© 2018
György Mátyásfalvi
ALL RIGHTS RESERVED

OPOS: OBJECT-PARALLEL OPTIMIZATION SOFTWARE

By

GYÖRGY MÁTYÁSFALVI

A dissertation submitted to the

School of Graduate Studies

Rutgers, The State University of New Jersey

In partial fulfillment of the requirements

For the degree of

Doctor of Philosophy

Graduate Program in Operations Research

Written under the direction of

Dr. Jonathan Eckstein

And approved by

_____

_____

_____

_____

_____

New Brunswick, New Jersey

October, 2018

ABSTRACT OF THE DISSERTATION

OPOS: Object-Parallel Optimization Software

By GYÖRGY MÁTYÁSFALVI

Dissertation Director:

Dr. Jonathan Eckstein

This dissertation describes OPOS, a C++ software library and framework for developing massively parallel continuous optimization software. We show that classical iterative optimization algorithms such as gradient projection and augmented Lagrangian methods can be parallelized to run efficiently on distributed memory machines using OPOS.

In Chapter 1 we provide some background on general optimization software and algorithms, as well as parallel software for LASSO and stochastic programming problems.

Chapter 2 introduces OPOS's software development methodology. We start out by describing a set of optimization-domain-specific C++ classes and routines that embody the building blocks of OPOS. The main goal of these classes and routines is to allow the user to code efficient, reusable, maintainable, and readily parallelizable optimization algorithms. OPOS enables the optimization software developer to build optimization algorithm classes that are independent of the problem structure as well as the program's desired execution.

Details of a spectral projected gradient algorithm by Birgin and Martínez and its implementation, OPSPG, are discussed in Chapter 3. Initially, we review the optimization algorithm and OPSPG's code. Next we describe an application to the LASSO problem,

and a novel data distribution technique which achieves an even load balance. Followed by implementation details of objective function and gradient evaluations given our data distribution. We close the chapter by presenting computational results.

Chapter 4 introduces the basic theory behind augmented Lagrangian algorithms and a specific version called `ALGENCAN`, which was developed by Birgin and Martínez. Then we discuss the building blocks of our object-parallel augmented Lagrangian software `OPAL`, which is based on `ALGENCAN`. `OPAL` is applied to solve linear stochastic programming problems. We describe a scenario-based data distribution technique using `PySP`, a python-based modeling software for stochastic programs. This is followed by implementation details of objective function, constraint and gradient evaluations given our data distribution. At the end of the chapter, we demonstrate our computational results.

Chapter 5 summarizes findings of our work and discusses future research opportunities for both LASSO and stochastic programming problems.

Acknowledgements

First and foremost, I want to thank my advisor Dr. Jonathan Eckstein, without whom this thesis would not have come to fruition. I am very grateful for all that he has taught me. His vast knowledge in the field of parallel computing, computational operations research, and nonlinear optimization have fueled my research throughout my time at Rutgers. I am thankful to him for introducing me to the cutting-edge field of parallel optimization software development. I am very excited to continue in this field, which is constantly growing and full of opportunities. I will always be appreciative of this. Finally, I can say that I have been blessed because Dr. Eckstein never once told me in six years that he does not have time for me or that he is busy, he answered all my emails, texts and calls, attended all my talks and always maintained a close relationship with me. It has been an honor getting to know Dr. Eckstein and working with him.

I would also like to express my special appreciation and thanks to Dr. Jean-Paul Watson for all the help that he has provided. Without him, parts of this thesis would not have been possible. Dr. Ivan Rodero's class in parallel computing and his assistance in using supercomputing infrastructures have been extremely important to this research. Dr. José Mario Martinez and Dr. Ernesto G. Birgin deserve my gratitude for helping me use their exceptional work in numerical optimization in my thesis. Additionally, I would like to thank Gabriel Hackebeil for the assistance in developing the `PySP` interface of `OPOS`. Appreciation and thanks is due as well to Dr. Paulo J. S. Silva, Dr. Martin Takáč, and Dr. Hongchao Zhang for the help they have provided in terms of testing various optimization software. I would like to thank all my teachers who have taught me and made me a better student and researcher. During my time at Rutgers I have taken a total of three classes from Dr. Endre Boros, an amazing scholar and a great friend who has supported me in every way he could. Dr. Prékopa, who passed away in September of 2016 was instrumental in getting me to come to Rutgers. He was a true

polymath who lived for his students and will be dearly missed. Thanks to Dr. Farid Alizadeh who is a great lecturer and whose semidefinite programming class was one of the most enjoyable classes I have taken. Thanks to Dr. William E. Hart, Dr. Jennifer L. Troup, Katie Wenzel, Dr. Andrew Bradley and Dr. Carl Laird at Sandia National Laboratories and Dr. James Demmel at UC Berkeley.

Rutgers University is a very large institution; navigating it is not always easy. Thanks to Arleen Verendia, Dr. Gonçalo Filipe, Lauren Tong, Cindy McDermott-Hicks, Jean Dillon, Terry Hart and Clare Smietana, who were always exceptionally helpful in untying institutional knots. Dr. Lynn Agre assisted us with grant-related issues, for which I am very thankful. However, I am most thankful for her friendship and for her kind words on a November morning during my first semester in grad school. When I was feeling overwhelmed with the difficulties of being a first year PhD student, her compassion helped me to regain my confidence and focus on the journey ahead.

I would like to express my appreciation to my friends and colleagues at Rutgers who have made these seven years seem to fly by. Marta Cavaleiro and Ai Kagawa made my life a lot easier because I knew that I could always count on them. My time at Rutgers would not have been the same without my amazing and knowledgeable friends Miriam Diller, Gianluca Gazzola, Dr. Bence Borda, Dr. Aritanan Gruber and Joonhee Lee. I thank Dr. Anh Ninh and Dr. Tomás Vyskocil for their support and for being part of my life. Moreover, I would like to thank Jian Wang, a fellow Rutgers grad student and my godson, for his friendship. I am grateful to Deniz Eskandani, Dr. Mohammad Mehdi Ranjbar and Dr. Javier Rubio Herrero for their friendship and kindness. Thanks to Dr. Mariya Naumova, Dr. Wang Yao, Katie D'Agosta, Dr. Yu Du, Dr. Curtis McGinity, Dr. Emre Yamangil, Peter Mursic, Dr. Jinwook Lee, Dr. Kemal Gürsoy, and Adam Lyko who were amazing colleagues. I am also thankful to Dr. Maria Janowska and Dr. Pawel Janowski, Jessica and Dr. Luis Criales, Dr. Alexandra Villiard, Drs. Lucy and Erick Chastain and other members of the Aquinas Group for their support. To Rutgers Club Swimming presidents Kevin Savage and Kyle Coffey; the best breaststroker on the team, Lorenz Loyola; and Mike Wyka, Gleb Kotousov, Dr. Jens Völker and Ken Niemi: thanks for the many laps in the pool.

I am very grateful to Br. Steven Bolton, Fr. Jeff Calia, Sr. Ellen Kraft and Br. Jude Lasota for their unconditional love and constant encouragement. Grad school would have been much harder without their support. Thanks to friends Victoria Coglianese, Anna Tomasello, Sr. Lorraine Doiron, Alyssa Soto, Katie Cerni, Lizaine Saranglao, Kristine London, Sr. Anna Palka, Marcelle Farhat, Nicole Peters, Melanie Blaszczak, Mike Lewon, Freddy Kurian, John Tomasello, Br. Dan Mohrfeld, Br. José Lim, Matt Frenkel, Rob Allsop, Fr. Keith Cervine, Br. Patrick Reilly, Br. Joe Donovan, Fr. Tom Odorizzi, Fr. Peter Cebulka, Joe Baricelli, Jason Dudziak, Phil Garcia, Ryan O'Shaughnessy, Dan Chedid, Mike Paul, Ryan Mellody and Fr. Gabe Zeis. Additionally, thanks to: St. Peter the Apostle University & Community Parish, Our Lady of Fatima Parish, Sisters of Jesus Our Hope and the Brotherhood of Hope.

I would not be the person who I am today if it were not for my friends and family members at home. I am very grateful to have my nephew Ádám, my brother-in-law Geri and my cousins Virág, Barbara, János, Kristóf, Gergö and their families. My greatest champions from high school are Olivér Nagy, Dávid Liptay, Áron Veress, Csaba Kutassy, Gábor Lukács, Máté Manninger, Anna Réthy, Ádám Szabó, András Téglás, Bence Csókay. Many thanks to them for their never-ending support and friendship. I would like to remember Gergely Szemerey a great friend, who passed away in a tragic accident on April 24$^{\text{th}}$ 2018. A special thanks to Áron; his counsel aided me dearly throughout my time here at Rutgers. I am grateful to Péter Csányi and Petra Nagy for making time to see me during their overseas trips; their friendship means a lot to me.

My sister, my mother, and my father have loved me unconditionally even before I was born and have supported me ever since in every way possible. No sacrifice was ever too big for them, and for this reason I have decided to dedicate this thesis to them. Thank you Anna, Mama and Papa: I love you all very much! I would also like to thank Fr. Attila Mékli who had a profound impact on our lives and continues to bring us closer to each other. Last, but most importantly, I would like to express my eternal gratitude to our Heavenly Father who is the Author of life and makes love possible.

This thesis was funded in part by the National Science Foundation grant number CCF-1115638 and Sandia National Laboratories.

This research used resources from the Rutgers Discovery Informatics Institute (RDI^2), a user facility supported by the Rutgers University Office of Research and Economic Development (ORED) [62].

This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562 [73, 80].

I acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this thesis [70].

Supplemental funding for this project was provided by the Rutgers University - Newark Chancellor's Research Office, and the Rutgers Business School - Newark/New Brunswick Dean's Office.

Mátyásfalvi, György May 2018, Highland Park, New Jersey

Dedication

To my sister, my mother, and my father.

Table of Contents

List of Figures

Chapter 1

Introduction

In this chapter we will start out by briefly exploring the motivation and goals of `OPOS`. This will be followed by a short introduction to optimization software and algorithms for smooth problems. Then we will examine the approaches that have been tried so far, to parallelize existing optimization algorithms, or design novel parallel methods, for large-scale least absolute shrinkage and selection operator (LASSO) problems [71], and stochastic programming (SP) problems [11], which are the test cases of `OPOS`. Finally, we will introduce standard performance measures for parallel software, which we use to evaluate our programs.

1.1 Motivation and Goals

High-Performance Computing (HPC) with massively parallel supercomputers address some of the most challenging computational problems we face today. Existing work on implementing classical optimization methods, with well established convergence properties, on such computer architectures has been somewhat limited. Software that can exploit the computational power of massively parallel supercomputers employ implementation tools such as `MPI` [65], `CUDA` [53], or `OpenMP` [22]. These tools however, require significant amounts of code "clutter" because they are linked to particular classes of hardware, and are organized around relatively low-level operations. The same underlying algorithm may have to be re-implemented multiple times to adapt to different hardware environments or applications, and the resulting code may be difficult to read. There is at present no expressive, portable high-level language that allows implementers to exploit most of the performance of the available hardware, something that would do

for parallel computing what the FORTRAN and C languages accomplished for serial computing in the 1960's through 1980's. In the absence of such a language breakthrough, a natural route to more elegant and portable implementation of parallel algorithms is to use established object-oriented programming concepts.

OPOS is an open source software library for solving general purpose continuous nonlinear optimization problems, both constrained or unconstrained, on massively parallel supercomputers. Implemented in C++, OPOS makes heavy use of object-oriented programming. It includes many classes that allow for a clear, concise, reusable and efficient implementation of optimization methods. In particular, the algorithms are implemented via abstract classes [20]. The result is clear, MATLAB-like yet efficient code that functions as an algorithmic template, readily applicable to various hardware platforms and data representations without the need for modification. For an optimization software to be functional, it needs to have good interfacing capabilities to other scientific software packages, such as modeling languages. OPOS' object-oriented design facilitates such interfacing, as well as application to various problem representations, without the need to change the core source code of the optimization algorithm itself. Within OPOS, one achieves concurrency by parallelizing optimization algorithms' underlying linear algebra operations. In this context, it is essential to employ algorithms that utilize only simple linear algebra operations that are relatively easy to implement in parallel. Hence, the goal of OPOS is to provide the scientific community with reusable, customizable, and efficient implementations of classical optimization algorithms, that have well known convergence properties, and scale well on HPC systems enabling the fast solution of large-scale problems.

1.2 Optimization Software and Algorithms

Solving optimization problems with computers usually involves interaction between an *algebraic modeling language*, or AML [42], processor and a *solver*. AMLs are high-level computer languages that enable the user to formulate decision models using algebraic notation. A *solver* is software that applies an optimization algorithm to a problem instance. Most AML processors interact with optimization solvers by producing files

that are read by the solver. The solver computes objective function, constraint and gradient values based on the input file generated by the AML processor. One popular such input format is the `nl` file format [29, 28], which was originally developed to interface the `AMPL` [25, 24] AML with solvers. Nowadays, many AMLs are capable of generating `nl` files and most solvers accept them as an input. The interface between `nl` based solvers and AMLs is facilitated by the open-source `AMPL` solver library (`ASL`) [28]. `ASL` provides routines for objective function, constraint and gradient computations, where the gradients are obtained via automatic differentiation [34]. This means that the solver developer does not have to invest significant effort into coding objective function and gradient routines but instead can focus on the implementation of the optimization algorithm, which makes `ASL` a very powerful tool for solver development. These techniques work well in a serial or even shared-memory parallel setting, however, they do not carry over readily to distributed memory parallel environments. As a result, presently there is no software library available for distributed memory parallel machines that would have the same capabilities as `ASL`. Therefore, parallel optimization software developers have to take a different approach when it comes to implementing solvers and establishing the interface between them and AMLs.

1.2.1 Popular Optimization Algorithms for Smooth Problems

Most iterative optimization methods for unconstrained smooth problems choose a starting point x_0 then find a direction d_k along which they improve the function value. The main difference between them arises in how they compute d_k. Some well established methods are, gradient methods, which use the objective function's gradient for d_k, Newton and Quasi-Newton methods that apply the inverse of the Hessian, or an approximation of it, to the gradient to obtain d_k, and conjugate gradient methods, which generate conjugate directions d_k using the objective function's gradient. These methods play an important role in constrained optimization algorithms, the focus of this section, as they often show up as subroutines in some shape or form. Consider the constrained

optimization problem

$$\begin{aligned} \min \quad & f(x) \\ \text{s.t.} \quad & x \in X, \end{aligned} \tag{1.1}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is smooth and $X \subset \mathbb{R}^n$ is a nonempty closed convex set. The choice of constrained optimization algorithm for solving (1.1) depends on the structure of X and the problem dimension.

A popular approach to solving (1.1) if the projection operation onto X is simple, for example X is a box, are gradient projection algorithms. Some state-of-the-art serial software implementations of gradient projection algorithms are `VE08` [72], `PORT 3` [27], `LANCELOT` [19], `SPG` [14] and `GENCAN` [13]. If projection onto X requires significant computational effort gradient projection methods become computationally too expensive. In that case the most commonly chosen algorithm to solve (1.1) are interior-point (barrier) methods [43, 81, 2]. Some popular interior-point solvers are `IPOPT` [77], `Knitro` [17], `LOQO` [75], and `MOSEK` [3]. Augmented Lagrangian methods [61] would also be suitable to solve (1.1), however, early implementations of augmented Lagrangian solvers have exhibited inferior convergence compared to interior-point based solvers. Therefore, there was less interest in developing augmented Lagrangian solvers and as a result there are fewer software packages available today. In terms of augmented Lagrangian solvers a serial implementation that has been around the longest is the `LANCELOT` package [19], developed by Conn, Gould, and Toint in the early 1990s. However, in recent years Birgin and Martínez have released a highly sophisticated serial augmented Lagrangian solver called `ALGENCAN` [12]. Nonetheless, superior convergence of interior-point methods comes at a price, since they require the solution of a system of equations at every iteration. Because efficient implementation of general matrix factorization for distributed memory systems is very challenging, there have been few attempts at developing interior-point solvers for distributed memory systems. More precisely, we currently only know of the work that was done by Gondzio and Grothey [32, 33, 31], and Zavala et al. [82], in this area. Depending on their implementation, augmented Lagrangian methods, however, do not necessarily require the direct solution of system of equations, therefore, we believe that they provide a good balance between complexity of

implementation and convergence speed, and as a result are a good choice for developing solvers for distributed systems. Augmented Lagrangian methods are preferentially employed in projects like `TAO` [57] and `ROL` [74] for solving scientific applications modeled by partial differential equations (PDEs), since optimization problems involving PDEs are best handled by algorithms that do not require expensive computations.

1.3 Solving LASSO and SP Problems on Distributed Systems

Presently, no software libraries exist that could determine efficient task allocation, synchronization and communication primitives for arbitrary function evaluations on distributed memory systems. Therefore, all distributed memory optimization software is problem-specific. To test our object-parallel framework we have chosen two problems of interest. The first is the LASSO problem [71], which frequently arises in the field of machine learning. The second is the class of stochastic programming problems [11] that appear in a variety of planning, logistics, and system control problems. Real-world applications often produce problems, belonging to either of these two classes, that have millions of variables or constraints. Therefore, fast solution of these problems requires the computational resources that only distributed systems can offer today.

1.3.1 LASSO

Solving the LASSO problem reduces to the following optimization problem:

$$\min_{x \in \mathbb{R}^n} \tfrac{1}{2} \|Ax - b\|_2^2 + \nu \|x\|_1 , \tag{1.2}$$

where $A \in R^{m \times n}$, $b \in \mathbb{R}^m$, and $\nu > 0$ is a given parameter. The most popular approach for solving these types of problems are distributed coordinate descent methods [8]. Some implementations are, for example, `Hydra` by Richtárik and Takáč [59], or `Grock` by Peng et al. [56]. Coordinate descent methods are well suited for distributed memory computation because these algorithms do not require the computation of the full gradient of the loss function $\frac{1}{2}\|Ax - b\|^2$, which reduces the amount of communication between the processors. However, scalability of these algorithms is data dependent

and they require extensive parameter tuning. This phenomenon may defeat the purpose of parallel computation, which is to obtain solutions fast. While one can argue that coordinate descent methods require less frequent, and less extensive communication (fewer processors communicate) than algorithms that compute the full gradient, most massively parallel supercomputers today have interconnect networks that can execute certain types of communication primitives very efficiently. This assumption does not hold for cluster computing systems, where for example, reduction-type operations may be very slow. In such environments, the coordinate descent approach may be the best one. However, `OPOS` is intended to run on supercomputers, with the assumption that reduction-type communication with relatively small amounts of data is fast and efficient. Therefore, our approach to solving the LASSO problem is to use the nonmonotone spectral projected gradient algorithm by Birgin and Martínez [14] on the smooth equivalent of (3.8), i.e:

$$\begin{aligned} \min_{x^+,x^- \in \mathbb{R}^n} \quad & \tfrac{1}{2} \|A(x^+ - x^-) - b\|_2^2 + \nu \mathbf{1}^\top (x^+ + x^-) \\ \text{s.t.} \quad & x^+, x^- \geq 0. \end{aligned} \tag{1.3}$$

1.3.2 Stochastic Programming

Existing work on parallel solution of stochastic programming problems has concentrated on decomposition methods or barrier methods, which we briefly mentioned earlier, in Section 1.2.1; see for example [32, 33, 31]. Most decomposition methods reduce the original problem, by either removing constraints (nested Benders [52]) or variables (Dantzig-Wolfe [30, 64]), from a restricted master problem (RMP) and at each iteration the RMP is modified by adding constraints or variables generated by an oracle, employing a dual solution of the RMP. A drawback of these approaches is that the data partitioning and communication for these methods on distributed systems would require significant effort, especially for multistage problems with more than two stages. Another way of decomposing stochastic programming problems is by treating each scenario as a subproblem that are solved separately. A popular algorithm for stochastic programming problems that is based on scenario decomposition, called *progressive hedging*, was proposed by Rockafellar and Wets [60]. An efficient master-slave-type parallel

implementation of progressive hedging for distributed memory systems is due to Watson et al. [78]. Currently, we are not aware of distributed-memory implementations of the augmented Lagrangian algorithm applied to multistage stochastic programming problems. The assumption is that an object-parallel augmented Lagrangian implementation should be more scalable than competing methods, like barrier methods, because it does not require parallelization of intricate linear algebra operations such as matrix factorization; instead it is enough to parallelize a limited number of relatively simple operations. In addition, one would expect that the resulting software will exhibit superior tail convergence to decomposition methods.

1.4 Parallel Complexity, Speedup, Efficiency and Scaling

In this section, we will briefly discuss the performance measurements that we use to evaluate our parallel software. Measuring the amount of resources that are required for running a parallel algorithm is more complicated than for a serial algorithm. Parallel computing literature differentiates between the following three measures:

1. Time complexity $T(N, P)$, the time it takes for the program to terminate on a problem instance of size N given P processors.
2. Worst-case communication complexity, the number of messages transmitted during the course of the algorithm.
3. P, the number of processors involved.

Besides complexity, *speedup* and *efficiency* are also important aspects of parallel algorithms. For the analysis of `OPOS`, *speedup* is computed by the ratio

$$S(N, P) = \frac{T(N, 1)}{T(N, P)} \times 100\%, \tag{1.4}$$

where $T(N, P)$ is as defined above in Item 1, and analogously $T(N, 1)$ stands for running the same parallel algorithm on a single processor. One could also compute $S(N, P)$ by replacing $T(N, 1)$ with the time required to solve the problem by the best existing serial algorithm, which would be more indicative of the absolute merit of the parallel algorithm. However, our work is focused on providing a framework for parallelizing

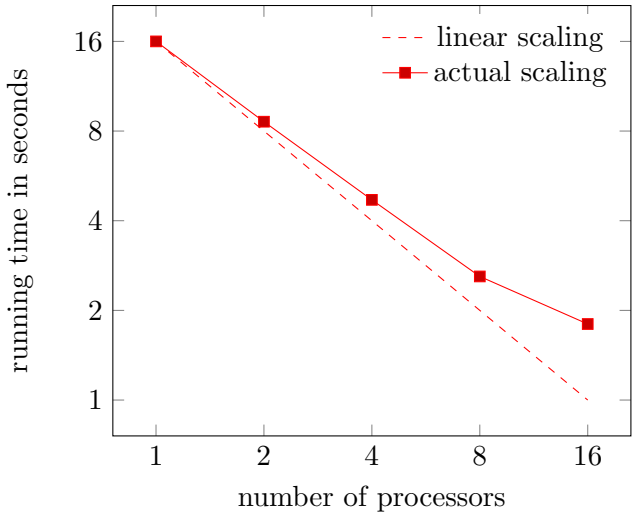


Figure 1.1: Example of a strong scaling graph.

existing optimization algorithms. Therefore, we are more interested in how well our framework achieves the parallelization of these particular algorithms, which is better reflected by the proposed ratio (1.4).

There are two scaling tests that we use to measure the fraction of time that a processor spends on pure computation as opposed to communication or idling. The first is a *strong scaling test*, where we run a fixed-size problem on a varying number of processors to see how the timing of the computation scales with the number of processors. Strong scaling graphs plot the running time of the program as the number of processors are increased while keeping the problem instance constant. As shown in the example in Figure 1.1, strong scaling graphs are essentially log-log plots, where the number of processors appear on the horizontal axis and the program's running time on the vertical axis.

The second is a *weak scaling test*, where we fix the amount of work per processor and compare the execution time over number of processors. Weak scaling graphs plot the running time of the program as both the number of processors and the problem sizes are increased to keep the amount of work per processor constant. Weak scaling is usually depicted on a semi-log plot, where the horizontal axis (number of processors) is the log axis and the vertical axis is linear. An example weak scaling graph is shown

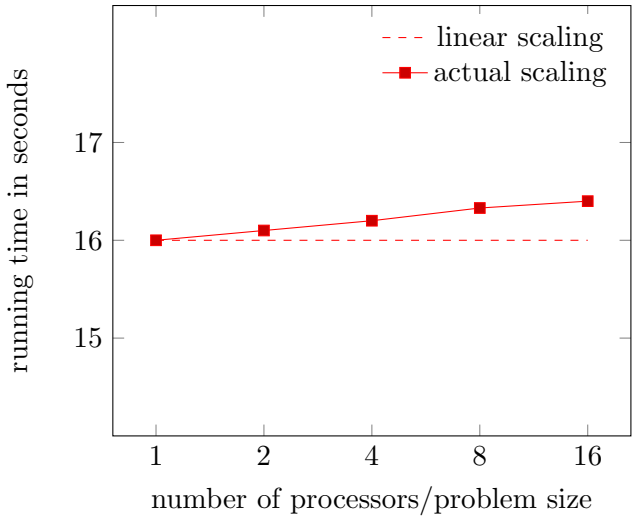


Figure 1.2: Example of a weak scaling graph.

in Figure 1.2, where the horizontal line indicates linear scaling.

Linear or *perfect scaling*, shows the running time if the code maintained 100% parallel efficiency, which could be attained if $S(N, P) = P$. Most parallel codes never attain perfect scaling, and start to shift away from the theoretical line as the number of processors increase. For strong scaling graphs this typically occurs because, while the amount of computation per processor decreases with increasing P, the communication time tends to increase.

Another popular tool to measure parallel code performance is to run in-depth diagnostics at runtime. This can be done using code profilers, such as `PARAVER` [55] or `ARM DDT` [40]. Profilers provide detailed information to the programmer about function calls, such as how much of the runtime was spent inside a particular function, which in turn can be used to determine how much time was spent in functions that are communication-related as opposed to functions that are computation-related.

Chapter 2

Object-Parallel Optimization Software Development

Software developers aim to produce maintainable code for various reasons. For example, maintainable code is easier to read, test and extend. This in turn improves the accuracy of the software and reduces development costs drastically. The key to produce maintainable source code is to reduce complexity. Especially, *cyclomatic complexity* [21], which measures the number of independent paths through a program's source code, essentially the number of branches in the code. The industry standard methodology for reducing complexity is through abstraction. Because abstraction also increases extensibility, one can argue that the right kind of abstraction also allows development of software that will be capable of solving more complicated problems in the future. This thesis proposes an *object-parallel* paradigm for continuous optimization software development, which introduces optimization-domain-specific primitives allowing the programmer a high level of abstraction. These object-parallel abstraction techniques enable the readability, maintainability, reusability, extensibility, and parallelizability of the optimization algorithm's source code.

The main idea behind our approach is to think of optimization algorithms as a series of simple vector manipulations such as vector additions, scalar multiplications, inner products, and norms. In the context of `BLAS` (*Basic Linear Algebra Subprograms*) [47, 15] these operations are referred to as *level 1* routines. For example, consider Algorithm 1, a general gradient descent method for differentiable unconstrained optimization problems of the form

$$\min_{x \in \mathbb{R}^n} f(x). \tag{2.1}$$

Algorithm 1: Gradient descent

Input: $\nabla f,\ x^0 \in \mathbb{R}^n,\ \epsilon_{\text{opt}}$

Output: x', approximate critical point of f

```
1 k = 0
2 while ||∇f(x^k)|| > ε_opt do
3     α^k ← lineSearch()
4     x^{k+1} = x^k − α^k ∇f(x^k)
5     k = k + 1
6 x' = x^k
```

Appropriate step sizes α^k for each iteration are provided by *line search algorithms* [54, 10]. Algorithm 1 generates a sequence of iterates $\left\{x^k\right\}$ converging to an approximate critical point of f. At this point, for the sake of simplicity, we will assume that a line search oracle provides us with the appropriate α^ks. Notice that by abstracting away the gradient computation we have reduced our algorithm to a simple scalar vector multiplication $\alpha^k \nabla f(x^k)$, a vector addition $x^k - \alpha^k \nabla f(x^k)$, and a norm computation $\left\|\nabla f(x^k)\right\|$. In other words, given the right abstraction, optimization algorithms can be reduced to a series of simple vector operations. This abstraction is necessary if we want to make the optimization algorithm independent of f, since if we added the procedure for the gradient computation to our gradient descent algorithm, we would have had to rewrite the algorithm for every single function f to be minimized. Let us assume that we are only interested in minimizing a few specific functions and we decide to include the gradient computation in the code. Even this scenario would greatly reduce maintainability and extensibility of our software. First of all it would increase the cyclomatic complexity of the code by introducing a branch for every function. Secondly, it is not hard to imagine that for some fs the gradient computation ∇f is a complicated procedure, which even if implemented carefully will result in hundreds of lines of code. Hence, it would be much harder to comprehend what the main steps of Algorithm 1 are.

Note that in Algorithm 1 we actually went one step further. We did not only abstract away the gradient computation but also the implementation details of the vector operations. If we did not do so then we would have to change Algorithm 1, for

example, in the following way:

Algorithm 2: Gradient descent

Input: $\nabla f,\ x^0 \in \mathbb{R}^n,\ \epsilon_{\text{opt}},\ n$

Output: x', approximate critical point of f

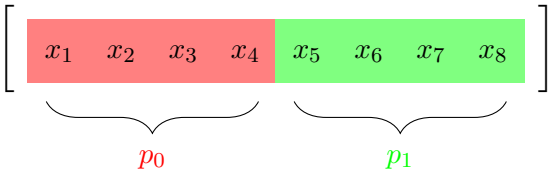
```
1  k = 0
2  e = 0
3  for i = 0; i < n; ++i do
4      e += (∇f(x^k)_i)^2
5  while e > ε_opt do
6      α^k ← lineSearch()
7      for i = 0; i < n; ++i do
8          x_i^{k+1} = x_i^k − α^k ∇f(x^k)_i
9      e = 0
10     for i = 0; i < n; ++i do
11         e += (∇f(x^{k+1})_i)^2
12     e = √e
13     k = k + 1
14 x' = x^k
```

Hence, we can conclude that by abstracting away the details of vector operations and optimization problem, our optimization algorithm's code becomes relatively maintainable and extensible. Furthermore, this kind of abstraction can be used to achieve SPMD (single program multiple data) parallelism [23] without the need to change the optimization algorithm's main code.

In the SPMD paradigm, each processor executes the same lines of code but the underlying data are different. To see how our abstraction scheme allows one to maintain the same code in the SPMD setting, consider for example that vector x has the following coordinates:

$$\begin{bmatrix} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 \end{bmatrix}$$

Assume that x is distributed across two processors p_0 and p_1 in such a way that p_0 owns the first four elements of x and p_1 owns the last four elements, i.e.:



This data distribution would change Algorithm 2 in the following way:

Algorithm 3: Parallel gradient descent

Input: ∇f, $x^0 \in \mathbb{R}^n$, ϵ_{opt}, n_{local}, `myId`

Output: x', local minimizer of f

$k = 0$
$e = e_s = e_r = 0$
for $i = \textit{myId} \times n_{local}$; $i < n_{local} + \textit{myId} \times n_{local}$; $++i$ **do**
    $e_s \mathrel{+}= \left(\nabla f(x^k)_i\right)^2$
if $\textit{myId} = 0$ **then**
    `send` $e_s \rightarrow p_1$
    `receive` $e_r \leftarrow p_1$
else
    `send` $e_s \rightarrow p_0$
    `receive` $e_r \leftarrow p_0$
$e = \sqrt{e_s + e_r}$
while $e > \epsilon_{opt}$ **do**
    $\alpha^k \leftarrow$ `lineSearch()`
    for $i = \textit{myId} \times n_{local}$; $i < n_{local} + \textit{myId} \times n_{local}$; $++i$ **do**
        $x_i^{k+1} = x_i^k - \alpha^k \nabla f(x^k)_i$
    $e = e_s = e_r = 0$
    for $i = \textit{myId} \times n_{local}$; $i < n_{local} + \textit{myId} \times n_{local}$; $++i$ **do**
        $e_s \mathrel{+}= \left(\nabla f(x^{k+1})_i\right)^2$
    if $\textit{myId} = 0$ **then**
        `send` $e_s \rightarrow p_1$
        `receive` $e_r \leftarrow p_1$
    else
        `send` $e_s \rightarrow p_0$
        `receive` $e_r \leftarrow p_0$
    $e = \sqrt{e_s + e_r}$
    $k = k + 1$
$x' = x^k$

Notice how the additional lines of code in the parallel version of the gradient descent

algorithm are all related to the vector addition procedure or the norm computation, which are both vector operations. But in a more abstract representation as in Algorithm 1 the details of all vector operations are independent from the optimization algorithm. Applying the same line of thinking to the gradient computation would also yield that no additional lines of code need to be added to the parallel version of the optimization algorithm's code if the gradient computation is abstracted away from the optimization algorithm. Therefore, we can conclude that Algorithm 1 would indeed remain unchanged in an SPMD-type parallel setting.

`C++` [69], an object-oriented programming language, has the capability to implement abstraction schemes without sacrificing too much performance. In `C++`, abstraction is achieved through the use of *abstract classes*. These classes allow the programmer to create objects that hide implementation details of various computations and therefore, can be used to abstract away computations related to the optimization problem and vector operations. In the next section, we will introduce some basics of `C++` classes, following that we will describe how we use `C++` techniques to create an abstraction scheme that allows the programmer to code optimization algorithms in a similar manner to Algorithm 1.

2.1 `C++` Classes

A `C++` class is a user-defined data type. By specifying such a data type we instruct the compiler how to allocate a particular piece of storage, and also how to manipulate that storage. In our case, *problem classes* will determine how an optimization problem's data are stored and how to compute various quantities related to the problem, such as objective function values, gradients, or constraint values. Similarly, *vector classes* will specify the storage of vectors and various vector operations that are used in optimization algorithms. *Objects* in `C++` are variables whose type is a class. The next section will provide some examples of `C++` classes and objects.

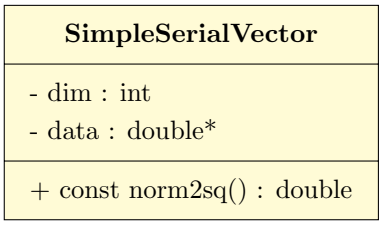
```
#ifndef SIMPLESERIALVECTOR_H_
#define SIMPLESERIALVECTOR_H_
#include "SimpleAbstractVector.h"

class SimpleSerialVector : public SimpleAbstractVector {
  int dim;
  double* data;
public:
  double norm2sq() const {
    double e = 0.0;
    for (int i=0; i<dim; ++i)
      e += data[i] * data[i];
    return e;
  }
};
#endif /* SIMPLESERIALVECTOR_H_ */
```

Figure 2.1: Class definition of `SimpleSerialVector`.

2.1.1 Examples of C++ Classes and Objects

In this section we will introduce simple C++ classes that are similar in nature to the ones that are used in OPOS, but abbreviated for reasons of space and clarity. These classes are not part of OPOS, and they only serve the purpose to establish the basic design ideas behind OPOS and various concepts related to C++ classes. Throughout this illustration we will use *UML diagrams* [26], which are the industry standard notation used to concisely describe C++ classes and their relationships. Note that in our examples we will omit specifying constructors and destructors. Take the class named `SimpleSerialVector` shown in Figure 2.1. This `SimpleSerialVector` class has three members, `dim`, the dimension of the vector, `data`, a pointer to the storage that holds the vector values, and a method `norm2sq()`, which returns the square of the euclidean norm. The UML class diagram for `SimpleSerialVector` has the following form:

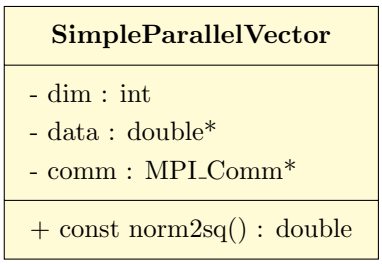


```
#ifndef SIMPLEPARALLELVECTOR_H_
#define SIMPLEPARALLELVECTOR_H_
#include "SimpleAbstractVector.h"
#include "mpi.h"

class SimpleParallelVector : public SimpleAbstractVector {
  int dim;
  double* data;
  MPI_Comm* comm;
public:
  double norm2sq() const {
    double local = 0.0, e = 0.0;
    for (int i=0; i<dim; ++i)
      local += data[i] * data[i];
    MPI_Allreduce(&local, &e, 1, MPI_DOUBLE, MPI_SUM, *comm
      ↪ );
    return e;
  }
};
#endif /* SIMPLEPARALLELVECTOR_H_ */
```

Figure 2.2: Class definition of `SimpleParallelVector`

The minus sign − in front of `dim` and `data` identify them as *private members*, meaning that only class member functions and friend methods have access to them. The plus sign + in front of `norm2sq()` specifies that it is a *public member*, which means that any method in the code has access to it. Next, let us define another class named `SimpleParallelVector` (Figure 2.2). The UML class diagram is as follows:



Notice that both `SimpleParallelVector` and `SimpleSerialVector` are derived classes of `SimpleAbstractVector`. We know this from the following lines at the beginning of the class definition:

```
class SimpleSerialVector : public SimpleAbstractVector
```

```
#ifndef SIMPLEABSTRACTVECTOR_H_
#define SIMPLEABSTRACTVECTOR_H_

class SimpleAbstractVector {

public:
  virtual double norm2sq() const = 0;
};
#endif /* SIMPLEABSTRACTVECTOR_H_ */
```

Figure 2.3: Class definition of `SimpleAbstractVector`

```
class SimpleParallelVector : public SimpleAbstractVector
```

This means that both `SimpleParallelVector` and `SimpleSerialVector` inherit the public members of `SimpleAbstractVector` class and have access to its *protected members*. Protected members are members that are accessible only by member functions of the class, its derived classes, and friend methods, we will see an example of protected members in Section 2.2.2. `SimpleAbstractVector` is defined in Figure 2.3. Here, `SimpleAbstractVector` only has one member `norm2sq()`, which is a *pure virtual* method. If any method of a class is pure virtual then that class becomes an *abstract class*. Abstract classes in `C++` are used to implement a unified interface for all of their derived classes. A unified interface is a necessary condition for our abstraction scheme, since we do not want to have different function calls, for example, for $\|x\|$, depending on how we implement the norm computation. Abstract classes in `C++` guarantee that every non-abstract-derived class will have an implementation of all pure virtual methods. In the UML diagram we indicate that a class is an abstract class, or that a function is pure virtual by setting its font to *italics*.

