The Journal of Supercomputing (2022) 78:17513–17542
https://doi.org/10.1007/s11227-022-04555-8

A survey of HPC algorithms and frameworks for large-scale gradient-based nonlinear optimization

Felix Liu[1,2] · **Albin Fredriksson**[2] · **Stefano Markidis**[1]

Accepted: 20 April 2022 / Published online: 20 May 2022

Abstract

Large-scale numerical optimization problems arise from many fields and have applications in both industrial and academic contexts. Finding solutions to such optimization problems efficiently requires algorithms that are able to leverage the increasing parallelism available in modern computing hardware. In this paper, we review previous work on parallelizing algorithms for nonlinear optimization. To introduce the topic, the paper starts by giving an accessible introduction to nonlinear optimization and high-performance computing. This is followed by a survey of previous work on parallelization and utilization of high-performance computing hardware for nonlinear optimization algorithms. Finally, we present a number of optimization software libraries and how they are able to utilize parallel computing today. This study can serve as an introduction point for researchers interested in nonlinear optimization or high-performance computing, as well as provide ideas and inspiration for future work combining these topics.

Keywords Nonlinear optimization · HPC · Interior-point method · Parallel computing

✉ Felix Liu
felixliu@kth.se

Albin Fredriksson
albin.fredriksson@raysearchlabs.com

Stefano Markidis
markidis@kth.se

[1] CST, KTH Royal Institute of Technology, Stockholm, Sweden

[2] RaySearch Laboratories, Stockholm, Sweden

1 Introduction

Large-scale numerical optimization has natural application in many fields. Examples of application areas include radiation treatment planning [1], optimal power flow problems for power grids [2], optimal control [3], machine learning algorithms such as support vector machines (SVM) [4], finance [5], and many others. Depending on the application, the optimization problems can become very large, having possibly millions of variables and constraints, which can be challenging and time-consuming to solve numerically. Examples of such demanding applications include optimal power flow, where the problem size can be so large that distribution across computational nodes is required for memory reasons [6], and treatment planning for radiation therapy, where computational speed is needed to enable ever more advanced treatment methods [7]. To be able to solve such problems efficiently requires the use of High-Performance Computing (HPC) hardware capable of handling problems requiring a large amount of memory and computing power to solve in a reasonable time.

In this paper, we consider constrained nonlinear optimization problems, where both the objective function and the constraints are nonlinear and possibly non-convex. Solving large scale optimization problems efficiently requires powerful algorithms and hardware. Utilizing HPC hardware efficiently may be key to being able to handle ever larger and more demanding optimization problems. Examples of such hardware include computing clusters, GPUs or FPGAs, where parallel programming is essential to utilize their full potential. This adoption of HPC hardware presents a challenge in devising algorithms and methods for nonlinear optimization that can utilize HPC systems to their fullest while maintaining numerical stability.

This paper aims to give an overview over different algorithms and solvers for nonlinear optimization. A primary focus of this paper will lie in surveying previous research on methods for parallelizing nonlinear optimization solvers and algorithms, as well as research on the use of High-Performance Computing (HPC) architectures, such as GPUs, for nonlinear optimization. We will focus on gradient-based methods for nonlinear optimization, where the gradients of the objectives and constraints are known, as opposed to gradient-free methods, such as evolutionary algorithms, which can be used without knowing the gradients of the problem. One advantage of gradient based methods is that many of them can provide theoretical convergence guarantees, at least locally around optimal solutions. However, for very complex optimization problems, where the gradient of the objective function may not even be known, evolutionary algorithms provide an approach to try to find a solution regardless.

The structure of this paper is as follows: Section 2 gives a background to nonlinear optimization and related algorithms. Section 3 presents an introduction to some basic concepts in parallel computing and HPC to a general audience. Section 4 presents a survey of previous work on parallelization of nonlinear optimization algorithms, broken down into different categories. Section 5 presents a selection of software packages for nonlinear optimization, with a brief presentation of their functionality, and—when applicable—how they are able to utilize parallel computing today. Finally, sect. 7 gives some concluding remarks.

2 Background—nonlinear optimization

A general mathematical optimization problem has the following form:

$$\begin{aligned} \text{min.} \quad & f(x) \\ \text{subject to:} \quad & g_i(x) \leq 0, \quad i \in \{1, ..., m_I\} \\ & h_j(x) = 0, \quad j \in \{1, ..., m_E\} \\ & x \in \mathcal{F}, \end{aligned} \tag{1}$$

where $x \in \mathbb{R}^n$, $f(x), g_i(x), h_j(x) : \mathbb{R}^n \to \mathbb{R}$, $\mathcal{F} \subset \mathbb{R}^n$ and m_I, m_E are the number of inequality and equality constraints, respectively. Note that, other optimization problems can often be converted to this standard form, for instance, a maximization problem can be converted to a minimization problem by changing the sign of the objective function. In the following, we introduce some basic concepts in constrained nonlinear optimization, for a more comprehensive introduction to the theory, we refer the interested reader to the textbook by Nocedal and Wright [8].

Many numerical algorithms for solving nonlinear optimization problems try to find solutions by finding points that satisfy the so-called Karush-Kuhn-Tucker (KKT) conditions for optimality (shown in Eq. (3)). In the following, we will state the KKT conditions, but not give a proof that they indeed are the first-order necessary conditions. For such a proof and more background, the interested is referred to Sun and Yuan [9]. A useful definition in the context of constrained optimization is the *Lagrangian*:

$$\mathcal{L}(x, \lambda, \kappa) = f(x) + \sum_{i=1}^{m_I} \lambda_i g_i(x) - \sum_{j=1}^{m_E} \kappa_j h_j(x), \tag{2}$$

where λ and κ are the so-called *Lagrange multipliers* for the inequality and equality constraints, respectively. The KKT conditions state, under some regularity conditions, that if x^* is a local optimum of (1), then there exists Lagrange multipliers λ_i, $i = 1, ..., m_I$, and κ_j, $j = 1, ...m_E$, such that the following hold:

$$\begin{aligned} \nabla_x \mathcal{L}(x^*, \lambda^*, \kappa^*) &= 0 \\ g_i(x^*) &\leq 0 \quad i \in \{1, ..., m_I\} \\ h_j(x^*) &= 0 \quad j \in \{1, ..., m_E\} \\ \lambda_i^* &\geq 0 \quad i \in \{1, ..., m_I\} \\ \lambda_i^* g_i(x^*) &= 0 \quad i \in \{1, ..., m_I\}. \end{aligned} \tag{3}$$

The conditions above only give the necessary optimality conditions: A point x that satisfies the conditions might not be locally optimal due to being a saddle point. Furthermore, for general optimization problems, the conditions only give information about local optimality: Even if the solution found is a local optimum, it might not be a global optimum. For convex problems, this problem is avoided, since all local optima are also global ones. Despite these limitations, the KKT conditions have

proven very useful in practice when it comes to solving general nonlinear optimization problems.

2.1 Algorithms

In the following section, we will introduce three popular classes of algorithms for solving general constrained nonlinear optimization problems:

- Interior-point methods (IPM)
- Sequential quadratic programming (SQP)
- Augmented Lagrangian methods (ALM).

Our aim here is not to give a detailed explanation of the theoretical background of these algorithms, but rather to introduce the basic ideas and to identify the most critical aspects from a computational perspective.

2.1.1 Interior point methods

The field of interior point methods (IPM) is a rich one, with a vast amount of previous research and ideas. In the following, we introduce key ideas in the so-called primal-dual interior point methods. For a more detailed presentation of the theory and history of interior point methods, the interested reader is referred to the work by Forsgren, Wong and Gill in [10]. For another, more recent, presentation on interior point methods, including theory but also some computational aspects, we refer to the work by Gondzio in [11].

Interior point methods are algorithms for solving nonlinear programs which are based on so-called barrier methods. To simplify the exposition, we will restrict ourselves to considering problems with only inequality constraints, i.e., problems of the form:

$$\begin{aligned} \text{min.} \quad & f(x) \\ \text{s.t.} \quad & g_i(x) \leq 0, \qquad i \in \{1, ..., m_I\} \\ & x \in \mathcal{F}, \end{aligned} \tag{4}$$

since the inequality constraints are what the barrier term in interior point method mainly deal with. Note that, this also modifies the KKT-conditions in Eq. (3) not to include the $h_j(x^*) = 0$ condition. The basic idea of barrier methods is to transform a constrained optimization problem into an unconstrained one by introducing a barrier term in the objective function for the inequality constraints. The barrier function needs to be selected in such a way that its value approaches infinity as the edge of the feasible region is approached, thus discouraging solutions from approaching this boundary. For interior point methods, a logarithmic barrier function of the form $\sum_{i=1}^{m_i} \log(-g_i(x))$ is used, such that the objective function $b(x)$, with the barrier term, becomes

$$b(x) = f(x) - \mu \sum_{i=1}^{n} \log(-g_i(x)), \tag{5}$$

where μ is often called the *barrier parameter*. The optimality conditions for this case are the same as described in (3) except for the final condition which changes from $\lambda_i g_i(x^*) = 0$ to $\lambda_i g_i(x^*) = \mu$ (sometimes referred to as *perturbed complementarity*). Under certain assumptions, when μ is decreased toward zero, the solution points $x(\mu)$ can be shown to converge to a locally optimal solution for the original problem as well. Hence, a practical approach to solve optimization problems using interior point methods is to successively solve the barrier problem while decreasing the value of μ toward zero, stopping when the KKT conditions are satisfied to within some predefined tolerance. A straightforward way of approaching this problem is to use, e.g., Newton's method to find stationary points of the logarithmic barrier objective (5). This approach gives rise to the so-called *primal* interior point methods. However, primal methods suffer from certain drawbacks and in practice the so-called *primal-dual* methods are more popular [10]. Primal-dual interior methods arise when the barrier sub-problems are solved by using Newton's method on the modified KKT optimality conditions, with $\lambda_i g_i(x^*) = \mu$, as opposed to directly on the unconstrained barrier problem.

