PARALLEL ALGORITHMS FOR NONLINEAR OPTIMIZATION

by

FİGEN ÖZTOPRAK

Submitted to the Graduate School of Engineering
and Natural Sciences in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Sabancı University

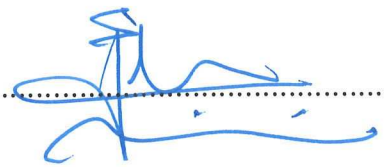
Spring 2011

PARALLEL ALGORITHMS FOR NONLINEAR OPTIMIZATION

by Figen Öztoprak

APPROVED BY:

Assoc. Prof. Dr. Ş. İlker Birbil
(Thesis Advisor) .....



Assist. Prof. Dr. Güllü Kızıltaş Şendur .....

Assoc. Prof. Dr. Orhan Feyzioğlu .....

Prof. Dr. Albert Erkip .....

Prof. Dr. Jorge Nocedal .....

DATE OF APPROVAL: June 30, 2011

Babama...

PARALLEL ALGORITHMS FOR NONLINEAR OPTIMIZATION

Figen Öztoprak

PhD Thesis, 2011

Thesis Advisor: Assoc. Prof. Dr. Ş. İlker Birbil

Keywords: parallel algorithm design, nonlinear programming, parallel optimization

Parallel algorithm design is a very active research topic in optimization as parallel computer architectures have recently become easily accessible. This thesis is about an approach for designing parallel nonlinear programming algorithms. The main idea is to benefit from parallelization in designing new algorithms rather than considering *direct parallelizations* of the existing methods. We give a general framework following our approach, and then, give distinct algorithms that fit into this framework.

The example algorithms we have designed either use procedures of existing methods within a *multistart* scheme, or they are completely new inherently parallel algorithms. In doing so, we try to show how it is possible to achieve parallelism in algorithm structure (at different levels) so that the resulting algorithms have a good solution performance in terms of robustness, quality of steps, and scalability. We complement our discussion with convergence proofs of the proposed algorithms.

DOĞRUSAL OLMAYAN ENİYİLEME İÇİN PARALEL ALGORİTMALAR

Figen Öztoprak

Doktora Tezi, 2011

Tez Danışmanı: Doç. Dr. Ş. İlker Birbil

Anahtar Kelimeler: paralel algoritma tasarımı, doğrusal olmayan programlama, paralel eniyileme

Paralel hesaplama mimarilerinin kolayca erişilebilir bir teknoloji haline gelmesi sonucu, paralel algoritma tasarımı konusu optimizasyon alanında güncelliğini korumaktadır. Bu tez, paralel doğrusal olmayan programlama algoritmaları tasarlamaya yönelik bir yaklaşımı konu almaktadır. Yaklaşımın ana fikri, mevcut yöntemleri *doğrudan paralelleştirmek* yerine, paralel hesaplamadan faydalanarak yeni algoritmalar tasarlamaktır. Dolayısıyla, önce yaklaşımıza uygun bir tasarım çerçevesi veriyor ve sonra da bu çerçevede kalan farklı algoritmalar sunuyoruz.

Tasarladığımız örnek algoritmalar ya mevcut yöntemlere ait prosedürleri *çokbaşlamalı* bir yapı içerisinde kullanmaktadırlar, ya da tamamen bu tezde geliştirilmiş yeni paralel yöntemlerdir. Bu şekilde, algoritmalarin (değişik seviyelerde) yapısal paralelliğinin, elde edilen algoritmalar iyi bir çözüm performansına sahip olacak şekilde nasıl başarılabi- leceğini göstermeye çalışıyoruz. Çalışmamızı önerilen algoritmaların yakınsama ispatları ile tamamlıyoruz.

Acknowledgments

I am indebted to my thesis advisor Dr. Ilker Birbil for all his substantial guidance, endless help, support, and encouragement throughout my Ph.D. study. He has been a wonderful advisor and teacher, and always a great friend. I owe a great deal to him for my academic progress, I can never thank him enough.

I would like to thank my thesis committee Dr. Orhan Feyzioğlu and Dr. Güllü Kızıltaş for their insightful comments that helped a lot with the progress of this research. They have been always helpful and friendly. I thank Dr.Feyzioğlu also for making Galatasaray University our Bosphorus-office.

I am thankful to the dean of Faculty of Engineering and Natural Sciences, Prof. Albert Erkip, for all his support during my graduate study at Sabancı University.

I am so much grateful to Prof. Jorge Nocedal for his invaluable mentoring and support since my visit to Northwestern University. What I learned from him greatly improved my understanding of several issues in nonlinear programming, and contributed a lot to this research. I do not know how to thank him.

Many thanks to Dr. Pınar Yolum for all her help and friendship. I enjoyed so much being a part of the MANGO project team. She made Boğaziçi University a home for us.

I have had many good friends at Sabancı University during the five years I spent there. I would like to thank all of them for their support and friendship. Special thanks to Taner Tunç, the greatest *colleague* of the world; and to Belma Yelbay, Nurşen Aydın, İbrahim Muter, Mahir Yıldırım, Ömer Özkırımlı, Halil Şen, Çetin Suyabatmaz, Nükte Şahin, Ezgi Yıldız, the best friends possible for sharing an office. I cordially thank to

my dear friends Nimet Cirnooğlu, Banu Turğut, and Engin Maşazade, whose support was always close even if they were far away. Many thanks to the nice friends and project mates at Boğaziçi University, Akın Günay and Başak Aydemir. Also special thanks to my great office mates at Northwestern University, Gillian Chin and Yuchen Wu.

Above all, I am hearty thankful to my family, my mum Nebahat, my brother İlker, and my sisters Filiz and Elmas. Without their support and patience, I could never complete this thesis. Finally, I would like to express my gratitudes to my dad Bekir Öztoprak, who I know has always been behind me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

In algorithm design, it is important to consider the following principal factors that determine the performance of an implementation: The computational requirements determined by the algorithm complexity and the problem scale, and the computational resources provided by the available hardware. The large-scale problems of the last decade are now considered as medium-sized thanks to the fast developments in computer hardware in the recent years. Consequently, the costly operations avoided in the past are now being used more frequently in algorithm design. The recent trend is, no doubt, towards parallel and distributed architectures. Nowadays, an algorithm is almost always designed by keeping in mind its parallelization. This brings the main question we focus in this thesis: How to benefit from the available parallel processing resources for solving nonlinear programming problems?

This thesis is about an approach for designing parallel nonlinear programming algorithms. In this chapter, we first introduce very briefly the nonlinear programming problem and its certain special cases that we will refer to throughout the dissertation. Then, we motivate and explain our approach. Finally, we give an overall plan of the thesis.

1.1 Nonlinear Programming Problem

Any optimization problem of n real variables can be defined using the generic form given by

$$\begin{aligned} &\text{minimize} && f(x), \\ &\text{subject to} && x \in \mathcal{F}. \end{aligned} \tag{1.1}$$

Here, $f : \mathbb{R}^n \to \mathbb{R}$ is called the objective function. The set of feasible solutions $\mathcal{F} \subseteq \mathbb{R}^n$ is, in general, defined by using a set of constraint functions and bounds on the variables, $x \in \mathbb{R}^n$. Formally, we have

$$\mathcal{F} := \{x \in \mathbb{R}^n : c(x) \geq 0,\ l \leq x \leq u\},$$

where $c : \mathbb{R}^n \to \mathbb{R}^m$ and $l, u \in \mathbb{R}^n$.

Nonlinear programming (NLP) covers those optimization problems with a nonlinear objective function or nonlinear constraint functions. In this work, we deal with nonlinear programming problems where the functions f and c are continuous, (first order) differentiable, and do not necessarily have further special characteristics such as convexity or separability.

A categorization of (1.1) is based on the set $\mathcal{F}$. When there are no restrictions on the value of x, i.e., $\mathcal{F} \equiv \mathbb{R}^n$, the resulting NLP problem is called an *unconstrained* problem. Otherwise, it is a *constrained* NLP problem. Moreover, when there are no constraint functions but only the bounds on the variables, i.e. $\mathcal{F} = \{x \in \mathbb{R}^n : l \leq x \leq u\}$, problem (1.1) is said to be a *bound-constrained* problem.

Another important definition is the *solution* of (1.1). In solving an NLP problem, we need to distinguish between the local and global solutions because the function f can have multiple minimizers in $\mathcal{F}$. A solution point $x_* \in \mathcal{F}$ is a *local solution* of (1.1), if $f(x_*) \leq f(x)$ for all $x \in N(x_*, \epsilon) \cap \mathcal{F}$ for some $\epsilon > 0$, where $N(x_*, \epsilon)$ denotes the ϵ-neighborhood of x_*, i.e.,

$$N(x_*, \epsilon) := \{x \in \mathbb{R}^n : \|x - x_*\| \leq \epsilon\}.$$

A local solution x_* is a *global solution* of (1.1), if $f(x_*) \leq f(x)$ for all $x \in \mathcal{F}$. Figure 1.1 taken from [7] illustrates the concepts of local and global minimizers on a two-dimensional unconstrained example. When the objective is to find a global minimizer, the NLP problem is called a *global optimization* problem. In the special case where both f and $\mathcal{F}$ are convex, the global and local solutions of (1.1) are identical.

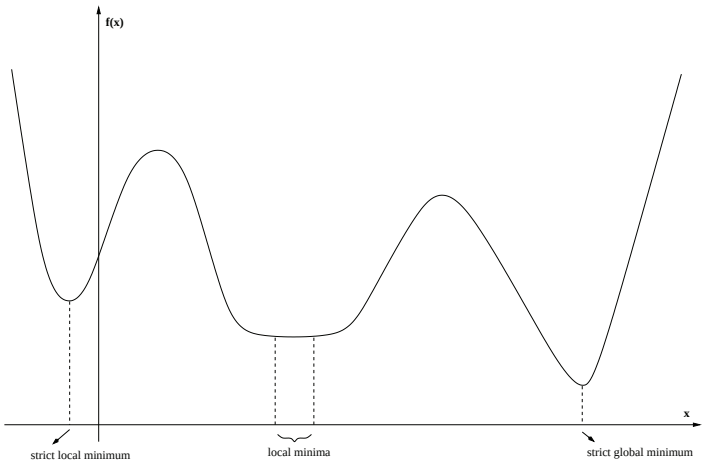


Figure 1.1: Local and global minimizers of a single dimensional problem [7]

This completes the definition of all the three classes of NLP problems covered in this study; namely, unconstrained local optimization, constrained local optimization, and bound-constrained global optimization problems. Unconstrained local optimization part of NLP has quite powerful deterministic methods. Two important groups are the line-search and trust-region methods. Global optimization has different challenges to cope with; therefore, seperate methods have been designed for the problems in that class. Since the general global optimization problem has a combinatorial nature, stochastic and heuristic methods have also been suggested. For constrained problems, there is an additional concern of feasibility besides optimality, which has lead to specific constrained optimization methods. Throughout this dissertation, we shall make use of a number of existing methods. When it comes to the details of these methods, we will refer to excellent books and reviews from the literature.

1.2 Motivations and Proposed Approach

Paradigm change in algorithm design. Considering the size and the complexity of a typical real-life problem, computational power has long been a vital resource for successful numerical optimization. Therefore, the researchers in scientific computing have a keen interest in parallel processing, which is, by all accounts, full of promises to generate the desired computational power for the resource-hungry algorithms. Following the recent developments in computer science, it is not hard to anticipate that parallel processing will not be an optional technology in the near future. First, since the improvement in the serial performance of a single processor has reached to its physical limits, parallel architectures have become unavoidable to obtain more computational power [2]. Thus, the current trend is towards the multi-level, multi-core parallel architectures [4]. Consequently, a new paradigm is in its adaptation process for scientific computing [8, 73, 45]. Currently, the multi-core architecture is the de facto standard even for rudimentary personal computers. Second, the distributed computing resources are more available today than they were ever. Cluster computing networks include hundreds of computers that can conduct computations simultaneously. This trend brings us the concern of taking the maximum benefit of those available parallel resources for solving optimization problems. Therefore, we need to adopt a new perspective and try to propose new algorithms that are inherently parallel. Nonetheless, the instruction level parallelism on such hardware is limited.

Two categories of parallel algorithms. Carrying parallel processing into NLP applications has been studied especially after mid-1980s (see Section 2.2). An existing approach to obtain parallel NLP algorithms is to *parallelize* existing methods. For example, the subproblem solution phase of the trust-region methods or the step calculation phase of the interior point methods are the most time-consuming parts of those algorithms. Therefore, a proper parallelization of those operations can provide significant speed-ups for the overall algorithms [17, 38]. The resulting algorithms aim to produce exactly the same results as their sequential counterparts; the only difference is the shared workload among several parallel processors. It is generally not a simple

issue to achieve that kind of parallelization because costly operations can be sequential in their nature, the parallelizable portion of the algorithm may not be large enough, or there may be other problems such as load imbalance, start-up and memory traffic overheads.

There are also studies where existing algorithms have been redesigned to introduce parallel tasks into their main flows. Clearly, the parallel algorithms obtained by this approach require modification to the original sequential algorithms. A nice example is parallel subspace minimization, where the subproblems obtained by projecting the original problem onto its subspaces are minimized in parallel, and then the distributed results are gathered [28]. To the best of our knowledge, there are not many studies that follow this second approach in the NLP field (see Section 2.2). The basic motivation of these few examples are reducing the cost of synchronization [13], as well as improving the workload distribution among parallel processors as in the above example. As we shall point out in Section 2.2, examples of this kind of parallelization can be mostly found in derivative-free optimization as well as in global and combinatorial optimization implementations.

In this thesis we propose designing algorithms in the second category above, which can also include parallelized portions in the sense of the first category. However, we do not only try to obtain a nice parallel performance but also to further benefit from the available parallel processing resources as we shall elaborate below.

The proposed approach. An algorithm can be more effectively applied on a parallel architecture, if it contains computationally-intense blocks of independent tasks that can be executed concurrently. Thus, if we can partition the algorithmic operations in a proper way, we may expect to achieve a better parallel performance. Let us define *designed-as-parallel* algorithms as inherently parallel algorithms, which are not necessarily direct parallelizations of any sequential methods. However, unlike some algorithms following the second approach mentioned above, the primary concern of our focus on designed-as-parallel algorithms is not only the workload distribution among parallel processing units, but also obtaining new, hopefully well-performing, standalone

algorithms. An important feature for designed-as-parallel algorithms is that they benefit from parallel processing to execute additional operations that may or may not look acceptable from a sequential point of view. An early nice example along these lines is the parallel variable metric algorithm [65, 71]. Here, the approximate Hessian matrix is updated along several independent directions instead of just the previous search direction as in the sequential method. In this example, the parallel computational power has provided a better way of approximating the curvature of the objective function. This example brings us to another important advantage of designed-as-parallel algorithms: parallel generation and use of problem information. Notice that designed-as-parallel algorithms can always include the first approach to parallelization. That is, while executing additional tasks, we may also distribute the costly operations. A good example is the multi-step multi-directional parallel variable metric algorithms [57]. In this example, multiple search directions that use different quasi-Newton update formulas are computed in parallel. Then, a parallel line-search routine is applied along each direction by doing parallel function and gradient evaluations which is, though not exactly, very close to the first category of parallelization of a cubic interpolation procedure (see Section 2.2.2).

In this thesis, we concentrate our efforts on an alternate point of view for using parallel processing in nonlinear programming applications: parallel computation can contribute to the performance of solution approaches not only by providing faster execution of their operations but also by executing additional tasks that may improve their operations.

Why would our approach work? When we consider general nonlinear optimization, the quadratic or superlinear rates of convergence are valid in the close vicinity of a solution point. Furthermore, it is hard to predict how fast an algorithm arrives to that close vicinity starting from an arbitrary point, even if the algorithm is globally convergent. We just restate what is well-known in NLP field; the practical performances of the algorithms can be quite problem dependent. Moreover, almost all practical algorithms use incomplete problem information. There are several ingenious examples in the literature that motivate using additional information in traditional algorithms

like memory and hybrid algorithms, see for example [32, 10, 49, 27]. These works have demonstrated how useful it may become to use some extra information within the original routines. Here, we suggest using parallel resources for achieving this in a way that the resulting overall algorithm has an inherently parallel structure suitable for execution on parallel processors. The approach we propose here emanated while we were working on the parallelization of a method for global optimization [54]. Our initial ideas were on the global optimization applications but soon we realized that these ideas could be generalized. Findings that came out of a literature survey on parallel and hybrid methods in local and global nonlinear optimization provided us further motivation, some of which are mentioned above. We then conducted some preliminary tests and obtained supportive numerical results.

A general framework. The approach we follow in this thesis is constructed on the basic ideas mentioned above. That is, we try to achieve two objectives in two ways: (1) good parallel-execution features, by a designed-as-parallel algorithm structure, (2) good solution performance, by using the *extra* problem information produced on parallel resources. This suggests a framework with the following three properties:

1. The overall workload consists of *blocks of tasks* that can be executed in parallel.
2. Some of these tasks are included to provide *extra problem information*.
3. There is an *interaction among the tasks* which enables information exchange.

Consequently, we try to design tasks that provide both categories of parallelizations mentioned above. It is important to consider the first category because providing an improvement in overall solution time may not be even possible without the additional gain one can obtain by extracting the existing parallelism in the originally sequential tasks. In this way, we aim to have a fine task-based structure with a high level of inherent parallelism that is suitable for concurrent calculations, which can further benefit from parallelization in developing efficient optimization algorithms.

In this thesis, we discuss different algorithms that apply the proposed approach at various levels for solving different classes of NLP problems. In Section 1.4, we give

an overall explanation of how they fit into the above described framework.

1.3 Contributions of The Thesis

Parallel processing enables performing concurrent operations. Given a certain computational task, we expect to decrease its execution time by distributing its computations among parallel processors. This is, in fact, the most common way used for designing parallel algorithms.

This thesis proposes an alternative point of view for designing parallel algorithms. Instead of following the path from sequential to parallel environments, we suggest to design algorithms that would fit into a parallel architecture from the start so that we do not carry the limitations of a sequential style of thinking into a parallel one. That is, we go through the opposite way and try to come up with new *designed-as-parallel* algorithms. Although the new algorithms themselves can be considered as sequential algorithms, we should note that having parallelization in mind itself does lead us to come up with such algorithms; otherwise, the proposed algorithms probably would have never been considered as a new sequential algorithm. We emphasize once more that we do not reject including conventional parallelization, e.g. parallel execution of linear algebra operations.

We describe a general framework and introduce different example algorithms following the approach above in different ways. Each of these algorithms is concentrated on a certain component of that approach to be able to provide more idea about its potential. These attempts have resulted in two completely new algorithms (Chapter 4), and two new mechanisms for extending the existing ones (Chapter 3).

Before we continue, we should note that our main purpose is to come up with general-purpose algorithms. Needless to say, a problem with a special structure can be solved in parallel with a greater efficiency using a problem dependent algorithm. However, such an approach would most probably be limited for solving general nonlinear programming problems.

To this end, we also need to state that designing new parallel algorithms is a very demanding task. After his experience on parallel nonlinear programming algo-

rithms, Schnabel [61] concludes in 1995 that *'... the consideration of parallelism has not led to the development of existing new general purpose optimization methods for small to medium size optimization methods. Instead, the capabilities of parallel computers have best been utilized by parallelizing existing sequential algorithms.'* However, his observations are not completely discouraging; in particular, he indicates that *'... once one considers large-scale problems, there are many opportunities for the development of interesting new parallel optimization algorithms, including possibilities that superior sequential methods may be discovered through this process.'*

1.4 Outline of The Thesis

There are three main parts in the rest of this dissertation. In the the first part (Chapter 2), we briefly introduce the basic issues in parallel computing. Chapter 3 and Chapter 4 are dedicated to the algorithms that are designed by following the ideas described in Section 1.2 in different ways and at different levels. These two chapters constitute the last two parts. We summarize the contents of Chapter 3 and Chapter 4 in Figure 1.2. This figure shows how each one of the proposed algorithms fit into the general framework described in Section 1.2. As noted in this figure, in each section we mainly aim to emphasize a certain component or property of the proposed approach.

The algorithms of Chapter 3 are higher level examples in the sense that their tasks executed in parallel include the procedures of several identical or distinct existing methods. Therefore, we call them *multistart strategies*. The main emphasis of that chapter is on the exchange of information produced in parallel in a way that the overall solution performance is improved. The resulting parallel algorithms are expected to behave different from the included individual methods. The framework introduced in Section 3.1 enables the use of multiple model functions and multiple parameter values in interaction. Its extensions to global optimization is described in Section 3.2, which finally yields an environment where cooperative optimization agents implement (different) methods concurrently.

In Chapter 4, we introduce two new algorithms which are lower level examples following the proposed approach. The main emphasis of this chapter is achieving

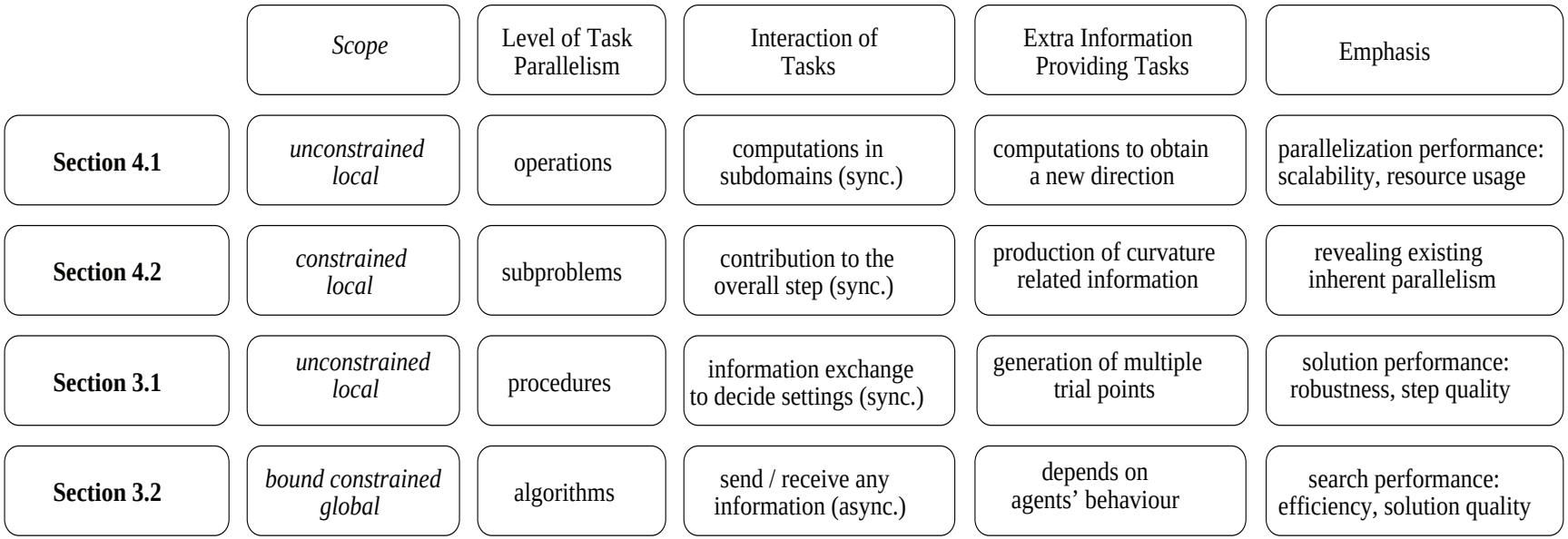


Figure 1.2: Overview of the thesis content

inherent parallelism, while introducing additional tasks to generate more problem information. The algorithm in Section 4.1 tries to generate and use a composite model function by creating a completely separable workload. In Section 4.2, the idea is to reveal the existing inherent parallelism in the structure of the NLP problem itself. While improving the parallelization features, we try to achieve a good solution performance by including additional computations.

Chapter 2

LITERATURE REVIEW

In this chapter, we first introduce some basic parallel programming terminology. Then, we present an overview of the existing work on parallel optimization algorithms.

2.1 Parallel Programming Overview

Two important milestones in the evolution of parallel computing technology until 80s are accepted to be the Iliac IV project with its achievements on processing arrays of data efficiently and the development of CRAY 1, which has introduced the concept of *vectorization* [11]. Among the other important developments since then are the rise of the *multithreading* paradigm, the emergence of the *multicore* architectures, and the availability of large *grid-computing* resources. These developments on the hardware side had a direct effect on programmers and algorithm designers. Consequently, several different parallel programming models along with numerous issues have emerged. In this section, we will only cover some of these concepts and issues that are referred in the subsequent parts of this dissertation.

Parallelism. In developing parallel algorithms, *data level* and *task level* decompositions of the overall workload can be considered [58]. Data level parallelism is available, if the same operations are to be applied to the small portions of a large amount of data. On the other hand, task level parallelism exists if there are multiple independent operations to be applied to the same data. The latter is also referred as *control*

level parallelism [11]. Apparently, both types of decompositions can be implemented in the same algorithm. One such example is the pipelining structure where different operations are implemented to different blocks of data at each given time unit.

Throughout this dissertation, we refer to the concept of *inherent parallelism* many times to define a computational workload which can be divided into computationally dense parts in a straightforward way. This can include both data and control level decompositions.

Synchronization. Most of the time, the overall workload of a program cannot be divided into completely independent parts. A simple example occurs when there is a precedence relationship among the tasks of an algorithm. This requires the *synchronization* of the ongoing operations on parallel processing units. There are many problems that are caused by the need for synchronization: On a shared memory architecture, synchronization increases the memory traffic overhead, whereas for an implementation on a distributed memory system, it increases the cost of communication. Synchronization is a very important factor causing *load imbalance* among processors. As a direct consequence, some parts of the resources stay idle.

A measure that compares the time spent for parallel computations to the time spent for communication is *granularity*. A parallel algorithm is said to have *fine-grained* parallelism, if it has many synchronization points per unit time, and *coarse-grained* parallelism otherwise.

Speed-up. A parallel application is expected to take less time than its execution on a single processor. The ratio of the parallel and serial execution times is called *speed-up*, and this is one of the basic performance measures used in parallel programming. To give a formal definition, let $T_*(n)$ denote the complexity of the best serial implementation of an algorithm on a given n-dimensional input data, and $T_p(n)$ be the complexity of the parallel algorithm on p processors. Then,

$$\text{Speed-up} := \frac{T_*(n)}{T_p(n)}.$$

We can never expect to see a speed-up over p. In fact, speed-up has another more restrictive upper bound as stated by the so-called Amdahl's Law. Let r_s denote the sequential portion of program, and $r_p = 1 - r_s$ be the portion executed in parallel. Then, Amdahl's Law dictates the following relation

$$\frac{T_1(n)}{T_p(n)} \leq \frac{1}{r_s + r_p/p}.$$

A related performance measure is called *efficiency*, and it is defined as

$$\text{Efficiency} := \frac{T_1(n)}{pT_p(n)}.$$

Apparently, the efficiency of a parallel execution cannot exceed 1, and it is generally smaller than this upper bound due to the sequential portions of a program or due to the synchronization-related inefficiencies.

Scalability. A parallel program is said to be *scalable*, if it achieves a better speed-up as the number of processors increases. However, this definition limits the scalability of an application with Amdahl's Law. In fact, Amdahl's Law is based on the assumption that the overall workload of a program stays constant as the number of processors increases, and this law also suggests that one needs to reduce the sequential portion of a program to improve speed-up. As an alternate approach, Gustafson suggests increasing the work done in parallel by executing the program on a larger data set [58]. Following this approach, a program can said to *scale* as it is implemented on more processors, if it keeps its efficiency when the dimension of the input data increases proportional to the number of processors. That is,

$$\lim_{n\to\infty} \frac{T_p(n)}{T_*(n)} = 0, \qquad \text{when } \frac{n}{p} \text{ is constant.}$$

Concurrency. *Threads* are parts of a program that share the data of the program but run independently unless they are synchronized explicitly by the program itself [58]. Multithreading brings *concurrency* to an application but it also raises new concerns.

For example, it is important to successfully manage the access of multiple threads to the global variables of the program because otherwise conflicts can easily occur.

2.2 Parallel Optimization

Parallel computing technology has long been used in numerical optimization. In this section, we try to give an overview of the existing work on parallel optimization, with a special emphasis on parallel nonlinear optimization algorithms. Our objective is not covering all the related work in the field, but rather providing an overall view by mentioning some examples in different categories. More comprehensive reviews of existing perspectives and different applications, particularly in nonlinear programming, can be found in [48, 61].

2.2.1 General Optimization

There are various examples of parallel algorithms that have been designed for solving different optimization problems. As nicely put in [11], the parallelism present in an optimization process can be related to the particular problem at hand or to the algorithm implemented.

Parallelism in problem structure. Certain problems are naturally separable. Consider, for example, an optimization problem with the objective function

$$f(x_1, \cdots, x_n) = f_1(x_1, x_K) + f_2(x_{k+1}, x_n),$$

where $k < n$. Clearly, any optimization procedure can be separately (possibly in parallel) implemented to the two components of f. An optimal solution point to this problem is then obtained by merging the solutions found in the two independent subdomains.

A nice example of parallel problem structure is found in a typical *global optimization* problem. Solving a general global optimization problem requires a complete search in its feasible domain. Then, it is possible to partition the feasible region of this problem, and conduct search in each part of the feasible region in parallel. This strategy

can be implemented in a clever way along with the branch-and-bound (BB) technique [18]. The idea is the communication of the information obtained in different branches. For instance, an upper bound on the global objective function value is found by one of the branches and then communicated to the other branches. However, the existence of parallelism does not mean that the implementation is straightforward. The difficulties, like the construction of the parallel search tree and the load balancing, arise in these implementations.

It is also possible to follow the domain decomposition idea in a stochastic manner for large unstructured global optimization problems. If the parallel search is done by multiple starts of the same (local) method and each of these executions start from a random point in a particular subset of the feasible region, this corresponds to a parallel multistart method. Various implementations of this general idea can be found in [18, 22, 59].

Another example that further exploits the parallelization in problem structure is a special *convex programming* problem, where the feasible region is defined only by bound constraints. This property makes the originally complex problem of computing a projection onto the feasible region easy and completely separable. Therefore, a gradient projection algorithm can be implemented easily. On the other hand, when the objective function is given by $f(x) = \sum_{i=1}^{m} f_i(x_i)$, where each f_i is a strictly convex function, and the feasible region is defined with a set of linear equations, then the dual problem becomes unconstrained. Thus, an unconstrained method suitable for parallel implementation is applicable [69]. A similar idea is also suggested for the subgradients [40].

A related case is the block separability of the matrix of constraint coefficients, which enables a natural decomposition of the problem again thanks to the structure of the set of constraint functions as well as the objective function. Problems with this

structure can be written in the form

$$\begin{aligned} \text{minimize} \quad & f(x) = \sum_{i=1}^{m} f_i(x_i) \\ \text{subject to} \quad & x_i \in \mathcal{F}_i, \quad i \in 1, \ldots, n \\ & x \in \Omega. \end{aligned}$$

Two example problems with this property are the *multicommodity network flow* problems and the linear programming formulations of *two-stage stochastic programming* problems. The (transpose of) constraint coefficient matrices of these problems has the following structure

$$\begin{pmatrix} A_1 & & & \\ & A_2 & & \\ & & \ddots & \\ & & & A_m \\ B_1 & B_2 & \ldots & B_m \end{pmatrix}.$$

Note that the last line violates block-diagonality of this matrix. Therefore, those problems can be solved in parallel first by modifying the original problem in a way that the constraints $x \in \Omega$ are no more explicitly present, for instance, by moving them to the objective function with penalty parameters. Then, the modified problem consists of independent subproblems, and hence, can be solved in parallel. This procedure is repeated until the modified problem yields a sufficient approximate solution to the original problem. This overall procedure is referred to as the model decomposition approach [11].

Parallelism in algorithm structure. Consider an iterative procedure in the form

$$x_i^{next} = T_i(x^{current}), \quad i = 1, 2, \ldots,$$

where $x^{current}$ denotes the current iterate, and x_i^{next} is i^{th} component of the next iterate point provided by the operator T_i. This quick example illustrates what we mean by par-

allel algorithm structure: The overall procedure has apparent independent components, which can be executed in parallel.

To emphasize the importance of parallelism in algorithm structure, let us consider two methods applied for solving *linear programming* problems, namely the simplex and the interior point methods. In fact, the general class of active set optimization algorithms, including the simplex method, are mostly sequential in nature. Although there are successful parallel implementations of the simplex method for certain special types of linear programming problems (see for example, [41]), there are no parallelizations of the method providing significant performance improvements over its sequential implementations for general sparse linear programs [35]. Interior point methods, on the other hand, seem to be more suitable for scalable parallelizations. The primary reason is the fact that the number of iterations requires to solve a given problem by an interior point algorithm is not very sensitive to the size of the problem. So, as the problem size gets larger, the interior point method is expected to converge in around the same number of iterations but the computational cost of each iteration increases. It is important to note that the main part of the computational cost of a single iteration is due to solving a linear system. So, its successful parallel implementations can be considered along with successful parallel linear algebra routines, like sparse Cholesky factorization routines [1, 38].