The diagram in Figure 2.4 illustrates a package called *Simple Vector Classes*, which contains all the vector classes we have mentioned so far. It also shows that `SimpleParallelVector` and `SimpleSerialVector` are derived classes of `SimpleAbstractVector`. This relationship between the classes is depicted by the arrows connecting them. The arrow always points to the class from which the data elements and member functions are

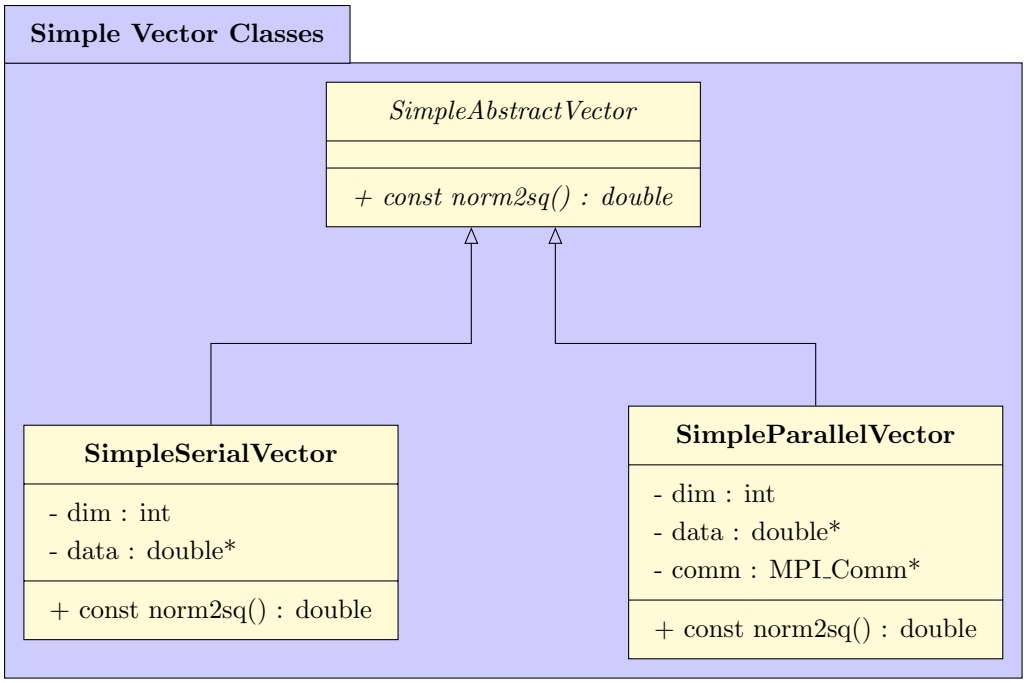


Figure 2.4: *Simple Vector Classes*, an example package.

inherited. In our case, this is `SimpleAbstractVector`. Finally, we note that UML diagram does not necessarily have to include details about the members of each individual class, in which case our diagram would take the form depicted in Figure 2.5.

2.1.2 Summary of UML Notation

In this section we provide a summary of the UML notation that we use throughout this thesis.

Abstract class: Class *Name* in italics

Public member: +

Protected member: #

Private member: −

Pure virtual function: *italics*

An association: Object `A` is aware of object `B`. For example, `A` contains a pointer or reference to `B`.

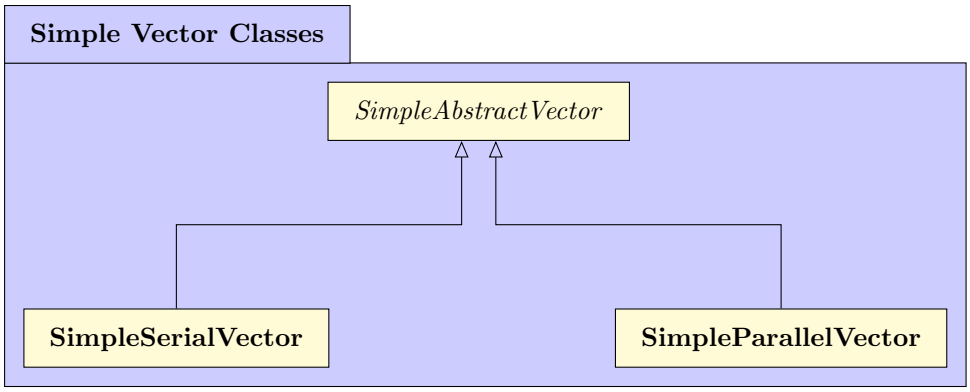
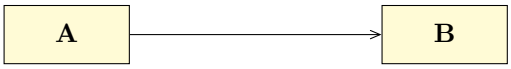
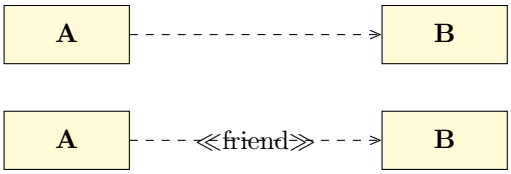


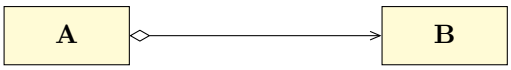
Figure 2.5: Simplified UML diagram of *Simple Vector Classes*.



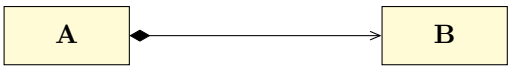
A dependency: `A` depends on `B` if `B` is a parameter variable or local variable of a method of `A`. Also includes friend relationships.



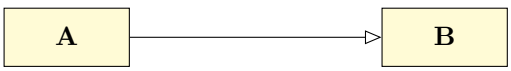
An aggregation: `A` is a container of object(s) of `B` or pointer(s) to `B`. However, objects of `B` do not have a strong life cycle dependency on `A`. In other words `A` can be destroyed without destroying `B`.



A composition: A is a container of object(s) of `B` or pointer(s) to `B`. `A` and object(s) of `B` have a strong life cycle dependency. In other words destroying `A` will also destroy `B`(s).

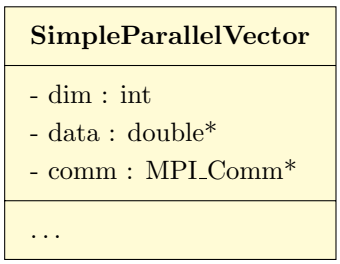


An inheritance: Class `A` is derived from class `B`. Class `A` is of type `B`.



2.2 OPOS Classes

In OPOS we create a unified interface to the optimization problem and vector operations through the abstract classes AbstractProblem and AbstractVector. Optimization algorithms in OPOS are implemented using these classes. The AbstractVector and AbstractProblem classes define a set of necessary optimization-specific primitives that will allow us to achieve the desired level of abstraction for optimization algorithms. For example, we can use the same function calls for ∇f independent of what type of function we aim to minimize, or for $\|x\|$ independent of how we implement the norm computation. In the following sections we will go over the implementation details of the AbstractVector and AbstractProblem classes and various other classes that are the fundamental building blocks of OPOS. In order to keep the description of the structure of OPOS understandable, we will not list all the details of the classes, and will omit mentioning most of the public members. Omission will be denoted by . . . . For example, if we decide to omit the public member norm2sq() from SimpleParallelVector the class table will look as follows:



UML diagrams describing relationships between the classes of OPOS will be kept concise by only including the minimal number of classes necessary to capture the general structure of our software.

2.2.1 Vector Classes of OPOS

AbstractVector is an abstract class that through its pure virtual methods creates a general interface that abstracts both the representation of vectors and the methods used to perform simple mathematical manipulations of vectors such as addition, scaling, or inner product. Methods of AbstractVector can be categorized as follows:

Clone methods: create a copy of the vector-object calling the clone method. Depending on which cloning method is called the individual coefficients of the vector can be copied from the original object, set to all zeros, or all ones.

Vector operations: include inner products, norm computations, min/max of coefficients of a vector, componentwise min/max of two vectors, and componentwise product of two vectors.

Operators: include overloaded operators,

[ ] returns a coefficient of a vector,

+ adds two vectors,

− subtracts two vectors,

× scales a vector by a scalar,

= sets the left hand side equal to the right hand side,

+ = adds the right hand side to the left hand side, and

− = subtracts the right hand side from the left hand side.

Our object-parallel framework requires that all vector classes have the ability to create copies of themselves. In particular, each class is required to have the following clone methods:

`clone()`: Creates a copy of the vector.

`zeroClone()`: Creates a copy of the vector, except that all coefficients are set to zero.

`oneClone()`: Creates a copy of the vector, except that all coefficients are set to one

Similarly, we also require that all vector classes are capable of performing a series of vector operations, and that standard algebraic operators are applicable to them. We have developed various implementations of vector classes, which are all derived from `AbstractVector`, these include:

`SerialVector`: a serial implementation using `C++` STL containers [18, 68]; all methods are custom coded.

`BareSerialVector`: a serial implementation using `C++ double` arrays; all methods are custom coded.

`BlasVector`: a serial implementation using `C++ double` arrays; when applicable, methods call BLAS routines for numerical operations.

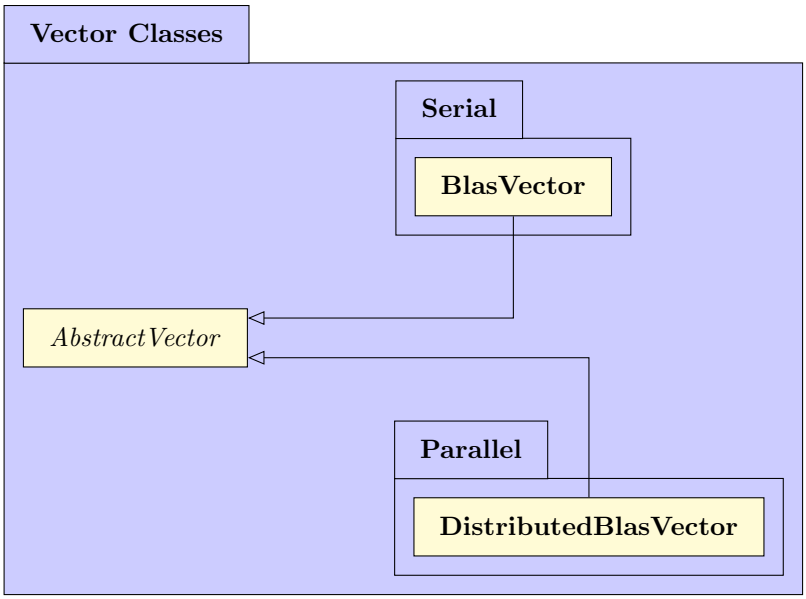


Figure 2.6: Inheritance graph of some of `OPOS`' vector classes.

`DistributedBlasVector`**:** a parallel version of `BlasVector`, that allows for variables to be replicated across various processors.

`EpetraVector`**:** encapsulates vectors represented by the `Epetra` [39] parallel linear algebra system. Vectors may be distributed among multiple processors, and `Epetra`'s `BLAS` is used for numerical operations.

Other classes may be derived from `AbstractVector` or from its derived classes. For example, one could derive a class from `EpetraVector` to represent vectors with specialized kinds of parallel data layouts. Our tests indicate that using `BlasVector` for serial runs and `DistributedBlasVector` for parallel runs are the most efficient. Therefore, in further discussions we will only include `BlasVector` and `DistributedBlasVector` classes. Their inheritance graph is depicted in Figure 2.6.

Replication of Variables

Take for example, matrix M with 7 columns and 8 rows, where non-zero entries are indicated by $*$:

$$M = \begin{bmatrix} * & * & * & * & & & & \\ * & * & * & * & & & & \\ * & * & * & * & & & & \\ * & * & * & * & & & & \\ & & & * & * & * & * \\ & & & * & * & * & * \\ & & & * & * & * & * \\ & & & * & * & * & * \end{bmatrix} \tag{2.2}$$

Let us assume that our optimization algorithm has to apply the linear map M to the decision variables x at every iteration.

$$Mx = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 \end{bmatrix} \times \begin{bmatrix} * & * & * & * & & & & \\ * & * & * & * & & & & \\ * & * & * & * & & & & \\ * & * & * & * & & & & \\ & & & * & * & * & * \\ & & & * & * & * & * \\ & & & * & * & * & * \\ & & & * & * & * & * \end{bmatrix} \tag{2.3}$$

This computation is fairly straightforward in a serial setting. However, in a distributed memory parallel setting the programmer may be required to partition matrix M across all processors. If our distributed memory system has two processing units (p_0, p_1), we need to partition M into two parts. Given M as in (2.2), we could partition it by assigning the first four rows to processor p_0 and the last four rows to p_1 as shown in Figure 2.7. For the proposed partition, Figure 2.8 illustrates the local matrix-vector multiplies that each processor needs to compute. In order for these local matrix-vector multiplies to produce a result equivalent to Mx, our software has to ensure that we store a copy of the coefficient x_4 in both processors p_0 and p_1. This means that given our data partitioning scheme for matrix M we have to adjust the representation of vector x by replicating coefficient x_4. Figure 2.9 depicts the representation of x in this distributed memory setting.

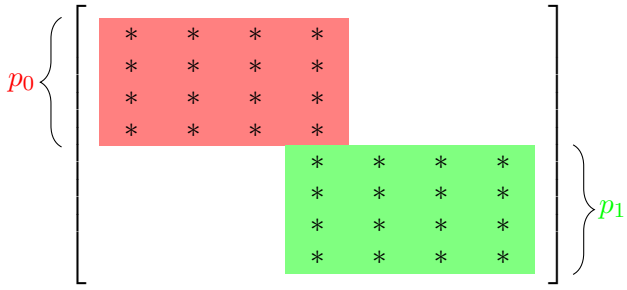


Figure 2.7: Partition of matrix M across two processors.

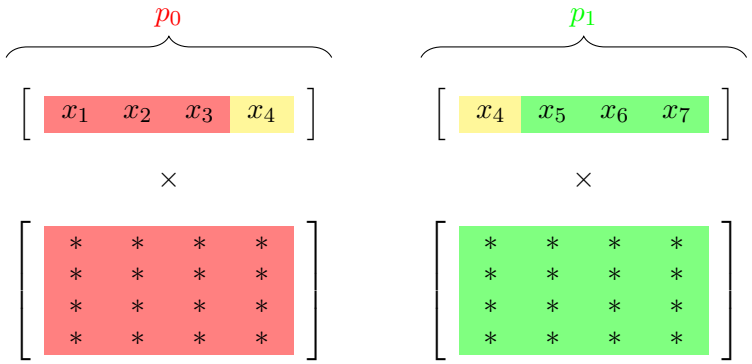


Figure 2.8: Local matrix-vector multiplies Mx.

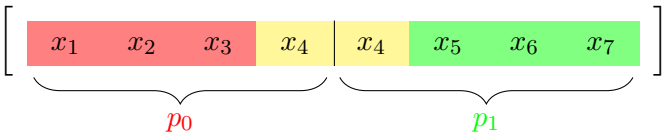


Figure 2.9: Representation of x in a distributed memory setting.

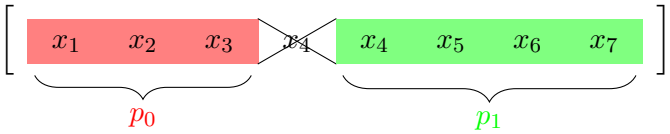


Figure 2.10: Contribution of coefficients to distributed norm computation.

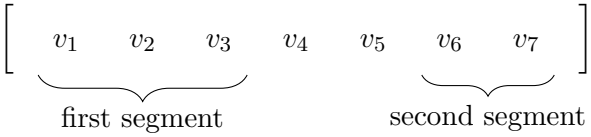


Figure 2.11: Segments of v contributing to distributed norm computation.

This replication does not have an effect on most vector operations in an SPMD-type parallel setting. However, inner products and some norm computations need to be modified in order to produce the correct results. We will address this difficulty by allowing the user to specify which local segments of each processor's vector should contribute to inner products and norms. For the vector x (Figure 2.10), we could say that processor p_0 will only contribute the segment with coefficients $1-3$ to the norm computation, and p_1 the segment with coefficients $4-7$. We can describe arbitrary segment patterns with the following information:

`numSegs`: Indicating the number of segments.

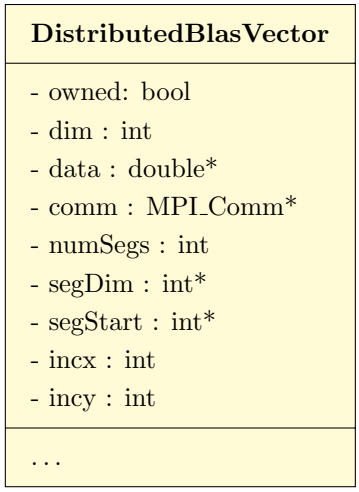
`segDim`: An array containing the length of each segment.

`segStart`: An array containing the position of the first element of each segment.

Therefore, all our parallel vector classes will have members `numSegs`, `segDim`, and `segStart` that will determine which segments ought to contribute to inner products and norm computations. For example, if we would like two segments of vector v to contribute to an inner product, where the first segment starts at position 1 and has length 3, and the second starts at position 6 and has length 2 (Figure 2.11). We would have the following `numSegs`, `segDim`, and `segStart` values: `numSegs`$= 2$, `segDim`$= [3, 2]$, and `segStart`$= [1, 6]$.

`DistributedBlasVector`

For brevity, we will only illustrate one of our vector classes, `DistributedBlasVector`, in more detail. `DistributedBlasVector` is derived from `AbstractVector`, as all of our vector classes, and has the following private members:



Member `dim` determines the length of the vector and `data` points to the storage holding the coefficients. If a vector upon destruction is not responsible for releasing the `data` array `owned` is set to `false` otherwise to `true`. As we have seen earlier `numSegs`, `segDim`, and `segStart` determine, which coefficients will be included in the inner product and norm computations. The object `comm` is specific to `MPI`, it determines which processes are involved in the communication, and `incx` and `incy` are specific to `BLAS`. We omit listing all the public members of `DistributedBlasVector` for the sake of brevity. Nonetheless, we will illustrate the workings of segments by showing the source code implementing the euclidean norm square computation in `DistributedBlasVector` in Figure 2.12.

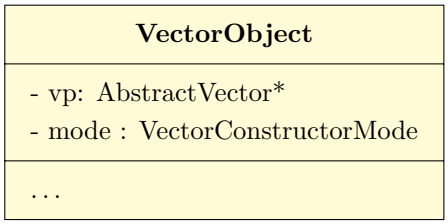
Enhancing Code Readability through the `VectorObject` Class

To avoid cluttering algebraic expression code with excessive pointer dereferencing, we defined a class, called `VectorObject`, which essentially encapsulates a pointer to an `AbstractVector`. `VectorObject` has two private members, an `AbstractVector*` pointer `vp`, and an enumerated type `mode`. The enumeration `VectorConstructorMode` consists

```
double norm2sq() const {
  double localProd = 0.0, prod = 0.0;
  for(int i=0; i<numSegs; ++i) {
    localProd += cblas_ddot(segDim[i], data+segStart[i],
      ↪ incx, data+segStart[i], incy);
  }
  MPI_Allreduce(&localProd, &prod, 1, MPI_DOUBLE, MPI_SUM,
    ↪ *comm);
  return prod;
}
```

Figure 2.12: `DistributedBlasVector`'s `norm2sq()` method.

of named constants that are used by `VectorObject`'s constructors and the destructor.



The enumerated type `mode` specifies whether, upon destruction of the `VectorObject`, its destructor should also destroy the vector `vp` points to. This is useful because it allows multiple `VectorObjects` to point to the same vector, and we can specify which one will be responsible for deleting it. This setup permits us to optimize memory usage, which is especially important if a program is dealing with large vectors. The `AbstractVector` class contains abstract methods for common operations such as inner products and assignment, and `VectorObject` contains corresponding "pass-through" methods to access them. For example, if `x` and `y` are `VectorObjects` encapsulating pointers to the same `AbstractVector`-derived class, one may write the `C++` expression `x.inner(y)` to denote $\langle x, y \rangle$. As shown below, evaluating this simple expression invokes the `inner` method of the underlying `AbstractVector`-derived class:

```
double inner(const VectorObject& y) const {
 return vp->inner(y.getAv());
}
```

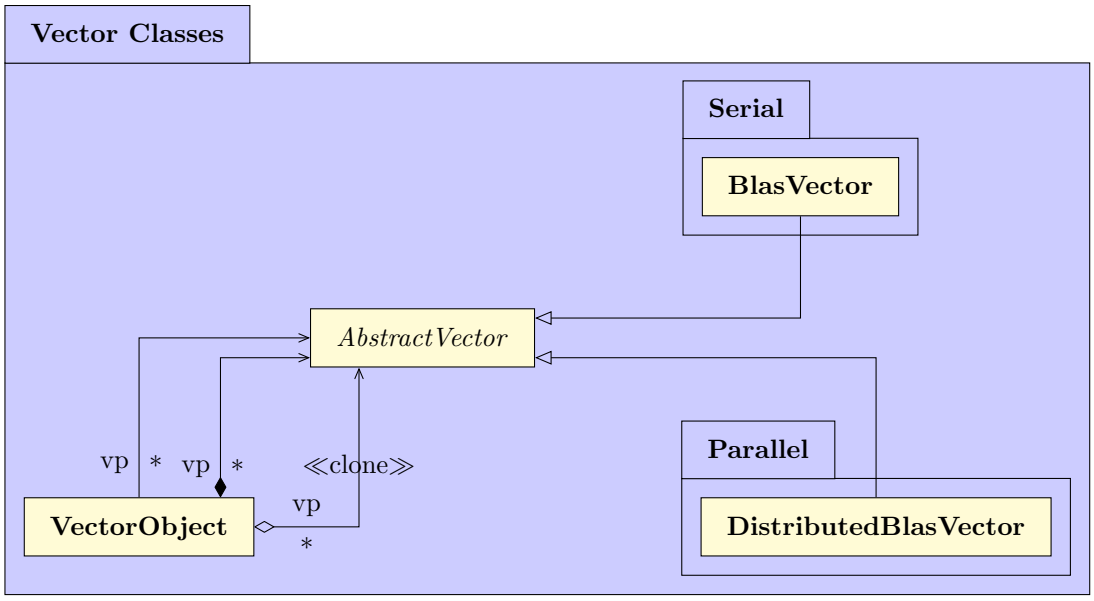


Figure 2.13: Relationships between the `VectorObject` class and other vector classes.

Cloning routines are also typically accessed through the `VectorObject` wrappers, and allow an algorithm to properly allocate all the vector storage it needs cloning by single "template" vector object passed to it. Because this construct allows us to write more easily maintainable and parallelizable code, it too is a fundamental building block of our object-parallel framework. The relationship between `VectorObject` class and other vector classes is illustrated in the UML chart in Figure 2.13, which only includes the `AbstractVector`-derived classes `BlasVector` and `DistributedBlasVector` in order to keep the chart simple.

Symbolic Temporaries: Efficient Operator Overloading through Delayed Evaluation

One of the key goals of our abstraction scheme is also to be able to write a limited range of vector-related expressions with the same simplicity as a prototyping environment like MATLAB, and yet still retain most of the performance and control available through `C++`. For example, suppose an algorithm contains the calculation $z = w + \alpha x - \beta y$, where w, x, y, and z are vectors and α and β are scalars. For simplicity and clarity, we would like to be able to translate this assignment into the `C++` statement

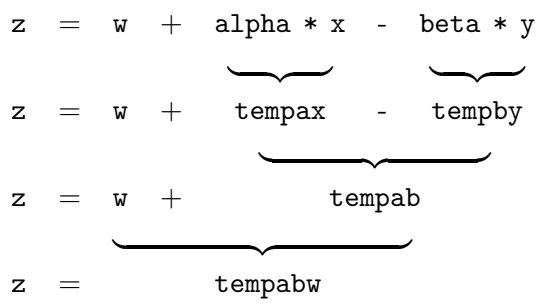


Figure 2.14: Creation of temporaries for $z = w + \alpha x - \beta y$.

```
z = w + alpha*x - beta*y,
```

where `w`, `x`, `y`, and `z` are all `VectorObject`s encapsulating pointers to the same `AbstractVector`-derived type and `alpha` and `beta` are of type `double`. If we were to define overloaded operators in the conventional `C++` manner, however, the resulting compiler temporary objects would cause excessive memory allocations and access operations. In the case of the expression above, for example, the standard `C++` approach to operator overloading would result in the following operations, as shown in Figure 2.14:

1. An `operator*(double&,VectorObject&)` method would allocate new memory for a similar `VectorObject` and fill it with the values of `x`, scaled by `alpha`, `tempax` in the figure.

2. A similar operation would create a temporary object containing `beta*y`, `tempby` in the figure.

3. An `operator-(VectorObject&,VectorObject&)` method would be invoked to calculate `alpha*x - beta*y` from the preceding two temporary objects and place the results in a third temporary `VectorObject`, `tempab` in the figure.

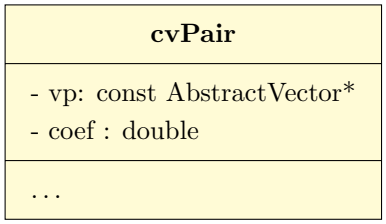
4. An `operator+(VectorObject&,VectorObject&)` method would be invoked to calculate the sum of `w` and the third temporary object, creating a fourth temporary `VectorObject`, `tempabw` in the figure.

5. Finally, an `operator=(VectorObject&,VectorObject&)` method would be invoked to copy the contents of the last temporary into `z`.

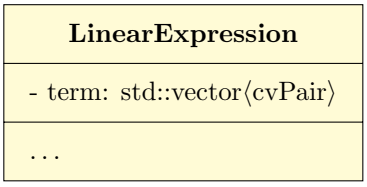
For relatively compact objects — for example, a representation of a complex number consisting of two `double`s — the overhead resulting from such operations might

not be significant or could be reduced or perhaps eliminated by "downstream" code optimization steps within the `C++` compiler. In situations in which each vector consists of many megabytes of data, possibly spread over different processor memory spaces, there will be a significant speed and memory penalty for executing the calculation in this manner. Specifically, it will allocate four unnecessary temporary objects, each of which will be written and read one time. If we were implementing this assignment calculation using BLAS routines, it could be coded with just two DAXPY calls and no temporary objects, although at the cost of some opaqueness in the code. In modern computer architectures, memory access operations and cache misses can be much more time-consuming than arithmetic calculations, so it is imperative to economize on memory operations if one is attempting to optimize performance. Furthermore, if the vectors `w`, `x`, `y`, and `z` are extremely large, allocating temporary objects of the same size might strain or exceed available system memory. For these reasons, the standard approach to operator overloading is not a viable option.

Instead, we have developed an alternative, "delayed evaluation" approach to operator overloading, involving two auxiliary classes called `cvPair` and `LinearExpression`. The `cvPair` class encapsulates a pointer to an `AbstractVector` and a `double`.



Our `LinearExpression` class is essentially an STL vector of `cvPair`s of the form (α_i, p_i), where α_i is the `double` and p_i is the pointer to the `AbstractVector`.



`LinearExpression`'s methods include a variety of simple numerical operators. For example, the addition operator `operator+(const VectorObject& right)` adds a `VectorObject` to a `LinearExpression` by appending an entry to the `cvPair` vector `term`.

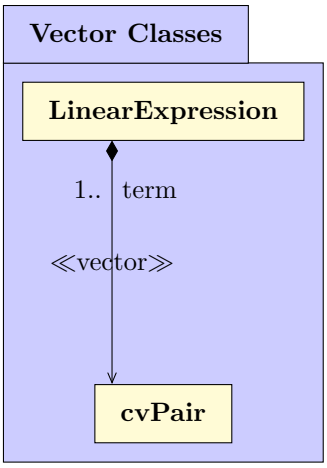


Figure 2.15: Relationship between `LinearExpression` and `cvPair`.

The relationship between the classes `cvPair` and `LinearExpression` is shown in Figure 2.15.

In this approach, all the overloaded operators involving `VectorObject`s, except those performing assignment, create `LinearExpression` objects. The result of evaluating the entire right-hand side of an assignment is a `LinearExpression` of the form $\big((\alpha_1, p_1), \ldots, (\alpha_\ell, p_\ell)\big)$. Then, an overloaded assignment operator, most commonly of the form `operator=(const LinearExpression& right)` has the job of calculating the vector $\sum_{i=1}^{\ell} \alpha_i(*p_i)$ (here, the "$*$" denotes pointer dereferencing). Consider the same assignment expression `z = w + alpha*x - beta*y` discussed above: with our overloaded `VectorObject` and `LinearExpression` methods, the actual evaluation of this expression is as follows:

1. The `operator*(const double left, const VectorObject& right)` method creates a temporary object of type `LinearExpression`, containing `(alpha,&x)` (that is, the value of `alpha` and the address of `x`).

2. Another application of `operator*(const double left, const VectorObject& right)` creates another temporary `LinearExpression`, containing `(beta,&y)`.

3. The method `operator-(const LinearExpression& left, const LinearExpression& right)` combines these two linear expressions and creates a third temporary `LinearExpression`, containing $\big($`(alpha,&x) (-beta,&y)`$\big)$.

4. The method `operator+(const VectorObject& left, const LinearExpression& right)` appends an additional pair to this `LinearExpression`, producing a `LinearExpression` of the form

$$\big(\texttt{(1,\&w)},\texttt{(alpha,\&x)},\texttt{(-beta,\&y)}\big).$$

5. The `VectorObject` method `operator=(LinearExpression& right)` computes the value of this last `LinearExpression` and places it in the memory dedicated to `z`. This task is accomplished by calling wrapper methods within the `VectorObject z`, which in turn call virtual methods of the `AbstractVector` class. Thus, how the assignment calculation is implemented will depend on exactly what kind of `AbstractVector`-derived representations the `VectorObject`s are encapsulating. If they are of type `BlasVector`, for example, the evaluation would indeed reduce to two DAXPY operations.

Our procedure generates exactly the same number of temporary objects as the conventional approach, but they are extremely compact objects, basically small arrays of pairs each consisting of 16 bytes (assuming a system with 64-bit addresses). We call these objects *symbolic temporaries* because each symbolically encodes the linear combinations of vectors that needs to be calculated. These objects are so small that they are likely to fit in the fastest level of processor cache, and the overhead involved in manipulating them is therefore likely to be negligible for large-scale applications. We may also call the technique *delayed evaluation* because it only performs arithmetic when it processes the assignment operation. At this point, the entire expression to be calculated is known and its evaluation can be optimized.

In addition to the `=` operator, we also overload `C++`'s `+=` and `-=` operators in similar ways, allowing them to be used efficiently on expressions involving `VectorObject`s. We also provide versions of the `inner` method for all four possible combinations of `VectorObject` and `LinearExpression` arguments. For example, a mathematical expression of the form $\langle x, y + \alpha z \rangle$ would be rendered using our `VectorObject` class as `x.inner(y + alpha*z)`. At run time, this expression would end up invoking an inner-product evaluation method in the `AbstractVector`-derived class encapsulated by `x` on a `LinearExpression` containing $\big(\texttt{(1,\&y)},\texttt{(alpha,\&z)}\big)$. Depending on how the encapsulated `AbstractVector`-derived class is implemented, this calculation might be done

very efficiently, without allocating significant temporary memory. Such inner-product calculations are the only circumstances in which we might evaluate the result of a `LinearExpression` before encountering an assignment operator.

It is certainly possible to extend our symbolic-temporary techniques to more complicated expressions than simple linear combinations of vectors. This effort might be worthwhile, although it could conceivably add significant complexity. The basic functionality we have implemented at this point is sufficient to create readable yet efficient code for the kind of optimization algorithms we have worked with so far.

Sufficiently powerful parallel-environment-targeted compilers could in principle produce more efficient code than our techniques, because the analysis of arithmetic expressions could be performed completely and optimized just once at compile time, rather than having some aspects of expression analysis extensively repeated at run time, as can effectively happen with our approach. However, such compilers do not presently exist, nor even does an active, accepted language standard such compilers could implement. The HPF standard [45] appears to be the closest the high-performance computing community has come to such a standard, but HPF compilers are not commonly available, and the language has some limitations due to its FORTRAN heritage. Because our run-time temporary objects are so small, the time required to manipulate them is likely to be negligible in the kind of large-scale applications that realistically require large-scale parallel computing. Finally, while the effort required to develop our symbolic temporary classes was considerable, it was minuscule compared to the effort required to develop a parallel language and compiler. The UML diagram in Figure 2.16 illustrates our `BLAS`-based vector class package, which includes all the helper classes we have discussed so far.

2.2.2 Problem Classes of OPOS

The `AbstractProblem` class constitutes a unified interface through which our optimization algorithms interact with problem instances. One defines a class of problems by deriving a class from `AbstractProblem` (although perhaps indirectly). `AbstractProblem`-derived classes hold problem instance data and contain methods related to

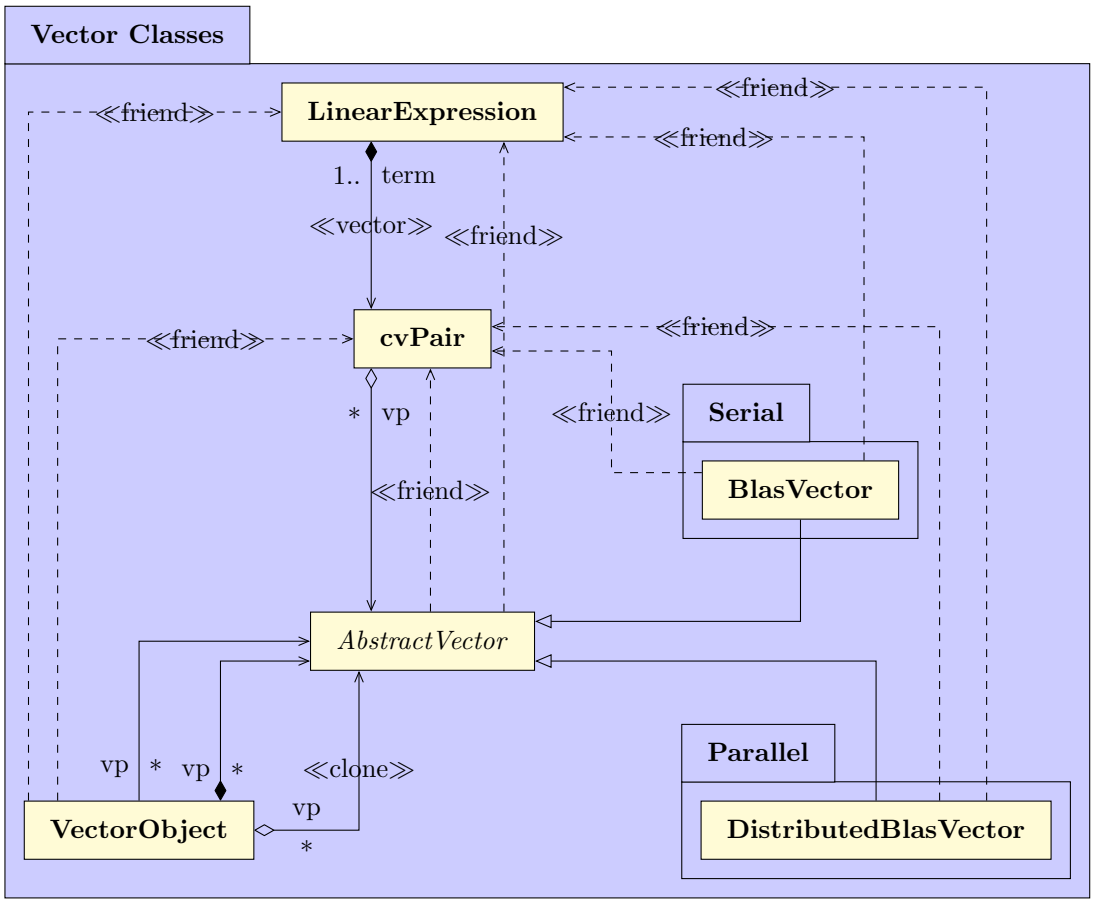


Figure 2.16: OPOS' BLAS-based vector class package.

them, such as objective function, constraint, and respective gradient evaluations. The problem class also stores problem data and related information, such as the number of variables, `VectorObject`s that represents the solution algorithm's starting point, and upper and lower bounds on variables, when present. In parallel settings, it is the `AbstractProblem`-derived class that determines how decision variables and problem instance data are distributed and possibly replicated among processors.

