2$	r	$\|e_h\|_\infty$	r	$\|e_h\|_\infty/\|\phi\|_\infty$
1/64	2.162×10^{-3}		7.647×10^{-3}		3.061×10^{-2}
1/128	1.240×10^{-3}	0.80	4.153×10^{-3}	0.88	1.662×10^{-2}

Table 4.2: Solution error for constant extrapolation, $d = 3$.

h	$\|e_h\|_2$	r	$\|e_h\|_\infty$	r	$\|e_h\|_\infty/\|\phi\|_\infty$
1/64	2.460×10^{-5}		6.020×10^{-5}		2.410×10^{-4}
1/128	6.226×10^{-6}	1.98	1.437×10^{-5}	2.07	5.751×10^{-5}

Table 4.3: Solution error for linear extrapolation, $d = 3$.

h	$\|e_h\|_2$	r	$\|e_h\|_\infty$	r	$\|e_h\|_\infty/\|\phi\|_\infty$
1/64	5.581×10^{-6}		1.689×10^{-5}		6.761×10^{-5}
1/128	1.384×10^{-7}	2.01	4.550×10^{-6}	1.89	1.820×10^{-5}

Table 4.4: Solution error for quadratic extrapolation, $d = 3$.

cost of solving with linear boundary extrapolation is equal. In contrast, the quadratic boundary treatment entails the drawback that discretization matrices lose symmetry. They are still positive definite, however. In the particular setting of this test problem as well as in others we have investigated (e.g. in real particle simulations) the system matrices were just ‘mildly’ nonsymmetric such that PCG could be applied safely and without performance loss.

ML variations

Multilevel preconditioners are highly sophisticated preconditioners. Not surprisingly, their construction is very time consuming. To build an SA-AMG preconditioner (1) the “grid” hierarchy (including aggregation and construction of tentative prolongator), (2) the final grid transfer operators (smoothing the prolongators), and (3) the coarse grid solver have to be set up.

In the following subsections we investigate various variants of the preconditioner or more precisely variants of the construction of the preconditioner when solving a sequence of related Poisson problems.

The default variant builds a new preconditioner in every time step. In the sequel we will investigate how costly this is. Other variants reuse portions of previous preconditioners.

We compare with the FFT-based Poisson solver [**?**] that OPAL (version 1.1.5) provides for open-space boundary conditions. The FFT kernel is based on a variant of the FFTPACK library [**?**, **?**].

Reusing the aggregation hierarchy

Since the geometry often changes only slowly in time, the *topology* of the computational grid does not or only rarely alters. Therefore, it makes sense to reuse the aggregation hierarchy and in particular the tentative prolongators for some or all iterations. Only smoothed prolongators and other components of the hierarchy, like smoothers and coarse solver, are recomputed [**?**, p.16]. This leads to a preconditioner variation in which the aggregation hierarchy is kept as long as possible. The numbers in Table 4.5 show that this minor change in the set up phase reduces construction times by approximately 30%.

cores	average of 10 time steps (s)	one time step (s)
16	6.98	10.3
32	4.36	6.44
64	2.38	3.48
128	1.34	1.91
256	0.735	1.04
512	0.518	0.745

Table 4.5: Preconditioner construction times. Left: average cost of 10 time steps reusing the hierarchy of the first iteration step. Right: cost of a time step when building the whole preconditioner.

Reusing the aggregation hierarchy is a feature provided by ML. It is intended for use in nonlinear systems solving. In our simulations it reduced the time per AMG solve in a simulation run by approximately 25%, see Table 4.6.

Reusing the preconditioner

We can be more aggressive by reusing the preconditioner of the first iteration throughout a whole simulation. Although the iteration count increased, the time-to-solution reduced considerably. To counteract an excessive increase of the iteration steps the preconditioner can be recomputed once the number of iterations exceeds a certain threshold. (This was not necessary in our experiments, though.)

Applying this approach to a cylinder-shaped beam pipe, a single preconditioner could be used throughout the entire simulation without any severe impact on the number of iteration steps.

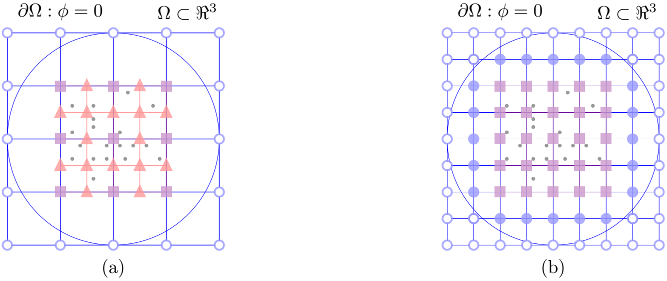


Figure 4.7: Sketch of the test cases with equal number of mesh points (a), and equal mesh resolution (b), respectively. Displayed are the shared (square), FFT only (triangle), and AMG only (filled circle) mesh points on a cross section of the grid plane. Illustrative particles (gray) inside the FFT domain denote the charge density.

In the following we compare accuracy and performance of the SA-AMG preconditioned conjugate gradient algorithm with a FFT-based solver. The principle difficulty in the comparison stems from the fact that the CG-based and the FFT-based solvers use different discretizations of the tube-shaped domain Ω. Although both use finite-difference discretizations, with PCG the whole domain is discretized while with FFT just a rectangular domain along the center line of the cylinder is taken into account. In some applications (beam pipes of light sources) the rectangular domain is contained in Ω. In our application the rectangular domain has a similar volume such that the boundaries can be intertwined. Therefore, we expect that the more accurate boundary treatment in the iterative solver has a noticeable positive effect on solution quality.

We came up with two test cases as illustrated in Figure 4.7.

- The first test case (a) displayed on the left corresponds to a situation where both methods have about the same number of unknowns. In the FFT-based approach only the close vicinity of the particle bunch is discretized. In contrast, in the PCG approach the grid extends to the whole domain, entailing a coarser grid than with the FFT-based approach.

- The second test case (b) displayed on the right corresponds to a situation where both methods have the same mesh resolution in the vicinity of the particles. This results in a higher number of mesh points for the PCG approach.

We consider a cylindrical tube of radius $r = 0.001\,\mathrm{m}$. The FFT-based solver used a grid with $128^2 \cdot 256 = 4,194,304$ nodes. The grid with a similar number of points but coarser resolution consisted of 3,236,864 grid points. To obtain the same resolution with PCG, 5,462,016 grid points were required. They are embedded in a 166 × 166 × 256 grid that coincides in the middle of the region of simulation with the 128 × 128 × 256 grid for the FFT-based solver. The boundary conditions were implemented by linear extrapolation (4.12). In Table 4.6 we give execution times for the first and second iteration in a typical simulation. Since we start with quite a good vector the number of steps in the second iteration is about 30% smaller than in the first iteration. This accounts for the reduced execution time in the runs where the complete preconditioners are recomputed for each iteration. The savings from this straightforward mode to the two cases reusing either hierarchy or the entire preconditioner amounts to approximately 20% and 40% respectively.

solver	reusing	mesh size	mesh points	first [s]	second [s]
FFT	—	128 × 128 × 256	4,194,304	12.3	—
AMG	—	128 × 128 × 256	3,236,864	49.9	42.2
AMG	hierarchy	128 × 128 × 256	3,236,864	—	35.5
AMG	preconditioner	128 × 128 × 256	3,236,864	—	28.2
AMG	—	166 × 166 × 256	5,462,016	81.8	71.2
AMG	hierarchy	166 × 166 × 256	5,462,016	—	60.4
AMG	preconditioner	166 × 166 × 256	5,462,016	—	43.8

Table 4.6: Simulation timings of one solve in the first and second time step, respectively, with $r = 0.001\,\mathrm{m}$. Equal number of mesh points (above) and equal mesh spacings (below) for FFT and AMG.

We inspected the PCG results for the two mesh sizes and found no physically relevant differences. This behaviour depends on the ratio of bunch and boundary radius. Based on the differences of the FFT and the iterative solver (e.g. boundary treatment and implementation details) it is hard to come up with a ‘fair’ comparison. There certainly exists a correlation between the number of performed (CG) iterations and the time to solution. Determining the right stopping criteria and tolerance therefore has an important impact on the performance. While still achieving the same accuracy of the physics of a simulation it could be possible to execute fewer CG iterations by using a higher tolerance. For the measurements in Table 4.6 we used the stopping criterion

$$\|r\|_2 \leq \varepsilon \|b\|_2,$$

with the tolerance $\varepsilon = 10^{-6}$.

These results illustrate that an increase in solution accuracy of approximately 2.3 in the best case (when the domain has irregularities) is incurred in moving from an FFT-based scheme to a more versatile approach. Of course, this more versatile approach gives rise to increased accuracy.

Coarse level solver

Another decisive portion of the AMG preconditioner is the coarse level solver. We applied either a direct solver (KLU) or used a couple of steps of Gauss-Seidel iteration. Our experiments indicate that for our problems ML setup time and scalability are not affected significantly by the two coarse level solvers. The difference in setup and application of KLU and Gauss-Seidel differ only within a few percent. Only the construction of the preconditioner is cheaper with the iterative coarse level solver.

As stated by Tuminaro & Tong [**?**], the number of iterations done by the iterative coarse level solver is crucial for the performance of the preconditioner. Too many iteration steps slow down the preconditioner without a corresponding increase of its quality. Too few iteration steps with the coarse level solver degrade the quality of the overall preconditioner and lead to an increased number of steps of the global system solver. We tuned the iterative coarse level solver such that the overall quality of the preconditioner was about the same as with the direct coarse level solver. It turned out that 3 iteration steps sufficed.

Coarse level size

We also tested the performance of the solver for varying sizes of the coarsest level. ML seems to perform best when the coarsest grid size is around 1000. With a limit of 1000, the coarsest grid sizes ranged from 128 to 849 when running a tube embedded in a 256 × 256 × 256 grid on 16 to 512 cores.

At the same time we tried to set the size of the coarsest level proportional to the total available number of cores in order to get a sufficiently large coarse level size. It turned out to be very difficult to set a coarse level size with heuristics like this. The factorization time increased to up to 2 s in contrast to 0.25 s for the case where the coarsest level size is limited by 1000.

Open-space vs. PEC Boundary Conditions

In this section we compare the impact of two different boundary conditions in the setting of a physical simulation consisting of an electron source (4 MeV) followed by a beam transport section [?]. As the pipe radius gets close to the particles in the beam, the fields become nonlinear due to the image charge on the pipe. We compare the root-mean-square (rms) beam size in a field-free region (drift) of a convergent beam, cf. [?, pp.171ff]. The beam pipe radius is $r = 0.00085\,\mathrm{m}$ in case of the PCG solver. This is an extreme case in which the particles fill almost the whole beam pipe and hence the effect is very visible. The capability to have an exact representation of the fields near the boundaries is very important, because the beam pipe radius is an important optimization quantity, towards lower construction and operational costs in the design and operation of future particle accelerators.

In Figure 4.8 we compare rms beam sizes for the two boundary conditions applied to the boundary of a cylinder with elliptical base-area as described in Section 4.2.1.

The differences are up to 40% (at $z = 1.435\,\mathrm{m}$) in rms beam size, when comparing the PEC and the open-space approach. We clearly see the shift of the beam size minimum (beam waist) towards larger z values and smaller minima, which means that the force of the self fields are larger when considering the beam pipe. This increase in accuracy justifies the accurate boundary treatment in situations where the spatial extent of the beam is comparable with that of the beam pipe.

4.2.5 Improving Domain Decomposition

To improve the convergence behavior of the CG methods we precondition (4.16) by smoothed aggregation-based algebraic multigrid (SA-AMG) preconditioners. Aggregation-based AMG methods cluster the fine grid unknowns to aggregates as representation for the unknowns on the next coarser grid.

The multigrid preconditioner and iterative solver are implemented with the help of the Trilinos framework [?, ?]. Trilinos provides state-of-the-art tools for numerical computation in various packages. The AztecOO package, e.g., provides iterative solvers and ML [?] provides multilevel preconditioners. By means of ML, we created our smoothed aggregation-based AMG preconditioner. We use ML's “decoupled” aggregation strategy [?] which constructs aggregates consisting of cubes of $3 \times 3 \times 3$ vertices. This strategy may entail non-optimal aggregates at the subdomain interfaces. The subdomains represent the portion of the problem dealt with by a processor or core. The partitioning in subdomains is done manually based on the encasing cubic grid.

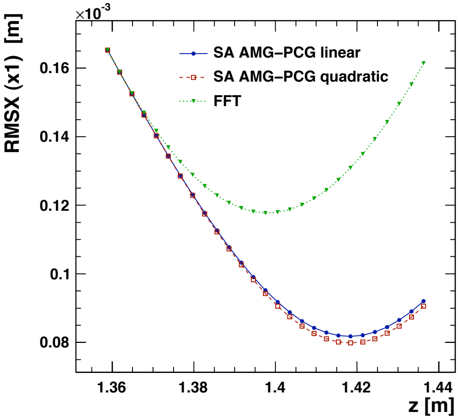


Figure 4.8: Comparison of rms beam size vs. position of a convergent beam. The FFT solver is applied for open-space boundary conditions, for PEC two variants of the SA AMG-PCG solver is used: with linear and quadratic boundary interpolation. (region of interest magnified). The computational domain Ω is a cylinder with $r = 0.00085\,\mathrm{m}$.

As suggested in [**?**] we choose a Chebyshev polynomial smoother. The employed coarse level solver (Amesos-KLU) ships the coarse level problem to node 0 and solves it there by means of an LU factorization. An alternative is to apply a few steps of an iterative solver (e.g. Gauss–Seidel) at the coarsest level. A small number of iteration steps decreases the quality of the preconditioner and thus increases the PCG iteration count. A large number of iteration steps increases the time for applying the AMG preconditioner. In [**?**] we found three Gauss–Seidel iteration steps to be a good choice for our application. In this paper, we use Chebyshev iteration.