A rough schematic overview of some important steps in interior-point methods is shown in Fig. 1. Note that, the figure intends to show a basic overview of the steps in an interior-point algorithm and practical implementations may contain additional steps, and exact details may vary between implementations. Computationally, the key steps in many interior-point methods are computing the objective function and constraints, as well as their gradients / Jacobians and Hessians. These values are used then to form the primal-dual systems of equations:

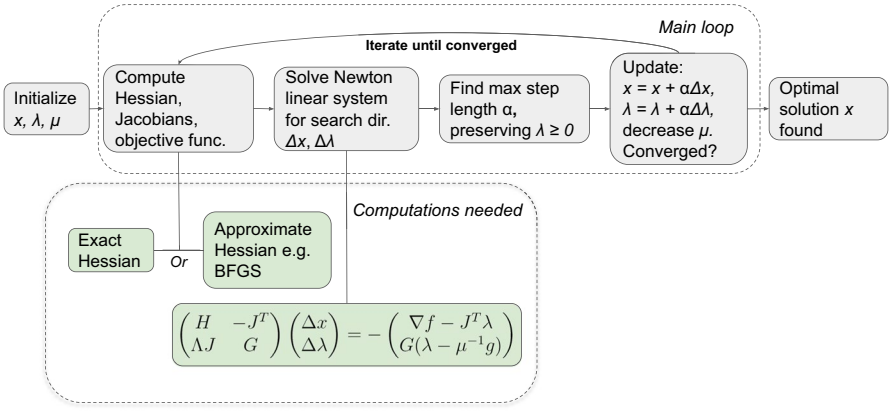


Fig. 1 Illustration of computational steps in primal-dual interior point methods. The gray boxes with arrows show the algorithms execution, with green boxes showing some important computational steps. Note that, this figure is an illustration of the key steps in primal-dual interior point methods and that actual implementations will differ in many details. Notation used in the system of equations is the same as in (6)

$$\begin{pmatrix} H & -J^T \\ \Lambda J & G \end{pmatrix} \begin{pmatrix} \Delta x \\ \Delta \lambda \end{pmatrix} = - \begin{pmatrix} \nabla f - J^T \lambda \\ G(\lambda - \mu^{-1} g) \end{pmatrix}, \tag{6}$$

where H denotes the hessian of the Lagrangian, J the Jacobian of the constraints, and where G and Λ are $diag(g)$ and $diag(\lambda)$, respectively. Note also that dependence on x, λ, μ in many of the variables are omitted for brevity. The primal-dual system (6) can be derived by applying Newton's method to the KKT-conditions with the *perturbed complementarity* constraint ($\lambda_i g_i(x^*) = \mu$). Solving a linear system similar to the one in Eq. (6) is one of the key computational aspects of any interior point method implementation, and we discuss some approaches for solving it efficiently in sect. 4.1.

Linear systems arising from interior point methods are often ill-conditioned and indefinite, making it difficult to use iterative linear solvers. This is one of the reasons many interior point method implementations, e.g., IPOPT, use direct linear solvers based on matrix factorization. This comes with the advantage that one can solve the same linear system for multiple different right-hand sides at relatively little extra cost, which is utilized in predictor-corrector type methods [12, 13].

2.1.2 Sequential quadratic programming

Sequential quadratic programming (SQP) is an algorithm for optimization problems first introduced in 1963 in the PhD thesis of Wilson [14], and later popularized by others in the 1970s. For an in-depth overview of the theory and history of the SQP algorithm, as well as advantages and disadvantages of the method, the interested reader is referred to [15] and [16]. In the following, we give a brief overview of the SQP algorithm and its properties.

In SQP, the optimization problem is solved iteratively by repeated solves of sub-problems—quadratic estimates of the original problem. The sub-problems can be solved using specialized algorithms for quadratic programming. Popular choices for many SQP solvers are active-set methods or interior point methods. Each sub-problem for an optimization problem of the form (1) takes the form

$$\begin{aligned} \min._d \quad & \frac{1}{2} d^T \nabla^2_{xx} \mathcal{L}(x_k, \lambda_k, \kappa_k) d + d^T \nabla f(x_k) \\ \text{s.t.} \quad & d^T \nabla g(x_k) + g(x_k) \geq 0 \\ & d^T \nabla h(x_k) + h(x_k) = 0, \end{aligned} \tag{7}$$

where $\mathcal{L}(x, \lambda, \kappa) = f(x) - \lambda^T g(x) - \kappa^T h(x)$ is the Lagrangian and $d \in \mathbb{R}^n$ is the search direction. The sub-problems (7) consist of a linearization of the constraints and a quadratic approximation of the Lagrangian. A motivating fact in using the QP approximation (7) in each iteration can be seen in the purely equality constrained case, when there are no inequality constraints. In such cases, the solution to the QP sub-problems is equivalent to the solution of the system of equations arising when applying Newton's method on the KKT conditions [17].

The most computationally demanding part of SQP methods for large-scale problems lies in finding solutions to the QP sub-problems. Many SQP algorithms for

large-scale problems use interior point methods to solve the QP sub-problems, this is the case for instance for the WORHP [18] solver used by the European Space Agency. However, in principle, any algorithms for solving quadratic programs would work. This also means that giving a general description of computational aspects of SQP is difficult, since it to a large extent depends on the choice of method for solving the QP sub-problem. We note, however, that many of the computational aspects we discuss regarding interior point methods, in particular on the solution of a Newton linear system of equations to find search directions, are in many cases applicable to SQP too. This can either be due to the SQP solver using an interior-point method to solve the QP-sub-problems, but also due to the fact that applying Newton's method to a system of equations derived from KKT conditions is used in many other QP-algorithms, such as active-set methods (see for example sect. 7 in [19]).

2.2 Augmented Lagrangian methods

Augmented Lagrangian methods (ALM) are similar to barrier methods, such as interior point methods, in that a constrained optimization problem is transformed into an unconstrained one by adding penalty terms to the objective. The augmented Lagrangian method that we discuss here is focused on equality constrained optimization problems (possibly including bound constraints on the variables of the form $l \leq x \leq h$) and requires some additional work to handle inequality constraints. One approach to handle inequality constraints is to introduce *slack variables* s_i, thereby transforming inequality constraints of the form $g_i(x) \leq 0$ to $g_i(x) + s_i = 0, \quad s_i \geq 0$, see [20]. To illustrate the method, consider now the equality constrained problem:

$$\begin{aligned} \text{min.} \quad & f(x) \\ \text{s.t.} \quad & g_i(x) = 0, \qquad i \in \{1, ..., m\}. \end{aligned} \tag{8}$$

The augmented Lagrangian method is based on adding a quadratic penalty to the Lagrangian, with the resulting unconstrained problem becoming:

$$\min_x \quad f(x) - \lambda_i g_i(x) + (\rho/2)\|g(x)\|^2. \tag{9}$$

The solution to the unconstrained problem gives the new value for the primal variables x, and the dual variables λ can be updated using $\lambda^{k+1} = \lambda^k + \rho g(x^k)$, which has the advantage of preserving dual feasibility [21]. Note that, the method can be extended to include bound constraints on the variables of the form $l \leq x \leq h$, where l, h are constant, as well. The bound constraints are then simply propagated from the original problem (8) to the sub-problems, which become bound-constrained.

Solving the sub-problems of the form (9) can in principle be done using any method for unconstrained, or bound-constrained in the presence of such, optimization. Some possible approaches include using projected gradient-based approaches to solve bound-constrained sub-problems, an approach which is implemented in the LANCELOT [22] software library.

2.3 Other algorithms

Although this review focuses on the gradient-based algorithms discussed previously in this section, these are not the only algorithms that exist for nonlinear optimization. One thing the previously discussed algorithms have in common is that they often require gradients (and sometimes also Hessians) of the objective function and constraints to find a solution. In contrast to these methods, there also exists a number of *derivative-free* optimization algorithms, that do not require gradients or Hessians. Examples of such algorithms include evolutionary algorithms [23], including swarm based algorithms such as particle swarm optimization (PSO) [24], probabilistic methods such as simulated annealing [25] and many others. A more comprehensive overview of non gradient-based methods for optimization can be found in the book by Alba [26].

The first major advantage of derivative-free methods is simply the case where the objective function or constraints are so complicated that the derivatives are intractable to compute or simply not known. In such cases, derivative-free methods provide a way to try to find an optimal solution regardless. Secondly, many derivative-free methods lend themselves especially well to parallelization. As an example, swarm-based methods often feature a large number of relatively independent agent searching the feasible region, which lends itself naturally to treating agents in parallel (see the work by Lalwani et al. [27] for a survey of parallel particle swarm optimization).

However, for problems where the derivatives are available and tractable to compute, gradient-based methods often perform better in terms of convergence and number of function evaluations, especially for larger number of variables [28]. Furthermore, gradients are often used in the underlying theory, for instance in the KKT-conditions of local optimality, which are often used in practice to determine convergence more rigorously. In this paper, we focus on gradient-based optimization algorithms, which may be the more attractive choice for problems where the gradients can be computed.

3 High-performance computing overview

While a specific definition for high-performance computing (HPC) is difficult to give, the term generally refers to the use of powerful, typically parallel computing architectures to solve difficult large-scale problems. Traditionally, these problems include everything from simulations in computational fluid dynamics [29], to studying interactions between proteins at the molecular level. In more recent years, the advent of deep neural network has presented another area where immense computing power is required to train such networks on large datasets [30].