The pattern search algorithms of *derivative-free optimization* is another good example of inherent parallelism in algorithm structure. Pattern search methods basically compute the objective function value at some points that are selected with respect to some specific pattern, like the vertices of a simplex. The current best is determined as the point with the best function value. The dimensions of the pattern is expanded or contracted until an improving step is achieved. This process is repeated with a new set of points until some termination criterion is satisfied. Clearly, each new point selected within the neighborhood is located on a direction originated at the current best. The parallel pattern search algorithms simply distribute these directions among parallel processors and parallelize the function evaluations. The more interesting application is the asynchronous version of the idea, where computations on different directions are again

done in parallel but now each thread holds its own copy of the current best. When a thread finds a value better than its current best, it broadcasts it to others, and when a thread receives a message with a current best better than the one it has, it simply replaces its current best information. It has been shown that the asynchronous processes will eventually converge to a stationary point of the objective function [37, 43].

A distinct example for this type of parallelism is the *asynchronous team* (A-team) environment, which has been originally designed for solving *combinatorial optimization* problems. An A-team is defined as a set of autonomous agents and a set of memories that are connected through a cyclic network [67]. Each agent applies (concurrently) some algorithms or modification operations on the solutions selected from its input-memory. The agents are possibly heterogeneous, and there is no central coordination or planning mechanism. To provide cooperation, input and output memories of agents are connected through communication channels so that an agent may select the output of another agent as its input. The system terminates when a persistent solution is obtained. The approach has been successfully adopted for solving different problems including the global optimization problems [63, 70].

2.2.2 Nonlinear Programming

Existing parallel nonlinear optimization studies can be classified under two categories that follow two main ideas for developing a parallel algorithm:

1. To parallelize an existing sequential algorithm.

2. To design a new inherently parallel method.

In this section, we will follow this categorization, and mainly discuss some examples of the work in the second category since they are highly relevant to the approach we follow in this thesis. We should mention that there are not many examples of parallel local nonlinear optimization algorithms that fall into the second category. We shall only give the main directions of the studies in the first category here. For more extensive review studies on parallel nonlinear optimization, we refer to [21, 12, 19, 48, 61].

Direct parallelizations. The significant part of the computational cost of a local optimization algorithm is generally related either to the function evaluations or the linear algebra operations. Several studies in the literature are basically based on the idea of increasing efficiency by parallelizing these costly parts of the existing sequential algorithms. The resulting algorithms produce completely the same results as their sequential counterparts (except the differences caused by round-off errors accumulated in different ways because of the parallel implementation). The only difference is the shared workload among several parallel processors. Careful implementations of this approach can achieve very good parallelization performance (see for instance, [5]).

In some problems, the function evaluations may be very time consuming and dominate the cost of linear algebraic operations. This can be, for example, due to the need for processing a huge amount of data to compute the functions. In this case, the parallelization of function evaluations can provide significant speed-ups. Computing or approximating the derivatives of the objective function and constraint functions can also be done in parallel (for example see [33], where automatic differentiation approach is used along with graph coloring techniques). If finite difference approximations are used, the parallel structure comes directly from the need for multiple function evaluations [60]. When the objective function can be formulated as a separable one, the gradient and Hessian calculations can be distributed among the parallel processors such that each process is only responsible for the computations corresponding to the certain components of the objective function [3]. The efficiency of function evaluations is crucial for the performance of nonlinear optimization algorithms, but parallelization of only these operations can hardly be seen as parallel optimization algorithm design.

On the other hand, especially for larger dimensional problems, efficiency of linear algebra operations may become more or equally critical as compared to the function or derivative evaluations. In this case, executing linear algebra operations in parallel may contribute to the algorithmic efficiency to a certain extent. Most basic linear algebra operations have already been redesigned and coded to be executed on parallel machines, like the excellent ScaLAPACK routines [55]. D'Apuzzo et al. [19] give a nice review of implementations that incorporate parallel linear algebra routines into

nonlinear optimization algorithms. Clearly, the success of the algorithms that follow this idea is highly dependent on the success of parallel linear algebra routines among many other issues. One important related subject is the parallelization of (sparse) matrix factorizations [36]. The success in factorization directly affects the parallel performance of (direct) interior point solvers of nonlinear optimization. Another hard but also very important related subject is the design of parallel preconditioners. Such a design would certainly affect a large class of nonlinear optimization algorithms like the inexact-Newton methods. However, as in the case of sequential preconditioning, it is hard to find out the absolutely best strategy: the scalability of a preconditioning implementation seems to be highly dependent on the (sparsity) structure of the problem at hand as well as the choice of the preconditioner [34].

Sequential to parallel. There are also parallel nonlinear optimization algorithms in the literature that are not direct parallel extensions of the existing sequential algorithms, but either modify the basic methods they extent or are new parallel algorithms in themselves. In fact, examples of this category can be mostly found in derivative-free optimization [20, 42] as well as in global and combinatorial optimization [18, 66, 67]. As a side note here, in some studies on combinatorial problems, it has been argued that this type of parallel algorithms may provide not only speed-up, but also better solutions within the same execution time [15, 16, 30].

An interesting example of parallel nonlinear programming algorithm in this category is the *parallel gradient distribution* proposed by Mangasarian [44]. Here, the n-dimensional gradient vector is distributed among p parallel processors so that processor l has the part $\nabla f(x_i)^l \in \mathbb{R}^{n^l}$ so that $\sum_{l=1}^{p} n^l = n$, $l = 1, \ldots, p$. Each parallel processor l runs an independent sequential local optimization algorithm, like quasi-Newton and conjugate gradient methods, for one or more iterations using $\nabla f(x_i)^l$. That step results in different search directions $d_i^l \in \mathbb{R}^{n^l}$ and step-sizes $\alpha_i^l \in \mathbb{R}$ obtained by the p processors, which are then synchronized by a convex combination operation to compute the

next iterate,

$$x_{i+1}^l = x_i^l + \nu_i^l \alpha_i^l d_i^l, \quad l = 1, \ldots, p,$$
$$\sum_{l=1}^{p} \nu_i^l = 1, \quad v_i^l \geq \delta > 0.$$

This synchronized parallel search is repeated until some termination criterion is satisfied. For the convex case, the algorithm has proven to be convergent to a point that satisfies first order necessary conditions. The descent condition on f required in each subspace plays an important role in the convergence. In numerical tests conducted by applying inexact Newton methods speed-up factors over 44% for $p \leq 16$ and $n \leq 1024$ have been observed.

We should also mention here another related work by Ferris and Mangasarian[25], which extends the idea in parallel gradient distribution by assigning each parallel processor an $(n^l + p - 1)$-dimensional subproblem with the decision vector (x^l, λ^l). The additional variables λ^l are multipliers that allow the lth processor make changes in the complement subspace $\mathbb{R}^{n^{\bar{l}}} = \mathbb{R}^n - \mathbb{R}^{n^l}$. So, each processor can compute a trial next iterate $\tilde{x} \in \mathbb{R}^n$ in the form $\tilde{x} = (\tilde{x}^l, x^{\bar{l}} + D^{\bar{l}}\tilde{\lambda}^l)$. Here, $D^{\bar{l}}$ consist of some descent directions in the subspaces other than $\mathbb{R}^l$. The resulting method is called *parallel variable distribution*. Fukushima [29] has proposed a generalization of this idea that includes both this algorithm and the parallel gradient distribution explained above, and he has shown the global convergence for this general class.

An extension of the above idea for constrained optimization is to distribute the constraints of the problem among p parallel processors [24]. Each processor then solves a subproblem with a smaller set of constraints, and with an objective function that is modified by adding the rest of the constraints to the augmented Lagrangian terms. The convergence of the procedure has been shown for convex programs.

Another interesting example is the multi-step, multi-directional quasi-Newton algorithms proposed by Phua et al. [57]. Here, at each iteration, p_1 parallel processors compute alternate search directions by applying different quasi-Newton update rules. That is, letting H_k^j denote the approximate inverse Hessian produced by the update

formula j, the directions

$$d_k^j = -H_k^j \nabla f(x_k),\ j = 1, \ldots, p_1$$

are computed. Then, a parallel cubic interpolation is implemented as the line search strategy with concurrent function evaluations at p_2 points along each direction d_k^j. Thus, overall $p_1 p_2$ processors are required. The direction and step length providing the lowest function value is selected as the next iterate. Finally, the inverse Hessian is updated with the BFGS formula to start the next iteration with a positive definite approximation. In numerical tests, which have been conducted with three update formulas and a total of 9 processors, the number of parallel function and gradient evaluations decreased up to 200% and this ratio had been shown to be as large as 28 times for some large-scale test problems.

The idea applied by Straeter [65] and van Laarhoven [71] in earlier papers has another motivation: they suggest calculating the gradient vectors in n independent directions in parallel, and use these values for updating the approximate Hessian. In other words, given n vectors of length ϵ in n linearly independent directions, $\delta^1, \delta^2, \ldots, \delta^n$, the gradient values are computed at points $x_k^j = x_k + \delta^j$ for $j = 1, \ldots, n$ and the approximate Hessian matrix is partially updated with each $((\nabla f(x_k^j) - \nabla f(x_k)), \delta_j)$ pair in a consecutive way. Both studies mentioned above have implemented the idea with rank-one update, and the authors have also shown the quadratic convergence of their methods. However, these methods may fail in practice because the approximate Hessians are not guaranteed to be positive definite. Freeman [26] applies the same idea with an update formula that provide positive definiteness. In practical tests, the resulting parallel quasi-Newton method has been observed to require less iterations than the existing sequential quasi-Newton algorithms.

Finally, let us note that the asynchronous parallelization of any algorithm would end up in a parallel method in this category. Let us complete this section with such an asynchronous parallel global optimization algorithm given in [6]. Here, each parallel thread executes the same sequential clustering algorithm starting from different initial

points as in a usual multistart application. In addition, there is a cooperation among the threads: Local minima obtained in any individual execution is kept in a shared memory so that the parallel threads check this set of known local solutions before they start a new execution, and try not to select an initial point in the attraction regions of already explored solutions. Clearly, the resulting parallel algorithm is asynchronous, and therefore it does not necessarily behave the same as its sequential counterpart.

Chapter 3

PROPOSED PARALLEL MULTISTART STRATEGIES

In this chapter, we introduce a framework for unconstrained optimization that follows the approach described in Section 1.2, and its extensions to global optimization. The framework performs in parallel multiple step computation procedures that interact in a certain way. This is why we categorize the framework and its extensions under *multistart strategies* heading.

3.1 Concurrent Search Framework

The concurrent search (CCS) framework is designed for solving the unconstrained NLP problem

$$\begin{aligned} &\text{minimize} \quad f(x), \\ &\text{subject to} \quad x \in \mathbb{R}^n. \end{aligned} \tag{3.1}$$

The proposed framework does not necessarily set forth completely new algorithms. Instead, we try to *improve* the performance of existing methods by executing *additional operations* in parallel as suggested in Section 1.2.

Basic idea. A basic challenge in solving the general unconstrained NLP problem is to find a path to the close neighborhood of a minimizer without failing or wasting too much effort in far-away regions. It is well-known that parameter initialization and update procedures are very influential on the performances of the solution methods.

Moreover, the success of a solution method heavily depends on the problem at hand. It is quite hard to guess which procedure would produce more successful steps starting from an arbitrary initial point.

In CCS, we shall try to use procedures which apply different solution methods or different parameter settings, and interact within an overall algorithm that is suitable for parallel execution. That is, we shall compute multiple trial points at each iteration of an overall algorithm. The main idea is to use different search procedures for computing multiple trial points. Then, based on these computations, the procedures share relevant problem information and aid each other in adjusting their own parameters. We then decide the next iterate by selecting the most successful trial point with respect to some success measure.

The framework does not exclude the parallel implementations of the individual step computation procedures, but it tries to use parallel processing to achieve further performance improvement by producing more successful overall steps.

General framework. To further explain and formalize the above idea, we provide one possible generalization of the proposed framework for p trial points. The outline is given in Algorithm 1. Let $\hat{x}_{k+1}^t$, $t = 1, \ldots, p$ denote the trial points computed at iteration k. Each trial point is obtained by a first order convergent iterative local search algorithm, and the acceptance test for the new trial point is also carried out by the same algorithm (line 9). A trial step computation may consist of, for example, the solution of a trust-region subproblem or a steplength computation of a line search method. It may also be the case that some of the algorithms are not applied completely at each step; for instance, a steplength search procedure may be distributed over several iterations of the concurrent search algorithm (detailed examples are given in Section 3.1.1).

When p trial points are computed at iteration k, there are three possible cases:

1. $\hat{x}_{k+1}^t$ is the only acceptable trial point. In this case, the algorithm accepts this point as the next iterate. That is $x_{k+1} = \hat{x}_{k+1}^t$.

2. There are multiple acceptable trial points. Then, the algorithm selects the trial point which provides the largest improvement in the objective function value.

Formally,

$$x_{k+1} = \arg\min_{t \in A} f(\hat{x}^t_{k+1}),$$

where A is the index set of the acceptable trial points (line 13).

3. There is no acceptable point. In this case, a new set of p trial points are computed by applying some usual backtracking-type operations like shrinking the trust-region radius or reducing the step size of the backtracking line-search.

This defines the trial point evaluation step (line 12), which is a synchronization point. When it is completed, the stopping criteria are checked, and if CCS does not terminate, then the next parallel iteration starts. Before starting the computation of the next trial points, the p algorithms refresh their parameters according to the problem information generated in the previous step by all included algorithms and the output of the trial point evaluation step. In particular, algorithm t updates its current iterate and parameters using the external information provided by others, if the previous step of CCS was successful but the trial point computed by algorithm t was not selected (line 6); otherwise, it applies its usual parameter update phase (line 8).

The resulting parallel algorithm always progresses from a single point, and hence, the derivative evaluations are shared among the threads. Clearly, this provides a memory usage advantage as well as computational savings in a shared memory architecture. Moreover, this structure is appropriate for obtaining further performance improvements by parallelization of these common derivative evaluations as well as the possible costly linear algebra operations within the step computation procedures. When those further parallelizations are included, an implementation of CCS may be executed by more parallel threads than the number of included algorithms. However, let us ignore further granularities here for simplicity, and assume that the operations of each algorithm is assigned to a single thread.

3.1.1 Implementation

In this section, we illustrate the CCS idea with two example algorithms. In the first example, we use two trial points ($p = 2$) at each iteration. We name the algorithm as

Algorithm 1: Concurrent Search

Input: x_0, p
$k = 0$;
while *checkStopping()=FALSE* **do**
  /* Begin Parallel Tasks */
  for $t = 1, \ldots, p$ **do**
    if $x_k \neq x_{k-1}$ *&* $x_k \neq \hat{x}_k^p$ **then**
      Set/Update parameters of (incomplete) algorithm t using information provided by algorithms $\{1, \ldots, t-1, t+1, \ldots, p\}$;
    else
      Set/Update parameters of (incomplete) algorithm t individually;
    Compute $\hat{x}_{k+1}^t$ and test its acceptance ;
    if $\hat{x}_{k+1}^t$ *is acceptable* **then**
      $A \leftarrow A \cup \{t\}$;
  /* End Parallel Tasks */
  /* Begin Synchronization */
  if $A \neq \emptyset$ **then**
    $x_{k+1} = \arg\min_{t \in A} f(\hat{x}_{k+1}^t)$;
  /* End Synchronization */
  $k \leftarrow k + 1$;

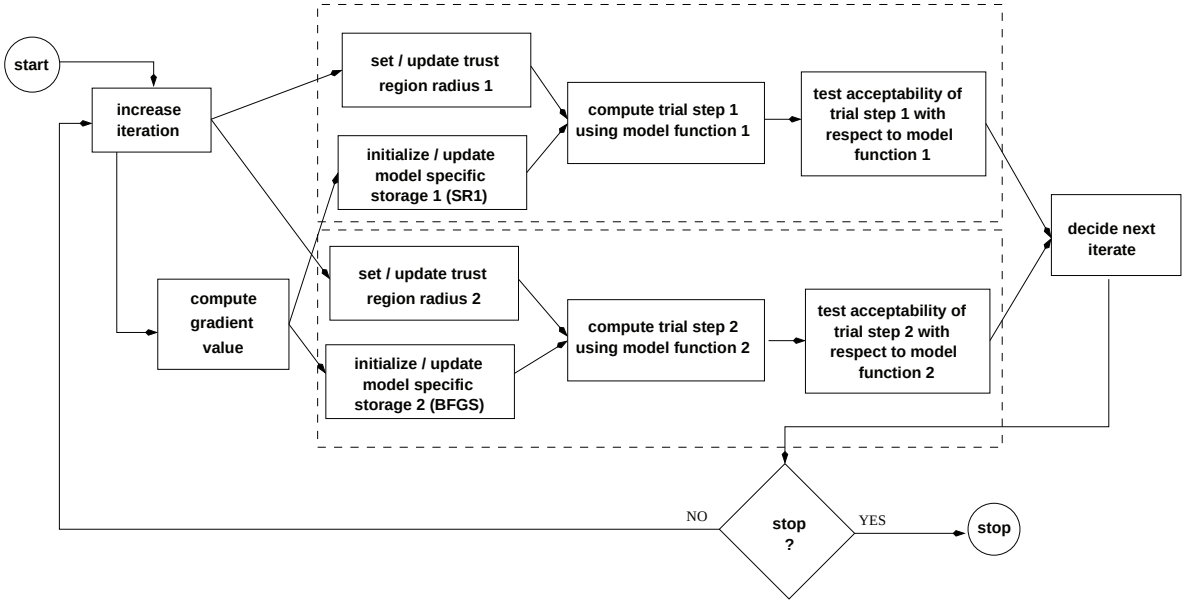


Figure 3.1: Flowchart of PTR2

PTR2 since both trial points are computed via solving trust-region subproblems. In the second example, a line search procedure is also incorporated, i.e. $p = 3$, and the resulting algorithm is called PTR2LS.

Implementation example with two trial points. PTR2 applies the concurrent search idea by solving two trust-region subproblems concurrently in the trial point computation phase. Each subproblem is set up with a different model function. Moreover, each trust-region radius value is set by interaction. That is, an exchange of information between two threads is used to determine the trust-region radii in the subsequent iteration.

In Figure 3.1, we illustrate the inherently parallel structure of PTR2. In this flowchart, each box stands for an algorithmic operation. In a task-based implementation of the algorithm, a task may cover one or more operations of this kind. In addition, an operation may be divided into multiple tasks, if Type-I parallelization is also in use. However, in the parallel software we implemented in this study, we designed tasks to achieve primarily Type-II parallelization. On the flowchart, independent blocks of operations are marked with dashed rectangles, which are suitable candidates to be defined as tasks of a Type-II parallelization.

The outline of our PTR2 implementation is given in Algorithm 2. We use two quadratic functions with different Hessian approximations, SR1 and BFGS, to setup the two model functions used in the respective subproblems. Based on the current (common) iterate of CCS, two trial points are computed and tested for acceptability by both tasks that are executed in parallel (lines 6 and 7 of Algorithm 2). Each task first sets the current iterate and the new trust-region radius value is determined according to the output of the previous iteration of CCS. Then the curvature approximation is updated when necessary. If both threads return acceptable points (line 8 of Algorithm 2), then the one providing the largest decrease in the objective function value is selected as the next iterate of CCS (line 9 of Algorithm 2). The details of the subprocedures `interactAndComputeTrialPointUseTR` and `evaluateTrialPoints` are given in Algorithms 5 and 3, respectively. Note that when there is at least one acceptable point at an iteration, PTR2 can take a nonzero step. Thus, when a particular TR implementation cannot find a successful iterate, the strategy may provide one for that particular thread. Likewise, when the strategy provides a successful iterate, it also provides information about a reasonable trust-region radius value in the new region. The threads work with different sizes of trust-regions to improve the exploration of the new region. When possible, one of the threads selects a larger radius value to enable taking larger steps from the current solution point.

Algorithm 2: An implementation of P2TR

1 **Input**: $x_0, k_{\max}, \epsilon, B_0^{\text{BFGS}} \Delta_0^{\text{BFGS}}, B_0^{\text{SR1}}, \Delta_0^{\text{SR1}}, \rho, \sigma_1, \sigma_2, \beta$
2 $P = \{$'SR1', 'BFGS'$\}$;
3 `initializeAlgorithms()`;
4 $A = \emptyset$; $k = 0$;
5 **while** *checkStopping()=FALSE* **do**
6 Throw task `interactAndComputeTrialPointUseTR('SR1')` ;
7 Throw task `interactAndComputeTrialPointUseTR('BFGS')` ;
8 `waitForAllTasks()` ;
9 `evaluateTrialPoints()` ;
10 $k \leftarrow k + 1$;

Figure 3.2 illustrates the behavior of P2TR on the highly nonconvex LOGHAIRY

Algorithm 3: `evaluateTrialPoints()`

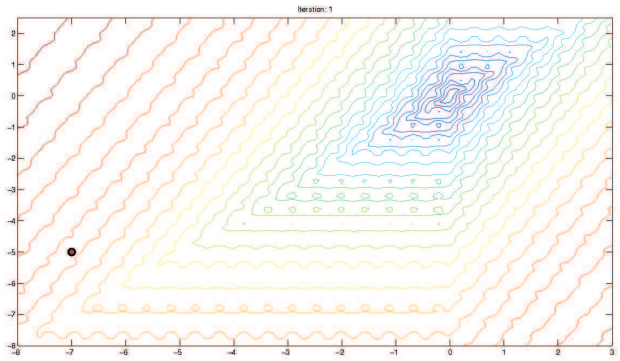
```
1 if A = ∅ then
2     x_{k+1} = x_k; bestAlgo = NULL;
3 else
4     x_{k+1} = arg min_{t∈P} f(x̄^t_{k+1}); bestAlgo = t ;
5     refreshGradient(); A = ∅ ;
```

problem of the CUTEr set [31]. In this illustration, the trial steps computed by using the BFGS-based model function are marked with (red) plus signs, the trial steps computed by using the SR1-based model are marked with (blue) cross signs, and the steps of PTR2 itself are marked with (black) circles. Figure 3.2(a), shows the starting point. The initial Hessian approximation is set to the identity matrix for both models in this preliminary implementation, and both threads start with the same trust-region radius value. Thus, the first trial points computed by both model functions are the same and that point is an acceptable one, so it is selected as the next iterate of PTR2 as we can see in Figure 3.2(b). The trust-region radii for both subproblems are not the same in the second iteration. As shown in Figure 3.2(b), the trial points computed by the algorithms could be quite far away from each other, since the model functions and trust-region radii are different. The point marked with a (red) plus sign is acceptable but the point marked with a (blue) cross sign is not. Therefore, as shown in Figure 3.2(c), the trial point marked with a (red) plus sign is selected even though it gave a higher objective function value. Next, both step computation operations are applied based on this new iterate. The thread marked with (blue) cross signs jumps to the successful point and updates its trust-region radius according to the radius of the thread marked with (red) plus signs (see Algorithm 5, lines 1-1). Therefore, new function information is provided to the blue (cross) thread and it is encouraged to take a larger step, if it is possible by starting its search on a larger trust-region than the red (plus) thread. Repeating this procedure, PTR2 reaches the solution in 53 parallel iterations. The complete path of suggested trial points and the steps of PTR2 are given in Figure 3.2(d). Figure 3.3 illustrates the behavior of both model functions when they are individually applied to solve the same problem. Comparing the complete paths of PTR2 and the individual algorithms,

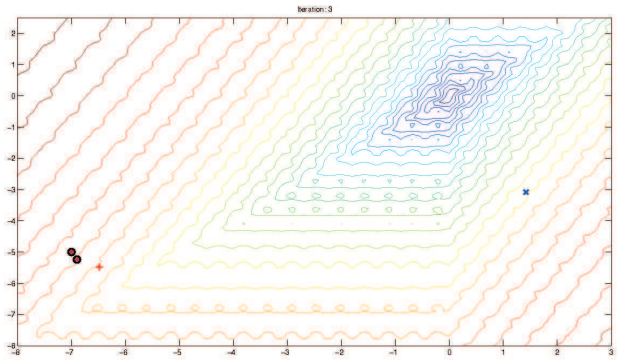
we observe that CCS may follow a completely different sequence of iterates. There is also another important observation: Note that the individual trust-region algorithm applied by the SR1-based model has required 249 iterations as shown in Figure 3.3(a). Likewise, Figure 3.3(b) shows that the individual algorithm applied by the BFGS-based model has required 113 iterations. Since PTR2 is able to solve the same problem in only 53 iterations, we observe that even if we could perfectly distribute the required iterations for either one of the individual algorithms on two parallel processors, the parallel number of iterations would be still higher than CCS.

To get a more comprehensive view about the potential of the proposed idea, we have tested PTR2 on a set of small-scale problems. The implementation details and the test results shall be presented in Section 3.1.3. These tests have shown us that we could get a decrease in the number of parallel iterations for most of the problems. Even better, we have observed that PTR2 could also decrease the number of gradient evaluations, which is equal to the total number of steps taken by the algorithm.

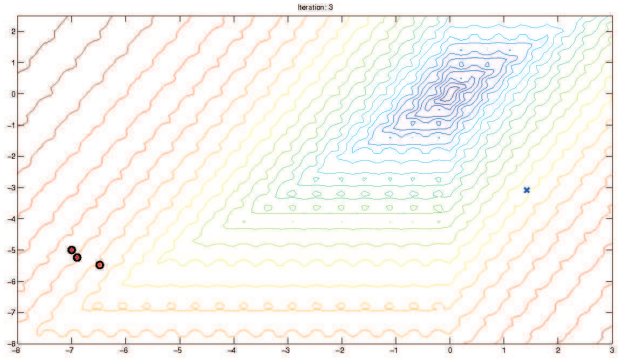
In the next example algorithm, PTR2LS, we expand PTR2 with another step computation procedure. To obtain a nonhomogenous CCS strategy, we include a line-search method in the new algorithm.



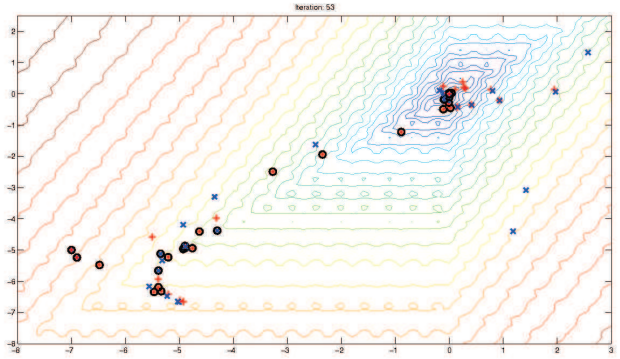
(a) Iteration 1: Initialization



(b) Iteration 3: The trial points

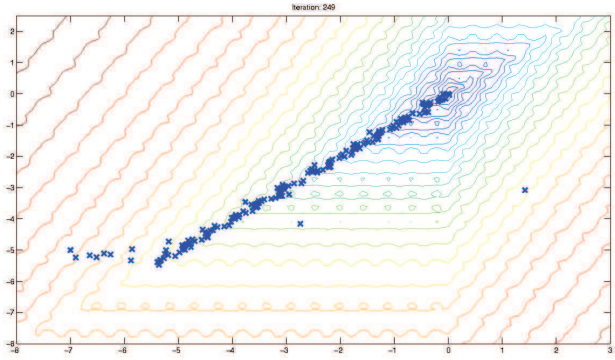


(c) Iteration 3: One iterate selected

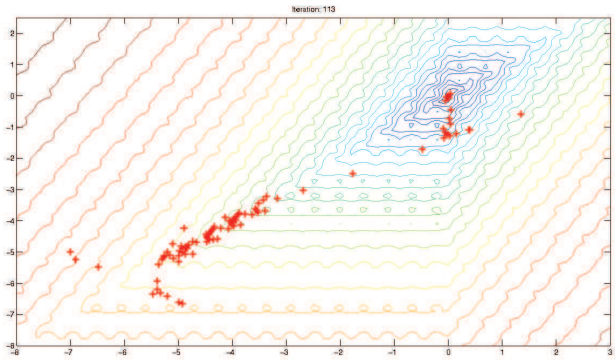


(d) Iteration 53: Complete path of PTR2

Figure 3.2: Steps of PTR2 on the LOGHAIRY problem starting from (-7,-5)



(a) Iteration 249: Complete path of individual TR with SR1



(b) Iteration 113: Complete path of individual TR with BFGS

Figure 3.3: Steps of individual TRSR1 and TRBFGS algorithms on the LOGHAIRY problem starting from (-7,-5)

Implementation example with three trial points. In PTR2LS, we also add to PTR2 a third trial step calculation task that apply a line-search algorithm. Thus, at each iteration of PTR2LS, three trial points are computed and tested in parallel as illustrated on a flowchart in Figure 3.4. Since line search is a special trust-region algorithm [14], PTR2LS can be seen as an extension of PTR2 which solves three different trust-region subproblems. However, as we explain below, we apply an incomplete line-search algorithm with the motivation of providing a better balanced workload distribution at each iteration.

Algorithm 4 is obtained by adding a third task (line 8) to Algorithm 2. The details of the new subprocedure `interactAndComputeTrialPointUseLS` is given in Algo-

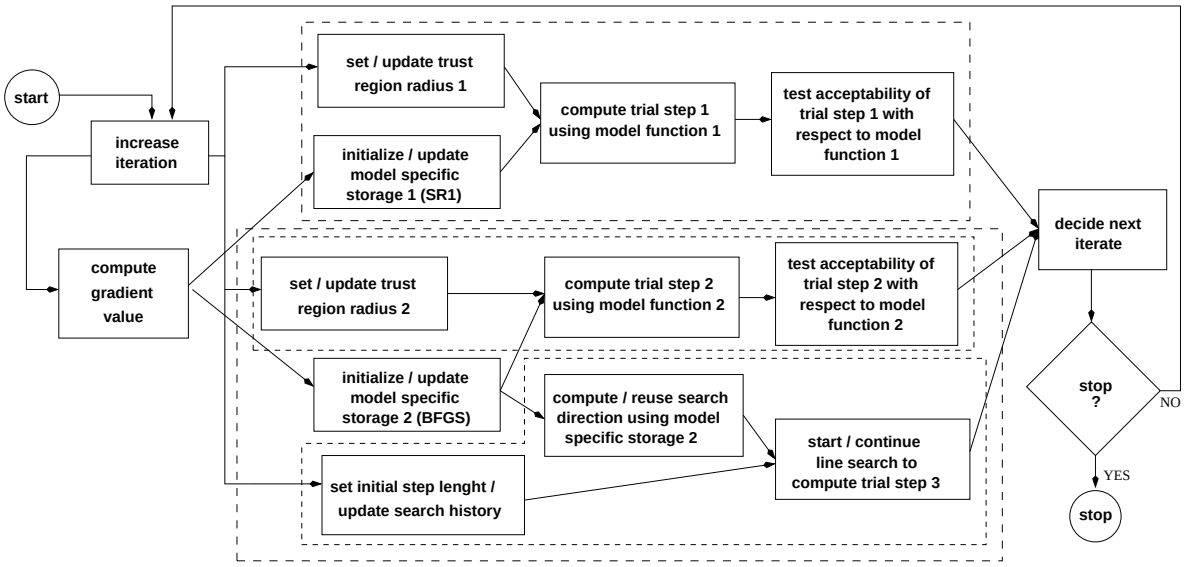


Figure 3.4: Flowchart of PTR2LS

rithm 6. The additional trial point is computed by an incomplete line-search procedure and shares the curvature information of the BFGS trust-region thread. The line-search thread sets its initial step size according to the output of the previous iteration, and this value may be affected by the trust-region radii of the two trust-region threads (see Algorithm 6, lines 1-1). Likewise, if the trial point computed by the line-search task is selected as the next iterate of CCS, its step size may affect the trust-region radius values of the trust-region threads (see Algorithm 5, lines 1-1). The interaction and exchange of information among the trust-region threads are quite similar to PTR2 .

To get some idea on the performance of PTR2LS, we again apply it first on a set of small-scale problems (see Section 3.1.3). Overall, the results revealed that the new algorithm has a better performance than PTR2 in terms of number of parallel iterations. However, we have observed that PTR2LS has taken smaller steps more frequently, which led to a slight increase in the number of gradient evaluations when compared against PTR2. We elaborate more on our findings in Section 3.1.3.

Details of step computation. The details of the step computation subprocedures used by PTR2 and PTR2LS are given in Algorithms 5 and 6.