Results concerning parallel performance of all variants are presented in Section 5.1.1.

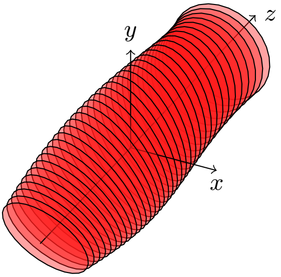


Figure 4.9: Sketch of a sliced particle bunch.

4.3 Envelope Tracker

This section is based on the solver and its parallelization described in [**?**].

As mentioned before, simulating particle accerlators is a computational intensive problem, particularly computing space charge forces. Commonly (as well as in OPAL) the arising N-Body problem is solved in every timestep using an FFT-based or iterative solver as introduced in the previous section. This, and additional computationally demanding components of a full 3D tracker, put a severe limitation on the time frame within which we can perform optimizations of particle accelerators.

In order to facilitate a poly-algorithmic approach (favored by the optimizer), we incorporated a simplified model (similar to [**?**]) into OPAL (named “envelope tracker”) providing a fast forward solver. With the fast forward solver we are able to reduced the time to solution by reducing the level of detail of the simulation while retaining important characteristics for the multi-objective optimizer. This can be achieved by replacing macro particles with slices, as depicted in Fig. 4.9. Slices describe the beam envelope, i.e. the extension of the beam in transversal direction. As with macro particles, forces such as space charge or external fields deform each slice. Since the Envelop Tracker employs a number of slices (commonly less than 1000) that is orders of magnitude smaller than the number of macro particles used in the 3D tracker, the execution time reduces drastically while the beam characteristics are retained. For example, due to the moderate number of slices, calculating the space charge forces by an $\mathcal{O}(n_{\text{slices}}^2)$ approach (calculating the interaction of one slice with all other slices for all slices) is feasible. We can even estimated the space charge forces cheaply in closed form. Compared with the 3D tracker (see Section~sec:it-sc), we trade a lower computational complexity for a lower

Symbol	Description
i	slice index
c	speed of light
z_i	longitudinal position of slice i
R_i	radius of slice i
β_i	speed of slice i relative to speed of light
γ_i	Lorentz factor of slices i
Q	bunch charge
L	bunch length
E_z^{ext}	total external electric field
E_z^{sc}	total space charge field
K	sum of focusing gradient of all active beamline elements

Table 4.7: Description of employed symbols.

level of detail.

The fast envelope tracker model and its implementations is discussed in Section 4.3.1. In order to understand performance we provide an overview of the theoretical complexity of all involved components. The scalability of our method is reported in Section 5.1.2 showing a reduction of the time to solution by two orders of magnitude (from 3D tracker to slice tracker with quadratic space charge model), and by even more with the analytical space charge model. This paves the way for large scale multi-objective optimization design of particle accelerators.

4.3.1 The Envelope Model

The envelope tracker solves an ordinary differential equation describing the equations of motion for homogeneously charged slices. Slices are in general ellipses (see Fig. 4.9).

The beam envelope model was first described by Sacherer in [**?**]. Even though the implementation supports elliptic beams we will derive the equations for cylindrical beams. Here, we use angular brackets to denote an average over time, e.g. for the beam size x,

$$\langle x(t)\rangle = \lim_{T\to\infty}\frac{1}{T}\int_0^T x(t)dt. \tag{4.18}$$

Denoting the rms beam size as $\sigma_x = \sqrt{\langle x^2 \rangle}$, we can derive the equation of motion for the beam envelope by differentiating with respect to time

$$\begin{aligned}\dot{\sigma}_x =& \frac{\langle x\dot{x}\rangle}{\sigma_x}, && (4.19)\\ \ddot{\sigma}_x =& \frac{1}{\sigma_x^3}\left[\langle x^2\rangle\langle \dot{x}^2\rangle - \langle x\dot{x}\rangle^2\right] + \frac{\langle x\ddot{x}\rangle}{\sigma_x}. && (4.20)\end{aligned}$$

Substituting

$$\dot{x} = \frac{p_x}{m\gamma} \tag{4.21}$$

and the root mean square normalized emittance

$$\epsilon_{n,x} = \frac{1}{mc}\sqrt{\langle x^2\rangle\langle p_x^2\rangle - \langle xp_x\rangle^2} \tag{4.22}$$

4.20 reduces to

$$\ddot{\sigma}_x = \frac{1}{\sigma_x^3}\left(\frac{c\epsilon_{n,x}}{\gamma}\right)^2 + \frac{\langle x\ddot{x}\rangle}{\sigma_x}. \tag{4.23}$$

Splitting the force term in $\ddot{x}$ to space-charge F_{sc} and external field K part,

$$\ddot{x} = -\gamma^2\beta\dot{\beta}\dot{x} + \frac{F_{\text{sc},x}}{m\gamma} \tag{4.24}$$

results in

$$\ddot{\sigma}_x + \left[\gamma^2\beta\dot{\beta}\right]\dot{\sigma}_x + \left[\frac{K}{m\gamma}\right]\sigma_x = \left[\frac{\langle xF_{\text{sc},x}\rangle}{m\gamma}\right]\frac{1}{\sigma_x} + \left(\frac{c\epsilon_n}{\gamma}\right)^2\frac{1}{\sigma_x^3}. \tag{4.25}$$

For cylindrical beams we have $\sigma_x = R/2$ and equation (4.25)for a slice with radius R_i (similar for axes by solving the equation twice for both axes independently) reduces to

$$\begin{aligned}&\ddot{R}_i + \gamma_i^2\beta_i\dot{\beta}_i\dot{R}_i + R_i\sum_j K_i^j = && (4.26)\\ &\frac{2c^2k_p}{\beta_i R_i} \times \left(\frac{G(\Delta_i, A_r)}{\gamma_i^3} - (1-\beta_i^2)\frac{G(\delta_i, A_r)}{\gamma_i}\right) + \frac{4\varepsilon_n c}{\gamma_i}\frac{1}{R_i^3},\end{aligned}$$

where k_p is the beam perveance, $G(\Delta_i, A_r)$ is the radial space charge term and $\Delta_i = z_i - z_{\text{tail}}$ is the distance from slice i to the tail of the bunch, $\delta_i = z_i + z_{\text{head}}$. $A_{r,i}$ denotes the slice rest frame aspect ratio $R_i/(\gamma_i L)$. The evolution of longitudinal motion for each slice is

$$\begin{aligned}\frac{d}{dt}\beta_i =& \frac{e_0}{m_0 c\gamma_i^3}\left(E_z^{\text{ext}}(z_i, t) + E_z^{\text{sc}}(z_i, t)\right), && (4.27)\\ \frac{d}{dt}z_i =& c\beta_i.\end{aligned}$$

The complete mathematical framework is given in [**?**].

Algorithm 6 states all necessary steps in order to solve these equations. The first three methods (invoked on line 2,4 and 5) handle the sampling of the time and position dependant external electric and magnetic fields present in a particle accelerator, e.g., cavities and magnets. While `switchElement()` ensures that, at the current position of the bunch, the correct beamline elements are active, the other two calculate the external electric and magnetic fields and slice deformation forces for each slice. Next we calculate self induced fields (see next two sections). During the emission phase the slices are gradually emitted at the cathode. Finally, in every timestep, we integrate the already emitted slices by a 5th order Runge-Kutta integration scheme with monitoring of local truncation errors (as presented in [**?**, pp. 714ff]) is used to solve (4.26) and (4.27). Since the sum of the external forces and the space charge forces can be computed beforehand, solving the ODE can be done for each slice independently and, therefore, in parallel.

Algorithm 6 Core code of the Envelope Tracker

```
 1: for all timesteps do
 2:     switchElements()
 3:     for all slices do
 4:         getExternalFields()
 5:         getKFactors()
 6:     end for
 7:     synchronizeSlices()
 8:     calcCurrent()
 9:     calcSpaceCharge()
10:     if not all slices emitted then
11:         emission()
12:     end if
13:     for all emitted slices do
14:         timeIntegration()
15:     end for
16:     $t \leftarrow t + \Delta t$
17: end for
```

A comparison shows that important quantities are within a 5% margin compared to the 3D tracker. In the context of our multi-objective optimization application this level of detail suffices. A comparison of both trackers is shown in Fig. 4.10. The simplest envelope tracker (OPAL-e) agrees remarkably well with the 3D particle tracker (OPAL-t).

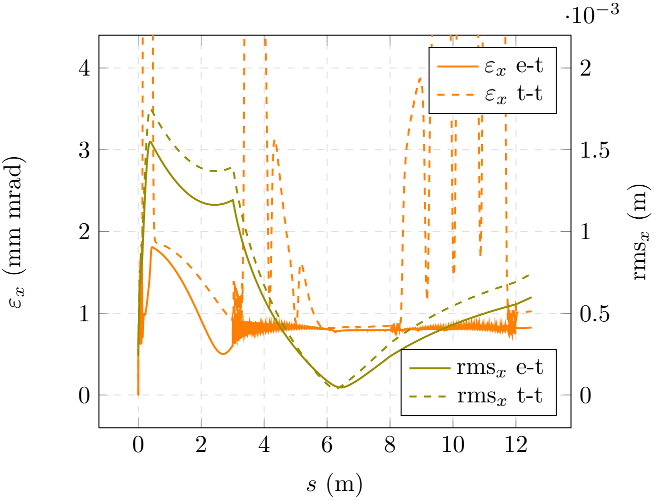


Figure 4.10: Comparison 3D tracker vs. envelope tracker.

N^2 space charge computation

In order to compute space-charge and current profile efficiently in parallel we collect β and z position (see Table 4.7) of all n slices a priori on all processors. This synchronization of slice information alone requires two `MPI_Allreduce` over an array of size n.

The actual space charge and current profile calculation are the most computationally involved in Algorithm 7. The N-Body problem requires $\mathcal{O}(n^2)$ operations and one global reduction on single floating point numbers. The current profile calculation is more expensive due to a Savitzky-Golay smoothing filter (implemented as in [?]), solving directly a linear system of equations as well as applying two convolutions. We denote the number of points of the current density that need to be smoothed by n_s. Typically n_s is only a small fraction of n.

Analytical space charge computation

In order to reduce the space charge computation costs we calculate an analytical approximation of space charge forces. This is achieved by introducing a factor z/L denoting the fraction of slices to the right of the slice under con-

Algorithm 7 $\mathcal{O}(n^2)$ space charge computation

```
for all $i \in$ slices do
    for all $j \in$ slices do
        $d_z \leftarrow |z_j - z_i|$                          ▷ distance from slice $j$ to $i$
        if $d_z >$ minimal slice distance we consider then
            $v \leftarrow$ calculated influence of slice $j$      ▷ scaled by $1/\sqrt{d_z^2}$
            if $z_j > z_i$                    ▷ check if $j$ is left or right of $i$ then
                $sm \leftarrow sm - v$
            else
                $sm \leftarrow sm + v$
            end if
        end if
    end for
    $F_{l,i} \leftarrow$ longitudinal space charge force depending on $sm$
    $F_{t,i} \leftarrow$ transversal space charge force can be computed independently
end for
```

sideration. Assuming a cylindrical beam shape the longitudinal space charge term becomes

$$E(z) = \frac{Q}{2\pi\epsilon_0 R^2} \left[\sqrt{\left(1-\frac{z}{L}\right)^2 + \left(\frac{R}{L}\right)^2} - \sqrt{\left(\frac{z}{L}\right)^2 + \left(\frac{R}{L}\right)^2} - \left|1-\frac{z}{L}\right| + \left|\frac{z}{L}\right| \right].$$

The radial space charge term can be derived similarly. Since the analytic formulation does solely depend on the bunch length and the z position of the slice under consideration, parallelization is trivial: the bunch length ($L = z_{\text{head}} - z_{\text{tail}}$) has to be computed once by finding the minimal and maximal slice z position using an `MPI_Allreduce`. Note that computing the current profile (`calcI()`) is not required under the analytic space charge model.

4.3.2 Theoretical Complexity

To understand the performance of the code we first cover a detailed analysis of the complexity of all methods introduced in Algorithm 6. In the following analysis we will denote the total number of slices by n, the total number of timesteps by t, with m the total number of beamline elements and s is the degrees of freedom of a slice. With the help of these abbreviations we deduce

Table 4.8: Complexity summary and number of `MPI_Allreduce` (NM) for various phases of envelope tracker (* denotes analytical space charge computation).

Phase	FLOPs	NM	size
switchElements()	$\mathcal{O}(m)$	–	
getExternalFields()	$\mathcal{O}(m)$	–	
getKFactors()	$\mathcal{O}(m)$	–	
synchronizeSlices()	$\mathcal{O}(1)$	2	n
calcCurrent()	$\mathcal{O}(n^3)$	1	n
calcSpaceCharge()	$\mathcal{O}(n^2)$	2	1
calcSpaceCharge()*	$\mathcal{O}(1)$	2	1
emission()	$\mathcal{O}(n)$	1	1
timeIntegration()	$\mathcal{O}(ns^2)$	–	

worst case bounds for the number of floating point operations (FLOPs), and summarized them in Table 4.8.

Starting from the top, we note that external fields methods do not require communication and operations are proportional to the number of currently active beamline elements. Subsequently, space charge forces have to be computed by one of the two described methods. Emission requires one collective reduction over one double and is done n times in total (independent of t). Finally, the Runge-Kutta integrator is called, requiring no communication and only $\mathcal{O}(ns^2)$ FLOPs.

Table 4.8 illustrates that the n^2 space charge model is computationally the most expensive part because we have to compute both the current and the more expensive space charges. With the analytical space charge model the current computation can be omitted and the actual space charge computation is cheaper. The other phases are comparably cheap and do not require additional communication. Therefore, we expect them to scale perfectly.