Simple Example Problem Classes

Because `OPOS`' problem classes include dozens of member variables and close to one hundred public methods it is impossible to give a clear overview of the workings of these problem classes by using them as examples. In addition, Chapter 3 and Chapter 4 each will deal extensively with specific problem classes of `OPOS`. Therefore, in this section we will illustrate the basic idea behind these problem classes by examples that are not actually part of `OPOS`, but follow the same design principles that we use in `OPOS`. We develop two `SimpleAbstractProblem`-derived class called `SimpleSerialProblem` and `SimpleParallelProblem`, which will serve as problem classes for the following optimization problem

$$\begin{aligned} \min \quad & \sum_{i=1}^{n} \left(x_i^2 - x_i + 1\right) \\ \text{s.t.} \quad & x_i \in \mathbb{R},\ i = 1 \ldots n. \end{aligned} \tag{2.4}$$

These classes are responsible for storing the problem's data, and for providing implementations to function and gradient evaluations. In a serial setting this can be achieved by either hard coding problem-specific methods in `SimpleSerialProblem` or using a library such as `ASL` [28] that provide an interface to an `nl` capable AML [42], in which case `SimpleSerialProblem` would become a problem-independent class. Because there are no such libraries for the parallel case, we have chosen to "hard code" all the methods for this example. Note that in these examples for the sake of brevity we will omit describing the destructors. The relationships between our example classes are illustrated in Figure 2.17.

The abstract problem class' private members are an integer `numVar` and a `VectorObject initialPrimal`. `SimpleAbstractProblem`'s constructor takes an integer as

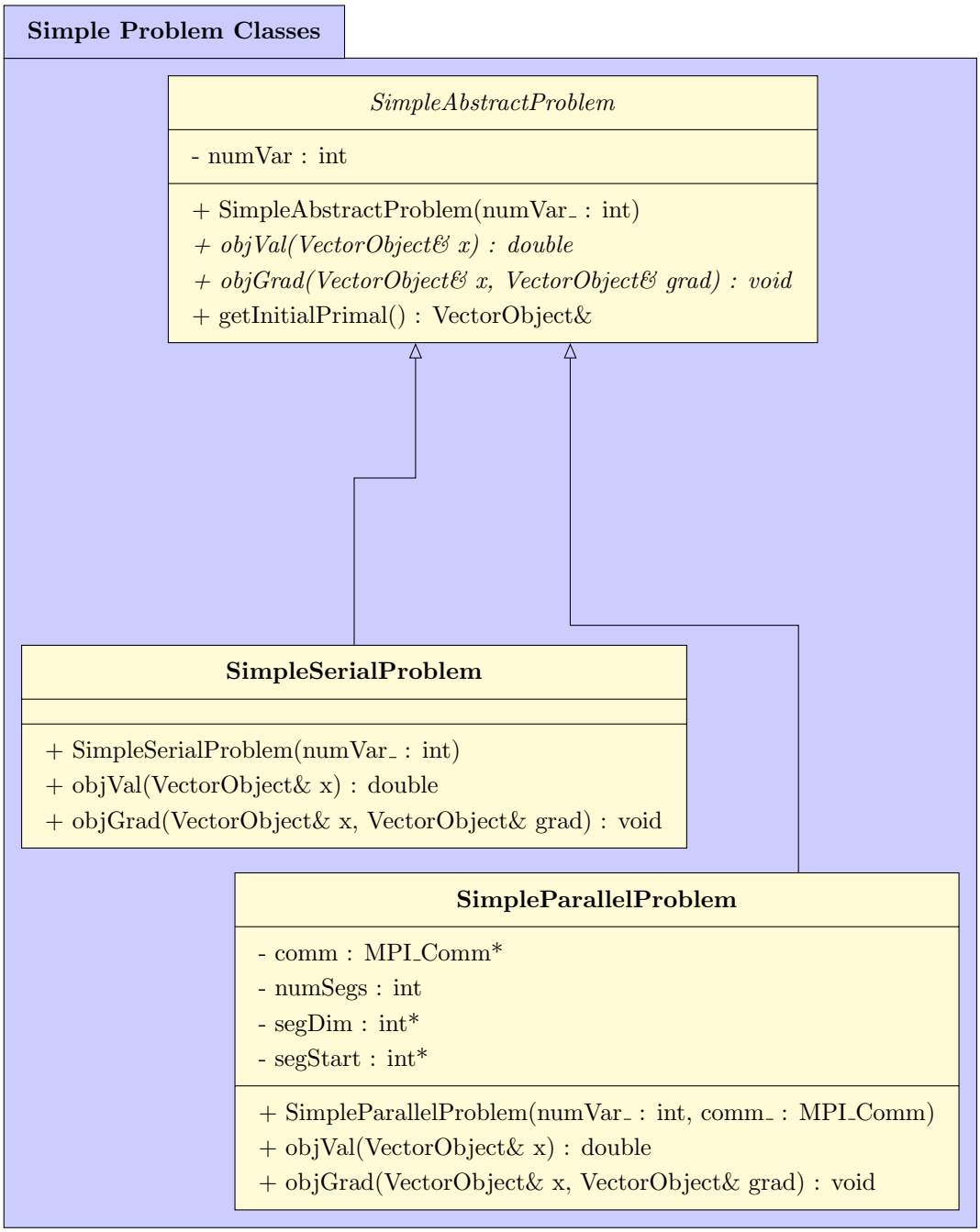


Figure 2.17: Inheritance structure of `SimpleAbstractProblem` derived classes.

```
#ifndef SIMPLEABSTRACTPROBLEM_H_
#define SIMPLEABSTRACTPROBLEM_H_
#include "VectorObject.h"
#include "LinearExpression.h"

class SimpleAbstractProblem {
  int numVar;
  VectorObject initialPrimal;
public:
  SimpleAbstractProblem(int numVar_):
    numVar(numVar_),
    inititialPrimal() { };

  virtual double objVal(VectorObject& x) = 0;
  virtual void objGrad(VectorObject& x, VectorObject& grad)
    ↪  = 0;
  virtual const VectorObject& getInitialPrimal() const {
    return initialPrimal;
  }
};
#endif /* SIMPLEABSTRACTPROBLEM_H_ */
```

Figure 2.18: Class definition of `SimpleAbstractProblem`.

an argument, which it uses to initialize `numVar`. Because the constructor does not have any information about the underlying vector class it calls the default constructor for the `VectorObject initialPrimal`, which sets its `AbstractVector` pointer to `NULL`. `SimpleAbstractProblem`'s pure virtual methods are `objVal()` and `objGrad()`, which enforces that all derived classes will have an implementation of these routines. Our abstract problem class also includes a method called `getInitialPrimal()` that returns the `initialPrimal` vector. The complete class definition of `SimpleAbstractProblem` is shown in Figure 2.18.

Both `SimpleSerialProblem` (Figure 2.19) and `SimpleParallelProblem` (Figure 2.20) are template classes that accept a vector class as the template argument. `SimpleSerialProblem` constructor receives `numVar`, which is the equivalent of n from (2.4). The constructor then calls `SimpleAbstractProblem` class' constructor and by doing so initializes `numVar` and `initialPrimal`. We then use `VectorObject`'s `setAvpMode()` method, which initializes `initialPrimal`'s `AbstractVector` pointer to the address

```
#ifndef SIMPLESERIALPROBLEM_H_
#define SIMPLESERIALPROBLEM_H_
#include "SimpleAbstractProblem.h"

template<class T>
class SimpleSerialProblem :
  public SimpleAbstractProblem {

public:
  SimpleSerialProblem(int numVar_):
    SimpleAbstractProblem(numVar_) {
    initialPrimal.setAvpMode(new T(numVar_), zero);
  };
  double objVal(VectorObject& x) {
    double value = 0.0;
    for(int i=0; i<numVar; ++i) {
      value += x[i]*x[i] - x[i] + 1.0;
    }
    return value;
  }
  void objGrad(VectorObject& x, VectorObject& grad) {
    for(int i=0; i<numVar; ++i) {
      grad[i] = 2.0*x[i] - 1.0;
    }
  }
};
#endif /* SIMPLESERIALPROBLEM_H_ */
```

Figure 2.19: Class definition of `SimpleSerialProblem`.

returned by `new T(numVar_)` and sets the `VectorConstructorMode` to `zero`, which means that upon creation the vector was initialized to all zeros and `initialPrimal`'s destructor is responsible for cleaning up the object. `SimpleSerialProblem`'s template argument accepts any `AbstractVector`-derived serial vector class, such as `BlasVector` for example. Methods `objVal()` and `objGrad()` compute the objective function and gradient values for (2.4).

In `SimpleParallelProblem` the `comm` object specifies the number of processors, P, that the program is using, the individual process `id`s, and various other variables that are important in a distributed memory environment. Problem data partitioning in this case is fairly straightforward, `numVar` is equal to either $\left\lceil \frac{n}{P} \right\rceil$ or $\left\lfloor \frac{n}{P} \right\rfloor$, which determines

how many coefficients of the decision vector x we assign to each processor. Given problem (2.4), our data distribution scheme does not require replication of variables, therefore, members `numSegs`, `segDim`, and `segStart` assume the trivial values 1, `numVar`, and 0. The procedure for initializing `initialPrimal` follows the same logic as in `SimpleSerialProblem`, except that in this case the template argument `T` needs to be a distributed vector class, such as `DistributedBlasVector` for example. Note, that for problem (2.4), in this parallel setup, `objVal()` needs to communicate between all the processors to return the correct function value. However, the `objGrad()` computations are parallelizable without the need for communication.

2.2.3 `OPOS` Algorithm Classes

Algorithm classes are not abstract classes. However, all of their vector operations are accessed through `VectorObject`s and they interact with problem instances by calling `AbstractProblem` virtual methods, which means that they are independent of the vector or problem classes we decide to use. Ignoring the class destructor, Figure 2.21 shows a possible implementation of an algorithm class for Algorithm 1, which is our gradient descent algorithm from the beginning of this chapter, using `VectorObject` and `SimpleAbstractProblem` classes. For now, let us treat the `lineSearch()` method as a "black box" and ignore that `GradientDescent` is a derived class of `LineSearchBasedMethod`. We will discuss these issues in detail below.

The algorithm class' constructor takes an abstract problem class pointer, which it then uses to initialize all of its members. Through the `VectorObject initialPrimal`, obtained from the problem class via `getInitialPrimal()`, the `GradientDescent` constructor determines the vector class that it will use to execute the linear algebra operations. `VectorObject` members are initialized by calling the `VectorObject` constructor in the member initializer list, which under the hood executes a cloning method on `initialPrimal`.

If we decided to solve problem (2.4) using Algorithm 1 in parallel, we would pass `SimpleParallelProblem` to `GradientDescent`'s constructor. Then when `GradientDescent`'s `minimize()` routine is called the `SimpleAbstractProblem` virtual method

```
#ifndef SIMPLEPARALLELPROBLEM_H_
#define SIMPLEPARALLELPROBLEM_H_
#include "SimpleAbstractProblem.h"
#include "mpi.h"

template<class T>
class SimpleParallelProblem :
  public SimpleAbstractProblem {
  MPI_Comm comm;
  int numSegs;
  int* segDim;
  int* segStart;
public:
  SimpleParallelProblem(int numVar_, MPI_Comm comm_ =
     ↪ MPI_COMM_WORLD):
    SimpleAbstractProblem(numVar_),
    comm(comm_),
    numSegs(1) {
    segDim = new int[1];
    segDim[0] = numVar;
    segStart = new int[1];
    segStart[0] = 0;
    initialPrimal.setAvpMode(new T(numVar_, numSegs, segDim
       ↪ , segStart, &comm_), zero);
  };
  double objVal(VectorObject& x) {
    double local = 0.0, value = 0.0;
    for(int i=0; i<numVar; ++i) {
      local += x[i]*x[i] - x[i] + 1.0;
    }
    MPI_Allreduce(&local, &value, 1, MPI_DOUBLE, MPI_SUM, *
       ↪ comm);
    return value;
  }
  void objGrad(VectorObject& x, VectorObject& grad) {
    for(int i=0; i<numVar; ++i) {
      grad[i] = 2.0*x[i] - 1.0;
    }
  }
};
#endif /* SIMPLESERIALPROBLEM_H_ */
```

Figure 2.20: Class definition of `SimpleParallelProblem`.

```
#ifndef GRADIENTDESCENT_H_
#define GRADIENTDESCENT_H_

#include "SimpleAbstractProblem.h"
#include "LineSearchBasedMethod.h"

class GradientDescent :
  public LineSearchBasedMethod {

SimpleAbstractProblem* pr; // problem
VectorObject x;   // point
VectorObject g;   // gradient
double stepSize;  // line search step size

public:

  // Constructor
  GradientDescent(SimpleAbstractProblem* pr_):
    pr(pr_),
    x(pr->getInitialPrimal(), copyData),
    g(pr->getInitialPrimal(), zero),
    stepSize(0.0) { };

  // Minimization method
  VectorObject& minimize(double epsilon) {
    x = pr->getInitialPrimal();
    pr->objGrad(x,g);
    while(g.norm2() > epsilon) {
      stepSize = lineSearch();
      x = x - stepSize * g;
      pr->objGrad(x,g);
    }
    return x;
  }
};
#endif /* GRADIENTDESCENT_H_ */
```

Figure 2.21: Object-parallel gradient descent algorithm class.

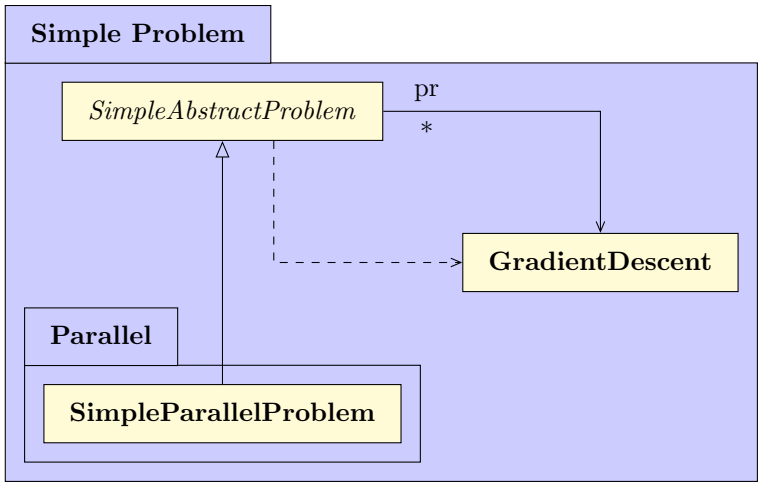


Figure 2.22: Parallel gradient descent algorithm on problem (2.4)

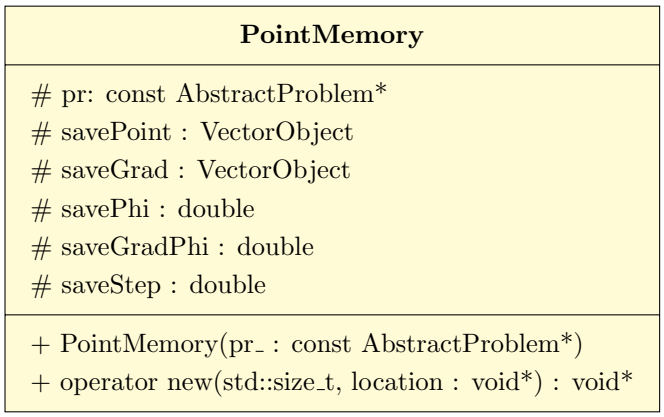
`objGrad()` would invoke the implementation in `SimpleParallelProblem`. The relationship between these classes is illustrated in Figure 2.22.

Line Search Support: Transparent Function and Gradient Caching

Like our gradient descent method (Algorithm 1) many optimization algorithms employ line search procedures to find suitable stepsizes α_k at each iteration. Essentially, after determining a step direction d^k, line search procedures perform some kind of procedure to determine the stepsize α^k in the step calculation $x^{k+1} = x^k + \alpha^k d^k$. To promote efficient implementation of such procedures, our framework provides a base class called `LineSearchBasedMethod` from which all optimization algorithms employing line searches are derived. This class stores built-in members that are scalar and vector quantities typically maintained by line search algorithms, including the objective values, gradients, and the search direction at the current iterate. Storing these values can help speed up the line search computations significantly. `LineSearchBasedMethod` implements storage and retrieval of these values through the `PointMemory` class.

```
PointMemory(const AbstractProblem* pr_):
  pr(pr_),
  savePoint(pr->getInitialPrimal(), zero),
  saveGrad(pr->getInitialPrimal(), zero),
  savePhi(std::numeric_limits<double>::quiet_NaN()),
  saveGradPhi(std::numeric_limits<double>::quiet_NaN()),
  saveStep(-1.0),
  computePhi(true),
  computeGrad(true) { };
```

Figure 2.23: `PointMemory` constructor.



The constructor of `PointMemory` initializes its members in a similar way to algorithm classes. As shown in Figure 2.23, `VectorObjects` are initialized based of of the problem class' `initialPrimal` vector. The overloaded `new` operator, or so-called *placement new* operator, allows us to create a contiguous array of `PointMemory` objects using consecutive calls to `new`. `LineSearchBasedMethod` holds a `PointMemory*` pointer. `OPOS`' `AbstractProblem` class contains a `generatePtMem()` method, which is called from `LineSearchBasedMethod` to create a contiguous `PointMemory` array of length `mem`. Implementation of `generatePtMem()` is shown in Figure 2.24. The ability to create a contiguous `PointMemory` array makes storage and retrieval of line search and application-specific data more efficient. Due to the `generatePtMem()` method, both `AbstractProblem` and its derived classes are dependent on `PointMemory`. Furthermore, `PointMemory` is associated with `AbstractProblem` because of its `AbstractProblem*`

```
virtual PointMemory* generatePtMem(int mem) {
  ptMem = (PointMemory*)malloc(mem * sizeof(PointMemory));
  for(int i=0; i<mem; ++i)
  new(ptMem+i) PointMemory(this);
  return ptMem;
};
```

Figure 2.24: `AbstractProblem`'s `generatePtMem()` method.

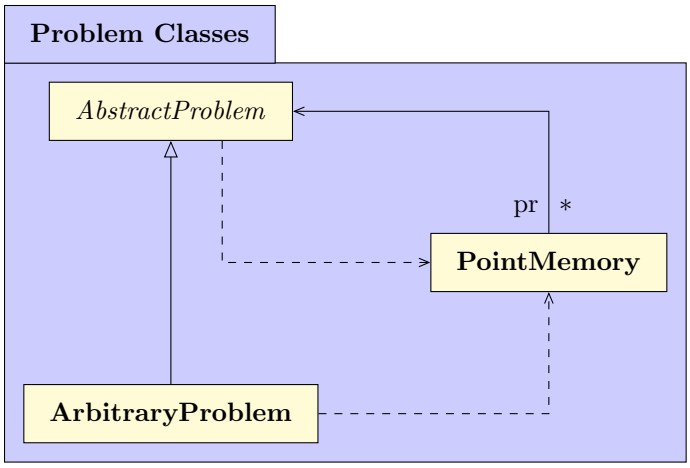


Figure 2.25: Problem class and `PointMemory` relationships.

pointer. These relationships are depicted in Figure 2.25, where we call the `AbstractProblem`-derived class `ArbitraryProblem`, which only serves illustrative purposes.

Furthermore, `LineSearchBasedMethod` provides methods `phi()` and `gradPhi()` for computing the value and gradient of the line search function $\phi_k(\alpha) = f(x^k + \alpha d^k)$, where f is the problem objective function. In some line search procedures, $\phi_k(\alpha)$ or $\nabla\phi_k(\alpha)$ may be evaluated more than once at the same value of α; for example, this phenomenon can occur in the conjugate gradient algorithm of [37]. Another possible situation is that the line search algorithm may compute $\phi_k(\alpha)$ and subsequently compute $\nabla\phi_k(\alpha)$ for the same value of α. For many problem classes, these two calculations may share significant common underlying computations, whose results it may be more efficient to cache than to recompute. The caching mechanism built into the `phi()` and `gradPhi()` methods prevents time-consuming recomputation of the function value or gradient in such cases, while keeping the solution algorithm code free of clutter from caching-related details.

Termination Classes

The termination condition of our object-parallel gradient descent algorithm in Figure 2.21 is to check whether the norm of the gradient is below `epsilon`, which is an argument of the `minimize()` routine. Let us assume that we would like to change our termination condition to something more complicated. For example, we would also like to terminate if we do not make enough progress, i.e. $\left\|x^k - x^{k+1}\right\| \leq \delta$. Now, if we want to add this additional condition, we would have to modify the gradient descent code. Many numerical implementations of optimization algorithms have even more complicated termination conditions, sometimes checking more than a dozen conditions at each iteration. In addition, solver developers often need to experiment with various termination conditions to empirically improve their code. It is easy to see how this could become cumbersome as the termination condition becomes more complicated, cluttering our algorithm's source code and reducing its maintainability. Our solution to this problem is to abstract away the termination checks from the optimization algorithm, just like we have done with vector operations and problem dependent computations. `OPOS` achieves this abstraction through its `AbstractTermination` class.

AbstractTermination
lsBase: LineSearchBasedMethod* # tol : double # iter : int # maxIter : int # flag : int
+ AbstractTermination(lsBase_ : LineSearchBasedMethod*, tol_ : double) + *check() : bool*

`AbstractTermination`'s pure virtual method is `check()`, which returns a `bool`. Using this class our object-parallel gradient descent algorithm's minimization routine could be modified as shown in Figure 2.26. This modification allows us to pass any `AbstractTermination`-derived class to the `minimize()` method, each with its own implementation of the `check()` routine.

```
VectorObject& minimize(AbstractTermination& termination) {
  x = pr->getInitialPrimal();
  pr->objGrad(x,g);
  while(termination.check()) {
    stepSize = lineSearch();
    x = x - stepSize * g;
    pr->objGrad(x,g);
  }
  return x;
}
```

Figure 2.26: Using `AbstractTermination` in gradient descent algorithm.

Chapter 3

Object-Parallel Spectral Projected Gradient Algorithm Applied to LASSO Problems

In this chapter we briefly discuss some theoretical background relating to *gradient projection* algorithms. This is followed by introducing the *nonmonotone spectral projected gradient* (SPG) method by Birgin and Martínez and a detailed description of the object-parallel algorithm class implementing it, which is referred to as `OPSPG`. We have used `OPSPG` to solve a series of LASSO problems both in serial and parallel. A detailed examination of the problem classes implementing LASSO problems is provided as well. Finally, we will present computational results obtained from distributed memory parallel runs of `OPSPG` on LASSO test problems.

3.1 Gradient Projection Algorithms

Gradient projection algorithms [6, 10, 8] are iterative descent algorithms, where the generated sequence $\{x_k\}$ remains feasible at each iteration k. These methods are often used to solve problems of the type

$$\begin{aligned} \min \quad & f(x) \\ \text{s.t.} \quad & x \in X, \end{aligned} \tag{3.1}$$

where f is a continuously differentiable convex function and $X \subset \mathbb{R}^n$ is a nonempty closed convex set such that projection onto X does not require significant computational effort. The iterates are generated by

$$x_{k+1} = P_X\left(x_k - \alpha_k \nabla f(x_k)\right), \tag{3.2}$$

where $\alpha_k > 0$ is a stepsize and $P_X(\cdot)$ denotes projection onto X, which is well defined given that X is closed and convex. Using Theorem 3.1.1 below, we can prove the descent

property $f(x_{k+1}) < f(x_k)$ for appropriate α_k.

Theorem 3.1.1 (Descent properties of gradient projection [8, p. 304]). *Let $f : \mathbb{R}^n \to \mathbb{R}$ be a continuously differentiable function, let X be a closed convex set. Then for all $x_k \in X$ and $\alpha > 0$:*

1. *If $x_k(\alpha) \neq x_k$, then $x_k(\alpha) - x_k$ is a feasible descent direction at x_k. In particular, we have*
$$\langle \nabla f(x_k), (x_k(\alpha) - x_k) \rangle \leq -\frac{1}{\alpha} \|x_k(\alpha) - x_k\|^2, \ \forall\ \alpha > 0. \tag{3.3}$$
2. *If $x_k(\alpha) = x_k$ for some $\alpha > 0$, then x_k satisfies the necessary condition for minimizing f over X,*
$$\langle \nabla f(x_k), (x - x_k) \rangle \geq 0, \ \forall\ x \in X. \tag{3.4}$$

Where for given $x_k \in X$, $x_k(\alpha) = P_X(x_k - \alpha \nabla f(x_k)),\ \alpha > 0$ is the projection arc.

3.2 Nonmonotone Spectral Projected Gradient (SPG) algorithm

Gradient projection methods are considered to be slow unless the line search algorithm employed to compute α_ks is designed carefully. For this reason, Birgin and Martínez in [14] incorporated a combination of two line search strategies, developed for unconstrained minimization, into SPG. These strategies comprise a nonmontone line search technique originally developed for Newton's method [35] and a stepsize selection procedure resembling the *Barzilai-Borwein* [5] method. Algorithm 4 provides a description of SPG, where the termination condition is defined in terms of *continuous projected gradient*.

Definition 3.2.1 (Continuous projected gradient). *Take a continuously differentiable function $f : \mathbb{R}^n \to \mathbb{R}$ over the closed convex set $X \subset \mathbb{R}^n$. We denote by*
$$\nabla^P f(x) = P_X(x - \nabla f(x)) - x \tag{3.5}$$
its continuous projected gradient, *where P_X denotes the projection onto X.*

Algorithm 4: `SPG`

Input: f, ∇f, X, $x_0 \in \mathbb{R}^n$, $\lambda_{\max}$, $\lambda_{\min}$, $\epsilon_{\text{opt}} > 0$

Output: x', local minimizer of f

1 $k = 0$

2 $x_k = P_X(x_k)$

3 $f_{\text{vals}} \leftarrow f(x_k)$

4 $\beta_k = \left\|\nabla^P f(x_k)\right\|_\infty$

5 **if** $\beta_k \neq 0$ **then**

6 $\quad \lambda_k = \min\left\{\lambda_{\max}, \max\left\{\lambda_{\min}, \frac{1}{\beta_k}\right\}\right\}$

7 **while** $\left\|\nabla^P f(x_k)\right\| > \epsilon_{opt}$ **do**

8 $\quad d_k = P_X(x_k - \lambda_k \nabla f(x_k)) - x_k$

9 $\quad \alpha_k \leftarrow$ `lineSearch()`

10 $\quad x_{k+1} = x_k + \alpha_k d_k$

11 $\quad s_k = x_{k+1} - x_k$

12 $\quad y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$

13 $\quad \beta_k = \langle s_k, y_k \rangle$

14 $\quad$ **if** $\beta_k \leq 0$ **then**

15 $\quad\quad \lambda_{k+1} = \lambda_{\max}$

16 $\quad$ **else**

17 $\quad\quad \lambda_{k+1} = \min\left\{\lambda_{\max}, \max\left\{\lambda_{\min}, \frac{\langle s_k, s_k \rangle}{\beta_k}\right\}\right\}$

18 $\quad f_{\text{vals}} \leftarrow f(x_{k+1})$

19 $\quad k = k + 1$

20 $x' = x_k$

SPG, essentially, employs two line search techniques. The first one is the spectral step in Line 6 and Line 17, where λ_k is the safeguarded "inverse Rayleigh quotient" corresponding to the average Hessian matrix $\int_0^1 \nabla^2 f(x_k + t s_k) dt$, namely

$$\frac{\|x_{k+1} - x_k\|^2}{\langle x_{k+1} - x_k, g_{k+1} - g_k \rangle}. \tag{3.6}$$

The safeguarding procedure in Lines 14 to 17 with parameters $\lambda_{\min}$ and $\lambda_{\max}$ protect against unstable step sizes resulting from unreliable curvature estimates. The second

line search with a nonmonotone Armijo condition from Line 9 is detailed in Algorithm 5.

Algorithm 5: `SPG lineSearch()`

Input: f, ∇f, $f_{\text{vals}}[]$, M, d_k, $0 < \alpha_{\min} < \alpha_{\max}$, $0 < \sigma_{\text{low}} < \sigma_{\text{up}} < 1$, $\gamma \in (0,1)$

Output: α_k, satisfying the nonmonotone Armijo condition

1 $f_{\max} = \max_{i=1\ldots M} \left\{ f^i_{\text{vals}} \right\}$
2 $\delta = \langle \nabla f(x_k), d_k \rangle$
3 $\alpha = 1$
4 $x_+ = x_k + \alpha d_k$
5 **while** $f(x_+) > f_{\max} + \alpha\gamma\delta$ **do**
6 **if** $\alpha \leq 0.1$ **then**
7 $\alpha = \dfrac{\alpha}{2}$
8 **else**
9 $\alpha_{\text{temp}} = \dfrac{-\delta\alpha^2}{2\left(f(x_+) - f(x_k) - \alpha\delta\right)}$
10 **if** $\alpha_{\text{temp}} < \sigma_{\text{low}}$ *or* $\sigma_{\text{up}} < \alpha_{\text{temp}}$ **then**
11 $\alpha_{\text{temp}} = \dfrac{\alpha}{2}$
12 $\alpha = \alpha_{\text{temp}}$
13 $x_+ = x_k + \alpha d_k$
14 $\alpha_k = \alpha$

SPG keeps track of the last M objective function values in f_{vals}, and the nonmonotone line search modifies the Armijo condition

$$f(x_k + \alpha d_k) \leq f(x_k) + \alpha\gamma \left\langle \nabla f(x_k), d_k \right\rangle, \tag{3.7}$$

where $\gamma \in (0,1)$ (as in Algorithm 5) by replacing $f(x_k)$ with the largest recent value.

3.3 `OPSPG`: Object-Parallel Implementation of SPG

`SPG` is an algorithm class that is derived from `ProjLineSearchBasedMethod`, which in turn is a derived class of `LineSearchBasedMethod`. `ProjLineSearchBasedMethod` has additional routines related to projected gradient methods such as, norm computations of the projected gradient (`pg`), or `projPhi()` and `projGradPhi()`, which ensure that the line search procedures maintain feasibility. `OPSPG`'s algorithm class `SPG`, like all of `OPOS`' optimization algorithms, is implemented using `VectorObject`, `AbstractProblem` and

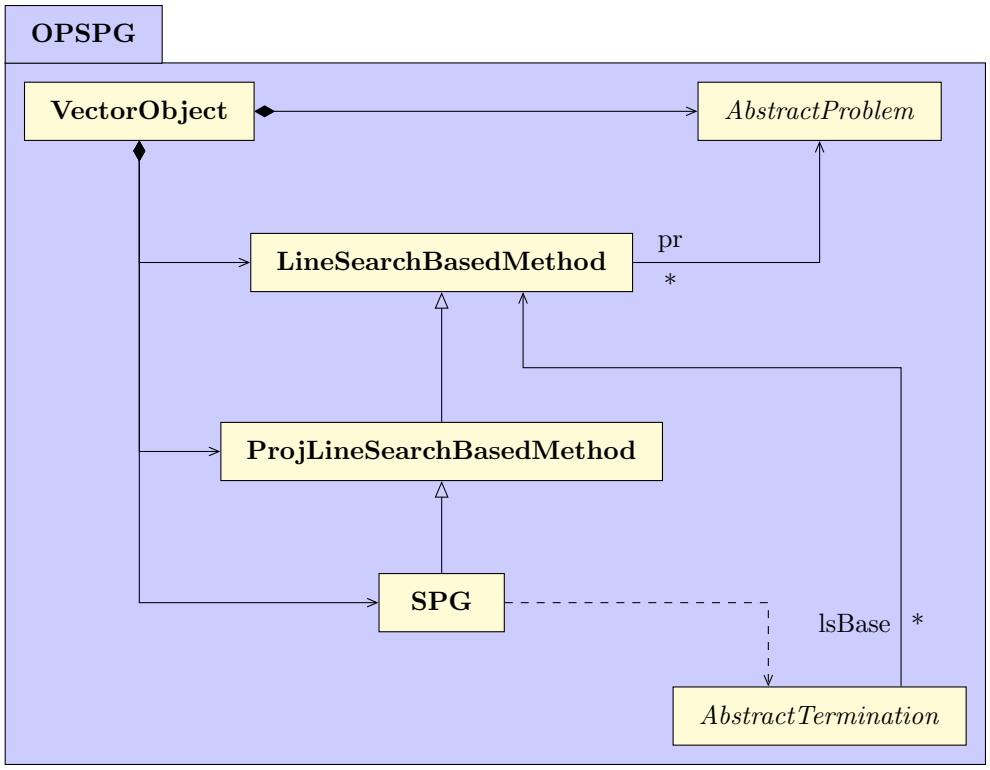


Figure 3.1: Relationships of the `OPSPG` package.

`AbstractTermination` classes. The complete inheritance graph of the `OPSPG` software package is illustrated in Figure 3.1. Figures 3.2 to 3.4 show `OPOS`' source code for Algorithms 4 and 5. The initialization of the minimization routine (Figure 3.2) uses `AbstractProblem` routines `localProjectOnBounds()` and `objValGrad()`. The function call `localProjectOnBounds(x)` projects vector x onto X, and `objValGrad(x,g)` computes the gradient at x, which is stored in g, and at the same time returns the objective function value at x. The main algorithm (Figure 3.3) and the line search (Figure 3.4) use `objGrad()`, which is an `AbstractProblem` routine, `check()` an `AbstractTermination` routine, and `phi()` and `reset()`, which are both `LineSearchBasedMethod` routines. The `reset()` routine adjusts `LineSearchBasedMethod`'s members for the new point x_{k+1}.

```
iter = 0;
pr->localProjectOnBounds(x); /* AbstractProblem routine  */
xBest = x;
double objValStart = objVal = pr->objValGrad(x, g); /*
   ↪ AbstractProblem routine  */
objValBest = objVal;
objValMax = objVal;
xk = x - g;
pr->localProjectOnBounds(xk); /* AbstractProblem routine
   ↪ */
pg = xk - x; /* projected gradient */
objValArray[0] = objVal;
for(int i=1; i<M; i++) {
  objValArray[i] = -1.0 * std::numeric_limits<double>::max
     ↪ ();
}
double pgNormInf = pg.normInf();
if(pgNormInf != 0.0) {
  stepSize = std::min(stepSizeMax, std::max(stepSizeMin,
     ↪ 1.0/pgNormInf));
}
```

Figure 3.2: Initialization of `minimize()` routine in `SPG`.

```
VectorObject& SPG::minimize(AbstractTermination&
   ↪ termination) {

/* Initialization here */

  while(termination.check() && iter < maxIter) {
        /* AbstractTermination routine */

    iter++;
    xk = x - stepSize*g;
    pr->localProjectOnBounds(xk);
    /* AbstractProblem routine  */
    d = xk - x;

    /* Perform line serach here */

    xk = x + alpha*d;
    pr->objGrad(xk, gk); /* AbstractProblem routine  */
    xd = xk -x;
    double xdNorm2sq = xd.norm2sq();
    gd = gk - g;
    double xdtgd = xd.inner(gd);
    if(xdtgd <= 0) {
      stepSize = stepSizeMax;
    }
    else {
      stepSize = std::max(stepSizeMin, std::min(stepSizeMax
         ↪ , xdNorm2sq/xdtgd));
    }
    x = xk;
    g = gk;
    objVal = phia;
    objValArray[iter%M] = objVal;
    xk = x - g;
    pr->localProjectOnBounds(xk);
    /* AbstractProblem routine  */
    pg = xk - x;
    if(objVal < objValBest) {
      objValBest = objVal;
      xBest = x;
    }
      reset(); /* LineSearchBasedMethod routine  */
    }
    return xBest;
  }
```

Figure 3.3: SPG's `minimize()` routine.

```
double gtd = g.inner(d);
objValMax = *(std::max_element(objValArray, objValArray + M
   ↪ ));
alpha = 1.0;
phia = phi(alpha); /* LineSearchBasedMethod routine */
lineIter = 0;
while(phia > objValMax + gamma*alpha*gtd && lineIter <
   ↪ maxLineIter) {
  lineIter++;
  if(alpha <= 0.1) {
    alpha /= 2.0;
  }
  else {
    alphaTemp = (-gtd*alpha*alpha )/(2.0*( phia - objVal -
       ↪ alpha*gtd ));
    if(alphaTemp < sigmaOne || alphaTemp > sigmaTwo*alpha)
      alpha = alpha/2.0;
  }
  phia = phi(alpha); /* LineSearchBasedMethod routine */
}
```

Figure 3.4: `SPG`'s `lineSearch()` routine.