In HPC, a distinction is often made between two different types of parallelism (see for example chapter 2.8 in Sterling et al. [31])

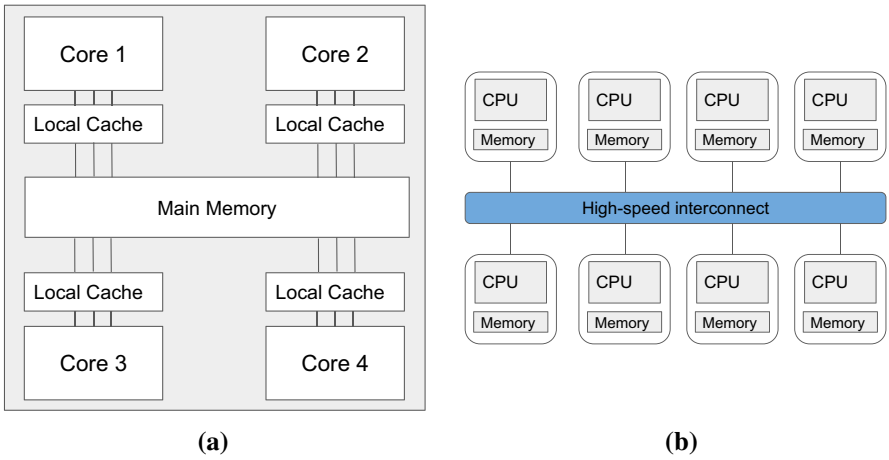


Fig. 2 Conceptual illustration of differences between shared and distributed-memory parallel machines. On the left is an illustration of a multi-core CPU, with small local caches for each core, but a large shared memory that all cores can access directly. The right figure illustrates a distributed memory system, where multiple CPUs, each with their own local memory, are connected through a high-speed network. Passing data between the CPUs need to be done explicitly, e.g., through message passing

1. Shared memory parallelism—where the parallelization is on a set of processors that share a common segment of memory.
2. Distributed memory parallelism—where different computing nodes perform calculations in parallel using their own local memory, but can communicate with each other over a network.

An illustration of how these different types of parallelism can be seen in practice is shown in Fig. 2. Fig. 2a shows a conceptual illustration of a shared-memory multi-core CPU, where each core has direct access to the main memory, which is shared between all cores. Fig. 2b illustrates a distributed computing cluster, where each computational node has its own local memory and can only communicate with other nodes through a high-speed interconnect.

Most modern CPUs today are capable of shared memory parallelism, having multiple computing cores having shared access to main memory and some levels in the cache hierarchy. Making full utilization of such hardware requires dividing the workload to be executed in parallel. Programming the machine to execute in parallel can be done through multiple approaches, for instance through the use of software libraries, such as pthread, or compiler directives through a system such as OpenMP [32], which is widely used in HPC.

Distributed memory parallelism is the basis of high-performance supercomputers. These generally consist of multiple independent computing nodes connected through a high-speed network. One of the most common ways to program such a machine to execute in parallel is using the Message Passing Interface [33], or MPI for short. Parallel MPI programs consist of several processes, with each process managing its own memory and variables. To communicate between

processes, several MPI functions can be used. These include functions for point-to-point communication such as `MPI_Send` and `MPI_Receive`, where a specific process sends data to another process, but also more sophisticated, collective methods such as broadcasting data to all processes within a group, or communicating data between all pairs of processes in a group.

In general, computational nodes in distributed memory machines also have shared memory parallelism, since the vast majority of CPUs on the market today are multi-core. This means that on many HPC machines, such as supercomputers, parallelism can be exploited at multiple levels. High level parallelism can be achieved using MPI, to distribute work between different computational nodes, but lower level parallelization can also be utilized within each computational nodes to further distribute work between different cores and threads on each node. Furthermore, many of the top supercomputers today use GPU accelerators as well [34], making them hybrid CPU-GPU systems, introducing yet another layer in potential parallelization. This introduces a challenge in how these systems should be programmed to make use of all available types of parallelism. One possible approach to make use of both shared and distributed memory parallelism is to use OpenMP for shared memory parallelism while distributing computation across nodes using MPI.

Another distinction is often made in HPC when discussing different hardware architectures, namely we distinguish between two kinds of architectures:

1. Latency oriented—focused on the ability to finish single computational tasks as quickly as possible.
2. Throughput oriented—focused on the peak amount of data that can be processed per unit time.

Modern CPU architectures when looked at from this perspective are often more *latency-oriented* architectures, with many levels of cache memory and deep instruction pipelines. A throughput-oriented architecture that has gained considerable popularity in recent years are GPU hardware accelerators [35]. GPUs have become popular for general purpose scientific computing due to their massive parallel computational efficiency, at a relatively low cost. This makes the use of accelerators especially attractive for users that need performance, but do not have large-scale supercomputing resources at hand. GPUs rely on a *data parallel* approach to computation, where the different parts of the input data can be processed in parallel. This is often the case in scientific computing, where input data often consist of large arrays of floating point numbers of some sort, which elements can, in large, be processed independently from each other. This enables GPUs to achieve high throughput levels for data-parallel problems where the input data are large, and since this is often the case in scientific computing, many problems can benefit greatly performance-wise from being able to utilize GPUs. However, we emphasize that not all algorithms and problems are suited to GPU acceleration, since a large degree of parallelism in the algorithm is required to make efficient use of the massive parallel computing power of the device.

Programming GPUs can be done using multiple different APIs. Some hardware vendors provide specialized APIs for their GPUs. This is the case for the CUDA programming model for Nvidia GPUs and HIP for AMD GPUs. There also exist other programming methods, not directly tied to specific hardware vendors, such as OpenCL [36] and OpenACC [37]. The advantage of using the more general APIs such as OpenCL or OpenACC lies in their ability to support multiple different types of hardware with a single programming model. However, what one gains in generality comes at a cost of specialization for different hardware; the hardware vendor specific programming models may be able to provide access to more specialized functionality and better performance by taking advantage of specific properties of their hardware.

In addition to CPUs and GPUs, the potential for other emerging hardware is also being seen in many areas of HPC. Examples of such architectures include reconfigurable hardware architectures, such as CGRAs [38] and FPGAs [39], as well as hardware specialized for certain types of workloads, such as artificial intelligence and machine learning [40]. The adoption of these technologies in traditional nonlinear optimization is still in its infancy, but may be an interesting topic for future research.

4 Parallelization strategies for nonlinear optimization algorithms

A crucial aspect in leveraging the computational power offered by HPC systems and hardware lies in being able to perform computations in parallel. Parallelization of the optimization solvers can be exploited at multiple different levels in the algorithm. If we consider the case of interior point methods, as illustrated in Fig. 1, we can identify two key steps that may become computational bottlenecks in the algorithm

- Computing the objective function, constraints and their derivatives (step two in Fig. 1).
- Solving a KKT linear system, similar to the one in Eq. (6) for search directions (step three in Fig. 1).

The first item, computing objective functions, constraints and derivatives, is very problem specific. As such, the opportunities for parallelization depend almost entirely on the specific optimization problem at hand. While this step can be a significant part of the computational effort for solving certain problems, implementing efficient methods for this is typically not done by the optimization solver library, but by the users on a per-problem basis. For example, as we discuss further in Sect. 5, interfacing to many solver libraries can be done in software, where the user provides their own functions for computing the objective, constraints and their derivatives, thus having full control over computing these values as efficiently as possible.

The second bottleneck is the solution of a linear system, similar to the one seen in Fig. 1, to find search directions. This step is often handled entirely by the optimization solver and thus is an important step to optimize to create efficient optimization

Table 1 Summary of the referenced works in the survey over parallelization methods, provided for an easier overview over the different topics. The table is organized by the subsections of the following text, which discuss different parallelization approaches

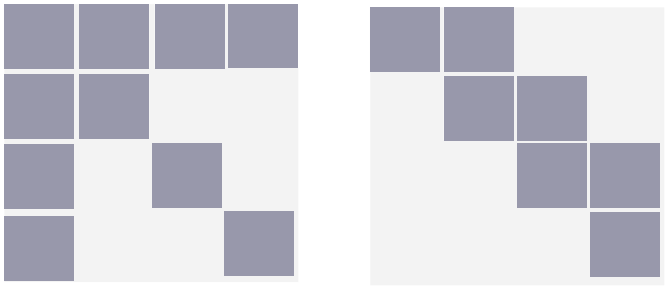
Parallelization Method	References
Problem Structure Exploitation	[41] (2008), [42] (2009), [43, 44] (2014), [45] (2015)
Parallel Direct Linear Solvers	[46] (2000), [47] (2004), [48] (2007), [49] (2006), (2007), [50] (2016), [51] (2019), [52] (2020)
Iterative Linear Solvers	[53] (2003), [54] (2005), [55] (2004), [56] (2007), [57] (2010), [11, 58, 59] (2012), [60] (2014)
Hardware Accelerators	[61] (2016), [62, 63] (2021)
Multiple Solver Instances	[64] (2016), [65] (2017)

solvers. Due to this being the step that optimization solvers have more control over, much of our discussion of previous work is focused on this step.

In the following, we review previous work on parallelization for nonlinear optimization. A summary of the reviewed works in the literature can be found in Table 1. Some of the work we review are related to existing optimization software, including IPOPT and PIPS-NLP, which we also include in our discussion on software packages for optimization in Sect. 5. For organizational purposes, we have separated our discussion of previous work on parallelization and our discussion on optimization solvers, but we note that some of the previous work on parallelization relates to software packages that we discuss more generally in Sect. 5.