Algorithm 5 explains the trust-region trial step calculation task. If there are

Algorithm 4: An implementation of PTR2LS

Input: $x_0, k_{\max}, \epsilon, B_0^{\text{BFGS}} \Delta_0^{\text{BFGS}}, B_0^{\text{SR1}}, \Delta_0^{\text{SR1}}, \rho, \sigma_1, \sigma_2, \beta, \alpha_0, \mu$
$P = \{$'SR1', 'BFGS$_1$', 'BFGS$_2$'$\}$;
`initializeAlgorithms()`;
$A = \emptyset$; $k = 0$;
while *checkStopping()=FALSE* **do**
    Throw task `interactAndComputeTrialPointUseTR('SR1')` ;
    Throw task `interactAndComputeTrialPointUseTR('BFGS`$_1$`')` ;
    Throw task `interactAndComputeTrialPointUseLS('BFGS`$_2$`')` ;
    `waitForAllTasks()`;
    `evaluateTrialPoints()`;
    $k \leftarrow k + 1$;

Algorithm 5: `interactAndComputeTrialPointUseTR`(myAlgo)

if *bestAlgo* $\neq$ *NULL* & *bestAlgo* $\neq$ *myAlgo* **then**
    if $\Delta_k^{bestAlgo} \neq$ *NULL* **then**
        $\Delta_k^{\text{myAlgo}} = \frac{2}{\beta} \Delta_k^{\text{bestAlgo}}$;
    else if $\alpha_0^{bestAlgo} \neq$ *NULL* **then**
        $\Delta_k^{\text{myAlgo}} = \max(\Delta_k^{\text{myAlgo}}, \alpha_0^{\text{bestAlgo}} \|d_{k-1}^{\text{bestAlgo}}\|)$;
if *bestAlgo* $\neq$ *NULL* **then**
    $B_k^{\text{myAlgo}} =$ `update(`B_{k-1}^{myAlgo}`)`;
$\bar{x}_{k+1}^{\text{myAlgo}} =$`solveSubproblem(`myAlgo`)` ;
$\rho = \frac{f(\bar{x}_{k+1}^{\text{myAlgo}}) - f(x_k)}{m_k^{\text{myAlgo}}(\bar{x}_{k+1}^{\text{myAlgo}}) - m_k^{\text{myAlgo}}(x_k)}$;
if $\rho \geq \sigma_1$ **then**
    $A \leftarrow A \cup \{\text{myAlgo}\}$;
    $f_{k+1}^{\text{myAlgo}} = f(\bar{x}_{k+1}^{\text{myAlgo}})$;
    if $\rho \geq \sigma_2$ **then**
        $\Delta_{k+1}^{\text{myAlgo}} = \frac{1}{\beta} \Delta_k^{\text{myAlgo}}$;
else
    $f_{k+1}^{\text{myAlgo}} = \infty$;
    $\Delta_{k+1}^{\text{myAlgo}} = \beta \Delta_k^{\text{myAlgo}}$

no acceptable points at the previous iteration, then the trust-region radius parameter is kept as it has been updated by shrinking in the previous iteration. If one of the trial points is accepted, then before starting the computation of the new trial points, the trust-region radii are updated by setting $\Delta_r = \frac{2}{\beta}\Delta_a$, where Δ_a and Δ_r are the radii corresponding to the accepted and rejected trial points, respectively, and β is a parameter in $(0, 1)$. If the step by a line-search procedure is accepted as the next iterate of CCS, and the current radius of any trust-region procedure, Δ_c, is smaller than norm of the successful initial line-search step of the last iteration, $\alpha_0\|d_{k-1}\|$, then we set $\Delta_c = \alpha_0\|d_{k-1}\|$ (lines 1-1). If a nonzero step is taken, the information used in approximating the curvature (B_k) is updated (lines 6-6). The trial point is then computed by solving the trust-region subproblem (line 8). The acceptance test for the trial point is carried out and the trust-region radius is updated by standard trust region acceptance decision and radius update procedures (lines 9-15).

Algorithm 6: `interactAndComputeTrialPointUseLS(`myAlgo`)`

1 **if** *bestAlgo* $\neq$ *NULL* & *bestAlgo* $\neq$ *myAlgo* **then**
2 $\quad \alpha_k \leftarrow \min(\alpha_k, \frac{\Delta_k^{\text{bestAlgo}}}{\|d_{k-1}\|})$;
3 **if** *bestAlgo* $\neq$ *NULL* **then**
4 $\quad d_k =$`computeDirection(`myAlgo`)` ;
5 $\quad$ `searchHistory = NULL; acceptanceFlag = FALSE;`
6 **else**
7 $\quad d_k = d_{k-1}$;
8 $[\alpha_k,$ `acceptanceFlag, searchHistory`$] =$`incompleteLineSearch(`myAlgo`)` ;
9 $\bar{x}_{k+1}^{\text{myAlgo}} = x_k + \alpha_k d_k$; $\alpha_{k+1} = \alpha_0$;

Algorithm 6 gives the details of the line-search computation task. If a step computed by a trust-region algorithm has been accepted in the previous iteration and the norm of the initial step in the most recent iteration, $\alpha_0\|d_{k-1}\|$, is larger than the radius of the accepted trust-region step, Δ_a, then the initial step-length of the line-search algorithm is set as $\alpha_k = \Delta_a/\|d_{k-1}\|$ (lines 1-1). The quasi-Newton search direction d_k is then updated, if a nonzero step has been taken in the previous iteration (line 4). Then, an incomplete line-search is implemented (line 8). That is, we give an upper

bound for the number of inner iterations of the line-search algorithm. If the line-search procedure is not completed before the given limit, then it is hibernated and labeled as unacceptable but its computations are stored. If none of the remaining trial points were acceptable in the previous iteration, then the line-search procedure continues its calculations from the point where it has been hibernated using the stored data. Otherwise, it restarts its calculations from the new iterate with an empty history.

Recall the first illustrative example, where the path of the concurrent search strategy was completely different than the paths of the individual algorithms (compare Figure 3.2(d) against Figure 3.3). This raises a natural question: Does switching among the threads prevent CCS from converging?

3.1.2 Convergence

Let us consider the CCS strategy applied with p trial points. As we mentioned in the introduction part, each trial point is computed by the step computation procedure of a convergent iterative algorithm. To make this statement more concrete, we require the each algorithm $t \in \{1, 2, \ldots, p\}$ satisfy the following property:

(A1) At a nonstationary point x_k, algorithm t computes a step satisfying

$$f(x_k) - f(\hat{x}_{k+r}^t) \geq \xi \|\nabla f(x_k)\|^q \tag{3.2}$$

for a finite value of r, $r \in \{1, 2, \ldots\}$, and $\xi > 0$.

In correspondence to the notation in Algorithm 1, $\hat{x}_{k+1}^t$ denotes here the trial point computed by algorithm t at iteration k. The condition (3.2) also gives a generic *acceptence* rule for the framework given in Algorithm 1; that is, if $\hat{x}_{k+1}^t$ satisfies (3.2), we set $A \leftarrow A \cup \{t\}$. Note that the methods used in the implementation examples given in Section 3.1.1 satisfy the condition (A1). In particular:

- Let m_k be the model function of f, Δ_k be the trust region radius and B_k the Hessian of f or an approximation to it (see [53] for the notation commonly used for trust region algorithms). If we then denote the steps produced by trust-region

algorithms at a nonstationary point x_k by $s_k(\Delta_k)$, then we know these steps satisfy the Cauchy decrease condition

$$m_k(0) - m_k(s_k(\Delta_k)) \geq c_1 \|\nabla f(x_k)\| \min\left(\Delta_k, \frac{\|\nabla f(x_k)\|}{\|B_k\|}\right),$$

where $\|B_k\| < \infty$, $c_1 \in (0, 1]$, and m_k is a model of f at x_k so that

$$\begin{aligned} f(x_k) - f(\hat{x}_k^t) &\geq c_2(m_k(0) - m_k(\hat{x}_k^t - x_k)) \\ &\geq c_2 c_1 \min(\Delta_k \|\nabla f(x_k)\|, \|B_k\|^{-1} \|\nabla f(x_k)\|^2). \end{aligned}$$

is provided after finite number of trial step computations.

- Well-known line search methods based on Armijo or Wolfe conditions provide steps $s_k(\alpha)$ that follow the gradient related directions so that

$$-\nabla f(x_k)^T s_k(\alpha) \geq c_3 \alpha \|\nabla f(x_k)\|^{r_1} \quad \text{and} \quad \|s_k(\alpha)\| \leq c_4 \alpha \|\nabla f(x_k)\|^{r_2}$$

hold for $r_1, r_2 \geq 1$, $c_3, c_4 > 0$. That guarantees to find $\alpha > 0$ satisfying

$$\begin{aligned} f(x_k) - f(\hat{x}_k^t) &\geq -c_5 \nabla f(x_k)^T (\hat{x}_k^t - x_k) \\ &\geq c_5 c_3 \alpha \|\nabla f(x_k)\|^{r_1}. \end{aligned}$$

That is, α stays bounded away from zero [7].

Note that at any iterate k, the interactive parameter selection scheme does not cause the above convergence properties fail for all trial steps computed by all algorithms because (i) there is no interaction, if there is no acceptable trial point at an iteration; (ii) a trial step cannot be acceptable, if it does not satisfy the above conditions; (iii) the parameters of the algorithm that has determined the current iterate are not affected by others.

Finally, let us give one more assumption about the problem before we give the convergence result.

(A2) The level set $L_0 = \{x : f(x) \leq f(x_0)\}$ is compact, and within L_0, f is differentiable and the gradient function ∇f is continuous.

THEOREM 3.1.1 *Let $\{x_k\}$ be a sequence of iterates generated by Algorithm 1 with infinitely many elements. Suppose assumptions (A1) and (A2) hold. Then, all limit points of $\{x_k\}$ are stationary points of f.*

PROOF. Since the CCS framework requires $f(x_{k+1}) \leq f(x_k)$, the sequence $\{x_k\}$ stays in the level set of f at x_0 – which is a compact set by (A2).

Recall that assumption (A1) enforces that a step satisfying (3.2) is computed (by any algorithm) at a nonstationary iterate x_k within at most r iterations. So, the set A defined in Algorithm 1 will be nonempty in at most r iterations and the algorithm will accept a new iterate. Therefore $\{x_k\}$ cannot have a constant subsequence.

Consider any convergent subsequence of $\{x_k\}$, and let S denote the index set of the elements in this subsequence. Let $k', k'' \in S$ such that $k'' > k'$ and $x_{k''} \neq x_{k'}$. Recall that Algorithm 1 assigns $x_{l+1} = x_l$ if $A = \emptyset$ at iteration l. So, $x_{k'} = x_k$ should hold for $k \leq k'$ such that either we have $A \neq \emptyset$ at iteration k, or $x_k = x_0$. Then, using line 13 of Algorithm 1 we have

$$\begin{aligned} f(x_{k'}) - f(x_{k''}) &= f(x_k) - f(x_{k''}) \\ &\geq f(x_k) - f(x_{k+1}) \geq f(x_k) - f(\hat{x}_{k+1}^t), \quad \text{for all } t \in A, \end{aligned}$$

which implies by assumption (A1) that

$$f(x_{k'}) - f(x_{k''}) \geq \xi \|\nabla f(x_{k'})\|^q, \quad \text{for } q \geq 1. \tag{3.3}$$

On the other hand, the continuity of f implies $\lim_{k \in S} f(x_k) \to f_*$. So,

$$f(x_{k'}) - f(x_{k''}) \to 0 \quad \text{as } k', k'' \to \infty,\ k', k'' \in S.$$

This gives $\|\nabla f(x_k)\| \to 0,\ k \in S$ by (3.3). □

Note that it would also be possible to discuss that the CCS framework guarantees to converge to a minimizer of f rather than only a stationary point. This would require at least one of the algorithms in CCS to have this property as well as a simple alteration of the termination criteria.

The discussion in this subsection also clarifies the motivation behind our CCS design. The application of the acceptance tests and the trial point selection rule guarantee convergence of the overall algorithm. Note that it is also possible to apply a nonmonotone version of CCS, and another option would be to apply synchronization after a fixed number of iterations instead of every iteration (line 14, Algorithm 1). In both cases, the convergence of the resulting algorithms could be shown following the similar arguments above.

3.1.3 Practical Performance

In this section, we present the results we have obtained with both illustrative CCS algorithms, PTR2 and PTR2LS. Throughout this section, we mainly investigate the performance of CCS against the individual algorithms included in CCS. We also introduce the large-scale versions of the two illustrative algorithms. We denote the large-scale versions of PTR2 and PTR2LS by L-PTR2 and L-PTR2LS, respectively.

We do benchmarks based on three criteria:

1. Number of parallel iterations until convergence, to see whether the new algorithms could really achieve a reduction in the number of parallel trial step calculations.

2. Number of gradient evaluations, which reflects the number of successful steps to the solution as well as the computational cost of gradient calculation.

3. Wall clock time, to evaluate if any reduction in number of iterations is large enough for providing a reduction in the total solution time.

The parallel algorithms are coded in C++ and compiled with Intel C++ compiler by using the `-fast` option. To code the parallel tasks, we have used Intel's Threading Building Blocks (TBB) library [68], and for the linear algebra operations, we have

used the Intel Math Kernel Library (MKL) [46]. The experiments are conducted on a computer with Quad-Core Intel(R) Xeon(R) CPU running at @ 2.66GHz and operating under GNU/Linux system.

The algorithms are tested on the unconstrained problems of the CUTEr collection [31]. The dimensions of these problems range from 2 to 10,000. The algorithms are first applied on small-scale problems ($n \leq 500$). After observing encouraging results on this set, the tests are repeated for larger problems ($1000 \leq n \leq 10,000$) using the large-scale adaptations of algorithms. Four standard termination conditions are used, setting $\epsilon = 10^{-5}$,$k_{max} = 10,000$, $PREC = 10^{-8}$ and $D\_MAX = 10^{16}$:

1. The gradient norm is sufficiently small, $\|g_k\| < \epsilon$.
2. The algorithm stalls, $\|x_{k+1} - x_k\| < 1.1PREC$.
3. The problem is probably unbounded, $\|x_{k+1} - x_k\| > 0.9D\_MAX$.
4. The maximum number of iterations are exceeded, $k > k_{max}$.

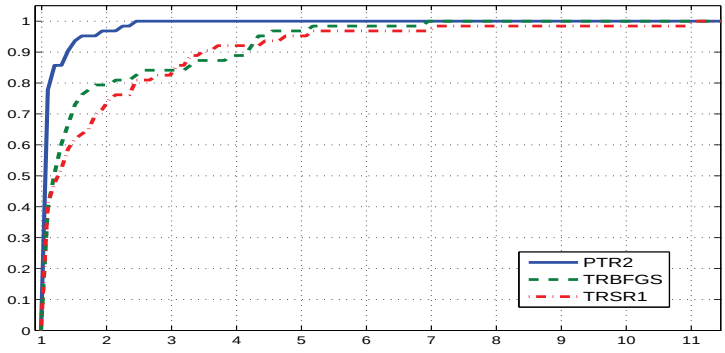
In our implementation, both trust-region models of PTR2 use quadratic models based on BFGS and SR1 quasi-Newton updates. For medium-to-large scale problems, we have used limited memory versions of these updates and set the number of corrections to 5 [52]. For the two update procedures, separate memories are reserved because a pair suitable for applying the SR1 update might not be suitable for applying the BFGS update. We have used compact form formulations of both limited memory update procedures. The line-search procedure of PTR2LS also implements the BFGS approximation of the curvature. For solving the trust-region subproblems, we have applied a dogleg strategy for small-scale problems, and for large-scale problems, we have used the Conjugate Gradient (CG) algorithm following the strategy proposed by Steihaug [64]. In our line-search procedure we have applied quadratic interpolation to determine the step-length. If an acceptable step length cannot be found in 5 line-search steps, then the incomplete line-search procedure is paused. In our numerical trials, we have observed that this number is a good choice to balance the time required for one trust-region iteration.

For scaling purposes, we have selected the initial Hessian approximation at iteration k as $B_k^0 = \delta_B I$ with $\delta_B = \dfrac{y_{k-1}^T y_{k-1}}{s_{k-1}^T y_{k-1}}$ as suggested in [53]. For the line-search algorithms, on the other hand, this scaling factor for the initial inverse Hessian approximation at iteration k is selected as $\delta_H = 1/\delta_B$. We did not apply any preconditioning in the CG procedure.

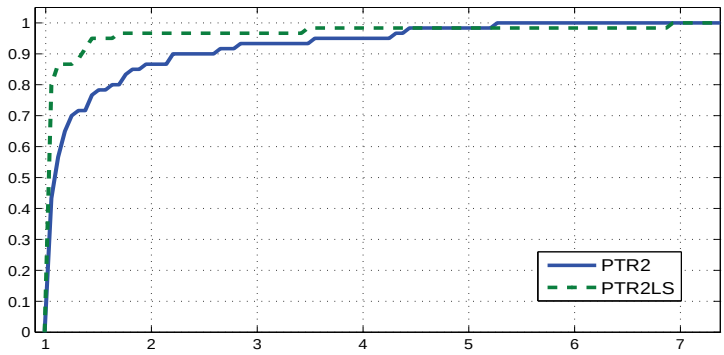
The remaining parameters are as follows: initial trust-region radius, $\Delta_0 = \|x_0\|$; initial step-size, $\alpha_0 = 1$; trust-region acceptability test and trust-region update parameters, $\rho = 0.1$, $\sigma_1 = 0.25$, $\sigma_2 = 0.75$; trust-region rescaling parameter, $\beta = 0.5$. Initial points are selected as the default ones provided by the CUTEr collection.

Small-scale problems. This first test set includes a total of 85 problems. There are no instances that either one of the individual algorithms, TR with BFGS (TRBFGS) or TR with SR1 (TRSR1), has successfully converged but PTR2 has failed. Except two cases, PTR2 has been able to find the solution found by the successful individual algorithm. Nonetheless, there are another two cases where both individual algorithms fail but PTR2 converges. For a clear comparison, the benchmark includes only those instances for which all three algorithms have converged to the same solution point. Therefore, 67 cases are included in the first benchmark. Figure 3.5(a) and Figure 3.6(a) give the performance profiles for the parallel number of iterations and the number of gradient evaluations, respectively (see Appendix A for a review on performance profiles). These profiles show that PTR2 indeed achieves a significant reduction in the number of parallel iterations. We do not report the wall clock time figures, since almost all problems are solved within less than a second.

In a similar way, we have also tested the performance of PTR2LS on these small-scale problems. The inclusion of line-search thread has caused a few instances to terminate with termination condition (ii). That is, the concurrent search stopped before convergence, since the steps have become very small. However, as Figure 3.5(b) illustrates, on average PTR2LS has provided a nice performance improvement over PTR2. For the number of gradient evaluations, however, Figure 3.6(b) shows that PTR2LS falls slightly behind PTR2 but still outperforms the individual algorithms.

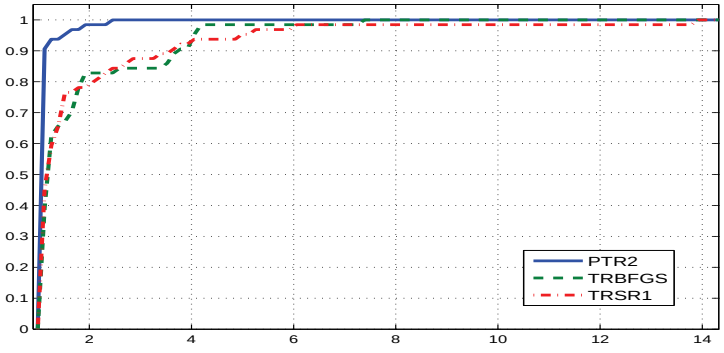


(a) PTR2

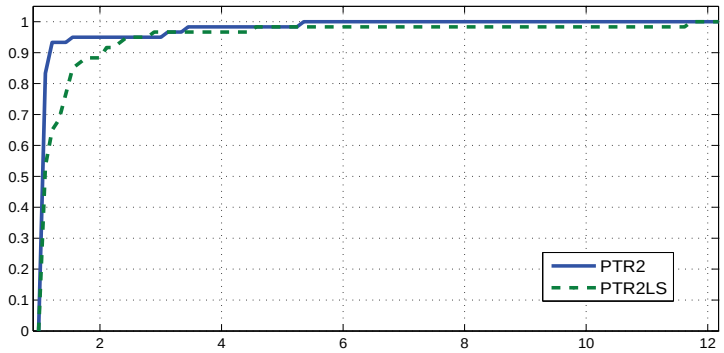


(b) PTR2LS

Figure 3.5: Performance profiles on the number of iterations on small-scale problems



(a) PTR2



(b) PTR2LS

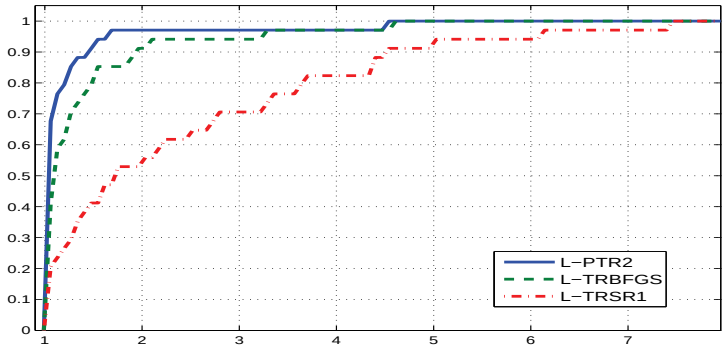
Figure 3.6: Performance profiles on the number of gradient evaluations on small-scale problems

Medium-to-large-scale problems. The second test set includes 64 problems with dimensions ranging from 1,000 to 10,000. The benchmark results for L-PTR2 are obtained with only 32 instances, for which all three algorithms, L-TRBFGS, L-TRSR1 and L-PTR2 have converged to the same solution point. When we examine these results, we observe that there are more instances than the small scale results (9 instances) such that all three algorithms have converged to different solution points. For the remaining 23 instances, at least one of the algorithms has failed to find a solution within the given tolerances. There are no instances that either one of the individual algorithms converges successfully but L-PTR2 can not.

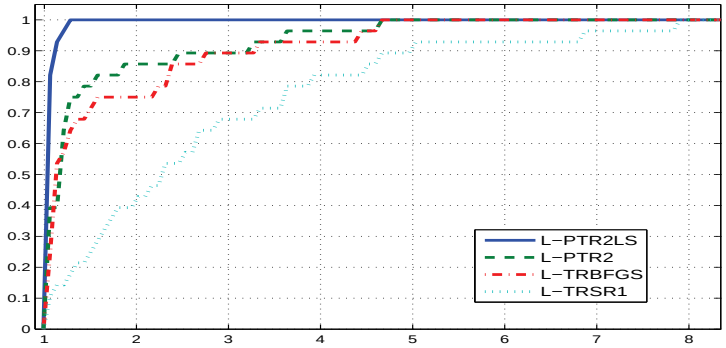
When it comes to L-PTR2LS, there are 29 instances where all three algorithms have converged to the same solution point. We have not included the line-search as a separate individual algorithm in our benchmark for only an incomplete line-search is applied within L-PTR2LS. As compared to L-PTR2, there are more cases for which all three algorithms have converged to different solution points (9 instances versus 14 instances). This could mean that the line-search procedure has diverted the algorithm to different solution points. Again, there are no instances that either one of the individual algorithms converges successfully but L-PTR2LS can not.

We first compare L-PTR2 and L-PTR2LS against the individual algorithms to see whether concurrent search could achieve a reduction in the number of iterations. The benchmark results are given in Figure 3.7. The success of L-PTR2LS is much more significant. As the figure shows L-PTR2LS reduces the number of iterations more than 2 times for around 20% of the cases. The second benchmark is about the number of gradient evaluations. Figure 3.8 demonstrates the success of L-PTR2 in reducing the number of gradient evaluations. On the other hand, L-PTR2LS performs slightly worse than L-PTR2 in terms of gradient evaluations. Recall that this difference between the number of iterations (total number of steps) and the number of gradient evaluations (number of successful steps) has been also observed with the small-scale problems.

Finally, we have compared the solution times for all algorithms to see whether the reduction in the number of parallel function evaluations and subproblem solutions has been really reflected to the wall clock time. To have a fair comparison with L-



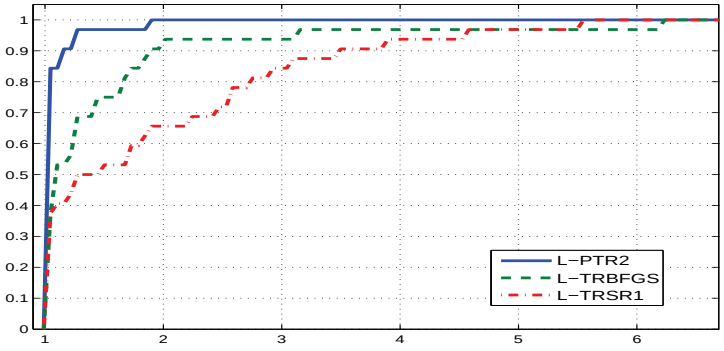
(a) L-PTR2



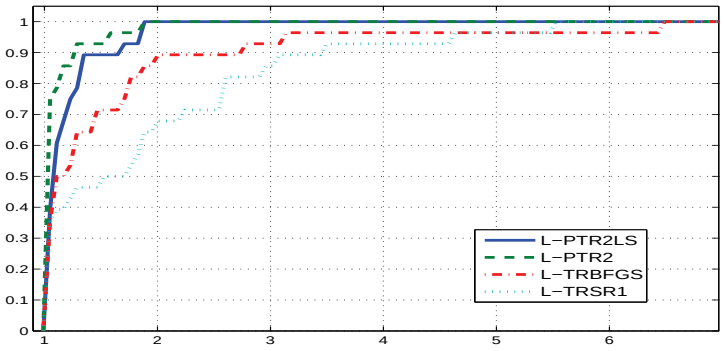
(b) L-PTR2LS

Figure 3.7: Performance profiles on the number of iterations on medium-to-large size problems

PTR2, we have configured the individual algorithms so that they use two threads in all their linear algebra operations [46]. Similarly, we compare the results of three-threaded L-PTR2LS after we have configured the individual algorithms to use three threads in their linear algebra operations. As shown in Figure 3.9, both L-PTR2 and L-PTR2LS have provided some improvement in the wall clock time figures. However, the increase in their performances is not as large as it has been for other performance criteria.

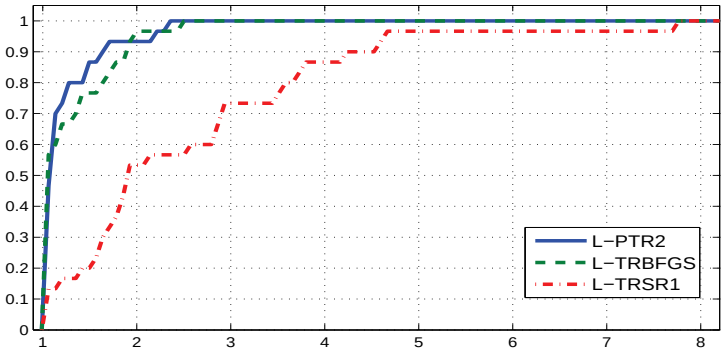


(a) L-PTR2

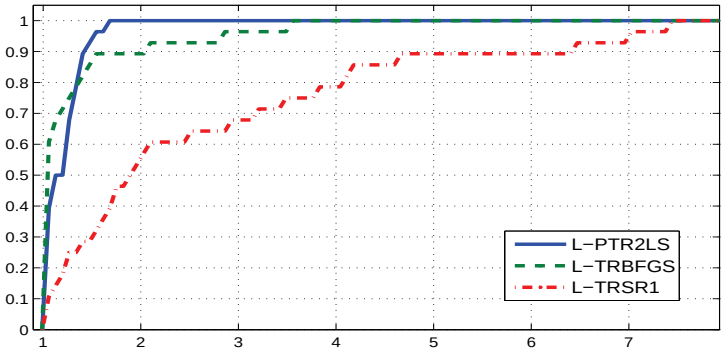


(b) L-PTR2LS

Figure 3.8: Performance profiles on the number of gradient evaluations on medium-to-large-scale problems



(a) L-PTR2



(b) L-PTR2LS

Figure 3.9: Performance profiles on the wall clock time on medium-to-large-scale problems

3.2 Extensions to Global Optimization

The CCS framework proposed in the previous section mainly concentrates on the part of the solution process until the algorithms arrive at a region in the close proximity of a local solution point. It may have little to contribute to the local convergence of the included methods in a region where full steps are always acceptable and trust-region constraints are always inactive. Therefore, when the process is quite close to the optimal solution, it may not be able to provide a better convergence rate than any of the included methods. However, we may expect significant performance improvement when the framework is modified with the purpose of recovering multiple local optimal solutions. This is an important objective, when one concentrates on global optimization.

In this section, we first consider an extension of CCS that can provide multiple different local local solutions of (3.1) when f is multimodal. Then, we discuss how it can be further extended to solving global optimization problems. In particular, we consider the case where the objective function f of problem (3.1) is not convex, and the aim is to find its global minimum in a region defined by a hypercube.

3.2.1 Concurrent Search for Multiple Solutions

The question in our mind is as follows: Would it be a good idea to compute within CCS framework as many candidate trial points as the number of available parallel units? Apparently, such an attempt may not necessarily *scale* well in converging to a single solution point. When the problem size is large, using more threads in Type-I parallelization would be more beneficial in improving the speed of overall solution process. However, when the objective becomes finding multiple different solutions rather than fast convergence to a single solution, it is then quite reasonable to increase the number of local search algorithms that run in parallel.

One such option is to *fork* the search over the solution space. That is, we modify the CCS framework as follows: When there are more than one acceptable points, we follow all or a number of them instead of selecting the one with the least objective function value. This modification is illustrated in Figure 3.10. Here, the symbols,

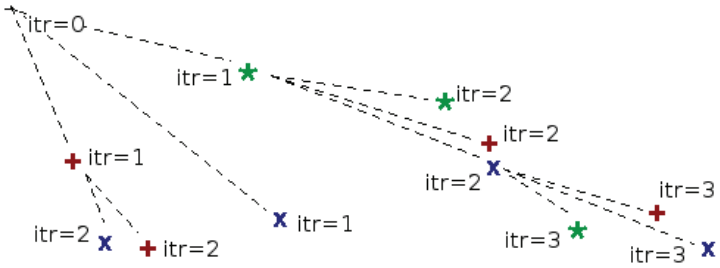


Figure 3.10: Forked search for $p = 3$

'$\times$','$*$', and '$+$' show the trial points computed by the three algorithms of CCS. In the first iteration, two of them was acceptable, so a seperate search is started from both points. So, we have 6 parallel trial point calculations in the second step. When the iterates in two branches become close, one of them are cut. Since the number of parallel resources is limited, it may become necessary to fathom some of the branches from further forking. The algorithm is not stopped when the first local solution is found. The search process seizes until a given time limit is exceeded or when the required number of local solutions is obtained.

In fact, the transition from basic NLP algorithms to CCS, and then to the forked CCS is a very nice example of the transition from *local search* to *global search*. In the next section, we will consider another transition step to a framework for global optimization. In this case, the tasks will consist of complete algorithms, and the task interaction will be asynchronous and flexible.

3.2.2 A Multiagent Framework

The aim in global optimization is finding the globally best solution of (3.1), usually within a bounded set $\mathcal{F} \subset \mathbb{R}^n$. Thus, global optimization methods have different concerns than the local optimization methods (important reviews of global optimization, its popular solution methodologies and applications can be found in [51, 62, 56]). In this section, we shall provide an example that adopts our parallel algorithm design approach to deal with the specific difficulties arising in global optimization problems.

Basic ideas. Solving a general global optimization problem, which does not have a special structure, requires an extensive search on the set of feasible solutions, $\mathcal{F}$. Thus,

it is very suitable for the application of various decomposition schemes, which can be described by a group of tasks and task interrelationships.

An approach that fits into this scheme is to run several instances of an algorithm in parallel starting from different points, which share information in a prescribed way. It has been argued that the resulting parallel algorithms may perform better in terms of solution quality thanks to the cooperation among individual search algorithms [30, 15]. The next idea would be to run different search algorithms in parallel and let them cooperate, which is referred as parallel hybrid algorithms. In this case, the tasks are no more identical, but the interaction is again predefined. There are existing examples of parallel hybrid algorithms that are reported to perform quite well [66, 47, 39]. While parallel hybrid algorithms cover various combinations of algorithms, a more adaptive and efficient scheme would be possible by allowing the interaction among the algorithms emerge during the solution process.

Let us consider the following example. Suppose that different local search algorithms run in parallel starting from different initial points, and at some point of their execution, each algorithm receives an external information like an incumbent solution obtained by another algorithm. Possibly, the new information is processed in a different way by each algorithm; i.e., one uses the information in directing its search, whereas another one disregards it for various reasons. Moreover, since the received information can be used in the middle of an ongoing execution, it will cause modifications in the original tasks of the algorithm. This suggests a system that will run various algorithms or procedures concurrently and enable them to cooperate in the way they determine. It may be the case that during the beginning of its execution, an algorithm may want to ask others in the system about their experiences (e.g., their current best point, potential areas worth exploring, and so on) but may decrease its communication with others as it explores the environment better. Thus, it is not feasible for a system designer to hard code expected communications; the need for such communications will only become apparent during run time. This defines an asynchronous algorithmic structure where a team of algorithms run concurrently and cooperate flexibly as they see fit.

The above requirements make a platform consisting of software agents an ideal

tool for implementing the proposed scheme. With this in mind, we have designed a new software environment called MultiAgent eNvironment for Global Optimization (MANGO).

Environment. MANGO is a Java-based multiagent environment that provides the necessary utilities to develop agents that can participate in a multiagent system to solve global optimization problems.

Each agent is implemented as a Java program that is developed using MANGO API. Agents may run on the same computer or may reside on different computers, which are distributed over the network. Every task, such as running an optimization algorithm, visualization of optimization results and administrative issues, is performed by agents. In general, each agent performs a specific task. However, there is no limitation on the number of tasks that an agent can perform in parallel.