3.4 Solving LASSO Problems with `OPSPG`

LASSO and its generalizations have gained great popularity due to large-scale data-analysis tools in recent years. In its original form, the LASSO problem may be written

$$\min_{x\in\mathbb{R}^n} \tfrac{1}{2}\|Ax-b\|^2 + \nu\|x\|_1\,, \tag{3.8}$$

where A is an $m\times n$ real matrix, $b\in\mathbb{R}^m$, and $\nu>0$ is a given parameter. Essentially, the problem is a form of linear regression augmented with an ℓ_1 penalty for moving the regression coefficients x away from zero. Often we have $n\gg m$, that is, A is a "wide" matrix with many more columns than rows. Our interest here will be in problem instance in which A is an extremely large matrix, either sparse or dense.

In its original form (3.8), the LASSO problem is an unconstrained convex minimization problem, but nonsmooth due the presence of the term $\nu\|x\|_1$. However, it is easily converted to a bound-constrained smooth optimization problem as follows: we let $x = x^+ - x^-$, where x^+ and x^- are two vectors in $\mathbb{R}^n$, both constrained to be

nonnegative. Letting $\mathbf{1}$ denote the vector of all 1's in $\mathbb{R}^n$, we may then write the term $\nu \|x\|_1$ as $\nu \mathbf{1}^\top (x^+ + x^-)$. Therefore, the problem (3.8) may be re-expressed as

$$\begin{array}{ll} \min\limits_{x^+, x^- \in \mathbb{R}^n} & \frac{1}{2} \|A(x^+ - x^-) - b\|^2 + \nu \mathbf{1}^\top (x^+ + x^-) \\ \text{ST} & x^+, x^- \geq 0. \end{array} \tag{3.9}$$

The problem is thus converted to minimization of a positive semidefinite quadratic (and hence smooth) function of (x^+, x^-) subject to the constraint that (x^+, x^-) must lie in the nonnegative orthant. Note that vectors (x^+, x^-) in the feasible region of this problem may possess an index i for which $x_i^+ > 0$ and $x_i^- > 0$, but this situation cannot occur in an optimal solution because setting $x_i^+ \leftarrow \max\{x_i^+ - x_i^-, 0\}$ and $x_i^- \leftarrow \max\{x_i^- - x_i^+, 0\}$ immediately yields a lower value of the objective function without violating the constraints. The problem formulation (3.9) is well-suited to solution by the SPG method. Therefore, we use this problem class to demonstrate how SPG, when implemented in our object-parallel framework, can be used as an efficient, scalable parallel algorithm, despite its MATLAB-like simplicity. We focus on the fairly common case that $n \gg m$, meaning that x, x^+, and x^- are very high-dimensional vectors in comparison with b.

Given the description of the SPG algorithm in Section 3.2, we know that it needs the objective function value at the initialization stage (`objValGrad()`) and during the line search procedure (`phi()`), the objective gradient at the initialization and inside the main while loop (`objValGrad()`, `objGrad()`), and needs to be able to perform projections both during initialization and within the main loop (`localProjectOnBounds()`). Let $h(x^+, x^-)$ denote the objective function of (3.9)

$$h(x^+, x^-) = \frac{1}{2} \big\| A(x^+ - x^-) - b \big\|^2 + \nu \mathbf{1}^\top (x^+ + x^-) \tag{3.10}$$

then the gradient becomes

$$\nabla h(x^+, x^-) = \begin{bmatrix} A^\top \big(A(x^+ - x^-) - b \big) + \nu \mathbf{1} \\ -A^\top \big(A(x^+ - x^-) - b \big) + \nu \mathbf{1} \end{bmatrix}, \tag{3.11}$$

and projecting onto the nonnegative orthant translates to

$$\begin{array}{rcl} x_i^+ & = & \max\left\{0, x_i^+\right\} \\ x_i^- & = & \max\left\{0, x_i^-\right\}. \end{array} \tag{3.12}$$

The problem classes implementing LASSO need to provide implementations of these routines, with special focus on the efficiency of the matrix multiplication $A(x^+ - x^-)$, needed by both the function and gradient calculations, and the subsequent multiplication by $A^\top$ needed to calculate the gradient, since these dominant the numerical work in SPG.

3.4.1 Solving LASSO in Serial

In serial, the most straightforward approach to solving LASSO, from an implementation perspective, would be to build a LASSO model in an AML environment, like `AMPL` or `PYOMO`, generate an `nl` file and use the generic `AslProblem` class within `OPOS` to provide the necessary problem class routines needed by SPG. This means that we do not have to implement a custom LASSO problem class. However, we have found that we can improve the runtime of SPG by using a dedicated LASSO problem class, where we have control over optimizing the objective function and gradient evaluations.

The `ASL`-based solver in Figure 3.5 takes as an argument that is the name of the `nl` file, which is stored in `problemName`. This is then used as an argument in the `AslProblem` object constructor, which finds the `nl` file and uses it to store all problem related data in memory. Next the driver creates a `VectorObject x`, which encapsulates a `BlasVector`, the algorithm object `SPG spg`, and termination object `ProjGradNormTermination termination`. `ProjGradNormTermination` class checks whether the norm of the projected gradient is less than the given tolerance, in this case 10^{-6}. We solve the problem by calling `spg`'s `minimize()` routine. Finally, the driver writes the solution to a `sol` file, which could be read back by the AML to interpret the results.

Given that both the data representation and the computations in LASSO are fairly simple, we can easily create a more efficient problem class `LassoProblem`, which directly reads in matrix A and vector b from a file, and has custom objective function and gradient implementations. For example, Figure 3.6 shows `LassoProblem`'s custom `objVal()` implementation that uses `BLAS` for matrix-vector multiplies. The `LassoProblem::objVal()` routine maps the input vector `v` as two vectors `x` and `y` to mimic

```
#include "SPG.h"
#include "BlasVector.h"
#include "GradNormTermination.h"
#include "AslProblem.h"

int main(int argc, char* argv[]) {
  int index = 0;
  for(int i=1; i<argc; ++i) {
    if(strcmp(argv[i], "-s") == 0)
      index = i+1;
  }
  const std::string problemName(argv[index]);
  AslProblem<BlasVector> asl(problemName);
  VectorObject x;
  x.setAvpMode(new BlasVector(asl.getNumVar()),zero);
  SPG spg(asl);
  ProjGradNormTermination termination(spg, 1.0e-6);
  x = spg.minimize(termination);
  char* msg = const_cast<char*>("Solution Found!");
  asl.writeSolution(msg, x.getData());
  return 0;
}
```

Figure 3.5: Serial `ASL` solver with `BlasVector`.

```
double LassoProblem::objVal(VectorObject& v) {
  numObjEval++;
  VectorObject x(v, 0, numCols-1, notOwnedSub);
  VectorObject y(v, numCols, 2*numCols-1, notOwnedSub);
  zPlus = x + y;
  zMinus = x - y;
  cblas_dgemv(CblasRowMajor, CblasNoTrans, m, n, 1.0, A.
    ↪ getData(), lda, zMinus.getData(), incx, 0.0, Az.
    ↪ getData(), incy);
  Az = Az - b;
  double objVal = 0.5*Az.norm2sq() + nu.inner(zPlus);
  delete x.getAvp();
  delete y.getAvp();
  return objVal;
}
```

Figure 3.6: Serial `LassoProblem`'s `objVal()` routine.

the converted formulation of the LASSO, where $x = x^+ - x^-$. This can be efficiently accomplished by using `VectorObject` constructors that do not create actual copies of `v` in memory, just pointers that point to a segment of `v`, where the first element is the second argument of the constructor, and the last the third. The rest of the `objVal()` routine follows from the mathematical formulation of the LASSO problem. This approach avoids the overhead that is associated with using `AslProblem` routines and provides a more efficient implementation of both the objective function and the gradient than the `ASL` library.

3.4.2 Solving LASSO in Parallel

As mentioned in Section 1.2, there are no software libraries that can do in a distributed memory parallel setting what `ASL` can in serial. Therefore, building custom LASSO problem classes is inevitable in the parallel case. These problem classes determine the data partitioning and the communication primitives that are related to problem-specific routines such as objective function evaluation. In order to attain good parallel efficiency for these routines it is fundamental that the data partitioning procedure assigns an even work load to each processor. We differentiate between two cases, the first case is where the underlying matrix A is dense, the second case is where matrix A is sparse. Each case has its own problem class, `DistributedDenseLassoProblem` and `DistributedSparseLassoProblem`.

There are various popular data formats for regression type problems such as `svm` [48] or matrix market `mm` [49]. LASSO problem classes can either directly handle some of these data formats or we have included converters in `OPOS`, which convert a specific format into one that can be handled by a LASSO problem class. However, in this section we will focus on what we have found to be the most efficient way of storing A and b. In the dense case we will represent the data in two binary files, one for the matrix A and the other for the labels b. The matrix file holds the coefficients in *column major order*, which means that the first m coefficients belong to the first column, etc. [50]. When the data are sparse, we store the matrix in four separate binary files: values, row numbers, position of column starts, and ends, which follows Intel `MKL`'s `BLAS CSC`

matrix storage format [66].

Depending on whether matrix A is dense or sparse, we use different data partitioning. For the dense case, we adopt the straightforward approach of simply partitioning responsibility for the elements of x^+ and x^-, along with the corresponding columns of A, among the available processor cores. When A is sparse we use a novel data partitioning technique that distributes the nonzeros of A evenly between processors. This technique achieves perfect load balance and is not influenced by the sparsity pattern of A. Therefore, for sparse matrices, instead of distributing individual columns between processors, which can lead to an uneven load balance, we use the nonzero partitioning technique. Let us note briefly, that this nonzero partitioning can result in one column of A being owned by multiple processors.

Before we start discussing the details of the dense and the sparse approach let us introduce some notation involving our analysis. Suppose we have P processors numbered $1, \ldots, P$, and let $\{J_1, \ldots, J_P\}$ be a collection of subsets of $\{1, \ldots, n\}$, with J_i denoting the set of indices assigned to processor i. In the dense case, $\{J_1, \ldots, J_P\}$ is a partition of $\{1, \ldots, n\}$. In the sparse case, $\{J_1, \ldots, J_P\}$ is a cover, since an index $j \in \{1, \ldots, n\}$ could be assigned to multiple J_i-s. We momentarily defer discussing how we determine the partition $\{J_1, \ldots, J_P\}$. Here, $x^+_{(i)}$ and $x^-_{(i)}$ denote the respective subvectors of x^+ and x^- consisting of the coefficients with indices assigned to processor i, that is, $x^+_{(i)}$ denotes the vector consisting of x^+_j for $j \in J_i$, and similarly for $x^-_{(i)}$. Likewise, $A_{(i)}$ denotes the matrix consisting of the columns of A_j of A for which $j \in J_i$. We replicate vector b in the memory of all processors, since b is a much shorter vector than x, the memory usage impact of this replication is limited. Throughout this discussion, we treat each processor as having its own memory; as discussed earlier, this does not prevent the method from being implemented on a system in which physical memory is shared.

Distributed Dense LASSO Problem

When the matrix A is dense we determine the partition members J_i by simply dividing the indices $\{1, \ldots, n\}$ into P contiguous groups whose size varies by at most one element:

$$\begin{array}{cccccccc} A_1 & A_2 & A_3 & A_4 & A_5 & A_6 & A_7 & A_8 \end{array}$$
$$\begin{bmatrix} * & * & * & * & * & * & * & * \\ * & * & * & * & * & * & * & * \\ * & * & * & * & * & * & * & * \\ * & * & * & * & * & * & * & * \\ * & * & * & * & * & * & * & * \\ * & * & * & * & * & * & * & * \\ * & * & * & * & * & * & * & * \\ * & * & * & * & * & * & * & * \end{bmatrix}$$

Figure 3.7: Dense matrix, nonzero entries indicated by *.

formally, we may take the first $n \mod P$ groups to have size $\lceil \frac{n}{P} \rceil$ and the remaining groups to have size $\lfloor \frac{n}{P} \rfloor$. For example, take the dense matrix in Figure 3.7, with 8 columns and 8 rows, if $P = 8$, then each column is assigned to a different processor as shown in Figure 3.8.

Given the column distribution scheme, we may calculate $A(x^+ - x^-)$ as follows:

1. Each processor i locally calculates $d_{(i)} = x^+_{(i)} - x^-_{(i)}$
2. Each processor i locally calculates $q_{(i)} = A_{(i)} d_{(i)} = A(x^+_{(i)} - x^-_{(i)})$
3. We sum the vectors $q_{(i)} \in \mathbb{R}^m$ over all processors, $\left(\sum_{i=1}^P q_{(i)}\right)$, using `MPI_Allreduce` operation. Every processor thus receives the vector $q = A(x^+ - x^-) = \sum_{i=1}^P q_{(i)}$.

The additional steps needed to calculate the objective function $h(x^+, x^-)$ are now as follows:

1. Each processor i locally computes $\omega_i = \sum_{j \in J_i} (x^+_j + x^-_j)$
2. We compute the sum $\omega = \sum_{i=1}^P \omega_i = \mathbf{1}^\top (x^+ + x^-)$ by an `MPI_Allreduce` operation. For efficiency on systems with high per-message communication overhead, this operation could conceivably be combined with the reduction operation needed to compute q (step 3 immediately above).

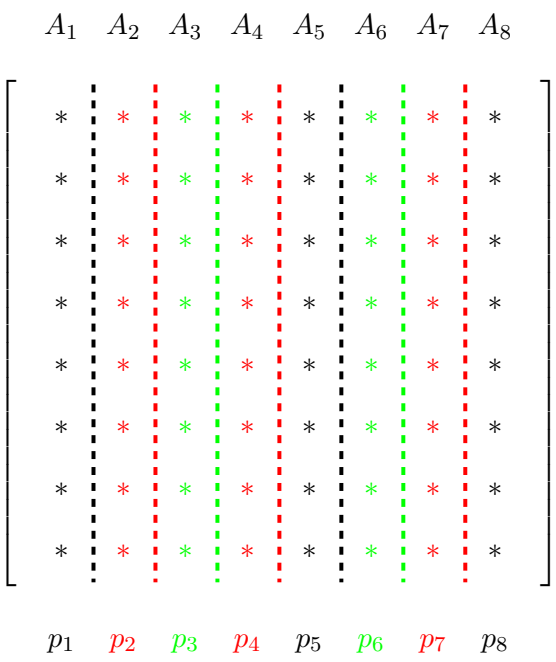


Figure 3.8: Column partition of dense matrix, where $P = 8$.

3. Each processor locally computes $r = q - b = A(x^+ - x^-) - b$ and then $h(x^+, x^-) = \frac{1}{2}\|r\|^2 + \nu\omega$.

At points (x^+, x^-) where the gradient $(y^+, y^-) = \nabla h(x^+, x^-)$ is also required, we proceed as follows:

1. Each processor locally computes $u_{(i)} = A_{(i)}^\top r = A_{(i)}^\top \big(A(x^+ - x^-) - b\big)$
2. Each processor locally computes $y_{(i)}^+ = u_{(i)} + \nu\mathbf{1}$ and $y_{(i)}^- = -u_{(i)} + \nu\mathbf{1}$, where $\mathbf{1}$ is an appropriate sized vector of all ones.

This procedure leaves the system with a representation of the gradient vector (y^+, y^-) that is distributed across processors in exactly the same manner as the decision variable vector (x^+, x^-). This is precisely what is needed for our parallel implementation of the SPG method (or any other first-order algorithm implementation). In summary, the function value and gradient may be calculated using one or two global reduction operations, local vector operations, and two local matrix multiplications (or just one if only the function value is needed). Because $n \gg m$, the local multiplications $A_{(i)}$

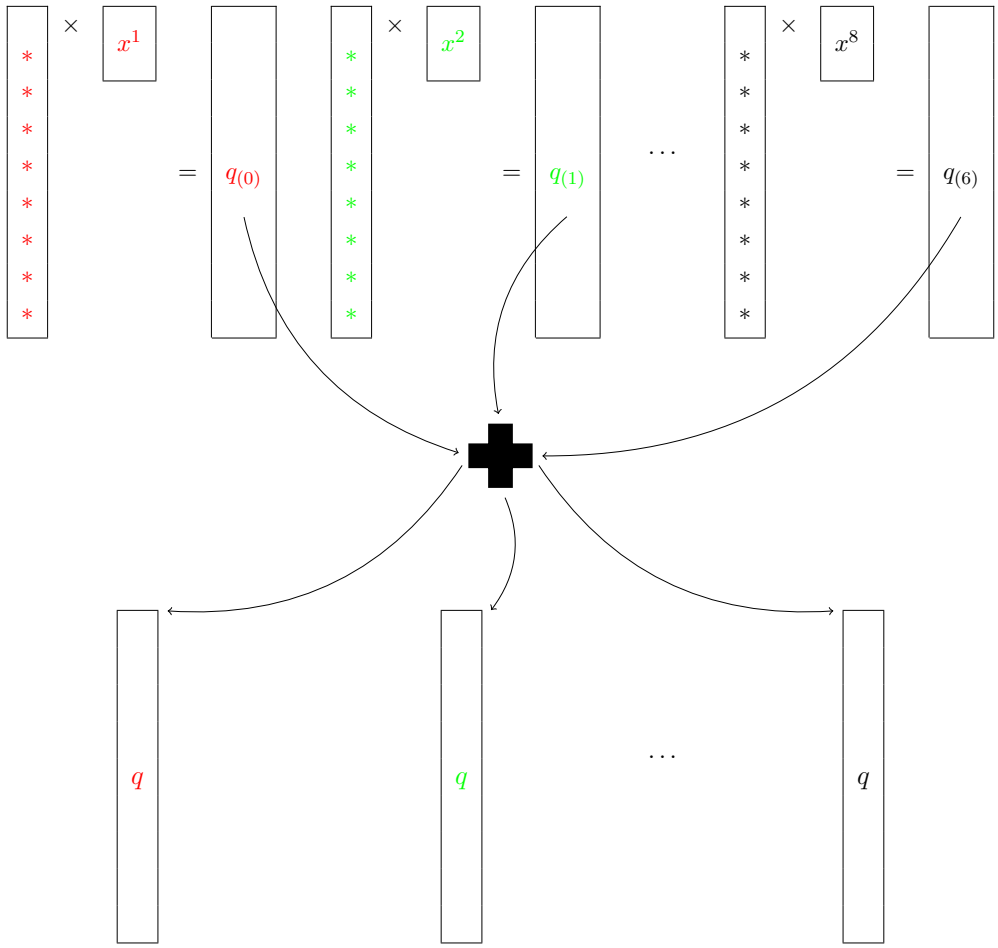


Figure 3.9: Column partition $q = A(x^+ - x^-)$

and $A^\top_{(i)}$ roughly involve the same amount of numerical work. The column partitioning scheme also balances the vector addition and scaling operations because it assigns between $\lfloor \frac{n}{P} \rfloor$ and $\lceil \frac{n}{P} \rceil$ indices j to each processor. When the matrix A is dense, these goals are compatible and attained by column partitioning. Figures 3.9 and 3.10 visually illustrate the above-described steps for the matrix shown in figure 3.7.

In terms of communication complexity, our most expensive operation is the reduction performed on the $q_{(i)}$s as described in Item 3. Most massively parallel supercomputers these days have interconnect networks that can perform reductions in $\mathrm{O}(\log P)$ time. Hence, the overall communication complexity is $\mathrm{O}(m \log P)$, where m is the size of the vectors $q_{(i)}$.

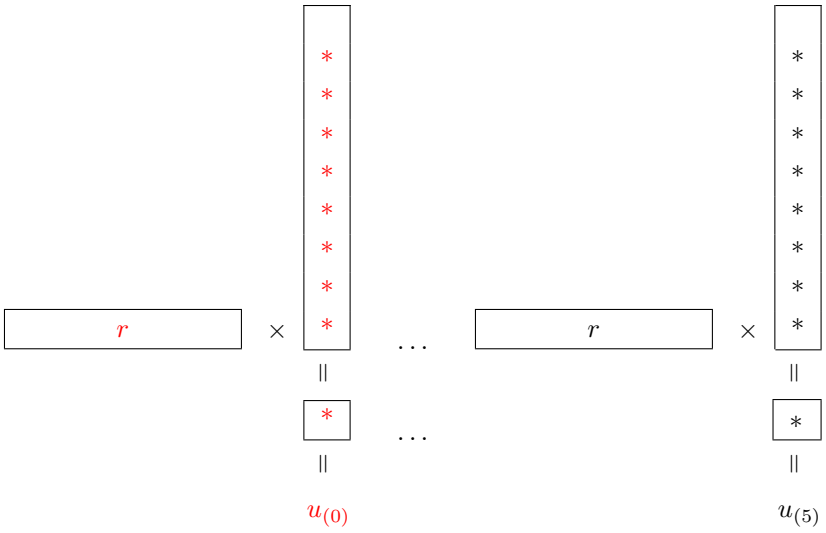


Figure 3.10: Column partition $u_{(i)} = r^T A_{(i)}$.

Distributed Sparse LASSO Problem

Before we get into the details of nonzero partitioning for sparse As, let us first discuss, why, in some sparse cases, column partitioning may not work well from a load balancing perspective. As mentioned earlier, the primary goal is to approximately balance the number of nonzero entries in the matrices $A_{(i)}$, as these nonzero counts essentially determine the amount of work required for the local matrix multiplications, the dominant portion of the workload. Matrices derived from real-world datasets, however, may have a small fraction of relatively dense columns containing a significant fraction of the total nonzeros. This can occur, for example, in text classification problems where certain letter combinations, such as "the", come up in virtually all documents, whereas others like "aaa" in only a few. Thus, simply dividing the indices $\{1 \ldots, n\}$ into P contiguous groups can result in a poor workload balance. Alternatively, one would have to solve the following problem: Given a list of integers $a_1, \ldots, a_n$, partition indices $\{1, \ldots, n\}$ into P sets $J_1, \ldots, J_P$ so as to minimize $\max_{i=1,\ldots,P}\{\sum_{j \in J_i} a_j\}$, where a_j stand for the number of nonzeros per column j, and P the number of processors. However, this problem is equivalent to the $\mathcal{NP}$-hard minimum makespan scheduling problem; see for example [76]. Though simple greedy approximation algorithms (factor 2) exist for this

problem, they would either require serial data read in, or very expensive communication. Therefore, rather than solving or approximating this problem at run time for each possible value of P, we will partition the nonzeros evenly between processors.

For the nonzero partitioning scheme we determine the partition members J_i by dividing the number of nonzeros, Z, into P contiguous groups, whose size varies by at most one element: formally, we may take the first $Z \mod P$ groups to have size $\lceil \frac{Z}{P} \rceil$ and the remaining groups to have size $\lfloor \frac{Z}{P} \rfloor$. This approach is shown in Figures 3.11 and 3.12, depicting the partition of 21 nonzeros of an 8 column matrix across 7 processors, where each processor ends up with exactly 3 nonzeros i.e., a prefect load balance. As figure 3.12 suggests the nonzero partitioning can assign parts of the same column to different processors. All local matrices $A_{(i)}$ are represented as sparse matrices in the processors p_i, if $A_{(i)}$ has a partial column then the missing nonzeros are simply treated as zeros. For example, take the local matrices $A_{(1)}$ and $A_{(2)}$ from 3.12, $A_{(1)}$ stored in p_0 is represented as a sparse matrix with 2 columns, which has 2 nonzeros in the first, and 1 nonzero in the second column, $A_{(2)}$ stored in p_1 is represented as a sparse matrix with 1 column, which has 3 nonzeros. This means that the global column A_2 appears in both processor p_0 and p_1, however, the nonzeros of A_2 are split between p_0 and p_1. This will affect how we divide coefficients of x^+ and x^- among the processors, since multiple $A_{(i)}$s can contain the same global column A_j. Whenever two or more processors are responsible for one column, we will call that column an *overlap zone*. For overlap zones we need to replicate coefficients of x^+ and x^- in as many processor as many own parts of that overlap zone. Given the example in Figure 3.12 for a vector $x \in \mathbb{R}^8$ we end up with the replication pattern shown in Figure 3.13, where x_2 is replicated twice, and x_4 three times.

Because we partition the nonzeros instead of the columns, we need to modify some of our linear algebra computations from the dense case. Nonzero partitioning and replication of coefficients of x^+ and x^- does not effect the matrix-vector multiplies $q = A(x^+ - x^-)$, because the missing local data will be completed during the reduction operation performed at the end of the matrix-vector multiply $\left(q = \sum_{i=1}^{P} q_{(i)}\right)$. But when it comes to vector-transpose-matrix multiplies $u = r^\top A$, which is communication

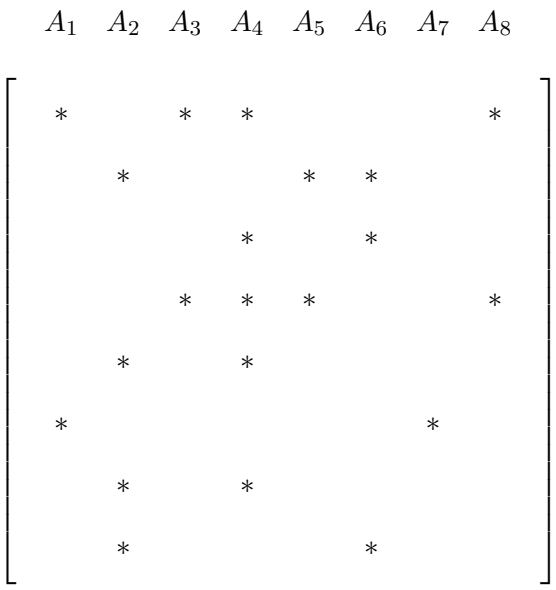


Figure 3.11: Sparse Matrix, nonzeros indicated by *.

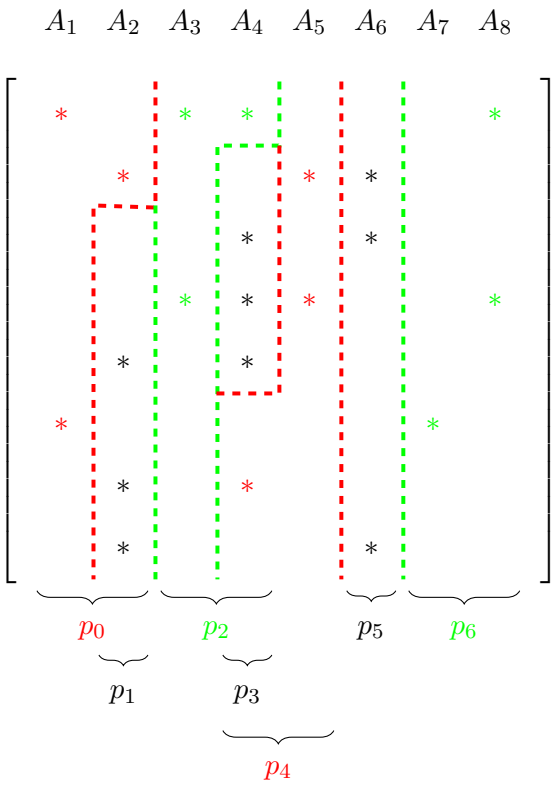


Figure 3.12: Even partition of nonzeros between processors.

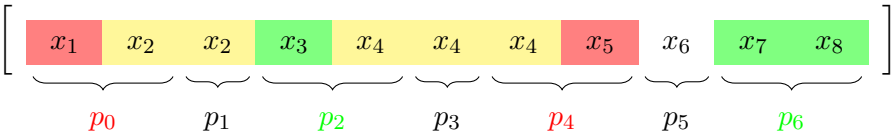


Figure 3.13: Representation of x given the nonzero partition in Figure 3.12.

free in its original form, the local computations $u_{(i)} = r^\top A_{(i)} = \big(A(x^+ - x^-) - b\big)^\top A_{(i)}$ need to be followed by a scalar sum-reduction and broadcast within each overlap zone to complete the missing data. This requires the `DistributedSparseLassoProblem` to set up a communicator for each overlap zone. These communicators only need to be set up once during runtime. The reduction on them will take $O(\log \pi)$ steps, where $\pi \leq P$ is the maximum number of processors in an overlap zone. Setting up the communicators can be achieved with the help of parallel scans [65] in $O(\log P)$ time, therefore, these operations do not worsen the communication complexity relative to the column partitioning approach.

The modified gradient $\nabla h(x^+, x^-)$ computation is as follows:

1. Each processor locally computes $u_{(i)} = r^\top A_{(i)} = \big(A(x^+ - x^-) - b\big)^\top A_{(i)}$
2. Processors in an overlap zone perform an `MPI_Allreduce` on a single coordinate u^j to adjust for the missing values, where the `MPI` communicator for the reduction operation consists of the processors in the overlap zone.

Notice that the distributed vector class will also need to adjust the segment information to a make sure that the inner product and norm computations only include replicated variables once. For example, $\mathbf{1}^\top(x^+ + x^-)$ also changes as follows:

1. Each processor i locally calculates $\omega_i = \mathbf{1}_{(i)}^\top(x_{(i)}^+ + x_{(i)}^-)$, where each coefficient x^j is included only once.
2. We sum $\omega_i \in \mathbb{R}$ with an `MPI_Allreduce` operation to get $\omega = \sum_{i=1}^P \omega_i$ across all processors.

These details are all illustrated in Figures 3.14 and 3.15.

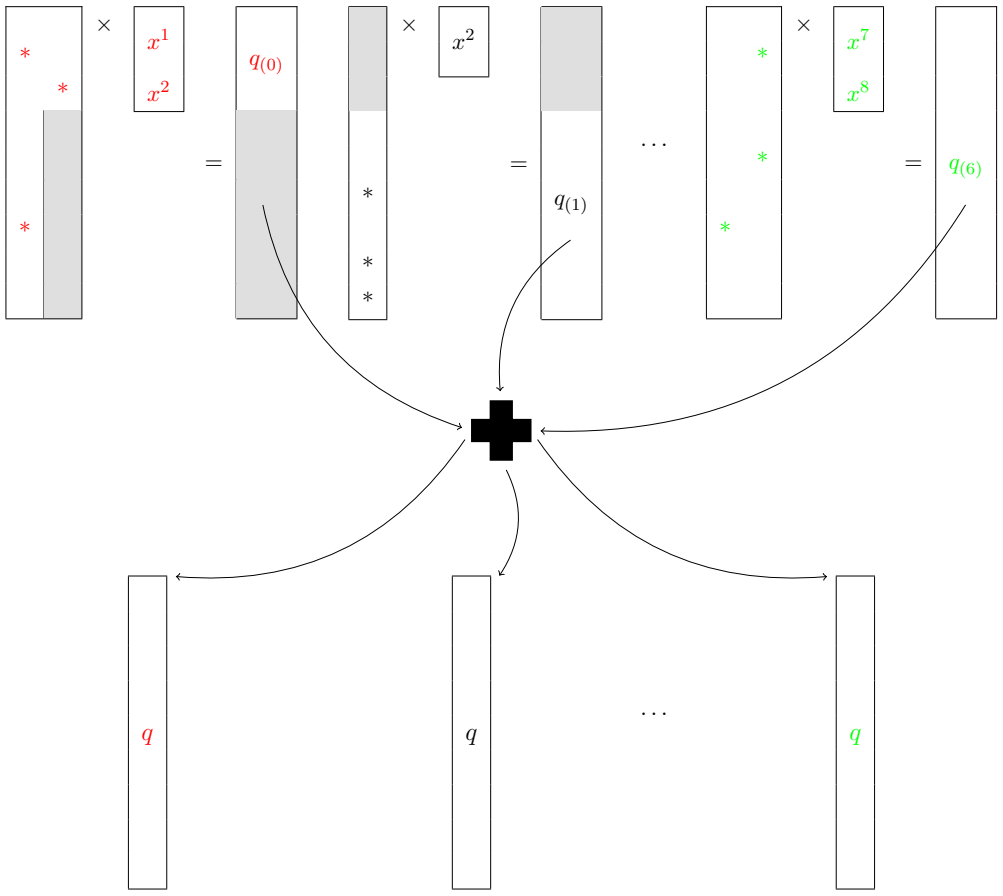


Figure 3.14: Nonzero partition $q = A(x^+ - x^-)$.

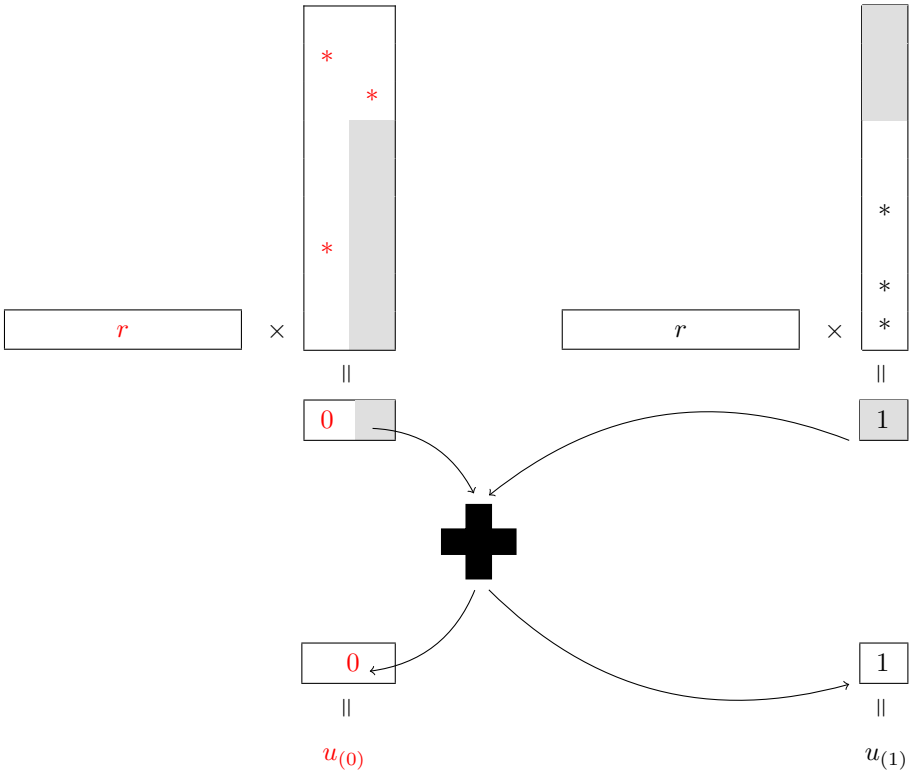


Figure 3.15: Nonzero partition $u_{(i)} = r^T A_{(i)}$, processor 0 and 1 are in one group

Parallel I/O

Before we continue to present computational results, we will briefly mention how we perform I/O operations in parallel. In our implementation, we used the Lustre file system to "stripe" our test input data, storing it across multiple disks. We then used `MPI`'s parallel I/O functions [65], such as `MPI_File_seek` and `MPI_File_read`, to read in the binary data files. In a binary file, each processor seeks to the position where its data starts, and then reads in an appropriate portion of the data. This approach prevents data read-in from being a bottleneck operation detrimental to scalability.

3.4.3 Computational Results

We now present four strong scaling results for `OPSPG` on various LASSO problems, two of which are derived from real-world datasets, and two of which we randomly generated. We ran the tests on the TACC Stampede supercomputer. Each node of this system consists of two 2.7 GHz 8-core Xeon processors with 32GB of RAM, and the nodes are connected by an InfiniBand network with a fat-tree-class topology. This system also has "Xeon Phi" accelerator cards, but for portability reasons we did not attempt to use them.

Text Classification

This sparse data set [44, 51] is a two-class variant of the UCI "Twenty Newsgroups" data set. The problem involves classifying messages as positive or negative in tone. The dataset is sparse, with A having about 1.35 million columns and 20,000 rows. As shown in Figure 3.16, our algorithm exhibits nearly linear scaling behavior through 256 processor cores, after which little further speedup is obtained, however, at that point, the total computation time is about 1 second.