4.1 Problem structure exploitation

For interior point methods, the structure of the optimization problem propagates down to the structure of the linear systems arising from the iterations of the solver. In certain cases, this fact can be used to solve the linear system in parallel. Traditionally, the Schur-complement of a matrix can be used to solve linear systems of equations by dividing the matrix into blocks. Some examples of blocked structures that may arise, and be exploited for parallelism, in linear systems are shown in Fig. 3. As we will see in the following, Schur-complement methods can also be used to exploit block structure in matrices to solve linear systems in parallel. As an example, Kang et al. show how the structure of linear systems for some optimal control problems, together with a Schur-decomposition, can be used to parallelize the solution of linear systems in interior point methods [45]. In that case, the linear systems have what is referred to as a block bordered diagonal structure, and the factorization and solution of the linear systems can be parallelized across blocks of the matrix. A similar approach is also used by Chiang et al. in [44], and this approach is implemented in the PIPS-NLP software library. The Schur-complement decomposition approach has also been studied within the IPOPT framework for nonlinear optimization in [41], where distributed parallelism using MPI is implemented within the IPOPT solver and performance is demonstrated for a parameter estimation problem with applications in chemical engineering. The work by Zavala et al. in [41] is further extended



(a) Block-bordered diagonal matrix

(b) Block banded matrix

Fig. 3 Illustrative examples of matrix structures that may be utilized for parallelism when solving linear systems

in [43] by solving the Schur-complement linear system using a preconditioned conjugate gradient algorithm. This enabled the linear system to be solved in a matrix-free fashion, without needing to explicitly form the Schur-complement, which could be a computational bottleneck for large problems. While these approaches are highly suited for specific types of optimization problems, they are not always applicable to general problems, since they exploit structure in the optimization problem (e.g., relationships between variables and constraints) which is not present for general problems.

To exploit structure in more general problems, it may be of interest to develop a parallel processing framework capable of utilizing structure more generally. In [42], Gondzio and Grothey develop such a framework, using object-oriented programming design, for exploiting block structure in the linear systems arising from primal-dual interior point methods. The linear systems are solved using a direct Cholesky factorization, and block structure is utilized by constructing a so-called *elimination tree* [66], representing dependencies between operations to carry out for the Cholesky factorization. The resulting tree can be used to determine blocks of the linear system that can be processed independently of other blocks, thus enabling parallelization. Their approach is implemented in the OOPS software library.

4.2 Parallel linear solvers

In cases where there are essentially no assumptions on the structure of the optimization problem and resulting linear systems, general parallel linear system solvers can still be used. In practice, many optimization libraries rely on a direct solver for linear systems of equation to find the search direction (i.e., solving a system similar to Eq. (6)), based on matrix factorization. Often times, the linear solver is an external dependency of the optimization library, and being an important computational bottleneck means that parallelization of this step can be crucial for performance.

A study on the parallelism and performance of linear solvers for the popular interior point solver IPOPT was performed by Tasseff et al. in [51], in the case of sparse,

symmetric and indefinite linear systems (as often arise in interior point methods). They compare a group of linear solvers from the HSL mathematical software library [49], PARDISO [47], MUMPS [46], SPRAL [50, 52], and WSMP [67]. Most of the reviewed solvers support parallelism in some form, either through the use of multi-threaded BLAS operations, or shared and distributed memory parallelism through, e.g., OpenMP or MPI. They find that the problem size is important to benefit from parallelization of the linear solver, with bigger problem sizes being more suited for solvers capable of parallelism, like PARDISO and SPRAL, compared to serial solvers such as MA27. Overall, Tasseff et al. found that the best performing solver varied quite significantly depending on the type and size of the problem. Another finding is that many common benchmark problems for optimization, such as the CUTEst library [68], contain mainly relatively small problems, where the authors found that serial solvers like MA27 from HSL performed well.

Other work addressing this important bottleneck include the work by Schenk et al. [48], where graph-matching based pre-processing is applied to the linear systems to make them more suitable for numeric factorization. Using the PARDISO linear solver and the IPOPT package for nonlinear optimization, the authors observed a parallel speedup of up to 5.2x using 8 IBM Power 4+ CPUs (relative to using only one) on some large-scale PDE-constrained optimal control problems.

4.2.1 Iterative linear solvers

The linear solver packages discussed in the previous paragraph are mainly direct solvers, using matrix factorization or variations of Gaussian Elimination to solve the systems, and indeed direct solvers are commonly used in most nonlinear optimization libraries. For very large problems, however, the use of direct solvers can be computationally infeasible, both because of the large number of operations required, and because the linear systems may be so large that they cannot be stored in the memory of a single computer (see for instance [69]). In this case, iterative linear solvers, where increasingly accurate approximations of the solutions are computed until some desired level of accuracy has been achieved, may be more suitable. In particular, many algorithms for solving linear systems iteratively (e.g., conjugate gradient methods) can utilize efficient, parallel implementations of basic linear algebra kernels, such as matrix-vector or vector-vector products. These kernels form the computational core of many iterative methods, and efficient implementations for different hardware are common (e.g., BLAS implementations). Furthermore, iterative methods can be performed in a matrix-free fashion, meaning the linear system does not need to be explicitly formed, which is particularly useful for large problems, where the linear system may be so large that it cannot fit in the memory of a single computational node at all. For comparisons between iterative and direct solvers, we refer to the work by Brussino and Sonnad [70].

The use of iterative linear solvers inside interior point methods is an idea that has existed for some time. In [11], Gondzio suggests that iterative linear solvers may be a key step forwards for interior point methods applied to truly large-scale optimization problems. Other works by Gondzio include the work in [58], where

an iterative Krylov solver is used for solving the KKT system in a matrix-free fashion (i.e., without explicitly forming the KKT-matrix). The method uses regularization and preconditioning to aid the convergence when solving the linear systems, however, the work concerns the solution of convex quadratic programs, and is thus not directly applicable to general nonlinear problems. However, since interior point methods can be used to solve the quadratic sub-problems from SQP methods (which in turn can solve general nonlinear problems), it may still be interesting for certain implementations.

Other works on iterative linear solvers for interior point methods also include the work by Forsgren et al. in [56], in which they study the use of a preconditioned conjugate gradient (PCG) method for KKT-systems from interior point methods. They work on an *augmented linear system*, which is equivalent to the standard Newton system of equations in interior point method, but with the added benefit of being positive definite. The authors apply a constraint preconditioner [71] on the augmented system and further carry out numerical experiments to demonstrate the feasibility of their approach on a number of benchmarking problems from the COPS dataset [72]. In [55], Bergamaschi et al. study the use of two incomplete Cholesky-type preconditioners for interior point methods. They prove some spectral properties of the preconditioned matrices, in particular that they are positive definite for convex problems, and demonstrate the effectiveness of their approach on a set of quadratic programs, with around an order of magnitude speedup on average for the iterative approach compared to a direct solver. Other works on preconditioners for linear systems from interior point methods include [53, 54].

Another category of work on utilizing iterative linear solvers for computing the search direction concerns the handling of the potentially inexact step calculations as well as criteria for ensuring the search directions will converge globally. Examples include the work by Curtis et al. in [59], where the authors utilize an iterative linear solver to compute an approximate search direction in an interior point algorithm. The work is based on an algorithm proposed in [57], which is an interior point method with inexact search directions, but that retains convergence guarantees to local optima. The work in [59] can be seen as an implementation of the algorithm proposed in [57] with many practical improvements and enhancements. The authors evaluate their implementations on both test problems from the CUTEst test set [68], as well as some custom PDE-constrained optimization problems. The authors found that especially for the larger PDE constrained problems, the inexact step calculation with an iterative linear solver performed significantly better than the approach in IPOPT using a direct linear solver. Further work on the method is done by Grote et al. in [60], using a preconditioner that is more efficient and suitable for parallelization.

4.3 Hardware accelerators

Another interesting opportunity for improved performance lies in the use of hardware accelerators like GPUs to solve the linear systems. GPUs provide a large amount of parallel processing power for a relatively low cost, thus making them an

attractive choice for many applications. However, not all problems are naturally well suited for GPU acceleration. A study by Świrydowicz et al. [62] evaluated linear solvers with GPU acceleration for interior point methods. They evaluated five different linear solver packages: cuSolver, PaStiX, SuperLU, STRUMPACK, SPRAL/SSIDS. Among the tested solvers and cases, the authors did not find that the GPU acceleration gave any major improvement in performance. Based on their results, a factor that could explain the GPU performance is pivoting strategies used, since pivoting could cause irregularities in memory accesses and data movement. Another factor impacting GPU performance in the tested packages was a large amount of memory allocations in the host for many of the tested libraries, causing the factorization step to take a lot of time.

In general, GPUs might have better performance when utilizing iterative methods for solving linear systems when compared to direct solvers, based on, e.g., factorization of the matrix. A problem with using iterative methods for nonlinear optimization solvers lies in that linear systems arising are often indefinite. This is often the case for interior point methods [51], which are widely used in nonlinear optimization.

One approach to overcome the problem with indefinite linear systems is explored by Cao et al. in [61], where instead of using an interior point method directly, the authors use an augmented Lagrangian method. The augmented Lagrangian method solves the original nonlinear problem by solving a series of bound constrained problems, similarly to how SQP methods solve a series of quadratic problems. The bound-constrained sub-problems were solved using an interior point method, with the advantage that the linear systems to be solved were positive definite for convex problems. Using a GPU implementation of the preconditioned conjugate gradients (PCG) to solve the linear systems, the authors evaluated their approach using a test-system consisting of an Nvidia Tesla C2050 GPU together with an Intel Xeon E5420 CPU on some problems from the COPS dataset [72] and were able to outperform the IPOPT optimization solver (see Section 5.1.1). Other approaches include the work by Regev et al. in [63], where a hybrid direct-iterative method for solving indefinite KKT systems of equations is considered. The approach uses Schur-complement decomposition for a re-written KKT system, which is solved using a possibly preconditioned CG. In forming the equations to solve in the Schur-decomposition, a direct Cholesky factorization of a submatrix is used, hence the hybrid approach. The authors also demonstrate the feasibility of their approach on a set of problems from power grids.

