Parallel Problem Generation for Structured Problems in Mathematical Programming

Feng Qiang

Doctor of Philosophy
University of Edinburgh
2015

Declaration

I declare that this thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text.

(*Feng Qiang*)

7 August 2015

Acknowledgements

First of all, I would like to thank my principle PhD supervisor Dr. Andreas Grothey, who is patiently guiding me through the course of my PhD research. Without his guidance, I will never be able to reach at this point. As a PhD supervisor, Dr. Grothey is responsible and considerate, and I have learned a lot from him for conducting research works. As a mentor and friend, Andreas is very friendly and always willing to help and to share his valuable life experiences. It has been a great time to work with him for all these years in Edinburgh.

I would also like to express my gratitude to my second supervisor Prof. Jacek Gondzio who is always there to offer me advice and support on both research and career directions.

I would also like to thank Robert M. Gower, with whom I had invaluable discussion on the automatic differentiation algorithms and their implementation issues.

Furthermore, I would like to thank the University of Edinburgh who provided the funding support for my entire PhD study and brought me this great opportunity to enhance my life experience.

Additionally, I would like to thank my parents and family members for their mental and physical supports which greatly helped me through some tough times.

Finally, I would like to thank all my friends and colleagues, especially my office mates: Yiming Yan and Waqquas Ahmed Bukhsh, with whom I had lots of laughter and good memory.

Lay summary

Because of the hierarchical natural of our world, large scale problems can be always described using structured approaches. This hierarchical structure could represent many real life objects, for example, an organization with multiple departments, an event tree of a stochastic process, *etc.* Once an underlying object is described using a mathematical model, decision-makers almost always use a computer-based optimization solver to work out an optimized decision for a given objective function incorporated in the model. Therefore, the model has to be also computerized in order to communicate with the optimization solver. As the cost of the parallel computing hardware decreases, more and more researches are now interested in solving an optimization problem using parallel approaches. In particular, large scale problems may take hours or even days to solve in serial, therefore it is necessary to utilize parallel algorithms and solve them in a timely fashion.

There are quite a few parallel solvers in the field already, but a well accepted parallel modelling approach is not yet established. This is currently identified as a run-time bottle-neck for solving large scale optimization problems.

In this research, we investigated the challenges involved in modelling large scale structured problem in parallel. We also presented a novel methodology for modelling and generating a structured problem in parallel using computer-based tool. The result of this research enabled us to implement a parallel modelling tool, namely Parallel Structured Model Generator (PSMG). PSMG aims to provide non-technical users an easy-to-use modelling framework for describing and solving structured optimization problem using parallel solvers. Our modelling framework also provides interface methods that enable an easy linkage between parallel optimization solvers and the model framework itself.

This thesis also includes benchmarks to demonstrate both serial and parallel performance of the PSMG. The benchmark results are obtained by modelling a few different structured problems of various sizes. The results also demonstrate that PSMG offers a promising run-time performance in both serial and parallel cases.

Abstract

The aim of this research is to investigate parallel problem generation for structured optimization problems. The result of this research has produced a novel parallel model generator tool, namely the Parallel Structured Model Generator (PSMG). PSMG adopts the model syntax from SML to attain backward compatibility for the models already written in SML [1]. Unlike the proof-of-concept implementation for SML in [2], PSMG does not depend on AMPL [3].

In this thesis, we firstly explain what a *structured* problem is using concrete real-world problems modelled in SML. Presenting those example models allows us to exhibit PSMG's modelling syntax and techniques in detail. PSMG provides an easy to use framework for modelling large scale nested structured problems including multi-stage stochastic problems. PSMG can be used for modelling linear programming (LP), quadratic programming (QP), and nonlinear programming (NLP) problems.

The second part of this thesis describes considerable thoughts on logical calling sequence and dependencies in parallel operation and algorithms in PSMG. We explain the design concept for PSMG's solver interface. The interface follows a *solver driven work assignment* approach that allows the solver to decide how to distribute problem parts to processors in order to obtain better data locality and load balancing for solving problems in parallel. PSMG adopts a delayed constraint expansion design. This allows the memory allocation for computed entities to only happen on a process when it is necessary. The computed entities can be the set expansions of the indexing expressions associated with the variable, parameter and constraint declarations, or temporary values used for set and parameter constructions. We also illustrate algorithms that are important for delivering efficient implementation of PSMG, such as routines for partitioning constraints according to blocks and automatic differentiation algorithms for evaluating Jacobian and Hessian matrices and their corresponding sparsity partterns. Furthermore, PSMG implements a generic solver interface which can be linked with different structure exploiting optimization solvers such as decomposition or interior point based solvers. The work required for linking with PSMG's solver interface is also discussed.

Finally, we evaluate PSMG's run-time performance and memory usage by generating structured problems with various sizes. The results from both serial and parallel executions are discussed. The benchmark results show that PSMG achieve good parallel efficiency on up to 96 processes. PSMG distributes memory usage among parallel processors which enables the generation of problems that are too large to be processed on a single node due to memory restriction.

Contents

Abstract **5**

1 Introduction **12**
1.1 Background of Algebraic Modelling Language 12
1.2 Structured Modelling Approaches Review 13
1.3 Motivation of Parallelization 14
1.4 PSMG overview 15
1.5 Structure of this Thesis 15

2 Structured Problems and Modelling Techniques **17**
2.1 SML Modelling Syntax Review 17
2.1.1 SML Keywords 17
2.1.2 Block Model 18
2.1.3 Stochastic Model 18
2.1.4 Explicit Variable Referencing 19
2.2 MSND: A Nested-block Structured Problem 19
2.3 ALM-SSD: A Multistage Stochastic Programming Problem 23
2.4 Discussion 27
2.4.1 Block Separability Assumption 27
2.4.2 Objective Function Treatment 27

3 Parallel Solvers for Structured Problems **29**
3.1 Structure Exploiting in Interior Point Method 29
3.1.1 Linear Algebra in IPM 29
3.1.2 Structure Exploiting in IPM 33
3.1.3 Parallel Allocation and Solving 33
3.2 Decomposition Algorithms 35
3.2.1 Benders' Decomposition 36
3.2.2 Parallel allocation and solving 37
3.3 Discussion 38
3.3.1 Modelling Nonconvex Problems 39

4 PSMG Design and Implementation **40**
4.1 Development Environment and Technique 40
4.2 Block and Stochastic Model 41
4.3 Structure Building Stage 43
4.3.1 Template Model Tree 43

4.3.2 Expanded Model Tree 47
4.4 Parallel Problem Generation Stage 49
4.4.1 Solver Driven Problem Assignment 49
4.4.2 Constraint Separability Detection 52
4.4.3 Function and Derivative Evaluation 52

5 Evaluating Derivatives in PSMG 54
5.1 Forward AD Algorithm 54
5.2 Reverse AD Algorithm 55
5.3 Computing Sparse Hessian 57
5.4 AutoDiff Library Module 60
5.5 Future Work in the AD Module 62
5.5.1 Some Parallel Consideration 62

6 PSMG Solver Interface 63
6.1 General Problem Formulation 63
6.2 Problem Structure Retrieval 66
6.3 Variable and Constraints Information 67
6.4 Local and Distributed Interface Methods 69
6.4.1 Inter-process Communication 70
6.4.2 Interface Method Summary 71
6.5 LP Problem Interface 71
6.5.1 Constraint Matrix Evaluation 72
6.5.2 Constraint Function Evaluation 74
6.5.3 Objective Function Evaluation 76
6.5.4 Objective Gradient Evaluation 77
6.6 QP Problem Interface 78
6.6.1 Hessian Matrix Evaluation 79
6.6.2 Objective Function Evaluation 80
6.6.3 Objective Gradient Evaluation 81
6.7 NLP Problem Interface 83
6.7.1 Jacobian Matrix Evaluation 84
6.7.2 Hessian of the Lagrangian Matrix Evaluation 88
6.7.3 Constraint Function Evaluation 94
6.7.4 Objective Function Evaluation 96
6.7.5 Objective Gradient Evaluation 97
6.8 Summary 97

7 Linking With Parallel Solvers 99
7.1 Structure Exploiting Interior Point Method 99
7.1.1 Building Matrix Structure 99
7.1.2 Building Vector Structure 102
7.1.3 Parallel Process Allocation 104
7.1.4 Parallel Problem Generation 105
7.2 Benders' Decomposition 107
7.2.1 Parallel problem generation 107

8 PSMG Performance Benchmark 109
8.1 LP and QP Problems . . . 109
8.1.1 Test Problem Sets . . . 109
8.1.2 Comparison Analysis with SML . . . 110
8.1.3 Serial Performance . . . 112
8.1.4 Parallel Performance . . . 114
8.1.5 Memory Usage Analysis . . . 114
8.2 NLP Problems . . . 116
8.2.1 SCOPF Problem . . . 117
8.2.2 Serial Performance . . . 118
8.2.3 Parallel Performance . . . 118
8.3 Modelling and Solution Time . . . 120
8.3.1 Serial Analysis . . . 120
8.3.2 Parallel Analysis . . . 121
8.3.3 Discussion . . . 122
8.4 Discussion . . . 122

9 Conclusions 124
9.1 Research Summary . . . 124
9.2 Future Work . . . 125

Appendices 127

A `AutoDiff_Library` Interface Methods 128

B Security Constrained Optimal Power Flow Model 131

C Asset Liability Management Model with Mean-Variance 138

List of Figures

2.1 The constraint matrix structure of MSND problem. 21

2.2 Scenario tree structure for a 3-stage ALM-SSD problem instance constructed from data file in Listing 2.4 (using parameters: `Parent` and `Probs`, and `NODES` set). The numbers on the tree edges are the probability of reaching each node from its parent node. 26

2.3 The constraint matrix structure of a 3-stage ALM-SSD problem instance based on the data file in Listing 2.4. 27

3.1 The augmented matrix structure for a two level problem. 33

3.2 A general problem formulation using Bender's decomposition scheme. 36

3.3 Benders' Decomposition solvers algorithm at each iteration k. 37

4.1 The template model tree for MSND problem model file specified in Listing 2.1. 44

4.2 Entities in each template model tree node of MSND problem. 45

4.3 The template model tree for a 3-stage ALM-SSD problem specified in Listing 2.3. The indexing expression associated with each tree node is implicitly constructed by PSMG. 46

4.4 Entities in each template model tree node of ALM-SSD problem. 47

4.5 The expanded model tree of an MSND problem instance constructed by PSMG using data file in Listing 2.2. 48

4.6 Expanded model tree for a 3-stage ALM-SSD problem instance constructed by PSMG using data file in Listing 2.4. 48

4.7 The PSMG workflow diagram with a parallel optimization solver. 50

5.1 The computational graph for function $f(x) = (x_{-1}x_{-2}) + \sin(x_{-2})x_0$. 55

5.2 Illustration for applying forward AD to evaluate $\frac{\partial f}{\partial x_{-1}}$ of $f(x) = (x_1x_{-2}) + \sin(x_{-2})x_0$. 56

5.3 Illustrating the forward sweep for applying reverse AD to evaluate $\nabla f(x)$ of $f(x) = (x_{-1}x_{-2}) + \sin(x_{-2})x_0$. 58

5.4 Illustrating the reverse sweep for applying reverse AD to evaluate $\nabla f(x)$ of $f(x) = (x_{-1}x_{-2}) + \sin(x_{-2})x_0$. 59

5.5 The class hierarchy diagram in AutoDiff Library for building function expression tree. 62

6.1 A general two level structured problem. 65

6.2 Double bordered block-angular structure for a two level structured problem 66

6.3 A general two level structured problem. 72
6.4 Structure of Jacobian matrix for a general two LP problem. 73
6.5 The objective gradient for a general two level LP problem. 77
6.6 A general two level structured problem. 79
6.7 The objective Hessian matrix structure for a general two level QP problem. 79
6.8 The objective gradient for a general two level QP problem. 81
6.9 Structure of Jacobian matrix for a general two level NLP problem. . . 84
6.10 Structure of Hessian of the Lagrangian matrix for a general two level NLP problem. 90

7.1 A potential matrix data type hierarchy diagram adopted by solver implementation. 100
7.2 The double bordered angular-block structure matrix created using an expanded model tree of a two level problem. Each nonzero block of the matrix is indicated by a pair of expanded model tree nodes. 101
7.3 A potential vector data type hierarchy diagram adopted by solver's implementation. 103
7.4 This figure demonstrates the expanded model tree and its corresponding vector structure. The dimension of each sub-vector can be obtained by accessing the interface properties of the expanded model tree node. . . 104
7.5 Parallel processor allocation of the rearranged augmented matrix and corresponding vector in OOPS. 105

8.1 Semi-log plot for generating MSNG problem instances using PSMG and SML-AMPL. The data is from Table 8.1. It clearly demonstrates that the performance of PSMG is far superior to SMP-AMPL. 111
8.2 Plot for PSMG's problem generation time for the MSND problems in Table 8.2. 113
8.3 Problem generation time for the ALM-SSD problems in Table 8.2 . . . 113
8.4 Problem generation time for the ALM-VAR problems in Table 8.2 . . 113
8.5 Parallel efficiency plot for MSND problem. 115
8.6 Parallel efficiency plot for ALM-SSD problem. 115
8.7 Total memory usage plot for generating problem *msnd30_100*. 116
8.8 Per processor memory usage for generating problem *msnd30_100*. . . . 116
8.9 Plot for PSMG's problem generation time for the first iteration of the SCOPF problems in Table 8.6. 119
8.10 Problem generation time for the subsequent iteration of the SCOPF problems in Table 8.6. 119
8.11 Per-iteration parallel efficiency plot for problem generation in the first iteration of the largest SCOPF problem in Table 8.6. 121
8.12 Per-iteration parallel efficiency plot for problem generation in the subsequent iterations of the largest SCOPF problem in Table 8.6. 121

List of Tables

6.1 Summary of the local and distributed interface methods for LP, QP and NLP problems. In this table, “L” donates the *local* interface method is implemented for a problem type. “D” means the *distributed* interface is implemented. “Same” means the interface methods are the same for both *local* and *distributed* implementations. This happens when the entity to be evaluated does not depend on the variable values. “×” means the evaluation routine for a problem type is not meaningful. . 71

8.1 Comparison of PSMG with SML-AMPL for generating MSND problem instances in serial execution. 111
8.2 Problem sizes and PSMG problem generation time for MSND, ALM-SSD and ALM-VAR problem instances. 112
8.3 PSMG speedup and parallel efficiency for a large MSND problem. . . 114
8.4 PSMG speedup and parallel efficiency for the largest ALM-SSD problem. 115
8.5 Parallel processes memory usage information for generating problem `msnd30_100`. 116
8.6 Problem sizes and PSMG problem generation time for SCOPF problem instances. 119
8.7 PSMG speedup and parallel efficiency for largest SCOPF problem in Table 8.6. 120
8.8 PSMG's modelling time for a set of MSNG problems and their corresponding solving time. 121
8.9 PSMG's modelling time the MSND problem. 122

Chapter 1

Introduction

1.1 Background of Algebraic Modelling Language

Mathematical programming techniques are widely used in various fields. Let us consider a mathematical programming problem in the general form,

$$\min_{x \in X} f(x) \text{ s.t. } g(x) \leq 0, \tag{1.1}$$

where $x \in \mathbb{R}^n$, $f : \mathbb{R}^n \to \mathbb{R}$, $g : \mathbb{R}^n$ and f, g are sufficiently smooth. Here x is the vector of decision variables, $f(x)$ is the objective function and $g(x)$ are the constraint functions. Most optimization solvers are implemented with an iterative algorithm. At each iteration, all or some of the values $f(x)$,$g(x)$,$\bigtriangledown f(x)$,$\bigtriangledown g(x)$,$\bigtriangledown^2 f(x)$, and $\bigtriangledown^2 g_i(x)$ are required at the current iterate $x \in X$.