Classification With Traffic Data

"PEMS-SF" is a dense data set from the UCI Machine Learning Repository [4], consisting of 15 months' worth of daily data describing the occupancy rate, across time,

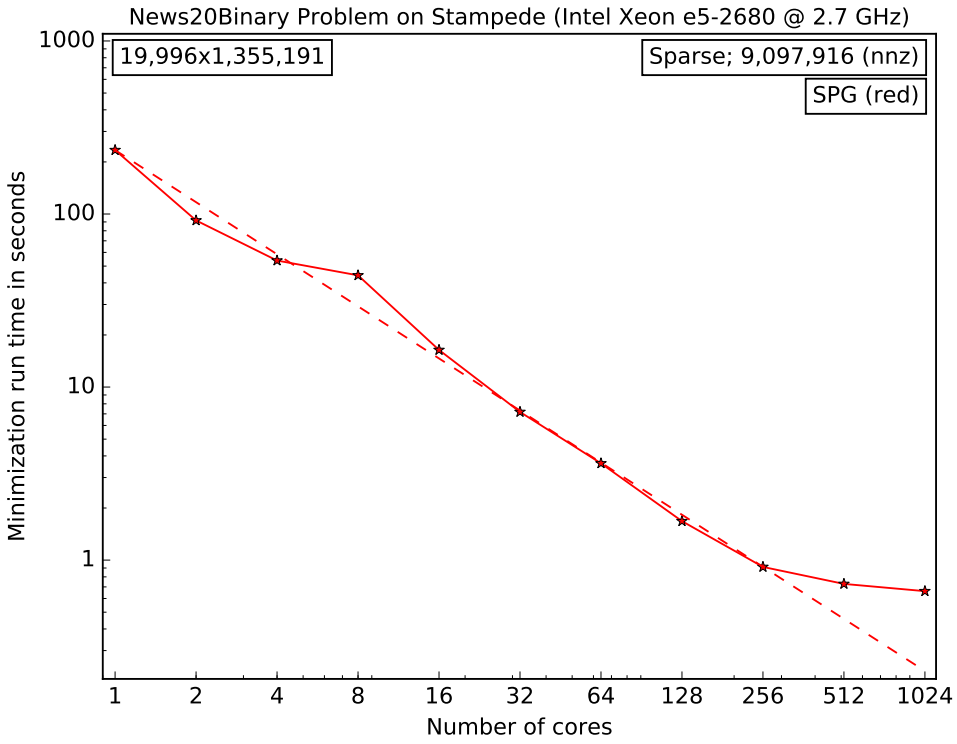


Figure 3.16: SPG speedup performance on text classification.

of various travel lanes in the San Francisco area freeway system. It has 267 rows and about 139,000 columns. The associated task proposed in the repository is to classify each observed day as the correct day of the week, from Monday to Sunday. To produce reasonable related LASSO instances, we first transformed this seven-class classification problem into a two-class variant by simply attempting to classify each observed day as either a weekend day or a weekday. As shown in Figure 3.17, our SPG obtains good speedups through 256 processors on this instance, but little or no additional speedup for higher processor counts.

Randomly Generated Instances

In this subsection, we give performance results for two large-scale instances that we randomly generated, one sparse and one dense, both having $10,000$ rows and 2.5 million columns. The sparse dataset has a density of 1%, with nonzeroes located randomly. In

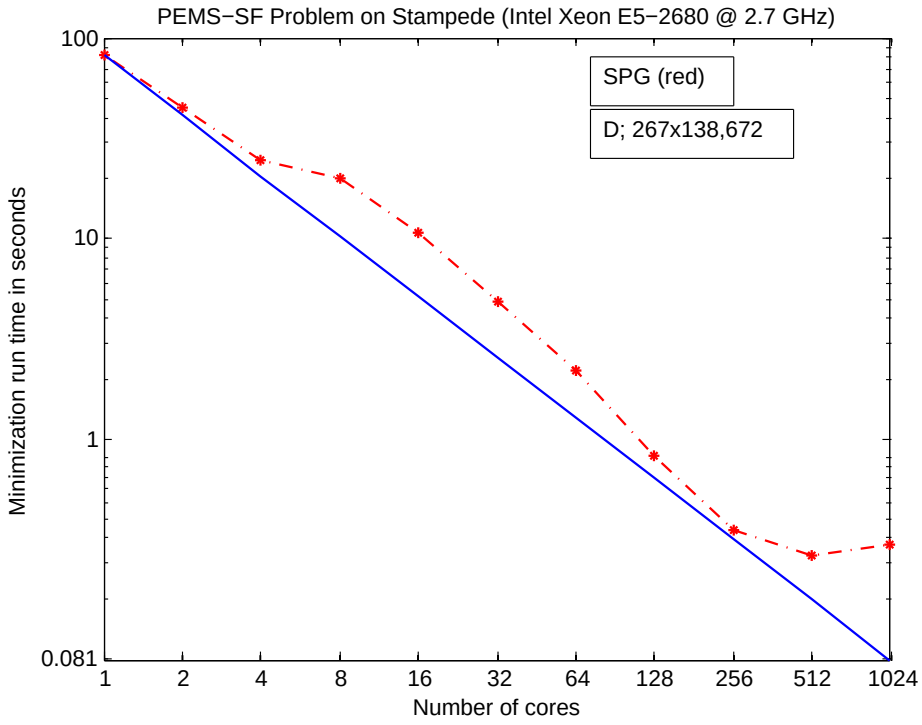


Figure 3.17: SPG speedup performance on traffic dataset.

both cases, the nonzero matrix coefficients were generated from a unit normal distribution. These examples serve to illustrate that when the processor/data ratio is sufficiently large, good scaling can be achieved for large numbers of processors. The dense data set consumes 200GB in binary format whereas the sparse dataset requires only 2GB. Results are shown in Figures 3.18 and 3.19. We solve the 200GB dense instance in 28.5 seconds on 4096 processors, with the algorithm performing 2448 matrix-vector multiplications. Speedups still appear to be nearly linear at this number of processors. Note that, due to its size, we did not attempt to solve this instance below 128 processor cores. The sparse dataset can be solved on a single processor, and speedups appear to be near-linear through 1024 processors.

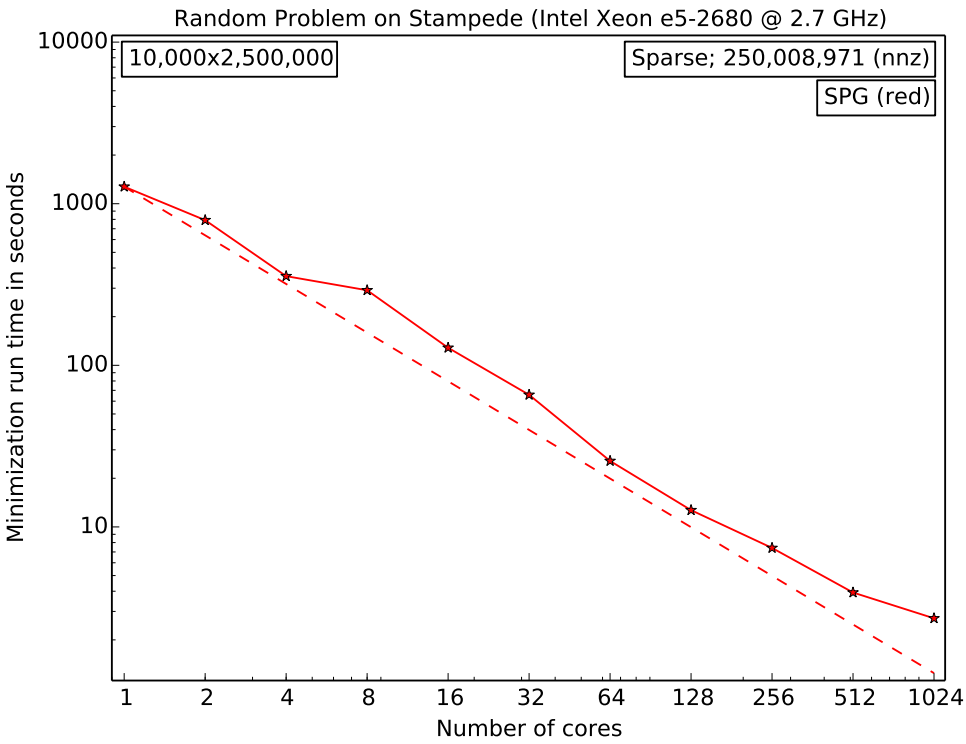


Figure 3.18: SPG speedup performance on random sparse data.

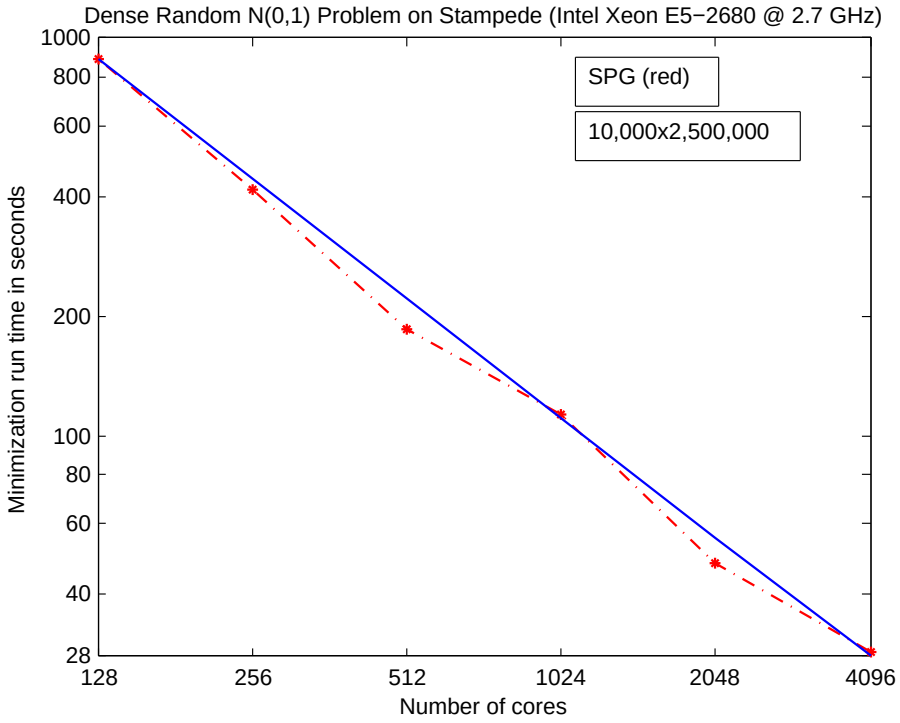


Figure 3.19: SPG speedup performance on random dense data.

Chapter 4

Object-Parallel Augmented Lagrangian Algorithm Applied to Stochastic Programs

We start the discussion of this chapter by first introducing proximal point algorithms [8], followed by the derivation of the augmented Lagrangian algorithm also known as the *method of multipliers* [7, 10, 8]. The next section gives a brief description of `ALGENCAN` [12], a serial implementation of the augmented Lagrangian algorithm developed by Birgin and Martínez [12], that works well for problems of the form

$$\begin{aligned} \min \quad & f(x) \\ \text{s.t.} \quad & h(x) = 0 \\ & g(x) \leq 0 \\ & x \in X, \end{aligned} \tag{4.1}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is convex, $h : \mathbb{R}^n \to \mathbb{R}^m$ is affine, and $g(x) = (g_1(x), \ldots, g_l(x))$, where $g_1, \ldots, g_l : \mathbb{R}^n \to \mathbb{R}$ are convex, f and g are continuously differentiable and $X \subset \mathbb{R}^n$ is a box $X = \{x \in \mathbb{R}^n \mid l \leq x \leq u\}$, where l and u are vectors respectively specifying lower and upper bounds. This is followed by a description of `OPAL`, which is an object-parallel implementation of the augmented Lagrangian method based on `ALGENCAN`. Finally, we close the chapter by discussing the solution of linear stochastic programming problems using `OPAL` and `PySP`, an algebraic modeling language for stochastic programs.

4.1 Proximal Minimization Algorithm

Proximal minimization algorithms are a class of iterative approximation methods used to minimize a convex function $f : \mathbb{R}^n \to \mathbb{R}$ over a convex set X, and more general problems. Iterative algorithms for minimizing f ideally generate a sequence $\{x_k\}$ that converges to an optimal solution. The sequence $\{x_k\}$ may not be feasible and is obtained

by solving at each iteration k an approximation to the original optimization problem. In other words, instead of minimizing f over X we approximately solve a series of problems of the form

$$x_{k+1} \in \arg\min_{x \in X_k} F_k(x), \tag{4.2}$$

where F_k is a function that approximates f and X_k is a set that approximates X. The goal is that minimization of F_k over X_k should be easier than minimization of f over X. The optimal solution x_{k+1} to the approximating problem is used to improve the next approximation F_{k+1} and X_{k+1}.

In particular, we consider the following algorithm

$$x_{k+1} \in \arg\min_{x \in X} \left\{ f(x) + \frac{1}{2\rho_k} \|x - x_k\|^2 \right\}, \tag{4.3}$$

where x_0 is an arbitrary starting point and $\rho_k > 0$ is a scalar parameter. This is the *proximal minimization algorithm* [61]. For a closed proper convex function $f : \mathbb{R}^n \to (-\infty, +\infty]$ the quadratic term in (4.3) makes the minimand strictly convex so it has a unique minimum [9, p. 117]. In other words, the proximal minimization algorithm "regularizes" the minimization of f. It is worthwhile to note that the sequence $\{f(x_k)\}$ generated by the $\{x_k\}$s from the proximal minimization algorithm is monotonically nonincreasing. This can be seen by the following inequality:

$$f(x_{k+1}) + \frac{1}{2\rho_k} \|x_{k+1} - x_k\|^2 \leq f(x_k) + \frac{1}{2\rho_k} \|x_k - x_k\|^2 = f(x_k) \tag{4.4}$$

As $\rho_k \to \infty$ the quadratic regularization term becomes small and the proximal minimization (4.3) approximates the minimization of f more closely. Thus, the connection with the approximation approach.

Theorem 4.1.1 (Basic convergence result for proximal minimization algorithms [8, p. 237])**.** *Let f' denote the optimal value $f' = \inf_{x \in \mathbb{R}^n} f(x)$, (possibly $-\infty$) and by X' denote set of minima of f (possibly empty), $X' = \operatorname{Arg\,min}_{x \in \mathbb{R}^n} f(x)$. Let $\{x_k\}$ be a sequence generated by the proximal minimization algorithm* (4.3)*. Then, if $\sum_{k=0}^{\infty} \rho_k = \infty$, we have*

$$f(x_k) \downarrow f', \tag{4.5}$$

and if X' is nonempty, $\{x_k\}$ converges to some point in X'.

4.2 An Augmented Lagrangian Algorithm

In short, the augmented Lagrangian algorithm is a proximal minimization algorithm applied to the dual of an optimization problem. Consider the primal problem from (4.1), repeated here for convenience,

$$\begin{aligned} \min \quad & f(x) \\ \text{s.t.} \quad & h(x) = 0 \\ & g(x) \leq 0 \\ & x \in X, \end{aligned} \tag{4.6}$$

and its dual function

$$q_(\lambda, \mu) = \inf_{(t,u) \in \mathbb{R}^m \times \mathbb{R}^l} \left\{ \langle \lambda, t \rangle + \langle \mu, u \rangle + p(t,u) \right\}, \tag{4.7}$$

where $p(t,u) = \inf_{x \in \mathbb{R}^n} \{F(x,t,u)\}$ is the *parametric value function* and $F(x,t,u)$ is the *parameterized primal function*

$$F(x,t,u) = \begin{cases} f(x) & \text{if} \quad x \in C_{t,u} \\ +\infty & \text{otherwise} \quad , \end{cases} \tag{4.8}$$

where $C_{t,u}$ is the *perturbed constraint set*

$$C_{t,u} = \{ x \in X \mid h(x) = t,\ g(x) \leq u \}. \tag{4.9}$$

Let us apply the proximal minimization algorithm to the dual problem

$$\begin{aligned} \max \quad & q(\lambda, \mu) \\ \text{s.t.} \quad & (\lambda, \mu) \in \mathbb{R}^m \times \mathbb{R}^l. \end{aligned} \tag{4.10}$$

We get the following sequence of iterates

$$(\lambda_{k+1}, \mu_{k+1}) \in \operatorname*{arg\,max}_{(\lambda,\mu) \in \mathbb{R}^m \times \mathbb{R}^l} \left\{ -\frac{1}{2\rho_k} \left(\|\lambda - \lambda_k\|^2 + \|\mu - \mu_k\|^2 \right) + q(\lambda, \mu) \right\}, \tag{4.11}$$

or equivalently

$$(\lambda_{k+1}, \mu_{k+1}) \in \operatorname*{arg\,min}_{(\lambda,\mu) \in \mathbb{R}^m \times \mathbb{R}^l} \left\{ \frac{1}{2\rho_k} \left(\|\lambda - \lambda_k\|^2 + \|\mu - \mu_k\|^2 \right) - q(\lambda, \mu) \right\}. \tag{4.12}$$

The relationship between conjugacy and duality [9, p. 137-138] tells us that $-q(\lambda, \mu) = p^*(-\lambda, -\mu)$, so substituting q by $-p^*$ we get

$$(\lambda_{k+1}, \mu_{k+1}) \in \operatorname*{arg\,min}_{(\lambda,\mu) \in \mathbb{R}^m \times \mathbb{R}^l} \left\{ \frac{1}{2\rho_k} \left(\|\lambda - \lambda_k\|^2 + \|\mu - \mu_k\|^2 \right) + p^*(-\lambda, -\mu) \right\}. \tag{4.13}$$

Claim 4.2.1 (Regularization term is a conjugate function)**.** *The regularization term*

$$R^*(\lambda,\mu) = \frac{1}{2\rho_k}\left(\|\lambda-\lambda_k\|^2 + \|\mu-\mu_k\|^2\right) \tag{4.14}$$

is the convex conjugate of the function

$$R(t,u) = \langle\lambda_k, t\rangle + \frac{\rho_k}{2}\|t\|^2 + \langle\mu_k, u\rangle + \frac{\rho_k}{2}\|u\|^2. \tag{4.15}$$

Proof of Claim 4.2.1.

$$\begin{aligned} R^*(\lambda,\mu) &= \sup_{(t,u)\in\mathbb{R}^m\times\mathbb{R}^l} \{\langle\lambda,t\rangle + \langle\mu,u\rangle - R(t,u)\} \\ &= \sup_{(t,u)\in\mathbb{R}^m\times\mathbb{R}^l} \left\{\langle\lambda,t\rangle + \langle\mu,u\rangle - \left(\langle\lambda_k,t\rangle + \frac{\rho_k}{2}\|t\|^2 + \langle\mu_k,u\rangle + \frac{\rho_k}{2}\|u\|^2\right)\right\} \end{aligned}$$

This maximization problem above can be solved by setting the derivative equal to zero since the function is convex and differentiable. We get the following solution

$$\begin{aligned} t &= \tfrac{1}{\rho_k}(\lambda-\lambda_k) \\ u &= \tfrac{1}{\rho_k}(\mu-\mu_k). \end{aligned}$$

Substituting for t and u the expressions above proves the claim by yielding

$$R^*(\lambda,\mu) = \frac{1}{2\rho_k}\left(\|\lambda-\lambda_k\|^2 + \|\mu-\mu_k\|^2\right). \tag{4.16}$$

□

Therefore, the proximal minimization algorithm applied to the dual can be reformulated as

$$(\lambda_{k+1},\mu_{k+1}) \in \operatorname*{arg\,min}_{(\lambda,\mu)\in\mathbb{R}^m\times\mathbb{R}^l} \{R^*(\lambda,\mu) + p^*(-\lambda,-\mu)\}. \tag{4.17}$$

We know that R is closed and convex. If $\exists\ (\tilde{t},\tilde{u}) \in \mathbb{R}^m\times\mathbb{R}^l$ and $\tilde{\gamma}\in\mathbb{R}$ such that the set

$$\left\{x \mid F(x,\tilde{t},\tilde{u}) \le \tilde{\gamma}\right\}$$

is nonempty and compact then according to the partial minimization theorem [9, p. 124] p is closed and convex. Let us take

$$(t_{k+1},u_{k+1}) \in \operatorname*{arg\,min}_{(t,u)\in\mathbb{R}^m\times\mathbb{R}^l} \{R(t,u) + p(t,u)\}, \tag{4.18}$$

which in turn may be written as

$$(t_{k+1}, u_{k+1}) \in \underset{(t,u)\in\mathbb{R}^m\times\mathbb{R}^l}{\arg\min} \left\{ \langle \lambda_k, t\rangle + \frac{\rho_k}{2}\|t\|^2 + \langle \mu_k, u\rangle + \frac{\rho_k}{2}\|u\|^2 + p(t,u) \right\} \tag{4.19}$$

then, according to the Fenchel duality theorem [9, p. 179] the following is true

$$(\lambda_{k+1}, \mu_{k+1}) \in \partial R(t_{k+1}, u_{k+1}),$$

yielding

$$\begin{aligned} \lambda_{k+1} &= \lambda_k + \rho_k t_{k+1} \\ \mu_{k+1} &= \mu_k + \rho_k u_{k+1}. \end{aligned} \tag{4.20}$$

Expression (4.19) is the augmented Lagrangian function, where (λ_k, μ_k) are the multipliers and $\rho_k > 0$ the penalty parameter. To get the standard form of the augmented Lagrangian algorithm we have to expand the minimization problem in (4.19)

$$\begin{aligned} &\inf_{(t,u)\in\mathbb{R}^m\times\mathbb{R}^l} \left\{ \langle \lambda_k, t\rangle + \frac{\rho_k}{2}\|t\|^2 + \langle \mu_k, u\rangle + \frac{\rho_k}{2}\|u\|^2 + p(t,u) \right\} &= \\ &\inf_{(t,u)\in\mathbb{R}^m\times\mathbb{R}^l} \left\{ \langle \lambda_k, t\rangle + \frac{\rho_k}{2}\|t\|^2 + \langle \mu_k, u\rangle + \frac{\rho_k}{2}\|u\|^2 + \inf_{x\in\mathbb{R}^n} \{F(x,t,u)\} \right\} &= \\ &\inf_{x\in C_{t,u},(t,u)\in\mathbb{R}^m\times\mathbb{R}^l} \left\{ f(x) + \langle \lambda_k, t\rangle + \frac{\rho_k}{2}\|t\|^2 + \langle \mu_k, u\rangle + \frac{\rho_k}{2}\|u\|^2 \right\}. \end{aligned} \tag{4.21}$$

If $x \in C_{t,u}$ then we know that $h(x) = t$, and that $g(x) \leq u$. Thus we can write the last line of (4.21) as

$$\inf_{x\in X} \left\{ f(x) + \langle \lambda_k, h(x)\rangle + \frac{\rho_k}{2}\|h(x)\|^2 + \inf_{\left\{u\in\mathbb{R}^l | g(x)\leq u\right\}} \left\{ \langle \mu_k, u\rangle + \frac{\rho_k}{2}\|u\|^2 \right\} \right\}. \tag{4.22}$$

The inner problem $\inf_{\left\{u\in\mathbb{R}^l | g(x)\leq u\right\}} \left\{ \langle \mu_k, u\rangle + \frac{\rho_k}{2}\|u\|^2 \right\}$ separates into l one-dimensional problems of the form

$$\begin{aligned} \min \quad & \mu_i^k u_i + \tfrac{\rho_k}{2} u_i^2 \\ \text{s.t.} \quad & g_i(x) \leq u_i \end{aligned} \tag{4.23}$$

This is a quadratic minimization problem. If the solution u_i' is below $g_i(x)$ we project back to the feasible set by setting u_i' equal to $g_i(x)$. To minimize the quadratic, we set its derivative to zero

$$(\mu_i^k u_i + \frac{\rho_k}{2} u_i^2)' = \mu_i^k + \rho_k u_i = 0, \tag{4.24}$$

which gives us

$$u_i' = -\frac{\mu_i^k}{\rho_k} \tag{4.25}$$

for the optimal solution unless $-\frac{\mu_i^k}{\rho_k} < g_i(x)$ or equivalently $\mu_i^k + \rho_k g_i(x) > 0$, in which case $u_i' = g_i(x)$. Therefore, we differentiate between the following two cases:

1. $\mu_i^k + \rho_k g_i(x) \leq 0$, where $u_i' = -\dfrac{\mu_i^k}{\rho_k}$ and the optimal solution value becomes the constant $-\dfrac{1}{2\rho_k}(\mu_i^k)^2$.
2. $\mu_i^k + \rho_k g_i(x) > 0$, where $u_i' = g_i(x)$ and yielding the optimal value $\dfrac{1}{2\rho_k}\left(\mu_i^k + \rho_k g_i(x)\right) - \dfrac{1}{2\rho_k}(\mu_i^k)^2$.

This gives us the following closed formula

$$\inf_{\left\{u \in \mathbb{R}^l | g(x) \leq u\right\}} \left\{ \langle \mu_k, u \rangle + \frac{\rho_k}{2}\|u\|^2 \right\} = \|\max\{0, \mu_k + \rho_k g(x)\}\|^2 - \frac{1}{2\rho_k}\|\mu_k\|^2, \tag{4.26}$$

where the "max" operator is applied componentwise. The μ multiplier update formula from (4.20) changes in the following manner

$$\begin{array}{rcll} \mu_{k+1} & = & \mu_k + \rho_k\left(-\dfrac{1}{\rho_k}\mu_k\right) = 0, & \text{if } \mu^k + \rho_k g(x) \leq 0 \\ \mu_{k+1} & = & \mu_k + \rho_k\left(g(x)\right), & \text{if } \mu^k + \rho_k g(x) > 0. \end{array} \tag{4.27}$$

Consolidating the above, we get the single formula

$$\mu_{k+1} = \max\{0, \mu_k + \rho_k g(x)\}. \tag{4.28}$$

Combining everything, we obtain the standard augmented Lagrangian algorithm,

$$\begin{array}{rcl} x_{k+1} & \in & \arg\min\limits_{x \in X} \quad \Big\{ f(x) + \langle \lambda_k, h(x)\rangle + \dfrac{\rho_k}{2}\|h(x)\|^2 + \\ & & \qquad\qquad \|\max\{0, \mu_k + \rho_k g(x)\}\|^2 - \frac{1}{2\rho_k}\|\mu_k\|^2 \Big\} \\ \lambda_{k+1} & = & \lambda_k + \rho_k h(x_{k+1}) \\ \mu_{k+1} & = & \max\{0, \mu_k + \rho_k g(x_{k+1})\}, \end{array} \tag{4.29}$$

where $\{\rho_k\}$ is a sequence of scalars with $\inf\limits_k \{\rho_k\} > 0$. The minimization above can be simplified by removing the constant term $-\frac{1}{2\rho_k}\|\mu_k\|^2$, giving us

$$\begin{array}{rcl} x_{k+1} \in & \arg\min\limits_{x \in X} \left\{ f(x) + \langle \lambda_k, h(x)\rangle + \dfrac{\rho_k}{2}\|h(x)\|^2 + \left\|[\mu_k + \rho_k g(x)]_+\right\|^2 \right\} \\ \lambda_{k+1} = & \lambda_k + \rho_k h(x_{k+1}) \\ \mu_{k+1} = & [\mu_k + \rho_k g(x_{k+1})]_+, \end{array} \tag{4.30}$$

where we use $[\,\cdot\,]_+$ to denote the componentwise $\max\{0, \cdot\}$ operation. We refer to the function minimized at every iteration as the augmented Lagrangian function

$$\mathcal{L}_{\rho_k}(x, \lambda_k, \mu_k) = f(x) + \langle \lambda_k, h(x) \rangle + \frac{\rho_k}{2} \|h(x)\|^2 + \left\| [\mu_k + \rho_k g(x)]_+ \right\|^2. \tag{4.31}$$

We can say the following about the convergence properties of the augmented Lagrangian method:

Claim 4.2.2 (Convergence properties of the augmented Lagrangian algorithm [8, p. 262-263]). *Given a sequence* $\{(x_k, \lambda_k, \mu_k)\}$ *generated by the augmented Lagrangian algorithm applied to the primal problem in* (4.1) *such that* $\inf_k \{\rho_k\} > 0$, *then* $\{(\lambda_k, \mu_k)\}$ *converges to an optimal dual solution and every limit point of* $\{x_k\}$ *is an optimal solution of the primal problem, provided that the dual problem* $\sup_{(\lambda,\mu)\in\mathbb{R}^m\times\mathbb{R}^l} \{q(\lambda, \mu)\}$, *has at least one optimal solution.*

Given an x', obtained via the augmented Lagrangian method, we say that it satisfies the Karush-Kuhn-Tucker (KKT) necessary conditions of optimality with respect to the minimization in (4.1) if x' is feasible and there exists $\lambda' \in \mathbb{R}^m$ and $\mu' \in \mathbb{R}^l_+$ such that

$$\nabla_x \mathcal{L}(x', \lambda', \mu') = 0 \text{ and } \mu'_i = 0 \text{ whenever } g_i(x') < 0, \ \forall\, i = 1 \ldots l. \tag{4.32}$$

In general, the termination condition for augmented Lagrangian algorithms involves some form of the KKT conditions.

4.3 ALGENCAN: A Practical Augmented Lagrangian Method

The sequence of steps presented in (4.30) as the augmented Lagrangian algorithm is referred to as the *outer loop*, whereas the minimization of $\mathcal{L}_{\rho_k}(x, \lambda_k, \mu_k)$ is referred to as the *inner loop*. Different versions of the augmented Lagrangian algorithm arise based on design decisions taken regarding the inner loop and the update procedure of the penalty parameter ρ, which were not specified in the previous section. `ALGENCAN` uses an active-set method called `GENCAN` to solve the subproblem

$$\begin{aligned} \min \quad & \mathcal{L}_{\rho_k}(x, \lambda_k, \mu_k) \\ \text{s.t.} \quad & x \in X. \end{aligned} \tag{4.33}$$

We begin by describing `ALGENCAN` using a top-down approach, starting with the outer loop of the augmented Lagrangian method and then moving on to the inner loop. Using KKT type termination conditions [12] an abstract description of the outer loop is depicted in Algorithm 6 below:

Algorithm 6: Augmented Lagrangian Outer Loop

Input: $f,\ h,\ g,\ X,\ x_0 \in \mathbb{R}^n,\ \lambda_0 \in \mathbb{R}^m,\ \mu_0 \in \mathbb{R}^l,\ \rho_0 > 0,\ \epsilon_{\text{opt}},\ \epsilon_{\text{feas}}$

Output: x' minimizer of f such that $x' \in X,\ h(x') = 0,\ g(x') \leq 0$

$k = 0$

$x_{k+1} \in \arg\min_{x \in X} \{\mathcal{L}_{\rho_k}(x, \lambda_k, \mu_k)\}$

while $\left\|\nabla_x^P \mathcal{L}_{\rho_k}(x_{k+1}, \lambda_k, \mu_k)\right\|_\infty > \epsilon_{opt}$ *and* $\Psi_{\rho_k}(x_{k+1}, \mu_k) > \epsilon_{feas}$ **do**

  $\lambda_{k+1} = \lambda_k + \rho_k h(x_{k+1})$

  $\mu_{k+1} = [\mu_k + \rho_k g(x_{k+1})]_+$

  $\rho_{k+1} \leftarrow$ `updateRho()`

  $k = k + 1$

  $x_{k+1} \in \arg\min_{x \in X} \{\mathcal{L}_{\rho_k}(x, \lambda_k, \mu_k)\}$

$x' = x_{k+1}$

Where $x_0 \in \mathbb{R}^n,\ \lambda_0 \in \mathbb{R}^m,\ \mu_0 \in \mathbb{R}^l_+,\ \rho_0 > 0$ are arbitrary starting values, ϵ_{opt} and ϵ_{feas} are tolerance values, $\Psi_\rho : \mathbb{R}^n \times \mathbb{R}^l_+ \to \mathbb{R}_+$ is a function checking for complementary slackness and feasibility violations, defined as

$$\Psi_\rho(x, \mu) = \max\left\{\|h(x)\|_\infty,\ \left\|\max\left\{-\frac{1}{\rho}\mu,\ g(x)\right\}\right\|_\infty\right\}, \tag{4.34}$$

and `updateRho()` is a "black box" method to update the penalty parameter ρ. The augmented Lagrangian outer loop presented in Algorithm 6 achieves our desired level of abstraction since the operations aside from various function calls all consists of `BLAS` level 1 vector operations. The next step in the discussion of `ALGENCAN` is a description of the subproblem solver `GENCAN`, which follows in the next section.

4.3.1 `GENCAN`: Solving the Subproblem via an Active-Set Method

Before we start to discuss the algorithm itself, we provide some definitions that are used throughout this section. We say that at the kth iteration the ith box constraint is *active*, if $x_i^k = l_i$ or $x_i^k = u_i$. We denote by $A(x)$ the set of active indices, such that

$$A(x) = \{i \mid \text{if } x_i = l_i \text{ or } x_i = u_i\}. \tag{4.35}$$

We will denote by $\bar{n}$ the number of free variables, i.e. $\bar{n} = n - |A(x)|$. Let us define for all $I \subset \{1, 2, \ldots, n, n+1, \ldots, 2n\}$

$$F_I = \{x \in X \mid x_i = l_i \text{ if } i \in I,\ x_i = u_i \text{ if } n + i \in I,\ l_i < x_i < u_i \text{ otherwise}\}. \quad (4.36)$$

Let V_I be the smallest affine subspace that contains F_I and S_I the parallel linear subspace to V_I. Let us denote by P_{S_I} the projection onto S_I, which from a computational perspective amounts to setting the coefficients $i \in A(x)$ to zero. We will refer to this operation as the *mask* operation symbolized by operator $\widehat{\cdot}$. The mask operator applied to the gradient of a function f at x corresponds to projecting the gradient at x onto S_I, i.e.

$$\widehat{\nabla} f(x) = P_{S_I}\left(\nabla f(x)\right). \quad (4.37)$$

The gradient of $\widehat{\nabla} f$ at x, which we refer to as the masked Hessian of f at x, is denoted by $\widehat{\nabla}^2 f(x)$. We define the *internal gradient* of a function f at x as the projection of the projected gradient onto the set S_I, denoted by

$$\widehat{\nabla}^P f(x) = P_{S_I}\left(P_X(x - \nabla f(x)) - x\right), \quad (4.38)$$

where X is a box as defined in (4.1). `GENCAN` does not explicitly use a mask operator. However, `OPAL` does, and therefore we have decided to describe `GENCAN` using this notation. $\mathcal{L}(x)$ will denote $\mathcal{L}_\rho(x, \lambda, \mu)$ since for the inner loop ρ, λ and μ are given constants fixed by the outer loop of the augmented Lagrangian algorithm.

Active-set methods for box constrained problems aim to identify a set of active box constraints that tentatively belong to the optimal face of the feasible region and then work towards finding the optimal solution in a reduced space $\mathbb{R}^{\bar{n}}$ by fixing the values of all x_i with $i \in A(x)$. If a particular test indicates that the set of active constraints do not belong to the optimal face, then the active constraints are abandoned and new ones are generated. `GENCAN` follows a similar dynamic. It uses a monotone SPG step to identify a new set of active box constraints and thus a new reduced space. Then, inside the reduced space, it uses a line-search Newton-CG method, also known as the truncated Newton method, to work towards the optimal solution. If $\left\|\widehat{\nabla}^P \mathcal{L}(x_k)\right\| < \eta \left\|\nabla^P \mathcal{L}(x_k)\right\|$

holds, GENCAN abandons the non-optimal face and adds new box constraints to the active set by performing an SPG step in the full space $\mathbb{R}^n$.