4.3.3 Parallelization

In this section we describe the parallelization of all relevant sections of the code, without considering the analytic space charge computation (as this is trivially parallelizable).

We distribute all slices in contiguous blocks on all available processors, creating a distribution with a load imbalance of at most 1 slice. The number of synchronization points is small, i.e. one per timestep in `synchronizeSlices()`. Once this synchronization takes place we only depend on a small number of collectives for single variables. The last communication related expensive part is calculating beam statistics. For the moment we can neglect this because, from an optimization point of view, we only need to calculate beam statistics a constant number of times during a simulation ($\ll t$).

The envelope tracker is parallelized by employing MPI collectives, such as e.g. `MPI_Allreduce`. On the other hand, OPAL relies on another framework, the Independent Parallel Particle Layer (IP^{2}L) [?], providing an abstract layer for handling parallel fields and particles. The envelope tracker implementation only uses OPAL's features to incorporate external electric and magnetic fields and the infrastructure, e.g. to handle input files. Since these features do not require much communication, reported performance results generally do not benchmark IP^{2}L, but only our pure MPI implementation of the envelope tracker. However, when examining the total MPI time it is important to take into consideration that using OPAL's features employ the IP^{2}Lmessage class (MPI layer using pre-posted `MPI_Isend`'s).

During the development of the code we first encountered quite limited scalability. An extensive profiling and benchmarking process, using the IBM Hpc-Toolkit (see Section 5.1.2), revealed that some parts of the original code were responsible as they were purely serial. This effect, well known as Amdahl's law [?], clearly was responsible for the initially bad parallel performance. Thus, while other parts scaled perfectly the serial part of the code became dominant and parallel performance dropped fast. This led to a large code review resulting in more optimizations and parallelization of serial parts. Currently the Savitzky-Golay smoother is the last part of the code that remains serial. Since the smoother is applied to only a very small fraction of the total number of slices ($n_s \ll n$), it currently only negligibly affects overall scalability, but is likewise affected by Amdahl's law.

Even after the revision of the parallelization of the code we noticed an unexpected deterioration of parallel performance when using a larger number of cores (i.e. only a handful of slices per MPI process). A further profiling and inspecting hardware performance counters revealed a slight load imbalance caused by a hard limit on the number of slices on process 0. This caused other processors to wait in collective routines. After removing this limitation, the load imbalance is, as previously claimed, at most 1 slice. For our application, achieving a tight load balance is very important. The number of slices per core rapidly declines to one or two when scaling to large number of processors.

In Section 5.1.2 we present the parallel performance of the described solver for both space-charge approaches.

Chapter

Results & Applications 5

In this chapter we report performance results for components individually and applying the multi-objective optimization framework to beam dynamic optimization problems. We start by presenting the parallel performance of the introduced particle accelerator simulation solvers in Section 5.1.1 and Section 5.1.2. In Section 5.1.3 we report the parallel performance of the solution propagation strategy introduced in Chapter 3.

Finally, we investigate real beam dynamic multi-optimization problems in Section 5.2, e.g., reproducing the Ferrario matching point as a proof of concept (see Section 5.2).

5.1 Parallel Performance

For a master/slave massively parallel multi-objective optimization framework it is important that all components individually show a superior scaling behavior and a being able to deliver results in a meaningful time frame. In this section we show that our forward solvers and the framework exhibits the necessary qualities and delivers results efficiently.

In Section 5.1.1 we the present very scalable implementation of a Poisson solver for space-charge computation (see Section 4.2) suitable to handle domains with irregular boundaries as they arise, for example, in beam dynamics simulations. This solver employs the conjugate gradient algorithm preconditioned by smoothed aggregation based algebraic multigrid. Overall we trade a slight increase of the time to solution for more accuracy compared to the FFT based space-charge computation.

Our fastest model, the envelope-tracker introduced in Section 4.3, we show excellent scaling up to large problems. The results presented in Section 5.1.2 show satisfactory parallel efficiency for even the small number of slices of current simulations. Indeed, we achieved an almost 2 orders of magnitude reduction of runtime for relevant simulations used in multi-objective optimization problem. In particular, in the context of multi-objective optimization, we are able to tune the run-time parameters of the forward solver in such a way that we are maximizing the overall performance and the scalability.

We conclude in Section 5.1.3 with the performance measurements of the full framework in action, benchmarking the latency at master nodes and the scalability.

5.1.1 Iterative Space-Charge Solver

In this section we discuss a set of numerical experiments and results that are designed to test different variants of the preconditioner and to compare of solvers and boundary extrapolation methods. Unless otherwise stated the measurements are done on a tube embedded in a rectangular equidistant $256 \times 256 \times 256$ grid. This is a common problem size in beam dynamics simulations. Most of the computations were performed on the Cray XT4 cluster of the Swiss Supercomputing Center (CSCS) in Manno, Switzerland. Computations up to 512 cores were conducted on the Cray Seastar cluster (BUIN) with 468 AMD dual core Opteron 2.6 GHz processors and a total of 936 GB DDR RAM on a 7.6 GB/s interconnect bandwidth network. Larger computations were performed on the XT3 cluster (PALU) with 1692 AMD dual core Opteron 2.6 GHz processors and a total of 3552 GB DDR RAM interconnected with a 6.4 GB/s interconnect bandwidth network.

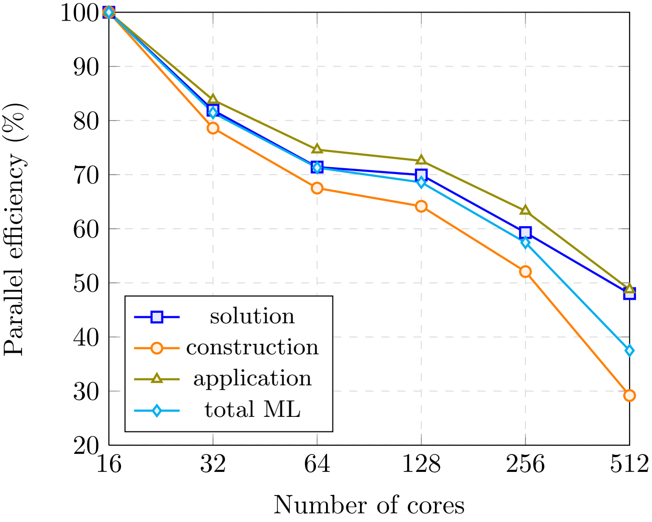
Throughout this section we will report the timings of portions of the code as listed in Table 5.1.

name	description
construction	time for constructing the ML hierarchy
application	time for applying the ML preconditioner
total ML	total time used by ML (= construction + application)
solution	time needed by the iterative solver

Table 5.1: Description of various timings used.

The efficiency of the AMG solver introduced in the previous chapter, using constant boundary extrapolation is shown in Figure 5.1. The top plot in this figure shows the results listed in Table 5.2 for a 256^3 grid are plotted. We observe an efficiency of approximately 62% for 256 cores relative to the timing on 16 processors, the minimal number of processors to solve the problem. The efficiency dropped just below 50% for 512 cores. The parallel efficiency is affected most by the poor scalability of the construction phase of ML. After studying various ML timings we could not identify a specific reason that causes this loss in efficiency for the construction phase other than the assumption that the problem is too small with respect to the number of cores.

Similar conclusions can be drawn for the parallel efficiency of our solver on a 512^3 grid with constant boundary treatment, see Table 5.3 and Figure 5.1.



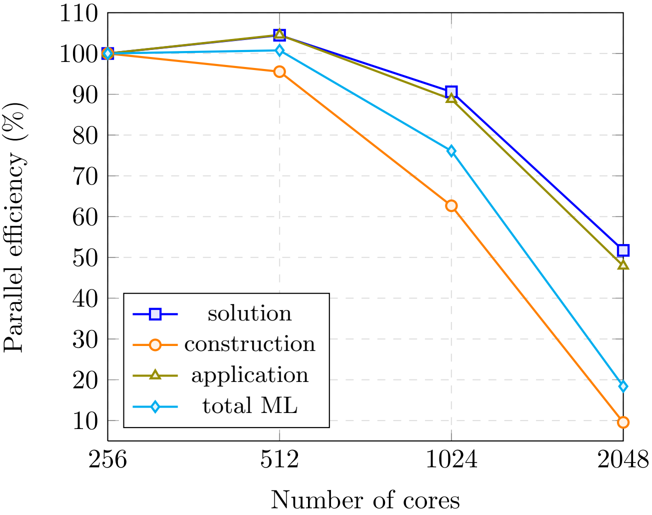


Figure 5.1: Relative efficiency for AMG on a tube embedded in a $256 \times 256 \times 256$ grid (top) and in a $512 \times 512 \times 512$ grid (bottom) with constant extrapolation at the boundary.

cores	solution [s]	construction [s]	application [s]	total ML [s]
16	10.91	7.75	9.52	17.28
32	6.66	4.93	5.68	10.61
64	3.82	2.87	3.19	6.06
128	1.95	1.51	1.64	3.15
256	1.15	0.93	0.94	1.88
512	0.71	0.83	0.61	1.44

Table 5.2: Timings for AMG on a tube embedded in a 256 × 256 × 256 grid with linear extrapolation at the boundary.

cores	solution [s]	construction [s]	application [s]	total ML [s]
256	10.51	5.79	8.70	14.49
512	5.03	3.03	4.16	7.19
1024	2.90	2.31	2.45	4.76
2048	2.54	7.58	2.27	9.85

Table 5.3: Timings for AMG on a tube embedded in a 512 × 512 × 512 grid with constant extrapolation at the boundary.

Again the construction phase of ML scales poorly for larger numbers of cores.

Notice that by applying the improvements discussed in Section 4.2.4, i.e., reusing (parts of) the preconditioner, the time needed for the construction phase can be reduced significantly or avoided altogether. If the preconditioner has to be built just once in an entire simulation the efficiency will get close to the 52% that we measured for the solution phase.

Finally, in Table 5.4 we report on timings obtained for the tube embedded in a 1024 × 1024 × 1024 grid. In Figure 5.2 the corresponding efficiencies are listed. For this large problem we observe good efficiencies. The solver runs at 82% efficiency with 2048 cores relative to the 512-cores performance. The construction phase is still performing the worst with an efficiency of 73%. In this setup the problem size is still reasonably large when employing 2048 cores. This consolidates our understanding of the influence of the problem size on the low performance of the aggregation in ML.

Tables 5.2–5.4 provide data to investigate weak scability. The timings for 2048 processors in Table 5.4 should ideally equal those for 256 processors in Table 5.3. In fact, they are quite close. A comparison of the timings in Tables 5.3 and 5.2 is not so favorable. The efficiency is at most 84%.

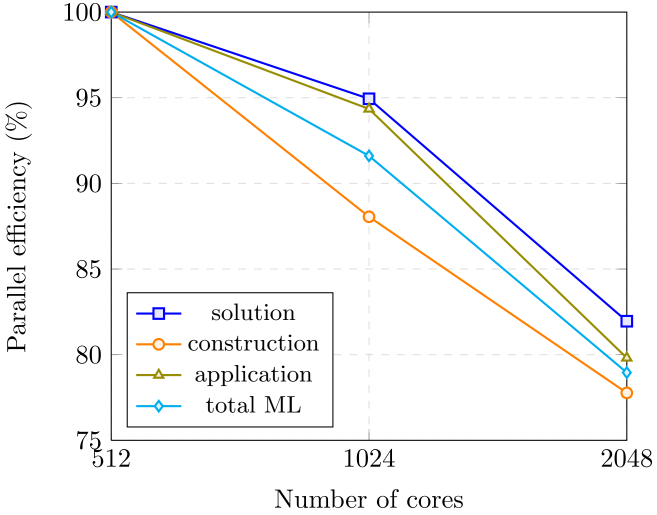


Figure 5.2: Relative efficiency for AMG on a tube embedded in a 1024 × 1024 × 1024 grid with linear extrapolation at the boundary, corresponding to the numbers in Table 5.4.

The numbers for 2048 processors in Table 5.3 show that the construction of the multilevel preconditioner becomes excessively expensive if the number of processors is high.

Discussion

The code exhibits satisfactory scalability up to 2048 processors with cylindrical tubes embedded in meshes with up to 1024^3 grid points. In the very near future, this approach will enable precise beam dynamics simulations in large particle accelerator structures with a level of detail not obtainable with previous approaches. The solver would profit from using adaptive mesh refinements reducing the number of grid points in regions that are less relevant for the space charge calculation.

cores	solution [s]	construction [s]	application [s]	total ML [s]
512	35.83	20.78	29.53	50.31
1024	18.87	11.80	15.65	27.46
2048	10.93	6.68	9.25	15.93

Table 5.4: Timings for AMG on a tube embedded in a $1024 \times 1024 \times 1024$ grid with linear extrapolation at the boundary.

Improved domain decomposition

As discussed above, the comparison with the FFT-based solver is in fact not completely trivial, since the computational domains of the two solvers differ. The FFT-based solver requires a rectangular computational domain whereas our finite difference solver approximates well the geometry of the device. In many situations the solutions of the two problems match quite well. However, in problems where the spatial extent of the beam is comparable with that of the beam pipe it is important to have an accurate representation of the field near the boundary. Then, the results of the computations can differ significantly, and the results of the FFT-based solver are questionable.

In this section we present the results addressing the issue of load balancing and communication overhead. We conduct the numerical experiments on the newest Cray XT-5 at the Swiss Supercomputing Center (CSCS). This machine consists of 3688 AMD hexa-core Opteron processors clocked at 2.4 GHz. The system has 28.8 TB of DDR2 RAM, 290 TB disk space, and a high-speed interconnect with a bandwidth of 9.6 GB/s and a latency of 5 μs[1].