MANGO enables agents to find each other through a directory system. It contains an extensible protocol for agents to communicate with each other. The protocol messages are related to solving problems, such as exchanging current best points, signaling areas already explored by others, and so on. Hence, agents can find others and cooperate with them on their own.

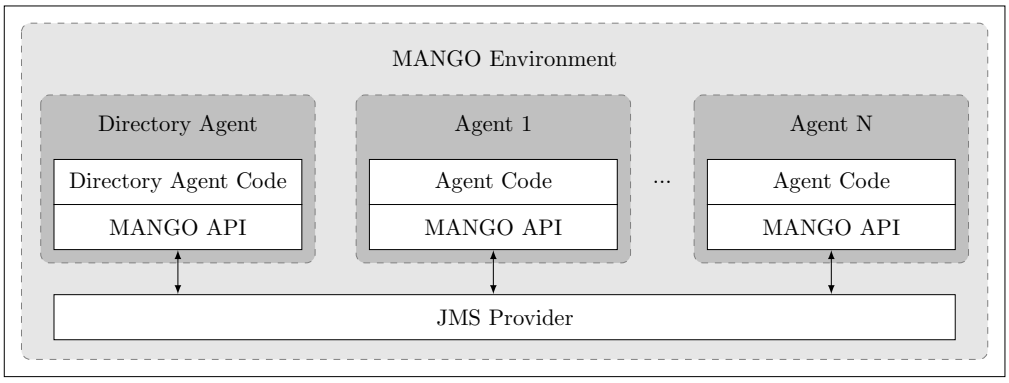


Figure 3.11: The MANGO environment

When a MANGO agent is being designed, there are three decision points that need to be considered:

1. *Optimization Algorithm:* The first point is the agent's main algorithm for at-

tacking the global optimization problem. This algorithm may be any known or newly-developed algorithm for solving a global optimization problem. The agent designer decides on this algorithm and implements it in the agent. The MANGO environment comes with a set of algorithms that can be used when developing new agents.

2. *Outgoing Messages:* The second point is related to when and with whom the agent is going to communicate during its execution. The communication is necessary for various reasons, but most importantly for coordination. That is, it is beneficial for an agent to position itself correctly in the environment. For example, two agents may not want to be searching the same area since probably if they search two different areas they may find a solution faster. Conversely, they may want to focus on a certain area rather than diverging.

   The questions of when and with whom to communicate are strictly related to the optimization algorithm that the agent is using. If the agent's own algorithm cannot handle certain tasks, the agent would need others' services to handle these. For example, if the agent's optimization algorithm cannot perform local search well, the agent may find it useful to find other agents that can offer local search service. As explained before, whether an agent does offer this service can be found out by querying the directory agent that keeps track of the services associated with each agent. Alternatively, an agent that can do local search well may be interested in finding out new areas to search when it finishes its local search. Hence, the agent may be interested in finding other agents that can suggest new areas to search.

   An agent may decide to take a leader role in the multiagent system and influence the others by suggesting areas to explore or refrain from. The choice of taking this role is up to the agent, but is also affected by the particular algorithm the agent is executing. That is, some algorithms can identify possibly promising areas quickly, and thus, it is reasonable for the agent to take this role and to inform others about the potential of these areas. Conversely, an agent may be designed to

play a leader or a follower role during design time. For example, a relatively rigid setting consisting of a leader agent executing an *interval search*-type algorithm, and a group of agents executing local search algorithms is also possible. In this example setting, the leader agent is intended to manage the overall search whereas the rest of the agents have only local tasks. The algorithm of the leader is able to partition the solution space in a specific way and process the information collected from different parts. That is, it assigns lower bounds to the objective function value attainable in each partition, eliminates some partitions based on these bounds, and does a new partitioning for the remaining area. The local search agents apply their algorithms as *requested* by the leader: within the region it tells, using the parameters it decides, and stop according to the termination criteria it asks.

3. *Incoming Messages:* The third decision point is related to if and how the agent is going to handle incoming messages. Note that since the communication is asynchronous, the agent will receive incoming messages during the execution of its algorithm. Incoming messages can be handled by the agent in two different ways. The first way is, interrupting the agent immediately when a message is received. In this case, the agent stops execution of its algorithm to handle the incoming message. The second way is, storing incoming messages without interrupting the agent. In this case, the agent checks for incoming messages whenever it is appropriate.

   One naive approach is to always answer or follow the incoming messages. For example, if an agent receives an explore message, it can always jump to the areas that is being suggested for exploration. Or, whenever it is prompted for the best solution it has found, it can return its current best solution. However, the following play an important role in how the incoming messages can be handled intelligently.

   The exploration state, that is how well the agent has explored the environment, is important in answering questions, since an agent may prefer not to answer

questions if it has not explored the environment well, or conversely, prefer not to follow orders (such as refrain messages) if it has explored the environment carefully. For example, in the beginning of the execution, when the agent did not have enough time to search properly, it may decide not to answer incoming messages related to the best solutions it has found, since its solution may not be representative.

Over the course of execution, an agent may model other agents based on the types of messages they are sending. Based on this model, an agent may decide how and if it is going to handle a message. For example, after certain iterations an agent may decide to ignore messages from a particular agent, if these messages have become too restrictive for the receiving agent to explore the region.

The structure and tools of MANGO allow the execution of a wide range of global optimization algorithms described as a set of interacting operations. In one extreme, it welcomes individual noncooperating agents, which is basically the traditional way of solving a global optimization problem. In the other extreme, autonomous agents existing in the environment cooperate as they see fit during run time.

3.2.3 Implementation Examples

The implementation examples in this section illustrate the main proposition of this thesis from a broader perspective. Here, as an extreme form of interrelated parallel tasks, we have cooperating agents. We next give examples of three optimization agents and three interaction scenarios that are implemented in the MANGO environment.

Agent realizations. In our implementation examples, we develop and run three sample agent types: Agent B, Agent T and Agent R. For each agent below, we describe the three important points: its algorithm, treatment of outgoing messages and incoming messages.

Agent B applies a BFGS quasi-Newton algorithm with line-search. The algorithm terminates either by recovering a local optimum or exceeding the maximum number of iterations. In fact, Agent B applies a modification to the BFGS quasi-Newton algorithm

as a result of its interaction with other agents, which shall be explained below. Also, since (3.1) has bound constraints in our case, it projects the steps computed by the original algorithm to the feasible region whenever the next iterate falls outside. We should also note that the agent repeatedly executes its local algorithm, i.e., when it converges to a local solution or exceeds the maximum number of iterations, it starts a new algorithm instance from a different initial point.

Within a system consisting of these three agents, Agent B has two types of outgoing messages. When the agent converges to a point x_f, starting from a point x_0, it sends the ball with center x_f and radius $\|x_f - x_0\|$ to all others, i.e., agents of type Agent T and Agent R, as a REFRAIN_REGION message. The intended meaning of this message is that the ball has been already explored by Agent B; so, others are better off searching elsewhere to increase their chances of finding the global minima. Note that the receiving agents are free to honor or ignore the message. The second type of message that this agent can send is INFORM_SOLUTION message. By design, it only sends this message to Agent T to notify its current best solution. This is to depict that an agent's outgoing message behavior may differ. In different settings, Agent B can send one of these two types, or both.

The only type of incoming message that Agent B is interested in is INFORM_SOLUTION message. When the agent receives this message, it assumes that the received point x_r is a local minimizer. So, in order not to converge one more time to one of the already discovered local solutions, it adds x_r to a list of *known local minimizers*, and tries to stay away from those points using a penalty function as its objective. The penalty function ϕ is obtained by adding a penalty term to the original objective function f so that approaching to the known local minimizers increases the value of ϕ. Formally, we have

$$\phi(x, P) := f(x) + \theta \sum_{y \in P} \frac{1}{\|x - y\|^2 + \epsilon}, \tag{3.4}$$

where P is the set of known local minimizers, ϵ is a small positive number, and $\theta$is a constant multiplier. In this way, the minimization algorithm applied by Agent B is expected to direct it towards different local solutions, providing a more extensive search

in the overall solution space. If Agent B receives any other type of message, it simply ignores it.

Agent T applies a trust-region algorithm, another local search method that also guarantees convergence to a local solution of (3.1) under some mild assumptions. Like Agent B, we also impose an upper bound on the maximum number of iterations that could be spent by the algorithm. Agent T does not modify the step computation procedure of the original algorithm, but it may stop a run before converging to a solution by using the information it receives from other agents. If there is no interaction, it restarts its algorithm from a random initial point after the termination of each single run, as in the case of Agent B. It also handles the bound constraints of (3.1) the same way as Agent B does.

In its interaction with Agents B and R, the only type of message Agent T sends is INFORM_SOLUTION message. It sends messages only to Agent B to share its current best solution when it converges to a local minimizer x_f or exceeds a predetermined maximum number of iterations.

Agent T processes two types of incoming messages: REFRAIN_REGION and INFORM_SOLUTION. It assumes that the REFRAIN_REGION messages include non-promising regions; thus, instead of spending its effort within such discouraged regions, it prefers starting a new local search in a possibly unvisited part of the solution space. The content of REFRAIN_REGION messages are balls $\mathcal{B}(x_c, r)$ that are characterized by a center x_c and a radius r. The received regions are added to a *refrained regions* set $\mathcal{R}$ as new elements, or they are merged with existing elements of this set. Whenever its current iterate x_k falls inside one of the balls in $\mathcal{R}$, i.e,

$$\|x_k - x_c\| \leq r, \quad \text{for some } \mathcal{B}(x_c, r) \in \mathcal{R},$$

Agent T immediately stops its ongoing run and starts a new run from a different initial point. If it receives an INFORM_SOLUTION message (i.e., another agent's current best solution), then Agent T acknowledges the message content solution point x_r, and starts its next run from this point if it confirms that x_r is not a local minimizer of f.

This is likely to be the case when the sender of the INFORM_SOLUTION message is Agent R.

Agent R uses a simple random search as its algorithm. In a single run, it evaluates the value of the objective function at a set of points that are uniformly sampled from the feasible region. The agent repeatedly executes this procedure. Its *current solution* is the point with the minimum objective value within all the evaluated sample points up to that moment; so, it is updated if a better solution is obtained in the most recent run.

When its current solution is updated, Agent R sends this point to Agent T via an INFORM_SOLUTION message. On the other hand, it is able to benefit from REFRAIN_REGION messages. As Agent T, it keeps the content of received messages in a list. The evaluation of the objective function is dismissed at the sampled points that fall into one of the balls residing in that list of refrained regions. This agent is very quick in exploration and provides diversification; nonetheless, it may be very inefficient in finding a global optimal solution.

Note the flexibility of the interactions among agents. First, the communications need not be symmetric. For example, Agent R sends messages to Agent T, but only receives messages from Agent B. Second, an agent can send certain types of messages to a particular agent but not others. For example, Agent B sends INFORM_SOLUTION to Agent T but REFRAIN_REGION to Agent R. Such variations on communication can be easily adapted in MANGO environment with minimal modifications on the agent and no modifications on the agent's algorithm.

Another important point to note is that while agents are built to cooperate, they can still operate without cooperation. That is, none of these agents have to receive messages to start working or need to send messages to continue. The cooperation is only an added value to the agents. If the other agents in the system fail for some reason, the remaining agents can still operate. In a similar vein, addition of agents to the system would not need any modifications on the agents' realizations. For example, we could have several Agent R's in the system and they would all receive messages that are directed to them. This enables agents to enter or leave the system as it suits them.

Cooperation scenarios. We next present several cooperation scenarios to illustrate an actual implementation of the sample agents introduced above. Our main purpose in this section is to show the possible effects of communication on individual agents as well as on the overall success of the cooperation for solving a problem. For ease of exposition, we shall start with a base scenario involving a very simple cooperation among the agents. Then, we shall gradually extend this scenario by adding other communication channels among the agents and point out some important observations.

We test our scenarios on three global optimization problems from the literature with different attributes, such as; dimension, structure and application domain. Table 3.1 gives the details of our test problems. The first two problems are given by More *et al.* [50]. These problems are reformulations of systems of nonlinear equations as optimization problems. Therefore, the known global optimal objective function values for both problems are zero (see third column of Table 3.1). They both have multiple minima and involve rather flat regions causing performance deterioration for gradient-based methods. The last problem is related to finding the lowest energy configuration of a molecular system. This particular problem is taken from Lennard-Jones clusters with 15 atoms [72]. Here, we note that 3 times the number of atoms gives the problem dimension as shown in the second column of Table 3.1. To bound the feasible region for sampling, we have assumed the problems are box constrained with the variable bounds given in the last column. Naturally, we make sure that the global optimum for each problem resides within the imposed bounds.

Table 3.1: Problem details and parameters

Problem Name	Dimension	Global Optimum	Imposed Variable Bounds
`Rosenbrock`	2	0.0000	[-100, 100]
`Broyden Tridiagonal`	10	0.0000	[-100, 100]
`LJCluster-15`	45	-52.3226	[-5, 5]

In the subsequent part we solve these test problems for different communication scenarios. Since we use random sampling, we report for each problem the average statistics over 10 runs. All runs are terminated after a duration (wall clock time)

proportional to the problem dimension has elapsed. That is, for each problem, the time to complete a run is taken as the minimum of 5 seconds times the dimension and 5 minutes. Clearly, this setting is to the advantage of the runs taken for individual agents because the entire computing resource is then dedicated to a single agent. We also note that all results are obtained on a dual core personal computer with an Intel Core i5 processor and 4 GB of RAM.

We first start with the results obtained with the individual agents. These results demonstrate the performances of the agents when they are started from randomly selected points. Since it is very common to use local search methods in such a multi-start setting for solving global optimization problems, these results illustrate what would most of the decision makers do in practice. We shall later use these results for comparing against the results obtained with the communication scenarios. Table 3.2 gives the average objective function values obtained by two agents separately. Since Agent R applies a simple random search, we have observed that its results are very far away from the global optimum. Therefore, we omit its results for further comparison but note that Agent R plays a role in the communication scenarios.

In Table 3.2 the figures that we shall later use for comparison are marked with boldface letters. As the figures show both agents are able to solve the first problem. However, Agent T finds the global optimum solution within fewer number of function evaluations on average than Agent B. Therefore, Agent T is used for comparison. When we check the last two problems, we observe that only one agent can find a solution individually. For problem 2, Agent B fails to find the global optimum solution but Agent T does converge to the global optimum. However, the performances of the agents are reversed for the last problem, and Agent B converges to the global optimum whereas Agent T fails.

Next we discuss the communication scenarios as illustrated in Figure 3.12. In Table 3.3, we compare the results obtained with the scenarios against the individual results that are summarized from Table 3.2. The third column of Table 3.3 gives the average objective function value obtained by the most successful agent, which is the one reported in the last column. Likewise, the fourth column shows the percentages

Table 3.2: The average objective function values obtained by the individual agents for the test problems over 10 runs

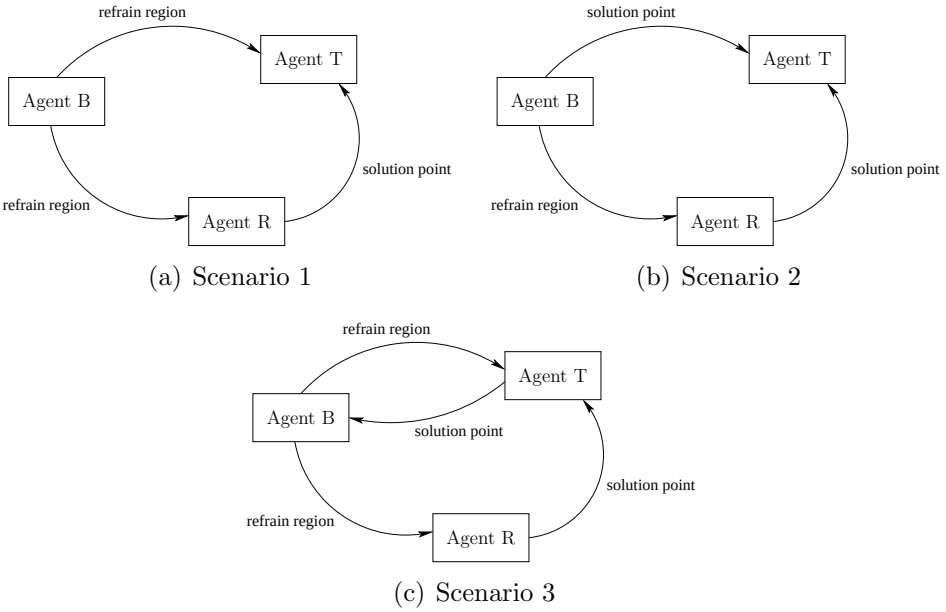
Problem Name	Agent B	Agent T
`Rosenbrock`	0.0000	**0.0000**
`Broyden Tridiagonal`	0.0745	**0.0000**
`LJCluster-15`	**-52.2270**	-47.2386

related to the average number of calls to the objective function until the best objective function value is recovered. These figures are given relative to the number of objective function calls required by the individuals. Thus, the values for the individual runs are omitted and the corresponding cells are marked with (-) signs. For instance, consider the problem `LJCluster-15`. In Scenario 1, Agent B is the most successful agent (see last column) in terms of average objective function value. As shown in the third column, Agent B required on average 82% of the number of function calls used by the individual runs for the same problem. However in the latter two scenarios the successful agents (Agent T for Scenario 1, Agent B for Scenario 2) have required on average slightly more function calls than the individual runs (4% and 5%, respectively). The fifth column gives a similar percentage comparison relative the individual runs in terms of wall-clock time.

Table 3.3: The average statistics over 10 runs for all communication scenarios

		Average OF*			Obtained by
		Value	Calls (%)	Time (%)	
Individuals	`Rosenbrock`	0.0000	-	-	Agent T
	`Broyden Tridiagonal`	0.0000	-	-	Agent T
	`LJCluster-15`	-52.2270	-	-	Agent B
Scenario 1	`Rosenbrock`	0.0000	43%	67%	Agent T
	`Broyden Tridiagonal`	0.0000	32%	57%	Agent T
	`LJCluster-15`	-52.1326	82%	79%	Agent B
Scenario 2	`Rosenbrock`	0.0000	25%	42%	Agent T
	`Broyden Tridiagonal`	0.0000	41%	54%	Agent T
	`LJCluster-15`	-52.3226	104%	84%	Agent T
Scenario 3	`Rosenbrock`	0.0000	23%	38%	Agent T
	`Broyden Tridiagonal`	0.0000	73%	93%	Agent T
	`LJCluster-15`	-52.2321	105%	109%	Agent B

*Objective function value



(a) Scenario 1

(b) Scenario 2

(c) Scenario 3

Figure 3.12: Communication scenarios

In the first scenario as shown in Figure 3.12(a), Agent B sends refrain messages to both Agent R and Agent T. The two receiving agents then try to avoid those regions exploited by Agent B. Aside from the refrain messages from Agent B, Agent T also receives some promising solution points to start with from Agent R. In this scenario, we expect Agent T to recover the global optimum quicker than it does when it works individually. As the average numbers of objective function calls in the fourth column of Table 3.3 show, for the first two problems, Agent T indeed finds the global optimum with less than half of the function calls it uses individually. As we observed with the individual runs, in the last problem Agent B is still the one that converges close to the global optimum. This is expected because Agent T is refrained from the regions that are exploited by Agent B, and hence, the success of Agent B in the vicinity of the global optimum keeps Agent T away from those regions. Although Agent B finds this solution faster than it does individually both in terms of wall-clock time and the number of objective function calls (columns 4 and 5), the quality of solution deteriorates slightly. This decrease in the solution quality can be attributed to the decrease in the computing power that is allocated to the individual agents when they communicate within a scenario.

Having observed the success of Agent B for the last problem, we next construct Scenario 2, where we try to lead Agent T to the global optimum in *all* problems. As illustrated in Figure 3.12(b), we accomplish this simply by replacing the refrain message sent from Agent B to Agent T with a solution point message. This change then encourages Agent T to start with those points, which have been recognized as promising but not properly exploited by Agent B particularly when, it terminates its procedure after exceeding the maximum number of iterations. The last column corresponding Scenario 2 in Table 3.3 shows that Agent T succeeds to find the global optimum in all problems. Moreover, it finds the exact global optimum in all 10 runs at the expense of a slight increase in the average number of objective function calls but an ample decrease in the wall-clock time. When it comes to the first two problems, as in Scenario 1, we observe a significant decrease in the average numbers of objective function calls as well as in the wall-clock times.

Figure 3.12(c) shows the last and the most versatile scenario in terms of communication. Unlike the previous two scenarios, Agent B now receives a feedback from Agent T. This feedback pays back for the last problem and Agent B finds a solution closer to the global optimum than the solutions it finds individually and in Scenario 1. However, this improvement comes with a small increase in the average number of objective function calls and the wall-clock time. On the other hand, we obtain the fastest convergence to the global optimum for the first problem with this scenario. Although not as good as the previous two scenarios, the global optimum for the second problem is found within fewer number of function calls than the individual runs. More remarkably, the wall-clock time to obtain this solution has significantly improved.

One important concern with distributed systems is related to their scalability. This concern can be posed as how much the system performance degrades when the size of the system increases in terms of number of threads and processed jobs. When discussing the scalability of MANGO in such a sense, we need to consider the increase in size with respect two different entities: agents and messages. Since each agent is a stand-alone Java program, the number of agents that can be run on a single machine depends on the individual specification of the machine. Nonetheless, as the MANGO

architecture permits running each agent on a different computer, the increase in the number of agents is not expected to cause a significant problem. Therefore, our focus in terms of scalability of physical resources is with the number of exchanged messages. Each message in the system passes through a central messaging system to reach its destination. If agents generate significantly more messages than that can be consumed by other agents, we expect to have delays and performance loss, even when the number of agents is low. This is an obvious observation, since in our implementation the agents shall then seize their major function of problem solving and devote all their resources to processing messages. This drawback can easily be avoided by limiting the number of messages sent and received by the agents, or processing the incoming messages only when the agent sees it fit.

Another important question about scalability is related to the performance improvement in solving a global optimization problem. That is, if we increase the number of agents when solving a particular problem, does this effort necessarily improve the success of the agents for finding the global optimum of a particular problem. Table 3.4 shows the average statistics over 10 runs for the ninety-dimensional cluster problem `LJCluster-30`. As before, we have assumed that each variable comes from the bounds $[-5, 5]$. The objective function value of the global optimum for this particular problem is -128.2515. The number of function calls are given relative to the numbers obtained with 2 agents (row 3). The results in the table demonstrate that as we increase the number agents up to a certain value we do obtain performance increase. When we reach 10 agents in total, we finally recover the global optimum in the best run. However, as we continue increasing the number of agents, the performance deteriorates. This can be attributed to two reasons: (i) the agents could be overwhelmed by processing excessive message passing, (ii) too much information creates pollution. The latter reason may be complemented with further explanation: As each agent tries to direct the others to different parts of the feasible region, it is quite possible that, particularly in earlier iterations, agents receive conflicting messages and hence, fail to explore the feasible region properly. Consequently, we note that one should not indiscriminately assume that the performance shall increase as the number of agents increase. The optimal parameter

for the number of agents is clearly problem-dependent and unfortunately, requires some parameter fine-tuning.

Table 3.4: The average statistics over 10 runs for problem LJCluster-30

Number of Agents	OF*Value		OF Calls (%)		Time (%)		Obtained by
	Best	Average	Best	Average	Best	Average	
1 B, 1 T	-127.4218	-126.6161	-	-	-	-	Agent T
2 B, 2 T	-128.0966	-126.2599	60%	95%	59%	147%	Agent B
5 B, 5 T	**-128.2515**	-126.9845	79%	65%	157%	128%	Agent B
10 B, 10 T	-127.4218	-126.5504	11%	25%	66%	146%	Agent T

*Objective function value

Chapter 4

PROPOSED INHERENTLY PARALLEL ALGORITHMS

The concurrent search framework of the previous chapter follows the approach of this thesis at a high level, in the sense that the tasks providing its inherent parallelism consist of complete algorithmic procedures. Thus, it can be seen as an extension of the included methods, since the interaction among the original operations causes their modification. In this chapter, we introduce two new algorithms that follow a similar approach at a lower level. In other words, we aim at designing algorithms that are themselves inherently parallel. The first algorithm is designed for unconstrained problems, whereas the second one is a constrained optimization algorithm.

4.1 A Parallel Algorithm for Unconstrained Optimization

The algorithm that shall be described in this section is a new algorithm, which aims to achieve inherent parallelism in lower level computational tasks. The *additional* operations are mainly used in constructing the model function for step computation.

4.1.1 Algorithm

Basic ideas. Classical methods of unconstrained optimization are based on local models of the problem, that are very powerful once the algorithm arrives at a close proximity of a local solution. However, the local models may not provide very efficient

steps if the algorithm starts from a *remote* point. The algorithm in this section attempts to use some extra problem information produced by additional computational resources. This extra information is then utilized to take *globally more efficient* steps as in the concurrent search example. However, unlike concurrent search, the proposed algorithm is based on a completely new model function. It has operations that can be executed in parallel but the output of individual threads are all lower level values that are not meaningful by themselves.

Thus, our basic motivation —as before— is to design a *new* algorithm that achieves both a good parallelization performance (i.e. an inherently parallel structure of computations) and a good solution performance as compared to some standard sequential algorithms. As we continue describing our algorithm, its connection with certain well-known methods, like nonlinear conjugate gradient algorithm, will become clear.

The new method starts its iterations with a gradient-related related direction and then, instead of applying a standard line search, it tries to improve this direction as well as determine its length. This is achieved by using an *extended* model function, that is constructed based on two linear models at two reference points. The step computation is done in an almost completely separable way. This provides the scalability of the new parallel algorithm. Furthermore, the scalability is maintained by keeping the synchronization operations at a minimum level. Another nice property of the new algorithm is that it is almost parameter-free.

Before we start explaining the new algorithm, we would like to emphasize that the consideration of a *global* model that can reduce to the *local* models at multiple reference points led us designing the new algorithm of this section, rather than concentrating on the parallel implementation ideas that are originated from the existing large-scale unconstrained optimization methods.

In the remaining part of this section, we shall describe the details of the step computation, then discuss its expected and observed parallelization properties. The section continues with the implementation details and a formal discussion on the convergence properties of the proposed method. We then conclude with some numerical results on

a set of problems.

Step computation. Let us start by considering the kth iteration of the algorithm with the current iterate x_k. To obtain the next iterate of the algorithm, x_{k+1} we need to find s_k so that we can set $x_{k+1} = x_k + s_k$. The proposed algorithm obtains s_k by executing inner iterations $t = 0, 1, \cdots$ and computing trial steps s_k^t. When this procedure exits, the incumbent s_k^t becomes s_k that we are after.

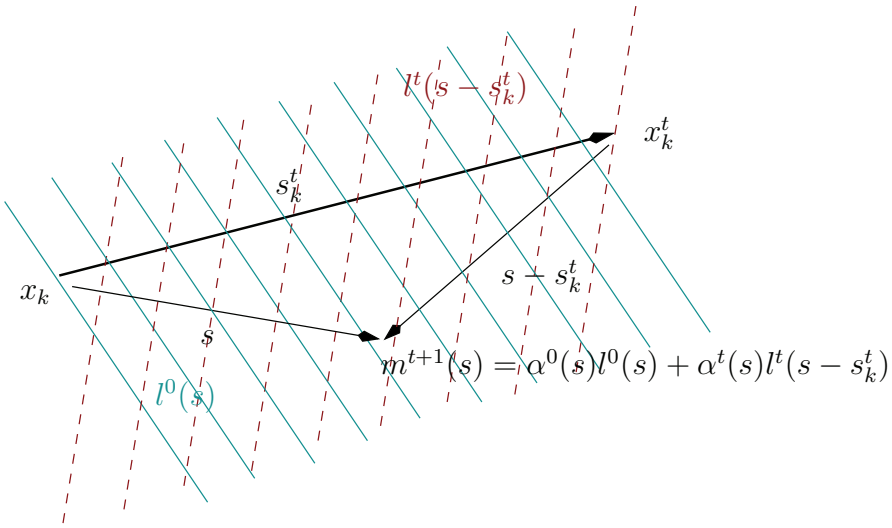


Figure 4.1: Construction of the model function $\hat{m}^{t+1}(d)$

To simplify the exposition, let us first define

$$x_k^t := x_k + s_k^t, \quad f_k^t := f(x_k^t), \quad g_k^t := \nabla f(x_k^t), \quad y_k^t := g_k^t - g_k, \quad \text{and} \quad \delta_k^t := (s_k^t)^T s_k^t.$$

At each inner iteration, the trial step, s_k^t is accepted as s_k, if it satisfies

$$f(x_k + s_k^t) - f_k \leq \rho g_k^T s_k^t \tag{4.1}$$

for some $\rho \in (0, 1)$. Otherwise, we use the information gathered around x_k and $x_k + s_k^t$ to come up with a model function that hopefully has a similar local behavior as f within the region in between those two reference points. Using already available f_k, g_k, f_k^t and g_k^t values, this model function is constructed in two steps. In the first step, a quadratic function based on combination of two linear reference models at x_k and $x_k + s_k^t$ are

constructed. That is, we start with

$$\hat{m}^{t+1}(s) := \alpha^0(s)l^0(s) + \alpha^t(s)l^t(s - s_k^t),$$

where

$$l^0(s) := f_k + g_k^T s,$$

$$l^t(s - s_k^t) := f_k^t + (g_k^t)^T(s - s_k^t).$$

This construction is illustrated in Figure 4.1.1. Note that both weights, $\alpha^0(s)$ and $\alpha^t(s)$ are functions of s. We shall show in our subsequent discussion that by construction, these weights always provide a convex combination, i.e., they are in $[0, 1]$ and add up to 1. We want to make sure that if s is closer to s_k^t, then the weight $\alpha^t(s)$ of the linear model around x_k increases. Similarly, $\alpha^t(0)$ should increase as $(s - s_k^t)$ gets closer to $(-s_k^t)$. This is achieved by measuring the length of the projections of s and $(s - s_k^t)$ on s_k^t as

$$\alpha^0(s) = \frac{(s - s_k^t)^T(-s_k^t)}{(-s_k^t)^T(-s_k^t)} \quad \text{and} \quad \alpha^t(s) = \frac{s^T s_k^t}{(s_k^t)^T s_k^t}. \tag{4.2}$$

In the second step, we control the lengths of s and $(s - s_k^t)$ by adding two regularization terms to $\hat{m}^{t+1}(s)$. Thus, our final model function becomes

$$m^{t+1}(s) = \hat{m}^{t+1}(s) + \frac{1}{2}\beta_1 s^T s + \frac{1}{2}\beta_2(s - s_k^t)^T(s - s_k^t). \tag{4.3}$$

After some derivation, the quadratic model (4.3) simplifies to

$$m^{t+1}(s) = f_m + g_m^T s + \frac{1}{2}s^T B_m s, \tag{4.4}$$

where

$$f_m := f_k + \tfrac{1}{2}\beta_2\delta_k^t,$$

$$g_m := g_k + \left(\frac{1}{\delta_k^t}(f_k^t - f_k + (g_k^t)^T(-s_k^t)) - \beta_2\right) s_k^t,$$

and

$$B_m := \frac{1}{\delta_k^t}((\beta_1 + \beta_2)I + s_k^t(y_k^t)^T + y_k^t(s_k^t)^T).$$

The parameters β_1 and β_2 play an important role in generating descent directions for the original function f, by using model function (4.4). These two parameters are also crucial to guarantee that $s = 0$ is not the minimizer for the model function $m^{t+1}(s)$, unless the current iterate is a stationary point for the original function. In other words, since

$$\|\nabla m^{t+1}(0)\|^2 \geq \|g_k\|^2,$$

the model m^{t+1} shall always provide a nonzero step s^{t+1} unless $\|g_k\| = 0$. Therefore, to provide a definite relationship between ∇m^{t+1} and ∇f, we set

$$\beta_1 = \frac{1}{\delta_k^t}(f_k - f_k^t + g_k^T s_k^t) \quad \text{and} \quad \beta_2 = \frac{1}{\delta_k^t}(f_k^t - f_k - (g_k^t)^T s_k^t). \tag{4.5}$$

Clearly, β_1 and β_2 may take negative values at some iterations. Interestingly, if f is convex, they are always negative and a trial point close to any of the reference points is discouraged. However, it is not difficult to show that both β_1 and β_2 are bounded, when we assume that f is twice Lipschitz continuous. To be exact, we have

$$|\beta_1| \leq \frac{L}{2} \quad \text{and} \quad |\beta_2| \leq \frac{L}{2},$$

where L is the Lipschitz constant for f.

When the parameter values β_1 and β_2 are set as given in (4.5), the components of the model function (4.4) become

$$f_m = \frac{1}{2}(f_k + f_k^t - (g_k^t)^T s_k^t), \qquad g_m = g_k,$$

and

$$B_m = \frac{1}{\delta_k^t}(-((y_k^t)^T s_k^t)I + s_k^t(y_k^t)^T + y_k^t(s_k^t)^T).$$

Remark 4.1.1 summarizes the established relationship between the model function m^{t+1} and the original objective function f.