Algorithm 7: GENCAN

Input: $\mathcal{L}$, X, $x_0 \in \mathbb{R}^n$, $\lambda \in \mathbb{R}^m$, $\mu \in \mathbb{R}^l$, $\theta \in (0,1)$, $\rho > 0$, $\eta \in (0,1)$, ϵ_{opt}

Output: x' minimizer of $\mathcal{L}$ over X

1 $k = 0$

2 $x_k = P_X(x_k)$

3 **while** $\left\|\nabla^P \mathcal{L}(x_k)\right\|_\infty > \epsilon_{opt}$ **do**

4 **if** $\left\|\widehat{\nabla}^P \mathcal{L}(x_k)\right\| < \eta \left\|\nabla^P \mathcal{L}(x_k)\right\|$ **then**

/* Spectral projected gradient step (in the full space $\mathbb{R}^n$) */

5 $\sigma_k \leftarrow$ spectralStep()

6 $d_k = P_X\left(x_k - \sigma_k \nabla \mathcal{L}(x_k)\right) - x_k$

7 $\alpha_k \leftarrow$ spgLineSearch()

8 $x_{k+1} = x_k + \alpha_k d_k$

9 $k = k + 1$

10 **else**

/* Truncated Newton step (in the reduced space $\mathbb{R}^{\bar{n}}$) */

11 $\widehat{\cdot} \leftarrow$ setMask()

12 Compute: $\widehat{d}_k$, approximate minimizer of $\left\{\frac{1}{2} d^\top \widehat{\nabla}^2 \mathcal{L}(x_k) d + \widehat{\nabla} \mathcal{L}(x_k)^\top d\right\}$

13 $\alpha_k \leftarrow$ gencanLineSearch()

14 $x_{k+1} = x_k + \alpha_k \widehat{d}_k$

15 $k = k + 1$

16 Set: $x' = x_k$

The spectralStep() is computed as in (3.6), the line search procedures spgLineSearch() and gencanLineSearch() are detailed in [13]. The setMask() operator determines if $i \in A(x)$ by checking if either of the following two inequalities is true

$$\begin{aligned} x_i^k &\leq l_i + \epsilon_m^{2/3} \times \max\{1, |l_i|\} \\ x_i^k &\geq u_i - \epsilon_m^{2/3} \times \max\{1, |u_i|\}, \end{aligned}$$

where ϵ_m stands for the machine epsilon, which is an upper bound on the relative error due to rounding in floating point arithmetic. The reduced-space step $x_{k+1} = x_k + \alpha_k \widehat{d}_k$ in Line 14 is achieved by only updating coefficients of x^{k+1} that correspond to free variables, that is, x_i for $i \notin A(x)$, whereas Line 8 happens in the full space. As described in [13], GENCAN uses a modified quadratic conjugate gradient method (QCG) to approximately solve the unconstrained minimization in Line 12. We will use

the following notation in the description of the `QCG` algorithm: $\widehat{H}_k = \widehat{\nabla}^2\mathcal{L}(x_k)$, $\widehat{g}_k = \widehat{\nabla}\mathcal{L}(x_k)$, $\kappa(d) = \frac{1}{2}d^\top \widehat{H}_k d + \widehat{g}_k^\top d$, and

$$D = \{d \in \mathbb{R}^n \mid \|d\| \leq \Delta \text{ and } l_\kappa = l - x_k \leq d \leq u - x_k = u_\kappa\}. \tag{4.39}$$

Algorithm 8: `QCG`

Input: κ, $\widehat{H}_k$, $\widehat{g}_k$, θ, D

Output: d' such that $\langle \widehat{g}_k, d' \rangle \leq -\theta \|\widehat{g}_k\| \|d'\|$

1 $j = 0$, $d_j = 0$, $r_j = \widehat{g}_k$, $p_j = -r_j$
2 $\alpha_j \leftarrow$ `lineSearchInit()`
3 $d_{j+1} = d_j + \alpha_j p_j$
4 $r_{j+1} = r_j + \alpha_j \widehat{H}_k p_j$
5 $j = j + 1$
6 **while** $\langle \widehat{g}_k, d_j \rangle \leq -\theta \|\widehat{g}_k\| \|d_j\|$ **do**
7 $\quad \beta_j = \dfrac{\|r_j\|^2}{\|r_{j-1}\|^2}$
8 $\quad p_j = -r_j + \beta_j p_{j-1}$
9 $\quad$ **if** $\langle p_j, r_j \rangle > 0$ **then**
10 $\quad\quad p_j = -r_j$
11 $\quad \alpha_j \leftarrow$ `lineSearch()`
12 $\quad d_{j+1} = d_j + \alpha_j p_j$
13 $\quad r_{j+1} = r_j + \alpha_j \widehat{H}_k p_j$
14 $\quad j = j + 1$
15 $d' = d_j$

In Line 1 of Algorithm 8 $d_j = 0$ stands for setting all the coefficients of d_j equal zero, r_j is the residual, p_j is the conjugate search direction and d' is the search direction from Algorithm 7. The line search in Line 2 is described immediately below

in Algorithm 9 and the line search from Line 11 in Algorithm 10.

Algorithm 9: `QCG lineSearchInit()`

Input: $\widehat{H}_k$, $\widehat{g}_k$, p_j, d_j, D

Output: α_j

1 $\alpha_{\max} = \max\{\alpha \geq 0 \mid d_j + \alpha p_j \in D\}$

2 **if** $p_j^\top \widehat{H}_k p_j > 0$ **then**

3 $\quad \alpha_j = \min\left\{\alpha_{\max}, \dfrac{\|\widehat{g}_k\|^2}{p_j^\top \widehat{H}_k p_j}\right\}$

4 **else**

5 $\quad \alpha_j = \alpha_{\max}$

Algorithm 10: `QCG lineSearch()`

Input: $\widehat{H}_k$, $\widehat{g}_k$, p_j, d_j, D

Output: α_j

1 $\alpha_{\max} = \max\{\alpha \geq 0 \mid d_j + \alpha p_j \in D\}$

2 **if** $p_j^\top \widehat{H}_k p_j > 0$ **then**

3 $\quad \alpha_j = \min\left\{\alpha_{\max}, \dfrac{\|\widehat{g}_k\|^2}{p_j^\top \widehat{H}_k p_j}\right\}$

4 **else**

5 $\quad$ return $d' = d_j$

4.4 `OPAL`: Object-Parallel Augmented Lagrangian Algorithm

As described in the previous section, the serial augmented Lagrangian implementation `ALGENCAN` can be sectioned into three layers. The algorithm classes in `OPAL` will follow an abstraction approach that is based on these three layers. The first layer is the augmented-Lagrangian outer loop (Algorithm 6), which is implemented in the algorithm class `AL`, the second is the inner loop (Algorithm 7), also known as the subproblem solver, implemented in the `GENCAN` class, and the third layer is the search direction computation of the subproblem solver (Algorithm 8), which is implemented in the `QuadCG` algorithm class. We have chosen `GENCAN` as our subproblem solver because it has very good convergence properties, and the underlying computations do not involve expensive linear algebra operations, such as matrix factorization, which would make the distributed memory parallel implementation difficult. Nonetheless, other

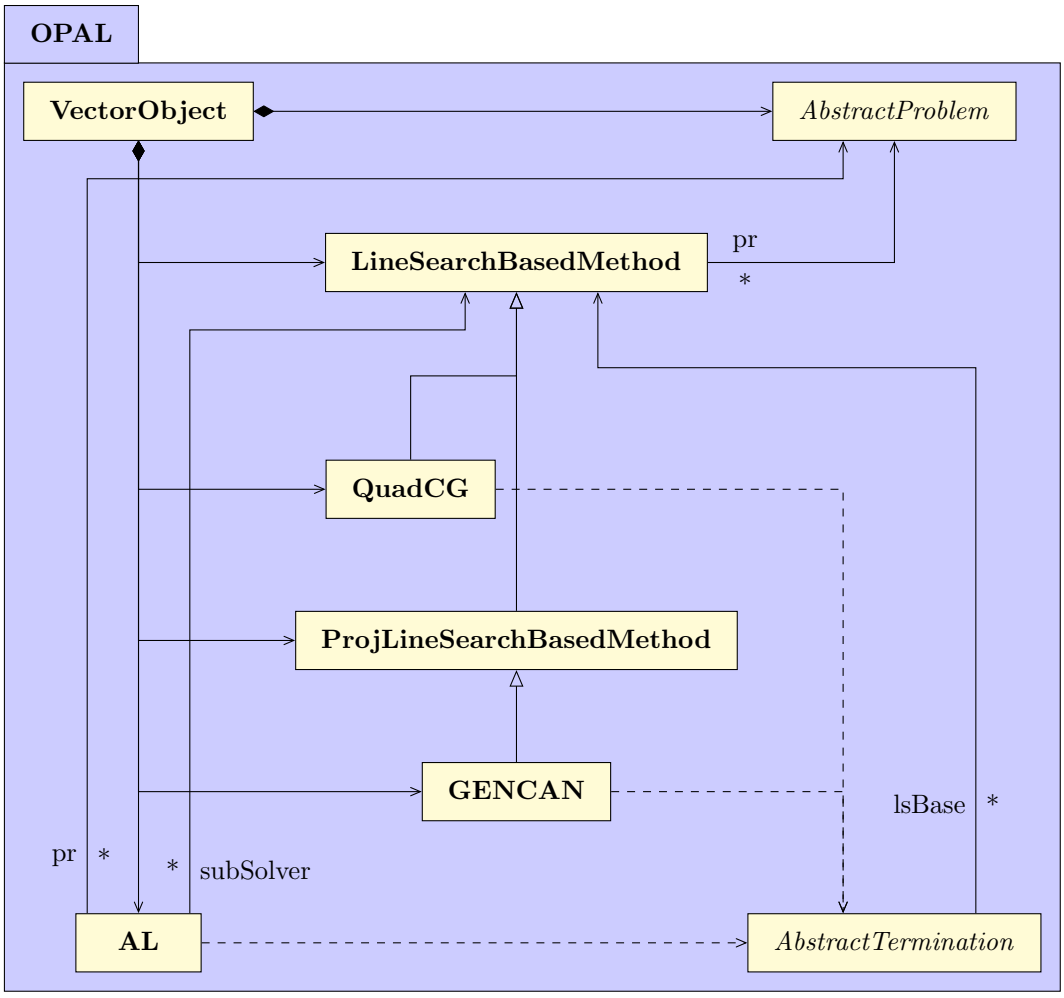


Figure 4.1: Relationships of the `OPAL` package.

subproblem solvers could also be used, for example, `OPSPG` from Section 3.3. `GENCAN` is derived from `ProjLineSearchBasedMethod`, and `QCG` from `LineSearchBasedMethod`. `OPAL`'s algorithm classes `AL`, `GENCAN`, and `QCG`, like all of `OPOS`' optimization algorithms, are implemented using `VectorObject`, `AbstractProblem` and `AbstractTermination` classes. The complete inheritance graph of the `OPAL` software package is illustrated in Figure 4.1.

4.4.1 Algorithm Classes: `AL`, `GENCAN`, `QCG`

The three-layer abstraction scheme allows extensibility and concise representation of the outer loop (`AL`), the subproblem solver (`GENCAN`), and the search direction computation

of the subproblem solver (QCG). OPAL's algorithm class designs need to avoid member routines that are not automatically parallelized by switching the underlying vector and problem class from serial to parallel. The most import routine is the `minimize()` routine of each of the algorithm classes AL, GENCAN, and QCG. After the necessary initializations, the `main()` function in OPAL's driver program calls AL's `minimize()` routine, which returns the solution vector x: `x = opal.minimize(termination)`. AL's `minimize()` routine, depicted in Figure 4.2, implements the augmented Lagrangian outer loop from Algorithm 6. The subproblem solver's `minimize()` routine, which minimizes the augmented Lagrangian function $\mathcal{L}(x)$, is called at every iteration of the outer loop. GENCAN's `minimize()` routine is depicted in Figures 4.3 to 4.5. The third layer of OPAL is the QCG algorithm, which is called during the reduced-space section of GENCAN to compute the search direction `d = quadSolver->minimize(cgTermination)`. Its initialization is illustrated in Figure 4.6, and its main loop in Figure 4.7. Note that the notation used in the source code of QCG differs from the notation used in Algorithm 8. In the source code, `x` refers to GENCAN's search direction d, `d` refers to QCG's search direction p, and `g` refers to the residual r. This change in notation is motivated by the modular approach of OPOS, since QCG could also be used as a stand alone unconstrained optimization algorithm, in which case it makes more sense to follow the conventional notation, in which x encapsulates the sequence of iterates, d the search direction and g the gradient.

4.4.2 Reduced Space Quadratic Approximation Problem Class

Given a smooth box constrained optimization problem ($\min f(x)$ s. t. $x \in X$), GENCAN computes the search direction $\widehat{d}_k$ of its reduced-space minimization phase by approximately minimizing the quadratic function $\kappa(d) = \frac{1}{2} d^\top \widehat{H}_k d + \widehat{g}_k^\top d$, where $\widehat{g}_k$ and $\widehat{H}_k$ are the gradient and the Hessian of f at x_k. This problem yields an approximation of the basic Newton step $d_k^{\mathrm{N}} = -\left(\nabla^2 f(x_k)\right)^{-1} \nabla f(x_k)$. However, κ is problem dependent because the value of $\widehat{H}_k$ and $\widehat{g}_k$ depends on the problem: $\min f(x)$ s. t. $x \in X$. This means that the quadratic function κ has to be generated automatically within GENCAN, since otherwise we would be required to implement it explicitly for every possible

```
VectorObject& OPAL::minimize(AbstractTermination&
   ↪ alTermination) {
  x = subSolver->minimize(spTermination);
  while (alTermination.check() && iter < maxAlIter) {
    iter++;
    if(numInequ > 0 && numEqu > 0) {
      pr->equConVal(x, hx);
      hxNormInf = hx.normInf();
      lambda = lambda + (*penaltyParam) * hx;
      pr->inequConVal(x, gx);
      gxNormInf = gx.normInf();
      gxMaxElement = gx.maxElement();
      mu = mu + (*penaltyParam) * gx;
      mu.localMax(zeroVec);
    }
    else if(numInequ > 0) {
      pr->inequConVal(x, gx);
      gxNormInf = gx.normInf();
      gxMaxElement = gx.maxElement();
      mu = mu + (*penaltyParam) * gx;
      mu.localMax(zeroVec);
    }
    else if(numEqu > 0) {
      pr->equConVal(x, hx);
      hxNormInf = hx.normInf();
      lambda = lambda + (*penaltyParam) * hx;
    }
    *penaltyParam = updatePenalty();
    x = subSolver->minimize(spTermination);
  }
return x;
}
```

Figure 4.2: `AL`'s `minimize()` routine.

```
VectorObject& GENCAN::minimize(AbstractTermination&
   ↪ termination) {
  iter = maxCount = 0;
  pr->localProjectOnBounds(x);
  quadPr->setMask(x);
  xNormInf = x.normInf();
  xNorm2 = x.norm2();
  quadPr->setTrustRad(std::max(100.0, 100.0 * xNormInf));
  double objValStart = objVal = objValk = pr->objValGrad(x,
     ↪ g);
  objValBest = std::numeric_limits<double>::max();
  xg = x - g;
  pr->localProjectOnBounds(xg);
  pg = xg - x;
  pgNorm2 = pgkNorm2 = pg.norm2();
  pgNormInf = pgkNormInf = pg.normInf();
  pgNorm2Best = std::numeric_limits<double>::max();
  pgNormInfBest = std::numeric_limits<double>::max();
  cgTermination->setSlopeIntercept();
  ig.elementProd(pg, mask);
  igNorm2 = ig.norm2();
  igNormInf = ig.normInf();
  if(pgNormInf != 0.0) {
    auxStep = pr->getRoot12MachineEps() * std::max(1.0,
       ↪ xNormInf/pgNormInf);
  }
  else {
    auxStep = 0.0;
  }
  xk = x + auxStep * pg;
  pr->objGrad(xk, gk);
  gd = gk - g;
  xd = xk - x;
  xdNormInf = xd.normInf();
  xdNorm2sq = xd.norm2sq();
  xdNorm2 = std::sqrt(xdNorm2sq);
  while(termination.check() && iter < maxIter) {
    iter++;
    if(leaveFace()) {
      /* Full space step here */
    }
    else {
      /* Reduced space step here */
    }
  }
  return x;
};
```

Figure 4.3: `GENCAN`'s `minimize()` routine.

```
objVal = objValk;
pgNormInf = pgkNormInf;
pgNorm2 = pgkNorm2;
spectralStep = spectralSearch();
xg = x - spectralStep * g;
pr->localProjectOnBounds(xg);
d = xg - x;
stepSize = spgLineSearch();
getxkgk(stepSize, xk, gk);
objValk = pr->objVal(xk);
xkNormInf = xk.normInf();
xkNorm2 = xk.norm2();
gd = gk - g;
xd = xk - x;
xdNormInf = xd.normInf();
xdNorm2sq = xd.norm2sq();
xdNorm2 = std::sqrt(xdNorm2sq);
quadPr->setTrustRad(std::max(trustRadMin, 10.0 * xdNormInf)
   ↪ );
x = xk;
g = gk;
xNormInf = xkNormInf;
xg = x - g;
pr->localProjectOnBounds(xg);
pg = xg - x;
pgkNormInf = pg.normInf();
pgkNorm2 = pg.norm2();
quadPr->setMask(x);
ig.elementProd(pg, mask);
igNorm2 = ig.norm2();
igNormInf = ig.normInf();
if(objVal < objValBest)
  objValBest = objVal;
if(pgNormInf < pgNormInfBest)
  pgNormInfBest = pgNormInf;
if(pgNorm2 < pgNorm2Best)
  pgNorm2Best = pgNorm2;
reset();
```

Figure 4.4: `GENCAN`'s full space portion of `minimize()` routine.

```
subTermination->setSubspaceNoTerm(true);
objVal = objValk;
pgNormInf = pgkNormInf;
pgNorm2 = pgkNorm2;
quadPr->setQuadApproxPoint(x,g,xNormInf);
cgTermination->setParameters();
d = quadSolver->minimize(cgTermination);
stepSize = subspaceLineSearch();
getxkgk(stepSize, xk, gk);
if(stepSize >= stepSizeMax) {
 pr->localProjectOnBounds(xk);
}
objValk = pr->objVal(xk);
xkNormInf = xk.normInf();
xkNorm2 = xk.norm2();
gd = gk - g;
xd = xk - x;
xdNormInf = xd.normInf();
xdNorm2sq = xd.norm2sq();
xdNorm2 = std::sqrt(xdNorm2sq);
quadPr->setTrustRad(std::max(trustRadMin, 10.0 * xdNormInf)
   ↪ );
x = xk;
g = gk;
xNormInf = xkNormInf;
xg = x - g;
pr->localProjectOnBounds(xg);
pg = xg - x;
pgkNormInf = pg.normInf();
pgkNorm2 = pg.norm2();
quadPr->setMask(x);
ig.elementProd(pg, mask);
igNorm2 = ig.norm2();
igNormInf = ig.normInf();
if(objVal < objValBest)
  objValBest = objVal;
if(pgNormInf < pgNormInfBest)
  pgNormInfBest = pgNormInf;
if(pgNorm2 < pgNorm2Best)
  pgNorm2Best = pgNorm2;
reset();
```

Figure 4.5: `GENCAN`'s reduced space portion of `minimize()` routine.

```
iter = 0;
maxIter = (int) mask.norm2sq();
x.fill(0.0);
double objValStart = objVal = objValk = 0.0;
g = pr->getQuadApproxFirstOrder();
gNorm2sq = g.norm2sq();
d = -1.0*g;
gtd = -1.0*gNorm2sq;
pr->applyObjHessian(x, d, Hd);
dtHd = d.inner(Hd);
stepSizeMax = pr->trustMaxStepSize(x, d);
if(dtHd > 0.0) {
  stepSize = std::min(stepSizeMax, gNorm2sq/dtHd);
}
else {
  stepSize = stepSizeMax;
}
x = x + stepSize*d;
objValk = objVal + 0.5*stepSize*stepSize*dtHd + stepSize*
   ↪ gtd;
g = g + stepSize*Hd;
gkNorm2sq = g.norm2sq();
iter++;
```

Figure 4.6: Initialization of `QCG`'s `minimize()` routine.

```
VectorObject& QuadSubspaceCG::minimize(AbstractTermination&
   ↪  termination) {

/* Initialize here */

  while( termination.check() && iter < maxIter) {
    iter++;
    objVal = objValk;
    beta = gkNorm2sq/gNorm2sq;
    d = -1.0*g + beta * d;
    if(d.inner(g) > 0) {
      d = -1.0*g;
      gtd = -1.0*gkNorm2sq;
    }
    else {
      gtd = -1.0*gkNorm2sq + beta*(gtd + stepSize*dtHd);
    }
    stepSizeMax = pr->trustMaxStepSize(x, d);
    pr->applyObjHessian(x, d, Hd);
    dtHd = d.inner(Hd);
    if(dtHd > 0.0) {
      stepSize = std::min(stepSizeMax, gkNorm2sq/dtHd);
    }
    else {
      return x;
    }
    x = x + stepSize * d;
    objValk = objVal + 0.5*stepSize*stepSize*dtHd +
       ↪ stepSize*gtd;
    g = g + stepSize*Hd;
    gNorm2sq = gkNorm2sq;
    gkNorm2sq = g.norm2sq();
    reset();
  }
  return x;
}
```

Figure 4.7: `QCG minimize()` routine's main loop.

problem class. This automatic generation of the κ function is achieved through the `ReducedQuadAppproxProblem` class. `ReducedQuadAppproxProblem` is an `AbstractProblem`-derived class, and its constructor takes a pointer to an `AbstractProblem` and a `bool`, which determines whether to compute κ using the full Hessian or just an approximation of it. We will refer to the `AbstractProblem`-derived object that is used to generate `ReducedQuadAppproxProblem` as the *main problem*. `ReducedQuadAppproxProblem` has members `xk` for x_k, `mfirstOrderTerm` for $\widehat{g}_k$, and a routine `applyObjHessian()` to compute Hessian-vector multiplies $\widehat{H}_k d$. It also holds a `mask` vector, which is responsible for determining the reduced space, meaning that coefficients that correspond to indices i, such that $i \in A(x)$ are 0, otherwise 1. Performing a componentwise multiplication of a vector with `mask` is equivalent to projecting it onto the set S_I, which we defined in Section 4.3.1. Given a point x_k, `ReducedQuadAppproxProblem` computes $\kappa(d) = \frac{1}{2} d^\top \widehat{H}_k d + \widehat{g}_k^\top d$ and its gradient $\nabla\kappa(d) = \widehat{H}_k d + \widehat{g}_k$ based on information from the main problem. For example, computing the function value of κ in the reduced space is done by calling `ReducedQuadAppproxProblem`'s `objVal()` method, which executes the following steps:

1. Project x onto S_I.
2. If the user has specified to use the approximate Hessian, then $\widehat{H}_k d$ is calculated by a difference quotient formula, using main problem class routines and then projected onto S_I.
2. If we would like to compute κ with the full Hessian then the main problem class' `applyObjHessian()` routine is called to compute $\widehat{H}_k d$, which is then projected onto S_I.
3. Once $\widehat{H}_k d$ is obtained the remaining quantities needed for $\kappa(d)$ such as $\widehat{g}_k$ are available from `ReducedQuadAppproxProblem`, then $\kappa(d)$ can be computed.

Figure 4.8 illustrates the source code of `ReducedQuadAppproxProblem`'s `objVal()` method.

```
double QuadApproxProblem::objVal(VectorObject& x) {
  numObjEval++;
  mx.elementProd(x, mask);
  if(approxHessian) {
    double mxNormInf = mx.normInf();
    if(mxNormInf == 0.0) {
      Hd = mx;
      return 0.0;
    }
    double tau = root12MachineEps;
    tau = root12MachineEps * std::max(xkNormInf/mxNormInf,
       ↪ 1.0);
    xkp = xk + tau * x;
    pr->objGrad(xkp, mgradxkp);
    mgradxkp.localElementProd(mask);
    Hd = (1.0/tau) * (mgradxkp - mfirstOrderTerm);
  }
  else {
    pr->applyObjHessian(xk, mx, Hd);
    Hd.localElementProd(mask);
  }
  gradLinComb = 0.5 * Hd + mfirstOrderTerm;
  return(mx.inner(gradLinComb));
}
```

Figure 4.8: `ReducedQuadAppproxProblem`'s `objVal()` routine.

4.4.3 OPAL as a Serial General-Purpose Nonlinear Solver

With the help of the AMPL Solver Library [28] (ASL) it is possible to build a serial problem class, which we will call AugLagAslProblem, that is capable of handling general nonlinear objective functions, constraints and their gradients. AugLagAslProblem is a derived class of AslProblem, which uses ASL and takes an nl file as input, and inherits most of its routines, except for the objective function and gradient computations, since those need to be modified to accommodate the subproblem solver. For this purpose we use routines from AslProblem to create methods that evaluate the augmented Lagrangian function

$$\mathcal{L}_\rho(x,\lambda,\mu) = \left\{ f(x) + \langle \lambda, h(x)\rangle + \frac{\rho}{2}\|h(x)\|^2 + \left\|[\mu + \rho g(x)]_+\right\|^2 \right\}$$

and its gradient. The source code for the objective function value computation is illustrated in Figure 4.9 with the following steps:

1. AslProblem's objVal() routine returns $f(x)$, which then is multiplied by objSign which takes the value 1 or -1 depending on whether we minimize or maximize the objective function, the result is saved in value.
2. If the problem has equality constraints we compute $\langle \lambda, h(x)\rangle + \frac{\rho}{2}\|h(x)\|^2$ using AslProblem's equConVal() method and add it to value.
3. If inequality constraints are present we compute $\left\|[\mu + \rho g(x)]_+\right\|^2$ using AslProblem's inequConVal() method and add it to value.
4. Finally, returning value, which is the function value of $\mathcal{L}_\rho(x,\lambda,\mu)$.

4.5 Solving Large-Scale Linear Stochastic Programming Problems

A variety of planning, logistics, and system control problems may be formulated as stochastic programming problems [63, 11, 58], resulting in very large models. Here, we will present an approach to solving the special case of this problem class, in which the objective function and the constraints are linear. Every such linear stochastic programming problem can be formulated as follows

```
double AugLagAslProblem::objVal(VectorObject &x) {
  numObjEval++;
  double value = objSign * AslProblem::objVal(x);
  if(numEquCon > 0) {
    equConVal(x, hx);
    value += lambda.inner(hx) + 0.5 * penaltyParam * hx.
       ↪ norm2sq();
  }
  if(numInequCon > 0) {
    inequConVal(x, gx);
    gx = mu + penaltyParam * gx;
    gx.localMax(zeroVec);
    value += (1.0/penaltyParam) * 0.5 * gx.norm2sq();
  }
  return value;
}
```

Figure 4.9: `AugLagAslProblem` class' `objVal()` routine.

$$
\begin{array}{lllllllll}
\min & (c_1)^\top x_1 & + & (c_2)^\top x_2 & + & \dots & + & (c_T)^\top x_T & \\
\text{s.t.} & A_1 x_1 & & & & & & & = b_1 \\
& B_2 x_1 & + & A_2 x_2 & & & & & = b_2 \\
& & & & \ddots & & & & \\
& & & & & B_T x_{T-1} & + & A_T x_T & = b_T \\
& x_1 \geq 0, & & x_2 \geq 0, & & \dots & & x_T \geq 0, &
\end{array}
\tag{4.40}
$$

where the uncertain data are $\xi_t := (c_t, B_t, A_t, b_t)$ and $\xi_1, \dots, \xi_T$ are revealed incrementally over T periods, T is the total number of stages, and x_t are decision vectors corresponding to time periods (stages) [63]. A constraint corresponds to stage t, if it does not contain any nonzero coefficients for variables $x_{\bar{t}}$ such that $\bar{t} > t$. For example, the constraint $B_2 x_1 + A_2 x_2 = b_2$ is a stage 2 constraint. We will refer to B_ts as *technology matrices* [11, p. 104]. A technology matrix B_t determines the influence of the decisions at stage $t-1$ on stage t. More precisely, technology matrices are the nonzero coefficients of stage $t-1$ variables in a stage t constraint.

One way to solve a stochastic programming problem with conventional optimization algorithms, such as augmented Lagrangian methods, is to solve the deterministic equivalent, also known as the extensive form. The extensive form of a linear stochastic programming problem is, essentially, a large linear programming problem, where the decision vectors x^ν are associated with specific realizations of ξ_t at each stage t. The following example will illustrate how to obtain from a general linear stochastic program its extensive form. Consider the three-stage stochastic program below:

$$\begin{array}{lllllll} \min & (c_1)^\top x_1 & + & (c_2)^\top x_2 & + & (c_3)^\top x_3 & \\ \text{s.t.} & A_1 x_1 & & & & & = b_1 \\ & B_2 x_1 & + & A_2 x_2 & & & = b_2 \\ & & & B_3 x_2 & + & A_3 x_3 & = b_3 \\ & x_1 \geq 0 & & x_2 \geq 0 & & x_3 \geq 0 & \end{array} \tag{4.41}$$

Let us assume that ξ_1 is known and, therefore, has only one realization, ξ_2 has two realizations each with respective probabilities π_1 and π_2 such that $\pi_1 + \pi_2 = 1$, and ξ_3 has four realizations with probabilities $\pi_1 + \pi_2 + \pi_3 + \pi_4 = 1$. This setup is illustrated in Figure 4.10 with the help of a scenario tree, where each node stands for the different realizations at a given stage. Thus, the root stands for the first stage (ξ_1) and the four leaves for the last stage (ξ_3). Consequently, each node belongs to a unique stage, whereas multiple nodes could be part of a given stage. We will call each root to leaf path a scenario S_j, where $j = 0, \ldots, r-1$ and r is the total number of scenarios. The total number of scenarios equals the total number of realizations at the last stage. Given the scenario tree formulation, our decision vectors x^ν correspond to nodes ν of the scenario tree. If we label each node of the tree in Figure 4.10 as $1, \ldots, 7$, then the extensive form of (4.41) becomes

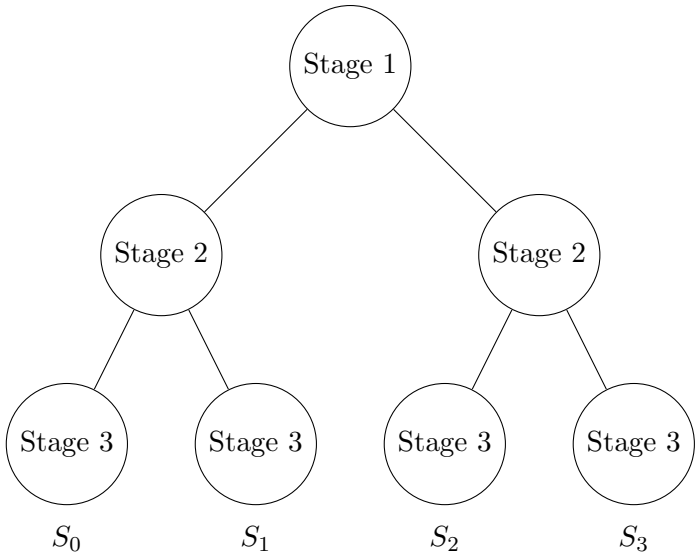


Figure 4.10: 3-stage scenario tree with a total of 4 scenarios $\{S_0, S_1, S_2, S_3\}$

$$
\begin{array}{lllllllll}
\min & (c^1)^\top x^1 + (c^2)^\top x^2 + (c^4)^\top x^4 + (c^5)^\top x^5 + (c^3)^\top x^3 + (c^6)^\top x^6 + (c^7)^\top x^7 & & & & & & & \\
\text{s.t.} & A^1 x^1 & & & & & & & = b^1 \\
& B^2 x^1 + & A^2 x^2 & & & & & & = b^2 \\
& & B^4 x^2 + & A^4 x^4 & & & & & = b^4 \\
& & B^5 x^2 + & & A^5 x^5 & & & & = b^5 \\
& B^3 x^1 + & & & & A^3 x^3 & & & = b^3 \\
& & & & & B^6 x^3 + & A^6 x^6 & & = b^6 \\
& & & & & B^7 x^3 + & & A^7 x^7 & = b^7 \\
& x^1 \geq 0 & x^2 \geq 0 & x^4 \geq 0 & x^5 \geq 0 & x^3 \geq 0 & x^6 \geq 0 & x^7 \geq 0, &
\end{array}
\tag{4.42}
$$

where the c^ν are weighted with the respective node probabilities, and the labeled scenario tree is shown in Figure 4.11. Similarly, all linear stochastic programs can be formulated as simple linear programs of the following form

$$
\begin{array}{ll}
\min & \langle c, x \rangle \\
\text{s.t.} & Mx - b = 0 \\
& x \in \mathbb{R}^n_+,
\end{array}
\tag{4.43}
$$

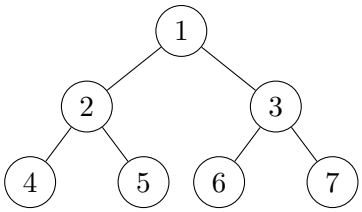


Figure 4.11: Labeled scenario tree of (4.41)

which is a smooth convex optimization problem. Therefore, we can apply the augmented Lagrangian algorithm. Given (4.43), the augmented Lagrangian function, its gradient, and its Hessian are

$$\begin{aligned} \mathcal{L}_\rho(x,\lambda) &= \langle c, x\rangle + \langle \lambda, Mx - b\rangle + \tfrac{\rho}{2}\|Mx-b\|^2 \\ \nabla_x \mathcal{L}_\rho(x,\lambda) &= c^\top + M^\top\lambda + \rho(M^\top(Mx-b)) \\ \nabla_{xx}\mathcal{L}_\rho(x,\lambda) &= \rho M^\top M. \end{aligned} \tag{4.44}$$

Thus, our problem class implementing the objective function, gradient and Hessian computations needs to be able to perform matrix-vector multiplies Mx and vector-transpose-matrix multiplies $\lambda^\top M$ efficiently (inner products $\langle c, x\rangle$, vector additions, and scalar multiplications are operations implemented in the vector class). Because `GENCAN` only requires Hessian-vector products $\nabla_{xx}\mathcal{L}_\rho(x,\lambda)d$ we do not need to perform matrix-matrix multiplies $M^\top M$ explicitly, since Hessian-vector products can be computed with a matrix-vector and a matrix-transpose-vector multiply $M^\top(Md)$ or equivalently a vector-transpose-matrix multiply $(Md)^\top M$.

4.5.1 Solving Linear Stochastic Programs in Serial

In serial, as mentioned in Section 4.4.3, we can use the `AugLagAslProblem` class to solve any smooth convex optimization problem. Since `AugLagAslProblem` is an `ASL` based problem class its input is an `nl` file. The most straightforward way of generating an `nl` file describing a stochastic program's extensive form (linear program) is to use an AML interpreter with stochastic programming functionality. A popular open-source

`Python`-based stochastic programming AML is `PySP` [79] an extension of `Pyomo` [38], which we have used to generate the extensive form of our stochastic programs.

4.5.2 Solving Linear Stochastic Programs in Parallel

As mentioned in Section 1.2, there are no software libraries that can perform in a distributed-memory parallel setting what `ASL` can in serial. Therefore, building a custom linear stochastic programming problem class is inevitable in the parallel case. Just like for LASSO, the parallel problem class determines the data partitioning and the communication primitives of problem-dependent methods such as objective, gradient and Hessian evaluations. Unlike for LASSO, however, a simple column, row or nonzero partitioning of $M \in \mathbb{R}^{n \times m}$ would not be efficient, because the extensive form of stochastic programs usually has very large numbers of both rows and columns, which means that communicating an entire row or column vector would be too expensive. Instead of a column-based or nonzero-based partition we use a scenario-based partitioning scheme, which was originally developed by Watson *et al.* for a progressive-hedging-based solver `ph` that is integrated with `PySP` [79].

Scenario-Based Partitioning

Scenario-based partitioning means that we create a separate problem instance for each scenario in the scenario tree and generate an `nl` file corresponding to that problem instance. Consider S_0 with nodes $\{1, 2, 4\}$ (Figure 4.12) of our 3 stage 4 scenario problem from (4.42). As depicted below, we extract the data and decision variables corresponding to nodes in S_0 and use it to formulate the problem (4.45).

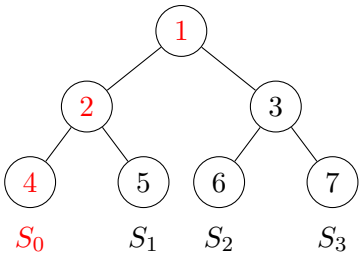


Figure 4.12: Scenario S_0 $\{1, 2, 4\}$

$$
\begin{array}{lllllllll}
\min & (c^1)^\top x^1 + (c^2)^\top x^2 + (c^4)^\top x^4 + (c^5)^\top x^5 + (c^3)^\top x^3 + (c^6)^\top x^6 + (c^7)^\top x^7 & & & & & & & \\
\text{s.t.} & A^1 x^1 & & & & & & & = b^1 \\
& B^2 x^1 + & A^2 x^2 & & & & & & = b^2 \\
& & B^4 x^2 + & A^4 x^4 & & & & & = b^4 \\
& & B^5 x^2 + & & A^5 x^5 & & & & = b^5 \\
& B^3 x^1 + & & & & A^3 x^3 & & & = b^3 \\
& & & & & B^6 x^3 + & A^6 x^6 & & = b^6 \\
& & & & & B^7 x^3 + & & A^7 x^7 & = b^7 \\
& x^1 \geq 0 & x^2 \geq 0 & x^4 \geq 0 & x^5 \geq 0 & x^3 \geq 0 & x^6 \geq 0 & x^7 \geq 0 &
\end{array}
$$

$$
\begin{array}{lllll}
\min & (c^1)^\top x^1 + (c^2)^\top x^2 + (c^4)^\top x^4 & & & \\
\text{s.t.} & A^1 x^1 & & & = b^1 \\
& B^2 x^1 + & A^2 x^2 & & = b^2 \\
& & B^4 x^2 + & A^4 x^4 & = b^4 \\
& x^1 \geq 0 & x^2 \geq 0 & x^4 \geq 0 &
\end{array}
\tag{4.45}
$$

We do this for each scenario $S_0, \ldots, S_{r-1}$, giving us r separate problems. For distributed-memory parallel solvers that employ a decomposition-based approach, like progressive hedging, these problems are treated and solved as separate instances at certain points in the decomposition procedure. Iteratively using the solutions to perturbed versions of these smaller problems, decomposition algorithms, with the help of special coordination steps, construct a solution that is optimal for the extensive form of the problem. However, our approach is different: we do not use a decomposition algorithm, but simply use the "decomposition" approach to partition our data in such a way that

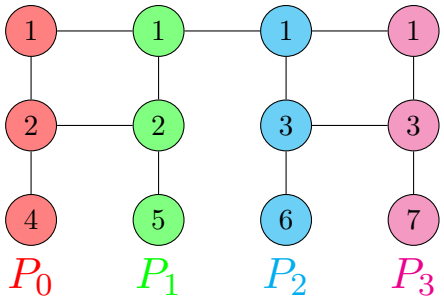


Figure 4.13: Node data replication for (4.42) on 4 processors.

yields efficient linear algebra computations in distributed memory parallel settings.