Our computational domains are embedded in a 3D rectangular domain, as illustrated in Fig. 4.5. The rectangular computational grid is generated inside the rectangular domain. Only grid points of the rectangular grid that are included in the computational domain Ω are used in the computation. In the computations in the previous section the partitioning of the computational domain was based on the partitioning of the *whole* rectangular domain. (This underlying rectangular grid is induced by the particle code OPAL [**?**] that participates at the overall computation.) Therefore, some subdomains contained far fewer grid points than others, causing severe load imbalance. In fact, in many cases there were subdomains that contained no grid points at all. In our new approach we partition the computational domain. In this section we compare the old and the new approach.

The computational domain we are dealing with in this paper is a circular

---

[1] `http://www.cscs.ch/455.0.html` retrieved on July 13, 2010.

cylinder embedded in a $1024 \times 1024 \times 1024$ grid. In this setup the problem size is still reasonably large when employing 2048 cores, i.e., a subcube contains (up to) $524'288$ grid points. We use linear extrapolation at the Dirichlet boundary Γ_1. The solver is the AMG-preconditioned conjugate gradient algorithm as implemented in Trilinos and discussed in section 4.2.2. Timings for three phases of the computation are given in Table 5.5. For this large problem with 840 million degrees of freedom we observe quite good efficiencies. The solver runs at 87% efficiency with 2048 cores relative to the 512-cores performance. Note that the solution phase contains the application of the preconditioner. Therefore, the difference of the two columns indicated by 'solution' and 'application' essentially gives the time for matrix-vector and inner products in the conjugate gradient algorithm. The column 'total ML' comprises the sum of columns 'construction' and 'application'. The construction phase is performing the worst with an efficiency of 71%. We found that much of the time in the construction of the preconditioner goes into the factorization of the coarsest level matrix. Therefore we decided to replace the direct solver KLU by one step of the Chebyshev semi-iterative method [**?**] with polynomial degree 10. The timings for this approach are given in Table 5.6. The times for the construction of the preconditioner have been reduced considerably, at the expense of a slightly more expensive solution phase. Now the construction phase scales perfectly. The iterations count changes only marginally from 20

cores	solution	construction	application	total ML
512	62.22 [1.00]	35.12 [1.00]	51.68 [1.00]	86.79 [1.00]
1024	32.95 [0.94]	19.95 [0.88]	27.47 [0.94]	47.41 [0.92]
2048	17.68 [0.87]	12.37 [0.71]	14.85 [0.87]	27.22 [0.80]

Table 5.5: Times in seconds and relative parallel efficiencies. The original data distribution is used, and the coarsest AMG level is solved with KLU.

cores	solution	construction	application	total ML
512	63.12 [1.00]	32.09 [1.00]	52.73 [1.00]	84.80 [1.00]
1024	33.54 [0.94]	16.31 [0.98]	28.04 [0.94]	44.35 [0.96]
2048	18.56 [0.85]	8.10 [0.99]	15.66 [0.84]	23.76 [0.89]

Table 5.6: Times in seconds and relative parallel efficiencies. The original data distribution is used, and the coarsest AMG level is solved iteratively.

to 19 (see Table 5.9).

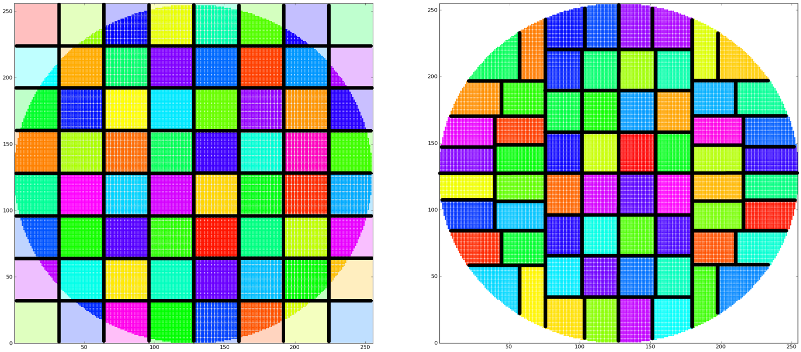


Figure 5.3: (LEFT) Cross section of data distribution on 512 cores on a $8\times8\times8$ processor grid (colors indicate data owned by a processor) and (RIGHT) the same data redistributed with recursive coordinate bisection (RCB).

Fig. 5.3 (left) illustrates the cross section in the case where the computational domain is embedded in a cube that has been subdivided in $8\times8\times8$ subcubes. It is easily seen that the four subcubes in the corners do not contain any points of the computational grid, while other subcubes contain at least a few grid points. In general, a fraction of only about $\pi/4 \approx 0.8$ of the grid points of the cube will be included in the computational domain. This number corresponds to the ratio of the volumes of the circular cylinder and its encasing cube. That is, about 20% of the nodes in the cube are not involved in the computation and, hence about 20% of the subcubes contain a reduced number of nodes. Since subcubes correspond to cores, empty or almost empty subcubes correspond to idle or underloaded cores.

Evidently, this partitioning entails a severe load imbalance. Nevertheless, the speedups shown in Table 5.5 do not fully reflect this imbalance. These good speedups are comprehensible if one considers the most loaded processes that correspond to the innermost cubes, i.e., those close to the z-axis. These subcubes contain $1024^3/p$ nodes. An increase of the processor number p leads to an optimal speedup, at least as long as the floating point operations dominate the work load. This is the case here, as even in the 2048 processor computation a core handles more than half a million nodes.

Although speedups are good (most likely influenced by the lack of comparison with one processor) with this crude distribution of work, a better balanced work will lead to improved execution times. After all, there are only about 80% of the 1.1 billion nodes inside the computational domain.

To better balance the load we partition data using the recursive coordinate bisection (RCB) algorithm as it is implemented in the Zoltan toolkit [**?**,**?**]. The Trilinos package Isorropia[2] provides a matrix-based interface to Zoltan. The recursive coordinate bisection (RCB) algorithm partitions the graph according to the coordinates of its vertices. The computational domain is first divided into two subdomains by a cutting plane orthogonal to one of the coordinate axes so that half the work load is in each of the subdomains. (Notice that other fractions than 50-50 are possible and in fact needed if the number of processors is not a power of 2.) That coordinate axis is chosen which is associated with the longest elongation of the domain. Clearly, this procedure can be applied recursively. RCB leads to nicely balanced data distributions. In rectangular grids, RCB is a particularly fast partitioner since the coordinates of the grid vertices are easily determined from their indices. In Fig. 5.3 on the right the cross section of the circular cylinder is shown with the partitioning into 64 subdomains. Now, all subdomains contain an almost equal number of nodes which leads to almost equal loads per processor. This subdivision evidently is more complicated than the one on the left of this figure. Except for subdomains in the center, i.e. close to the z axis, there are more than just six neighbors (in 3D). With the increased number of neighbors the number of messages that are to be sent in the communication steps increases. The communication volume does not change much. (Notice that Trilinos constructs the communication pattern transparent to the user.)

cores	solution	construction	application	total ML
512	50.32 [1.00]	27.37 [1.00]	44.00 [1.00]	71.37 [1.00]
1024	28.14 [0.89]	16.82 [0.81]	24.82 [0.89]	41.58 [0.86]
2048	15.26 [0.82]	15.81 [0.43]	13.47 [0.82]	29.28 [0.61]

Table 5.7: Times in seconds and relative parallel efficiencies. Data is distributed by RCB. The coarsest AMG level is solved with KLU.

In Tables 5.7 and 5.8 the execution times of our code is given with the data redistributed by RCB. These numbers are to be compared with those in Tables 5.5 and 5.6, respectively, where the original data distribution was used. For 512 cores the execution times are significantly smaller with the RCB distribution, about 20%, as the previous discussion suggests. Tables 5.10

---

[2] See `http://trilinos.sandia.gov/packages/isorropia/`

cores	solution	construction	application	total ML
512	51.08 [1.00]	25.65 [1.00]	44.89 [1.00]	70.55 [1.00]
1024	27.38 [0.93]	12.96 [0.99]	24.51 [0.92]	37.07 [0.95]
2048	14.76 [0.87]	6.69 [0.96]	13.10 [0.86]	19.79 [0.89]

Table 5.8: Times in seconds and relative parallel efficiencies. Data is distributed by RCB. The coarsest AMG level is solved iteratively.

cores	iterations RCB KLU and iterative	iterations original KLU	iterations original iterative
512	20	20	20
1024	20	20	20
2048	19	20	21

Table 5.9: Iteration count for various configurations: RCB vs. original data distribution and KLU vs. iterative coarse level solver.

and 5.11 give more details. There, in brackets, the efficiencies of the runs with the original rectangular distribution are listed relative to the 512 processor run with the RCB distribution.

When using the iterative solver on the coarsest level, speedups and thus efficiencies are quite close, cf. Tables 5.6 and 5.8. However, there are significant differences when the coarsest level system is solved directly. In this case, the efficiencies deteriorate more quickly with the RCB distribution than with the original distribution. A more detailed analysis revealed that the significant difference between original and RCB partitioning is caused by the LU factorization of the matrix of the coarsest level. In fact, the factorization takes

cores	solution	construction	application	total ML
512	1.00 [0.81]	1.00 [0.78]	1.00 [0.85]	1.00 [0.82]
1024	0.89 [0.76]	0.81 [0.69]	0.89 [0.80]	0.86 [0.75]
2048	0.82 [0.71]	0.43 [0.55]	0.82 [0.74]	0.61 [0.66]

Table 5.10: Parallel efficiencies of the RCB partitioned runs and relative parallel efficiencies of the runs with original data distribution. The coarsest AMG level is solved with KLU.

cores	solution	construction	application	total ML
512	1.00 [0.81]	1.00 [0.80]	1.00 [0.85]	1.00 [0.83]
1024	0.93 [0.76]	0.99 [0.79]	0.92 [0.80]	0.95 [0.80]
2048	0.87 [0.69]	0.96 [0.79]	0.86 [0.72]	0.89 [0.74]

Table 5.11: Parallel efficiencies of the RCB partitioned runs and relative parallel efficiencies of the runs with original data distribution. The coarsest AMG level is solved iteratively.

3.64 sec on 1024 cores and 7.86 sec on 2048. We do not see a reason for this drastic increase of factorization time since the problem sizes are quite close (2048 vs. 1865). The fact that the partitions with RCB have more neighbors does not seem to be a strong enough reason. In any case, larger differences would have to be visible in Tables 5.6 *and* 5.8.

Here we restricted ourselves to problems posed on grids of size $1024 \times 1024 \times 1024$. In [**?**], investigating the original solver, we also included $512 \times 512 \times 512$ and $256 \times 256 \times 256$ grids. On the former grid the solver scaled similarly as on the 1024^3 grid, on the latter grid quite poorly. By replacing the direct coarse level solver by an iterative coarse level solver we expect similar improvements in the parallel performance of our solver also for these smaller problem sizes.

Discussion

A real world example where the solver was used in a beam dynamics code (OPAL) shows the relevance of this approach by observing up to 40% difference in the rms beam size when comparing to the FFT-based solver with open domains. The code exhibits excellent scalability up to 2048 processors with cylindrical tubes embedded in meshes with up to 1024^3 grid points. This enables precise beam dynamics simulations in large particle accelerator structures with a level of detail not obtained before.

In real particle simulations (and other test cases encountered) system matrices arising from quadratic boundary treatment (Shortley-Weller) are only ‘mildly’ nonsymmetric such that PCG can be applied.

5.1.2 Envelope Tracker

Results presented in this section were measured on an IBM Blue Gene/P system. One node consists of a Quad core 450 PowerPC running at 850 MHz and a peak performance of 13.6 GFLOP/s (per node). The machine has 5

Table 5.12: Blue Gene/P specifications.

Node Properties	
Node processors (compute and I/O)	Quad core 450 PowerPC
Processor frequency	850 MHz
Coherency	Symmetrical multiprocessing
L1 Cache (private)	32 KB per core
L2 Cache (private)	14 stream prefetching
L3 Cache size (shared)	8 MB
Main store memory/node	2 GB or 4 GB
Main store memory bandwidth	16 GBps
Peak performance	13.6 GFLOPS (per node)
Torus network	
Bandwidth	6 GB/s
Hardware latency (nearest neighbor)	64 ns (32-byte packet) 512 ns (256-byte packet)
Hardware latency (worst case)	3 μs (64 hops)
Global collective network	
Bandwidth	2 GB/s
Hardware latency (round-trip worst case)	2.5 μs

networks, two of which are of particular interest for our application: a 3D Torus network and a (tree) network for collective communication. Details are given in Table 5.12 (see [?]).

Experimental Setting

The Blue Gene/P system offers 3 modes of operation. In `VN` mode (default for all our experiments), each of the 4 cores per compute node spawns an MPI process. In the `DUAL` mode we have 2 MPI processes and each of them can spawn 2 shared memory threads. Finally, the `SMP` mode features 1 MPI process per node which can have 4 threads.

We measured timings of different components with help of IP^{2}Ltimers (providing a simple wrapper for `MPI_Wtime` for minimum, average and maximum) and the HpcToolkit [?] for MPI analysis and in-depth details regarding spent cycles. Total time measures the total run time from start to end. In addition, we report timings of space charge and current density calculation (lines 8 and 9 in Algorithm 6), the time integration (line 14 in Algorithm 6) and external field evaluation (line 4 in Algorithm 6). The benchmark simulation performs the first 2000 timesteps of PSI's actual 250 MeV Injector facility. This models the first part of PSI's SwissFEL [?] project, which has a major influence in achieving the design goals of tightly focused and intense light beams. In this light, it is of great importance have access to fast models in order to optimize and understand the beam dynamics of this first section.

Strong Scaling

In Fig. 5.4 minimum, maximum and average timings for 1'000 slices are plotted. We see that minimum and maximum timings are narrow, showing that the partition is well balanced. As expected, space charge and current profile dominate parallel performance. Timings of the ODE solver and external field evaluation are one order of magnitude smaller and at some point scale below of the total communication time (after 128 cores).