4.4 Multiple solver instances

For certain types of problems, parallelism can be leveraged by launching multiple instances of a solver on different computational nodes or cores. At first, this might seem redundant, since it only gives multiple different solutions, but does not necessarily produce a single solution in a faster time. However, there can still be value in launching multiple instances of a solver on a given problem. One such use comes

from the fact that many numerical optimization solvers have many tunable parameters to guide the optimization. These can include, convergence tolerances, parameters related to how the step length in each iteration is chosen among others. While most solvers will aim to provide reasonable default values for the parameters, choosing optimal ones can be highly problem specific.

An example of work utilizing launching multiple solver instances with different parameter settings was done by Geffken and Bũskens in [64]. The authors explore two main approaches of utilizing parallelism by launching multiple instances, first across the line (FATL) and best of all. The first across the line approach aims to find an optimal solution faster by launching multiple solver instances in parallel and finishing when one of the instances first returns an optimal solution. The best-of-all approach tries to find a better solution by waiting until all launched instances finish, and selecting the best of the solutions.

Other work utilizing this type of parallelism includes the work by Hu et al. in [65], where a GPU acceleration is used for SQP. The proposed algorithm uses a predictor-corrector based interior point method to solve the quadratic sub-problems and leverages the cuSOLVER [73] library to solve the KKT-systems on the GPU in a batched fashion.

5 Software libraries for nonlinear optimization

In the previous section, we discuss different approaches for parallelization of nonlinear optimization solvers. Moving on in the current section, we review a selection of popular software libraries for nonlinear optimization. Note that, the following is not intended to be a complete list of available libraries for nonlinear optimization, but a selection loosely based on focus on parallelism and/or large-scale problems, intended to give readers an idea of the types of optimization packages that are available for different types of problems.

5.1 Software packages

We begin by giving a brief introduction to some software packages for nonlinear optimization. Later, we will also discuss methods of interfacing ones optimization problem to a solver followed by a short summary.

5.1.1 IPOPT

IPOPT [74] is an optimization solver for general nonlinear optimization problems using an interior point method. To specify optimization problems, IPOPT provides an interface to the AMPL modeling language, as well as programmatic interfaces in C++, C, Fortran, Java and R. The IPOPT code itself is written in C++ and is

available as open-source on Github[1]. Computationally, as is the case for many interior point methods, IPOPT relies heavily on solving sparse, symmetric, indefinite linear systems. For this purpose, IPOPT uses external libraries which are supplied by the user. There is support for a number of different solvers with different performance characteristics. Some studies have been done regarding the performance of different solvers in IPOPT, see for instance [51, 62]. In general, one can likely expect that the best linear solver to use will vary depending on problem type. For easy problems, [51] find good performance from MA57 and MA27 from the HSL, while PARDISO and SPRAL were found to perform well for more difficult problems. IPOPT also provides some support for GPU acceleration through the interface to the SPRAL linear solver, however, some studies indicate limited benefit of GPU acceleration for IPOPT [62].

IPOPT is intended to be usable for large-scale nonlinear optimization problems, an example of such work can be found in the work by [48], where problems with up to 12 million constraints and 6 million variables are solved using IPOPT, together with the author's proposed pre-processing methods for the KKT-systems.

5.1.2 PETSc/TAO

The Toolkit for Advanced Optimization, or TAO, is an optimization library developed at Argonne National Laboratories on top of the PETSc [75] library, which contains an extensive collection of routines for general, scalable linear algebra, solution of linear and nonlinear systems of equations among others. PETSc/TAO is primarily written in C, with support for GPUs through CUDA, HIP or OpenCL. TAO (and PETSc) provides programmatic interfaces in C/C++ and Fortran, where the user specifies routines for calculating required values like the objective function, gradients and potentially Hessians. PETSc/TAO also has a Python interface through the `petsc4py` package [76]. TAO supports multiple different types of optimization problems, including unconstrained problems, problems with box constraints, i.e., constraints of the form $l_i \leq x_i \leq h_i$, where x_i are the optimization variables and l_i and h_i are constant upper and lower bounds for the variables, respectively, PDE constrained optimization, and general constrained nonlinear optimization, among others. For general nonlinear optimization problems, as we are mainly concerned with in this review, TAO provides an implementation of the Augmented Lagrangian Method, as well as a primal-dual interior point method.

For the interior point method, the PETSc/TAO User Manual [77] recommends using a direct linear solver, e.g., by using the command line option `-pc_type lu`, due to ill-conditioning of the linear systems. Furthermore, the manual recommends using an external package for solving linear systems, such as SuperLU_dist [78] or MUMPS [46], both of which are capable of utilizing distributed memory parallelism through MPI.

[1] https://github.com/coin-or/Ipopt.

5.1.3 HiOp

HiOp is an HPC solver for nonlinear optimization problems developed at the Lawrence Livermore National Laboratory. HiOp has specific interfaces for a number of different types of nonlinear problems. The first interface is for problems with a very large number of variables (up to possibly billions), but only up to approximately 100 general constraints. The underlying algorithm is an interior point method which utilizes a limited memory quasi-Newton approximation of the Hessian. The implementation and parallelization strategy used for the first interface is described in [79].

Secondly, HiOp also provides an interface for general sparse nonlinear problems. For this interface, general problems can be supplied by the user, without any assumptions on the structure or nature of the problem, as long as the Jacobians and Hessian of the Lagrangian is available. The third and final interface provided is a mixed dense-sparse interface, where the optimization variables are split into "sparse" and "dense" groups, x_s and x_d, respectively. This split, together with an assumption on the structure of the Hessian of the Lagrangian $\mathcal{L}$, namely that the Hessian when taken with respect to first x_s and the x_d and vice versa is zero: $\nabla^2_{x_d x_s}\mathcal{L} = \nabla^2_{x_s x_d}\mathcal{L} = 0$. This enables HiOp to utilize dense linear algebra kernels in its computations at a much higher rate, where accelerators like GPUs can perform significantly better in comparison with the sparse case [80]. To keep the size of changes to the original source code as small as possible, the RAJA abstraction library [81] was used, which allows kernels to be written using RAJA's C++-like API, with porting to different hardware accelerators handled by RAJA's backend.

HiOp is written in C++ and can be interfaced thorugh an object-oriented C++ interface. The interfacing is described in the user manual [82]. The user provides an implementation deriving from an abstract class provided by HiOp, together with the required functions. Each of the previously described interfaces for dense, mixed dense-sparse and sparse problems provides one abstract class which users derive from. The user then provides concrete implementations for the required virtual functions representing the optimization problem, and the solver takes care of the rest.

HiOp is also capable of distributed memory parallelism through MPI, particularly for its dense interface for problems with a limited number of general constraints [82]. To achieve this, the data for the optimization problem, in particular for the objective function and constraints and their corresponding gradients, need to be distributed across the MPI ranks. In HiOp, this distribution is specified by the user by overriding the `get_vecdistrib_info` function. Generally, HiOp's philosophy is to distribute the quantities whose size depends on the number of optimization variables n, such as the gradient of the objective function, the optimization variables themselves and the Jacobian of the constraints.

HiOp's performance has been demonstrated on large-scale problems, including examples from topology optimization [79] where good parallel scaling on thousands of cores is demonstrated. Note that, this example uses the HiOp interface supporting a very large number of variables, but a relatively small number of general constraints (approximately 100).

5.1.4 PIPS

PIPS[2] is a suite of parallel optimization solvers developed at Argonne National Laboratories, with the solvers designed to be able to utilize problem structure for parallelization. PIPS includes a distributed memory parallel solver for nonlinear programs called PIPS-NLP [44], which is a structure utilizing interior point method. Users can specify their optimization problems to the solver by using the AMPL or Pyomo modeling languages, or by specifying callback functions programmatically in C or C++. The PIPS-NLP code itself is written in C++.

The key feature of PIPS-NLP is the ability to exploit structure in problems for parallelization, as we discussed in sect. 4.1. PIPS-NLP can exploit the block bordered diagonal structure (see Fig. 3a) in linear systems arising from certain types of optimization problems, e.g., from stochastic optimization or optimal power flow. By utilizing the problem structure, PIPS-NLP achieves good parallel scaling for, e.g., optimal power flow problems [44] and has also been used for the solution of other large scale problems in optimal control [83].

5.1.5 OOPS

OOPS [84], or Object Oriented Parallel Solver, is an optimization solver specialized in finding structure in the Jacobian and Hessian matrices and utilizing it for parallelization. The underlying algorithm for OOPS is an interior point method, and interfacing to it can be done using Structured Modeling Language (SML) [85], which is an extension of the AMPL modeling language.

Structure in matrices is exploited in OOPS by representing block structure using a tree. The nodes of the tree represent blocks in the matrices, and each node can in itself have special structure as well, which is accomplished in an object-oriented fashion, by each node containing an implementation of an abstract class `Matrix`, representing a structured, or general matrix of some kind.

OOPS is parallelized using MPI, and the parallelization can be done by utilizing the block-representation of matrices described previously. The parallel performance of OOPS has been demonstrated on stochastic programming problems arising from financial planning with over a billion variables [86].