REMARK 4.1.1 *The model function $m^{t+1}(s)$ satisfies*

$$m^{t+1}(s_k^t) = f_k^t + \frac{1}{2}\beta_1\delta_k^t = \frac{1}{2}(f_k^t + f_k + g_k^T s_k^t),$$

and

$$m^{t+1}(0) = f_k + \frac{1}{2}\beta_2\delta_k^t = \frac{1}{2}(f_k + f_k^t - (g_k^t)^T s_k^t).$$

Therefore,

$$\min_d \; m^{t+1}(d) < \frac{1}{2}(f_k + f_k^t).$$

Moreover, the gradient of the model function at the two reference points, x_k and $x_k + s_k^t$ are given by

$$\nabla m^{t+1}(0) = g_k + \left(\frac{1}{\delta_k^t}(f_k^t - f_k + (g_k^t)^T(-s_k^t)) - \beta_2\right) s_k^t$$

and

$$\nabla m^{t+1}(s_k^t) = g_k^t + \left(\frac{1}{\delta_k^t}(f_k - f_k^t + g_k^T s_k^t) - \beta_1\right)(-s_k^t),$$

respectively. This implies

$$\nabla^{t+1} m(0) = g_k \quad \text{and} \quad \nabla^{t+1} m(s_k^t) = g_k^t.$$

As indicated in Remark 4.1.1, the model values at x_k and x_k^t correspond to the averages of the original function value f and the approximations of the linear models l^t and l^0 at those points, respectively. Since the model gradient at $s = 0$ is equal to the gradient of f at x_k, any descent direction computed for m^{t+1} is also a descent direction for f.

The next step is to find an approximate solution to subproblem (4.4). We first start as if the model function $m^{t+1}(s)$ is convex (we shall later consider the nonconvex case). If we denote the minimizer of $m^{t+1}(s)$ by s^N, then after some derivation we obtain

$$s^N = \theta^g g_k + \theta^y y_k^t + \theta^s s_k^t, \tag{4.6}$$

where

$$\theta^g = \frac{\delta_k^t}{(y_k^t)^T s_k^t},$$

$$\theta^y = -\frac{\delta_k^t}{(y_k^t)^T s_k^t} \frac{(y_k^t)^T g_k}{\|y_k^t\|^2},$$

and

$$\theta^s = -\frac{\delta_k^t}{(y_k^t)^T s_k^t} \frac{(s_k^t)^T g_k}{\|s_k^t\|^2} = -\frac{(s_k^t)^T g_k}{(y_k^t)^T s_k^t}.$$

Note that the basic trial stepin inner iteration $t+1$ is in the subspace spanned by g_k, g_k^t, and the previous trial step s_k^t. We should also emphasize at this point that the computation of s^N requires just a few floating point operations, once the necessary inner products are completed. Therefore, the computation of s^N is very suitable for parallelization, since inner products require only n independent basic operations. Furthermore, there is no need to wait for the completion of an inner product before starting another one; i.e., they are all independent. As (4.6) is the main component of the trial step computation used in our algorithm, we are on the right track to come up with a fast and scalable method. In Section 4.1.2, we shall elaborate on the number of basic operations required by the proposed algorithm and discuss in detail its almost-purely parallel implementation.

We handle the nonconvex model function m^{t+1} by applying a constraint on the length of the selected step, s. Such a restriction on the length is crucial because a nonconvex m^{t+1} may not have a bounded minimizer along the directions given by some s. Recall that our model function is constructed *around* the previous trial step s_k^t, by considering the area lying between x_k and $x_k + s_k^t$. With this consideration in mind, we require our step s satisfy

$$\|s\|^2 + \|s - s_k^t\|^2 \leq \|s_k^t\|^2. \tag{4.7}$$

Note that this constraint controls not only the lenght of s but also its variation from the previous trial point s_k^t. More importantly, this choice of the constraint ensures that both weight functions $\alpha^0(s)$ and $\alpha^t(s)$ are in $[0, 1]$. This observation is formally given in Lemma 4.1.1.

LEMMA 4.1.1 *The weighting functions α^0 and α^t given in* (4.2) *satisfy*

$$\alpha^0(s) \in [0,1] \quad \textit{and} \quad \alpha^t(s) \in [0,1],$$

provided that the constraint (4.7) *holds at inner iteration $t+1$ of iteration k.*

PROOF. By constraint (4.7), we have

$$\|s\|^2 + \|s - s_k^t\|^2 \leq \|s_k^t\|^2 \quad \Rightarrow \quad 0 \leq s^T s_k^t \leq (s_k^t)^T s_k^t.$$

Both sides of the inequality on the right can be obtained by

$$0 \geq s^T(s - s_k^t) = \|s\|^2 - s^T s_k^t \geq -s^T s_k^t \implies s^T s_k^t \geq 0,$$

and

$$\begin{aligned} s^T s_k^t - (s_k^t)^T s_k^t &= (s_k^t)^T (s - s_k^t) \\ &= (-(s - s_k^t) + s)^T (s - s_k^t) \\ &= -\|s_- s_k^t\|^2 + s^T(s - s_k^t) \leq 0 \implies s^T s_k^t \leq (s_k^t)^T s_k^t. \end{aligned}$$

□

Note that the region defined by (4.7) is never empty. First of all, any step $s = \gamma s_k^t$ is feasible. Also, there is always a feasible step in the steepest descent direction. In fact, the step minimizing m^{t+1} along that direction by satisfying (4.7) is given by $s^C = -\theta^C g_k$, where

$$\theta^C = \min\left\{\frac{g_k^T s_k^t}{g_k^T g_k}, \max\left\{0, \frac{\delta_k^t (g_k^T g_k)}{2(g_k^T y_k^t)(g_k^T s_k^t) - (g_k^T g_k)((y_k^t)^T s_k^t)}\right\}\right\}.$$

The only loose end in our algorithm is how to determine s_k^0, i.e, the direction to start the inner iterations. We shall discuss in our analysis that any gradient related direction can be chosen for s_k^0 to obtain a convergent algorithm. However, we note that at iteration k, we already have the step, s_{k-1} and the gradient, g_{k-1} of the previous iteration as well as the current gradient, g_k. Therefore, we choose to apply the so-called

memoryless BFGS method and obtain a local model at x_k as

$$m_k^0(s) = f_k + g_k^T s + \frac{1}{2} s^T \left(\beta I - \beta \frac{s_{k-1} s_{k-1}^T}{s_{k-1}^T s_{k-1}} + \frac{y_{k-1} y_{k-1}^T}{y_{k-1}^T s_{k-1}} \right) s.$$

The curvature term in m_k^0 is obtained by applying the standard BFGS update formula with $B_{k-1} = \beta I$ and the pair (s_{k-1}, y_{k-1}), where $y_{k-1} = g_k - g_{k-1}$. Consequently, the first step of the inner iterations becomes

$$s_k^0 = -\frac{1}{\beta} g_k + \left[\frac{1}{\beta} \frac{y_{k-1}^T g_k}{y_{k-1}^T s_{k-1}} - \frac{s_{k-1}^T g_k}{y_{k-1}^T s_{k-1}} \left(1 + \frac{1}{\beta} \frac{y_{k-1}^T y_{k-1}}{y_{k-1}^T s_{k-1}} \right) \right] s_{k-1} + \frac{1}{\beta} \frac{s_{k-1}^T g_k}{y_{k-1}^T s_{k-1}} y_{k-1}. \quad (4.8)$$

It is well-known that the *curvature condition* $s_{k-1}^T y_{k-1} > 0$ is required to be able to guarantee that the resulting step s_k^0 follows a descent direction[53].

Finally, note that we have $s_k^0 \in \text{span}(g_{k-1}, s_{k-1}, g_k)$; for the step s^N given in (4.6) this yields $s^N \in \text{span}(g_{k-1}, s_{k-1}, g_k, g_k^1, \ldots, g_k^t)$.

4.1.2 Implementation

The overall algorithm fits into the basic framework covering most iterative nonlinear programming algorithms. An outline, explaining the computation of trial steps s_k^t, is given in Algorithm 7. The details of the step computation and its parallel implementation follow.

Standard inner iteration. Following the discussion in the previous section, we define our step computation subproblem in inner iteration $t+1$ of iteration k as

$$\begin{aligned} &\text{minimize} && m^{t+1}(s), \\ &\text{subject to} && \|s\|^2 + \|s - s_k^t\|^2 \leq \|s_k^t\|^2. \end{aligned} \quad (4.9)$$

At a standard inner iteration, basically a bunch of inner products shall be required to compute the step and evaluate its success. Let us introduce the following notation

Algorithm 7: Overall algorithm

1 **Input**: x_0
2 $k = 0$
3 **while** *checkConvergence()=FALSE* **do**
4 $k = k + 1$
5 $t = 0$
6 Compute s_k^0 using (4.8).
7 **while** *checkStepAcceptability()=FALSE* **do**
8 Compute $v_1, v_2, v_3, v_4, v_5, v_6$.
9 Compute $\theta^g, \theta^s, \theta^y$ as in (4.10), and γ as in (4.11)
10 Compute $v_7 = g_k^T s^N$ and κ as given in (4.12) and (4.13), respectively.
11 Set the value of trial step s_k^{t+1}:

$$s_k^{t+1} = \begin{cases} s^N & \text{if } v_7 \leq -\epsilon v_5 \text{ and } \kappa > 1, \\ \kappa s^N & \text{if } v_7 \leq -\epsilon v_5 \text{ and } \kappa < 1, \\ \gamma s_k^t & \text{if } v_7 > -\epsilon v_5 \text{ or } \kappa = 1. \end{cases}$$

  $t = t + 1$
12 $x_k^t = x_k + s_k^t$
13 $x_{k+1} = x_k^t$

for these vector multiplications

$$v_1 = (s_k^t)^T y_k^t, \qquad v_2 = (s_k^t)^T s_k^t, \qquad v_3 = (y_k^t)^T y_k^t,$$
$$v_4 = (y_k^t)^T g_k, \qquad v_5 = g_k^T g_k, \qquad v_6 = (s_k^t)^T g_k.$$

An (approximate) solution to subproblem (4.9) is obtained as given in lines 8-11 of Algorithm 7. The details of these computations are as follows:

$$\theta^g = \frac{v_2}{v_1}, \qquad \theta^y = -\frac{v_2}{v_1}\frac{v_4}{v_3}, \qquad \theta^s = -\frac{v_6}{v_1}, \tag{4.10}$$

$$\gamma = -\frac{v_6}{2(f_k^t - f_k - v_6)}, \tag{4.11}$$

$$v_7 = g_k^T s^N = \theta^g v_5 + \theta^y v^4 + \theta^s v_6, \tag{4.12}$$

$$\kappa = \frac{\theta^g v_6 + \theta^y v^1 + \theta^s v_2}{(\theta^g)^2 v_5 + (\theta^y)^2 v^3 + (\theta^s)^2 v_2 + 2(\theta^g \theta^y v_4 + \theta^y \theta^s v^1 + \theta^g \theta^s v_6)}. \tag{4.13}$$

Here, the relation (4.10) is obtained by simply rewriting (4.6), the relation (4.13) computes the length of the projection of s_k^t on s^N, and the relation (4.11) is a steplength in the direction of s_k^t obtained via quadratic interpolation. There is no need to do another vector multiplication during the acceptability check since we have

$$g_k^T s_k^{t+1} = \begin{cases} v_7 & \text{if } s_k^{t+1} = s^N, \\ \kappa v_7 & \text{if } s_k^{t+1} = s^N, \\ \gamma v_6 & \text{if } s_k^{t+1} = \gamma s_k^t. \end{cases}$$

Clearly, the total cost of non-parallelized operations becomes negligible as n increases. The costly operations are completely done in independent subdomains as explained in detail in Figure 4.2.

First inner iteration. Recall that the first inner iteration of the algorithm is computed using the memoryless BFGS update as given in (4.8). Similar to the standard inner iteration, the most costly operations are vector multiplications. In particular, the following inner products are required.

$$\begin{aligned} u_1 &= y_{k-1}^T g_k, \qquad u_2 = y_{k-1}^T s_{k-1}, \qquad u5 = g_k^T g_k, \\ u_3 &= s_{k-1}^T g_k, \qquad u_4 = y_{k-1}^T y_{k-1}. \end{aligned}$$

Clearly, the computational properties of the first inner iteration is very close to that of iteration $t > 1$. As a side note, we note that the initial scaling is carried out by setting $\beta = u_4/u_2$.

To be able to compute the memoryless BFGS step d_1^o at the first iteration, the value of ∇f is computed at $x_0 + s_0$ prior to the computation of d_1^0, where s_0 is a very tiny step along the gradient at x_0. This can be seen as a (kind of) preprocessing operation since it provides an idea about the scale of the problem, and hence, enables setting the initial β reasonably.

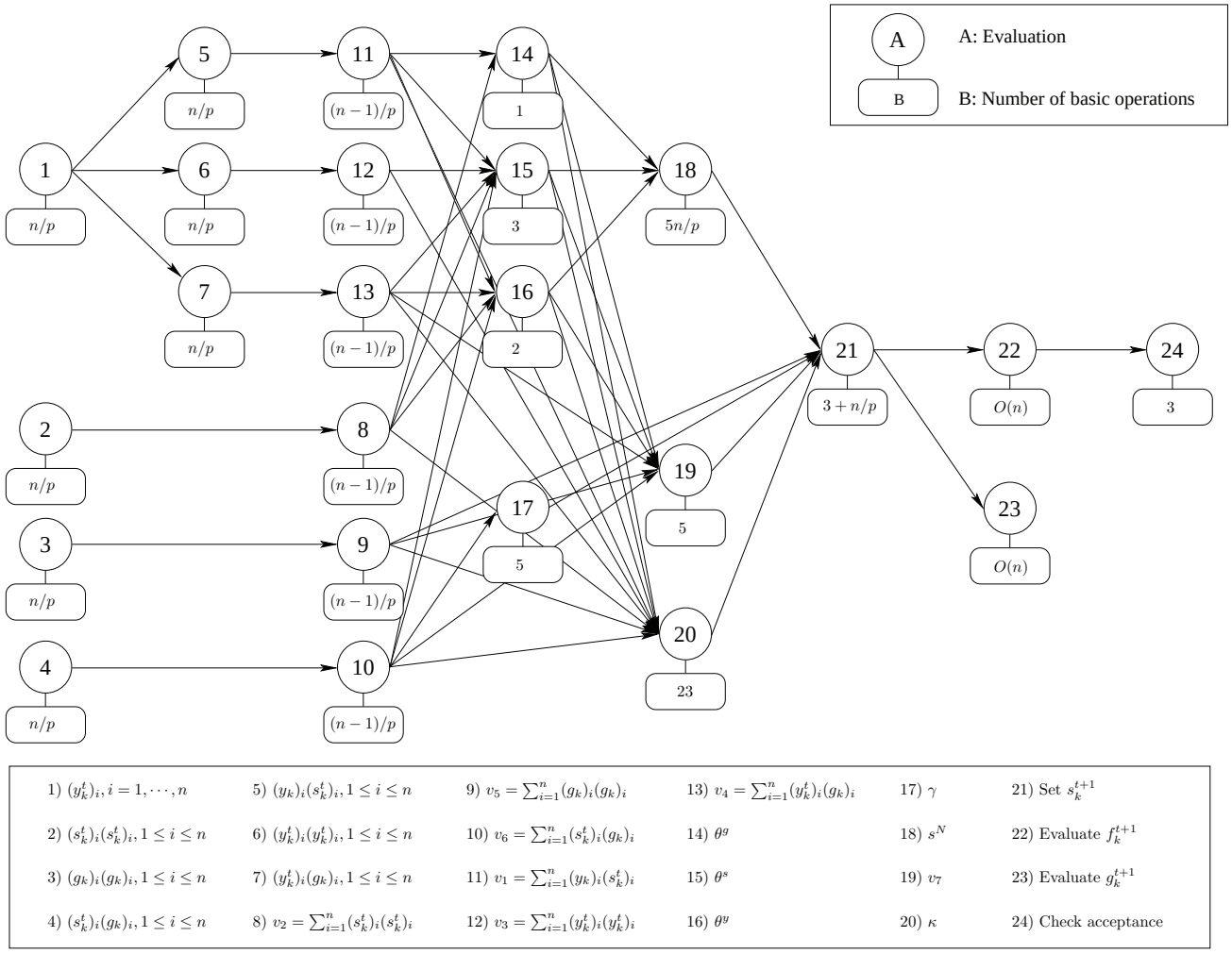


Figure 4.2: Parallelization of the proposed algorithm

4.1.3 Convergence

The convergence of the proposed algorithm is provided basically by keeping the directions of its steps *gradient related*. However, we could not directly refer to the existing convergence results for line-search algorithms since the directions of steps of our algorithm change during inner iterations, and the steplength is not computed through a one dimensional function. In the following convergence note, we only assume that the objective function f is continuous and differentiable, and its gradient ∇f is continuous and bounded.

LEMMA 4.1.2 *For any two consecutive inner iterations t and $t+1$ at iteration k of Algorithm 7, the trial steps satisfy*

$$\|s_k^{t+1}\| \leq \nu \|s_k^t\|, \quad \text{for some } 0 < \nu < 1.$$

PROOF. First recall that the feasible solution $s_k^{t+1} = s_k^t$ is not allowed by the algorithm. Suppose that $\|s_k^{t+1} - s_k^t\| \geq \tau \|s_k^t\|$, for some small enough $\tau < 1$. Then relation (4.7) provides

$$\|s_k^{t+1}\|^2 + \|s_k^{t+1} - s_k^t\|^2 \leq \|s_k^t\|^2 \quad \Rightarrow \quad \|s_k^{t+1}\|^2 \leq \|\nu s_k^t\|^2$$

for $\nu = \sqrt{1-\tau^2}$. Since $\|s_k^t\|, \|s_k^{t+1}\| > 0$, this implies $\|s_k^{t+1}\| \leq \nu \|s_k^t\|$. □

LEMMA 4.1.3 *Suppose s_k^t is a sufficient descent step in the sense that*

$$g_k^T s_k^t \leq -\epsilon g_k^T g_k \quad \text{and} \quad \|s_k^t\| \leq M \|g_k\| \tag{4.14}$$

for some small enough $\epsilon > 0$,and some finite $M > 0$. Then, the next trial step s_k^{t+1} produced by the algorithm is also a sufficient descent step.

PROOF. Recall that, as a safeguard operation, the algorithm sets $s_k^{t+1} = \gamma s_k^t$ with $\gamma \in (0,1)$ if s^N in (4.6) does not satisfy the first condition in (4.14) with $g_k^T s^N \leq -\epsilon g_k^T g_k$ for any given ϵ (see Algorithm 7, line 11). Moreover, γs_k^t is a descent step for m^{t+1} at

$d = 0$, since

$$\nabla m^{t+1}(0)^T \gamma s_k^T = \gamma g_k^T s_k^t < 0,$$

and it always lies in the feasible region defined by (4.7). Let $\epsilon > 0$ be small enough so that s_k^t satisfies the first condition in (4.14) for $\epsilon > (1/\gamma)c$ with $c > 0$. Then, γs_k^t satisfies it for $\epsilon > c$.

The second condition in (4.14) is enforced by relation (4.7). If $\|s_k^t\| \leq M\|g_k\|$, then $\|s_k^{t+1}\| \leq \nu M\|g_k\| \leq M\|g_k\|$ since $\|s_k^{t+1}\| \leq \nu\|s_k^t\|$ by Lemma 4.1.2, and $\|g_k\| > 0$. This shows the desired result. □

COROLLARY 4.1.1 *The steps produced by Algorithm 7 satisfies the following.*

I. All inner iterations of the algorithm produce sufficient descent steps in the sense of (4.14).

II. The step computed at inner iteration t of iteration k satisfies $\|s_k^t\| \geq \epsilon\|g_k\|$, for some small $\epsilon > 0$.

PROOF. Part (I) follows by Lemma 4.1.3 since the first inner iteration s_k^0 produced by the standard memoryless BFGS method is a sufficient descent step. Part (II) directly follows from (4.14). □

LEMMA 4.1.4 *At any iteration k, the algorithm finds an acceptable step in finite number of inner iterations, unless $\|g_k\| = 0$.*

PROOF. Consider any iteration k. By Corollary 4.1.1, the steps computed at each inner iteration satisfy (4.14). Therefore, $0 \leq \|s_k^t\| < M\|g_k\|$, for all t. Moreover, $\{\|s_k^t\|\}$ is a monotonically decreasing sequence by Lemma 4.1.2.

Suppose the acceptance criterion (4.1) is never satisfied as $t \to \infty$. Then, by using (4.14) and Taylor's theorem, we have

$$\begin{aligned} & f_k^t - f_k > \rho g_k^T s_k^t, \quad \text{for all } t \\ \implies & g_k^T s_k^t + o(\|s_k^t\|) > \rho g_k^T s_k^t, \\ \implies & o(\|s_k^t\|) > (1-\rho)(-g_k^T s_k^t) \geq (1-\rho)\epsilon\|g_k\|^2. \end{aligned}$$

As $t \to \infty$, the left hand side of the last inequality approaches to zero since $\|s_k^t\| \to 0$, but the left hand side stays positive. This gives a contradiction, and hence, an acceptable $\|s_k^t\|$ should be obtained in finite number of inner iterations. □

THEOREM 4.1.1 *Let $\{x_k\}$ be the sequence of iterates generated by Algorithm 7. Then, any limit point of $\{x_k\}$ is a stationary point of the objective function f.*

PROOF. Consider any subsequence of $\{x_k\}$ with indices $k \in \mathcal{K}$ such that

$$\lim_{k \in \mathcal{K}} x_k = \hat{x}.$$

By Lemma 4.1.4, there exists $\|s_k\| > \xi$ satisfying (4.1) for some $\xi > 0$ at any iteration k. So, for any $k, k' \in \mathcal{K}$ with $k' > k$ and by using (4.14), we have

$$f_k - f_{k'} \geq f_k - f_k^t \geq -\rho g_k^T s_k \geq \rho\epsilon \|g_k\|^2. \tag{4.15}$$

Since $\{x_k\}, k \in \mathcal{K}$, converges to $\hat{x}$, the continuity of f implies that $f_k \to \hat{f}$, $k \in \mathcal{K}$. Therefore $f_k - f_{k'} \to 0$ as $k, k' \to \infty$, and we obtain $\nabla f(\hat{x}) = 0$ by (4.15) and the continuity of ∇f.

□

4.1.4 Practical Performance

In this section, we shall provide some results on the performance on the proposed algorithm by checking both its parallelization and solution success.

The algorithm is coded in C++ using Intel Threading Building Blocks (TBB) as we have done for the CCS algorithm of the previous chapter. All tests are conducted on a 64-bit computer with two Quad-Core Intel(R) Xeon(R) CPUs running at @ 2.66GHz.

We have selected 3 problems from the CUTEr collection[31], whose dimensions can be varied to obtain small- to large-scale problems. This shall be essential for for testing the scalability of the proposed algorithm. The selected problems are COSINE, NONCVXUN and QUARTC. We used the tolerance value of $1.0e{-}5$ in all tests, the step

acceptability is checked using condition (4.1) with $\rho = 0.1$. We set both the maximum number of inner iterations and the maximum number of outer iterations equal to 100.

Let us start with some examples to show the contributions of the *extra* computations to the solution process. To test this, we solve all three problems with their default dimensions using the new algorithm and also with the memoryless BFGS line-search algorithm using quadratic interpolation. (Recall that the first inner iteration of the new algorithm is computed by using the memoryless-BFGS formula.) In this test, both algorithms are run sequentially. The results are given in Table 4.1. We note that the column entitled "Iterations" indicates the number of outer iterations. As the figures in Table 4.1 show, the use of extra information has a nice potential to contribute to the solution performance. For NONCVXUN, the new algorithm in fact makes less inner iteration computations but this may not be sufficient to close the gap caused by the extra gradient evaluations.

Second, we test the scalability of the algorithm. We set the dimension of all three problems to $N = 10,000$, $N = 100,000$, and $N = 1,000,000$, and solve each of the resulting instances using the new algorithm and by creating $p \in \{1, 2, 4, 6, 8\}$ threads. In Table 4.2, we give the complete set of time values obtained. Figure 4.3 gives the plots on the achieved speed-ups. As expected, better speed-up values are obtained as the problem sizes increase. However, even the largest test problem instances we have solved require less computational resources than the full capacity available with our eight-core machine. This explains the decrease in the slopes of the speed-up lines in Figure 4.3 as the number of threads increase.

We next focus on the resource usage ability of the new algorithm, which is an indicator of its inherent parallelism. We give the efficiency values in Table 4.3. As the numbers in this table indicate, the efficiency increases as the dimension of the problems increase. However, the workload becomes relatively small to fully use the entire resources as the number of threads approach to the number of physical cores.

Finally, we consider the load balance issue. To observe the usage of capacity and the workload distribution of the resources during the solution process, we plot the CPU usage during the solution of 1.000.000 dimensional instance of the problem COSINE

when 1,2 and 8 threads exist (see Figure 4.4). The plots are obtained by recording the CPU usage information per second. Figure 4.4(a) reveals the idle resources when the program runs sequentially. In fact, during the sequential run, the average resource usage stays at a level of only %7.7 (the average usage of the eight cores). When eight threads are used, this average raises up to %62. Figure 4.4(c) shows the usage of all available resources, as well as the distribution of workload.

Table 4.1: Contribution of extra computations

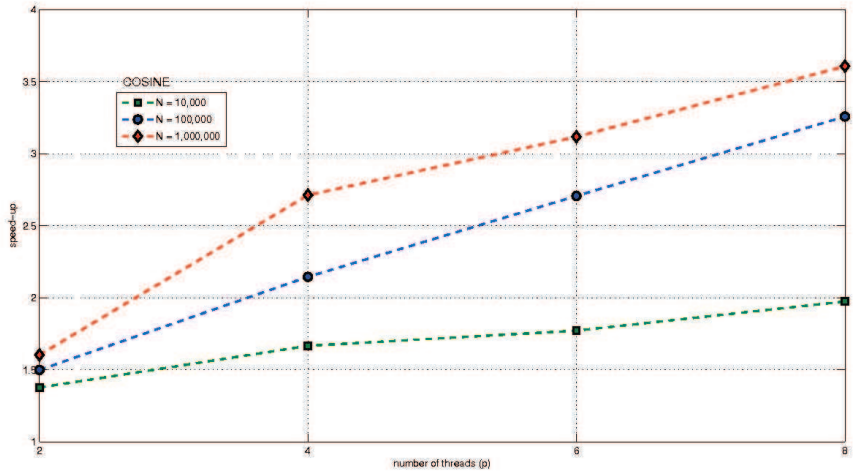
		New Algorithm		Memoryless BFGS	
Problem	Dimension	Iterations	Time (sec.)	Iterations	Time (sec.)
COSINE	10,000	10	0.050318	13	0.060445
NONCVXUN	1,000	17	0.004366	17	0.004361
QUARTC	10,000	57	0.406762	101(fail)	0.928862

Table 4.2: Solution times (in seconds) for varying values of N and p

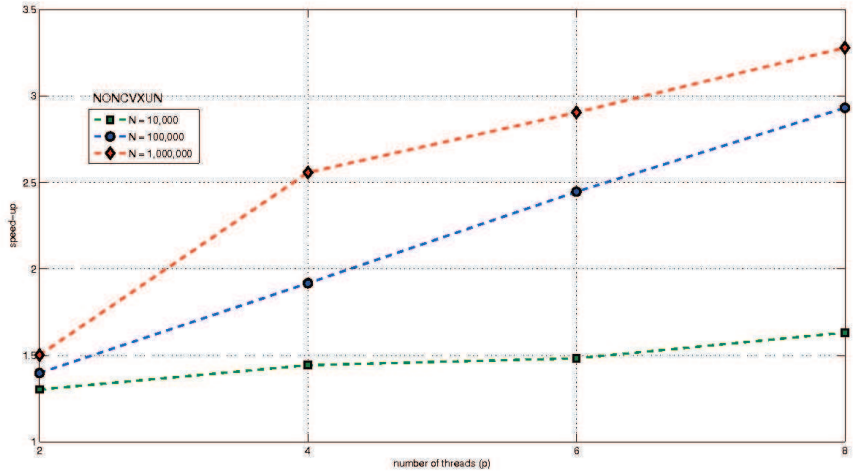
Problem	Dimension	$p = 1$	$p = 2$	$p = 4$	$p = 6$	$p = 8$
COSINE	10,000	0.050318	0.036539	0.030190	0.028396	0.025487
COSINE	100,000	0.520222	0.346923	0.242567	0.192243	0.159741
COSINE	1,000,000	7.10205	4.42888	2.61853	2.279	1.9695
NONCVXUN	10,000	0.051121	0.039229	0.035408	0.034462	0.031322
NONCVXUN	100,000	0.58058	0.415344	0.302796	0.23733	0.198083
NONCVXUN	1,000,000	7.76298	5.16787	3.03568	2.6724	2.36767
QUARTC	10,000	0.406762	0.258718	0.217847	0.200329	0.174812
QUARTC	100,000	1.91539	1.15041	0.735746	0.593911	0.472822
QUARTC	1,000,000	20.473	11.7138	7.04873	5.58571	4.66796

Table 4.3: Efficiency of the parallel program as the problem sizes increase(%)

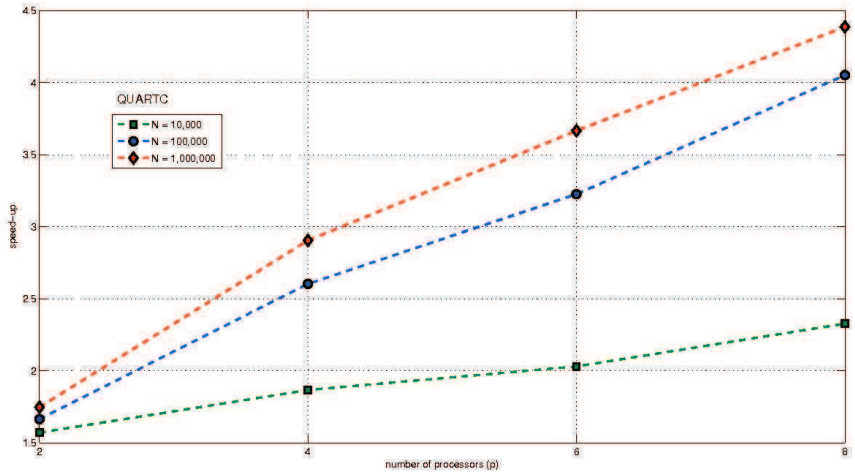
Problem	Dimension	$p = 2$	$p = 4$	$p = 6$	$p = 8$
COSINE	10,000	68.86%	41.67%	29.53%	24.68%
COSINE	100,000	74.98%	53.62%	45.10%	40.71%
COSINE	1,000,000	80.18%	67.81%	51.94%	45.08%
NONCVXUN	10,000	65.16%	36.09%	24.72%	20.40%
NONCVXUN	100,000	69.89%	47.93%	40.77%	36.64%
NONCVXUN	1,000,000	75.11%	63.93%	48.41%	40.98%
QUARTC	10,000	78.61%	46.68%	33.84%	29.09%
QUARTC	100,000	83.25%	65.08%	53.75%	50.64%
QUARTC	1,000,000	87.39%	72.61%	61.09%	54.82%



(a) COSINE

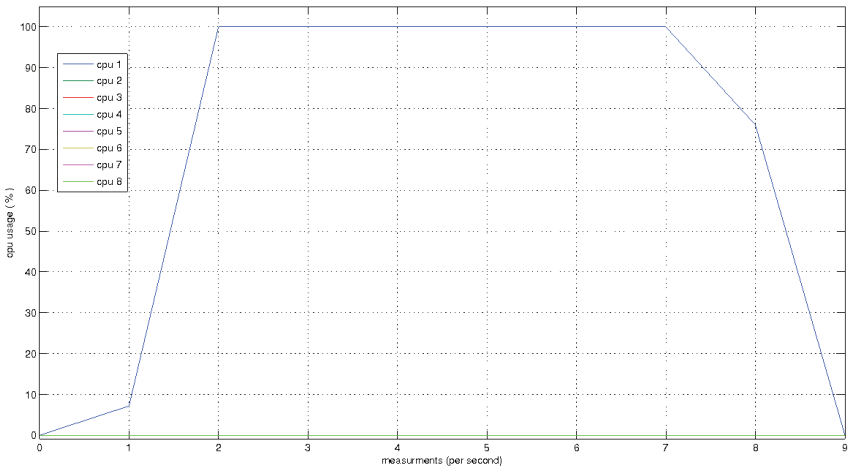


(b) NONCVXUN

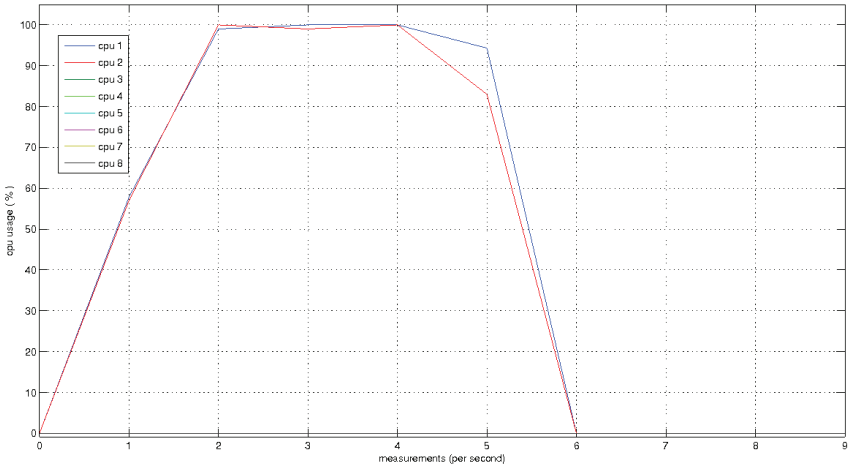


(c) QUARTC

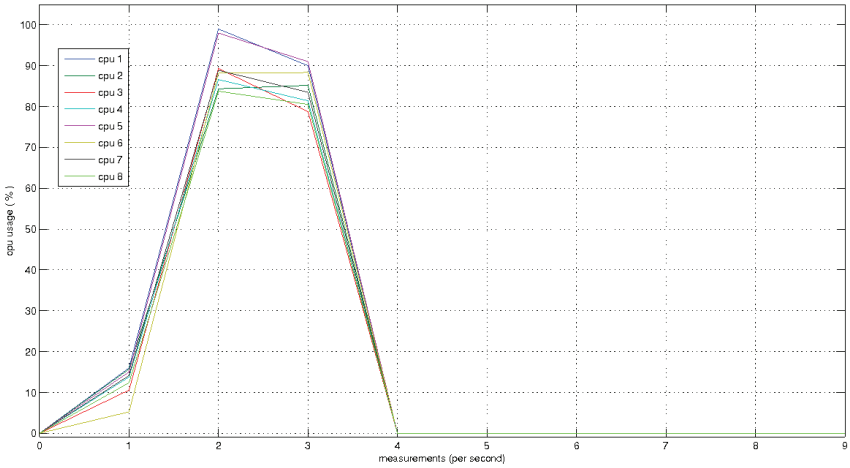
Figure 4.3: Plots of speed-up values as the problem sizes increase



(a) $p = 1$



(b) $p = 2$



(c) $p = 8$

Figure 4.4: CPU usage (per second) during the solution processes with 1,2, and 8 threads

4.2 A Parallel Algorithm for Constrained Optimization

In this section, we propose a new (parallel) algorithm for solving the general nonlinear constrained optimization problem given by

$$\begin{aligned} &\text{minimize} && f(x), \\ &\text{subject to} && c_i(x) \geq 0, && i \in \mathcal{C}, \end{aligned} \tag{P}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is the objective function and $c_i : \mathbb{R}^n \to \mathbb{R}$, $i \in \mathcal{C}$ are the constraint functions. To represent the constraints in a compact form, we also define the vector-valued function $c(x) = (c_1(x), \cdots, c_m(x))^\intercal$, where m is the cardinality of set $\mathcal{C}$. We assume that the objective function is bounded on the feasible domain. Moreover, we allow our algorithm to work with (locally) infeasible problems. Consequently, an acceptable output from our algorithm for problem (P) is either a feasible optimal solution or a proof that the problem may not have any feasible solutions. We measure the *overall violation* at point $x \in \mathbb{R}^n$ as the l_∞ norm of the individual constraint violations. That is, we define

$$v(x) := ||\max\{-c(x), 0\}||_\infty,$$

where the max operator is applied component-wise. Using this notation, we also define the *feasibility problem* as

$$\begin{aligned} &\text{minimize} && v(x), \\ &\text{subject to} && x \in \mathbb{R}^n. \end{aligned} \tag{F}$$

Our main objective in the subsequent parts of this section is to design an *inherently parallel* algorithm, which shows a promising performance when compared against the state-of-the-art solution methods.