As of two cores the MPI timings are constant, independent of the total number of cores employed and only slightly increase again as of 128 cores. Notice that, even with this rather small problem size, the achieved bandwidth is quite satisfactory, although still far form the peak. In particular, for 128 cores on core 0 we get

$$\frac{N_{\text{calls}} \times \text{bytes} \times 8}{t \times 2^{20}} = \frac{7999 \times 7988.0\text{b} \times 8}{1.583\text{s} \times 2^{20}} \approx 308\text{MB/s},$$

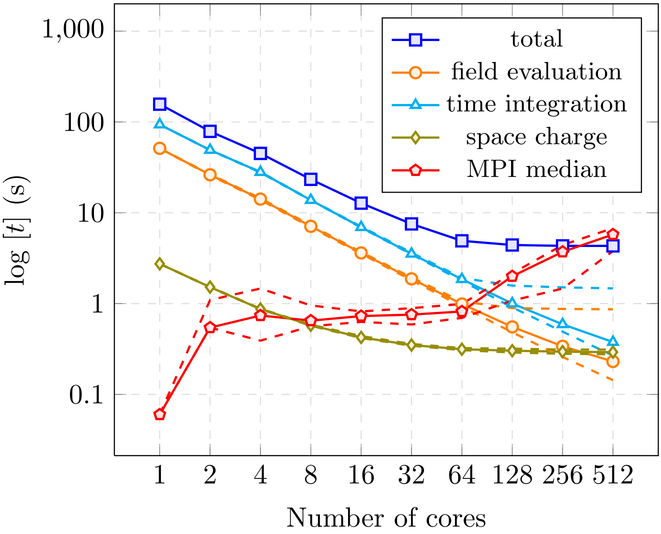
for all `MPI_Allreduce` of size n. Collectives of single values (4 and 8 bytes) are latency dominated. One indication is that the total time of 6007 × 4 byte messages and 16006 × 8 byte messages are, relative to the difference in data volume, close (0.245 and 0.333). Furthermore, considering the "ideal" latency

$$\frac{t}{N_{\text{calls}}} = \frac{0.245\text{s}}{6007} \approx 41\mu\text{s},$$

we are only latency bound for small collectives.

On the Blue Gene/P it is possible to switch between different communication strategies for the collectives. This can be achieved by changing environment variables that control the DCMF layer (see [?]), i.e. `DCMF_*`. We experimented with two different values: `TREE` and `GLOBAL` for the allreduce operation. `TREE` forces the `MPI_Allreduce` to use the collective network whereas `GLOBAL` uses the global collective network protocol (see [?] for more details). The default is using the tree and direct put protocol. In both cases we did not see any improvement.

The parallel efficiency for measured average timings is shown in Fig. 5.5. When increasing the number of cores from 1 to 4 field evaluation and ODE solver suffer from cache effects. Independently of where cache utilization is



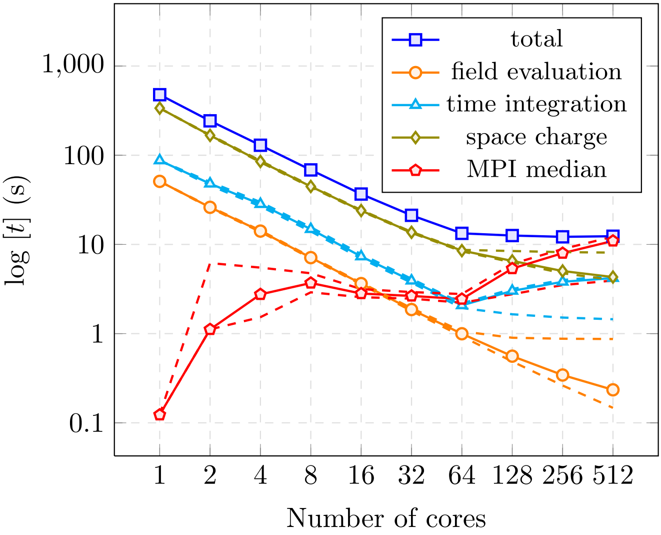
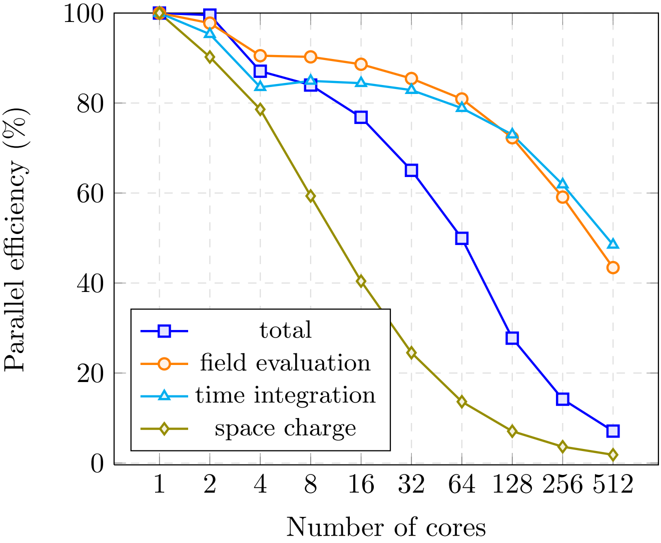


Figure 5.4: Timings for simulation with 1‘000 slices with analytical (Top) and n^2 (Bottom) space charge model, minimum and maximum dashed, average solid line.



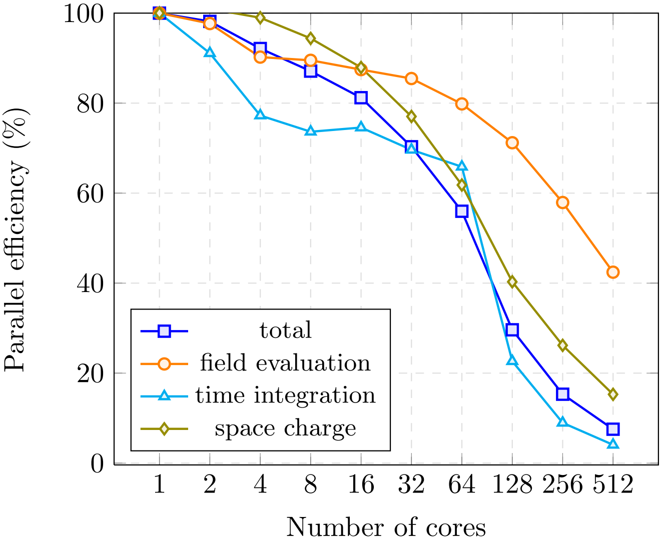


Figure 5.5: Parallel efficiency for average timings of 1‘000 slices for analytical (Top) and n^2 (Bottom) space charge.

on one core, by increasing the number of cores, the amount of data per core shrinks and therefore cache performance drops.

As we already mentioned, current simulations use up to 1000 slices, which clearly limits overall scalability to at most 1000 MPI processes. However, in the future (inclusion of coherent synchrotron radiation and other physical phenomena) we can require a significantly higher resolution and therefore more slices. In view of these facts and to demonstrate the parallel performance potential of the code with regard to larger number of cores we provide the average timings for a larger problem (10‘000 slices) in Fig. 5.6 and 5.7. As expected, we see a much better parallel efficiency (≈ 85% at 128 relative to 8 cores). The reason for this can be found by inspecting the saturation of the bandwidth. For 10‘000 slices on core 0 out of 128 we get

$$\frac{7999 \times 80000.0\text{b} \times 8}{7.385\text{s} \times 2^{20}} \approx 661\text{MB/s}.$$

We also note that the timings for ODE solver, field evaluation as well as MPI timings are almost two magnitudes smaller than space charge and current density calculation for the n^2 space charge model.

Weak Scaling

Weak scaling is shown in Fig. 5.8. Since the overall complexity of the n^2 space charge model is quadratic, we fix the work per core to

$$\lceil \sqrt{n_c} \times 100 \rceil,$$

and for the analytical space charge model respectively to

$$n_c \times 100,$$

slices for n_c cores and starting with 100 slices on 1 core. For n^2 space charge we note a drop in execution time for all parts except for space charge and current density calculation, where we notice a slightly increasing tendency caused by the serial smoother. An increasing number of latency bound small allreduce collectives imply a growing trend of total MPI time with increasing number of cores. Again, field evaluation and ODE solver performance is very good. The steep increase in MPI time towards large number of cores is caused by IP^{2}Lposting lots of `MPI_Iprobe`’s (16.8s on 2048 cores).

The analytical space charge calculation has almost ideal weak scaling, only the MPI time shows an increasing tendency towards the end caused by excessive MPI shutdown times for IP^{2}L. As seen with the n^2 space charge model, this is caused by IP^{2}L(here 12.3s on 2048 cores and 20.3s on 4096 cores).

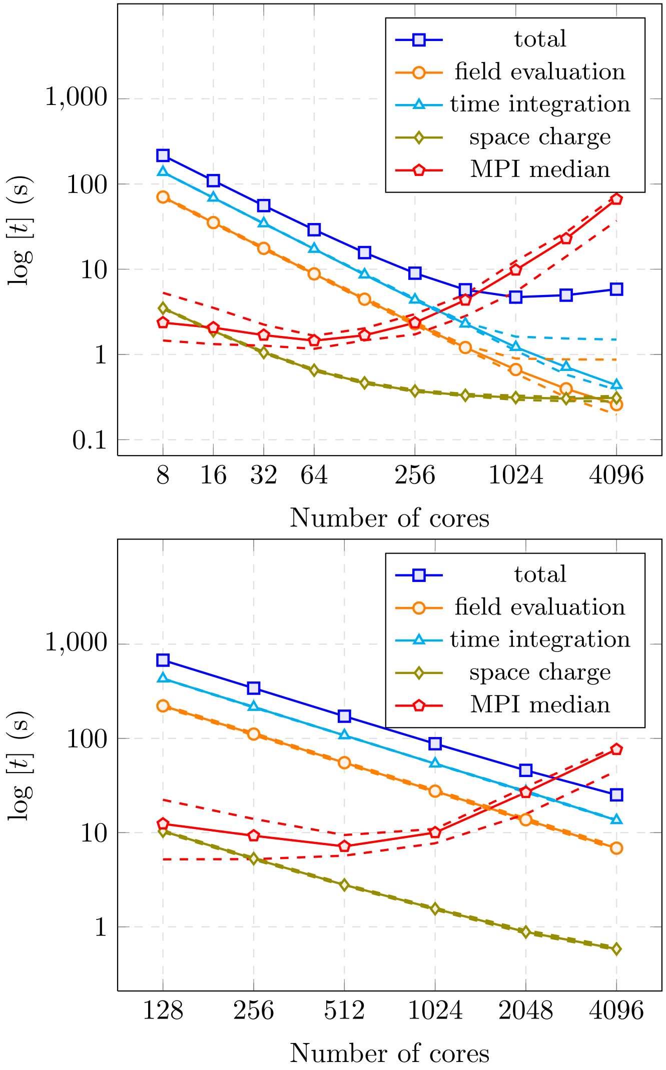


Figure 5.6: Average timings (solid line) for 10‘000 slices (Top) and 500’000 slices (Bottom) with analytical space charge model. (Minimum and maximum dashed line)

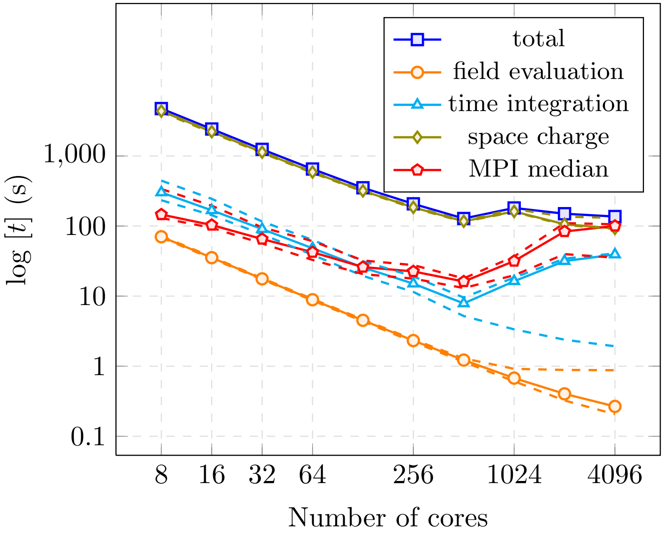


Figure 5.7: Average timings (solid line) for n^2 space charge model with 10‘000 slices. (Minimum and maximum dashed line)

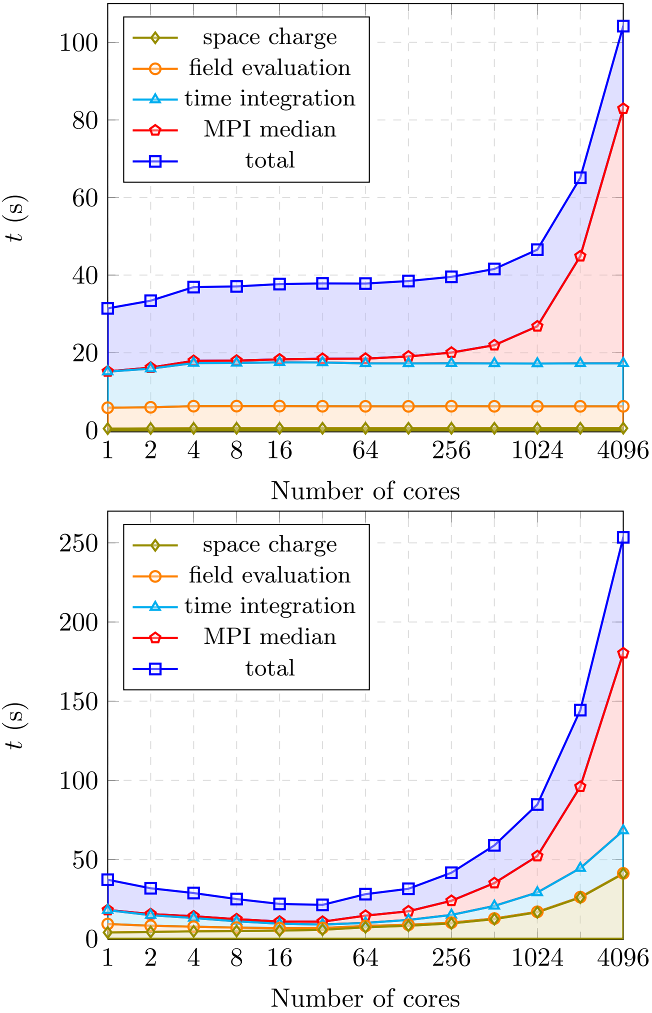


Figure 5.8: Weak scaling (average timings) of analytical (TOP) and n^2 (BOTTOM) space charge model.