To avoid hardcoding of problem specific routines for function and derivative evaluation, in most cases a problem is modelled using an algebraic modelling language(AML) of the modeller's choice. A few survey papers [4, 5] have studied the advantages of using an AML to describe an optimization problem over other methods (such as MPS form [6, 7] or *matrix generator form*). Most AMLs use common algebraic abstraction (such as *index sets, parameters, variable, etc.*) to describe an optimization problem allowing to produce a model with good understandability. AMLs usually also offer the feature of *separation of model and data* which can effectively improve the maintainability of a model. AMPL [3], GAMS [8], AIMMS [9] and Xpress-Mosel [10] are popular choices of commercial modelling languages. GLPK is an open source counterpart which is offering a subset of the AMPL language but is restricted to modelling linear problems, or mixed integer problems [11]. By using an AML, a modeller is able to focus on specifying the underlining mathematical relations of the problem using common algebraic abstraction from the AML. Otherwise, the modeller is most likely required to write matrix generator programs to interactive with an optimization solver, which is usually a tedious and error-prone process. Therefore, AML is a great bridge between the optimization solver and the modeller.

Few recently developed AMLs, such as FlopC++ [12], Pyomo and JuMP [13] are using operator overloading techniques and are built on top of a general purpose programming languages: C++, Python and Julia respectively. They allow the optimization model to be specified in a declarative style in a similar way as in AMPL or GAMS. Since FlopC++ is built on top of C++, it requires explicit compilation to execute code. Pyomo, on the other hand, is developed on top of an interpretive language–Python [14]. Arguably, the restriction arising from building a modelling language on top of a general purpose Object-Oriented language make it more cumbersome to use. According to the author of Pyomo, "Pyomo has a more verbose and complex syntax" [15] than AMLs like AMPL. On the other hand, the use of a general purpose programming language means that it is easy to embed optimization models in software application [12].

1.2 Structured Modelling Approaches Review

As the need for accurate modelling increases, the size and complexity of mathematical models increases likewise. The underlying mathematical model of a particular problem may composed of multiple sub-models at various scale levels. In order to produce an accurate and efficient model behavior, multiscale modelling technique are used in various physics, chemistry, material science applications *etc.* Multiscale modelling is refered to a modelling style that is used to express a system with simultaneously models at different scales. Weinan stated that our society is organized in a hierarchical structure, and the macroscale model behavior is produced by the microscale models and their structure and dynamics [16]. In [16], Weinan discussed various multiscale modelling techniques and their corresponding applications.

In the field of mathematical programming, the multiscale modelling is usually used for solving multiscale decision-making problems [17]. The multi-level model are usually result of the underlying organization's hierarchical structure or multi-stage stochastic process. It is common that a large scale optimization problem is modelled with millions of constraints and variables. Because of the hierarchical nature of the real world, a large scale problem is usually not only sparse but also *structured*. Here *structure* means that there exists a discernible pattern in the constraint matrix. This pattern is usually the result of an underlying problem generation process, such as discretizations in space, time or probability space; many real world complex optimization problems are composed of multiple similar sub-problems and relations among the sub-problems, where those sub-problems can be considered as microscale models. Our research goal is to investigate a convenient modelling approach for describing large scale structured decision-making problems that can be efficiently linked with various parallelized optimization solvers.

For many years, researchers have been working on using such problem structure to parallelize optimization solvers in order to reduce their solution finding time. Algorithms, such as Dantzig-Wolfe [18, 19] and Benders' Decomposition [20], and interior point solvers, such as OOPS [21, 22, 23] and PIPS [24] can take advantage of such structure to speed up the solver by parallelization and enable the solution of larger problems. To use such techniques the solver needs to be aware of the problem's constraint matrix and structure. However the AMLs we mentioned above do not have the capabilities to express the problem's structure and convey it to the solver.

The modelling language SAMPL [25] is a stochastic extension to AMPL [3] using AMPL's suffix declaration. StAMPL [26] is another modelling tool for describing stochastic programming problems. It can specify the scenario tree structure as data and allows separation of the stochastic tree topology from the abstract mathematical relation between stages. There are also other approaches that are able to convey the problem structure to the optimization solver. The mechanism is to pass structure information by annotation of the unstructured model. This can be done by annotation on top of MPS format [27]. There are attempts made for automatic recovery of the problem's matrix structure from the unstructured model [28], however this process is computationally intensive and produces far from perfect results.

Colombo et al. have presented a more detailed review of these approaches in [1]. Those modelling system that do offer capabilities to express structure, are either not general approaches (*e.g.* specialized to stochastic programming), or they require assembling the complete unstructured model before annotations can be parsed, which is usually infeasible for large problems because of memory limitation of a single node.

On the other hand, SML [1] is designed as a generic AML for describing any structured problem. SML extends the modelling syntax from AMPL by introducing additional high level abstraction such as *block, stage*, etc., therefore a concise and straightforward model can be produced for a structured problem.

1.3 Motivation of Parallelization

The total time required to solve an optimization problem is the combination of time consumed for problem generation and function evaluations in the AML plus the time consumed for the optimization solver. Although the former is often a comparatively small part of the overall process, for a large scale optimization problem with millions of variables and constraints, the model generation process itself becomes a significant bottleneck for both memory and execution speed, especially when the optimization solver is parallelised, while the model generator is not. Therefore parallelization, not only of the solver, but also of the problem generation and function evaluation is necessary. The need for parallel model generation has also been recognised by the European Exascale Software Initiative EESI [29]. It is worth mentioning Huchette et al. who have recently presented StochJuMP [30]–a parallel extension for the JuMP. However StochJuMP is only for modelling stochastic programming problems, and not yet supporting the *separation of model and data.* In our research, we have studied the challenges and difficulties when performing a parallel problem generation for structured problems. The result of this work has produced a generic parallel algebraic modelling tool, namely Parallel Structured Model Generator (PSMG). PSMG adopts the proven modelling syntax from SML for describing structured problems. PSMG also provides a generic solver interface implementation that is sufficient to be adapted by different structure exploiting solvers. Meanwhile, the initial proof-of-concept generator for SML was implemented as an AMPL pre- and post-processor. It uses the AmplSolver [31] library to compute the function and derivative evaluations through `*.nl`-file, which means it uses a file system based interface for communicating between the solver and AMPL. The file-system based interface has been proven to be inefficient for parallel processing;on the other hand, PSMG is a standalone algebraic modelling generator and does not depend on AMPL. The elimination of file system interface has resulted in huge performance gains in PSMG for generating large structured problems. As a result, PSMG's model parser requires additional implementation work for handling the full semantics of the modelling syntax. A data parser is also implemented for PSMG to load the corresponding data files for models. The usage of the modelling syntax of SML is also documented in [2]. In other words, a major percentage of work is spent in software design and implementation to make PSMG released as an open-sourced modelling tool. PSMG was firstly presented in [32] where PSMG's high-level design concept and performance for modelling linear programming problems were demonstrated. In this thesis, we provide in-depth details about PSMG design and its solver interface, and we also present the extension work that allows PSMG to model nonlinear programming problems.

1.4 PSMG overview

From the modeller's point of view, the problem is described in a model and data file. PSMG builds the *template model tree* and *expanded model tree* from the files (discussed in Chapter 4). The template model tree represents the problem's high level abstraction specified in the model file, whereas the expanded model tree represents an actual problem instance by *expanding* the template model tree with the data file. The expanded model tree is generate in two stages: model processing stage and problem generation stage. PSMG firstly generates the skeleton of expanded model tree with minimal processing effort. This expanded model tree skeleton contains just enough information (*i.e.* number of constraints and variables in each sub-problems) for PSMG to convey the problem structure to the parallel solver processes. Then at the problem generation stage, PSMG computes and populates the part of the content of the expanded model tree that is necessary for function and derivative evaluations on a local process. The *expanded model tree* also acts as the solver interface between a structure exploiting solver and PSMG (discussed in Chapter 6).

The template model tree and expanded model tree skeleton are built on every parallel processes. PSMG employs a solver driven work assignment approach for parallel problem allocation (discussed in Chapter 4). This design allows PSMG to be easily adopted by various parallel solvers with different inter-process communication patterns.

1.5 Structure of this Thesis

The rest of this thesis is organized as follows: Chapter 2 presents the structured problems and their modelling technique adopted in PSMG. In this chapter, we present an overview of PSMG's model and data syntax by demonstrating the model and data files for two structured problem examples. PSMG's modelling syntax is mostly adapted from the SML but the parser has been re-implemented for better performance. A data parser is also implemented for PSMG to handle a subset of AMPL's data syntax. Chapter 3 briefly reviews two types of parallel solving approaches (*i.e.* using structure exploiting interior point method and decomposition algorithms) in order to identify the required tasks to be performed by a parallel modelling system. Chapter 4 explains the model processing of PSMG in detail. We describe important design and implementation decisions in PSMG, as well as some major components of PSMG. PSMG implements the state-of-the-art auto differentiation(AD) algorithms for computing Jacobian and Hessian matrices and their corresponding sparsity patterns. Thus, we also explain the AD algorithms and their implementation in Chapter 5. Chapter 6 presents PSMG's solver interface for LP, QP and NLP problems. The interface methods for problem structure retrieval and function and derivative evaluations are demonstrated. Chapter 7 explains the work involved in linking PSMG's solver interface with optimization solvers. Chapter 8 presents PSMG's performance benchmark results for both serial and parallel executions. The run-time speed and memory usage results are provided and discussed for several structured problems. Finally, in Chapter 9 we present the conclusions of this thesis and a list of possible future work that may be continued in this research area.

Chapter 2

Structured Problems and Modelling Techniques

In this chapter, we first briefly review the modelling syntax of SML, and then we present two nested-structured problems modelled in SML. The examples we used are Multi-Commodity Survivable Network Flow (MSND) and Asset Liability Management with Second Order Stochastic Dominance constraints (ALM-SSD). This chapter also serves the purpose to showcase how PSMG modelling syntax is used to express problem structure. The examples we presented in this chapter are also used to evaluate PSMG's performance in Chapter 8.

2.1 SML Modelling Syntax Review

SML modelling syntax is documented in the SML User's Guide [2] in detail. For sake of completeness of this thesis, we also briefly illustrate some key concepts in its modelling syntax.

2.1.1 SML Keywords

SML's modelling syntax is build on top of AMPL's with added feature to describe block structures in a problem. The modeller can still use normal statements in AMPL's format to declare a set, parameter, constraint, objective, *etc.* In addition to the keywords used in AMPL, other keywords are also reversed in SML for describing the structure of the problem. Those keywords are listed below.

- `block` – declares a block entity.
- `stochastic` – modifier of a block entity that indicates it is a stochastic block.
- `using` – modifier of a stochastic block statement that specifies the sets used to construct the scenario tree of this stochastic process.
- `stage` – declares a stage entity.
- `ancestor` – ancestor function, *e.g.* `ancestor(i)` references i-th ancestor stage.

- `deterministic` – indicates the entities defined within the stochastic block are repeated only once for each stage.

- `Exp` – expectation function, *e.g.* `Exp(expr)` represents the expectation of the expression expressed by `expr`.

2.1.2 Block Model

The syntax for specifying a block model is given below.

```
block_statement:
    block name_of_block [indexing_set]: {
        [statements]
    }
```

The `statements` can be any of normal AMPL statements (for describing an entities in this block such as `set, param, subject to, var, minimize, etc.`) or further `block_statement` (for describing a problem with nested structure). The expression inside the square brackets `[]` is optional. The square brackets itself are not a part of the syntax. It is obvious that the block statement can be conveniently used to describe similar block structure of a problem that is repeated over an indexing set expression. Between the curly brackets {}, it define the scope of this block statements, which is similar to the scope in many programming language such as C, JAVA, *etc.* The entities declared inside the scope can be implicitly referenced within the same scope. On the other hand, entities declared outside the scope should be explicitly referenced.

Indexing Set

The grammar for an indexing set is very similar to the one used for AMPL. The indexing set is used to define entities such as set, parameter, variable, constraint, block, *etc.* The indexing set syntax is crucial for using SML to describe similar blocks. The grammar for declaring an index set is given below.

```
indexing_set:
    {[j in] set_expression}
```

An optional dummy variable `j` can be declared and used inside the `set_expression` declaration or the entity that is declared with this indexing set. If this indexing set is used for defining a block model, the dummy variable can also be referenced in the scope of the block statements.

2.1.3 Stochastic Model

The stochastic model offers a convenient syntax for modelling stochastic programming problems. The syntax for specifying a stochastic block model is given below.

```
stochastic_block:
```

```
block name_of_block stochastic using([i in] node_set,
prob_param, parent_param, [j in] stage_set): {
    [stochastic statements]
}
```

The stochastic block requires a few sets and parameters to be specified inside the brackets `()`. `node_set` and `parent_param` contains node names and their corresponding parent relationship to form the scenario tree. `prob_param` gives the probability on the scenario tree branch. `stage_set` provides the full stage set names of this scenario tree. The `stochastic statements` can be any normal AMPL statements, `block_statement` or `stage_statement`. By default, the entities declared in the scope of the `stochastic_block` are repeated for every stage and node on the scenario tree, unless the entity declaration statement is further restricted by a `stage_statement`.

The `stage_statement` offers a convenient feature for declaring entities only present in a subset of *stage_set*. The syntax for the `stage_statement` is given below.

```
stage_statement
    stages set_expression:{
        [statements]
    }
```

The entities declared in the scope of the `stage_statement` are repeated for every node in the stage set represented by the `set_expression`.