Each scenario's data is assigned to one processor, for example, scenario S_0 is assigned to processor p_0, scenario S_1 to processor p_1, and so forth. Theoretically, it is also possible to assign bundles of scenarios to individual processors, but for the sake of simplicity we only discuss the case where each processor receives one scenario's data. Thus, from now on we will assume that the number of processors always equals the number of scenarios. Because some tree nodes occur in multiple scenarios, for example, the root node (1) is part of every scenario, our scenario partitioning method also means that to some degree we replicate node data across processors. The replication pattern for (4.42) is shown in Figure 4.13: node 1 data (yellow) is replicated in every processor, node 2 data (violet) in processors p_0 and p_1, and node 3 data (orange) in processors p_2 and p_3. Note that leaf node information is not replicated. The resulting constraint data distribution of (4.42) across 4 processors is shown in Figure 4.14. Note, that this data partitioning procedure assigns an even work load to each processor, which is important for parallel efficiency.

Because our distributed M replicates non-leaf node data, we also have to replicate the corresponding coefficients of x. As shown in Figure 4.15 x^1 (yellow) that corresponds to node 1 is replicated in every processor, x^2 (violet) that corresponds to node 2 is replicated in processors p_0 and p_1, and x^3 (orange) that corresponds to node 3 is replicated in processors p_2 and p_3. In terms of the dual variable λ, we adopt a similar strategy, coefficients of λ that correspond to replicated constraints are also replicated as shown in Figure 4.16.

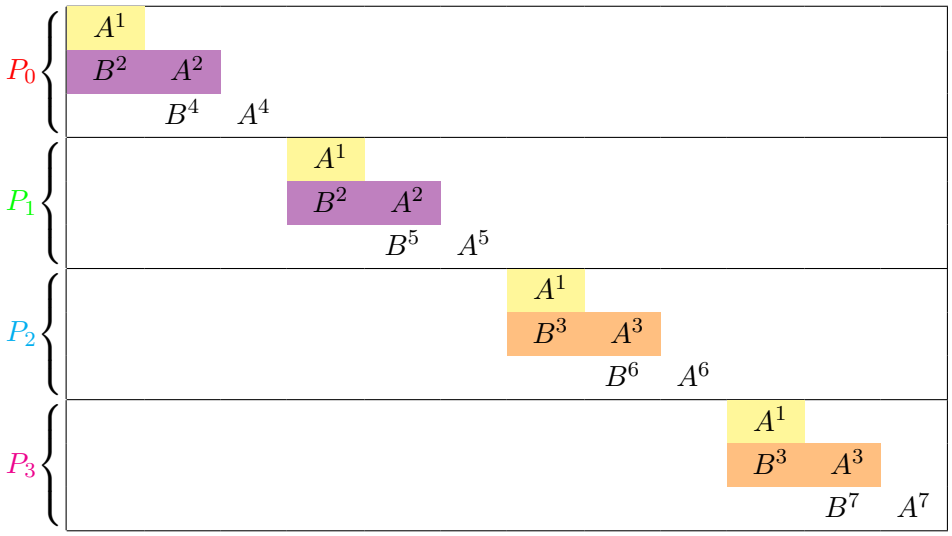


Figure 4.14: Data distribution of the constraint matrix M from (4.42) across 4 processors.

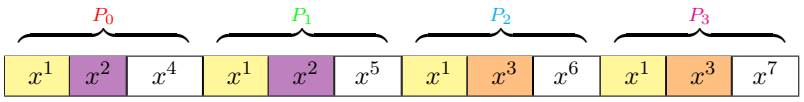


Figure 4.15: Partially replicate non-leaf node variables of x.

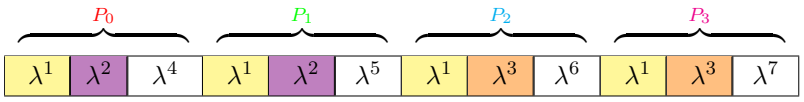


Figure 4.16: Partially replicate non-leaf node variables of λ.

Given a general scenario tree and an arbitrary non-leaf node ν, the following procedure is used to determine which processors replicate ν's data:

1. Label each leaf node of the scenario tree with a processor label (given that the number of processors equals the number of scenarios, each leaf node will receive a unique label).
2. Determine ν's leaf-node descendants.
3. ν's data is replicated in the processors that label its leaf-node descendants.

For `OPAL` we perform the labeling in a "left-to-right" manner. This means that we label the leftmost leaf node with processor p_0, the leftmost leaf node's neighbor with processor p_1 and so forth. We will refer to this procedure as *left-to-right labeling*.

The `DistributedLinStochProgAugLagAslProblem` Class

Our `DistributedLinStochProgAugLagAslProblem` class implements all methods that are required by `OPAL` for distributed linear stochastic programs with scenario-based partitioning. Each processor's `DistributedLinStochProgAugLagAslProblem` object holds one scenario's data, which is in accordance with the data distribution scheme in the previous section and the SPMD paradigm of `OPOS`. The constructor for `DistributedLinStochProgAugLagAslProblem` is responsible for reading in the scenario data from the appropriate `nl` file. In order that our distributed linear algebra operations work properly, it is important that neighboring scenarios' data be assigned to neighboring processors in a left-to-right manner, i.e. scenario S_0 is assigned to processor p_0, scenario S_1 to processor p_1 and so forth. This means that, given (4.42), assigning S_0 to p_0 and S_1 to p_2 would be an invalid assignment. We have developed a `PySP` scenario data generation plugin that ensures the proper ordering of scenarios and labels the `nl` files accordingly. `OPAL`'s driver makes sure that each processor reads in the appropriate scenario data based on the `nl` file labeling. In addition to the scenario ordering, our `PySP` plugin also has to ensure that the replicated decision vectors and constraints have the same position in the scenario problem formulation in each processor. To elaborate, consider S_0 and S_1 and the linear programs composed of their data:

S_0 problem

$$
\begin{array}{lllll}
\min & (c^1)^\top x^1 + (c^2)^\top x^2 + (c^4)^\top x^4 & & & \\
\text{s.t.} & A^1 x^1 & & & = b^1 \\
& B^2 x^1 + & A^2 x^2 & & = b^2 \\
& & B^4 x^2 + & A^4 x^4 & = b^4 \\
& x^1 \geq 0 & x^2 \geq 0 & x^4 \geq 0 &
\end{array}
$$

S_1 problem

$$
\begin{array}{lllll}
\min & (c^1)^\top x^1 + (c^2)^\top x^2 + (c^5)^\top x^5 & & & \\
\text{s.t.} & A^1 x^1 & & & = b^1 \\
& B^2 x^1 + & A^2 x^2 & & = b^2 \\
& & B^5 x^2 + & A^5 x^5 & = b^5 \\
& x^1 \geq 0 & x^2 \geq 0 & x^5 \geq 0. &
\end{array}
$$

Notice that these two problems have identical data except for the decision vectors x^4, x^5 and the node data (A^4, B^4, b^4, c^4) and (A^5, B^5, b^5, c^5). In order for our linear algebra operations to work properly, our `PySP` plugin has to ensure that decision vectors and constraints that are identical for different scenarios have the same ordering in their respective `nl` files, meaning the first set of constraints in both S_0's and S_1's `nl` file has to be $A^1 x^1 = b^1$ followed by $B^2 x^1 + A^2 x^2 = b^2$, and the decision variables have to occupy the same positions i.e. $\left[x^1, x^2, \ldots\right]$. The `nl` file generator in `PySP` does not enforce any particular ordering, and the position of constraints and variables in an `nl` file depends on various factors, such as the specific problem instance and current computer architecture. Therefore, `DistributedLinStochProgAugLagAslProblem` class's constructor has to first sort the scenario data to make sure that replicated data are in the same order in each processor. To be able to sort the data, we had to assign identifications to constraints and variables, which is done in our `PySP` plugin through `ASL` suffixes that can convey auxiliary information on variables, constraints, objectives and problems [28, 29].

Given the scenario based data partitioning, we next describe the design of our distributed linear algebra operations Mx and $\lambda^\top M$, using (4.42) as an example. As mentioned earlier our constraint matrix M is distributed as shown in Figure 4.14.

Computing the Matrix-Vector Product Mx

Let us consider the operation $Mx = z$ as shown in Figure 4.17. Every processor has the necessary data that is required to compute its portion of the product Mx. After

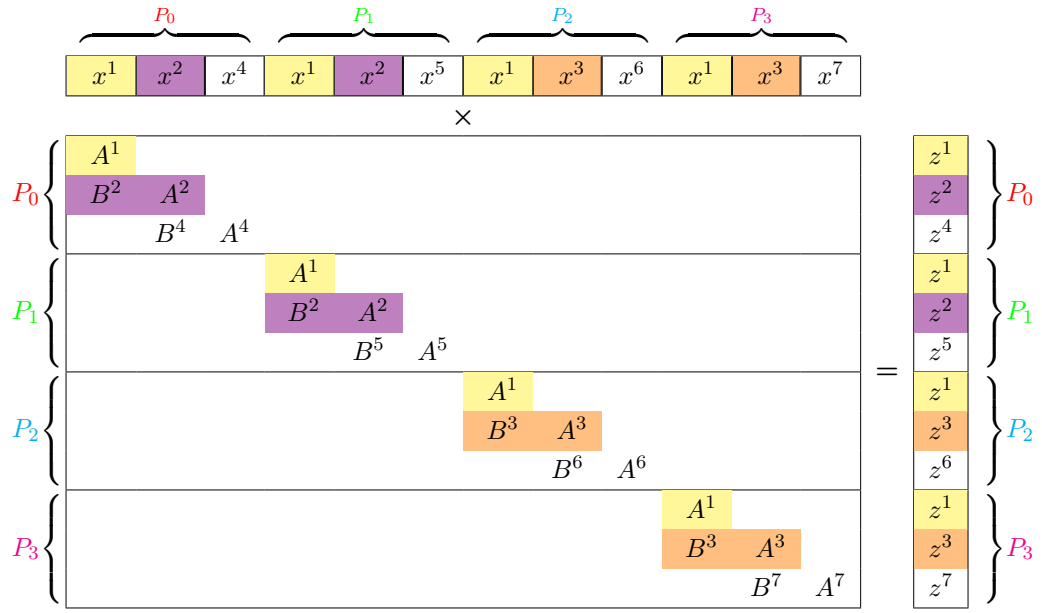


Figure 4.17: Distributed matrix-vector multiply $Mx = z$.

each processor computes its local matrix-vector multiplies the result vector z has the same values as it would have in the serial case, also, every processor's z^1 is identical, p_0 and p_1 have identical z^2-s etc., just as the replication pattern suggest. Hence, the Mx operation is communication free.

Computing the Vector-Transpose-Matrix Product $\lambda^\top M$

Since the total number of scenarios r equals the total number of processors P, and each scenario S_i is assigned to processor p_i, in this section, the index i will run from 0 to $P-1$. Let us consider the operation $\lambda^\top M = y$ as shown in Figure 4.18. We will refer to any row, or combination of rows, of a technology matrix B^ν as *technology matrix data*, and to $\Lambda^\nu = (\lambda^\nu)^\top B^\nu$ as the *technology matrix product*. After computing the local vector-transpose-matrix multiplies, y^1 has the following values:

In processor p_0: $y^1 = (\lambda^1)^\top A^1 + (\lambda^2)^\top B^2$.

In processor p_1: $y^1 = (\lambda^1)^\top A^1 + (\lambda^2)^\top B^2$.

In processor p_2: $y^1 = (\lambda^1)^\top A^1 + (\lambda^3)^\top B^3$.

In processor p_3: $y^1 = (\lambda^1)^\top A^1 + (\lambda^3)^\top B^3$.

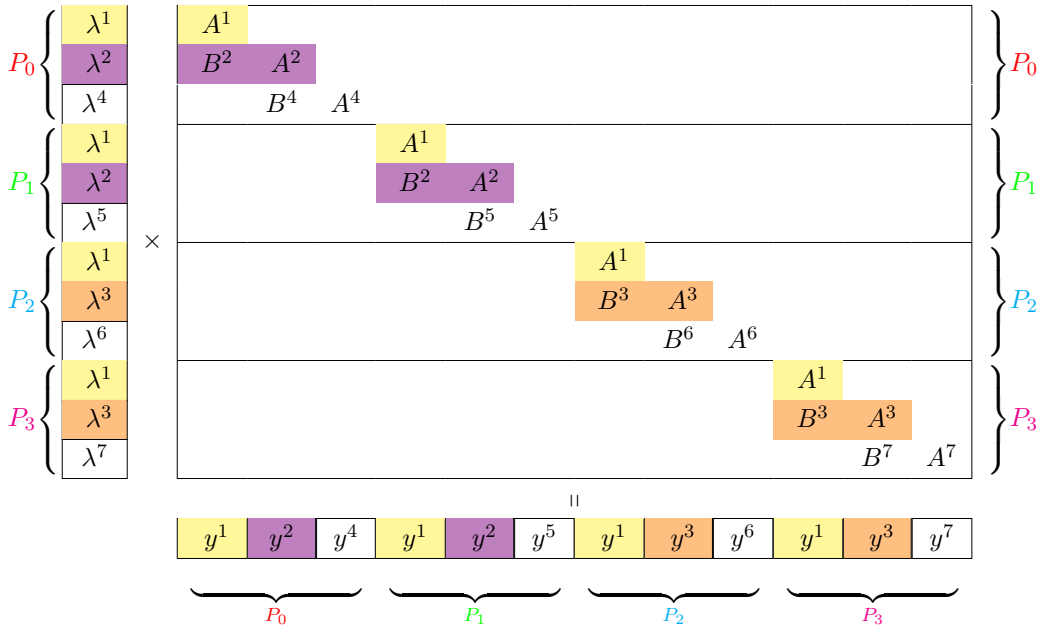


Figure 4.18: Distributed vector-transpose-matrix multiply $\lambda^\top M = y$.

Considering the extensive form in (4.42), y^1 should have a value of $y^1 = (\lambda^1)^\top A^1 + (\lambda^2)^\top B^2 + (\lambda^3)^\top B^3$ in each processor. Therefore, to get the correct y^1-s, in addition to the local vector-transpose-matrix multiplies, our program also has to communicate missing technology matrix products. More precisely, it needs to add $\Lambda^3 = (\lambda^3)^\top B^3$ to processor p_0's and p_1's y^1 and $\Lambda^2 = (\lambda^2)^\top B^2$ to processor p_2's and p_3's y^1. Similarly to y^1, the local vector-transpose-matrix multiplies do not yield the correct values for y^2 and y^3:

In processor p_0: $y^2 = (\lambda^2)^\top A^2 + (\lambda^4)^\top B^4$.

In processor p_1: $y^2 = (\lambda^2)^\top A^2 + (\lambda^5)^\top B^5$.

In processor p_2: $y^3 = (\lambda^3)^\top A^3 + (\lambda^6)^\top B^6$.

In processor p_3: $y^3 = (\lambda^3)^\top A^3 + (\lambda^7)^\top B^7$.

The extensive form (4.42) tells us that processors p_0 and p_1 should have $y^2 = (\lambda^2)^\top A^2 + (\lambda^4)^\top B^4 + (\lambda^5)^\top B^5$ and processors p_2 and p_3 should have $y^3 = (\lambda^3)^\top A^3 + (\lambda^6)^\top B^6 + (\lambda^7)^\top B^7$. In other words, processor p_0's y^2 is missing $\Lambda^5 = (\lambda^5)^\top B^5$, processor p_1's y^2 is missing $\Lambda^4 = (\lambda^4)^\top B^4$, processor p^2's y^3 is missing $\Lambda^7 = (\lambda^7)^\top B^7$, and

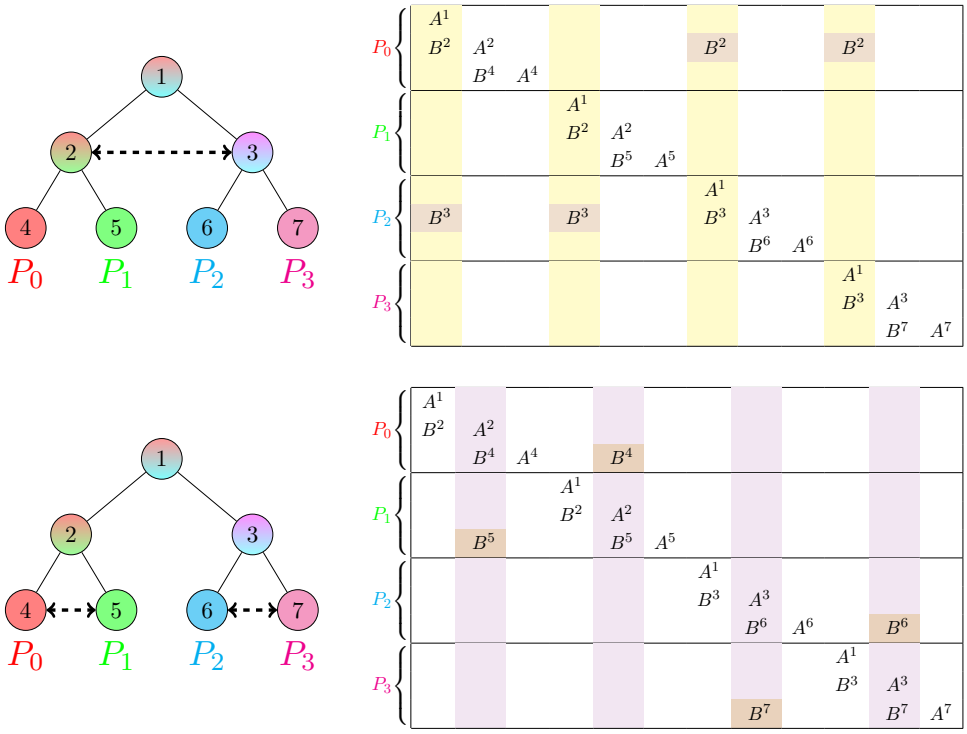


Figure 4.19: Missing data for $\lambda^\top M$ multiply.

processor p^3's y^3 is missing $\Lambda^6 = (\lambda^6)^\top B^6$. The missing technology matrix data for the distributed vector-transpose-matrix multiply in Figure 4.18 is illustrated in Figure 4.19.

Let us consider vector-transpose-matrix multiplies in general. For any node ν we have

$$y^\nu = (\lambda^\nu)^\top A^\nu + u^\nu, \tag{4.46}$$

where $u^\nu = \sum_{\gamma \in \mathcal{G}^\nu} \Lambda^\gamma$ and $\mathcal{G}^\nu$ is the set of children of node ν. If $\mathcal{G}^\nu = \emptyset$ then u^ν equals the zero vector. To see why this is the case, consider the general stochastic programming formulation in (4.40). Note, that technology matrix data in a stage-t constraint, where $t > 1$, can only correspond to stage-$t-1$ variables (stage 1 constraints do not contain any technology matrix data). Therefore, if node ν is at stage $t-1$ in the scenario tree ($t > 1$), then only stage-t constraints corresponding to its children can contain technology matrix data related to node ν variables. In other words, the block of columns in M that corresponds to y^ν can only contain technology matrices

B^γ, where $\gamma \in \mathcal{G}^\nu$. Our goal is to be able to compute u^ν for each non-leaf node in the scenario tree.

A scenario S_i and a stage t uniquely identify a node ν. For example, as shown in Figure 4.12 S_0's stage-3 node is 4. Let us define for every scenario S_i, Λ_i^t the technology matrix product of S_i's stage-t node, and the vector v_i

$$v_i = \begin{bmatrix} \Lambda_i^2 \\ \vdots \\ \Lambda_i^T \end{bmatrix}. \tag{4.47}$$

We will show that each processor p_i can compute u^ν-s necessary for their respective y_i^ν computations, with essentially, two parallel scans on slightly modified v_i vectors. At this point, for the sake of simplicity, we postpone specifying the nature of these modifications.

Given P processors and their data v_0, v_1, v_2, ..., v_{P-1}, a *parallel scan*, also known as parallel prefix sum [46, 16] or just simply scan, of the sequence v_0, v_1, v_2, ..., v_{P-1}, is its sequence of partial "sums" u_0, u_1, u_2, ..., u_{P-1}:

$$\begin{aligned} u_0 &= v_0 \\ u_1 &= v_0 + v_1 \\ u_2 &= v_0 + v_1 + v_2 \\ &\vdots \\ u_{P-1} &= u_{P-2} + v_{P-1}. \end{aligned} \tag{4.48}$$

A scan can be defined for any associative operator $\diamond$: $u_i = u_{i-1} \diamond v_i$. A *segmented scan* is a scan that resets the accumulation when it crosses a user-defined "segment" or boundary. An example of a segmented scan with two segments on 4 processors and the sequence v_0, v_1, v_2, v_3 is

$$\begin{aligned} u_0 &= v_0 \\ u_1 &= u_0 + v_1 \\ u_2 &= v_2 \\ u_3 &= u_2 + v_3, \end{aligned} \tag{4.49}$$

where the segment boundary is at $i = 1$.

For the $\lambda^\top M = y$ operation, `DistributedLinStochProgAugLagAslProblem` class routines use segmented scans on v_i-s to spread the missing data across processors. First, let us note that each processor p_i can compute its own v_i vector locally, since it owns all the data of scenario S_i. Each v_i is made up of blocks of v_i^t, where v_i^t are coefficients of v_i that belong to stage-t technology matrix products, where $t = 2, \ldots, T$.

When "scanning" v_i^t-s we determine the segments based on the parent nodes of S_i's stage-t node. The segment boundaries between v_i^t-s are at every such i, where the parent of S_i's stage-t node is different from the parent of S_{i+1}'s stage-t node. Given our 3 stage example, the boundary for v_i^3-s, $i = 0, \ldots, 3$, is at $i = 1$ because the parent node of 5 (S_1's stage-3 node) is node 2, however, the parent of node 6 (S_2's stage-3 node) is node 3. In general, we can say that the number of segments for v_i^t-s depends on the number of nodes at stage $t - 1$. For example, for v_i^2-s, $i = 0, \ldots, 3$, we define only one segment because at stage 1 the scenario tree only has one node, which is the parent of all nodes at stage 2, whereas for v_i^3-s, $i = 0, \ldots, 3$, we define 2 segments because there are two nodes at stage 2. Figure 4.20 shows a segmented scan using the addition operator, on v_i-s, $i = 0, \ldots, 3$, for the 3-stage example. If the binary operator of a scan is the addition operator and the first element of the sequence is v_0 owned by processor p_0, then we refer to it as *forward scan*. Given that $v_i^t = \Lambda_i^t$ (4.47) we conclude that the forward scan only yields final results for u_1^3 and u_3^3, which are equivalent to u^2 and u^3 respectively. The remainder of the u_i^t-s are still not final meaning that they do not yield u^ν-s. In order for all u_i^t-s to yield u^ν-s we have to do two things:

1. Remove redundant v_i^2-s from stage-2 blocks.
2. Perform a second scan with a different operator.

To remove redundant v_i^2-s we will modify our v_i vectors in the following manner; $v_i^t = \Lambda_i^t$ if S_i's stage-t node is a leaf or if the leaf node of S_i is the leftmost descendant of S_i-s stage-t node, otherwise $v_i^t = 0$. Figure 4.21 shows the forward scan on our modified v_i-s, $i = 0, \ldots, 3$, for the 3-stage example, which yields final results for u_2^2, u_3^2, u_1^3, and u_3^3, because $u_2^2 = u^1$, $u_2^3 = u^1$, $u_1^3 = u^2$, and $u_3^3 = u^3$. Note, that at this point we have computed all the necessary u^ν-s for our 3-stage example. Finally, using a

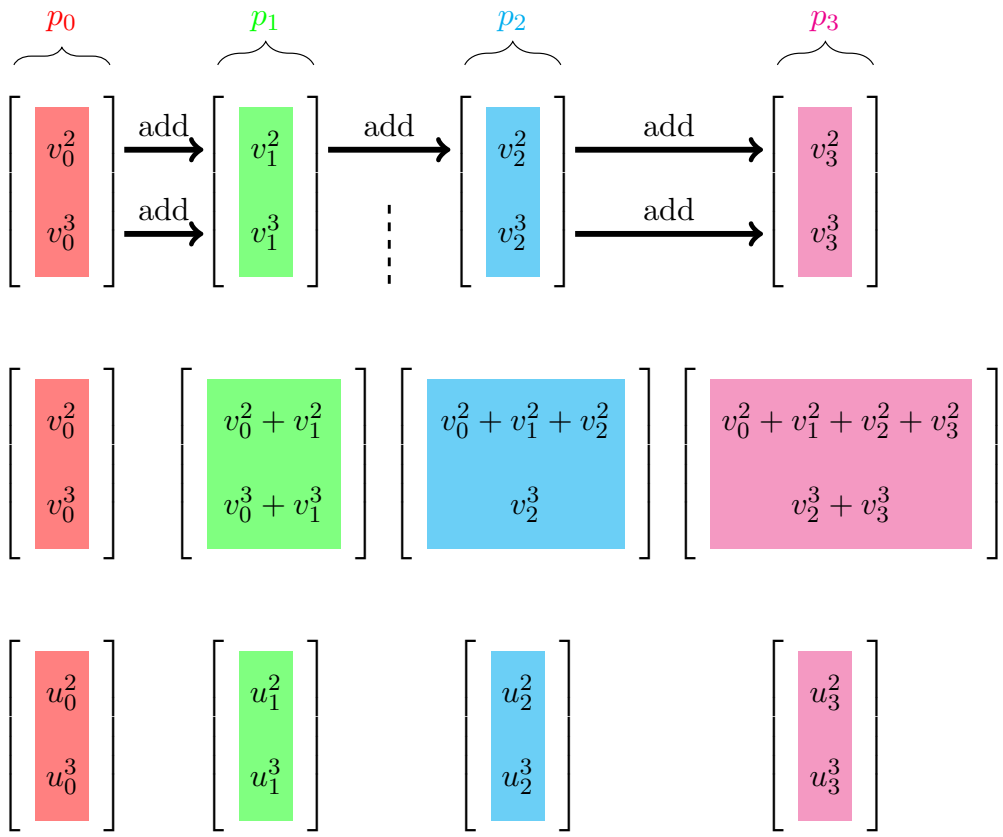


Figure 4.20: Forward scan of v_i-s, $i = 0, \ldots, 3$.

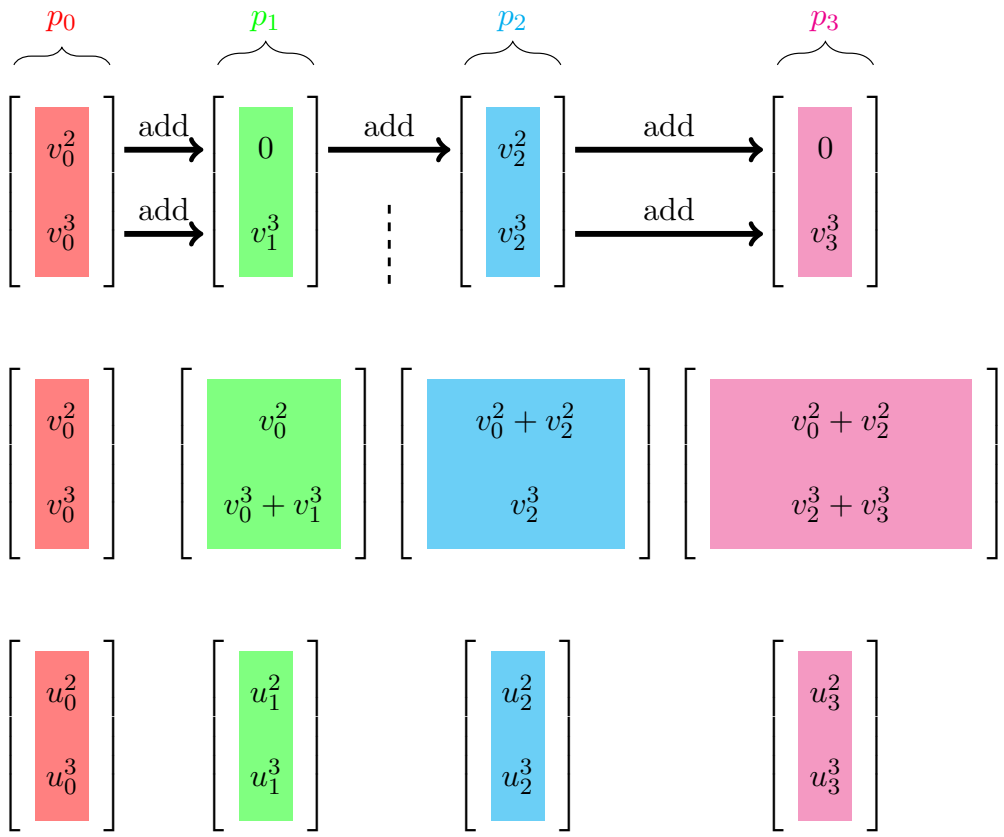


Figure 4.21: Forward scan on redefined v_i-s, $i = 0, \ldots, 3$.

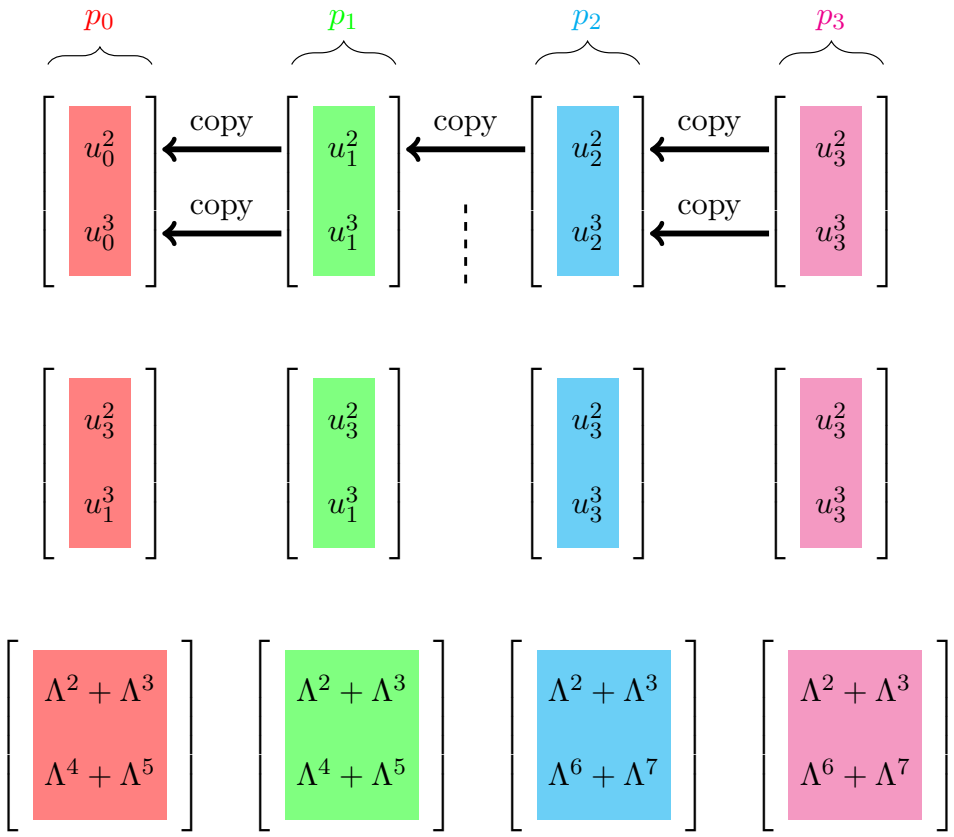


Figure 4.22: Backward scan on u_i-s, $i = 0, \ldots, 3$.

backward scan, where the first element of the sequence is u_{P-1}, the result of the forward scan owned by processor p_{P-1}, and the binary operator is $=$. The backward scan on u_i-s, $i = 0, \ldots, 3$ is illustrated in Figure 4.22. After the backward scan all processors p_i have the necessary technology matrix products (Λ^ν) needed to compute their respective y^ν-s.

Given P processors and a general scenario tree $\mathcal{T}$ with T stages and r scenarios such that $r = P$, the steps below summarize how each processor p_i receives the necessary u^ν-s that are required to compute their respective y_i^ν-s:

1. Each processor p_i computes their modified v_i vectors, where v_i^t a stage-t block of v_i equals Λ_i^t, if scenario S_i's stage-t node is a leaf, or if the leaf node of S_i is the leftmost descendant of S_i's stage-t node.

2. Perform a forward scan on v_i-s, where the segment boundaries between v_i^t-s are at every such i, where the parent of S_i's stage-t node is different from the parent of S_{i+1}'s stage-t node.

3. Perform a backward scan on u_i-s, the result of the forward scan, using the same segments as defined above.

The necessary information that a processor needs to perform the forward and backward scans is provided to `DistributedLinStochProgAugLagAslProblem` by our `PySP` plugin, which adds related suffix information to the `nl` files. Based on this information, each processor p_i is aware of the necessary segment, node and stage information needed to determine whether its scenario's (S_i) stage-t node is a leaf, or if the leaf node of S_i is the leftmost descendant of S_i's stage-t node.

The communication complexity of scan operations on P processors is $\mathrm{O}(v \log P)$, where v is the local vector length [46]. Therefore, the overall complexity of computing $\lambda^\top M$ is

$$\mathrm{O}\left(\sum_{t=1}^{T-1} n_t \log P\right), \tag{4.50}$$

where n_t is the length of v_i^t, which is equivalent to $(\lambda^\nu)^\top B^\nu$ for nodes ν at stage t. Thus,

$$\mathrm{O}\left(\sum_{t=1}^{T-1} n_t \log P\right) \subset \mathrm{O}\left(n \log P\right),$$

where n is the total number of decision variables in one scenario. In practice, `OPAL` performs a sparsity check on all B^ν-s, which detects all columns of B^ν that are zero. If a column of B^ν is all zeros `OPAL` does not need to communicate the coefficients of v_i^t associated with that column. Hence, `OPAL` adjusts the size of v_i^t-s according to the results of the sparsity check.

Vector Operations

Vector addition and scalar multiplication operations, such as $x_{k+1} = x_k + \alpha d_k$, are automatically parallel and communication free because of the data distribution and the underlying parallel vector class. However, inner products for both primal (x) and dual (λ) variables need to account for variable replication. `DistributedLinStochProgAugLagAslProblem`'s constructor passes the appropriate replication information to the

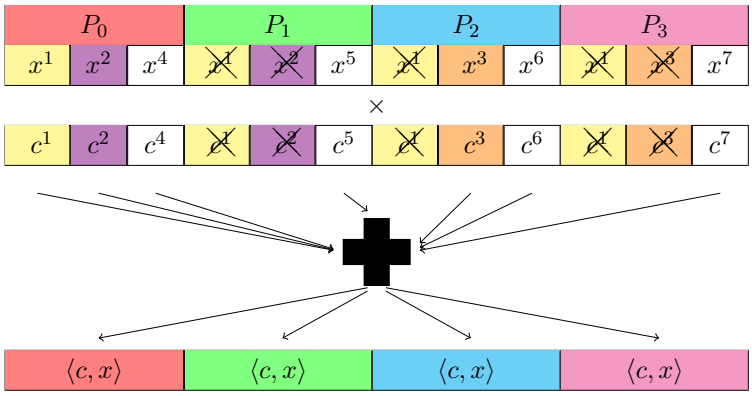


Figure 4.23: Global inner product $\langle c, x \rangle$ for (4.42).

parallel vector class' constructor through the `VectorObject` method `setAvpMode()`. For example, the global $\langle c, x \rangle$ computation for (4.42) is illustrated in Figure 4.23.