5.1.6 WORHP

We Optimize Really Huge Problems, or WORHP, is a solver for nonlinear optimization problems by the European Space Agency. As the name suggests, the focus of the library is to solve large-scale optimization problems arising from areas such as optimal control. WORHP uses a sequential quadratic programming algorithm, where the quadratic sub-problems arising from the SQP iterations are solved using an interior point method [18]. WORHP aims to be an optimizer usable in industrial-grade settings as well as for academic use. One unique design decision of WORHP

[2] https://github.com/Argonne-National-Laboratory/PIPS.

is the use of a so-called *reverse communication* paradigm [87]. In most libraries for optimization, a typical workflow may involve the user supplying functions to the optimization library to compute objective functions, gradients and the like, after which the optimization library solves the problem from beginning to end, until some convergence criteria have been met. The reverse communication paradigm changes this by letting the user call the solver repeatedly, which then does a step of the computation before returning to the user what the next expected step is. This approach can give users more control over the process of solving the optimization problem.

5.1.7 Knitro

Knitro [88] is an optimization solver developed by the company Artelys. Knitro has support for general nonlinear programs, as well as mixed-integer nonlinear programs. Furthermore, users can interface their problems to the Knitro solver in a large number of ways, including through multiple programming languages: C, C++, Fortran, Java, Python, C# and Julia, as well as interfaces to GAMS, AMPL, MATLAB, Microsoft Excel, R and MPL. The Knitro code itself is under a proprietary license.

For nonlinear problems, the Knitro solver provides four different algorithms [88, 89]. Two of the algorithms are interior point methods, called Interior/Direct and Interior/CG and differ mainly in how the barrier sub-problems are solved. The method used for Interior/Direct is described in [90] and employs the classic approach of solving a series of primal-dual KKT system of equations using a direct linear solver to find search directions for the next iterate. The Interior/CG method is based on the method described in [91] and solves the barrier sub-problems for the search directions using an SQP-inspired method. The equality constrained quadratic sub-problems from the SQP iterations are solved using a preconditioned conjugate gradient method. Knitro further includes an active set algorithm, based on a so-called sequential linear quadratic programming (SLQP) method, described in [92]. Finally, an SQP method is also available in Knitro for nonlinear problems.

5.1.8 SNOPT

SNOPT [93] is an SQP solvers for constrained nonlinear optimization problems. The SNOPT implementation, as any SQP method, solves a sequence of quadratic sub-problems to find search directions. The quadratic sub-problems are solved using an active-set method implemented in the SQOPT package [94]. The SNOPT code is written in Fortran 77, but has interfaces to multiple programming languages such as C, C++ and MATLAB. Furthermore, many modeling languages have support for SNOPT, including GAMS, AMPL and AIMMS.

When interfacing to SNOPT from a programming language, a number of different interfaces can be used, called `SnoptA`, `SnoptB`, `SnoptC` and `NpOpt` [95]. These interfaces provide similar functionality, but the functions used to interface to the solver may differ slightly. For example, the `SnoptC` interface allows the user to give a single function for calculating the objective function and constraints, which can be useful if the objective and constraints share data or intermediate calculations.

For a more thorough presentation of the different interfaces, we refer to the SNOPT user guide [95].

5.2 Interfacing and usage

In order for an optimization library to be useful, there needs to be a way for the users to specify the optimization problems that they wish to solve. To do this, most libraries will provide a number of interfaces to their optimization solvers.

One approach is to interface the solver with a modeling language specifically developed for optimization libraries. In this approach, the user specifies their optimization problem in a custom modeling language for optimization problems. The back-end of the modeling language implementation then provides an interface for the problem that an optimization solver can use. One advantage of this approach is that a large number of solvers will usually have support for a given modeling language, so that the user can use the multiple optimization solvers without having to re-specify their optimization problem for each solver. Furthermore, specifying optimization problems in a custom modeling language might be easier for optimization experts with programming experience. Examples of modeling languages for optimization include GAMS [96], AMPL [97], GNU MathProg [98], AIMMS [99], among others. Most of these, with the exception of MathProg, are proprietary and may require a license to use.

With most optimization solvers being software libraries, another way to interface to solvers is using a regular programming language interface. A common approach for this is for the user to write functions to calculate the values required by solver to run. These functions will typically require the current values of the decision variables as input, and output things like the objective function value, the gradient of the objective function, the Jacobian of the constraints, the Hessian of the Lagrangian function, etc. The method for providing these functions to the solver will vary depending on the programming language, in C and C++, function pointers can be used, in Fortran, one can use procedure pointers, many other languages, such as Python, have first-class functions which can be passed as function arguments directly. This way, the optimization solver can provide functions asking the user to provide the required functions for calculating relevant values of the optimization problem.

Some advantages of using programmatic interfaces include being able to have precise control over how the objective function, Jacobians and Hessians are calculated. Since almost any type of function could be used as an objective or constraint in nonlinear optimization, some problems may have objectives and constraints that are very expensive to calculate. Being able to provide one's own functions for computing those values provides a higher level of flexibility, but also an opportunity for performance optimizations. Problems where the objective function and gradients are very expensive to compute could in certain cases be parallelized or offloaded to accelerators, leading to a large improvement in the solution time, irrespective of the solver used.

5.3 Summary

In this section, we have presented optimization packages for nonlinear optimization, and a summary of some features of the presented codes can be found in Table 2. We note that this is not a complete list of software packages for nonlinear optimization, but rather a selection based on maturity and focus on support for large-scale problems and HPC. The support for parallel computing and HPC hardware varies between the different solvers we have considered, although most solvers are capable of shared-memory parallelism using multi-threading. In particular, we see that the support for GPU acceleration may be limited in many of the solvers. Of the solvers we have considered, only HiOp has GPU support natively, and that for a specific class of problems (see sect. 5.1.3 for details). IPOPT's support for GPUs comes from its support for some external libraries for solving linear systems with GPU support, however, the performance benefit for optimization problems may be limited [62].

A number of the solvers have support for distributed, multi-node computing through MPI. Notably, these are PETSc/TAO through its extensive support for distributed linear algebra, PIPS and OOPS for problems with certain structure, and HiOp for problems with a limited number of general constraints. This type of parallelism may be especially useful for extremely large problems, for which a single compute node may not even have sufficient memory.

To the authors' knowledge, recent works benchmarking the performance of different optimization solvers are quite limited. One benchmarking effort that the authors are aware of are the benchmarking efforts performed by Hans Mittelmann, which are available online[3]. The relevant benchmark for this review is referred to as `AMPL_NLP`, and of the solvers discussed in this paper, the ones also included in the benchmark are IPOPT, SNOPT, WORHP and Knitro. On the approximately 50 test problems (ranging in size from approximately 100 to 250000 variables and constraints) used in Mittelmann's benchmarks, Knitro is the best performing, followed closely by IPOPT and WORHP, with SNOPT being the worst performing. We note that these results may not generalize to arbitrary nonlinear problems, since the performance of different solvers will vary depending on the characteristics of the optimization problem.

Of the solvers considered in this paper, IPOPT has the advantage of being open-source and freely available as well as mature and with rich functionality. We believe that this, together with its strong performance on the Mittelmann benchmarks, makes it a good first candidate for those needing to solve general optimization problems. However, its limited support for GPUs, and lack of multi-node support may make it unusable for problems with an extremely large number of variables or constraints, in which case a solver like PETSc/TAO or HiOp may be more suitable. From the aforementioned benchmarks performed by Mittelmann, WORHP and Knitro also perform well and may be a good choice for those who are able to get a license for those codes.

[3] http://plato.asu.edu/bench.html.

Table 2 Summary of features and capabilities of the optimization solvers, including programming language that the library is written in, software license, algorithms and different kinds of multi-processing

Library	Language	License	Algorithm	Multi-thread	Multi-node	GPU
IPOPT	C++	EPL	IPM	Depending on ext. linear solver	No	From ext. linear solver
PETSc/TAO	C, Python	BSD	IPM, ALM	Depending on ext. linear solver	MPI	From PETSc
PIPS	C++	Open-Source	IPM	Yes	MPI	No
HiOp	C++	BSD	IPM	From BLAS/LAPACK	MPI	RAJA
WORHP	C, Fortran	Proprietary	SQP	Yes	No	No
Knitro	C/C++	Proprietary	SQP, IPM, SLQP	Yes	No	No
SNOPT	Fortran	Proprietary	SQP	No	No	No
OOPS	C	Proprietary	IPM	No	MPI	No

6 Discussion - future research

There exists, as we have discussed, a significant body of previous research on parallelization utilization of HPC hardware for nonlinear optimization algorithms. Still, we believe there are many possibilities for future research, some of which we will discuss in the following section.

Considering the limited options for optimization libraries capable of utilizing GPUs (as discussed in Sect. 5.3), work on GPU-capable optimization algorithms and solvers could present an interesting avenue for future research. In particular, we believe that the use of GPU-accelerated iterative linear solvers to solve KKT-systems is a promising approach, especially considering the results from the work by Tasseff et al. [51] on benchmarking GPU-accelerated direct linear solvers for nonlinear optimization. To this end, inspiration could be gathered from the survey in Sect. 4.2.1, where previous work on applying iterative linear solvers to KKT-systems from nonlinear optimization is presented. Much of the discussion in the referenced previous work on this topic concerns interesting numerical aspects of using iterative methods for optimization, such as efficient preconditioning and suitable formulations of the interior point algorithm. How suitable those methods are for acceleration using GPUs could be a topic for future research.

Another possibility for future research lies in exploring the use of emerging accelerator architectures, such as FPGAs, for nonlinear optimization applications. While there is previous work on using FPGAs for some classes of optimization problems, model predictive control being a noteworthy example [100], their use for general nonlinear optimization seems to be a topic that is less explored. Considering the trends in HPC to move toward more heterogeneous computing, assessing the potential of emerging hardware on nonlinear optimization problems seems to be an important topic to further investigate.