In the stochastic block, the variable declaration statement can be further restricted by the `deterministic` keywords to make it only repeat once for each stage.

2.1.4 Explicit Variable Referencing

To reference a variable outside of the current scope in a block statement, the `dot` notation that is similar to that found in an Object-Oriented programming language (*eg.* Java) is introduced. For example, *name_of_block*`[i].`*name_of_var* references the variable *name_of_var* declared in *name_of_block* block indexed by `i`. We will demonstrate the usage of the `dot` notation in the MSND problem in Section 2.2.

In stochastic statement, ancestor stages can be referenced by `ancestor(i)` function where `i` is a positive integer value. Combining the `ancestor(i)` with `dot` notation, a variable from ancestor stages can be referenced as `ancestor(i).`*`name_of_var`*. The usage of the `ancestor` function is demonstrated in AML-SSD problem in section 2.3.

2.2 MSND: A Nested-block Structured Problem

In the MSND problem, the objective is to install additional capacity on the edges of a transportation network so that several commodities can be routed simultaneously without exceeding link capacities even when one of the links or nodes should fail. The mathematical formulation for this problem is given below in (2.1). The sets, parameters and decision variables used in this model formulation are explained below.

Sets

- $\mathcal{N}$ represents the node set of the network.
- $\mathcal{E}$ represents the edge set of the network.
- $\mathcal{C}$ represents the set of commodities.

Parameters

- $b_k, k \in \mathcal{C}$ is the demand vector for k-th commodity.
- $C_l, l \in \mathcal{E}$ is the base capacity for l-th edge.
- $c_j, j \in \mathcal{E}$ is the per unit cost of installing additional capacity on j-th edge.

Variables

- $x_k^{(n,i)}, k \in \mathcal{C}, i \in \mathcal{N}$ is the flow vector of k-th commodity on the reduced network after removing the i-th node from the full network.
- $x_k^{(e,j)}, k \in \mathcal{C}, j \in \mathcal{E}$ analogous to above, represents the flow vector after removing the j-th edge from the full network.
- $s_j, j \in \mathcal{E}$ is the additional capacity to be installed on edge j.

We use $A^{(n,i)}$ and $A^{(e,j)}$ to represent the node-edge incident matrix for the reduced network after removing the i-th node and j-th edge respectively.

$$
\begin{aligned}
\min \sum_{l \in \mathcal{E}} & c_l s_l && && \text{(2.1a)}\\
\text{s.t.}\quad & A^{(n,i)} x_k^{(n,i)} = b_k && \forall k \in \mathcal{C}, i \in \mathcal{N} && \text{(2.1b)}\\
& A^{(e,j)} x_k^{(e,j)} = b_k && \forall k \in \mathcal{C}, j \in \mathcal{E} && \text{(2.1c)}\\
& \sum_{k \in \mathcal{C}} x_{k,l}^{(e,j)} \leq C_l + s_l && \forall j \in \mathcal{E}, l \in \mathcal{E} && \text{(2.1d)}\\
& \sum_{k \in \mathcal{C}} x_{k,l}^{(n,i)} \leq C_l + s_l && \forall i \in \mathcal{N}, l \in \mathcal{E} && \text{(2.1e)}\\
& x \geq 0, s_j \geq 0 \;\; \forall j \in \mathcal{E}. && && \text{(2.1f)}
\end{aligned}
$$

The constraints (2.1b) and (2.1c) are flow balance constraints for each node or edge failure scenario respectively. These two sets of constraints ensure demands are satisfied for each commodity in the reduced network. The constraints (2.1d) and (2.1e) are edge capacity constraints, which ensure the flow passing through each edge does not exceed the capacity limit of the edge.

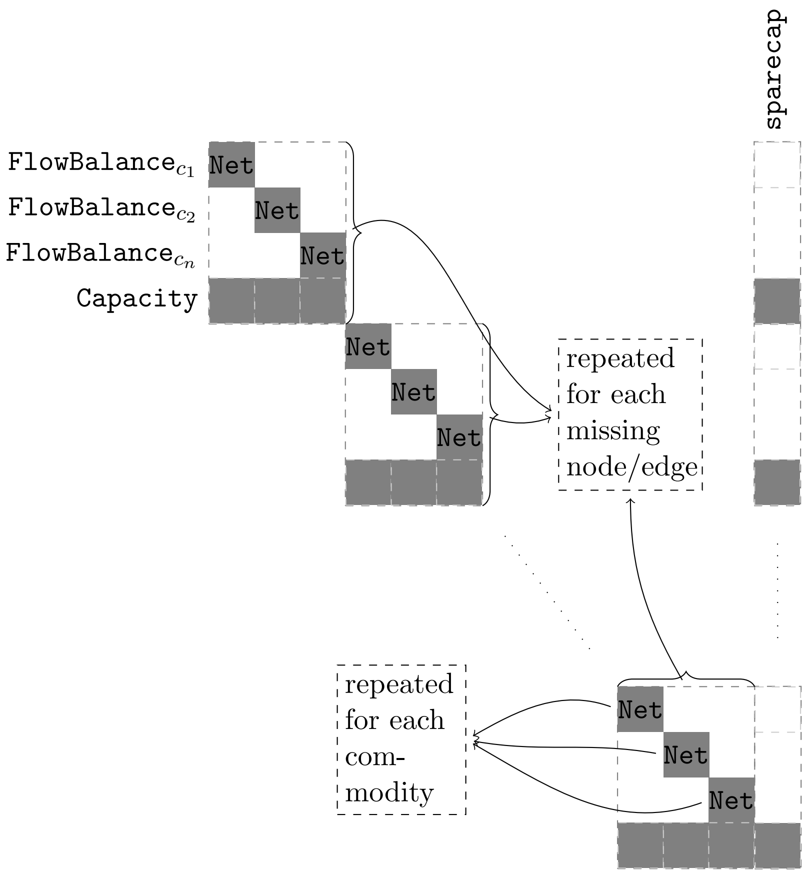


Figure 2.1: The constraint matrix structure of MSND problem.

The PSMG model for this problem is given in Listing 2.1. This model uses `block_statements` to describe repeated common structures in the problem, and then the full problem is build using nested block statements of two layers. Firstly the problem can be constructed by modelling a basic building block, namely the node-edge incident matrix formed by the flow balance constraint. Secondly, we can repeat these blocks for all commodities. Finally we can repeat the nested block again for each missing edge and node case to build the full problem. The flow balance constraints in lines 11–13 and 24–26 describe a node-edge incidence matrix for a network with a missing edge or node respectively. The `block_statements` at lines 9 and 22 repeat this structure for each commodity, while the `block_statements` at lines 7 and 19 repeat for each missing edge or node, creating a nesting of sub-problem blocks. In addition, we have `Capacity` constraints (at lines 14 and 25) to model the edge capacity for routing multiple commodities. These are also the linking constraints for `Net`-blocks.

Based on the MSND problem model file, the structure of the problem can be summarised as following. In the top-level master problem, two sub-problems are declared which corresponds to the `MCNFArcs` and `MCNFNodes` block in the model file. We call them the level-1 sub-problems in this structured problem (assuming master problem is the level-0 problem). In each of the level-1 sub-problems, a level-2 sub-problem is also declared which corresponds to the `Net`-blocks. The actual constraint matrix for a problem instance can be realised when providing a corresponding data file. A sample data file with 3 nodes and 3 arcs is given in Listing 2.2.

The constraint matrix of this problem instance is given in Figure 2.1. As it can be seen the problem displays a nested block structure that is expanded from the problem's declaration, a fact that is not immediately obvious from the mathematical description, but can be easily observed from the model file in PSMG's format.

```
1   set NODES, ARCS, COMM;
2   param cost{ARCS}, basecap{ARCS}, arc_source{ARCS}, arc_target{ARCS};
3   param comm_source{COMM}, comm_target{COMM}, comm_demand{COMM};
4   param b{k in COMM, i in NODES} := if(comm_source[k]==i) then comm_demand[k] else
         if(comm_target[k]==i) then -comm_demand[k] else 0;
5   var sparecap{ARCS}>=0;
6
7   block MCNFArcs{a in ARCS}: {    #declaration of MCNFArcs block
8     set ARCSDIFF := ARCS diff {a};
9     block Net{k in COMM}: {       #declaration of Net block inside MCNFArcs
10      var Flow{REMARCS}>=0;
11      subject to FlowBalance{i in NODES}:
12        sum{j in REMARCS:arc_target[j]==i} Flow[j]
13        - sum{j in REMARCS:arc_source[j]==i} Flow[j] = b[k,i];
14    }
15    subject to Capacity{j in REMARCS}:
16      sum{k in COMM} Net[k].Flow[j] <= basecap[j] + sparecap[j];
17  }
18
19  block MCNFNodes{n in NODES}: {  #declaration of MCNFNodes block
20    set REMNODES := NODES diff {n};
21    set REMARCS := {m in ARCS:arc_source[m]!=n and arc_target[m]!=n};
22    block Net{k in COMM}: {       #declaration of Net block inside MCNFNodes
23      var Flow{ARCS} >= 0;
24      subject to FlowBlance{i in REMNODES}:
25        sum{j in REMARCS:arc_target[j]==i} Flow[j]
26        - sum{j in REMARCS:arc_source[j]==i} Flow[j] = b[k,i];
27    }
```

```
  subject to Capacity{j in REMARCS}:
  sum{k in COMM} Net[k].Flow[j] <= basecap[j] + sparecap[j];
}
minimize costToInstall: sum{x in ARCS} sparecap[x]*cost[x];
```

Listing 2.1: Model file for MSND problem.

```
set NODES := N1 N2 N3 ;
set ARCS := A1 A2 A3 ;
param: cost basecap arc_source arc_target :=
  A1 1 4 N1 N2
  A2 4 7 N1 N3
  A3 5 6 N2 N3;
set COMM := C1 C2 C3 ;
param: comm_source comm_target comm_demand :=
  C1 N1 N2 0
  C2 N3 N2 3
  C3 N2 N3 0;
```

Listing 2.2: Sample data file for MSND problem.

2.3 ALM-SSD: A Multistage Stochastic Programming Problem

PSMG can also model a stochastic programming problem conveniently using the `stochastic_statement` from SML. Here we present an example of a stochastic programming problem, namely an Asset & Liability Management problem with second order stochastic dominance constraints (ALM-SSD) [33].

In the multi-stage ALM-SSD problem, the goal is to find an optimal portfolio strategy that is no worse than a specified benchmark by second order stochastic dominance [34]. The benchmark is typically the performance of a market index or a competitor's portfolio.

Let a scenario tree be given by a set of nodes $\mathcal{L}$ with probabilities p_i of reaching node $i \in \mathcal{L}$. We also denote all nodes in stage t by $\mathcal{L}_t$ and the parent node of node i by $\pi(i)$. Let $\mathcal{A}$ be a set of assets to invest in. Let V_j be the initial price of asset j and $r_{i,j}$ be the rate of return if asset j is held in node i. We seek to invest an initial capital I. On each node $i \in \mathcal{L}$, we define variables $x^h_{i,j}, x^s_{i,j}, x^b_{i,j}$ to represent the amount held, sold and bought of asset j respectively and c_i to represent the amount of cash held. Buying and selling incurs proportional transaction costs of γ.

Second order stochastic dominance is a constraint controlling the risk the investor allowed to take. It can be formulated by saying that the expected shortfall of the optimized portfolio at various *shortfall levels* $N_l, l \in \mathcal{F}$, should be less than the expected shortfall at the same levels for a selection of benchmark portfolios $b \in \mathcal{B}$. The benchmark portfolios are typically market indices or competitors portfolios. Let $M_{b,l}$ the expected shortfall of benchmark portfolio $b \in \mathcal{B}$ with respect to N_l.

A mathematical description of the ALM-SSD model is given below:

$$\max \quad \sum_{i\in\mathcal{L}_T} p_i(\sum_{j\in\mathcal{A}} V_j x^h_{i,j} + c_i) \tag{2.2a}$$

$$\text{s.t.} \quad (1+\gamma)\sum_{j\in\mathcal{A}}(x^h_{0,j}V_j) + c_0 = I, \tag{2.2b}$$

$$c_{\pi(i)} + (1-\gamma)\sum_{j\in\mathcal{A}}(V_j x^s_{i,j}) = c_i + (1+\gamma)\sum_{j\in\mathcal{A}}(x^h_{i,j}), \quad \forall i \neq 0, i \in \mathcal{N} \tag{2.2c}$$

$$(1+r_{i,j})x^h_{\pi(i),j} + x^b_{i,j} - x^s_{i,j} = x^h_{i,j}, \qquad \forall i \neq 0, i \in \mathcal{L}, j \in \mathcal{A} \tag{2.2d}$$

$$\sum_{j\in\mathcal{A}}(V_j x^h_{i,j}) + c_i + b_{i,l} \geq N_l, \qquad \forall i \neq 0, i \in \mathcal{L}, l \in \mathcal{F} \tag{2.2e}$$

$$\sum_{i\in\mathcal{L}_s} p_i b_{i,l} \leq M_{l,b}, \qquad \forall b \in \mathcal{B}, l \in \mathcal{F} \tag{2.2f}$$

$$x \geq 0,\ b_{i,l} \geq 0, \qquad \forall l \in \mathcal{F}, \forall i \neq 0, i \in \mathcal{L} \tag{2.2g}$$

We can contrast the mathematical description with the model file in SML (Listing 2.3). SML provides a `stochastic_block` (line 15) to describe a model block repeated for every node of a scenario tree. The arguments in bracket `()` after the keyword `using` are used to describe the scenario tree of a stochastic programming problem. Furthermore, `stage_statement` can be used within the `stochastic_block` for declaring entities that repeated only for certain stage sets. Using this modelling syntax for stochastic programming problems allows the modeller to separate the abstract mathematical relation among stages from the shape of the scenario trees. Without modifying the model file, a stochastic programing problem can be solved according to different scenario trees from various scenario generation processes.