4.2.1 Algorithm

Basic ideas. The basic idea of the proposed algorithm is based on the straightforward decomposition of a constrained optimization problem: Problem (P) is defined by several real-valued functions f and c_i, $i \in \mathcal{C}$. The desired solution point for this problem should lie at the intersection of the level sets of a group of constraint functions, and it should also be a (local) minimizer of an objective function among all points in (a part of) this intersection set. Therefore, a solution of problem (P) can be obtained by solving a group of interrelated unconstrained optimization problems; the minimization of the objective function and the minimization of the violation of each constraint. Any solution process needs to follow a way that somehow takes into account all these *component optimization problems*.

Our approach to design such an algorithm is based on the basic idea above. In other words, we try to combine a group of steps each of which is obtained by solving one of the component problems. Clearly, the combination of these steps should take into account the relationships among the components. This approach is illustrated in Figure 4.5, where a constrained optimization problem with two constraints is considered. At an optimal solution x_* of this problem, we have three functions minimized: the objective function f, the violation of the first constraint, denoted by v_1, and the violation of the second constraint, denoted by v_2. The approach of the proposed algorithm is to compute independent steps for each of these component problems denoted by d_f, d_{v_1} and d_{v_2}, respectively. After obtaining these steps, an aggregate step d is computed.

An iteration of the new algorithm consists of four parts. In the first part, at most two linear programming problems are solved to determine an improvement direction involving only the first order information coming from each component function. The LPs in this part are close to the subproblems of a sequential linear programming (SLP) algorithm [74]. Unlike an SLP approach, however, the second order information is collected from all the component functions by solving a set of trust region subproblems, in the a second part. This step yields a set of directions that we use to alter the improvement direction of the first part by solving an additional linear program (LP) in the third part. In the fourth stage, backtracking operations are carried out, if

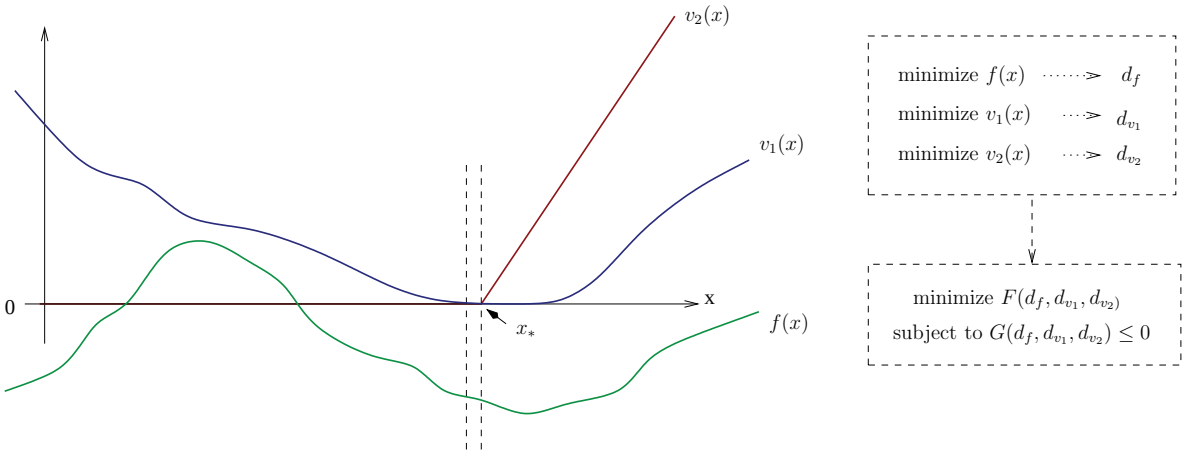


Figure 4.5: Illustration of the basic idea on a single dimensional problem

the progress made by the algorithm is not sufficient. Overall the algorithm requires solving only linear programming and trust region subproblems, which are in general significantly easier than solving general quadratic programming subproblems.

Improvement directions. At a nonoptimal and possibly infeasible iterate x_k, we expect the best available step to follow a direction that improves the current solution with respect to all component problems. Therefore, we start each iteration by checking whether such a direction exists. We define a *mutual improvement direction* (M-direction) as a direction that decreases the value of both the objective function f and the mostly violated constraints. Depending on the current iterate x_k being feasible or not, the procedure for computing a mutual improvement direction d^M may require solving two linear programs.

When is x_k infeasible, we first check whether there exists a step that improves the constraint violations by solving

$$\begin{aligned} \underset{u^F, d^F}{\text{minimize}} \quad & u^F, \\ \text{subject to} \quad & -c_i(x_k) - \nabla c_i(x_k)^\intercal d^F \leq u^F, \quad i \in \mathcal{C}, \\ & u^F \geq 0, \\ & \|d^F\|_\infty \leq M, \end{aligned} \tag{F-direction-LP}$$

where M is a large enough finite number. Its sole purpose is to ensure that $\|d^F\|$ is bounded. Note that by setting $u^F = v(x_k)$ and $d^F = 0$, we obtain an initial feasible solution for this LP.

Let $(\bar{u}^F, \bar{d}^F)$ denote the optimal solution to problem (F-direction-LP). If $\bar{d}^F = 0$, or equivalently, $\bar{u}^F = v(x_k)$, we conclude that x_k is an infeasible stationary point. Otherwise, we solve another LP to search for a mutual improvement direction d^M that also improves the objective function, while providing the same level of decrease in the violation as obtained by $\bar{d}^F$. That is, we solve

$$\begin{aligned} \underset{d^M}{\text{minimize}} \quad & \nabla f(x_k)^\intercal d^M, \\ \text{subject to} \quad & -c_i(x_k) - \nabla c_i(x_k)^\intercal d^M \leq \bar{u}^F, \quad i \in \mathcal{C}, \\ & \|d^M\|_\infty \leq M. \end{aligned} \tag{M-direction-LP}$$

Note that the feasibility of problem (M-direction-LP) is ensured, since $d^M = 0$ is a trivial feasible solution Clearly, if the current iterate x_k is feasible, that is $v(x_k) = 0$, then there is no need to solve (F-direction-LP). In that case we directly solve (M-direction-LP) by setting $\bar{u}^F = 0$.

Let $\bar{d}^M$ denote the optimal solution to problem (M-direction-LP). If $\nabla f(x_k)^\intercal \bar{d}^M < 0$, then we have a mutual improvement direction. Once such a direction is obtained, we continue with computing a step s^A by solving problem (A-step-LP). On other hand, when the optimal solution satisfies $\nabla f(x_k)^\intercal \bar{d}^M \geq 0$, we set $\bar{d}^{FI} = \bar{d}^M$ and call $\bar{d}^{FI}$ as a *feasibility improvement direction* (F-direction). Then, we dedicate our efforts to improving the feasibility and solve (F-step-LP). Both (A-step-LP) and (F-step-LP) problems shall be explained later. If the optimal solution turns out to be $\bar{d}^M = 0$ when $v(x_k) = 0$, then the algorithm terminates since there is no direction improving f without violating some of the constraints. In other words, we obtain that the current solution is a first order stationary point for problem (P).

To this end, we only use the first order information coming from the objective function and the constraints. Therefore, if we revisit our basic idea of looking at the overall problem as a composition of several interrelated problems, then the mutual im-

provement direction can be considered as an *aggregate gradient* for the overall problem. Next, we collect the second order information for each component function hoping that we can improve upon the progress that can be obtained by the mutual or feasibility improvement directions.

Base steps. To provide some curvature related information to our potential steps, we will use a set of guide vectors that we call the base steps (B-steps). Each B-step is computed by solving a trust region subproblem including one of the component function. That is, if we denote the objective function or one of the constraint functions by $h \in \mathcal{C} \cup \{f\}$, then we solve

$$\min \left\{ \frac{1}{2}(s_h^B)^\intercal H_h^k s_f^B + \nabla h(x_k)^\intercal s_h^B \ : \ \|s_h^B\| \leq \Delta_h \right\}, \qquad \text{(B-step-TR(h))}$$

where H_h^k, $h \in \mathcal{C} \cup \{f\}$ denotes the Hessian or an approximation to it. We denote the optimal solution to this problem by $\bar{s}_h^B$.

It is important to note that every function h has its own trust region parameter, which reflects its characteristics and scale around the current iterate. In a sense, this provides a kind of preconditioning by defining new coordinates with a corresponding multidimensional trust region. However, it is not hard to show that not every $\bar{d}^M$ is in the convex cone generated by the B-steps, and they may not necessarily span a nonempty cone even when some M-directions exist. Thus, if we require all steps taken by the algorithm to be in the cone spanned by B-steps, it may fail in some cases. Our idea is to use them as guides and encourage our steps to follow them as closely as possible. This shall be our main objective in the third stage below.

Before we continue with the remaining stages, let us define the index sets for violated, active and satisfied constraints by

$$\mathcal{V}_k = \{i \in \mathcal{C} : c_i(x_k) < 0\}, \mathcal{A}_k = \{i \in \mathcal{C} : c_i(x_k) = 0\} \text{ and } \mathcal{S}_k = \{i \in \mathcal{C} : c_i(x_k) > 0\},$$

respectively.

Aggregate and feasibility steps. Once we obtain a mutual improvement direction, $\bar{d}^M$, we may look for an *aggregate step* (A-step), s^A that benefits from the second order information coming the functions f, and c_i, $i \in \mathcal{V} \cup \mathcal{A}$. Likewise, if we obtain a feasibility improvement direction, $\bar{d}^{FI}$ then we focus on improving the overall violation. Thus, we search for a *feasibility step* (F-step), s^F using the second information only from c_i, $i \in \mathcal{V} \cup \mathcal{A}$. As it is common for some unconstrained optimization methods, we also try to rotate our aggregate gradient according to the second-order information. Following the same analogy, we shall require these guiding steps be *gradient-related*.

We start with finding an A-step. The set of gradient-related directions is given by

$$\mathcal{G}_k^A = \{i \in \mathcal{V}_k \cup \mathcal{A}_k \cup \{f\} : (\bar{d}^M)^\intercal \bar{s}_i^B \geq 0\}.$$

We then set up the following LP to find a mutual improvement direction that benefits from the second order information:

$$\begin{aligned}
\underset{\pi_M, \pi, s^A}{\text{maximize}} \quad & \sum_{i \in \mathcal{G}_k^A \cap \{\mathcal{V}_k \cup \{f\}\}} w_i \pi_i, \\
\text{subject to} \quad & s^A = \pi_M \bar{d}^M + \sum_{i \in \mathcal{G}_k^A} \pi_i \bar{s}_i^B, \\
& \pi_M + \sum_{i \in \mathcal{G}_k^A} \pi_i \leq 1, \qquad \text{(A-step-LP)} \\
& \nabla f(x_k)^\intercal s^A \leq \tau \nabla f(x_k)^\intercal \bar{d}^M, \\
& -c_i(x_k) - \nabla c_i(x_k)^\intercal s^A \leq v(x_k) - \tau(v(x_k) - \bar{u}^F), \quad i \in \mathcal{C}, \\
& \pi_M, \pi_i \geq 0, i \in \mathcal{G}_k^A,
\end{aligned}$$

where w_i, $i \in \mathcal{G}_k^A \cap \{\mathcal{V}_k \cup \{f\}\}$ are the weights assigned to each direction according to their success in terms of the improvement they achieve with respect to the merit function (see Section 4.2.2 for details) and $\tau \in (0, 1)$ is a parameter that indicates the reduction we require s^A to achieve in linear models of the component problems as a percentage of the reduction provided by $\bar{d}^M$. The optimal solution is given by $\bar{\pi}_M$, $\bar{\pi}_i$, $i \in i \in \mathcal{G}_k^A$ and $\bar{s}_A$. Problem (A-step-LP) defines the step vector s^A as a nonnegative combination of

$\bar{d}^M$ and the gradient-related B-steps. Since the aim is to select $\bar{s}^A$ as closely as possible to the B-steps, the objective function tries to increase the corresponding coefficients. Note that the right-hand-side values $\bar{u}^F$ and $\nabla f(x_k)^\intercal \bar{d}^M$ come from the solutions of the previous LPs, (F-direction-LP) and (M-direction-LP).

Problem (A-step-LP) can be seen as a kind of multidimensional line search on our B-steps. Clearly, a multiplier π_i, $i \in \mathcal{G}_k^A$ can be zero if the corresponding B-step causes a deterioration in the other objectives. Note that the steps are bounded by proper trust region limits in all B-step dimensions, as well as the linear models of the currently satisfied constraints. Moreover, (A-step-LP) always has the trivial feasible solution $s_A = \bar{d}^M$, and its objective function is always bounded by the second constraint along with the nonnegativity of variables π_M and π_i, $i \in \mathcal{G}_k^A$. However, it may be the case that $\mathcal{G}_k^A = \emptyset$. In this case, there is no need to solve an LP; we need to follow the *aggregate gradient*. Therefore, in such a case we simply set $\bar{s}^A = \bar{d}^M$.

We next discuss obtaining a F-step. When it is not possible to take an aggregate step, but there is a nonzero feasibility improvement direction given by $\bar{d}^{FI}$, we concentrate on the feasibility components of our problem and look for a step that improves the overall violation. Following similar motivations as for the aggregate step, we attempt to improve the feasibility improvement direction by using the second order information. The set of gradient-related directions in this case becomes

$$\mathcal{G}_k^F = \{i \in \mathcal{V}_k \cup \mathcal{A}_k : (\bar{d}^{FI})^\intercal \bar{s}_i^B \geq 0\}.$$

The optimal feasibility step, $\bar{s}^F$ is then obtained by solving

$$\begin{aligned}
\underset{\pi_{FI},\pi,s^F}{\text{maximize}} \quad & \sum_{i\in\mathcal{G}_k^F\cap\mathcal{V}_k} w_i\pi_i, \\
\text{subject to} \quad & s^F = \pi_{FI}\bar{d}^{FI} + \sum_{i\in\mathcal{G}_k^F} \pi_i \bar{s}_i^B, \\
& \pi_{FI} + \sum_{i\in\bar{\mathcal{G}}} \pi_i \leq 1, \qquad \text{(F-step-LP)} \\
& \nabla f(x_k)^\intercal s^F \leq \eta\mu_k^{-1}\tau(v(x_k) - \bar{u}^F), \\
& -c_i(x_k) - \nabla c_i(x_k)^\intercal s^F \leq v(x_k) - \tau(v(x_k) - \bar{u}^F), i \in \mathcal{C}, \\
& \pi_{FI}, \pi_i \geq 0, i \in \mathcal{G}_k^F,
\end{aligned}$$

where $\bar{\pi}_{FI}$, $\bar{\pi}$, $i \in \mathcal{G}_k^F$ and $\bar{s}^F$ form the optimal solution. The values $\bar{u}^F$ and $\nabla f(x_k)^\intercal \bar{d}^{FI}$ are obtained from problems (F-direction-LP) and (M-direction-LP) solved for computing $\bar{d}^{FI}$. The parameters w_i, $i \in \mathcal{G}_k^F \cap \mathcal{V}_k$ and τ are defined the same way as before, and we set $\bar{s}^F = \bar{d}^{FI}$ whenever $\mathcal{G}_k^F = \emptyset$. The third constraint is obtained using 4.25, since $\nabla f(x_k)^\intercal \bar{d}^{FI} > 0$ and $(v(x_k) - \bar{u}^F) > 0$. The nonnegative values μ_k and η will be explained in the subsequent parts. Note that problem (F-step-LP), like problem (A-step-LP), is bounded due to the second constraint as well as the nonnegativity of variables π_{FI} and π_i, $i \in \mathcal{G}_k^F$. Again, setting $s^F = \bar{d}^{FI}$ gives us a feasible solution.

The first three stages of the algorithm is summarized in Figure 4.6. As we mentioned in the beginning of this section, this figure also shows that the main computational effort of the proposed algorithm comes from solving at most 4 linear programs and $m + 1$ trust region subproblems.

We remark that since we did not specify the value of M in problems (F-direction-LP) or (M-direction-LP), the scale of $\bar{d}^M$ or $\bar{d}^{FI}$ may pose numerical difficulties in solving problems (A-step-LP) and (F-step-LP). We shall elaborate on this issue in Section 4.2.2.

Multi-component backtrack. At this point, we have a trial iterate $\tilde{x}_{k+1} = x_k + s_k$, where s_k is either equal to $\bar{s}^A$ or $\bar{s}^F$. Next, we have to decide whether to accept or reject

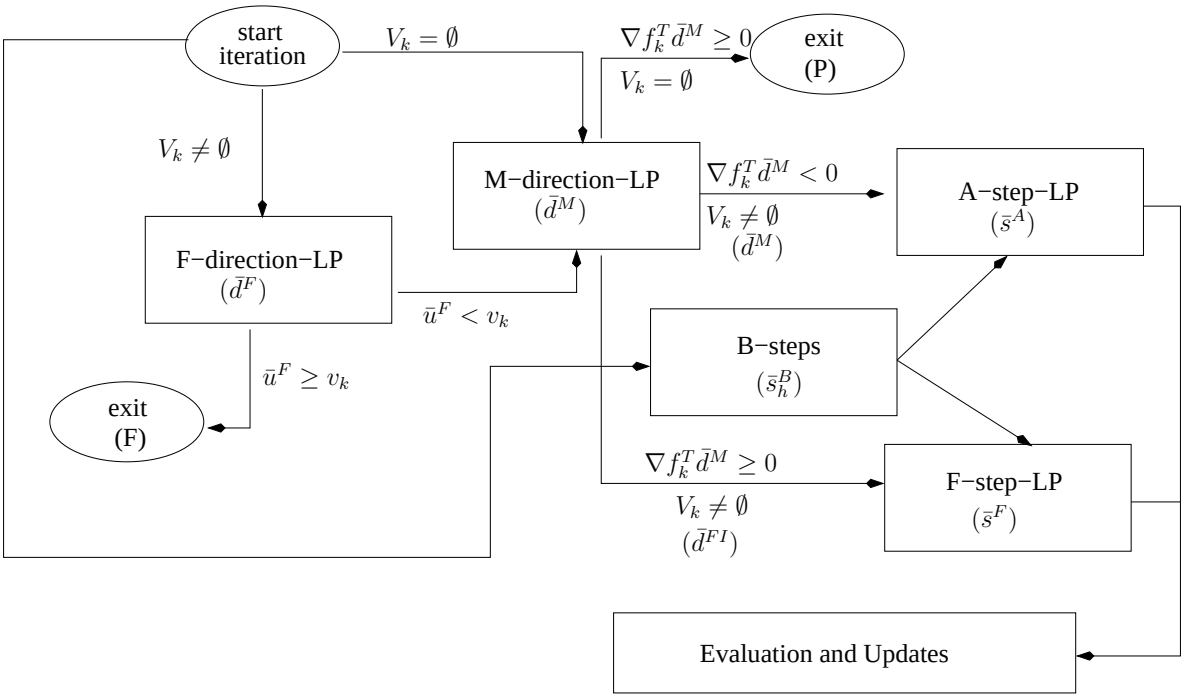


Figure 4.6: Flow of the step computation procedure

the new iterate. Naturally, the difficult part of designing a step acceptance procedure is to decide what to do when the current step is not acceptable.

We label a trial step as acceptable, if it provides an overall improvement for the component problems. Otherwise, we implement a backtracking procedure, where the overall improvement is measured via the merit function

$$\phi(x) := \mu f(x) + v(x) = \mu f(x) + \| \max\{0, -c(x)\} \|_\infty. \tag{4.16}$$

The sufficient decrease at iteration k is then tested using a linear approximation of ϕ given by

$$l_k^\phi(d) := \mu_k l_k^f(d) + l_k^v(d), \tag{4.17}$$

where

$$\begin{aligned} l_k^f(d) &:= f(x_k) + \nabla f(x_k)^\intercal d, \\ l_k^v(d) &:= \max_{i \in \mathcal{C}} \{-c_i(x_k) - \nabla c_i(x_k)^\intercal d, 0\}. \end{aligned} \tag{4.18}$$

$\tilde{x}_{k+1}$ is accepted, if for a given $\xi > 0$, it satisfies

$$\phi(x_k) - \phi(\tilde{x}_{k+1}) \geq \xi(l_k^\phi(0) - l_k^\phi(s_k)). \tag{4.19}$$

This acceptance procedure follows the ideas of existing procedures. In particular, the selection of the penalty parameter μ_k is done following the approach in [9] (see (4.25)).

When step s_k is not acceptable, we propose a *multi-component backtracking procedure* that is based on reducing some of the multipliers, $\pi_i$to obtain a new trial step.

Let s_k^t denote the trial step obtained at the t^{th} backtrack iteration. To obtain convergence, there are two critical requirements on this backtracking operation:

1. The norm of the consecutive trial steps should decrease. That is,

$$\gamma \|s_k^{t+1}\| \leq \|s_k^t\|, \quad \text{for some } \gamma > 1. \tag{4.20}$$

2. The trial steps should satisfy the linear decrease conditions enforced in problems (A-step-LP) or (F-step-LP). To recall, for some $\tau > 0$ we should check either

$$\begin{aligned} &\nabla f(x_k)^\intercal s_k^t \leq \tau \nabla f(x_k)^\intercal \bar{d}^M, \\ &-c_i(x_k) - \nabla c_i(x_k)^\intercal s_k^t \leq v(x_k) - \tau(v(x_k) - \bar{u}^F), \quad i \in \mathcal{C}, \end{aligned} \tag{4.21}$$

or

$$\begin{aligned} &\nabla f(x_k)^\intercal s_k^t \leq \eta \mu_k^{-1} \tau(v(x_k) - \bar{u}^F), \\ &-c_i(x_k) - \nabla c_i(x_k)^\intercal s_k^t \leq v(x_k) - \tau(v(x_k) - \bar{u}^F), \quad i \in \mathcal{C}, \end{aligned} \tag{4.22}$$

respectively.

Under these two requirements, different backtracking procedures can be designed. A straightforward choice is to re-solve all the subproblems with the updated values of the subproblem parameters M and Δ_h. A faster approach that we also use in our convergence proof is to construct a decreasing sequence of parameter values, $\{\tau^t\}_{t\geq 1}$. The above requirements are always satisfied for some $s_k^t \neq 0$ when we set $\tau = \tau^t$. Then, the step s_k^t can be obtained by setting $\tilde{d}^M = \tau^t \bar{d}^M$ and resolving problem (A-step-LP) or problem (F-step-LP) only once. We elaborate on this backtracking approach in

Section 4.2.3.

4.2.2 Implementation

In this section, we first give the steps of the proposed algorithm. Then, we explain how one can apply scaling and determine the direction weights. Our discussion on the implementation details concludes with an approach to update the trust-region radii used in the B-step subproblems.

Step 1 Algorithm 8 gives the details of the first step of the algorithm. We first check the existence of an F-direction. If the optimal solution of this subproblem indicates stationarity for the violation measure, we terminate the algorithm. Otherwise, the M-direction subproblem is solved. If a mutual improvement direction is available, an A-step is computed; otherwise, we move on to computing an F-step.

Algorithm 8: Aggregate Gradient

```
if v(x) = 0 then
    Solve M-direction-LP
    if ∇f(x_k)^T d̄^M ≥ 0 then
        exit:optimal
else
    Solve F-direction-LP
    if ū^F ≥ v(x_k) then
        exit:infeasible
    Solve M-direction-LP
    if ∇f(x_k)^T d̄^M < 0 then
        step = A
    else
        d̄^FI = d̄^M
        step = F
```

Step 2 A summarized in Algorithm 9, in the second step, we first collect the second order information from each component function. Then, we compute an A-step or an F-step.

Algorithm 9: Step Computation

for $i \in \mathcal{A}_k \cup \mathcal{V}_k$ **do**
  Solve `B-step-TR(`$c_i(x_k)$`)`
if *step = A* **then**
  Solve `B-step-TR(`$f(x_k)$`)`
  Solve `A-step-LP`
  $s_k = \bar{s}^A$
if *step = F* **then**
  Solve `F-step-LP`
  $s_k = \bar{s}^F$
$\tilde{x}_{k+1} = x_k + s_k$

Step 3 In this step, the trial point $\tilde{x}_{k+1}$ computed in Step 2 is accepted, if it provides a sufficient improvement in the merit function ϕ. The parameter μ_k, which has the role of ensuring that the steps computed by the algorithm decrease $l_k^\phi(0)$, is updated only when an F-step is computed. The details of this step is summarized in Algorithm 10.

Algorithm 10: Evaluation

if *step = F* & $\nabla f(x_k)^\intercal \bar{d}^{FI} > 0$ **then**
  $\mu_{k+1} = \min\{\mu_k, \eta \frac{v(x_k) - \bar{u}^F}{\nabla f(x_k)^\intercal \bar{d}^{FI}}\}$
else
  $\mu_{k+1} = \mu_k$
if $\phi(x_k) - \phi(\tilde{x}_{k+1}) < \xi_1(l^\phi(0) - l^\phi(\tilde{x}_{k+1} - x_k))$ **then**
  `M-Backtrack`
else
  $x_{k+1} = \tilde{x}_{k+1}$.

Step 4 In the last step, we do the parameter updates as given in Algorithm 11. The update procedure is adapted from the usual trust region approach: We compute the success ratio for each model function, by dividing the actual reduction (*ared*) to the predicted reduction by the model function (*pred*), and update the trust region radii accordingly. Since the linear models of all constraints affect the length of the first order step, we evaluate their success according to their linear models

($pred_i$ is computed with respect to the linear approximation). Moreover, $pred_i$, $i \in \mathcal{V}_k \cup \mathcal{A}_k \cup f$, is computed with respect to the quadratic approximation if the B-step corresponding to this constraint have contributed to the current step.

Algorithm 11: Updates

```
if step = A then
    ρ_f = ared_f(x̃_{k+1} − x_k) / pred_f(x̃_{k+1} − x_k)
for i ∈ C do
    ρ_i = ared_i(x̃_{k+1} − x_k) / pred_i(x̃_{k+1} − x_k)
for i ∈ C ∪ f do
    if ρ_i > ξ_2 & ||x̃ − x_k|| ≥ Δ_i then
        Increase Δ_i;
    if ρ_i < ξ_3 then
        Decrease Δ_i;
```

Scaling and weighting. Recall that the objective functions of problems (A-step-LP) and (F-step-LP) involve weights for the gradient-related B-steps. A possible way to set these weights is to take into consideration the merit function and the measure of constraint violation. Then, we can set

$$\begin{aligned} w_f &= \mu_k, \\ w_i &= -c_i(x_k)/v(x_k), \qquad \text{for } i \in \mathcal{V}_k. \end{aligned}$$

Another issue that has a potential effect on the efficiency of the A-step computation is the length of the aggregate-gradient $\bar{d}^M$ (or $\bar{d}^{FI}$). One possible approach for scaling of $\bar{d}^M$ is to use the lengths of the so-called Cauchy steps for the component problems. We denote these lengths by $\tilde{\alpha}_i, i \in \mathcal{A}_k \cup \mathcal{V}_k$ and they are given by

$$\tilde{\alpha}_i = \min\{\alpha_i, \frac{\Delta_i}{||\nabla c_i(x_k)||}\}, \quad i \in \mathcal{A}_k \cup \mathcal{V}_k,$$

where

$$\alpha_i = \begin{cases} \dfrac{\nabla c_i(x_k)^\intercal \nabla c_i(x_k)}{\nabla c_i(x_k)^\intercal \nabla^2 c_i(x_k) \nabla c_i(x_k)}, & \text{if } \nabla c_i(x_k)^\intercal \nabla^2 c_i(x_k) \nabla c_i(x_k) > 0; \\ 1, & \text{otherwise.} \end{cases}$$

The steplength α_f corresponding to the objective function is computed in a similar manner. Then, we scale $\bar{d}^M$ by setting the value M in problems (F-direction-LP)and (M-direction-LP) as a weighted average of the sizes of the scaled component gradients (the Cauchy steps). We select the weights proportional to the closeness to $\bar{d}^M$ and measure that closeness using the cosine of the angles, between each component function's gradient. That is, we evaluate

$$\theta_f = \frac{\nabla f(x_k)^\intercal \bar{d}^M}{\|\bar{d}^M\| \|\nabla f(x_k)\|}, \quad \theta_i = \frac{\nabla c_i(x_k)^\intercal \bar{d}^M}{\|\bar{d}^M\| \|\nabla c_i(x_k)\|}, \; i \in \mathcal{A}_k \cup \mathcal{V}_k.$$

Then, we normalize these weights whenever the total value of the cosines is greater than 1. Thus, the scale of the aggregate gradient is computed by

$$M = \frac{1}{\max\{1, \theta_f + \sum_{i \in \mathcal{A} \cup \mathcal{V}} \theta_i\}} (\theta_f \tilde{\alpha}_f \|\nabla f(x_k)\| + \sum_{i \in \mathcal{A}_k \cup \mathcal{V}_k} \theta_i \tilde{\alpha}_i \|\nabla c_i(x_k)\|).$$

Clearly, another measure of the closeness could also be used in a similar way.

Recall that in the proposed algorithm, it is possible to define a different trust region radius for each component problem. If these radii are updated according to their contributions to the progress, then we somehow obtain a scaling effect implicitly. This leads us to the following discussion.

Multi-component trust region. The *multi-component trust region* consists of the collection of those component problem trust-regions. When a trial iterate is computed, the trust region radii of component problems are all updated, with respect to either their linear or quadratic models, depending on their role in the step computation.