Discussion

Even though we notice a small loss in parallel efficiency for small problems, we demonstrated that the code is well parallelized (for larger problems). Bandwidth saturation enforces a hard limit on performance with respect to problem size. Fortunately, the optimizer requires thousands of forward solves when solving a multi-objective optimization problem. This enables us to determine how many cores are necessary to maximally saturate the bandwidth or to achieve an acceptable parallel efficiency for one forward solve and subsequently run the maximal number of forward solves in parallel.

5.1.3 The Master/Slave Framework

Results presented in this section were measured on an IBM Blue Gene/Q [**?**] system. One compute card consists of one chip with 16 PowerPC A2 cores running at 1.6 GHz (4 hardware threads) and a peak performance of 204.8 GFLOP/s (per node) drawing 55 watts and provides 16 GB of memory. Node boards are assembled from 32 compute cards where all compute cards are linked in a torus fashion. Finally, 16 node boards are combined into a midplane, where a rack has one or two midplanes (depending on the I/O drawers). The machine is connected in a 5D Torus network each link provides a user bandwidth of 1.8 GB/s per link and a latency between 80 ns and 3 μs (near, far). More details are given in [**?**].

The performance gain of adding additional worker nodes depends heavily on the computational complexity of the tasks the workers have to solve. For computationally dominating tasks, the master will be mostly idle and waiting for results. The opposite holds if the workers have to deal with light tasks that can be computed quickly. In this scenario the master will be overwhelmed by messages and job requests.

The more interesting case is an experimental setting where the worker payload has a high computational cost. To that end, we use the framework in combination with the OPAL simulation component, applied to the following optimization problem:

$$\begin{aligned}
\min \quad & [\delta E,\, \varepsilon_x]^T \\
\text{s.t.} \quad & 0.00025 \leq \sigma_x = \sigma_y \leq 0.00029 \\
& 110.0 \leq \mathrm{KS}_{\mathrm{RF}} \leq 120.0 \\
& 0.0 \leq \mathrm{LAG}_{\mathrm{RF}} \leq 0.05 \\
& 25.0 \leq \mathrm{Volt}_{\mathrm{TW},\, i} \leq 40, \quad i \in \{1, 2\} \\
& 0.1 \leq \mathrm{KS}_{\mathrm{TW},\, j} \leq 0.5, \quad j \in \{1, \ldots, 8\}
\end{aligned}$$

n_p	n_w	t_{wall} [s]	t_{comm} [s]	efficiency
16	14	5468.865	689.206	-
32	30	2706.857	390.105	1.01
64	62	1641.878	282.984	0.83
128	126	970.585	219.520	0.70
256	254	728.767	211.228	0.47

Table 5.13: Timings and efficiency for one island with population size of 260 running for 780 function evaluations.

The problem minimizes the energy spread and emittance (see Section 4.1.1), while we vary the laser spot size σ, the other design variables correspond to field strengths of the first focusing magnet, the lag of the gun, and the configuration of the first two accelerating traveling wave structures.

We configured the evolutionary algorithm to use the NSGA-II selector in combination with the blend crossover operator and the independent bit mutation operator. The forward solver uses the envelope tracker introduced in Section 4.3 to evaluate individuals.

We used the HpcToolkit [**?**] to measure total runtime and communication time. Notice that in order to cleanly shutdown the execution we need to wait until all MPI processes executing simulation have finished. Once the optimization converged, all MPI processes are notified but running simulations cannot be aborted. This generates an overhead increasing with the number of worker processes used (`MPI_Abort` interferes with the measurements).

Scalability

We start by inspecting the performance when using only one island and consequently only one master. In Figure 5.9 and Table 5.13 show the measured timings and parallel efficiency for 780 function evaluations using 8 tasks per compute card.

Notice that the communication time is roughly the same for 128 and 256 core runs. This is due to the fact that we approach the ideal case, where each worker only evaluates one function evaluation. This represents a lower bound on the communication for this setting.

After we established a way to disarm hot-spots in the network in Section 3.5 we still have to show that additional resources are not wasted, e.g., the scalability of the system is retained when adding more masters.

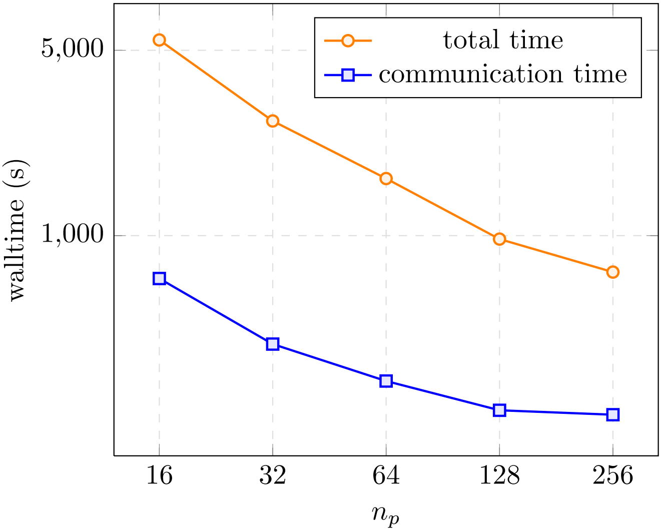


Figure 5.9: Scalability using only one island (1 master and 1 optimizer). We use a population of size 260 and run it for 780 function evaluations.

Due to the random and asynchronous nature of the approach measuring scalability is difficult and we have to agree on a *similarity* criterion to distinguish how similar two solutions are. This property allows us to express scalability relative to the case using one island and allows us to compare the parallel performance independent of the randomized process introduced by the asynchronous nature of the implementation. As introduced in Chapter 3 we can use the hypervolume as a convergence criteria. We measure the time to solution as time span between start and the generation where the hypervolume s is ε-close to a reference solution $\hat{s}$. Since this measured time span does not take into account the reduction of workers *per master* node, we divide this total by the number of function evaluations

$$t_{\text{forward}} = \frac{t_{\text{total wall}}}{n_{\text{function evals}}}, \text{ and} \tag{5.1}$$

$$t_{\text{comm}} = \frac{t_{\text{total comm}}}{n_{\text{function evals}}}. \tag{5.2}$$

Clearly, the measured speedup of our benchmark problem does not necessarily transition to other multi-objective optimization problems. Due to the fact that the convergence sequence of different problems will be similar, as long as not too many duplicate individuals are computed (can be alleviated by using a global database storing hashes of evaluated forward solves and queried with a simple one-sided MPI call), we expect the parallel efficiency to be independent of the underlaying multi-objective optimization problem.

In the following we want to show that introducing additional islands is indeed beneficial for scalability. To that end, we compare the time per function evaluation of the one island case with the multiple islands case using the same amount of cores and a constant number of workers (equal to the smallest one island case). The results are shown in Figure 5.10 and Table 5.14. The results visualize that the parallel performance indeed gains from splitting the available computational resources onto multiple islands.

The reduction in communication time (see Table 5.14) is substantially reduced in the multiple islands case, by the reduction in congestion around the master nodes. For example when we compare the results for 256 cores, we see that we are latency bound for the `MPI_Recv` operations on 6 island case

$$\mathcal{L} = \frac{t}{N_{\text{calls}}} = \frac{0.095 \text{ s}}{1020} \approx 0.93\ \mu\text{s}, \tag{5.3}$$

but compared to the one island case

$$\frac{0.187 \text{ s}}{1622} \approx 0.12 \text{ ms}. \tag{5.4}$$

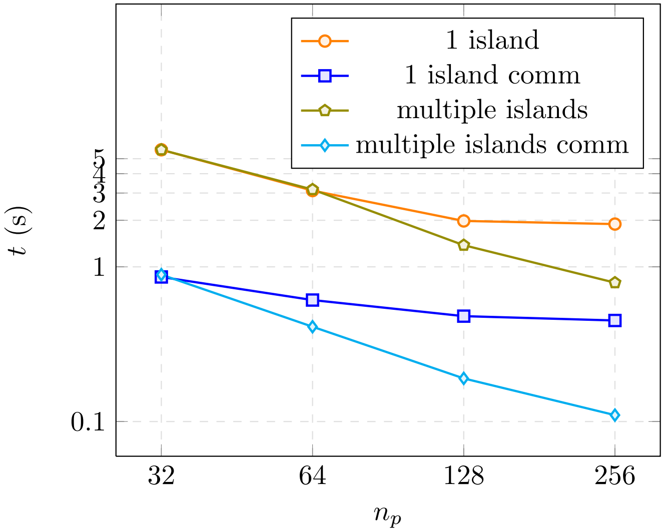


Figure 5.10: Time relative to the number of performed forward solves for one island versus multiple islands with constant island size (1 master, 1 optimizer and 30 single-core workers).

n_p	n_i	t_{forward} [s]	t_{comm} [s]	speedup
32	1	5.71	0.86	-
64	1	3.11	0.61	1.84
128	1	1.98	0.48	2.88
256	1	1.89	0.45	3.02
64	2	3.16	0.41	1.81
128	4	1.38	0.19	4.31
256	6	0.79	0.11	7.23

Table 5.14: Timings and Speedup relative to 1 master case. All islands consist of one master, one optimizer and 30 single core workers.

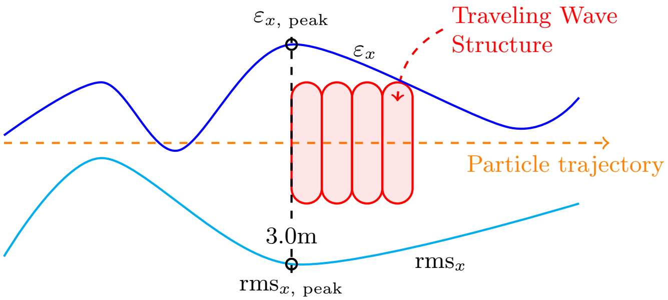


Figure 5.11: Illustration of the Ferrario matching criteria: beam emittance attains a maximum and rms beamsize a minimum at the entrance to the first accelerating traveling wave structure.

5.2 Application: Ferrario Matching

In this section, based on paper [**?**][3], we show how the multi-objective framework is used to solve optimization problems in the design of particle accelerators. As a verification and proof of concept we reproduce the Ferrario matching point discovered by Ferrario *et al.* [**?**], by formulating the problem as a multi-objective optimization problem. Using the low-dimensional and fast nature of the new simulation code Homdyn [**?**], an extensive beam dynamics study was conducted.

One of the results of the study presented in [**?**] was the discovery of a novel working point. The authors noticed that the second emittance minimum can profit from the additional emittance compensation in the accelerating traveling wave structure ensuring that the second emittance minimum occurs at a higher energy. This property is attained if the beam emittance has a maximum and the root mean square (rms) beam size has a minimum at the entrance of the first accelerating traveling wave structure. This behavior is illustrated in Figure 5.11.

By artificially reproducing this working point as the solution of a multi-objective optimization problem given in equations (5.5) to (5.13), we demonstrate the automation of discovering optimal beam dynamics behaviors given a set of desired objectives.

---

[3] submitted to SISC

$$
\begin{aligned}
\min \quad & [\Delta \text{rms}_{x,\text{peak}} = |3.025 - \text{rms}_{x,\text{peak}}|, && (5.5)\\
& \Delta \varepsilon_{x,\text{peak}} = |3.025 - \varepsilon_{x,\text{peak}}|, && (5.6)\\
& |\text{rms}_{x,\text{peak\_pos}} - \varepsilon_{x,\text{peak\_pos}}|]^T && (5.7)\\
\text{s.t.} \quad & q = 200\,[\text{pC}] && (5.8)\\
& \text{Volt}_{\text{RF}} = 100\,[\text{MV/m}] && (5.9)\\
& \sigma_L \leq \sigma_x = \sigma_y \leq \sigma_U && (5.10)\\
& \text{KS}_L \leq \text{KS}_{\text{RF}} \leq \text{KS}_U && (5.11)\\
& \text{LAG}_L \leq \text{LAG}_{\text{RF}} \leq \text{LAG}_U && (5.12)\\
& \Delta z_{L\text{KS}} \leq \Delta z_{\text{KS}} \leq \Delta z_{U\text{KS}} && (5.13)
\end{aligned}
$$

The first 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 (see Figure 5.11). The third objective (5.7) adds a condition preferring solutions that have their emittance and rms peak locations at the same z-coordinate. Equations (5.8) and (5.9) define constraints for initial conditions for the simulation: charge, gun voltage and laser spot size. Design variables given in (5.10) to (5.13) correspond to field strengths of the first focusing magnet, its displacement, and the phase of the gun.