Constraint (2.2b) in the above formulation is the start-budget constraint and only appears in root stage (line 20 in Listing 2.3). Constraints (2.2c) and (2.2d), lines 28–30 in the model file, are cash-balance and inventory constraints respectively. They are repeated in every node except the root stage node. The last two constraints (2.2e) and (2.2f) are the linear formulation of second-order stochastic dominance (line 33 and 36 in Listing 2.3). Note that the SSD formulation introduces constraints that link all nodes of the same stage. While this is uncommon for stochastic programming problem, SML provides features to model such constraints concisely (through the `Exp` construct in line 38). The `Exp(`***expr***`)` represents the expectation of *expr* which is equivalent to a weighted sum of *expr* from every nodes in a stage. Our benchmarking results given in Chapter 8 also confirm that the performance of PSMG is not harmed by the presence of expectation constraints.

```
1  param T;
2  set TIME ordered = 0..T;
3  param InitialWealth;
4  set NODES;
5  param Parent{NODES} symbolic;   # parent of nodes
6  param Probs{NODES};             # probability distribution of nodes
7  set ASSETS;
8  param Price{ASSETS};
9  param Return{ASSETS, NODES};    # returns of assets at each node
```

```
set BENCHMARK;                  # BENCHMARK = number of benchmark realization
param VBenchmk{BENCHMARK};      # values of benchmarks
param HBenchmk{BENCHMARK};      # 2nd order SD values of benchmarks - calc'ed
param Gamma;

block alm stochastic using (nd in NODES, Parent, Probs, st in TIME): {
  var x_hold{ASSETS} >= 0;
  var shortfall{BENCHMARK} >= 0;
  var cash >=0 ;
  stages {0}: {
      subject to StartBudget:
         (1+Gamma)*sum{a in ASSETS} (x_hold[a]*Price[a]) + cash  = InitialWealth ;
  }
  stages {1..T}: {
      var x_sold{ASSETS} >= 0;
      var x_bought{ASSETS} >= 0;

      subject to CashBalance:
         ancestor(1).cash - (1-Gamma)*sum{a in ASSETS} (Price[a]*x_sold[a])
         = cash + (1+Gamma)*sum{a in ASSETS} (Price[a]*x_bought[a]);
      subject to Inventory{a in ASSETS}:
         x_hold[a] - Return[a,nd] * ancestor(1).x_hold[a] - x_bought[a] + x_sold[a] = 0;

      subject to StochasticDominanceSlck{l in BENCHMARK}:
         sum{a in ASSETS}(Price[a]*x_hold[a]) + cash + shortfall[l] >= VBenchmk[l]*InitialWealth;

      subject to StochasticDominanceExp{l in BENCHMARK}:
          Exp(shortfall[l]) <= HBenchmk[l]*InitialWealth;
  }
  stages {T}:{
      var wealth >= 0;

      subject to FinalWealth:
         wealth - sum{a in ASSETS} (Price[a]*x_hold[a]) - cash = 0;
      maximize objFunc: wealth;
  }
}
```

Listing 2.3: Model file for ALM-SSD problem.

Base on the model file (Listing 2.3),the structure of the ALM-SSD problem can be summarised as following. The master problem contains the constraints and variables declared in `stage 0` which also corresponds to the root node in the scenario tree. Then the level-0 sub-problems are the stage 0 nodes in the scenario tree, and level-1 sub-problems are the stage 1 nodes, and so on. The number of stages and the scenario tree structure are determined by the data file.

```
param T := 2;
set NODES := 1 2 3 4 5 6 7 ;
param: Parent := 1 null
                 2 1
                 3 1
                 4 2
                 5 2
                 6 3
                 7 3;
param: Probs :=  1 1
                 2 0.500000000000
                 3 0.500000000000
                 4 0.500000000000
                 5 0.500000000000
                 6 0.500000000000
                 7 0.500000000000;
param Gamma  :=  0.001000000000;
param InitialWealth := 10000.000000000000;
```

```
set ASSETS :=  ACGL  ACHC ;
param: Price :=
          ACGL 59.260000000000    ACHC 47.280000000000;
param: Return :=
          ACGL 1 0                ACHC 1 0
          ACGL 2 0.982538851056   ACHC 2 1.015950659294
          ACGL 3 0.968106648667   ACHC 3 0.995135363790
          ACGL 4 0.982538851056   ACHC 4 1.015950659294
          ACGL 5 0.968106648667   ACHC 5 0.995135363790
          ACGL 6 0.982538851056   ACHC 6 1.015950659294
          ACGL 7 0.968106648667   ACHC 7 0.995135363790;

set BENCHMARK := b0 b1 ;
param: VBenchmk :=      b0 0.992647842600
                        b1 0.996635529701;
param: HBenchmk :=      b0 0.001993843550
                        b1 0.000000000000;
```

Listing 2.4: Sample data file for ALM-SSD problem.

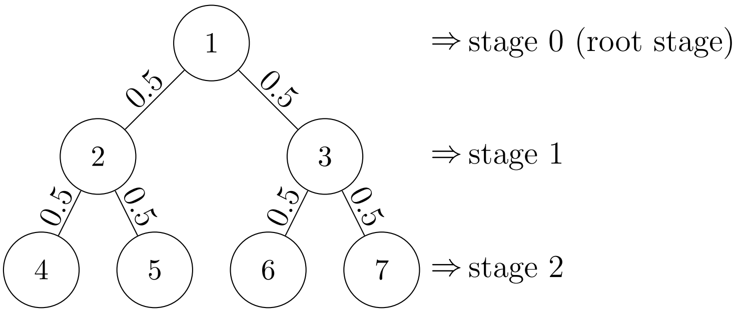


Figure 2.2: Scenario tree structure for a 3-stage ALM-SSD problem instance constructed from data file in Listing 2.4 (using parameters: `Parent` and `Probs`, and `NODES` set). The numbers on the tree edges are the probability of reaching each node from its parent node.

A sample 3-stage stochastic problem instance for the ALM-SSD model is given in Listing 2.4. In this data file, the shape of the scenario tree is specified using sets and parameters: `Node, Parent, TIME`. The probability of each tree branch is specified in parameter, `Probs`. Figure 2.2 presents the scenario tree constructed from the given data file. Those parameters are also referenced in the stochastic block declaration in Listing 2.3 (line 15).

The constraint matrix of this problem (illustrated in Figure 2.3 for the scenario tree in Figure 2.2) has the nested block-angular structure typical of multistage stochastic programming problems with additional expectation constraints.

Each sub-blocks of the constraint matrix is identified by the intersection of the constraints and variables declared at the corresponding nodes in the scenario tree. The dark grey blocks are the SSD constraints that involve taking an expectation over all nodes at the same stage in the scenario tree. In generally, the constraints having expectation expression in a stochastic programming problem are always placed in the root node of the scenario tree as linking constraints.

The light grey blocks represent the cash balance, inventory, start budget and final wealth constraints correspondingly at each node. The start budget constraint only appears in the root node (`node 1`), whereas the final wealth constraints are in the final stage nodes (`node 4, 5, 6` and `7`). Again the problem structure is not immediately obvious from its mathematical description.

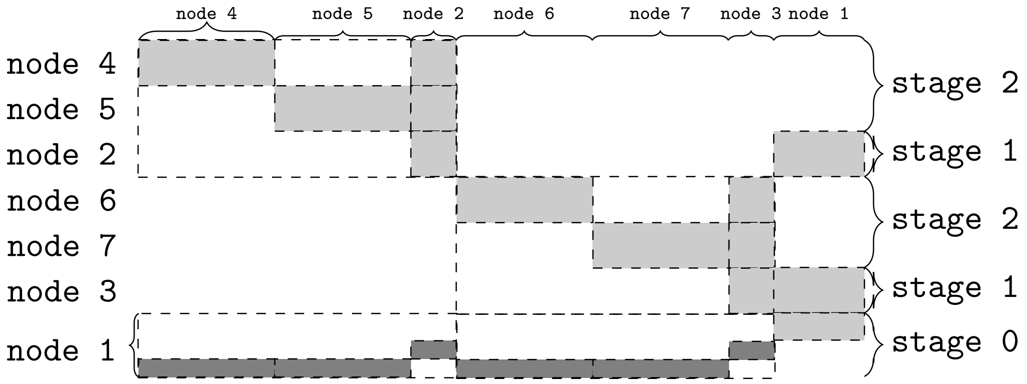


Figure 2.3: The constraint matrix structure of a 3-stage ALM-SSD problem instance based on the data file in Listing 2.4.

2.4 Discussion

2.4.1 Block Separability Assumption

There is a very important assumption that should be hold true while using PSMG as a modelling tool. We call this the *block separability assumption* which is stated below.

Block Separability Assumption The variables declared inside a sub-problem have to be additively separable (in constraint expressions) with respect to the variables declared in the sub-problems from its sibling branches.

This assumption should be taken care of when modelling linking constraints in a parent level problems. This assumption can be easily achieved by pushing non-separable variables into the same sub-problem. This assumption is always true for modelling Linear Programming (LP) problems, as all variables in LP problems are additively separable with each other. By enforcing the *block separability assumption*, the Jacobian and Hessian matrices for problems modelled in PSMG always demonstrate a double bordered block-angular structure (possibly nested) which can be exploited by parallel solvers. This assumption also helps PSMG to achieve an efficient parallel problem generation design.

2.4.2 Objective Function Treatment

PSMG also implements the same semantics for objective function modelling as in SML. SML's modelling syntax allows a single objective function to be modelled in its separable structure in every block. Therefore, it is required for the modeller to use either `maximize` or `minimize` consistently to model each objective function part in every block (or to leave absent if the block does not contribute to the objective function). Furthermore, an objective function can not associate with an indexing set expression in its declaration. The objective function expression declared in `stochastic` block is always considered by taking its expectation. For example, in the converted `block` model of a 3-stage ALM-SSD problem in Listing

4.1, the objective function is multiplied by the product of `Prob[n1]` (probability of reach from the node in `stage0` to `stage1`) and `Prob[n2]` (probability of reach from the node in `stage1` to `stage2`). The final objective function expression based on the converted block model is exactly the expectation of the objective function declared in the original stochastic model in Listing 2.3 (line 44). Moreover, the objective expression modelled in a block only allow to use variables declared in this block itself or its ancestor blocks, whereas variables declared in its descendant blocks can also be referenced in the constraint declaration using the dot notation (*e.g.* line 15 and 27 of the MSND model in Listing 2.1).

Chapter 3

Parallel Solvers for Structured Problems

In this chapter, we mainly review the current parallel structure exploiting optimization solvers and algorithms for large scale structured problems.

Generally speaking, there are two categories of solver algorithms for parallel solution of large scale structured problems. One is implemented using the structure exploiting interior point method (IPM), and the other uses decomposition algorithms.

Different parallel solver algorithms may employ different allocation strategies by considering the load balancing to minimise the inter-process communication needed. By studying the parallel algorithms in this chapter, it provides enough background for explaining the design and implementation decisions of PSMG in the later chapter.

3.1 Structure Exploiting in Interior Point Method

The interior point method offers poly nominal run-time complexity to solve large scale problems efficiently. Its parallel implementation by exploiting block structure of the problem matrix has been proven successful for solving large scale problem with 10^9 variables [23]. Some known parallel solvers in this category are OOPS [21] and PIPS [24].

IPM can be used to solve LP, QP and NLP problems. Now, we should review the basic linear algebra in IPM.

3.1.1 Linear Algebra in IPM

LP and QP problem

Given the LP (3.1) and QP (3.2) problem formulation below.

$$\begin{aligned} \text{(LP)}\min\ & c^T x \\ \text{s.t.}\ & Ax = b \\ & x \geq 0 \end{aligned} \tag{3.1}$$

$$
\begin{aligned}
\text{(QP)}\min \;\; & c^T x + \frac{1}{2} x^T Q x \\
\text{s.t.} \;\; & Ax = b \\
& x \geq 0
\end{aligned}
\tag{3.2}
$$

where $A \in \mathbb{R}^{m \times n}, Q \in \mathbb{R}^{n \times n}, c \in \mathbb{R}^n, and\ b \in \mathbb{R}^n$. Because of the similarity between QP and LP problem formulation, the results for LP problem can be followed by setting $Q = 0$. So, let us look at the QP problem formulation in (3.2). We can use a logarithmic barrier term to replace the inequality constraint ($x \geq 0$). The result gives us the corresponding barrier problem in (3.3), where the barrier parameter $\mu > 0$.

$$
\begin{aligned}
\min \;\; & c^T x + \frac{1}{2} x^T Q x - \mu \sum_{j=1}^{n} \ln x_j \\
\text{s.t.} \;\; & Ax = b,
\end{aligned}
\tag{3.3}
$$

The Lagrangian L for the barrier problem (3.3) is given below.

$$
L(x, y, \mu) = c^T + \frac{1}{2} x^T Q x - y^T (Ax - b) - \mu \sum_{j=1}^{n} \ln x_j \tag{3.4}
$$

We can derive the first order optimality conditions below for the barrier problem.

$$
\begin{aligned}
\nabla_x L(x, y, \mu) &= c - A^T y - \mu X^{-1} e + Qx = 0 \\
\nabla_y L(x, y, \mu) &= Ax - b = 0
\end{aligned}
\tag{3.5}
$$

where $X = \text{diag}\{x_1, x_2, ..., x_n\}$ and $e = (1, \cdots, 1)$

Let $s = \mu X^{-1} e$, ($i.e.$ $XSe = \mu e$). Then the first order optimality conditions for the barrier problem are

$$
\begin{aligned}
& Ax = b \\
& A^T y + s - Qx = c, \\
& XSe = \mu e \\
& x, s \geq 0
\end{aligned}
\tag{3.6}
$$

The interior point method evaluate one Newton step of the system of equations (3.6) before updating μ. The value of the barrier parameter μ is reduced at each iteration, so that the solution of this sequence of iterates is guaranteed to converge to the optimal solution of the original problem.