An important observation about the multi-component trust region approach is that it provides an implicit quadratic constraint for the length of s_k. That is, since s^k is

a convex combination of a group of base steps, problems (A-step-LP) and (F-step-LP) provide a trust region constraint on its norm

$$\|s_k\| = \|\pi_f \bar{s}_f^B + \sum_{i \in \mathcal{A}_k \cup \mathcal{V}_k} \pi_i \bar{s}_i^B\| \leq \pi_f \Delta_f + \sum_{i \in \mathcal{A}_k \cup \mathcal{V}_k} \pi_i \Delta_i.$$

4.2.3 Convergence

We start our convergence analysis by stating the KKT conditions for problems (F) and (P). Note that if problem (F) is written in the form of a constrained optimization problem

$$\begin{aligned} \text{minimize} \quad & u, \\ \text{subject to} \quad & c(x) + u \geq 0, \\ & u \geq 0, \end{aligned}$$

then its constraints always satisfy MFCQ. Then, $x_* \in \mathbb{R}^n$ is a *stationary point for problem (F)*, if there exists $\lambda^c \in \mathbb{R}^n_+$, $\lambda^u \in \mathbb{R}_+$ satisfying

$$\begin{gathered} \textstyle\sum_{i \in \mathcal{C}} \lambda_i^c \nabla c_i(x_*) = 0, \\ \textstyle\sum_{i \in \mathcal{C}} \lambda_i^c + \lambda^u = 1, \\ (c_i(x_*) + u^F)\lambda_i^c = 0, \quad i \in \mathcal{C}, \\ u^F \lambda^u = 0, \\ u^F \geq 0, \\ c_i(x_*) + u^F \geq 0, \quad i \in \mathcal{C}. \end{gathered} \tag{4.23}$$

Suppose that the constraints of (P) satisfy MFCQ at $x_* \in \mathbb{R}^n$. Then x_* is a *stationary for problem* (P), if there exists $\lambda^c \in \mathbb{R}^n_+$ satisfying

$$\begin{gathered} -\nabla f(x_*) + \textstyle\sum_{i \in \mathcal{C}} \lambda_i^c \nabla c_i(x_*) = 0, \\ c_i(x_*)\lambda_i^c = 0, \quad i \in \mathcal{C}, \\ c_i(x_*) \geq 0, \quad i \in \mathcal{C}. \end{gathered} \tag{4.24}$$

In our subsequent discussion, we call $x_k \in \mathbb{R}^n$ as a *termination point* if one of the following statements hold:

I. The stationarity conditions (4.23) for problem (F) are satisfied at x_k and $v(x_k) > 0$.

II. $v(x_k) = 0$ and x_k satisfies the stationarity conditions (4.24) for problem (P).

III. $v(x_k) = 0$ and MFCQ are violated at x_k

The assumptions that we shall use in our analysis are as follows:

A1. The functions f, c_i, and their gradients ∇f, ∇c_i are Lipschitz continuous, for all $i \in \mathcal{C}$.

A2. f is bounded below on the feasible domain.

Our first result below shows that all linear programming problems used in the proposed algorithm are well-defined.

LEMMA 4.2.1 *At each iteration k, the linear programming problems (F-direction-LP), (M-direction-LP), (A-step-LP), and (F-step-LP) are feasible and bounded.*

PROOF. As already indicated when describing these subproblems, the solutions $d^F = 0$, $u^F = v(x_k)$, $d^M = 0$, $s^A = \bar{d}^M$, and $s^F = \bar{d}^{FI}$ are always feasible solutions for the corresponding subproblems; also, the objective functions of these problems have the lower bounds $\bar{u}^F \geq 0$, $\nabla^f(x_k)^\intercal \bar{d}^M \geq -\|\nabla f(x_k)\| M\sqrt{2}$, $\sum_{i\in\mathcal{G}^A} w_i\pi_i \geq 0$, $\sum_{i\in\mathcal{G}^F} w_i\pi_i \geq 0$, respectively. Therefore, all linear programming subproblems are always feasible and bounded, and finite optimal solutions to these problems always exist. □

LEMMA 4.2.2 *Consider any $x_* \in \mathbb{R}^n$.*

I. Conditions (4.23) *are satisfied at x_* if and only if $\bar{d}^F = 0$ is an optimal solution to (F-direction-LP) defined at x_*.*

II. Conditions (4.24) *are satisfied at a feasible point x_* where MFCQ qualifications are satisfied if and only if $\bar{d}^M = 0$ is an optimal solution to (M-direction-LP) at x_*.*

PROOF.

I. Consider the optimality conditions for (F-direction-LP) at x_*. $\exists \lambda^c, \lambda^M, \bar{\lambda}^M \in \mathbb{R}^n_+, \lambda^u \in \mathbb{R}_+$such that:

$$\begin{gathered}
\textstyle\sum_{i\in\mathcal{C}} \lambda_i^c \nabla c_i(x_*) + \lambda^M - \bar{\lambda}^M = 0, \\
\textstyle\sum_{i\in\mathcal{C}} \lambda_i^c + \lambda^u = 1, \\
(c_i(x_*) + \nabla c_i(x_*)^\intercal \bar{d}^F + u^F)\lambda_i^c = 0,\ i \in \mathcal{C}, \\
u^F \lambda^u = 0, \\
(\bar{d}^F + Me)^\intercal \lambda^M = 0, \\
(-\bar{d}^F + Me)^\intercal \bar{\lambda}^M = 0, \\
c_i(x_*) + \nabla c_i(x_*)^\intercal \bar{d}^F + u^F \geq 0,\ i \in \mathcal{C}, \\
u^F \geq 0, \\
\bar{d}^F \leq Me, \\
-\bar{d}^F \leq Me.
\end{gathered}$$

Here, e denotes the vector $(1, \ldots, 1)^\intercal$. These conditions are equivalent to the conditions given by (4.23) for $\bar{d}^F = 0$; therefore the stationarity of $\bar{d}^F = 0$ for (F-direction-LP) and the stationarity of x_* for the feasibility problem (F) imply eachother.

II. Consider the optimality conditions for (M-direction-LP) at x_*.

$\exists \lambda^c, \lambda^M, \bar{\lambda}^M \in \mathbb{R}^n_+, \lambda^u \in \mathbb{R}_+$ such that:

$$\begin{gathered}
-\nabla f(x_*) + \textstyle\sum_{i\in\mathcal{C}} \lambda_i^c \nabla c_i(x_*) + \lambda^M - \bar{\lambda}^M = 0, \\
(c_i(x_*) + \nabla c_i(x_*)^\intercal \bar{d}^M + v(x_*))\lambda_i^c = 0,\ i \in \mathcal{C}, \\
(\bar{d}^M + Me)^\intercal \lambda^M = 0, \\
(-\bar{d}^M + Me)^\intercal \bar{\lambda}^M = 0, \\
c_i(x_*) + \nabla c_i(x_*)^\intercal \bar{d}^M + v(x_*) \geq 0,\ i \in \mathcal{C}, \\
\bar{d}^M \leq Me, \\
-\bar{d}^M \leq Me.
\end{gathered}$$

When $v(x_*) = 0$ and $\bar{d}^M = 0$, the above conditions are equivalent to the conditions given in (4.24); this proves the desired result.

□

COROLLARY 4.2.1 *Suppose $x_k \in \mathbb{R}^n$ is not a termination point. Then, one of the subproblems (F-direction-LP) or (M-direction-LP) has a nonzero optimal solution at x_k.*

The result shown by the next corollary is based on observing that the subproblem (F-direction-LP) defined at x_* in fact solves

$$\underset{\|d^F\|_\infty < M}{\text{minimize}} \quad l_*^v(d^F).$$

Likewise, in case of (M-direction-LP) we have

$$\begin{aligned} \underset{\|d^M\|_\infty < M}{\text{minimize}} \quad & l_*^f(d^M), \\ \text{subject to} \quad & l_*^v(d^M) \leq l_*^v(\bar{d}^F). \end{aligned}$$

COROLLARY 4.2.2 *Let $l_*^f(d)$ and $l_*^v(d)$ correspond to the linear models (4.18) at x_*.*

I. Conditions (4.23) are satisfied at x_* if and only if $l_*^v(\bar{d}^F) - l_*^v(0) = 0$.

II. Suppose that MFCQ qualifications are satisfied at x_*. Conditions (4.24) are satisfied at x_* if and only if $l_*^f(\bar{d}^M) - l_*^f(0) = 0$ and $l_*^v(\bar{d}^M) - l_*^v(0) = 0$.

In this sense, the subproblems (M-direction-LP) and (F-direction-LP) evaluate stationarity of the current iterate x_k. The quantities $l_k^v(0) - l_k^v(\bar{d}^{FI})$, $l_k^v(0) - l_k^v(\bar{d}^M)$ with $l_k^f(0) - l_k^f(\bar{d}^M)$ provide stationarity measures for problems (F) and (P).

REMARK 4.2.1 *Suppose that x_k is not a termination point. Then, the improvement directions $\bar{d}^M$ and $\bar{d}^{FI}$ computed at x_k are descent directions for the linear model l_k^ϕ of the penalty function ϕ in the following sense:*

I. if $v(x_k) = 0$,

$$l_k^\phi(0) - l_k^\phi(\bar{d}^M) \geq \mu_k(l_k^f(0) - l_k^f(\bar{d}^M)) > 0,$$

for any $\mu_k > 0$,

II. if $v(x_k) > 0$ and k is an A-step iteration,

$$l_k^\phi(0) - l_k^\phi(\bar{d}^M) \geq \mu_k(l_k^f(0) - l_k^f(\bar{d}^M)) + (l_k^v(0) - l_k^v(\bar{d}^M)) > 0,$$

for any $\mu_k > 0$,

III. if $v(x_k) > 0$ and k is an F-step iteration,

$$l_k^\phi(0) - l_k^\phi(\bar{d}^M) \geq (1-\eta)(l_k^v(0) - l_k^v(\bar{d}^{FI})) > 0,$$

provided that μ_k is set to the largest value satisfying

$$\mu_k \nabla f(x_k)^\intercal \bar{d}^{FI} \leq \eta \left(v(x_k) - \bar{u}^F\right) \text{ for some small } \eta \in (0,1). \tag{4.25}$$

Given that the steps s_k computed by the algorithm satisfies conditions (4.21) or (4.22), this remark implies

$$l_k^\phi(0) - l_k^\phi(s_k) > 0.$$

Next, we prove the existence of s_k satisfying (4.21) or (4.22).

LEMMA 4.2.3 *At inner iteration t of iteration k, there exists nonzero trial steps $s_k^t \in \mathbb{R}^n$ satisfying conditions* (4.20) *as well as* (4.21) *or* (4.22) *for $\tau = \beta^t$, $\beta \in (0,1)$, and for fixed values of improvement directions $\bar{d}^M$ or $\bar{d}^{FI}$.*

PROOF. Suppose $t = 2$ and k is an A-step. Consider the trial step $s_k^2 = \beta \bar{d}^M$. Since the piecewise linear model l_k^v is convex,

$$l_k^v(\beta \bar{d}^M) \leq (1-\beta) l_k^v(0) - \beta l_k^v(\bar{d}^M),$$

yielding

$$l_k^v(0) - l_k^v(\beta \bar{d}^M) \geq \beta(l_k^v(0) - l_k^v(\bar{d}^M)).$$

Also, as for the linear model l_k^f, $\beta\bar{d}^M$ satisfies

$$l_k^f(0) - l_k^f(\beta\bar{d}^M) = -\beta\nabla f(x_k)^\intercal \bar{d}^M = \beta(l_k^f(0) - l_k^f(\bar{d}^M)).$$

Now, replacing $\beta\bar{d}^M$ above with $\beta^{t-1}\bar{d}^M$, and repeating the same arguments prove that the conditions (4.21),

$$\begin{aligned} l_k^v(0) - l_k^v(s_k^t) &\geq \beta^{t-1}(l_k^v(0) - l_k^v(\bar{d}^M)), \\ l_k^f(0) - l_k^f(s_k^t) &\geq \beta^{t-1}(l_k^f(0) - l_k^f(\bar{d}^M)), \end{aligned}$$

are satisfied for $s_k^t = \beta^{t-1}\bar{d}^M$. Also, since $\beta \in (0,1)$, this choice of s_k^t satisfies condition (4.20), i.e., $\|s_k^t\| \leq \beta\|s_k^{t-1}\|$.

Similarly, $s_k^t = \beta^{t-1}\bar{d}^{FI}$ satisfies (4.20) and (4.22) for $\tau = \beta^t$ when k is an F-step iteration. □

Now, we need to show that the trial step computation procedure will end in finite number of iterations so that $\|s_k^t\|$ stays bounded away from zero at a nonstationary point x_k.

REMARK 4.2.2 *Note that the condition* (4.20) *allows only a linear decrease in* $\|s_k^t\|$. *Therefore, we have*

$$\lim_{t\to\infty} \frac{\beta^{t-1}}{\|s_k^t\|} = c > 0.$$

This relationship is trivially satisfied by the choice of $s_k^t = \beta^{t-1}\bar{d}^M$ *above. In fact, it is not hard to show that a faster decrease in* $\|s_k^t\|$ *would not allow the second set of conditions in* (4.21) *or* (4.22) *hold since both* $l_k^f(0) - l_k^f(s_k^t)$ *and* $l_k^v(0) - l_k^v(s_k^t)$ *are* $O(\|s_k^t\|)$.

LEMMA 4.2.4 *At each iteration* k, *unless* x_k *is a termination point, there exist nonzero steps* s_k *(A-step or F-step) satisfying the acceptance criterion*

$$\phi(x_k + s_k) - \phi(x_k) \leq \xi(l_k^\phi(s_k) - l_k^\phi(0)) \quad \textit{for some } \xi > 0.$$

PROOF. If x_k is not a termination point, then there exist nonzero directions

$\bar{d}^M$ or $\bar{d}^{FI}$ by Lemma 4.2.2. Therefore at least one nonzero step s_k is available since $s^A = \bar{d}^M$ and $s^F = \bar{d}^{FI}$ are always feasible solutions to subproblems (A-step-LP) and (F-step-LP), respectively.

Consider a series of trial steps $s_k^0, s_k^1, \ldots, s_k^t$ at iteration k, satisfying conditions (4.20) and (4.21). As we proved in Lemma 4.2.3, such a series of nonzero steps exists.

Suppose that the desired criterion is never satisfied as $t \to \infty$. This indicates

$$\phi(x_k + s_k^t) - \phi(x_k) > \xi(l_k^\phi(s_k^t) - l_k^\phi(0)), \text{ for all } t.$$

Equivalently,

$$\begin{aligned} \phi(x_k + s_k^t) - \phi(x_k) - (l_k^\phi(s_k^t) - l_k^\phi(0)) &> (\xi - 1)(l_k^\phi(s_k^t) - l_k^\phi(0)), \\ \Rightarrow \qquad \phi(x_k + s_k^t) - l_k^\phi(s_k^t) &> (\xi - 1)(l_k^\phi(s_k^t) - l_k^\phi(0)). \end{aligned}$$

This yields for the left-hand-side

$$\begin{aligned} \phi(x_k + s_k^t) - l_k^\phi(s_k^t) = \; & \mu_k f(x_k + s_k^t) + \max_{i \in \mathcal{C}}\{-c_i(x_k + s_k^t), 0\} \\ & -(\mu_k f(x_k) + \mu_k \nabla f(x_k)^\intercal s_k^t + \max_{i \in \mathcal{C}}\{-c_i(x_k) - \nabla c_i(x_k)^\intercal s_k^t, 0\}) \\ \\ \leq \; & \mu_k \tfrac{1}{2} L_f \|s_k^t\|^2 + \max_{i \in \mathcal{C}}\{-c_i(x_k + s_k^t) + c_i(x_k) + \nabla c_i(x_k)^\intercal s_k^t, 0\} \\ \leq \; & \mu_k \tfrac{1}{2} L_f \|s_k^t\|^2 + \tfrac{1}{2} L_c \|s_k^t\|^2 \end{aligned}$$

for some constants $L_f > 0$ and $L_c > 0$, since f and c_i are twice Lipschitz continuous so that

$$\begin{aligned} f(x_k + s_k^t) &\leq f(x_k) + \nabla f(x_k)^\intercal s_k^t + \tfrac{1}{2} L_f \|s_k^t\|^2, \\ c_i(x_k + s_k^t) &\leq c_i(x_k) + \nabla c_i(x_k)^\intercal s_k^t + \tfrac{1}{2} L_c \|s_k^t\|^2, \; i \in \mathcal{C} \end{aligned}$$

holds. Therefore, we have

$$\mu_k \frac{1}{2} L_f \|s_k^t\|^2 + \frac{1}{2} L_c \|s_k^t\|^2 \geq \phi(x_k + s_k^t) - l_k^\phi(s_k^t) > (1 - \xi)(l_k^\phi(0) - l_k^\phi(s_k^t)).$$

On the other hand, condition (4.22) implies when τ is set to β^{t+1} as in Lemma 4.2.3

$$l_k^\phi(0) - l_k^\phi(s_k^t) \geq \beta^{t-1}(1-\eta)(l_k^v(0) - l_k^v(\bar{d}^{FI}))$$

for F-steps, and

$$l_k^\phi(0) - l_k^\phi(s_k^t) \geq \beta^{t-1}(\mu_k(l_k^f(0) - l_k^f(\bar{d}^M)) + l_k^v(0) - l_k^v(\bar{d}^M))$$

for A-steps. Since the values $(1-\eta)(l_k^v(0) - l_k^v(\bar{d}^{FI}))$ and $(\mu_k(l_k^f(0) - l_k^f(\bar{d}^M)) + l_k^v(0) - l_k^v(\bar{d}^M))$ stay constant and are always positive, we can write

$$\mu_k \frac{1}{2} L_f \|s_k^t\|^2 + \frac{1}{2} L_c \|s_k^t\|^2 > (1-\xi)(l_k^\phi(0) - l_k^\phi(s_k^t)) \geq (1-\xi)\beta^{t-1}\Delta_l,$$

where $\Delta_l > 0$ is a constant value. Then, dividing both sides with $\|s_k^t\|$ and taking the limit as $k \to \infty$ gives

$$\lim_{k\to\infty} \frac{\mu_k \frac{1}{2} L_f \|s_k^t\|^2 + \frac{1}{2} L_c \|s_k^t\|^2}{\|s_k^t\|} > \lim_{k\to\infty} (1-\xi)\Delta_l \frac{\beta^{t-1}}{\|s_k^t\|}$$

holds for all t. However, this is not possible since $\|s_k^t\| \to 0$ as $t \to \infty$ but the right hand side of the above expression approaches to a positive value (see Remark 4.2.2) whereas the left hand side gives zero. This shows the desired result. □

Global convergence. We will now give the global convergence result using the above shown properties of the algorithm.

THEOREM 4.2.1 *Suppose $\{x_k\}$ is the sequence of the iterates computed and accepted by the algorithm. Suppose that $\{x_k\}, k \in S$, is a subsequence of $\{x_k\}$ such that $\lim_{k\in S} x_k = x_*$. Then, x_* is a stationary point of problem (F). If x_* is feasible and MFCQ holds, then stationarity conditions for problem (P) are satisfied at this point.*

PROOF. Since the function $\phi(x) = \mu f(x) + v(x)$ is continuous, we have $\lim_{k\in S} \phi(x_k) = \phi(x_*)$. As a consequence,

$$\lim_{k\in S}(\phi(x_k) - \phi(x_k + s_k)) = 0 \quad \Longrightarrow \quad \lim_{k\in S}(l_k^{\phi}(s_k) - l_k^{\phi}(0)) = 0, \tag{4.26}$$

by our acceptance rule.

We first show that x_* is a stationary point for problem (F). Let us consider the case where $\{x_k\}, k \in S$ is obtained by infinitely many F-steps. In this case, (4.26) implies

$$\lim_{k\in S}(l_k^v(0) - l_k^v(s_k)) = 0 \quad \Longrightarrow \quad \lim_{k\in S}(l_k^v(0) - l_k^v(\bar{d}^{FI})) = 0,$$

since $l_k^v(0) - l_k^v(s_k)$ is always positive and the condition (4.25) is satisfied by F-steps. So, x_* is stationary for (F) (see Corollary 4.2.2).

Moreover, since

$$\mu_k(l_k^f(\bar{d}^{FI}) - l_k^f(0)) \leq \eta(l_k^v(0) - l_k^v(\bar{d}^{FI})) \leq \frac{\eta}{\tau}(l_k^v(0) - l_k^v(s_k)),$$

we have

$$\lim_{k\in S} \mu_k(l_k^f(\bar{d}^{FI}) - l_k^f(0)) = 0.$$

Then, in this case, at least one of $\mu_k \to 0$, or $l_k^f(\bar{d}^{FI}) - l_k^f(0) \to 0$ holds, $k \in S$. In the latter case, if $v(x_*) = 0$ and MFCQ holds, then the limit point x_* is stationary for (P) by Corollary 4.2.2. On the other hand, since μ_k is set to the largest value satisfying condition (4.25), it should stay bounded away from zero when $\lim_{k\in S}(l_k^f(\bar{d}^{FI}) - l_k^f(0)) = 0$. So, if $\mu_k \to 0$, then

$$\lim_{k\in S}(l_k^f(\bar{d}^{FI}) - l_k^f(0)) = c_1 > 0$$

should hold for some $c_1 \in \mathbb{R}_+$. This indicates $\|\bar{d}^{FI}\| \geq c_2 > 0$ always satisfied for a small enough positive value c_2, while $\lim_{k\in S}(l_k^v(\bar{d}^{FI}) - l_k^v(0)) = 0$. But this is possible only if the linear models of $c_i,\ i \in \mathcal{C}$ are linearly dependent, i.e., MFCQ cannot hold at x_*.

Similarly, if $\{x_k\}, k \in S$ is produced by infinitely many A-steps, (4.26) directly implies

$$\lim_{k\in S}(l_k^f(s_k) - l_k^f(0)) = 0 \quad \text{and} \quad \lim_{k\in S}(l_k^v(s_k) - l_k^v(0)) = 0.$$

Therefore, x_* is a stationary point for (F) by the same argument as above.

Moreover, since μ_k cannot decrease during A-steps,

$$\lim_{x\in S}(l_k^f(0) - l_k^f(s_k)) = 0 \quad \Longrightarrow \quad \lim_{x\in S}(l_k^f(0) - l_k^f(\bar{d}^M)) = 0,$$

implies $\nabla f(x_k)^\intercal d^M \to 0$, $k \in S$. So, $d^M = 0$ becomes an optimal solution to the mutual improvement subproblems as $k \to \infty$. If the limit point x_* is feasible and MFCQ holds, it satisfies stationarity conditions (4.24) of problem (P) by Lemma 4.2.2. □

Relationship to SQP steps. Since our algorithm tries to take curvature related steps, a relevant question would be its relationship with the well-known sequential quadratic programming (SQP) algorithm of nonlinear programming [53]. Since SQP can be seen as a realization of the Newton's method, this would also be relevant in terms of evaluating its convergence rate. However, the rapid convergence of the SQP algorithm is related to the iterations after it identifies the correct active set of the problem. Therefore, the important question then becomes: Can we show any relationship between s^A of our algorithm and a step s^x computed by solving a quadratic programming subproblem? Let us give a remark before we continue.

REMARK 4.2.3 *Let $S = span\{\nabla c_i : i = 0, \ldots, p\}$, where c_0 denotes the objective function. Also define the matrix $R_{p\times n}$ whose $(i+1)^{st}$ column is given by $s_i^\intercal \sum_j (\nabla^2 c_j)$, $i, j = 0, \ldots, p, j \neq i$, where s_i are the Newton steps for the individual problems. That is, $\nabla^2 c_i s_i = -\nabla c_i$. Let the vector (s^x, s^λ) denote the EQP step. Then, we can write $s^x = \sum \pi_i s_i$, provided that the multipliers π satisfy $\pi^\intercal R^\intercal u = 0$, for all $u \in S^\perp := \mathbb{R}^n - S$.*

Now, we can state the question formally: Suppose all the necessary assumptions for the local convergence of Newton steps are satisfied. Then, does the optimal solution vector π_k^A of (A-step-LP) satisfy the condition given in Remark 4.2.3 for all iterations

$k > k_M$ when the active set at iterations $k > k_M$ is the correct active set at x_*?

This question can be partly answered for certain special cases. Take for instance the case when the Hessian matrices $\nabla^2 c_i$ are all diagonal. However, we conjecture that it may not hold for the general case. Our first numerical results with the proposed method support this view.

4.2.4 Practical Performance

An implementation of fully parallel version of the described algorithm can be quite involved. This is true especially when one aims at implementing parallel algorithms for solving trust region and linear programming subproblems. One can also consider warm-up strategies, since the algorithm may solve similar linear programming problems from one iteration to the next. In the same vein, it is not necessary to solve the trust region subproblems to optimality, where an approximate solution may be sufficient for fast convergence. In this section, however, we do not dwell on these implementation details but devote our efforts to give some examples for showing the performance of the proposed algorithm.

Parallel implementation properties. The proposed algorithm involves a large number of inherently parallel tasks. The assignment of tasks to processors can be done dynamically and at the level of basic operations. However, for ease of exposition, we shall explain the algorithm's parallel structure by assigning a group of constraints to each processing unit. Then, all evaluations and computations related to those constraints shall be carrried out by that processing unit. Moreover, we shall also discuss the workload distribution among the parallel processors at a higher level as depicted in Figure 4.7. In this figure, we describe the high-level parallelization profile of the new constrained algorithm. In this figure, the expressions $O(LP(n, m))$ and $O(TR(n))$ stand for the complexity of solving an LP with n variables and m constraints and the cost of solving an n dimensional unconstrained trust-region problem, respectively. To keep the figure simple, the cases where the algorithm starts from a feasible point (therefore skips F-direction-LP), and the case where (F-step-LP) problem is solved instead

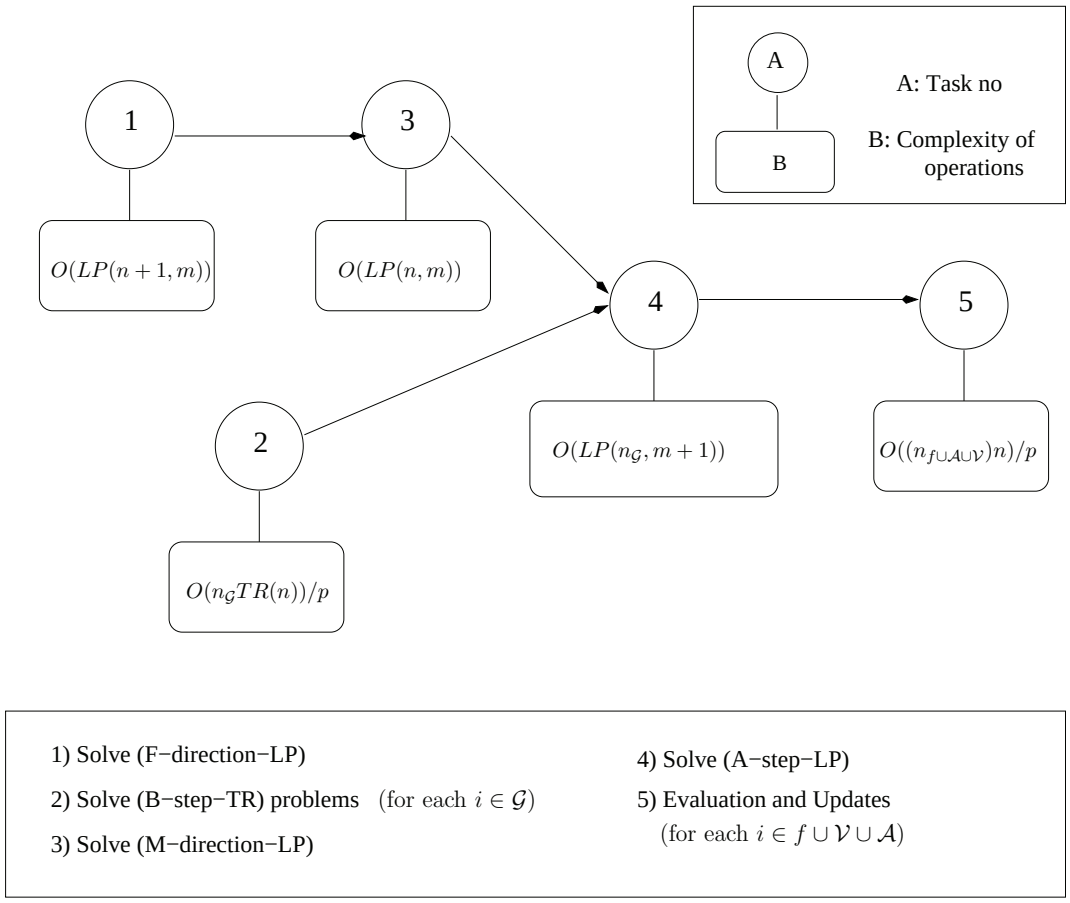


Figure 4.7: Parallelization of the algorithm at the level of its tasks

of (A-step-LP) is not included, since these two cases have almost equivalent computational requirements. Tasks 1, 3 and 4 require solution of linear programs. Tasks 2 and 5 consist of independent operations about the *component problems* as we call them. Therefore, they can be distributed according to the constraint-processor assignment mentioned above.

A relevant question here is the parallel implementation of each high-level task given in Figure 4.7; in particular, parallel solution of the linear programs included in Tasks 1, 3, and 4. Since we are able to provide a good initial feasible solution for each linear program, it is reasonable to consider implementation of a parallel Simplex method. However, as we have mentioned in Chapter 2, parallel Simplex implementations are known to be efficient only for problems with some special structures, e.g., when the coefficient matrix is dense. This suggests that the parallel implementation of

the proposed algorithm can be expected to perform better in certain cases, e.g., when the Jacobian of the problem is dense. Even so, we should also note at this point that it would be possible to design procedures for finding approximate solutions to (A-step-LP) and (F-step-LP) subproblems, i.e. Task 4, with better parallelization properties because the convergence of the algorithm does not require these two linear programs to be solved exactly.

Use of quadratic information. To see how the quadratic information provided by (B-step-TR) and (A-step-LP) subproblems help, we illustrate all the components computed by the algorithm on a sample two-dimensional problem.

The sample problem has four constraints. In Figure 4.8(a), the starting point is marked with a black circle on the contour plot of the problem. The directions $\bar{d}^F$, $\bar{d}^M$, and the base steps $\bar{s}_f^B$, $\bar{s}_1^B$, $\bar{s}_2^B$, $\bar{s}_3^B$, and $\bar{s}_4^B$ are marked in Figure 4.8(a). In Figure 4.8(b), the resulting A-step, $\bar{s}^A$ is marked again with a black circle, the rotation from $\bar{d}^M$ can be clearly seen in this figure. The algorithm is able to converge to the optimal solution point for this problem after 5 additional iterations.

Observations. In this part, we provide some results on the practical behavior of the new algorithm to give a further insight about its potential. The algorithm is coded in MATLAB. As the LP solver, the active-set algorithm available as an option in `linprog` is used. Feasible starting points are provided to the LP solver. For computing B-steps, the Conjugate Gradients algorithm with Steihaug's strategy is used(CHECK THIS!). Therefore, exact solutions to these subproblems are not necessarily computed. We have compiled a few small-scale inequality constrained problems from the CUTEr collection[31] for our initial tests. In Table 4.2.4, the properties of the test problems are given in the second and third columns.

We compare our results to that of the SQP-solver available in `fmincon` of MATLAB, i.e., by setting its `Algorithm` option to `active-set`. We have two main observations on the numerical performance of this implementation of our new algorithm:

1. The new algorithm can rapidly approach to a local solution before the SQP algo-

rithm starts taking full steps,

2. The new algorithm may unfortunately slow down in the close proximity of a local solution point, particularly, in a region where the SQP algorithm takes full steps.

We illustrate those observations on the example of problem HS100 by starting the algorithms from the default initial point for this problem. In Figures 4.9(a)-4.10(b), the progress of the steps of the new algorithm is compared to that of the SQP algorithm implemented in `fmincon`. The iterations are divided into two groups with respect to the steplengths of the SQP algorithm. The SQP algorithm converges in a total of 13 iterations for HS100, and it always takes full steps in the last five iterations. In Figures 4.9(a)-4.9(b), we plot the progress in approaching to the solution point in the first 9 iterations of both algorithms. Figure 4.9(a) shows the drastic difference between the current objective function value and the objective function value at the final solution point. Likewise, Figure 4.9(b) illustrates decrease in the value of constraint violation for both algorithms.