4.5.3 Computational Results

We tested OPAL on the continuous relaxation of a multistage stochastic capacity expansion problem modeled by Ahmed et al. [1]. The deterministic model considers T stages, which is the planning horizon, over which the capacity investment costs, and demands are known. The goal is to determine, what levels of resources ($\mathcal{I}$) to acquire and at which stage, in order to satisfy the demand, while minimizing the total cost over the entire planning horizon. Let us denote with x_{it} the capacity expansion of resource type $i \in \mathcal{I}$ at stage t and y_{it} to denote the binary variable for the corresponding capacity expansion decision. Thus, the deterministic problem takes the following form

$$\min \quad \sum_{t=1}^{T} \sum_{i \in \mathcal{I}} \left(c_{it}^{\mathrm{v}} x_{it} + c_{it}^{\mathrm{f}} y_{it} \right) \tag{4.51}$$

$$\text{s.t.} \quad 0 \leq x_{it} \leq M_{it} y_{it} \qquad t = 1, \ldots, T;\ i \in \mathcal{I} \tag{4.52}$$

$$\sum_{\tau=1}^{t} \sum_{i \in \mathcal{I}} x_{i\tau} \geq d_t \qquad t = 1, \ldots, T \tag{4.53}$$

$$y_{it} \in \{0, 1\} \qquad t = 1, \ldots, T;\ i \in \mathcal{I}, \tag{4.54}$$

where c_{it}^{v} is the variable investment cost, c_{it}^{f} the fixed investment cost, d_t the demand parameter, and M_{it} the variable upper bounds on the capacity additions. Constraint (4.52) enforces that capacity acquisition levels are bounded, and constraint (4.53) ensures that the production capacity is sufficiently large to satisfy the demands. For the stochastic version, we assume that $\xi_t = \left(c_{it}^{\mathrm{v}}, c_{it}^{\mathrm{f}}, d_t\right)$ evolve as discrete-time stochastic processes with a finite probability space.

Because of y_{it} this formulation is not continuous, therefore, we have relaxed the integrality constraint (4.54) to $y_{it} \in [0, 1]$. In addition, problem (4.51)- (4.54) is also not in the form presented in (4.40), because of the inequality constraints and because the demand constraints (4.53) corresponding to stage t include variables up to stage 1, whereas the formulation in (4.40) only allows for a stage t constraint to include variables that belong to stage t or $t-1$. `OPAL` is capable of directly handling inequality constraints, so we do not have to add slack variables to convert them into equality constraints. However, we need to add another set of equality constraints to remove the dependency of stage t demand constraints on variables of stage $t-2$ and earlier. The relaxed reformulation thus has the following form

$$\begin{aligned}
\min \quad & \sum_{t=1}^{T} \sum_{i \in \mathcal{I}} \left(c_{it}^{\mathrm{v}} x_{it} + c_{it}^{\mathrm{f}} y_{it} \right) \\
\text{s.t.} \quad & \sum_{i \in \mathcal{I}} x_{i1} - z_1 = 0 \\
& \sum_{i \in \mathcal{I}} x_{it} + z_{t-1} - z_t = 0 && t = 2, \ldots, T \\
& x_{it} - M_{it} y_{it} \leq 0 && t = 1, \ldots, T;\ i \in \mathcal{I} \\
& z_{it} \geq 0,\ y_{it} \in [0, 1],\ z_t \geq d_t && t = 1, \ldots, T;\ i \in \mathcal{I},
\end{aligned} \tag{4.55}$$

which is in the form of (4.1), a general convex optimization problem with bound constraints X.

We ran the test on the Caliburn supercomputer [62] at Rutgers Discovery Informatics Institute (RDI^2). Each node of this system consists of two 2.1 GHz 18-core Xeon processors with 256GB of RAM, and the nodes are connected by Intel's Omni-Path interconnect network. Figure 4.24 illustrates a weak-scaling graph of the runtimes per

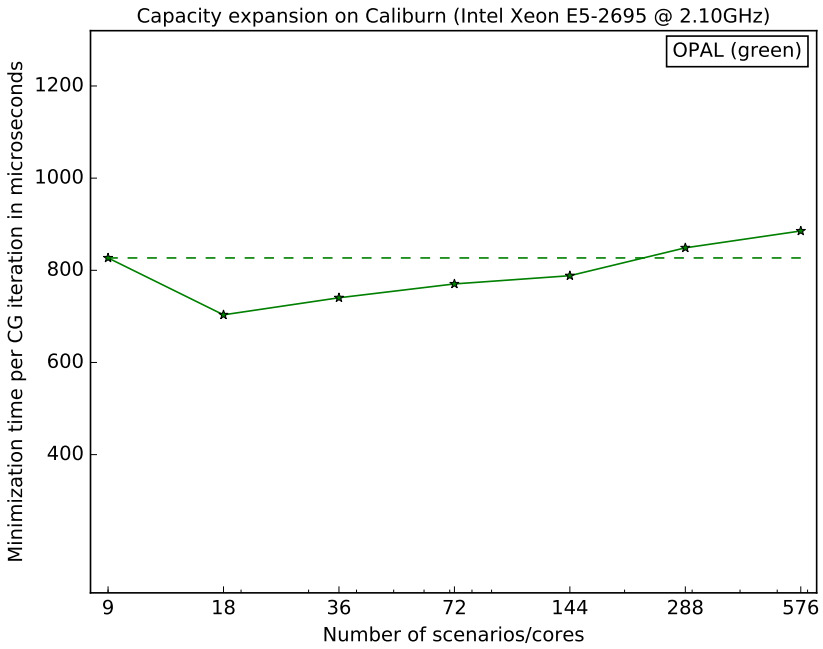


Figure 4.24: Weak-scaling graph of `OPAL` on a multi-stage linear stochastic program.

`QCG` iteration. The graph was obtained by increasing the number of processors proportionally to the number of scenarios, beginning with 9 and ending with 576, which ensured that the test runs on multiple nodes used all the 36 cores available on a Caliburn node. The linear programming problem corresponding to the 576-scenario case has about 2 million variables and about 1 million constraints. Figure 4.24 shows that `OPAL` exhibits exceptional weak-scaling efficiency in terms of the inner loop, where most of the work is performed.

Testing `OPAL` for strong scaling efficiency is currently only possible in a limited form, since strong scaling tests would require the development of a `PySP` plugin that is capable of generating `nl` files that contain "bundled" scenarios. While writing such a plugin is theoretically possible, it is currently outside the scope of this thesis. Instead, we have compared `OPAL` with `ALGENCAN` on a 9-scenario capacity expansion problem that has $40,010$ variables and $20,010$ constraints. For the strong scaling test of `OPAL` we will use the following speedup formula

$$S(N,P) = \frac{T(N,1)}{T(N,P)} \times 100\%, \tag{4.56}$$

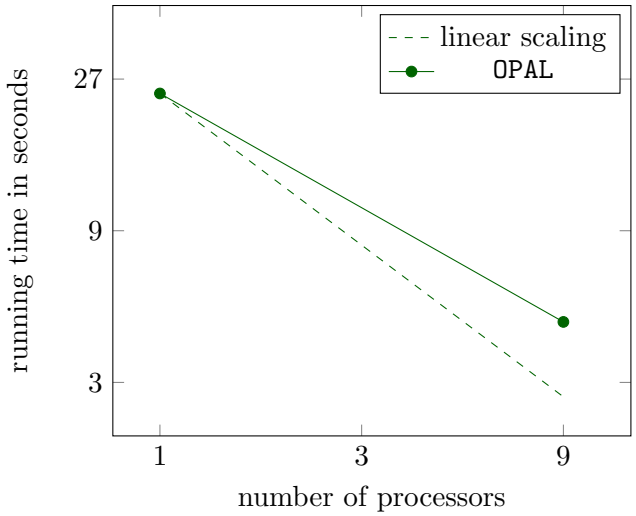


Figure 4.25: Comparing ALGENCAN to OPAL.

where $T(N, 1)$ is the running time of ALGENCAN. Before discussing the results, we would like to mention that because of how PySP generates the extensive form of a stochastic program, the problems given to OPAL and ALGENCAN are not identical. ALGENCAN's problem formulation includes additional constraints and variables, which could be removed from the problem, hence, the solution to the two problems is the same. Nonetheless, ALGENCAN's problem is bigger than OPAL's. However, during the test runs we have allowed ALGENCAN to use various procedures that would boost its performance such as, objective and constraint scaling, fixed variable removal, dynamic adjusting of the penalty parameter and a special acceleration process [12, p. 161]. ALGENCAN, which is a serial solver, solves the problem instance in 24.3 seconds, whereas OPAL solves it in 4.6 seconds using 9 processors. OPAL's strong scaling graph is shown Figure 4.25.

Figure 4.26 illustrates OPAL's parallel efficiency, which is calculated by the following formula

$$E(N, P) = \frac{T(N, 1)}{P \times T(N, P)}, \tag{4.57}$$

where $T(N, 1)$ is the running time of ALGENCAN. In Figure 4.26 we insrease the problem size as we increase the number of scenarios to keep the work per processor constant. For 2-, 8-, and 16-scenario problems OPAL and ALGENCAN perform a similar number of outer and inner iterartions, however, since ALGENCAN actually solves a somewhat larger

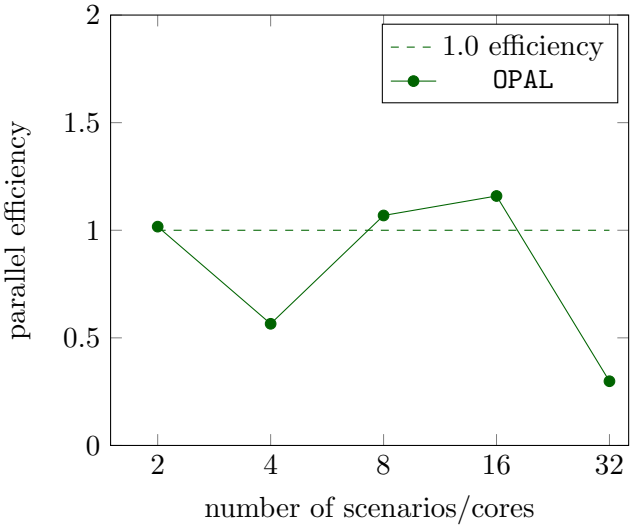


Figure 4.26: Parallel efficiency of OPAL on a log-linear graph.

problem the efficiency can go beyond 1.0. In the 4- and 32-scenario cases ALGENCAN converges faster to the solution than OPAL and performs fewer outer and inner iterations as a result the efficieny decreases substantially.

While OPAL's performance is comparable to other well established nonlinear solvers on smaller problem instances, when it comes to large-scale problems, where both the number of variables and constraints are in the region of $O(10^5)$ or more, we have noticed that the overall performance of OPAL is slow compared to other commercial solvers. We attribute this observation to the following:

1. OPAL does not implement any pre-solve procedures, which in many large-scale instances removes a lot of the constraints and variables from the original problem.
2. QCG does not precondition the constraint matrix M, which could substantially improve the total number of iterations it takes for GENCAN to find a good search direction $\widehat{d}_k$.
3. OPAL performs fewer subproblem termination checks than ALGENCAN, which in some cases results in more work for the subproblem solver.

Chapter 5

Conclusion

The work we have done so far suggests that the object-parallel framework is suitable for the implementation of optimization algorithms efficiently and in a readable manner. Since the resulting source code is clean from "clutter," we assume that it will be much easier to maintain and further improve `OPOS`' code-base. The benefits of the operator overloading techniques are that it makes the solver-level code easier to understand and maintain, and that it facilitates relatively easy development of new solver algorithms. The philosophy is that if the application developers can focus on efficiently performing just a few operations, namely function and gradient evaluation, then our object-parallel framework can use those low-level operations to construct an efficient parallel solver.

Given the strong scaling results of `OPSPG` in Chapter 3, classical first-order methods if implemented in `OPOS`, can be very effective in solving large-scale regression-type problems. These problems are of central importance in the field of machine learning, so it is our goal to extend `OPOS`' capabilities in this area. In particular, we are interested in implementing a novel proximal gradient method [41], which is similar to the SPG algorithm, except that it employs a shrinkage operation instead of the classical projection operation used in SPG. This work would add to the number of robust first-order methods available for large-scale regression-type problems that are less application-dependent than coordinate descent methods. Another useful addition would be to extend our existing parallel problem classes to be able to efficiently handle problems where $A \in \mathbb{R}^{m \times n}$ is tall, i.e. $m \gg n$ or both m and n are large.

While `OPAL`'s inner loop scaling results are very promising, there is still work that needs to be done in order for it to be competitive with existing parallel software for stochastic programing problems. This work, however, is mostly of numerical nature:

ensuring fast convergence of the outer loop of the augmented Lagrangian algorithm. Our experience tells us that numerical algorithms for constrained optimization problems require the adjustment of the algorithm's parameters based on the problem's data and structure. Therefore, we plan to add an `xml`-based problem class that allows users to efficiently manage optimization algorithm parameters. This class could also incorporate routines that would examine the underlying data and make some of these adjustments automatically.

As mentioned at the end of Section 4.5.3, adding a preconditioner would most likely boost `QCG`'s performance and thus so the overall performance of `OPAL`. While adding a parallel preconditioner for general constrained optimization problems requires significant amount of work and research, developing one for linear stochastic programs as presented in Equation (4.40) is a realistic goal and we hope to make this addition to `OPAL` in the future.

Lastly, in this dissertation we have presented an implementation of the augmented Lagrangian algorithm that is based on `ALGENCAN`; however, the augmented Lagrangian algorithm has many variations depending on the subproblem solver that is used for the inner loop. We would like to improve `OPAL`'s capabilities by increasing the number of subproblem solvers that it can use. The first step we have taken in this direction is the implementation of a nonlinear conjugate gradient algorithm by Hager and Zhang [36], which can be used as a subproblem solver in `OPAL` if the original problem does not have any variable bounds or the variable bounds are reformulated explicitly as constraints in the problem.

The proposed future work on `OPAL` will be more straightforward to accomplish than attempting similar work on other methods, like Newton barrier, because `OPAL` does not require parallelization of intricate linear algebra operations such as matrix factorization; instead, it is enough to parallelize a limited number of relatively simple operations. The resulting software will exhibit better tail convergence than decomposition methods.

Index

Bibliography

[1] Shabbir Ahmed, Alan J. King, and Gyana Parija. "A Multi-Stage Stochastic Integer Programming Approach for Capacity Expansion Under Uncertainty". In: *J. of Global Optimization* 26.1 (May 2003), pp. 3–24. ISSN: 0925-5001. DOI: 10.1023/A:1023062915106. URL: https://doi.org/10.1023/A:1023062915106.

[2] Farid Alizadeh. "Interior Point Methods in Semidefinite Programming with Applications to Combinatorial Optimization". In: *SIAM Journal on Optimization* 5.1 (1995), pp. 13–51. DOI: 10.1137/0805002. eprint: https://doi.org/10.1137/0805002. URL: https://doi.org/10.1137/0805002.

[3] MOSEK ApS. *Introducing the MOSEK Optimization Suite 8.1.0.49*. 2018. URL: https://docs.mosek.com/8.1/intro/index.html.

[4] K. Bache and M. Lichman. *UCI Machine Learning Repository*. 2014. URL: http://archive.ics.uci.edu/ml.

[5] Jonathan Barzilai and Jonathan M. Borwein. "Two-Point Step Size Gradient Methods". In: *IMA Journal of Numerical Analysis* 8.1 (1988), pp. 141–148. DOI: 10.1093/imanum/8.1.141. URL: http://dx.doi.org/10.1093/imanum/8.1.141.

[6] D. Bertsekas. "On the Goldstein-Levitin-Polyak gradient projection method". In: *IEEE Transactions on Automatic Control* 21.2 (Apr. 1976), pp. 174–184. ISSN: 0018-9286. DOI: 10.1109/TAC.1976.1101194.

[7] Dimitri P. Bertsekas. *Constrained Optimization and Lagrange Multiplier Methods*. Cambridge, MA: Academic Press, Inc., 1982. ISBN: 1-886529-04-3.

[8] Dimitri P. Bertsekas. *Convex Optimization Algorithms*. Nashua, NH: Athena Scientific, 2015. ISBN: 978-1-886529-28-1.

[9] Dimitri P. Bertsekas. *Convex Optimization Theory*. Belmont, MA: Athena Scientific, 2009. ISBN: 978-1-886529-31-1.

[10] Dimitri P. Bertsekas. *Nonlinear Programming*. 2nd. Belmont, MA: Athena Scientific, 1999. ISBN: 1-886529-00-0.

[11] J.R. Birge and F. Louveaux. *Introduction to Stochastic Programming*. 2nd. New York, NY: Springer, 2011. ISBN: 978-1-4614-0236-7. DOI: 10.1007/978-1-4614-0237-4. URL: https://doi.org/10.1007/978-1-4614-0237-4.

[12] E. Birgin and J. Martínez. *Practical Augmented Lagrangian Methods for Constrained Optimization*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2014. DOI: 10.1137/1.9781611973365. eprint: http://epubs.siam.org/doi/pdf/10.1137/1.9781611973365. URL: http://epubs.siam.org/doi/abs/10.1137/1.9781611973365.

[13] Ernesto G. Birgin and José Mario Martínez. "Large-Scale Active-Set Box-Constrained Optimization Method with Spectral Projected Gradients". In: *Computational Optimization and Applications* 23.1 (Oct. 2002), pp. 101–125. ISSN: 1573-2894. DOI: 10.1023/A:1019928808826. URL: https://doi.org/10.1023/A:1019928808826.

[14] Ernesto G. Birgin, José Mario Martínez, and Marcos Raydan. "Nonmonotone spectral projected gradient methods on convex sets". In: *SIAM J. Optim.* 10.4 (2000), pp. 1196–1211. ISSN: 1052-6234. DOI: 10.1137/S1052623497330963. URL: http://dx.doi.org.proxy.libraries.rutgers.edu/10.1137/S1052623497330963.

[15] *BLAS (Basic Linear Algebra Subprograms)*. Mar. 2018. URL: http://http://www.netlib.org/blas/.

[16] Guy E. Blelloch. *Vector Models for Data-parallel Computing*. Cambridge, MA, USA: MIT Press, 1990. ISBN: 0-262-02313-X.

[17] Richard H. Byrd, Jorge Nocedal, and Richard A. Waltz. "Knitro: An Integrated Package for Nonlinear Optimization". In: *Large-Scale Nonlinear Optimization. Nonconvex Optimization and Its Applications*. Ed. by G. Di Pillo and M. Roma. Vol. 83. Boston, MA: Springer, 2006, pp. 35–59. ISBN: 978-0-387-30063-4. DOI: 10.1007/0-387-30065-1_4. URL: https://doi.org/10.1007/0-387-30065-1_4.

[18] *C++ containers library*. Mar. 2018. URL: http://en.cppreference.com/w/cpp/container.

[19] A. R. Conn, N. I. M. Gould, and P. L. Toint. *Lancelot: A Fortran Package for Large-Scale Nonlinear Optimization (Release A)*. 1st. Springer Publishing Company, Incorporated, 2010. ISBN: 3642081398, 9783642081392.

[20] Cubbi. *abstract class*. Accessed: 2018-01-25. 2013. URL: http://en.cppreference.com/w/cpp/language/abstract_class.

[21] *Cyclomatic complexity*. Mar. 2018. URL: https://en.wikipedia.org/wiki/Cyclomatic_complexity.

[22] Leonardo Dagum and Ramesh Menon. "OpenMP: An Industry-Standard API for Shared-Memory Programming". In: *IEEE Comput. Sci. Eng.* 5.1 (Jan. 1998), pp. 46–55.

[23] Frederica Darema. "The SPMD Model: Past, Present and Future". In: *Proceedings of the 8th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. London, UK, UK: Springer-Verlag, 2001, pp. 1–. ISBN: 3-540-42609-4. URL: http://dl.acm.org/citation.cfm?id=648138.746808.

[24] Robert Fourer, David M. Gay, and Brian W. Kernighan. "A Modeling Language for Mathematical Programming". In: *Manage. Sci.* 36.5 (May 1990), pp. 519–554. ISSN: 0025-1909. DOI: 10.1287/mnsc.36.5.519. URL: http://dx.doi.org/10.1287/mnsc.36.5.519.

[25] Robert Fourer, David M. Gay, and Brian W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press/Wadsworth, 1993. ISBN: 0-9426-232-7.

[26] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. 3rd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0321193687.

[27] P. A. Fox, A. P. Hall, and N. L. Schryer. "The PORT Mathematical Subroutine Library". In: *ACM Trans. Math. Softw.* 4.2 (June 1978), pp. 104–126. ISSN: 0098-3500. DOI: 10.1145/355780.355783. URL: http://doi.acm.org/10.1145/355780.355783.

[28] David M. Gay. *Hooking your Solver to AMPL*. Tech. rep. 97-4-06. Bell Laboratories, Apr. 1997.

[29] David M. Gay. *Writing .nl Files*. Tech. rep. SAND2005-7907P. Sandia National Laboratories, 2005.

[30] Jacek Gondzio, Pablo González-Brevis, and Pedro Munari. "Large-scale optimization with the primal-dual column generation method". In: *Math. Prog. Comp.* 8 (2016), pp. 47–82. DOI: 10.1007/s12532-015-0090-6. URL: https://doi.org/10.1007/s12532-015-0090-6.

[31] Jacek Gondzio and Andreas Grothey. "Exploiting structure in parallel implementation of interior point methods for optimization". In: *Comput. Manag. Sci.* 6.2 (2009), pp. 135–160. DOI: 10.1007/s10287-008-0090-3. URL: http://dx.doi.org/10.1007/s10287-008-0090-3.

[32] Jacek Gondzio and Andreas Grothey. "Parallel interior-point solver for structured quadratic programs: application to financial planning problems". In: *Ann. Oper. Res.* 152 (2007), pp. 319–339. DOI: 10.1007/s10479-006-0139-z. URL: http://dx.doi.org/10.1007/s10479-006-0139-z.

[33] Jacek Gondzio and Andreas Grothey. "Solving non-linear portfolio optimization problems with the primal-dual interior point method". In: *European J. Oper. Res.* 181.3 (2007), pp. 1019–1029.

[34] Andreas Griewank et al. *Automatic differentiation of algorithms : theory, implementation, and application*. English. Conference Proceedings. "Proceedings of the first SIAM Workshop on Automatic Differentiation, held in Breckenridge, Colorado, January 6-8, 1991"–T.p. verso. 1992.

[35] L Grippo, F Lampariello, and S Lucidi. "A Nonmonotone Line Search Technique for Newton's Method". In: *SIAM J. Numer. Anal.* 23.4 (Aug. 1986), pp. 707–716. ISSN: 0036-1429. DOI: 10.1137/0723046. URL: http://dx.doi.org/10.1137/0723046.

[36] William W. Hager and Hongchao Zhang. "A new conjugate gradient method with guaranteed descent and an efficient line search". In: *SIAM J. Optim.* 16.1 (2005), pp. 170–192. ISSN: 1052-6234.

[37] William W. Hager and Hongchao Zhang. "Algorithm 851: CG_DESCENT, a conjugate gradient method with guaranteed descent". In: *ACM Trans. Math. Softw.* 32.1 (2006), pp. 113–137. ISSN: 0098-3500. DOI: 10.1145/1132973.1132979.

[38] William E. Hart et al. *Pyomo–optimization modeling in python*. 2nd. Vol. 67. Springer Science & Business Media, 2017.

[39] Michael A. Heroux et al. "An overview of the Trilinos project". In: *ACM Trans. Math. Softw.* 31.3 (2005), pp. 397–423. ISSN: 0098-3500. DOI: http://doi.acm.org/10.1145/1089014.1089021.

[40] *HPC Tools ARM DDT*. Apr. 2018. URL: https://www.arm.com/products/development-tools/hpc-tools/cross-platform/forge/ddt.

[41] Yakui Huang and Hongwei Liu. "A Barzilai-Borwein Type Method for Minimizing Composite Functions". In: *Numer. Algorithms* 69.4 (Aug. 2015), pp. 819–838. ISSN: 1017-1398. DOI: 10.1007/s11075-014-9927-8. URL: http://dx.doi.org/10.1007/s11075-014-9927-8.

[42] Josef Kallrath, ed. *Modeling Languages in Mathematical Optimization*. Vol. 88. Applied Optimization. Norwell, MA: Kluwer Academic Publishers, 2004. ISBN: 978-1-4020-7547-6. DOI: 10.1007/978-1-4613-0215-5.

[43] N. Karmarkar. "A New Polynomial-time Algorithm for Linear Programming". In: *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*. STOC '84. New York, NY, USA: ACM, 1984, pp. 302–311. ISBN: 0-89791-133-4. DOI: 10.1145/800057.808695. URL: http://doi.acm.org/10.1145/800057.808695.

[44] S. Sathiya Keerthi and Dennis DeCoste. "A modified finite Newton method for fast solution of large scale linear SVMs". In: *Journal of Machine Learning Research* 6 (2005), pp. 341–361.

[45] Ken Kennedy, Charles Koelbel, and Hans Zima. "The Rise and Fall of High Performance Fortran: An Historical Object Lesson". In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. San Diego, California: ACM, 2007, pp. 7-1–7-22. ISBN: 978-1-59593-766-7. DOI: 10.1145/1238844.1238851. URL: http://doi.acm.org/10.1145/1238844.1238851.

[46] Vipin Kumar et al. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1994. ISBN: 0-8053-3170-0.

[47] C. L. Lawson et al. "Basic Linear Algebra Subprograms for Fortran Usage". In: *ACM Trans. Math. Softw.* 5.3 (Sept. 1979), pp. 308–323. ISSN: 0098-3500. DOI: 10.1145/355841.355847. URL: http://doi.acm.org/10.1145/355841.355847.

[48] *LIBSVM Data: Classification, Regression, and Multi-label*. Apr. 2018. URL: https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/.

[49] *Matrix Market Exchange Formats*. Apr. 2018. URL: https://math.nist.gov/MatrixMarket/formats.html.

[50] *Matrix Storage Schemes for BLAS Routines*. Apr. 2018. URL: https://software.intel.com/en-us/mkl-developer-reference-c-matrix-storage-schemes-for-blas-routines.

[51] Tom Mitchell. *Machine Learning*. McGraw Hill, 1997. ISBN: 0070428077.

[52] James Murphy. *Benders, nested benders and stochastic programming: An intuitive introduction*. CUED/F-INFENG/TR.675. 2013. URL: https://arxiv.org/abs/1312.3158.

[53] John Nickolls et al. "Scalable Parallel Programming with CUDA". In: *Queue* 6.2 (2008), pp. 40–53.

[54] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. 2nd. New York, NY: Springer-Verlag, 2006. ISBN: 978-0-387-30303-1. DOI: 10.1007/978-0-387-40065-5.

[55] *Paraver: a flexible performance analysis tool*. Apr. 2018. URL: https://tools.bsc.es/paraver.

[56] Z. Peng, M. Yan, and W. Yin. "Parallel and distributed sparse optimization". In: *2013 Asilomar Conference on Signals, Systems and Computers*. Nov. 2013, pp. 659–646. DOI: 10.1109/ACSSC.2013.6810364.

[57] *Portable, Extensible Toolkit for Scientific Computation*. Apr. 2018. URL: https://www.mcs.anl.gov/petsc/.

[58] András Prékopa. *Stochastic Programming*. Kluwer Academic Publishers, 1995.

[59] Peter Richtárik and Martin Takáč. "Distributed Coordinate Descent Method for Learning with Big Data". In: *Journal of Machine Learning Research* 17 (2016).

[60] R. T. Rockafellar and Roger J.-B. Wets. "Scenarios and Policy Aggregation in Optimization Under Uncertainty". In: *Mathematics of Operations Research* 16.1 (1991), pp. 119–147. DOI: 10.1287/moor.16.1.119. eprint: https://doi.org/10.1287/moor.16.1.119. URL: https://doi.org/10.1287/moor.16.1.119.

[61] R.T. Rockafellar. "Augmented Lagrangians and applications of the proximal point algorithm in convex programming". In: *Math. Oper. Res.* 1.3 (1976), pp. 97–116.

[62] *Rutgers Discovery Informatics Institute: Caliburn*. May 2018. URL: https://rdi2.rutgers.edu/aci.

[63] Alexander Shapiro, Darinka Dentcheva, and Andrzej Ruszczyński. *Lectures on Stochastic Programming Modeling and Theory*. 2nd. MPS-SIAM Series on Optimization, 2014.

[64] Kavinesh J. Singh, Andy B. Philpott, and R. Kevin Wood. "Dantzig-Wolfe Decomposition for Solving Multistage Stochastic Capacity-Planning Problems". In: *Operations Research* 57.5 (2009), pp. 1271–1286. DOI: 10.1287/opre.1080.0678. eprint: https://doi.org/10.1287/opre.1080.0678. URL: https://doi.org/10.1287/opre.1080.0678.

[65] Marc Snir et al. *MPI: The Complete Reference*. MIT Press, 1994.

[66] *Sparse BLAS CSC Matrix Storage Format*. Apr. 2018. URL: https://software.intel.com/en-us/mkl-developer-reference-fortran-sparse-blas-csc-matrix-storage-format.

[67] *Stampede User Guide*. URL: https://portal.tacc.utexas.edu/archives/stampede.

[68] *Standard Template Library*. Mar. 2018. URL: https://en.wikipedia.org/wiki/Standard_Template_Library.

[69] Bjarne Stroustrup. *The C++ Programming Language*. 3rd. Boston, MA, USA: Addison-Wesley, 2000.

[70] *Texas Advanced Computing Center (TACC) The University of Texas at Austin*. URL: http://www.tacc.utexas.edu.

[71] Robert Tibshirani. "Regression shrinkage and selection via the lasso: a retrospective". In: *J. R. Stat. Soc.* 73.3 (2011), pp. 273–282.

[72] Philippe Toint. *Subroutine VE08: Harwell subroutine library*. AEA Tech. 2. 1995a, pp. 1162–1174.

[73] J. Towns et al. "XSEDE: Accelerating Scientific Discovery". In: *Computing in Science & Engineering* 16.5 (Sept. 2014), pp. 62–74. ISSN: 1521-9615. DOI: 10.1109/MCSE.2014.80. URL: doi.ieeecomputersociety.org/10.1109/MCSE.2014.80.

[74] *Trilinos*. Apr. 2018. URL: https://trilinos.org/.

[75] Robert J. Vanderbei. "LOQO: An Interior Point Code for Quadratic Programming". In: *Optimization Methods and Software* 11 (1999), pp. 451–484.

[76] V. Vijay Vazirani. *Approximation Algorithms*. New York, NY, USA: Springer-Verlag, 2001, 2003. ISBN: 978-3-540-65367-7.

[77] Andreas Wächter and Lorenz T. Biegler. "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming". In: *Math. Program.* 106 (2006), pp. 25–57. DOI: 10.1007/s10107-004-0559-y. URL: https://doi.org/10.1007/s10107-004-0559-y.

[78] Jean-Paul Watson and David L. Woodruff. "Progressive hedging innovations for a class of stochastic mixed-integer resource allocation problems". In: *Computational Management Science* 8.4 (Nov. 2011), pp. 355–370. ISSN: 1619-6988. DOI: 10.1007/s10287-010-0125-4. URL: https://doi.org/10.1007/s10287-010-0125-4.

[79] Jean-Paul Watson, David L Woodruff, and William E Hart. "PySP: modeling and solving stochastic programs in Python". In: *Mathematical Programming Computation* 4.2 (June 2012), pp. 109–149. ISSN: 1867-2957. DOI: 10.1007/s12532-012-0036-1. URL: https://doi.org/10.1007/s12532-012-0036-1.

[80] Nancy Wilkins-Diehr et al. "An overview of the XSEDE extended collaborative support program". English (US). In: *High Performance Computer Applications - 6th International Conference, ISUM 2015, Revised Selected Papers*. Vol. 595. Communications in Computer and Information Science. Germany: Springer Verlag, Jan. 2016, pp. 3–13. ISBN: 9783319322421. DOI: 10.1007/978-3-319-32243-8_1.

[81] Stephen J. Wright. *Primal-dual Interior-point Methods*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1997. ISBN: 0-89871-382-X.

[82] Victor M Zavala, Carl D Laird, and Lorenz T Biegler. "Interior-point decomposition approaches for parallel solution of large-scale nonlinear parameter estimation problems". In: *Chemical Engineering Science* 63 (2008), pp. 4834–4845. URL: https://doi.org/10.1016/j.ces.2007.05.022.

Appendix A

Computational Tools

We have used a variety of computational tools to test, improve and run OPOS. This appendix provides a summary of these tools.

A.1 OPSPG Tools

Results presented in Section 3.4.3 were obtained by running OPSPG on TACC's Stampede [67], which since has been decommissioned. Stampede's system specifications can be seen in Figure A.1. Figure A.2 lists build tools and libraries used to compile and run OPSPG on Stampede.

A.2 OPAL Tools

Results presented in Section 4.5.3 were obtained by running OPAL on RDI^2's Caliburn [62]. Caliburn's system specifications can be seen in Figure A.3. Figure A.4 lists build tools and libraries used to compile and run OPAL on Caliburn.

As mentioned in Section 4.5.3 scaling results for OPAL are limited to weak-scaling

Stampede	
Compute Nodes	
Host processor	Dual Xeon E5-2680 @ 2.7GHz (16 cores per node)
Coprocessor	1/61 Xeon Phi SE10P @ 1.1GHz (not used)
Interconnect	
Network	Mellanox FDR InfiniBand
File system	
Parallel file system	Lustre

Figure A.1: Stampede system specifications.

OPSPG on Stampede	
Compiler	Intel icc
MPI	MVAPICH2
Dense BLAS	Intel MKL and Epetra
Sparse BLAS	Intel MKL and Epetra

Figure A.2: Build tools and libraries used on Stampede.

Caliburn	
Compute Nodes	
Processor	Dual Xeon E5-2695v4 @ 2.1GHz (36 cores per node)
Interconnect	
Network	Intel Omni-Path

Figure A.3: Caliburn system specifications.

OPAL on Caliburn	
Compiler	GNU gcc
MPI	MVAPICH
Dense BLAS	OpenBLAS
Sparse BLAS	Custom

Figure A.4: Build tools and libraries used on Caliburn.

	Outside MPI	MPI_Barrier	MPI_Allreduce	MPI_Comm_rank	MPI_Comm_size	MPI_Comm_create	MPI_Scan	MPI_Init	MPI_Finalize
THREAD 1.1.1	91.24 %	0.00 %	6.77 %	0.39 %	0.00 %	0.00 %	1.57 %	0.00 %	0.03 %
THREAD 1.2.1	90.91 %	0.01 %	7.00 %	0.38 %	0.00 %	0.00 %	1.68 %	0.02 %	0.01 %
THREAD 1.3.1	90.64 %	0.01 %	7.28 %	0.37 %	0.00 %	0.00 %	1.68 %	0.01 %	0.01 %
THREAD 1.4.1	90.49 %	0.01 %	7.38 %	0.38 %	0.00 %	0.00 %	1.71 %	0.01 %	0.01 %
THREAD 1.5.1	90.52 %	0.00 %	7.37 %	0.37 %	0.00 %	0.00 %	1.70 %	0.01 %	0.02 %
THREAD 1.6.1	90.40 %	0.01 %	7.48 %	0.38 %	0.00 %	0.00 %	1.70 %	0.01 %	0.02 %
THREAD 1.7.1	90.65 %	0.00 %	7.25 %	0.38 %	0.00 %	0.00 %	1.68 %	0.01 %	0.02 %
THREAD 1.8.1	90.73 %	0.00 %	7.19 %	0.38 %	0.00 %	0.00 %	1.68 %	0.01 %	0.03 %
THREAD 1.9.1	90.50 %	0.01 %	7.44 %	0.38 %	0.00 %	0.00 %	1.64 %	0.00 %	0.03 %
Total	816.08 %	0.04 %	65.16 %	3.41 %	0.00 %	0.01 %	15.04 %	0.08 %	0.17 %
Average	90.68 %	0.00 %	7.24 %	0.38 %	0.00 %	0.00 %	1.67 %	0.01 %	0.02 %
Maximum	91.24 %	0.01 %	7.48 %	0.39 %	0.00 %	0.00 %	1.71 %	0.02 %	0.03 %
Minimum	90.40 %	0.00 %	6.77 %	0.37 %	0.00 %	0.00 %	1.57 %	0.00 %	0.01 %
StDev	0.24 %	0.00 %	0.22 %	0.00 %	0.00 %	0.00 %	0.04 %	0.01 %	0.01 %
Avg/Max	0.99	0.67	0.97	0.98	0.74	0.39	0.98	0.54	0.62

Figure A.5: Communication profile of `OPAL` on a 9-scenario problem.

tests. Therefore, we have created a communication profile of `OPAL` on Caliburn using `PARAVER` [55] to gain a better understanding of `OPAL`'s performance. The profile was run on a 9-scenario instance with 9 processors; its results are depicted in Figure A.5. Column `Outside MPI` (blue) shows for each processor the total amount of time it spends outside of `MPI` calls, the remainder of the columns (green) show time spent in specific `MPI` function calls. Figure A.5 shows that `OPAL` only spends 10% of its running time communicating.