The Newton steps can be derived as following. First, let us define the vector function F

$$
F(x, y, s) = \begin{bmatrix} Ax - b \\ A^T y + s - Qx - c \\ XSe - \mu e \end{bmatrix} \tag{3.7}
$$

Then

$$\nabla F(x,y,s) = \begin{bmatrix} A & 0 & 0 \\ -Q & A^T & I \\ S & 0 & X \end{bmatrix} \tag{3.8}$$

The Newton direction $(\triangle x, \triangle y, \triangle s)$ for solving the system of linear equations in (3.6) at each iteration is obtained by solving the system of linear equations in (3.9)

$$\nabla F(x,y,s) \cdot \begin{bmatrix} \triangle x \\ \triangle y \\ \triangle s \end{bmatrix} = \begin{bmatrix} b - Ax \\ c - A^T y - s + Qx \\ \mu e - XSe \end{bmatrix} \tag{3.9}$$

We can further simplify the equation system in (3.9) by eliminating the $\triangle s$ ($i.e.$ $\triangle s = -X^{-1} S \triangle x + X^{-1}(\mu e - XSe)$ to obtain the augmented system illustrated in (3.10).

$$\begin{bmatrix} -Q - \Theta^{-1} & A^T \\ A & 0 \end{bmatrix} \cdot \begin{bmatrix} \triangle x \\ \triangle y \end{bmatrix} = \begin{bmatrix} \epsilon_d - X^{-1}\epsilon_\mu \\ \epsilon_p \end{bmatrix} \tag{3.10}$$

where

$$\begin{aligned} \Theta &= XS^{-1} \\ \epsilon_p &= b - Ax \\ \epsilon_d &= c - A^T y - s + Qx, \\ \epsilon_\mu &= \mu e - XSe. \end{aligned}$$

NLP problem

Given the general NLP problem formulation below

$$\begin{aligned} \min \quad & f(x) \\ \text{s.t.} \quad & g(x) \leq 0 \end{aligned} \tag{3.11}$$

where $x \in \mathbb{R}^n$, $f : \mathbb{R}^n \to \mathbb{R}$, $g : \mathbb{R}^n$ and f, g are sufficiently smooth and twice differentiable. The inequality constraint in (3.11) can be converted into an equality by introducing a non-negative slack variable z. Therefore, we can formulate the NLP problem in an equivalent form in (3.12).

$$\begin{aligned} \min \quad & f(x) \\ \text{s.t.} \quad & g(x) + z = 0 \\ & z \geq 0 \end{aligned} \tag{3.12}$$

By adding a logarithmic term in the objective function to replace the inequality constraint ($z \geq 0$), the barrier problem can be formulated as below.

$$\begin{aligned} \min \quad & f(x) - \mu \sum_{j=1}^{m} lnz_i \\ \text{s.t.} \quad & g(x) + z = 0 \end{aligned} \tag{3.13}$$

The Lagrangian L for the barrier problem in (3.13) is given below.

$$L(x, y, z, \mu) = f(x) + y^T(g(x) + z) - \mu \sum_{j=1}^{m} lnz_i \tag{3.14}$$

Similarly, the first order optimality conditions can be derived from the following.

$$\begin{aligned} \nabla_x L(x, y, z, \mu) &\Rightarrow \nabla f(x) + \nabla g(x)^T y = 0 \\ \nabla_y L(x, y, z, \mu) &\Rightarrow g(x) + z = 0 \\ \nabla_z L(x, y, z, \mu) &\Rightarrow ZYe = \mu e \end{aligned} \tag{3.15}$$

where $Y = \text{diag}\{y_1, y_2, \cdots, y_m\}$ and $Z = \text{diag}\{z_1, z_2, \cdots, z_m\}$. IPM for NLP problems solves the system of equations (3.15) also by Newton's method. The barrier parameter μ is also gradually reduced at each each iteration, so that the solution converges to the optimal solution of the original problem.

Define vector function F as below.

$$F(x, y, z) = \begin{bmatrix} \nabla f(x) + \nabla g(x)^T y \\ g(x) + z \\ ZYe - \mu e \end{bmatrix} \tag{3.16}$$

Then

$$\nabla F(x, y, z) = \begin{bmatrix} Q(x, y) & A(x)^T & 0 \\ A(x) & 0 & I \\ 0 & Z & Y \end{bmatrix} \tag{3.17}$$

The Newton direction $(\triangle x, \triangle y, \triangle z)$ for solving the sytem of linear equations in (3.15) at each iteration is obtained by solving the system of linear equations in (3.18)

$$\nabla F(x, y, z) \cdot \begin{bmatrix} \triangle x \\ \triangle y \\ \triangle s \end{bmatrix} = \begin{bmatrix} -\nabla f(x) - A(x)^T y \\ -g(x) - z \\ \mu e - YZe \end{bmatrix} \tag{3.18}$$

where

$$\begin{aligned} Q(x, y) &= \nabla^2 f(x) + \sum_{i=1}^{m} y_i \nabla^2 g_i(x) \\ A(x) &= \nabla g(x) \in \mathbb{R}^{m \times n} \end{aligned} \tag{3.19}$$

It is worth noting that $Q(x, y)$ is also the Hessian of the Lagrangian for the NLP problem in (3.11), where y_i are the dual variables.

Similar to LP and QP cases, we can simplify the equation system in (3.18) to obtain the augmented system given in (3.20).

$$\begin{bmatrix} Q(x, y) & A(x)^T \\ A(x) & -ZY^{-1} \end{bmatrix} \cdot \begin{bmatrix} \triangle x \\ \triangle y \end{bmatrix} = \begin{bmatrix} -\nabla f(x) - A(x)^T y \\ -g(x) - \mu Y^{-1} e \end{bmatrix} \tag{3.20}$$

3.1.2 Structure Exploiting in IPM

From the above discussion, the augmented system can also be presented using a common structure for LP, QP and NLP problems as below.

$$\Phi = \begin{bmatrix} -Q - \Theta_p^{-1} & A^T \\ A & \Theta_D \end{bmatrix} \tag{3.21}$$

For LP and QP problems, $\Theta_D = 0$, and A is the constant Jacobian matrix for their linear constraint functions. Q is a constant Hessian matrix for the objective function (where $Q = 0$ for LP problem). For nonlinear problem, $\Theta_P^{-1} = 0$, and Q is the Hessian of the Lagrangian function that changes at every iteration. A also changes at every iteration for nonlinear problems. However the structure of the augmented system stays the same.

Structure exploiting IPM is able to rearrange the augmented matrix into a double bordered block-angular matrix. This requires A and Q matrices to have block-angular structures in them. PSMG, on the other hand, guarantees the block-angular structure in A and Q matrix by the *block separability assumption* in its model formulation. The block-angular structure is also crucial to enable parallel solving of the augmented system $\Phi x = b$ in IPM. Without loss of generality, let us give an augmented system matrix of a general two level problem which can be viewed in left of the Figure 3.1. The A and Q matrices both demonstrate the double bordered block-angular structure. After exploiting the structure of the augmented system, the rearranged matrix also has a double bordered block-angular structure shown on the right side in Figure 3.1.

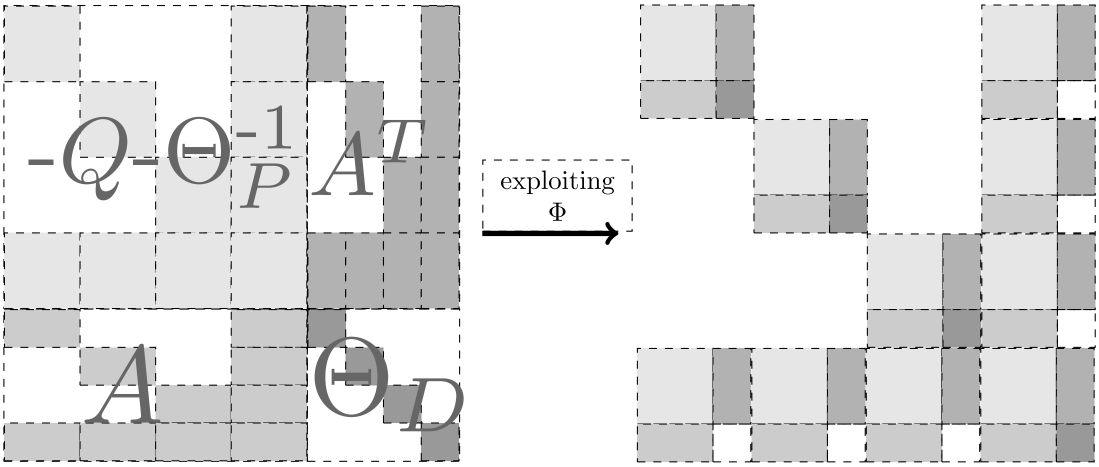


Figure 3.1: The augmented matrix structure for a two level problem.

3.1.3 Parallel Allocation and Solving

We can use similar notation from [35] to represent the augmented system matrix Φ in Figure 3.1, where $\Phi_i \in \mathcal{R}^{n_i \times n_i}, i = 0, \cdots, n$ and $B_i \in \mathcal{R}^{n_0 \times n_i}, i = 1, \cdots, n$. Then Φ has $N = \sum_{i=0}^{n} n_i$ rows and columns. Each of the Φ_i and B_i are assembled from the corresponding parts of the A and Q matrices by a structure exploiting process in section 3.1.2.

$$\Phi = \begin{bmatrix} \Phi_1 & & & & B_1^T \\ & \Phi_2 & & & B_2^T \\ & & \ddots & & \vdots \\ & & & \Phi_n & B_n^T \\ B_1 & B_2 & \dots & B_n & \Phi_0 \end{bmatrix} \tag{3.22}$$

Given two block matrices L and D with the following structure.

$$L = \begin{bmatrix} L_1 & & & & \\ & L_2 & & & \\ & & \ddots & & \\ & & & L_n & \\ L_{n,1} & L_{n,2} & \dots & L_{n,n} & L_c \end{bmatrix} \quad D = \begin{bmatrix} D_1 & & & & \\ & D_2 & & & \\ & & \ddots & & \\ & & & D_n & \\ & & & & D_c \end{bmatrix}$$

and

$$\Phi_i = L_i D_i L_i^T \tag{3.23a}$$

$$L_{n,i} = B_i L_i^{-T} D_i^{-1} \tag{3.23b}$$

$$C = \Phi_0 - \sum_{i=1}^{n} B_i \Phi_i^{-1} B_i^T \tag{3.23c}$$

$$= L_c D_c L_c^T \tag{3.23d}$$

Then Φ can be decomposed into a generalised Cholesky factorization form as $\Phi = LDL^T$.

The solution to the system $\Phi x = b$, where $x = (x_1, \cdots, x_n, x_0)^T, b = (b_1, \cdots, b_n, b_0)^T$ can be obtained by the following operations:

$$z_i = L_i^{-1} b_i, \quad i = 1, \cdots, n \tag{3.24a}$$

$$z_0 = L_c^{-1}(b_0 - \sum_{i=0}^{n} L_{n,i} z_i) \tag{3.24b}$$

$$y_i = D_i^{-1} z_i, i = 0, \cdots, n \tag{3.24c}$$

$$x_0 = L_c^{-T} y_0 \tag{3.24d}$$

$$x_i = L_i^{-T}(y_i - L_{n,i}^T x_0), \quad i = 1, \cdots, n \tag{3.24e}$$

The factorization operations in (3.23) and the subsequent solve operations in (3.24) can be performed in parallel. The execution order of the operations in (3.23) and (3.24), and also the matrix block allocation strategy on parallel processes may have a signification influence on the overall efficiency of the parallel solving of the equation system $\Phi x = b$.

In [35], Gondzio and Grothey explained an efficient parallel solving and allocation strategy that is implemented in OOPS [21]. Their parallel strategy can be seen in two stages: a parallel factorization stage (as in (3.23)); and a parallel solving stage (as in (3.24)). In their implementation, C (in (3.23c)) is computed from terms $(L_i^{-1}B_i^T)D_i^{-1}(L_i^{-1}B_i^T)$, the outer products of sparse rows of $L_i^{-1}B_i^T$. The matrices $L_{n,i}$ in (3.23b) are not required to evaluate explicitly. Later on in the solving stage, $L_{n,i}z_i$ (in (3.24b)) is calculated by $B_i(L_i^{-T}(D_i^{-1}z_i))$, and similarly $L_{n,i}^T x_0$ (in (3.24e)) is calculated by $D_i^{-1}L_i^{-1}B_i^T x_0$.

The works involved in each of the two stages are summarized as following:

- Factorization Stage

  F.1 Factorizes $\Phi_i = L_i D_i L_i^T$ and computes $C_i = B_i L_i^{-T} D_i L_i^{-1} B_i^T, \forall i = 1, \cdots n$

  F.2 Computes $C = \Phi_0 - \sum_{i=1}^n C_i$

  F.3 Factorizes $C = L_c D_c L_c^T$

- Solving Stage

  S.1 Computes $z_i = L_i^{-1} b_i$ and $l_i = B_i L_i^{-1} D_i^{-1} z_i, i = 1, \cdots n$

  S.2 Computes $l = b_0 - \sum_{i=1}^n l_i$

  S.3 Computes $z_0 = L_c^{-1} l$

  S.4 Computes $y_i = D_i^{-1} z_i, i = 1, \cdots n$

  S.5 Computes $y_0 = D_c^{-1} z_0$

  S.6 Computes $x_0 = L_c^T y_0$

  S.7 Computes $x_i = L_i^{-T}(y_i - D_i^{-1} L_i^{-1} B_i^T x_0), i = 1, \cdots n$

In above steps, **F.1**,**S.1**,**S.4** and **S.7** are the parallel steps. **F.2** is performed with a MPI all reduce operation, therefore C exists on every parallel processes. **F.3** is executed on every processes, therefore each process has a copy of L_c and D_c without inter-process communication. **S.2** is also computed using a *MPI AllReduce* operation, therefore l is available on every processes and **S.3** can be computed everywhere. y_0 and x_0 are also computed on every processes in step **S.5** and **S.6** respectively. OOPS allocates vectors x_0, y_0, z_0, l and matrices C, Φ_0 on every parallel processes to minimise inter-process communication and maximise the parallel efficiency. However, other allocation scheme is also possible as long as the required inter-process communication routine is implemented properly.

From above review of the parallel structure exploiting IPM solver, a parallel modelling system that can be linked with solver in this category should be able to assist the solver to setup problem matrix structure in blocks and allow the solver to make the allocation strategy.

3.2 Decomposition Algorithms

Another category of solvers that can also be easily made in parallel execution are those which implement decomposition algorithms, such as Dantzig-Wolfe [18,

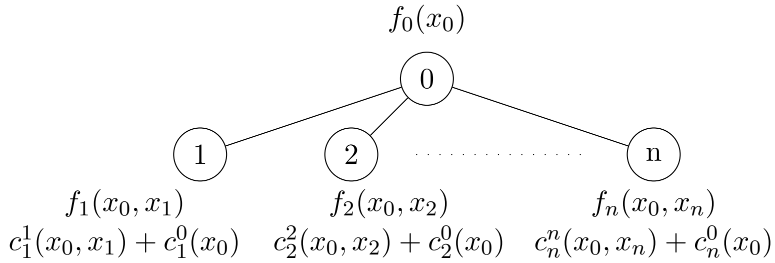

$f_0(x_0)$
0
1
2
n
$f_1(x_0, x_1)$
$c_1^1(x_0, x_1) + c_1^0(x_0)$
$f_2(x_0, x_2)$
$c_2^2(x_0, x_2) + c_2^0(x_0)$
$f_n(x_0, x_n)$
$c_n^n(x_0, x_n) + c_n^0(x_0)$

Figure 3.2: A general problem formulation using Bender's decomposition scheme.

19] or Benders' [20] Decomposition. In this section, we first review the mathematical formulation in Benders' Decomposition algorithm. It helps us to identify the routines that should be provided by the modelling language in order for a decomposition solver to setup and solve the problem in parallel.