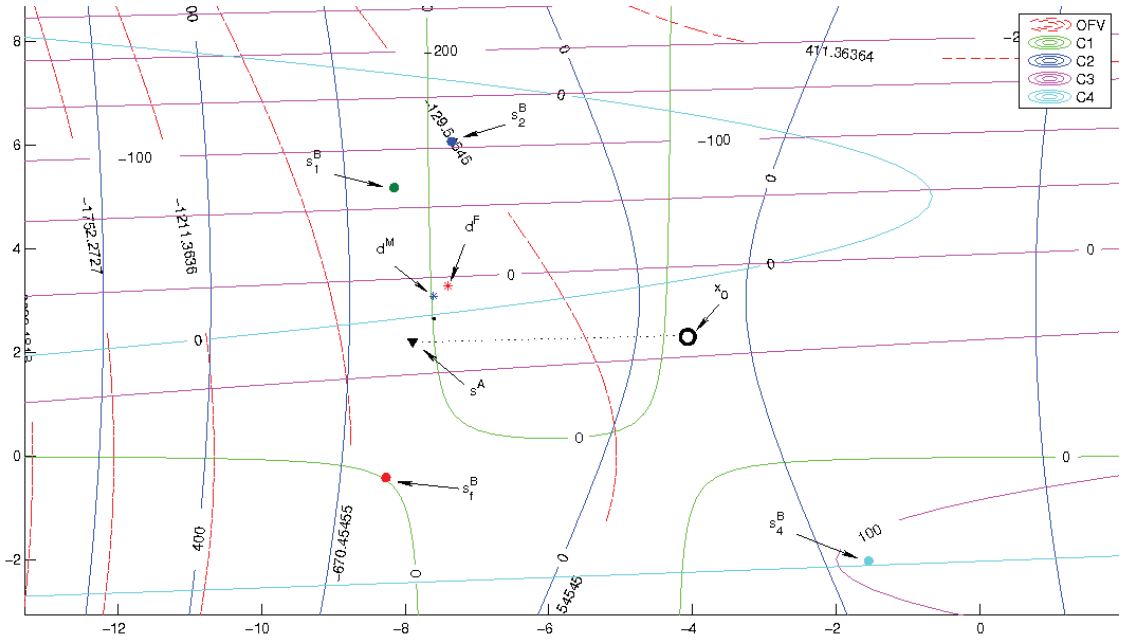
Figures 4.10(a)-4.10(b) show similar plots for the remaining iterations until convergence. These figures illustrate the behavior of the algorithm as we mentioned above: The algorithm is generally successful in getting close to the solution point but its progress may be slower than another algorithm that switches to full steps.

In Table 4.2.4, we give the initial test results using both algorithms for the rest of the problems. The default starting points of these problems generally enable the SQP algorithm to take full steps, as can be seen in the 8^{th} column of Table 4.2.4 (the figures in parentheses), and the new algorithm cannot reveal its strenght in approaching to a local area around a solution point.

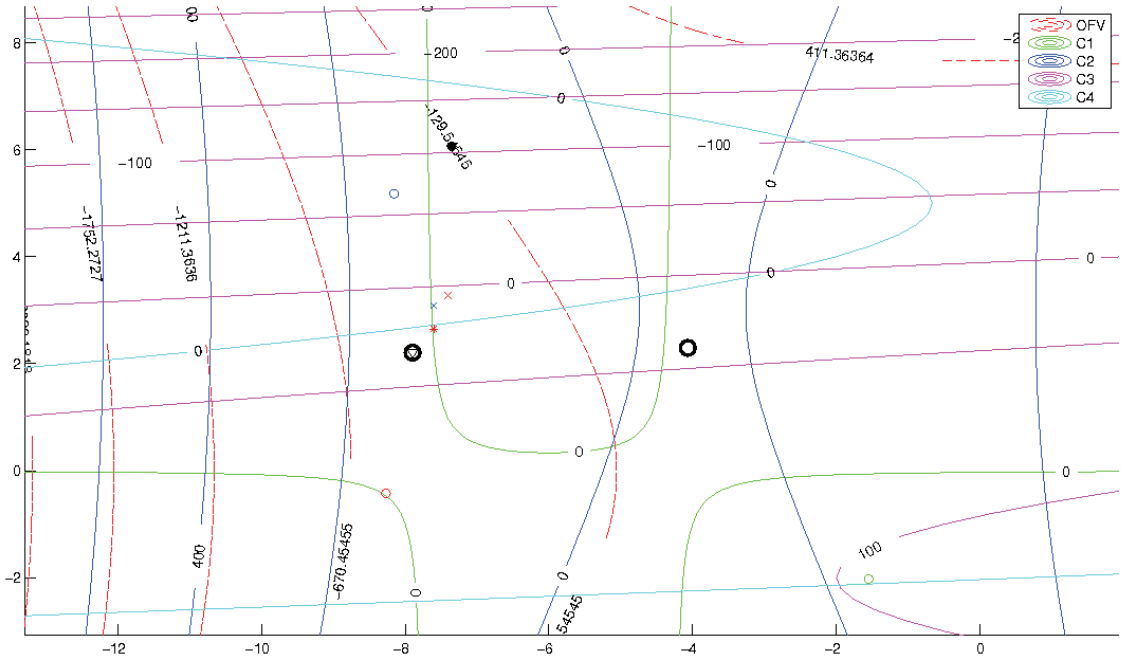
Table 4.4: Number of iterations and function evaluations from the original starting points

			ConCCS				SQP (fmincon)			
Problem	n	m	f	v	Itr.	Eval.	f	v	Itr.*	Eval.
HS100	7	4	680.63	9.77E-07	21	35	680.63	3.16E-06	12(5)	45
HAIFAS	13	9	-0.45	2.46E-05	27	40	-0.45	2.92E-05	6(4)	17
CB2	2	2	1.9522	1.98E-06	25	41	1.9522	4.62E-06	6(6)	13
MISTAKE	9	13	-1	2.69E-06	25	40	-1	2.05E-06	16(13)	36
MAKELA1	3	2	-1.414	2.91E-06	10	15	-1.414	5.45E-06	8(8)	17

* The numbers in parentheses are the number of full steps taken by the SQP algorithm.

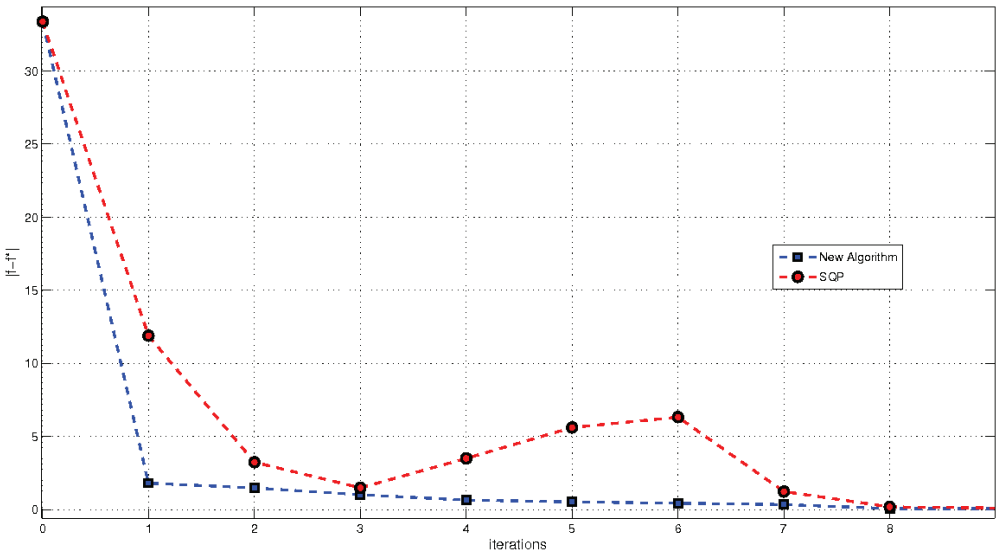


(a) mutual improvement and base steps

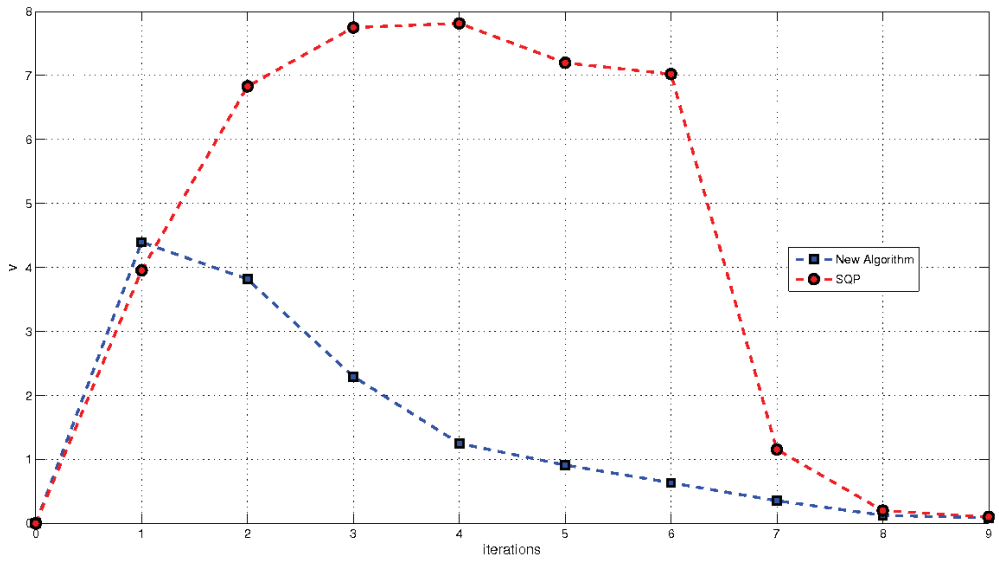


(b) the aggregate step

Figure 4.8: Illustration of the step computation for the new constrained algorithm

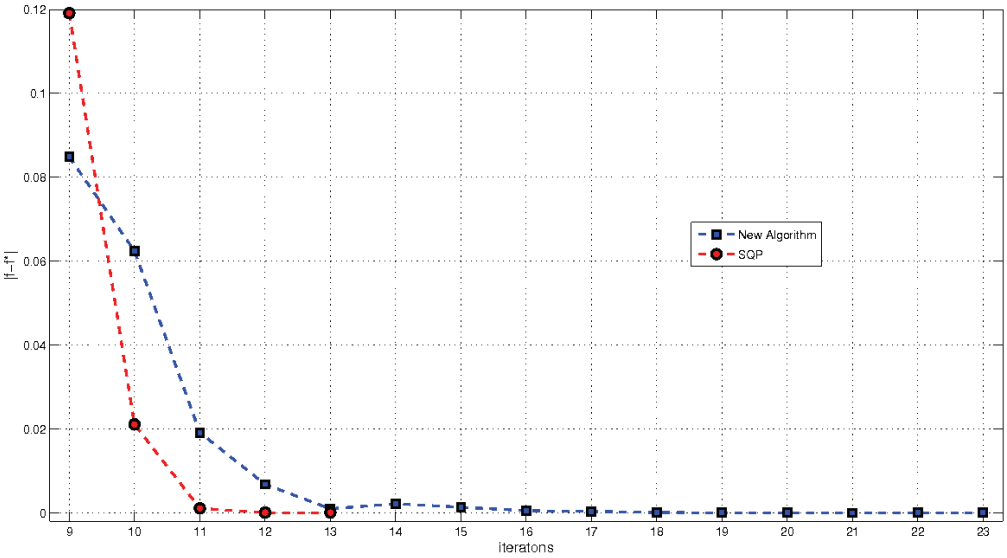


(a) $|f(x_k) - f(x_*)|$

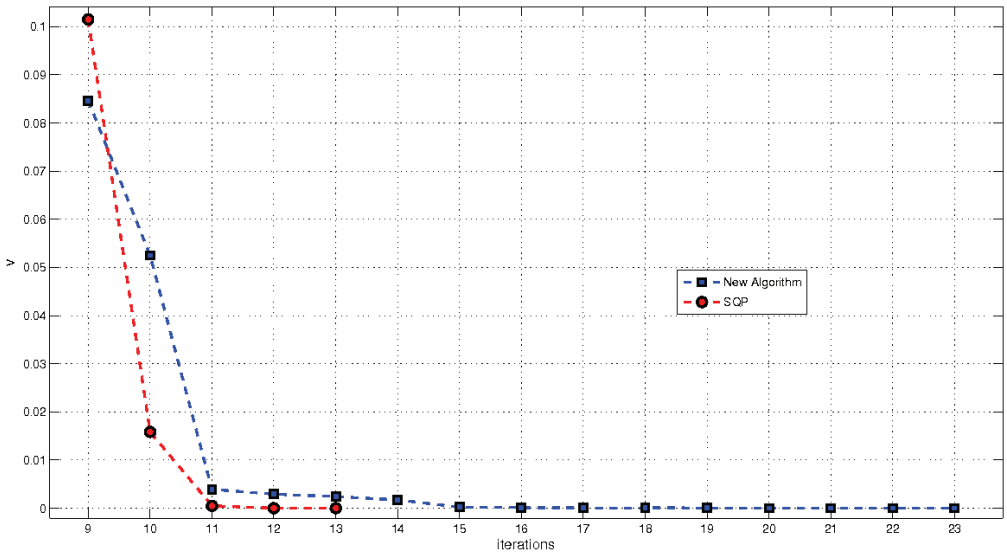


(b) $v(x_k)$

Figure 4.9: Progress provided by the two algorithms – iterations 1-9



(a) $|f(x_k) - f(x_*)|$



(b) $v(x_k)$

Figure 4.10: Progress provided by the two algorithms after iteration 9

Chapter 5

CONCLUSION

In this chapter, we first give some remarks on the findings of this thesis study. Then, we discuss future research directions.

5.1 Concluding Remarks

In this thesis, we presented an approach as a first attempt to investigate the benefits of parallel processing for solving nonlinear programming problems. We tried to reveal the potential of the idea by designing distinct algorithms, which apply the proposed approach at different levels to solve various classes of nonlinear programming problems.

The focus of the algorithms in Chapter 3 was improving the solution performance by using the extra problem information produced in parallel . In our numerical tests, we observed that nice reductions can be achieved in the number of iterations until convergence. Moreover, more robust algorithms could be obtained. For the global optimization extensions, we observed that improvements can be provided in search performance; that is, better solutions can be discovered more quickly thanks to the information exchange. However, we also observed especially in our local applications that the success of a high level application of the idea may fail to provide a real advantage in terms of solution time. In fact, for local optimization problems, the only way to provide scalability of the algorithms of this chapter seems to include further decompositions of their high-level tasks.

In Chapter 4, we considered task parallelism in a lower level and proposed two new

algorithms. The algorithm we designed for unconstrained optimization in Section 3.1 demonstrated that an algorithm following our approach can achieve nice speed-up numbers and an efficient usage of the available parallel resources. We also observed that the extra computations included in this algorithm contributed to the solution performance as expected. The scalability of the algorithm is obtained by decomposing its operations executed in smaller dimensional spaces. The constrained optimization algorithm of the next section (Section 4.2), on the other hand, attempted to decompose the problem into a set of unconstrained optimization problems. The idea there was to use the curvature-related steps by solving (in parallel) a set of unconstrained and linear programing problems. After some preliminary numerical results, we observed that the resulting algorithm is successful in approaching a local solution, but once it becomes close to the solution point it converges slowly as compared to a standard sequential quadratic programming method.

As we have conjectured at the very beginning of this work, we experienced that the *freedom* provided by the availability of parallel computing resources enabled us to end up with new algorithms, that would not be normally considered in a sequential settings. Two particular examples in this sense are the concurrent search algorithms of Section 3.1 and the constrained algorithm of Section 4.2.

In all our implementations, we primarily considered a multicore programming environment. Multicore architectures have recently become the standard processor technology and expected to be even more progressive in the future. So, we believe that we have provided a perspective and some distinct examples on using this technology in solving optimization problems. Our efforts, therefore, may constitute a basis for further contributions to this flourishing and fruitful research area.

5.2 Future Research Directions

Our experience with several different algorithms gave us a very good idea about the opportunities and the limitations of the approach we proposed in this thesis. As a consequence, we have a long list of future research directions.

One important lesson we have learned was the limitations of a high-level paral-

lelization for solving local optimization problems. Thus, we plan to design new algorithms in the concurrent search framework of the first chapter with a further granularity. We believe that a proper selection of included algorithms and a well-written parallel code would provide much better speed-up values than we observed here.

For global optimization problems, on the other hand, there are less concerns in this sense. The MANGO environment described in Section 3.2 is very suitable for developing more sophisticated algorithms than the ones described in this section. In particular, we are very curious about its performance in solving real-life large scale global optimization problems.

A comprehensive future work is certainly the parallel implementation of the constrained algorithm proposed in Section 4.2. This would require to solve a number of implementation issues. As we have already mentioned in that section, it would be possible to apply modifications and improve the local performance of this algorithm. An interesting question we have in mind is to see whether this algorithm becomes really advantageous for problems where the number of constraints are small as compared to the number of variables.

We already observed the success of the parallel program in Section 4.1. We believe that it can be further improved by conducting more tests, and learning more about its behavior. It would be interesting to observe the performance of this algorithm (or its variants) on a distributed architecture in solving a very large-scale optimization problem.

It is possible, and hopefully encouraged with this thesis, to design completely different algorithms following our general approach. This is a challenging but also a very rewarding endeavor.

Our experience provided us a number of *side ideas* as well as *different* algorithms. For example, the framework we proposed in Chapter 3 also gave us a perspective for constructing an alternative practical objective in solving a (nonconvex) nonlinear programming problem. A nonlinear programming solver would normally terminate when it finds the first available solution of a given problem. However, another objective for an nonlinear programming solver would be to find the best solution it can find within

a given time limit. We believe it would be interesting to see whether this perspective has a practical value.

Appendix A

Review on Performance Profiles

Dolan and More [23] propose the performance profiles to evaluate and compare the performances of a set of solvers on a set of test instances.

Suppose that we have a set of algorithms denoted by S and a set of instances denoted by I whose cardinalities are n_s and n_i, respectively. Let the performance measure be the total number of iterations. For each instance $i \in I$ and algorithm $s \in S$, $t_{i,s}$ is defined as the total number of iterations found for instance i by algorithm s. The performance on instance i by algorithm s is compared to the best performance among all algorithms applied on the same problem. To plot the performance profile figures, we follow these steps: First, the performance ratio, given by

$$r_{i,s} = \frac{t_{i,s}}{\min(t_{i,s} : s \in S)},$$

is computed. Then, to assess the overall performance of an algorithm we compute

$$\rho_s(\tau) = \frac{1}{n_i} |\{i \in I : r_{i,s} \leq \tau\}|,$$

where $\tau \in \mathbb{R}$ and $|\cdot|$ denotes the cardinality of a set. The value $\rho_s(\tau)$ shows the probability for algorithm $s \in S$ such that a performance ratio $r_{i,s}$ is within a factor τ of the best possible ratio. The function ρ_s is the (cumulative) distribution function for the performance ratio. Therefore, the value of $\rho_s(1)$ is the probability that the algorithm s dominates the rest of the algorithms.

Bibliography

[1] E. D. Andersen and K. D. Andersen. A parallel interior-point algorithm for linear programming on a shared memory machine. Core discussion papers, Université catholique de Louvain, Center for Operations Research and Econometrics (CORE), 1998.

[2] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley, 2006.

[3] B. M. Averick and J. J. More. Parallel gradient distribution in unconstrained optimization. *SIAM Journal on Optimization*, 4:708–721, 1994.

[4] J. M. Bahi, S. Contassot-Vivier, and R. Couturier. *Parallel Iterative Algorithms*. Chapman & Hall / CRC, 2007.

[5] S. Benson, M. Krishnan, L. Mcinnes, J. Nieplocha, and J. Sarich. Using the GA and TAO toolkits for solving large-scale optimization problems on parallel computers. *ACM Trans. Math. Softw.*, 33, 2007.

[6] M. Bertocchi. A parallel algorithm for global optimization. *Optimization*, 21(3):379–386, 1990.

[7] D. P. Bertsekas. *Nonlinear Programming*. Athena Scientific, 1999.

[8] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov. The impact of multicore on math software. In *Applied Parallel Computing: State of*

the Art in Scientific Computing, volume 4699/2009, pages 1–10. Springer Berlin / Heidelberg, 2009.

[9] R. H. Byrd, N. I. M. Gould, J. Nocedal, and R. A. Waltz. On the convergence of successive linear-quadratic programming algorithms. *SIAM Journal on Optimization*, 16:471–489, 2005.

[10] R. H. Byrd, J. Nocedal, and C. Zhu. Towards a discrete Newton method with memory for large scale optimization. In G. Di Pillo and F. Giannessi, editors, *Nonlinear Optimization and Applications*. Plenum, 1996.

[11] Y. Censor and S. A. Zenios. *Parallel Optimization: Theory, Algorithms, and Applications*. Oxford University Press, 1997.

[12] K. L. Chow. Parallel unconstrained optimization. Depth paper, University of Toronto, Department of Computer Science, 1993.

[13] D. Conforti and R. Musmanno. A parallel asynchronous Newton algorithm for unconstrained optimization. *Journal of Optimization Theory and Applications*, 77(2):305–322, 1993.

[14] A. R. Conn, N. I. M. Gould, and P. L. Toint. *Trust-Region Methods*. SIAM, Philadelphia, PA, USA, 2000.

[15] T. G. Crainic, M. Gendreau, P. Hansen, and N. Miladenovic. Cooperative parallel variable neighborhood search for the p-median. *Journal of Heuristics*, 10:293–314, 2004.

[16] V.-D. Cung, S. L. Martins, C. C. Ribeiro, and C. Roucairol. Strategies for the parallel implementation of metaheuristics. *Essays and Surveys in Metaheuristics*, 2002.

[17] M. D'Apuzzo and M. Marino. Parallel computational issues of an interior point method for solving large bound-constrained quadratic programming problems. *Parallel Computing*, 29(4):467–483, 2003.

[18] M. D'Apuzzo, M. Marino, A. Migdalas, P. M. Pardalos, and G. Toraldo. Parallel computing in global optimization. In E. J. Kontoghiorghes, editor, *Handbook of Parallel Computing and Statistics*. CRC Press, 2005.

[19] M. D'Apuzzo, M. Marino, A. Migdalas, P.M. Pardalos, and G. Toraldo. Nonlinear optimization: A parallel linear algebra standpoint. pages 259–282. Chapman & Hall/CRC, 2006.

[20] J. E. Dennis and V. Torczon. Direct search methods on parallel machines. *SIAM Journal on Optimization*, 1:123–145, 1991.

[21] J. E. Dennis and Z. Wu. Parallel continuous optimization. In J. Dongarra, editor, *Sourcebook of parallel computing*, pages 649–670. Morgan Kauffman, 2003.

[22] L. C. W. Dixon and M. Jha. Parallel algorithms for global optimization. *Journal of Optimization Theory and Applications*, 79(2):385–395, 1993.

[23] E. D. Dolan and J. J. More. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91:201–213, 2002.

[24] M. C. Ferris and O. L. Mangasarian. Parallel constraint distribution. *SIAM Journal on Optimization*, 1:487–500, 1991.

[25] M. C. Ferris and O. L. Mangasarian. Parallel variable distribution. *SIAM Journal on Optimization*, 4:102–126, 1994.

[26] T. L. Freeman. A parallel unconstrained quasi-newton algorithm and its performance on a local memory parallel computer. *Applied Numerical Mathematics*, 7:369–378, 1991.

[27] L. Frimannslund and T. Steihaug. A class of methods combining L-BFGS and truncated Newton. Reports in Informatics 319, University of Bergen, Norway, 2006.

[28] A. Frommer and R. A. Renault. A unified approach to parallel space decomposition methods. *Journal of Computational and Applied Mathematics*, 110(1):205–223, 1999.

[29] M. Fukushima. Parallel variable transformation in unconstrained optimization. *SIAM Journal on Optimization*, 8(4):658–672, 1998.

[30] M. Gendreau and T. G. Crainic. Cooperative parallel tabu search for capacitated network design. *Journal of Heuristics*, 8:601–627, 2002.

[31] N. I. M. Gould, D. Orban, and Toint P. L. CUTEr (and SifDec), a constrained and unconstrained testing environment, revisited. Technical Report TR/PA/01/04, CERFACS, 2004.

[32] N.I.M. Gould, S. Lucidi, M. Roma, and P. L. Toint. A line-search algorithm with memory for unconstrained optimization. Technical Report RAL-TR-98-003, Rutherford Appleton Laboratory, 1998.

[33] A. Griewank and G. Corliss. Issues in parallel automatic differentiation. In C. H. Bischof, editor, *Automatic Differentiation of Algorithms*, pages 100–113. SIAM, 1991.

[34] A. Gupta and V. Kumar. Performance and scalability of preconditioned conjugate gradient methods on parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 6(5):455–469, 1995.

[35] J. A. J. Hall. Towards a practical parallelisation of the simplex method. *Computational Management Science*, 7(2):139–170, 2010.

[36] M. T. Heath, E. Ng, and B. W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33(3):420–460, 1991.

[37] P. D. Hough, T. G. Kolda, and V. J. Torczon. Asynchronous parallel pattern search for nonlinear optimization. *SIAM Journal on Scientific Computing*, 23:134–156, 2001.

[38] G. Karypis, A. Gupta, and V. Kumar. A parallel formulation of interior point algorithms. In *Proceedings of the 1994 conference on Supercomputing*, Supercomputing '94, pages 204–213. IEEE Computer Society Press, 1994.

[39] E. Kaszkurewicz, A. Bhaya, and B. Baran. Parallel asynchronous team algorithms: Convergence and performance analysis. *IEEE Transactions on Parallel and Distributed Systems*, 7(7):677–688, 1996.

[40] K. C. Kiwiel and P. O. Lindberg. Parallel subgradient methods for convex optimization. In D. Butnariu, Y.Censor, and S.Reich, editors, *Inherently parallel algorithms in feasibility and optimization and their applications*, pages 335–344. Elsevier, 2001.

[41] D. Klabjan, E. L. Johnson, and G. L. Nemhauser. A parallel primal–dual simplex algorithm. *Operations Research Letters*, 27:47–55, 2000.

[42] T. G. Kolda. Revisiting asynchronous parallel pattern search for nonlinear optimization. *SIAM Journal on Optimization*, 16(2):563–586, 2005.

[43] T. G. Kolda and V. J. Torczon. On the convergence of asynchronous parallel pattern search. *SIAM Journal on Optimization*, 14:939–964, 2004.

[44] O. L. Mangasarian. Parallel gradient distribution in unconstrained optimization. *SIAM Journal on Control and Optimization*, 33:1993–1145, 1995.

[45] M. Manguoglu, A. H. Sameh, and O. Schenk. PSPIKE: A parallel hybrid sparse linear system solver. In *Euro-Par 2009 Parallel Processing*, volume 5704, pages 797–808. Springer Berlin / Heidelberg, 2009.

[46] Intel Math Kernel Library. `http://www.intel.com/software/products/mkl`.

[47] N. Melab, E.-G. Talbi, and S. Cahon. Paradiseo: A framework for the reusable design of parallel and distributed metaheuristics. *Journal of Heuristics*, 10:357–380, 2004.

[48] A. Migdalas, G. Toraldo, and V. Kumar. Nonlinear optimization and parallel computing. *Parallel Computing*, 29:375–391, 2003.

[49] J. L. Morales and J. Nocedal. Enriched methods for large-scale unconstraines optimization. *Computational Optimization and Applications*, 21:143–154, 2002.

[50] J. J. Moré, B. S. Garbow, and K. E. Hillstrom. Testing unconstrained optimization software. *ACM Transactions on Mathematical Software*, 7(1):17–41, 1981.

[51] A. Neumaier. Complete search in continuous global optimization and constraint satisfaction. *Acta Numerica*, (13):271–369, 2004.

[52] J. Nocedal. Updating quasi-Newton matrices with limited storage. *Mathematics of Computation*, 35:773–782, 1980.

[53] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, 2006.

[54] F. Öztoprak and Ş. İ. Birbil. Implementation of a fixing strategy and parallelization in a recent global optimization method. In *Euro Mini Conference on Continuous Optimization and Knowledge-Based Technologies (EurOPT-2008)*, 2008.

[55] ScaLAPACK Home Page. http://www.netlib.org/scalapack.

[56] P. M. Pardalos, H. E. Romeijn, and H. Tuy. Recent developments and trends in global optimization. *Journal of Computational and Applied Mathematics*, 124(1-2):209–228, 2000.

[57] P.K.-H. Pkua, W. Fan, and Y. Zeng. Parallel algorithms for large-scale nonlinear optimization. *International Transactions in Operations Research*, 5(1):67–77, 1998.

[58] J. Reinders. *Intel Threading Building Blocks*. O'Reilly, 2007.

[59] G. Rudolph. Parallel approaches to stochastic global optimization. In W. Joosen and E. Milgrom, editors, *Parallel Computing: From Theory to Sound Practice, Proceedings of the European Workshop on Parallel Computing*, pages 256–267. IOS Press, 1992.

[60] R. B. Schnabel. Concurrent function evaluations in local and global optimization. *Computer Methods in Applied Mechanics and Engineering*, 64:537–552, 1987.

[61] R. B. Schnabel. A view of the limitations, opportunities, and challenges in parallel nonlinear optimization. *Parallel Computing*, 21:875–905, 1995.

[62] F. Schoen. Stochastic techniques for global optimization: A survey of recent advances. *Journal of Global Optimization*, 1(3):207–228, 1991.

[63] D. S. Siirola, S.Hauan, and A. W. Westerberg. Toward agent-based process systems engineering: Proposed framework and appllication to non-convex optimization. *Computers and Chemical Engineering*, 27:1801–1811, 2003.

[64] T. Steihaug. The conjugate gradient method and trust regions in large scale optimization. *SIAM Journal on Numerical Analysis*, 20:626–637, 1983.

[65] T. A. Straeter. A parallel variable metric optimization algorithm. Technical Report NASA TN D-7329, NASA Technical Note, Langley Research Center, Hampton, VA, USA, 1973.

[66] E.-G. Talbi. A taxonomy of hybrid metaheuristics. *Journal of Heuristics*, 8:541–564, 2002.

[67] S. Talukdar, L. Baerentzen, A. Gove, and P. De Souza. Asynchronous teams: Cooperation schemes for autonomous agents. *Journal of Heuristics*, 4(4):295–321, 1998.

[68] Intel Threading Building Blocks. `http://www.intel.com/software/products/tbb`.

[69] J. N. Tsitsiklis and D. P. Bertsekas. *Parallel and Distributed Computation: Numerical Methods*. Athena Scientific, 1997.

[70] K. Tyner and A. Westerberg. Multiperiod design of azetropic seperation systems i: An agent based approach. *Computers and Chemical Engineering*, 25:1267–1284, 2001.

[71] P. J. M. van Laarhoven. Parallel variable metric algortihms for unconstrained optimization. *Mathematical Programming*, 33:68–81, 1985.

[72] D. J. Wales and J. P. K. Doye. Global optimization by basin-hopping and the lowest energy structures of Lennard-Jones clusters containing up to 110 atoms. *The Journal of Physical Chemistry A*, 101(28):5111–5116, 1997.

[73] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178–194, 2009.

[74] J. Zhang, N.-H. Kim, and L. Lasdon. An improved succesive linear programming algorihm. *Management Science*, 31(10):1312–1331, 1985.

Curriculum Vitae

Education

- 2006 – 2011, Ph.D., Industrial Engineering, Sabancı University
  Dissertation: Parallel Algorithms for Nonlinear Optimization
- 2003 – 2005, M.S., Industrial Engineering, İstanbul Technical University
  Thesis: Multiagent Decision Support: The Application of Street Management
- 1999 – 2003, B.S., Industrial Engineering, İstanbul Technical University
  Graduation Project: Analysing the Performance of Logistics Activities Using Artificial Neural Networks

Experience

- Jan 2010−Jan 2011, Predoctoral Fellow, Northwestern University
- Feb 2006−Jan 2010, Teaching Assistant, Sabancı University
- Jul 2004−Feb 2006, Engineer, Istanbul Metropolitan Municipality
- Feb−Jul 2004, Analyst, AnkaFergana Consulting

Papers

- F.B. Aydemir, A. Günay, F. Öztoprak, Ş.İ. Birbil, P. Yolum, *An Agent-Based Environment for Solving Global Optimization Problems Cooperatively*, submitted.

- F. Öztoprak, Ş.İ. Birbil, *Concurrent Search Algorithms for Unconstrained Optimization*, under revision.

- F. Öztoprak, Ş.İ. Birbil, *A Symmetric Rank-One Quasi-Newton Method Using Negative Curvature Directions*, to appear in Optimization Methods and Software.

- S. Çiftlikli, F. Öztoprak, Ö. Erçetin, K. Bülbül, *Distributed Algorithms for Delay Bounded Minimum Energy Wireless Broadcasting*, International Journal of Interdisciplinary Telecommunications and Networking, Vol.1, No.2, 2009.

Proceedings

- A.Günay, F.Öztoprak, Ş.İ. Birbil, P. Yolum, *Solving Global Optimization Problems using MANGO*, Agent-Based Optimization(ABO 2009), in 3rd International KES Symposium on Agents and Multi-Agent Systems, Technologies and Applications (KES AMSTA 2009), Uppsala, Sweden, 2009.

- F.Öztoprak, Ş.İ. Birbil, *Implementation of a Fixing Strategy and Parallelization in a Recent Global Optimization Method*, Euro Mini Conference on Continuous Optimization and Knowledge-Based Technologies (EurOPT-2008), Neringa, Lithuania, 2008.

- L. Kerçelli, A. Sezer, F. Öztoprak, P. Yolum, Ş.İ. Birbil, *MANGO: A MultiAgent ENvironment for Global Optimization*, 1st International Workshop on Optimization in Multiagent Systems(OPTMAS), in 7th International Joint Conference on Autonomous Agents and Multi-Agent Systems(AAMAS'08), Estoril, Portugal, 2008.

- Ş.Ö. Şahin, F. Ülengin, F. Öztoprak, *Analysing Performance of Supply Chain Management Activities Using Artificial Neural Networks*, 10th World Conference on Transport Research, İstanbul, Turkey, 2004.