In order to compute the peaks we employed an additional Python script. This script was called in the OPAL input file, after the simulation finished using the `SYSTEM` functionality. Once the peaks (in a given range) were located, the two objectives (5.5) and (5.6) were computed and their values written into corresponding files. The custom `fromFile` functor allows us to access the values stored in the peak finder Python script result files

```
//rmsx:  OBJECTIVE, EXPR="fromFile("rms_x-err.dat", "var")";
//emitx: OBJECTIVE, EXPR="fromFile("emit_x-err.dat", "var")";
//match: OBJECTIVE, EXPR="fabs(fromFile("emit_x-peak.dat", "var") -
                              fromFile("rms_x-peak.dat", "var"))";
```

The design variables and the assembly of the multi-objective optimization problem can be included in the OPAL input file as shown below:

```
//d1:  DVAR, VARIABLE="SIGX", LOWERBOUND="0.00025", UPPERBOUND="0.00029";
//d2:  DVAR, VARIABLE="FIND1_MSOL10_i", LOWERBOUND="110", UPPERBOUND="120";
//d3:  DVAR, VARIABLE="D_LAG_RGUN", LOWERBOUND="-0.1", UPPERBOUND="0.1";
//d4:  DVAR, VARIABLE="D_SOLPOS", LOWERBOUND="-0.05", UPPERBOUND="0.05";
```

Table 5.15: Initial conditions for the envelope tracker.

name	initial value
Gun voltage	100 MV
Bunch charge	200 pC
$\mathrm{DT}_{Beamline}$	1.5 ps
Number of slices	400

```
//objs:    OBJECTIVES = (rmsx, emitx);
//dvars:   DVARS = (d1, d2, d3, d4);
//constrs: CONSTRAINTS = ();
//opt:     OPTIMIZE, OBJECTIVES=objs, DVARS=dvars,
           CONSTRAINTS=constrs;
```

All numerical experiments in this sections were executed on the Felsim cluster at PSI. 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.

Convergence Study

The envelope tracker, introduced in Section 4.3, was chosen as the forward solver. We performed a beam convergence study in order to tune the simulation input parameters to achieve the best trade-off between simulation accuracy and time to solution. These parameters include the number of slices (`NSLICE`) used for the envelope-tracker simulations, simulation timestep (`DT`) and gun timestep (`DTGUN`).

Before the simulation can be executed a number of beam optics parameters have to be defined in an input file. Table 5.15 shows the values of these parameters for the envelope-tracker. All simulations were performed up to 12.5 m of the SwissFEL 250 MeV injector [?] beam line, with energies reaching up to 120 MeV.

The parameter that affects the performance most is the number of slices. We scanned the range from 100 to 1000 slices to determine the minimal number of slices required for stable results using various timesteps. The important results (100, 400 and 800 slices) of this scan are shown in Figure 5.12. Using this data we settled for 400 slices – increasing the slice number only minimally improves convergence of the results, therefore using more slices is inefficient.

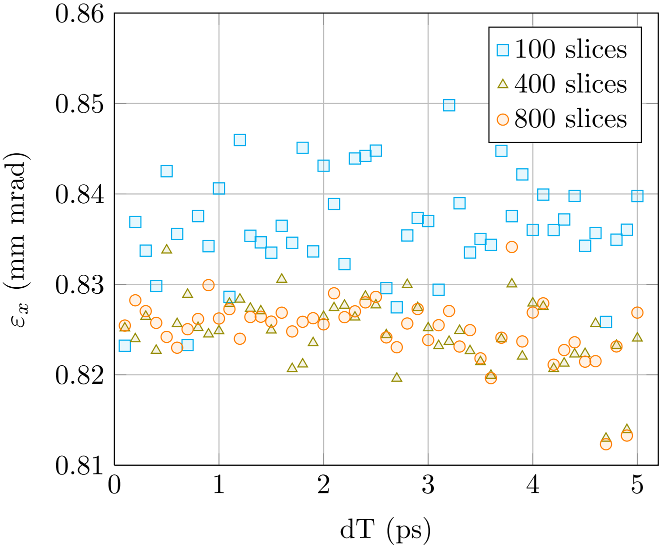


Figure 5.12: Envelope-tracker with different number of slices and simulation time steps.

In a next step the influence of different time steps was examined. To that end a series of optimization runs with 100, 400 and 800 slices and varying timestep was performed. Figure 5.13 show the Pareto front for 400 slices. We have observed that increasing the number of slices while lowering the timestep produces more detailed results.

Optimization Results

Each of the 40 points on the Pareto front, shown in Figure 5.13, represents an optimal solution, where emittance and beamsize values are compromised to achieve the best agreement with the Ferrario matching point. We selected individual 3 based on a comparison of the emittance and beamsize characteristics of all solutions and by retaining the feasibility of the beam line optics parameters. The design variables, emittance and beamsize of the selected solution are shown in Table 5.16 and Figure 5.14. With the multi-objective optimization framework we attain the same working point as reported in [**?**].

Using the input parameters of the selected solution, we performed a stability

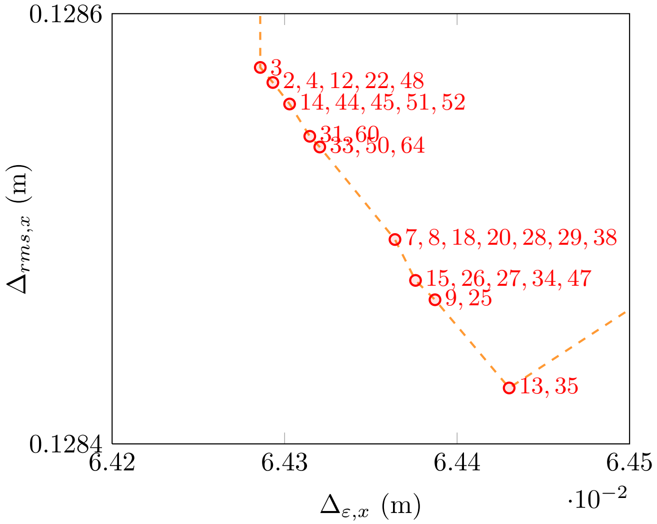


Figure 5.13: Pareto front for the 1000th generation with 40 individuals using 400 slices (interesting region magnified), a simulation timestep of 1.5 ps. The individual 3 was selected for further investigations.

analysis by varying the slice number and the time step for both the gun and the beam line. Figure 5.12 shows that the exit emittance stabilizes for 400 slices and various time steps. No difference between 800 and 400 slices is visible as their minimum maximum extension seems to be in the same range of 0.024 mm mrad.

For validation purposes we compared the results of the envelope-tracker using the analytical space charge model with the OPAL 3D macro particle tracker. The benchmark was run on the first 12.5 meters of the SwissFEL 250 MeV injector. The results for both rms beamsize and emittance are shown in Figure 5.15. A good agreement between the two codes can be observed. The difference of the larger emittance along the solenoids in case of 3D tracker that is not seen by the envelope-tracker is due to the different definition of the particle momenta (canonical vs. mechanical). Both trackers agree within acceptable limits [**?**].

The beam size and emittance optimization was successful and emittance damping is observed in the accelerating cavity. Also, the influence of the

Table 5.16: The design variables for individual 3.

name	value
σ_x	0.262 mm
Solenoid displacement	28.8467 mm
Gun voltage lag	0.0159067 MV
Solenoid current	111.426 A

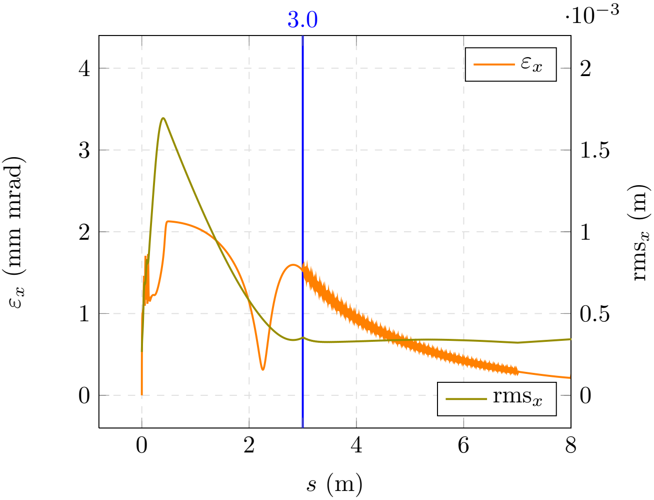


Figure 5.14: Beamsize and emittance of individual 3.

number of slices employed by the envelope tracker with respect to convergence was investigated. We found that the optimal slice number is around 400 for the problem addressed here, producing the best trade-off between computational cost and accuracy of the solution of the simulation. Similarly the gun and beam line time step was investigated confirming the convergence of the results. This first study shows that the framework is ready to tackle problems arising in the domain of beam dynamics.

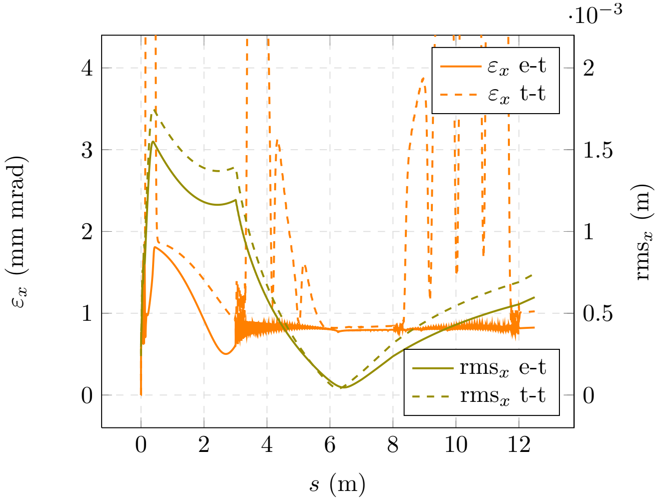


Figure 5.15: Comparison 3D-tracker versus envelope-tracker in case for rms_x and ε_x.

Chapter

6

Conclusions

In this thesis we strived to address the question if and how a scalable and general purpose multi-objective optimization framework is feasible. On the road towards achieving this goal we designed a flexible general purpose framework and implemented scalable components to tackle simulation based multi-objective optimization problems. Indeed, for the concrete setting of multi-objective optimization problems arising in the domain of particle accelerator design and commissioning, using evolutionary algorithms, we demonstrated that our approach is feasible and scalable. In contrast to other scalable simulation components or multi-objective optimization algorithms, our framework is able to be adopted in large multi-objective optimization problems arising in various fields.

For the design of the framework (see Chapter 3) we selected a master/slave approach to completely separate the simulation component and the optimizer. We believe that this approach enables a wide range of possible combinations while we retain a simple and effective interface defining the interaction between the components. It is well known though that the scalability of master/slave implementations can be quite poor. To that end we introduce multiple master/slave groups working on subproblems while using only local communication. In regular intervals we interchange solution states between neighboring master nodes on a particular graph. This graph can be tailored to the parallelization of the parallel optimizer components, e.g. a Cartesian graph. Admittedly, we introduce a small additional hurdle with respect to the parallelization in the cases where this graph has to be adapted for other optimizer algorithms. In our opinion this overhead is justified by the gained flexibility in tailoring the optimizer algorithm to the problem, in contrast to the fixed optimizer used by similar frameworks and in most cases the already implemented graphs suffice. Nevertheless, with this we can efficiently apply the framework to multi-objective optimization problems arising in a wide range of fields. Even in cases where the underlaying model is not known exactly, fixed or on the fly measurements can be used to guide the search for the Pareto optimal solutions.

An additional beneficial insight was to include the network topology in deciding the number of master/slave groups and their placement. Utilizing the network topology information to attain better scaling algorithms has been used in other fields, e.g. numerical linear algebra [**?**]. We firmly believe that both, the network awareness and maximizing the local communication, will gain in importance on the road towards exascale systems and massively parallel applications. We conducted measurements that illustrate (see Chapter 5) that this has a measurable effect on the parallel efficiency.

In order to "populate" the framework we implemented an evolutionary algorithm (see Chapter 2) making use of the PISA [**?**] framework. When we first implemented the variator we noticed a huge parallel bottleneck at the end of each generation. In some situations a few workers kept producing infeasible results while all the other workers and masters were idling. We addressed this by implementing our variator to employ "continuous generations" by passing the evaluated results to the selector as soon as k individuals have been evaluated. This successfully circumvents the previous global synchronization point at the end of each generation. Naturally, calling the selector more often increases the computational load, but on the other hand we throw away dominated solutions sooner and therefore increase the convergence rate of the algorithm. In addition, calling the selector more often does not influence the performance of the simulation component.

To solve the forward problems arising in multi-objective problems in the domain of particle accelerators, we extended OPAL by two new forward solvers (see Chapter 4). We stress that the requirements for the simulation component are twofold. First, it is very important to have parallel efficient and fast simulation models in order to achieve an acceptable overall performance of the framework. Second, in some cases, we require higher resolution simulation results. One solver reduces time to solution by reducing the dimensionality of the model and the other increasing the simulation accuracy by using an iterative solver for solving the space charge problem respecting beamline boundaries. We demonstrate excellent scalability of the solvers and the improvement in time to solution or accuracy respectively. Our parallel methods for the low dimensional solver achieve a strong and weak scalability improvement of at least two orders of magnitude in today's actual particle beam configurations, reducing total time to solution by a substantial factor.

Once we combined an optimizer and a simulation component, the end user can use the assemblage to run optimizations. We quickly realized that an easy fashion of specifying optimization problems is important and has to be extendable as well. To that end, we implemented a powerful expression parser which allows the user to specify objectives and constraints using standard mathematical operators in a convenient fashion. For users requiring more complex expression we provide a simple way to extend the parser with additional operators. In our vision this flexible expression parser combined with the framework component separation enables the framework to act as a bridge between different fields of research: specialists can focus their attention and knowledge to implement or adapt the responsible component of the framework.

Finally, users often need to be able to visually inspect the solution computed by the framework. We implemented a web based visualization service that could be run on the compute cluster directly. In our opinion implementing visualization tools as web services will be key in the future: data can be processed and inspected upon generation directly on the cluster and accessed from everywhere by browser. The presented multi-objective optimization framework, the visualization and the long standing experience in the field provides a solid basis for better understanding and improving the decision making process in the design and operation of particle accelerators.