Finally, when looking at disruptive computing technologies, Quantum Computing and Noisy Intermediate-Scale Quantum (NISQ) systems and algorithms are arising as potential candidates to speed up the solution of optimization problems. In particular, Variational Quantum Algorithms (VQAs) [101] are based on a hybrid classical and quantum approach combining a classical optimizer with training a parametrized quantum circuit. When solving nonlinear equations and optimization problems, a fundamental challenge is that the underlying quantum mechanics mathematical framework is linear and different approaches were proposed to tackle this challenge. Ref. [102] introduces an approach using multiple copies of variational quantum states in the cost-evaluation approach to computing nonlinear functions and a tensor-network programming paradigm. Ref. [103] proposes an alternative approach based on basis functions encoded as nonlinear features maps and a parametrized quantum circuit to represent a linear combination of these basis functions. The increase in quantum computing capabilities, error-correcting algorithms, and the development of new quantum algorithms will likely place quantum computers as one of the primary workforces for solving optimization in the next future.

7 Conclusions

In this paper, we have surveyed previous work on parallelization and high-performance computing for solving large-scale nonlinear optimization problems. The computational core of many algorithms for nonlinear programming, such as interior point methods, lies in solving systems of linear equations to determine a search direction for each iteration. Methods for parallelizing this computation range from problem-specific work on utilizing the structure of certain optimization problems to parallelize the computation into blocks, to more general approaches such as using parallelization in general matrix factorization routines, or using iterative linear solvers instead of direct ones when computing search directions, leading to increased efficiency. The latter method, namely using iterative linear solvers for the search directions, may also be more suited for GPU acceleration.

There are existing software packages capable of utilizing HPC computing resources for nonlinear optimization, with a number of the solvers considered in this review being capable of distributed parallelism through MPI, for instance. The support for GPU accelerations is more limited in the solvers we consider, however, with IPOPT having limited support for GPU acceleration through external linear solver libraries, and HiOp supporting GPU acceleration for problems with a specific mixed dense-sparse structure. This may thus present an interesting avenue for future research.

Funding Open access funding provided by Royal Institute of Technology.

Data Availability Statement Data sharing not applicable to this article as no datasets were generated or analyzed during the current study.

References

1. Brahme A (2000) Development of radiation therapy optimization. Acta Oncol 39(5):579–595
2. Frank S, Steponavice I, Rebennack S (2012) Optimal power flow: a bibliographic survey i. Energy Syst 3(3):221–258
3. Rao AV (2009) A survey of numerical methods for optimal control. Adv Astronaut Sci 135(1):497–528
4. Piccialli V, Sciandrone M (2018) Nonlinear optimization and support vector machines. 4OR 16(2):111–149
5. Bartholomew-Biggs M (2006) Nonlinear optimization with financial applications. Springer, New York, NY, USA

6. Capitanescu F (2016) Critical review of recent advances and further developments needed in ac optimal power flow. Electric Power Syst Res 136:57–68
7. Jia X, Ziegenhein P, Jiang SB (2014) Gpu-based high-performance computing for radiation therapy. Phys Med Biol 59(4):151
8. Nocedal J, Wright SJ (1999) Numerical Optimization. Springer, New York, NY, USA
9. Sun W, Yuan Y-X (2006) Optimization theory and methods: nonlinear programming, vol 1. Springer, Boston, MA, USA
10. Forsgren A, Gill PE, Wright MH (2002) Interior methods for nonlinear optimization. SIAM Rev 44(4):525–597
11. Gondzio J (2012) Interior point methods 25 years later. Eur J Oper Res 218(3):587–601
12. Mehrotra S (1992) On the implementation of a primal-dual interior point method. SIAM J Optim 2(4):575–601
13. Gondzio J (1996) Multiple centrality corrections in a primal-dual method for linear programming. Comput Optim Appl 6(2):137–156
14. Wilson RB (1963) A simplicial algorithm for concave programming. PhD thesis, Harvard University
15. Boggs PT, Tolle JW (1995) Sequential quadratic programming. Acta Numerica 4:1–51
16. Gill PE, Wong E (2012) Sequential quadratic programming methods. Mixed integer nonlinear programming. Springer, NewYork, pp 147–224
17. Griva I, Nash SG, Sofer A (2009) Linear and nonlinear optimization. Siam, Philadelphia. Chap. 15.5
18. Büskens C, Wassel D (2012) The esa nlp solver worhp. In: Modeling and optimization in space engineering, pp. 85–110. Springer, New York
19. Wong E (2011) Active-set methods for quadratic programming. PhD thesis, University of California, San Diego
20. Nocedal J, Wright SJ (2006) Penalty and augmented lagrangian methods. Numer Opt 1:497–528
21. Boyd S, Parikh N, Chu E (2011) Distributed optimization and statistical learning via the alternating direction method of multipliers. Now Publishers Inc, Hanover, MA, USA
22. Conn AR, Gould G, Toint PL (2013) LANCELOT: a fortran package for large-scale nonlinear optimization (Release A) vol. 17. Springer, Berlin, Heidelberg, New York
23. Simon D (2013) Evolutionary optimization algorithms. Wiley, Hoboken, NJ, USA
24. Zhang Y, Wang S, Ji G (2015) A comprehensive survey on particle swarm optimization algorithm and its applications. Mathematical problems in engineering
25. Bertsimas D, Tsitsiklis J (1993) Simulated annealing. Stat Sci 8(1):10–15
26. Alba E (2005) Parallel metaheuristics: a new class of algorithms. Wiley, Hoboken, NJ, USA
27. Lalwani S, Sharma H, Satapathy SC, Deep K, Bansal JC (2019) A survey on parallel particle swarm optimization algorithms. Arab J Sci Eng 44(4):2899–2923
28. Martins JR, Ning A (2021) Eng Design Opt. Cambridge University Press, Cambridge, United Kingdom
29. Tufo HM, Fischer PF (1999) Terascale spectral element algorithms and implementations. In: Proceedings of the 1999 ACM/IEEE Conference on Supercomputing, p 68
30. Coates A., Huval B, Wang T, Wu D, Catanzaro B, Andrew N (2013) Deep learning with cots hpc systems. In: International Conference on Machine Learning, pp 1337–1345 (2013). PMLR
31. Sterling T, Brodowicz M, Anderson M (2017) High performance computing: modern systems and practices. Morgan Kaufmann, San Fransisco, California, USA
32. Dagum L, Menon R (1998) Openmp: an industry standard api for shared-memory programming. IEEE Comput Sci Eng 5(1):46–55
33. Forum MPI (2021) MPI: A Message-Passing Interface Standard Version 4.0.
34. Khan A, Sim H, Vazhkudai SS, Butt AR, Kim Y. (2021) An analysis of system balance and architectural trends based on top500 supercomputers. In: The International Conference on High Performance Computing in Asia-Pacific Region, pp 11–22
35. Owens JD, Houston M, Luebke D, Green S, Stone JE, Phillips JC (2008) Gpu computing. Proc IEEE 96(5):879–899
36. Munshi A, Gaster B, Mattson TG, Ginsburg D (2011) OpenCL programming guide. Pearson Education, Boston, MA, USA
37. Farber R (2016) Parallel programming with OpenACC. Newnes, Cambridge, MA, USA
38. Podobas A, Sano K, Matsuoka S (2020) A survey on coarse-grained reconfigurable architectures from a performance perspective. IEEE Access 8:146719–146743