3.2.1 Benders' Decomposition

Benders' Decomposition algorithm is firstly presented in [20] for solving mixed integer programming problems. But it is not difficult to generalise this scheme to solve NLP problems [36, 37].

Given the general problem formulation to use Benders' Decomposition as below,

$$\begin{aligned} \min_{x_i, x_0} \quad & f_0(x_0) + \sum_{i=1}^{n} f_i(x_0, x_i) \\ s.t. \quad & c_i^i(x_0, x_i) + c_i^0(x_0) \leq 0, \quad \forall i \in \{1, \cdots, n\} \end{aligned} \tag{3.25}$$

It can be easily observed that the above formulation in (3.25) is a two level general problem without the linking constraint in the master problem.

The corresponding problem structure can be presented in a tree format in Figure 3.2. x_0 can be considered as the complicating variables in the master problem linking n sub-problems.

After applying Benders' Decomposition algorithm, this problem can be formulated into two levels: a master problem (3.26); n sub-problems (3.27). $v_i(x_0)$ is the value function for each sub-problem $i \in \{1, \cdots, n\}$ at a given x_0. The objective function in the original problme (3.25) can be split into several separable parts, where each sub-problem i has an *objective function part* $f_i(x_0, x_i)$. The constraints can also be split into two additively separable parts: $c_i^i(x_0, x_i)$; and $c_i^0(x_0)$.

$$\min_{x_0} \quad f_0(x_0) + \sum_{i=1}^{n} v_i(x_0) \tag{3.26}$$

$$\begin{aligned} v_i(x_0) = \min_{x_i} \quad & f_i(x_0, x_i) \\ \text{s.t.} \quad & c_i^i(x_0, x_i) \leq \text{-}c_i^0(x_0) \end{aligned} \tag{3.27}$$

For simplicity of the discussion, let us assume $v_i(x_0)$ are convex functions and each sub-problem is feasible. The master problem (3.26) can be relaxed with a sequence of linear realization of the value function v_i^j evaluated at each x_0 from sub-problems. The corresponding formulation is given below.

$$\begin{aligned} \min_{x_0, v_i} \quad & f_0(x_0) + \sum_{i=1}^{n} v_i \\ \text{s.t.} \quad & v_i \geq v_i(x_0^{(j)}) + g_i^{(j)}(x_0 - x_0^{(j)}), \quad j \in J, \quad i \in \{1, \cdots, n\} \end{aligned} \tag{3.28}$$

where J represents the set of indices that $v_i(x_0^{(j)})$ has been already evaluated, and $g_i^{(j)}$ is a subgradient for $v_i(x_0^{(j)})$ which is also the dual variable values of each sub-problem i evaluated at optimal for a given $x_0^{(j)}$.

At each iteration k, the sub-problems (3.27) are solved to optimal at a given point $x_0^{(k)}$ of the master problem. $v_i^{(k)}$ is an underestimator for each sub-problem i, which can be used to refine the master problem (3.28) and generate an upper bound for the original problem (3.25). Then the master problem (3.28) is solved, yielding a new lower bound for the original problem. The algorithm terminates when the gap between the upper and lower bounds is less than a predefined epsilon value. This iterative solution process works at least for convex problems.

Figure 3.3 demonstrates the Bender's decomposition algorithm for solving the two level problem (in 3.28 and 3.27) at iteration k. It is noted that each of the sub-problems can be computed independently in parallel provided the value of master variable x_0 is given at each iteration.

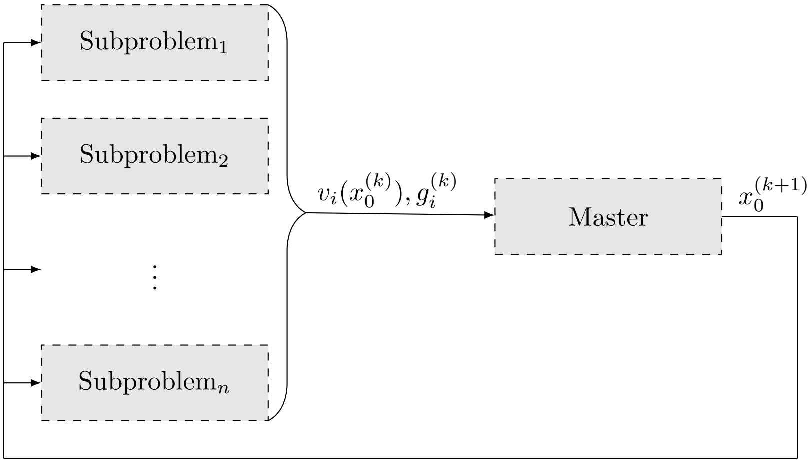


Figure 3.3: Benders' Decomposition solvers algorithm at each iteration k.

3.2.2 Parallel allocation and solving

A straightforward parallel solving strategy for a Benders' Decomposition solver could be done by solving each sub-problem on a parallel process and master problem on all the processes. This work can be summarized as following:

B.1 Initializes $x_0, UB \leftarrow \infty, LB \leftarrow$ -∞;

B.2 Solves sub-problems in 3.27;

B.3 Obtain a new upper bound UB_{new} of the master problem, set $UB \leftarrow \min\{UB_{new}, UB\}$;

B.4 Introduces new constraints in the constraint set of the relaxed master problem in 3.28;

B.5 Solves the relaxed master problem in 3.28 to obtain a lower bound, LB;

B.6 if $UB - LB \geq \epsilon$ goto **B.2**, otherwise algorithm terminates.

In order to minimize the inter-process communication, x_0 should be allocated on every parallel processes. From above list, **B.2** is the parallel step, and the resulting vectors $v_i(x_0^{(j)})$ and $g_i^{(j)}$ are needed to create new constraint in the relaxed master problem. Furthermore, the master problem can also be solved using a parallel algorithm, therefore **B.5** can be computed in parallel as well. It may require the vectors $v_i(x_0^{(j)})$ and $g_i^{(j)}$ to be communicated on parallel processes. At each iteration the solver is required to wait for all sub-problems to be solved before it can solve the new relaxed master problem, and then the solver proceeds to the next iteration if necessary (by checking the condition in **B.6**).

As we can see that a modelling system that can assist solvers to identify the problem hierarchical structure and provide function and derivative evaluations in master and sub-problem context could be very helpful for parallel solvers that implements a decomposition strategy.

3.3 Discussion

According to the discussion of two categories of parallel solver algorithms above, we can conclude the following major abilities that should be considered when designing a parallel modelling system.

- Problem Structure Setup
    - be able to identify the problem structure in block matrices.
    - be able to identify the problem hierarchical structure in master and sub-problems.
- Problem Generation
    - be able to generate each matrix blocks of A and Q in the augmented system in (3.22) on allocated processes.
    - be able to evaluate the function and derivative values of the constraints or the objective function in its additively separable parts.

In general, a parallel modelling system should be able to assist the solver for setting up the problem structure on parallel processes and generating the corresponding sub-problem matrices or the matrix blocks on the allocated process. Usually this parallel allocation strategy is decided by the solver algorithm after considering the inter-process communication cost and load balance.

3.3.1 Modelling Nonconvex Problems

When the optimization problem is formulated as a convex problem, a local optimal solution is also the global optimal solution. Therefore, only one set of possible optimal solution will be produced by the solver for a convex optimization problem. On the other hand, multiple local optimal solutions could be produced for a nonconvex optimization problem. Vanderbei *et. al.* described the modifications conducted for extending an interior point method solver–LOQO to solve a general nonconvex nonlinear programming problems [38]. Vanderbei *et. al.* also demonstrated that different initial values could produce different local optimal solution, and those initial values could be computed by the solver [38]. Therefore, our desired modelling system should be able to take initial values for both primal and dual variables in the model. It is up to the solver to track a set of local optimal solutions for nonconvex optimization problem.

MINLP models are heavily used in various engineering area, such as oil and gas field infrastructure planning [39], gas network design [40, 41], aircraft route planning[42], vehicle path planning[43], *etc.*. Usually the generalized Benders' Decomposition (GBD) method (described in Section 3.2.1) is used for solving MINLP problems in parallel. However the GBD method is not guaranteed for solution of a nonconvex problem [44]. Li *et. al.* presented a nonconvex generalized Benders' Decomposition (NGBD) method for obtaining the global optimization of a two-stage stochastic mixed-integer nonlinear programming problems (MINLPs) [44]. In order to link with optimization solver that implements NGBD method, the modelling system should also have the ability to model integer variables and passing their indexing information to the solver. This can be easily achieved by adding an modifier field for variable declaration statement to indicate the variable types, such as *integer, binary*, or *real.* The interface method for passing the integer variables index location to the solver should also be implemented.

Chapter 4

PSMG Design and Implementation

In this chapter, we firstly explain PSMG's development environment, and then we focus on discussing the important design and implementation issues of PSMG in the rest of this chapter. PSMG has a two stage design for generating a structured problem. The first stage is the structure building stage, where PSMG reads the model and data files and builds an internal skeletal representation of the information that describing the problem with minimal processing. The work in this stage happens on every parallel processes. The second stage is the parallel problem generation stage, where the problem structure is conveyed to the solver so that the solver is able to make parallel problem allocation strategy and request the specific blocks or sub-problems to be generated on the processor where they are needed. Major work in processing the problem happens at the second stage.

4.1 Development Environment and Technique

Because PSMG is a modelling tool mainly for large scale problems, performance is an important factor to be considered in PSMG's design and implementation. We also make PSMG available as an open-source software, therefore it is quite important to keep a good modular design in the PSMG's coding base. We feel the Object-Oriented implementation is the most suitable programming technique for serving this purpose. The parallelization of PSMG is achieved by using Open-MPI library–an open source Message Passing Interface implementation [45]. In addition, with the help of the Gnu's Flex [46] and Bison [47] tool and their corresponding C/C++ code generation ability, we can conveniently develop the parser for PSMG's model and data files with reasonable efforts. Therefore, we have chosen C++ as the implementation language for PSMG. Furthermore, PSMG can be compiled with either Intel or gnu compiler.

A commercial licensed modelling tool usually involves years of development efforts in code quality and runtime efficiency. Therefore it is likely to have better performance than PSMG for generating a problem in serial case. However, there is not yet any commercial modelling tool that does parallel problem generation at this time. Since PSMG is design to achieve parallel problem generation that works together with any parallel structure exploiting solvers, we believe PSMG generate the problem faster in parallel process than any other serial model generator. Because PSMG is open-sourced, it is easy for others to add new features or interface with it, whereas a proprietary modelling tool provides very limited customization ability for a user.

4.2 Block and Stochastic Model

The SML modelling syntax is specially designed for describing structured problem with blocks. In general, a structured problem can be described using either `block` or `stochastic` modelling syntax. The `block` syntax allows PSMG to declare reusable building blocks and relations between blocks to model the problem along with its structure, whereas the stochastic block provides a convenient syntax for a modeller to describe the entities in stochastic programming problems by stages. In a stochastic programming problem, nodes at the same stage usually contain similar abstraction. It is easy to see the corresponding problem structures of a `block` model. On the other hand, a `stochastic` model can always be converted to a block model after providing an actual scenario tree in the corresponding data file of the model. The conversion process requires the following major steps.

- evaluating the number of stages in the problem (using parameter values from data file if necessary);
- duplicating the stage block and its declared variables for each stage in the stage set;
- attaching an index set expression of nodes for each stage block;
- interpreting the variables with `ancestor` keyword by replacing them with the ones declared at their corresponding ancestor stage;
- translating variables declared with `deterministic` keyword based on its scope (*e.g.* either in `stochastic` or `stage` block);
- moving constraints declaration with `Exp` keyword to the root stage, and replacing `Exp(expression)` with the weighted sum of the `expression`;
- rewriting objective expression by multiplying the probability from the root stage (*i.e.* converting to the expectation of the objective expression.).

It is worth noting that this conversion process may need to access the data file to decide the number of stages in a stochastic programming problem instance. For example, to evaluate the stage set `TIME` in the ALM-SSD model (Listing 2.3), PSMG needs to access the dependent parameters from the data file (*i.e.* `T` in this case). This stochastic-to-block model conversion process is also documented in the previous SML implementation [1], but PSMG offers better run-time efficiency.

Listing 4.1 presents the `block` model for a 3-stage ALM-SSD problem as a result of performing the conversion process on the original `stochastic` model in Listing 2.3. The number of stages are evaluated using parameter values in the data file given in Listing 2.4. This process is done with a specially implemented routine inside PSMG. The modeller does not require to know the technical details behind it. In fact, this converted `block` model given below is to demonstrate the relationship between a `stochastic` and `block` model, and it is only held in the corresponding memory objects in PSMG.

From the converted `block` model (Listing 4.1), we can observe that PSMG creates dummy index variables for the index expression at each stage to represent the nodes of this stage. PSMG generates dummy variables (`n0, n1` and `n2`) for each stage block at lines 17, 31 and 47 respectively. The root node is determined by the indexing set expression, "`n0 in Child[null]`" at line 17, and this expression returns a node set that contains the root node only.

PSMG creates the compound set, `Child`, and uses it to construct the indexing set expression for each `stage` block, whereas SML uses `Parent` set for this purpose. This allows PSMG to iterate the provided `Parent` set from the data file only once and to build the `Child` set, which is used to determine the children node set for every node. Without using of the `Child` compound set, SML is required to iterate the `Parent` set at each node to build its children node set. Hence, the run-time efficiency is improved by using the temporary `Child` compound set in PSMG.