Further Work

In this section we briefly mention possible research directions that we identified as important for further study.

Network model Our initial network introduced in Section 3.5.1 is rather simple but sufficient for our purposes. Investing in more detailed models could possibly lead to improvements in the scalability. A further investigation acquiring data for different models and their performance would reveal the impact and sensitivity of the network model.

Node task parallelization Using threads on nodes to schedule work will further enhance scalability on future and present machines. This basically brings island parallelism onto an additional sub level. Each node acts like an island, solutions are propagated up in the "island tree".

Mixing Evolutionary Algorithm Parallelization Strategies As shown in Chapter 5, the scalability of the one master setup is limited by the number of sample points on the Pareto surface and the increasing communication at the links surrounding the master node. By introducing additional master nodes, this effect can be diminished and the framework is not bound to a number of processors any more. Naturally, by increasing the number of master nodes the number of workers per master start to decrease, resulting in a slower convergence of the master (less workers equal less function evaluations). Two possible strategies can redeem this situation: we can either divide the search space amongst the master nodes to improve the time to solution or all master nodes work on the same problem and exchange solutions every k generations in order to improve the quality of the solution. A logical next step would consist of mixing both approaches. First we quickly approximate the Pareto surface by dividing the search space and in a second stage we switch to approach producing more stable solutions.

Homotopy Optimizer In a recent master thesis [?] we investigated an alternative efficient and scalable multi-objective optimization algorithm for producing evenly sampled Pareto fronts. The examined approach was proposed by Pereyra et. al. in [?]. Its key feature is to represent the Pareto front with equispaced discrete samples by adding an equality constraint that ensures a specific spacing between sample points. The method was generalized, improved and parallelized to the n-dimensional case. Additionally, a surrogate strategy replaces costly simulations by interpolation of known (previous) simulation results, reducing the computational cost by a large factor. This optimizer was successfully integrated into the framework and is subject of a future paper.

Visualization Implementing additional visualizations for Pareto surfaces can be useful when inspecting solution states. In particular dimension reduction methods and “dimensional unfolding” is a feature we plan to add in the near future. Rendering solution points in a 3D as a point surface (EWA splatting) would additionally improve the visibility of Pareto surfaces in 3 dimensions.

Control system interaction Interactions between the machine (control system) and the model can be established. Experimental physics and industrial control system (EPICS)[1], a open source software toolkit for real-time control systems, can be used to couple the model to the control system.

More applications Last but not least we would like to apply the framework to a broader set of problems from various fields of research and build a thorough collection of optimization algorithms that can be applied to solve these problems.

---

[1] http://www.aps.anl.gov/epics/

Appendix

A The opt-pilot Framework

This is a general-purpose framework for simulation-based multi-objective optimization methods that allows the automatic investigation of Pareto fronts.

The implementation is based on a Master/Slave paradigm, employing several masters and groups of workers. To tackle the emerging huge problems efficiently, we employ network topology-aware mappings of masters and slaves.

The framework is easy to extend and use with other simulation-based forward solvers, optimization algorithms and network mapping strategies.

A.1 Building

In order to build opt-pilot install following dependencies:

- CMake at least 2.8: http://www.cmake.org/
- OPAL: http://amas.web.psi.ch plus dependencies and compiled with `-DBUILD_LIBOPAL=1`
- Boost at least 1.49.0: http://www.boost.org/
- Google Tests: http://code.google.com/p/googletest/ for unit testing

Aside from the usual OPAL environment (see OPAL users guide) the cmake find modules require an additional environment variable (`OPAL_LIB_PREFIX`) pointing to the location of the OPAL library (`libOPAL.a`) and the CLASSIC library (`libCLASSIC.a`). Both libs will be expected to be in `$OPAL_LIB_PREFIX/lib`.

Make sure to build at least the boost libraries chrono, serialization and system:

```
./bootstrap.sh --with-libraries=chrono,serialization,system
./b2 install --prefix=PREFIX
```

In order to be able to compile unit tests (optional) install gtest:

1. download latest version from:
   http://code.google.com/p/googletest/downloads/list

2. unpack and change into the source directory
3. run `mkdir build; cd build; cmake`
4. install libs and include files and set `GTEST_PREFIX` accordingly:

```
cp -r ../include/gtest /usr/local/include
cp lib*.a /usr/local/lib
```

A.1.1 Homotopic Optimizer

If the homotopic optimizer is to be used, the submodule containing the code must also be initialized and cloned. This must be done from the opt-pilot root directory with

git submodule init git submodule update

or using SVN (untested)

cd Optimizer svn checkout http://svn.github.com/fosterac/Thesis.git Homotopy

A.1.2 Configuring

Create a build directory (set the environment variable `OPT_BUILD_DIR` to this path) and a `do_configure` file.

Example:

```
cmake \
    -D CMAKE_CXX_COMPILER:FILEPATH=mpicxx \
    -D CMAKE_BUILD_TYPE:STRING=RelWithDebInfo \
    -D HDF5_LIBRARIES:PATH=/path/to/hdf5/lib/libhdf5.so\
    -D HDF5_INCLUDE_DIRS:PATH=/path/to/hdf5/include \
    ..
```

Subsequently run

```
./do-configure && make && make test
```

to build and test the project.

In order to help reproducing bugs please **always** run `cmake` before building a production version (saving git hash and build date to the `config.h`).

A.2 Environment and Command Line Arguments

Set ENV variables (only required for OPAL):

- `FIELDMAPS`: folder containing fieldmaps
- `TEMPLATES`: directory containing the template input file
- `SIMTMPDIR`: directory for temporary creation of simulation files (must exist)

Program arguments:

- `--inputfile=fname`: input file containing optimization problem
- `--outfile=fname`: name used in output file generation
- `--outdir=dirname`: name of directory used to store generation output
- files (generated if non-existing)
- `--initialPopulation=num`: size of the initial population
- `--num-masters=num`: number of master nodes
- `--num-coworkers=num`: number processors per worker
- `--selector=path`: path of the selector (PISA only)
- `--dump-dat=freq`: dump old generation data format with frequency (PISA only)
- `--num-ind-gen=num`: number of individuals in a generation (PISA only)

Convergence arguments:

- `--maxGenerations=num`: number of generations to run
- `--epslion=num`: tolerance of hypervolume criteria
- `--expected-hypervol=num`: the reference hypervolume
- `--conv-hvol-prog=num`: converge if change in hypervolume is smaller

Run, example:

```
mpirun -np 4 ${OPT_BUILD_DIR}/pisa-opal/pisa-opal.exe \
   --inputfile=FiPha3Opt1.tmpl --outfile=results.dat \
   --maxGenerations=5 --initialPopulation=10 --num-masters=1
```

A.2.1 The Drivers

We provide 4 drivers (sources all located in `Drivers`):

- PISA stand-alone (`pisa-standalone.exe`): A simple driver providing a stand-alone setting for running optimizations using PISA (see `Tests/PisaStandAlone`).
- PISA OPAL (`pisa-opal.exe`): Pisa EA optimizer in combination with OPAL as forward solver (see `Tests/Opal`) for an example.
- Homotopy in both stand-alone and OPAL flavor.

A.2.2 Installing a Selector

We include PISA's nsga2 selector. Build the selector with

```
make nsga2
```

in the build directory. To use the selector either use the `--selector` option or copy the selector to the run directory:

```
mkdir nsga2
cp $OPT_BUILD_DIR/extlib/nsga2/nsga2 nsga2/
cp $OPT_SRC_DIR/extlib/nsga2/nsga2_param.txt nsga2/
```

A.3 Documentation

After running

```
make doc
```

in the `build` directory, the (Doxygen) source code documentation can be found in the `build/Documentation/`.

Appendix

Visualization Tools B

B.1 Python Visualization Script

This simple python script helps visualize the results (Pareto front) generated by the opt-pilot framework.

The visualization is pretty basic, no fancy 3D stuff and by using x and y axis plus a box size of result points we can display 3 objectives (3 dimensions).

B.1.1 Installation

To run the visualization tool the following Python libraries are required:

- python ≥ 2.7
- numpy $\geq 1.7.0$
- scipy $\geq 1.7.0$
- matplotlib $\geq 1.2.0$

On a recent Ubuntu Linux (¿= 10.4) the following command can be used to install all required packages to run the visualizer:

```
apt-get install python python-numpy python-matplotlib python-scipy
```

and on OSX the libraries can be installed using e.g. port:

```
port install python27 py27-numpy py27-scipy py27-matplotlib
```

B.1.2 Command Line Arguments

- objectives: specify 3 objectives you want to visualize (check header of result file for available objectives)
- path: specify the path of the result files
- filename-postfix (default: results.dat): specify a custom file postfix of result files

- outpath: path for storing resulting pngs
- video (un-tested): name of the video

The following example shows a common execution of the visualization script:

```
python2 visualize_pf.py --objectives=%OBJ1,%OBJ2,%OBJ3 \
                        --path=/path/to/data \
                        --outpath=/path/to/results \
                        [--generation=n]
```

The `--generation` argument only displays the nth generation.
To visualize the old data files (not JSON) use the

```
--filename-prefix=results.dat
```

argument. By default result files are assumed to be in the (new) JSON format.

B.1.3 Values of Design Variables

There are two ways to find design variables for a given individual:

- click on the colored box (click again to hide)
- open the matching result file (see below)

After selecting a point the given ID can be used to find the corresponding design variables. In order to do that open the result file of the corresponding generation and find the ID on the left-most column. After the objective values all design variables (in the order as hinted in the header) are listed.

B.2 Pareto Explorer

The following tools are required to get encounter up and running:

- node.js: `http://nodejs.org/` ≥ 0.6.x
- npm: `http://npmjs.org/` ≥ 1.x
- IE10 developer preview, Firefox 7 or Chrome 14-16 (any later versions may work but it's not guaranteed)

Node.js and npm can e.g. be installed using ports on OSX:

```
sudo ports install nodejs
curl https://npmjs.org/install.sh | sudo sh
```

B.2.1 Installation

After you install all dependencies (see above), use the following guide to get the code and start the server:

```
# Get the code
git clone git@github.com:iff/pareto-explorer.git
cd pareto-explorer

# Install node packages (external dependencies)
npm install

# Start the server and open in browser (it runs on port 3333)
./node_modules/coffee-script/bin/coffee server.coffee
open http://localhost:3333
```

B.2.2 File Format

We use JSON to transfer the set of Pareto points to the visualization server. A file can be uploaded in the corresponding mask. The JSON file should have the following format:

```
sols = [
    {
        "ID": 0,
        "ode [KeV]":      10.3988,
        "oex [mm mrad]":  2.23702e-06,
        "orx [mm]":       0.00129222,
        "velocity [m/s]": 0.68,
        "energy [MeV]":   0.004,
        "rms_z [mm]":     0.001
    },
    {
        "ID": 2,
        "ode [KeV]":      9.36923,
        "oex [mm mrad]":  1.70587e-06,
        "orx [mm]":       0.00132264,
        "velocity [m/s]": 0.58,
        "energy [MeV]":   0.024,
        "rms_z [mm]":     0.003
    },
    ...
]
```

The keys are used as axis labels or identifiers. The only mandatory field is the `ID` field. These values should be unique. An example file is provided in the `example-data` directory.

Curriculum Vitae

Personal Information

Date of birth	April 5th, 1982
Place of birth	Zürich, Switzerland
Citizenship	Eglisau ZH, Switzerland

Education

2010 – 2013	Ph.D. student and teaching assistant at the Computer Science Department at ETH Zürich.
2002 – 2008	Master of Science ETH in Computer Science at ETH Zürich with a major in computational science and a minor in compiler engineering.
1997 – 2002	High School (”Gymnasium”), Mathematisch Naturwissenschaftliches Gymnasium Rämibühl, Matura Typus C, Zürich, Switzerland.

Talks

- ICAP 2009, *A fast parallel Poisson solver on irregular domains applied to beam dynamic simulations*

- PMAA 2010, *A fast parallel Poisson solver on irregular domains*

- SIAM OP11, *Multi-Objective PDE Constrained Optimization for a 250 MeV Injector*

- SIAM PP12, *Massively Parallel Multi-Objective Optimization and Application*

- ISC 2012 (PRACE Award), *A Fast and Scalable Low Dimensional Solver for Charged Particle Dynamics in Large Particle Accelerators*,

- ICAP 2012, *A Massively Parallel General Purpose Multi-objective Optimization Framework, Applied to Beam Dynamic Studies*

Publications

- A. Adelmann, C. Kraus, Y. Ineichen, S. Russell, Y. Bi, and J.J. Yang. *The Object Oriented Parallel Accelerator Library (OPAL), Design, Implementation and Application*, Proceedings ICAP09, San Francisco, CA, 2009.
- A. Adelmann, P. Arbenz, and Y. Ineichen. *A fast parallel Poisson solver on irregular domains applied to beam dynamic simulations*, JCP, March 2010.
- C. Wang, A. Adelmann, and Y. Ineichen. *A field emission and secondary emission model in OPAL*, Proceedings HB2010, Morschach, Switzerland, 2010.
- A. Adelmann, P. Arbenz, Y. Ineichen. *Improvements of a fast parallel Poisson solver on irregular domains*, In Applied Parallel and Scientific Computing (PARA 2010). J. Kristjan (ed.). Lecture Notes in Computer Science 7133, pp. 65-74. Springer, Heidelberg, 2012.
- J. Progsch, Y. Ineichen, A. Adelmann. *A New Vectorization Technique for Expression Templates in C++*, AJUR, March 2012.
- Y. Ineichen, A. Adelmann, C. Bekas, A. Curioni, P. Arbenz. *A Fast and Scalable Low Dimensional Solver for Charged Particle Dynamics in large Particle Accelerators*, Computer Science - Research and Development, pp. 1-8. Springer, Heidelberg, 2012.