39. Mittal S (2020) A survey of fpga-based accelerators for convolutional neural networks. Neural Comput Appl 32(4):1109–1139
40. Reuther A, Michaleas P, Jones M, Gadepally V, Samsi S, Kepner J (2020) Survey of machine learning accelerators. In: 2020 IEEE High Performance Extreme Computing Conference (HPEC), pp 1–12. IEEE
41. Zavala VM, Laird CD, Biegler LT (2008) Interior-point decomposition approaches for parallel solution of large-scale nonlinear parameter estimation problems. Chem Eng Sci 63(19):4834–4845
42. Gondzio J, Grothey A (2009) Exploiting structure in parallel implementation of interior point methods for optimization. CMS 6(2):135–160
43. Kang J, Cao Y, Word DP, Laird CD (2014) An interior-point method for efficient solution of block-structured nlp problems using an implicit schur-complement decomposition. Comput Chem Eng 71:563–573
44. Chiang N, Petra CG, Zavala VM (2014) Structured nonconvex optimization of large-scale energy systems using pips-nlp. In: 2014 Power Systems Computation Conference, pp 1–7. IEEE
45. Kang J, Chiang N, Laird CD, Zavala VM (2015) Nonlinear programming strategies on high-performance computers. In: 2015 54th IEEE Conference on Decision and Control (CDC), pp 4612–4620. IEEE
46. Amestoy PR, Duff IS, L'Excellent J-Y, Koster J (2000) Mumps: a general purpose distributed memory sparse solver. In: International Workshop on Applied Parallel Computing, pp 121–130. Springer
47. Schenk O, Gärtner K (2004) Solving unsymmetric sparse systems of linear equations with pardiso. Futur Gener Comput Syst 20(3):475–487
48. Schenk O, Wächter A, Hagemann M (2007) Matching-based preprocessing algorithms to the solution of saddle-point problems in large-scale nonconvex interior-point optimization. Comput Optim Appl 36(2):321–341
49. Duff IS (2006) Sparse system solution and the hsl library. Some Topics Ind Appl Math 8:78–94
50. Hogg JD, Ovtchinnikov E, Scott JA (2016) A sparse symmetric indefinite direct solver for gpu architectures. ACM Tran Math Softw(TOMS) 42(1):1–25
51. Tasseff B, Coffrin C, Wächter A, Laird C (2019) Exploring benefits of linear solver parallelism on modern nonlinear optimization applications. arXiv preprint arXiv:1909.08104
52. Duff I, Hogg J, Lopez F (2020) A new sparse ldl't solver using a posteriori threshold pivoting. SIAM J Sci Comput 42(2):23–42
53. Axelsson O, Neytcheva M (2003) Preconditioning methods for linear systems arising in constrained optimization problems. Numerical Linear Algebra Appl 10(1–2):3–31
54. Lukšan L, Matonoha C, Vlček J (2005) Interior point methods for large-scale nonlinear programming. Opt Methods Softw 20(4–5):569–582
55. Bergamaschi L, Gondzio J, Zilli G (2004) Preconditioning indefinite systems in interior point methods for optimization. Comput Optim Appl 28(2):149–171
56. Forsgren A, Gill PE, Griffin JD (2007) Iterative solution of augmented systems arising in interior methods. SIAM J Optim 18(2):666–690
57. Curtis FE, Schenk O, Wächter A (2010) An interior-point algorithm for large-scale nonlinear optimization with inexact step computations. SIAM J Sci Comput 32(6):3447–3475
58. Gondzio J (2012) Matrix-free interior point method. Comput Optim Appl 51(2):457–480
59. Curtis FE, Huber J, Schenk O, Wächter A (2012) A note on the implementation of an interior-point algorithm for nonlinear optimization with inexact step computations. Math Program 136(1):209–227
60. Grote MJ, Huber J, Kourounis D, Schenk O (2014) Inexact interior-point method for pde-constrained nonlinear optimization. SIAM J Sci Comput 36(3):1251–1276
61. Cao Y, Seth A, Laird CD (2016) An augmented lagrangian interior-point approach for large-scale nlp problems on graphics processing units. Comput Chem Eng 85:76–83
62. Świrydowicz K, Darve E, Jones W, Maack J, Regev S, Saunders MA, Thomas SJ, Peleš S (2021) Linear solvers for power grid optimization problems: a review of gpu-accelerated linear solvers. Parallel Comput 11:102870
63. Regev S, Chiang NY, Darve E, Petra CG, Saunders MA (2011) A hybrid direct-iterative method for solving kkt linear systems. arXiv preprint arXiv:2110.03636
64. Geffken S, Büskens C (2016) Worhp multi-core interface, parallelisation approaches for an nlp solver. In: Proceedings of the 6th International Conference on Astrodynamics Tools and Techniques, 2016-03-14 - 2016-03-17, Darmstadt, Germany

65. Hu X, Douglas CC, Lumley R, Seo M (2017) Gpu accelerated sequential quadratic programming. In: 2017 16th International Symposium on Distributed Computing and Applications to Business, Engineering and Science (DCABES), pp. 3–6. IEEE
66. Duff IS, Erisman AM, Reid JK (2017) Direct methods for sparse matrices. Oxford University Press, Oxford
67. Gupta A, Joshi M, Kumar V (2001) Wsmp: a high-performance shared and distributed-memory parallel sparse linear equation solver. IBM Research Division RC, 22038
68. Gould NI, Orban D, Toint PL (2015) Cutest: a constrained and unconstrained testing environment with safe threads for mathematical optimization. Comput Optim Appl 60(3):545–557
69. Pommerell C, Fichtner W (1994) Memory aspects and performance of iterative solvers. SIAM J Sci Comput 15(2):460–473
70. Brussino G, Sonnad V (1989) A comparison of direct and preconditioned iterative techniques for sparse, unsymmetric systems of linear equations. Int J Numer Meth Eng 28(4):801–815
71. Keller C, Gould NI, Wathen AJ (2000) Constraint preconditioning for indefinite linear systems. SIAM J Matrix Anal Appl 21(4):1300–1317
72. Dolan ED, Moré JJ, Munson TS (2004) Benchmarking optimization software with cops 3.0. Technical report, Argonne National Lab., Argonne, IL (US)
73. cuSOLVER Reference Guide. https://docs.nvidia.com/cuda/cusolver/index.html. Accessed: 2022-02-02
74. Wächter A, Biegler LT (2006) On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. Math Program 106(1):25–57
75. Balay S, Gropp WD, McInnes LC, Smith BF (1997) Efficient management of parallelism in object oriented numerical software libraries. In: Modern Software Tools in Scientific Computing, pp. 163–202. Birkhäuser Boston, Boston, MA
76. Dalcin LD, Paz RR, Kler PA, Cosimo A (2011) Parallel distributed computing using python. Adv Water Resour 34(9):1124–1139
77. Balay S, Abhyankar S, Adams MF, Benson S, Brown J, Brune P, Buschelman K, Constantinescu E, Dalcin L, Dener A, Eijkhout V (2021) PETSc/TAO users manual. Technical Report ANL-21/39 - Revision 3.16, Argonne National Laboratory
78. Li XS, Demmel JW (2003) Superlu_dist: a scalable distributed-memory sparse direct solver for unsymmetric linear systems. ACM Trans Math Softw(TOMS) 29(2):110–140
79. Petra CG (2019) A memory-distributed quasi-newton solver for nonlinear programming problems with a small number of general constraints. J Parallel Distrib Comput 133:337–348
80. Peles S, Perumalla M, Alam M, Mancinelli AJ, Rutherford RC, Ryan J, Petra CG (2021)Porting the nonlinear optimization library hiop to accelerator-based hardware architectures. Manuscript submitted for publication
81. Beckingsale DA, Burmark J, Hornung R, Jones H, Killian W, Kunen AJ, Pearce O, Robinson P, Ryujin BS, Scogland TR (2019) Raja: Portable performance for large-scale scientific applications. In: 2019 Ieee/acm International Workshop on Performance, Portability and Productivity in Hpc (p3hpc), pp 71–81. IEEE
82. Petra CG, Chiang N-Y (2021) Hiop - user guide version 0.5. Technical Report LLNL-SM-743591, Lawrence Livermore National Laboratory. Accessed 2021-10-07. https://github.com/LLNL/hiop/blob/develop/doc/hiop_usermanual.pdf
83. Chiang N-Y, Zavala VM (2016) Large-scale optimal control of interconnected natural gas and electrical transmission systems. Appl Energy 168:226–235
84. Gondzio J, Sarkissian R (2003) Parallel interior-point solver for structured linear programs. Math Program 96(3):561–584
85. Grothey A, Hogg J, Woodsend K, Colombo M, Gondzio J (2009) A structure conveying parallelizable modeling language for mathematical programming. In: Parallel Scientific Computing and Optimization. Springer. New York, NY, USA
86. Gondzio J, Grothey A (2006) Solving nonlinear financial planning problems with 109 decision variables on massively parallel architectures. WIT Trans Model Simul 21:43
87. Wassel D (2013) Exploring novel designs of nlp solvers: architecture and implementation of worhp. PhD thesis, Universität Bremen
88. Byrd RH, Nocedal J, Waltz RA (2006) Knitro: An integrated package for nonlinear optimization. In: Large-scale Nonlinear Optimization, pp. 35–59. Springer, New York
89. Artelys Knitro User's Manual. https://www.artelys.com/docs/knitro/. Accessed: 2022-02-01

90. Waltz RA, Morales JL, Nocedal J, Orban D (2006) An interior algorithm for nonlinear optimization that combines line search and trust region steps. Math Program 107(3):391–408
91. Byrd RH, Hribar ME, Nocedal J (1999) An interior point algorithm for large-scale nonlinear programming. SIAM J Optim 9(4):877–900
92. Byrd RH, Gould NI, Nocedal J, Waltz RA (2003) An algorithm for nonlinear optimization using linear programming and equality constrained subproblems. Math Program 100(1):27–48
93. Gill PE, Murray W, Saunders MA (2005) Snopt: An sqp algorithm for large-scale constrained optimization. SIAM Rev 47(1):99–131
94. Gill PE, Murray W, Saunders MA, Wong E (2018) User's guide for sqopt 7.7: Software for large-scale linear and quadratic programming. Department of Mathematics, University of California, San Diego, La Jolla, CA, Center for Computational Mathematics Report CCoM, 18–2
95. Gill, P.E., Murray, W., Saunders, M.A., Wong, E.: User's guide for SNOPT 7.7: Software for large-scale nonlinear programming. Center for Computational Mathematics Report CCoM 18-1, Department of Mathematics, University of California, San Diego, La Jolla, CA (2018)
96. GAMS User's Guide. https://www.gams.com/latest/docs/UG_MAIN.html. Accessed: 2022-02-02
97. Fourer R, Gay DM, Kernighan BW (1990) A modeling language for mathematical programming. Manage Sci 36(5):519–554
98. Makhorin A (2000) Modeling language gnu mathprog. Relatório Técnico, Moscow Aviation Institute, 63
99. Bisschop JJ (2022) AIMMS - Optimization Modeling. AIMMS B.V., 2006. AIMMS B.V.. Available at: https://documentation.aimms.com/_downloads/AIMMS_modeling.pdf
100. Lau MS, Yue S-P, Ling KV, Maciejowski JM (2009) A comparison of interior point and active set methods for fpga implementation of model predictive control. In: 2009 European Control Conference (ECC), pp. 156–161. IEEE
101. Cerezo M, Arrasmith A, Babbush R, Benjamin SC, Endo S, Fujii K, McClean JR, Mitarai K, Yuan X, Cincio L et al (2021) Variational quantum algorithms. Nature Reviews. Physics 3(9):625–644
102. Lubasch M, Joo J, Moinier P, Kiffner M, Jaksch D (2020) Variational quantum algorithms for nonlinear problems. Phys Rev A 101(1):010301
103. Kyriienko O, Paine AE, Elfving VE (2021) Solving nonlinear differential equations with differentiable quantum circuits. Phys Rev A 103(5):052416