Once the stochastic model is converted into a block model, there is no difference in handling the two types of model in PSMG. This is an important fact considered in designing and implementing PSMG.

```
1  param T;
2  set TIME ordered = 0..T;
3  param InitialWealth;
4  set NODES;
5  param Parent{NODES} symbolic;    # parent of nodes
6  param Probs{NODES};              # probability distribution of nodes
7  set ASSETS;
8  param Price{ASSETS};
9  param Return{ASSETS, NODES};     # returns of assets at each node
10 set BENCHMARK;                   # BENCHMARK = number of benchmark realization
11 param VBenchmk{BENCHMARK};       # values of benchmarks
12 param HBenchmk{BENCHMARK};       # 2nd order SD values of benchmarks - calc'ed
13 param Gamma;
14
15 set Child{NODES} within NODES;
16 # stage 0
17 block stages0 {n0 in Child[null]) {
18     var x_hold{ASSETS} >=0;
19     var shortfall{BENCHMARK} >=0;
20     var cash >=0;
21     subject to StartBudget:
22         (1+Gamma)*sum{a in ASSETS} (x_hold[a]*Price[a]) + cash  = InitialWealth;
23     # weighted sum of stage 1 nodes
24     subject to StochasticDominanceExp_up1{l in BENCHMARK}:
25         sum{n1 in NODES: Child[n0]=n1} ( Prob[n1] * stage1[n1].shortfall[l] ) <=
              HBenchmk[l]*InitialWealth;
26     # weighted sum of stage 2 nodes
27     subject to StochasticDominanceExp_up2{l in BENCHMARK}:
28         sum{n1 in NODES: Child[n0]=n1} ( Prob[n1] * sum{n2 in NODES: Child[n1]=
             n2}(Prob[n2]*stage1[n1].stage2[n2].shortfall[l] ) )= HBenchmk[l]*
             InitialWealth;
29
30     # stage 1
31     block stages1 {n1 in Child[n0]}: {
32         var x_hold{ASSETS} >=0;
33         var shortfall{BENCHMARK} >=0;
34         var cash >=0;
```

```
        var x_sold{ASSETS} >= 0;
        var x_bought{ASSETS} >= 0;

        subject to CashBalance:
            stage0[n0].cash - (1-Gamma)*sum{a in ASSETS} (Price[a]*x_sold[a])
            = cash + (1+Gamma)*sum{a in ASSETS} (Price[a]*x_bought[a]);
        subject to Inventory{a in ASSETS}:
            x_hold[a] - Return[a,n1] * stage0[n0].x_hold[a] - x_bought[a] +
                x_sold[a] = 0;
        subject to StochasticDominanceSlck{l in BENCHMARK}:
         sum{a in ASSETS}(Price[a]*x_hold[a]) + cash[n1] + shortfall[l] >=
             VBenchmk[l]*InitialWealth;

        # stage 2
        block stages2 {n2 in Child[n1]}:{
            var x_hold{ASSETS} >=0;
            var shortfall{BENCHMARK} >=0;
            var cash >=0;
            var wealth >= 0;
            var x_sold{ASSETS} >=0;
            var x_bought{ASSETS} >=0;

            subject to CashBalance:
              stage1[n1].cash - (1-Gamma)*sum{a in ASSETS} (Price[a]*x_sold[a])
                    = cash + (1+Gamma)*sum{a in ASSETS} (Price[a]*x_bought[a]);
            subject to Inventory{a in ASSETS}:
                x_hold[a] - Return[a,n1] * stage1[n0].x_hold[a] - x_bought[a] +
                    x_sold[a] = 0;
            subject to StochasticDominanceSlck{l in BENCHMARK}:
                sum{a in ASSETS}(Price[a]*x_hold[a]) + cash + shortfall[l] >=
                    VBenchmk[l]*InitialWealth;

            subject to FinalWealth:
                wealth - sum{a in ASSETS} (Price[a]*x_hold[a]) - cash = 0;

            #weight sum of the objective function
            maximize objFunc: Prob[n1]*Prob[n2]*wealth;
      }
    }
}
```

Listing 4.1: Explicitly-blocked model for a simple 3-stage ALM-SSD problem.

4.3 Structure Building Stage

PSMG analyses the structured problem from its model and data file, and builds two tree structures to represent this problem. One structure is called the *template model tree* and the other one is called the *expanded model tree*. In the rest of this section, we use the examples from previous chapter to demonstrate these two tree structures in detail.

4.3.1 Template Model Tree

The template model tree is PSMG's minimal internal representation of the problem's high level abstraction. The tree is built by reading and analyzing the model file directly for problem modelled using `block`-statement. For problem modelled with `stochastic` programming syntax, the template model tree is built on the translated block model which is the result of performing the conversion process (explained in section 4.2) on the original stochastic model.

Problem modelled using block syntax

The template model tree naturally presents the nested block statement declared in the model file of a structured problem modelled using `block` syntax. Each tree-node corresponds to one `block`-statement declared in the model file and contains a list of entities declared in this block. The entities can point to any of the set, parameter, variable, constraint or sub-block declared at the corresponding block. At this point the description of the entities is very much as described in the model file; they are analyzed with respect to their syntactical components (*i.e.* indexing expressions, variable references, etc), but no further processing is performed. In particular, indexing expressions and summation sets are not expanded, nor is any information from the data file used. Thus the template model tree encodes the problem and sub-problem dependency information as described by the block statements in the model file, while each node of the tree serves as a template description of the entities in this block that are to be made concrete at the later processing stage. Each tree-node is also associated with an indexing set expression which will be used for generating the expanded model tree. In other words, each tree node is the basic building block that can be reused or duplicated according to its associated indexing set to construct the full problem instance, namely the expanded model tree.

Figure 4.1 demonstrates the template model tree structure for the MSND problem. There are one root model and four sub-models in this template model tree. The root model has references to the `MCNFArcs` and `MCNFNodes` sub-models which are represented as two child nodes. Each of the sub-models `MCNFArcs` and `MCNFNodes` also has reference to a nested sub-model named `Net`. Even though the two `Net` sub-models have the same name, their declarations are in different scopes in the model file (Listing 2.1). Therefore they are referred to as different sub-models, one is `Root.MCNFArcs.Net` and the other one is `Root.MCNFNodes.Net`.

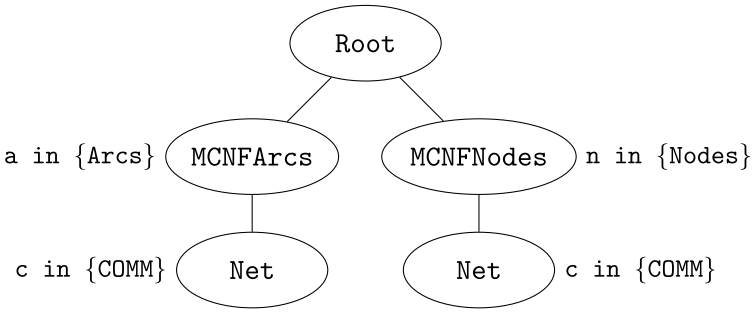


Figure 4.1: The template model tree for MSND problem model file specified in Listing 2.1.

The entities contained in each tree node are listed in Figure 4.2. The root node represents the full problem, and the child tree nodes represent the sub-problems. It is worth mentioning that most of the entities, such as sets, parameters, variables and sub-models declared in each tree node are associated with an indexing set expression (*e.g.* `cost{ARCS}, MCNFArcs{a in ARCS}`, etc.). However in building the template model tree, these indexing expressions are not evaluated and stored only in its high level definition form in the tree.

- `Root` node
  - Sets: `NODES, ARCS, COMM`
  - Parameters: `cost, basecap, arc_source, arc_target, comm_source, comm_target, comm_demand`
  - Variables: `sparecap`
  - Objective constraint: `costToInstall`
  - Sub-models: `MCNFArcs, MCNFNodes`
- `MCNFArcs` node
  - Set: `ARCSDIFF`
  - Variables: `capslack`
  - Constraints: `Capacity`
  - Sub-model: `Net`
- `MCNFNodes` node
  - Set: `NODESDIFF, ARCSDIFF`
  - Variables: `capslack`
  - Constaints: `Capacity`
  - Sub-model: `Net`
- `Net` node (child of `MCNFAcrs`)
  - Variables: `Flow`
  - Constraints: `FlowBalance`
- `Net` node, (child of `MCNFNodes`)
  - Variables: `Flow`
  - Constraints: `FlowBalance`

Figure 4.2: Entities in each template model tree node of MSND problem.

Problem modelled using stochastic syntax

The template model tree for a stochastic problem is not straightforward to be observed from its original model file in `stochastic`-block syntax, but the tree structure can be easily obtained if we look at the translated `block` model file. The translated `block` model fixes the stages number of this stochastic problem, and the rest of the model is equivalent to its original `stochastic` model. Figure 4.3 demonstrates the template model tree for a 3-stage ALM-SSD problem as a result of translating the stochastic model given in Listing 2.3. The number of stages is evaluated using parameters values provided in the data file given in Listing 2.4. An indexing set expression of a node set is also added for each stage block. PSMG uses this indexing set expression to build the expanded model tree for this stochastic programming problem instance later.

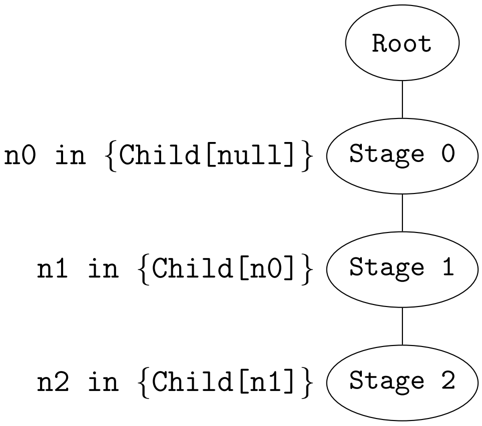


Figure 4.3: The template model tree for a 3-stage ALM-SSD problem specified in Listing 2.3. The indexing expression associated with each tree node is implicitly constructed by PSMG.

We can observe that each of the tree nodes represents one stage block in the translated model (Listing 4.1), and a list of entities and its high level declaration are also stored in each of the tree nodes. The entities contained in each of the tree nodes are listed in Figure 4.4.

We can observe that variable declarations are duplicated in necessary stage blocks in accordance with their declared stage scopes. For example, variables such as `x_hold, shortfall and cash` are duplicated in all the stage blocks, and variables, such as `x_sold` and `x_brought` are only presented in `stage1` and `stage2`.

The constraints `StochasticDomianceExp_up1` and `StochasticDomiStochas--ticDominanceceExp_up2` from `stage0` node are built from the constraint declaration, `StochasticDomianceExp` in the original `stochastic` model (at line 36 in Listing 2.3). The constraints involved with expectation expressions are always moved to the `stage0` (first stage), because variables declared in all the nodes of the same stage can only be accessed from their common ancestor. To demonstrate this, let us consider the 3-stage stochastic problem instance presented in Figure 2.2. The expected `shortfall` for nodes `2 and 3` in `stage1` can be evaluated only at the node of the root stage, namely `node1`. The expected `shortfall` for nodes

- `Root` node
  - Parameters: `T, InitialWealth, Parent, Probs, Prices, Return, VBenchmk, HBenchmk`
  - Sets: `TIME, NODES, ASSETS, BENCHMARK, Child`
  - Sub-model: `sage0`
- `stage0` node
  - Variables: `x_hold, shortfall, cash`
  - Constraints: `StartBudget,StochasticDomianceExp_up1, StochasticDominanceExp_up2`
  - Sub-model: `stage1`
- `stage1` node
  - Variables: `x_hold, shorfall, cash, x_sold, x_brought`
  - Constraints: `CashBalance, Inventory, StochasticDomianceSlck`
  - Sub-model: `stage2`
- `Stage2` node
  - Variables: `x_hold, shortfall, cash,wealth, x_sold, x_brought`
  - Constraints: `CashBalance, Inventory, StochasticDomianceSlck, FinalWealth`
  - Objective: `objFunc`

Figure 4.4: Entities in each template model tree node of ALM-SSD problem.

`4, 5, 6` and `7` can only be evaluated at their common ancestor node which is also `node1` in `stage0`.

The template model tree for a stochastic model also has a root node to store sets, parameters and variables declared outside of the stochastic block. Such variables behave like deterministic variables for the entire stochastic process, which is same as declaring the variables with *deterministic* keywords inside a stochastic block.

PSMG builds the template model tree for problems modelled using either `block` or `stochastic` syntax. This design allows PSMG to share the most common structure processing routine for the two model types. Once the template model tree is built, the rest of the problem structure processing routines are the same for problems modelled using either `block` or `stochastic` syntax.

4.3.2 Expanded Model Tree

The *expanded model tree* represents an instance of the problem which is generated after reading the problem data. It is obtained by creating copies of each sub-model in the template model tree in accordance with the associated indexing expression. In this process, data file is only used to expand the index set expression of each tree node; the other indexing set, such as the one used in the variable, parameter, and constraint declarations are not expanded and remain as high level template in the template model tree node. However the cardinality of the variables and constraints in each sub-model are counted to provide the problem size information. This allows the expanded model tree node to store the scalar information (such as number of local constraints and local variables), therefore only minimal amount of memory is used for storing the problem structure. Each node in the expanded model tree also has a pointer to the corresponding template model tree node, where the declaration templates of local entities can be retrieved from. The expanded model tree thus provides the context in which to interpret the template variable and constraint definitions in the template model tree.

Figure 4.5 demonstrates an expanded model tree for an MSND problem instance constructed by PSMG after reading the data file (in Listing 2.2). This expanded model tree is generated by repeating `MCNFArcs, MCNFNodes` and `Net` tree nodes in the template model tree (Figure 4.1) for each set value of their associated indexing set. For example, the indexing set expression '`a in` {`Arcs`}' in the template model tree is expanded into three nodes (`MCNFArcs_A1`, `MCNFArcs_A2` and `MCNFArcs_A3`) in the expanded model tree.

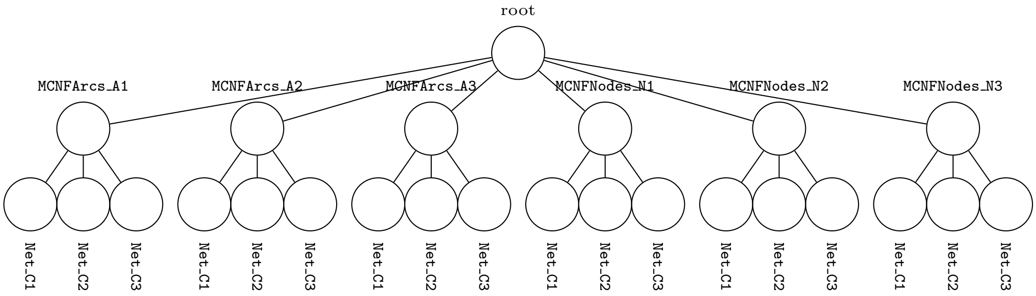


Figure 4.5: The expanded model tree of an MSND problem instance constructed by PSMG using data file in Listing 2.2.

Figure 4.6 demonstrates an expanded model tree for a 3-stage ALM-SSD problem instance using its template model tree (in Figure 4.3) and the data file (in Listing 2.4). The expanded model tree is generated by repeating the `stage0, stage1` and `stage2` tree nodes in the template model tree for each set value of their associated indexing set. That is, the indexing set expression {`Child[null]`} is evaluated to the node set {`1`} which only contains the root node in the scenario tree. `Child[1]` is evaluated to a node set {`2,3`}. `Child[2]` and `Child[3]` are evaluated to node sets {`2,3`} and {`4,5`} respectively.

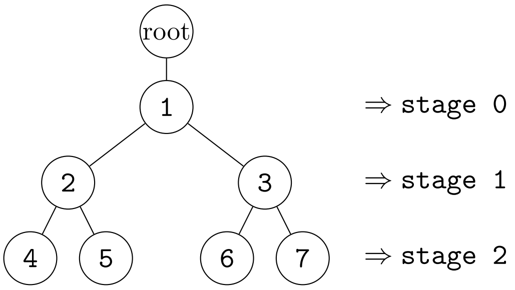


Figure 4.6: Expanded model tree for a 3-stage ALM-SSD problem instance constructed by PSMG using data file in Listing 2.4.

Once the expanded model tree is constructed by PSMG, it is passed to the solver. The solver is able to retrieve the concrete structure and the size information about the problem and each of its sub-problems from the expanded model tree. Each expanded model tree node represents a sub-problem, whereas the root node represents the top-level problem. Constraint and variable numbers for each of the sub-problems can be retrieved from its corresponding expanded model tree node. The problem structure layout matches with the expanded model tree structure; hence, the solver can traverse the expanded model tree recursively to retrieve the structure information and set up the problem for parallel solution.

It is also worth mentioning that PSMG's memory usage in this stage is restricted to information which the solver needs to decide on the distribution of problem components to processors, namely, the structure of the problem (*i.e.* number of blocks and the relationship among blocks, and the size of each block). Due to the minimal processing involved, this phase is very fast. According to our performance benchmark tests in Chapter 8, this processing finishes within seconds for problems of more than a million variables and constraints.

Now PSMG can proceed to the second stage for generating a structured problem, namely the parallel problem generation stage. The expanded model tree then acts as an interface between PSMG and the solver.

4.4 Parallel Problem Generation Stage

In this stage, PSMG generates blocks of the problem's Jacobian and Hessian matrices as required by the solver. In order to evaluate a particular block, PSMG will expand the indexing set expressions for variables, constraints and compute any temporary sets and parameters needed for function and derivative evaluations in this block. The indexing set for summation expressions are also evaluated and stored in memory only when it is necessary. The work involved in this stage takes a major percent of the whole model processing time. PSMG is able to distribute this work among parallel processors. In addition, PSMG implements state-of-the-art automatic differentiation (AD) algorithms that can efficiently compute accurate Jacobian and Hessian blocks without truncation-error.

4.4.1 Solver Driven Problem Assignment

To avoid unnecessary communication it is evident that function and derivative evaluation routines for a particular part of the the problem (and by extension the generation of the necessary data) should be performed on the processor that is also assigned to have this part of the problem. Moreover, allocation of sub-problems to processors should be taken into account load balancing and cost of inter-process communications. We note that both these issues are highly dependent on the algorithm used by the solver and can only be judged by the solver.

This leads us to follow the design of a solver driven work assignment approach for PSMG's solver interface, in which initially a minimal set of information describing the problem structure is extracted from the model and passed to the solver. The solver then decide how to distribute problem components among processors based on this structure and subsequently initiate the remainder of the model processing and function evaluations through call-back functions on a processor-by-processor basis. It is clear that the expanded model tree fits this purpose.

On-demand problem processing

Figure 4.7 illustrates the overall work-flow between PSMG and the solver.

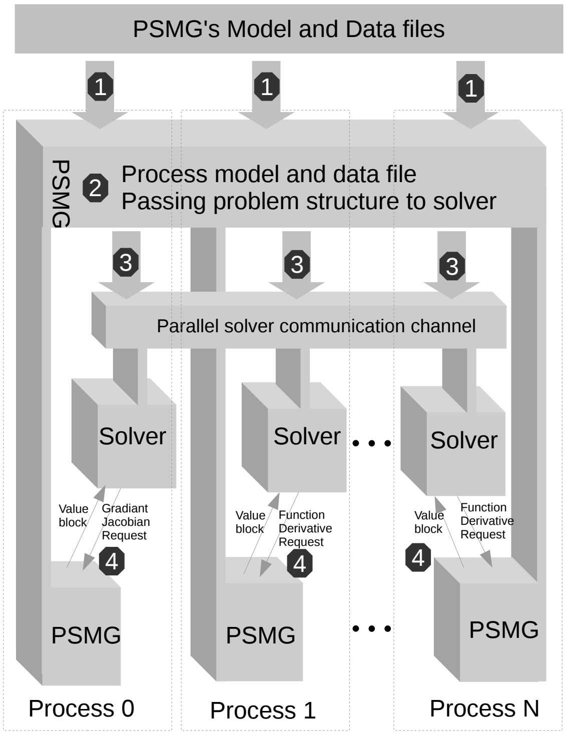

PSMG's Model and Data files
1
1
1
PSMG
2
Process model and data file
Passing problem structure to solver
3
3
3
Parallel solver communication channel
Solver
Solver
Solver
Value block
Gradiant Jacobian Request
4
Value block
Function Derivative Request
4
Value block
Function Derivative Request
4
PSMG
PSMG
PSMG
Process 0
Process 1
Process N

Figure 4.7: The PSMG workflow diagram with a parallel optimization solver.

The steps of in the work-flow diagram in Figure 4.7 can be summarised as below:

1. PSMG reads model file and data file.
2. PSMG extracts the template tree and builds the expanded model tree to describe the problem structure information (as described in the structure building stage (4.3)).
3. PSMG passes the expanded model tree to the parallel solver. The solver will then decide how to distribute the problem parts, *i.e.* the nodes of the expanded model tree, among processes to achieve optimal load balancing and data locality.
4. The solver uses PSMG's callback function interface to request the specific blocks or subproblems to be evaluated on a processor per processor basis.

Steps 1–3 form the structure building stage. After building and passing the problem structure information to the solver, every solver process knows the size and structure information of the entire problem. Once this information is available, there is no further need for any communication among the PSMG processes. The solver can then employ an appropriate distribution algorithm to assign blocks to available processes in order to achieve load balancing and minimize solver internal communication. After that, every parallel processes can request the function and derivative values of each block to be generated in parallel from PSMG. This work corresponds to step 4 above, namely the parallel problem generation stage.

In order for PSMG to evaluate function and derivative values for a particular block of a sub-problem, additional processing needs to take place. Some major work items involved in step 4 are listed below.

- expanding the indexing set expression of the variables and constraints;
- initializing the variables and build concrete constraint expression;
- expanding the sum expressions in the constraint declaration;
- computing the set and parameter values that are needed;
- initializing the memory block used by automatic differentiation routines.

The amount of computing work involved and memory allocated are dependent on the size and complexity of the block required by the solver. Therefore, to maximise PSMG's parallel problem generation efficiency, it is important for the solver to adopt a balanced parallel allocation scheme by considering the number of blocks and the size of each block. The expanded indexing set, any temporary set and parameter values are stored in memory for future use.

Memory consideration

The total memory used in PSMG consists of the memory used for storing problem structure and problem data that required for computing the function and derivative evaluations. The problem structure is represented by the prototype model tree and expanded model tree. The expanded model tree only stores scalar information (*i.e.* number of variables and constraints in each tree node) before the parallel problem generation stage starts. Therefore it contributes a very minimal percentage of the total memory usage, which is confirmed in the performance evaluation result for PSMG in Chapter 8.

At the parallel problem generation stage (step 4 in Figure 4.7), additional memory is allocated to store the problem data in the expanded model tree. This "lazy" approach of data computation guarantees that processing power and memory is only used when and on the process where it is necessary.

Therefore, PSMG can effectively distribute the memory usage for storing problem data among parallel processes. However, the problem structure information is duplicated on every processes. This design is crucial for enabling the solver driven problem distribution to achieve the on-demand problem generation and further eliminates inter-process communication in both the model generation and function evaluation stages.

Furthermore, the names for variables and constraints are not stored in the data memory, but rather can be dynamically generated upon request from the solver. In most cases, the constraint and variable names are needed for debug or solution reporting purpose. Storing those names for problem with millions of constraints and variables could consume a considerable amount of memory. Therefore it is not worthwhile to store them.

Callback function interface with solver

PSMG uses the expanded model tree as the interface for communicating the problem structure to the solver. PSMG implements C/C++ type of solver interface as member functions or properties of the `ExpandedModel` class. It is quite convenient for solver to employ PSMG's interface as a callback function to require each block (of Jacobian or Hessian matrix) or sub-vector (of objective gradient, constraint values, etc.) on demand in parallel. Further details about PSMG's solver interface methods are discussed in Chapter 6.

4.4.2 Constraint Separability Detection

In a nested structure problem, constraint expression in a sub-problem can be modelled with variables declared from itself or either its ancestor or descendant problems. Since PSMG evaluates Jacobian and Hessian matrix by blocks, and function and derivative values by their separable parts, it is quite useful to separate the whole constraint expression into parts based on blocks. Therefore at the problem generation stage, only the part of constraint which contributes to the requested block is evaluated. This can also be useful to verify the *block separability assumption* in Section 2.4.1.

PSMG achieves this by recursively tracing the constraint expression declarations and grouping the constraint expression in line with the separability of the variables used in this constraint. Each constraint is grouped into several additively separable parts. For a linear constraint, each constraint part contributes to one block in the Jacobian.

For example, the `Capacity` constraint expression declared in the MSND problem in Listing 2.1 (line 15) will be partitioned into two parts: `sum{k in COMM} Net[k].Flow[j] + capslack[j]`; and `sparecap[j]`, where the first part is used for evaluating the diagonal block, and the second part is used to evaluate the right hand side border block (in Figure 2.1).

For nonlinear constraint, a separable part of this constraint may contribute to one or more blocks in the Jacobian and Hessian matrix, since variables in this separable part may from different levels of the problem.

4.4.3 Function and Derivative Evaluation

The majority of the time spent in PSMG's problem generation stage is on evaluating Jacobian and Hessian matrix at a given point x. An efficient implementation for computing such values is paramount for PSMG to achieve excellent performance. In particular, for Nonlinear Programming (NLP) problems, the Jacobian and Hessian matrices may need to be evaluated for every iteration. The derivative evaluation is a very interesting research area by itself, therefore we discuss PSMG's function and derivative evaluation design in the next Chapter.

Chapter 5

Evaluating Derivatives in PSMG

In general, there are three ways of computing derivatives, numerical differentiation, symbolic differentiation and automatic differentiation. Numerical differentiation approximates the derivative values using the rate of change between two or more function points. It is prone to truncation error [48], sometimes making the result useless [49], but it can return the derivative value quickly. Symbolic differentiation first requires an algebraic expression of the derivatives function to be computed, and then the derivatives at a function point can be evaluated using the computed symbolic expression. Symbolic differentiation method brings no truncation error, but it needs additional memory space to store the derivative function expressions and also requires considerable amount of programming work for an efficient implementation. Also, memory space is a critical resource for solving large scale optimization problems. The automatic differentiation(AD) method neither requires the symbolic expression to be computed nor introduces any truncation error (that is outside of the machine's precision) for evaluating derivatives of a function, therefore the AD method is the most efficient among the three methods. Because of the superiority of AD over other two classical methods for derivative evaluation, most AMLs adopt AD method in their implementation. We have also implemented the AD routine in PSMG.

5.1 Forward AD Algorithm

A number of research papers contributed to the theoretical development of AD [50, 51, 52]. Griewank and Walther have also studied the memory and run-time complexity for AD algorithms in [53].

To simplify the discussion, we use the same notation as introduced by Griewank and Walther in [53]. We can consider a scalar function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ which can be represented using a list of elemental functions $(\phi_{1-n}, \cdots, \phi_l)$. Each elemental function ϕ_i is either unary or binary, $\forall i \in \{1, \cdots, l\}$ and $\phi_i = x_i$ for $i \in \{1-n, \cdots, 0\}$. Using the elemental function list, it is quite convenient to construct the computational graph as a directed acyclic graph $G(N, A)$, where the size of node set, $|N| = n + l$. Let v_i represent the output value of ϕ_i, $\forall i \in 1-n, \cdots, l$; hence, $v_i = \phi_i(v_j)$ if ϕ_i is an unary function and arc $(j, i) \in A$; and $v_i = \phi_i(v_j, v_k)$ if ϕ_i is a binary function and arcs (j, i) and $(k, i) \in A$. We should also define a dependence relation $\prec$, where $i \prec j$ means arc $(j, i) \in A$. Then we can write more concisely $v_i = \phi_i(v_j)_{i \prec j}$ for $i \in 1, \cdots, l$ and $v_i = \phi_i = x_i$ for $i \in \{1-n, \cdots, 0\}$.

The main idea in AD is to apply the chain rule along the computation graph of a function expression to evaluate the derivative. According to the direction of propagation, AD algorithms can be categorized in two modes: forward; and reverse. We demonstrate these two modes by an example. Figure 5.1 demonstrates computational graph for function expression: $f(x) = (x_{-1}x_{-2}) + x_0 \sin(x_{-2})$.

In forward AD, we should declare a list of tangent variables $(\dot{v}_{1-n}, \cdots, \dot{v}_l)$ for each node in the computational graph. We need to choose a *seed direction* in x, and then apply the forward AD algorithm (Algorithm 1) to compute the derivative. The *seed direction* is the component in x vector that is considered to be variable and rest of the components in x are kept constant while applying the chain rule, therefore choosing a seed direction allows the corresponding partial derivative to be evaluated using forward AD algorithm. A point on the function should also be provided to indicate where on the function the derivative is evaluated. For example, $\frac{\partial f}{\partial x_{-1}}$ can be evaluated using Algorithm 1 by setting input $a = -1$, where a is the seed direction parameter. Figure 5.2 illustrates the computation steps for evaluating $\frac{\partial f}{\partial x_{-1}}$.

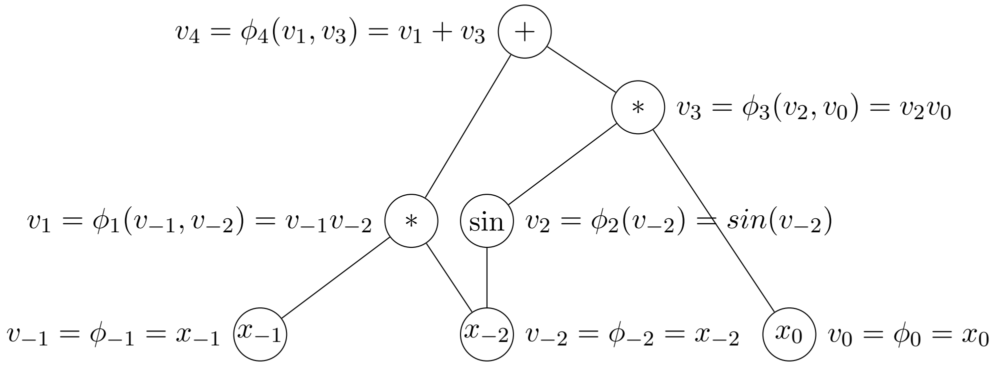

$v_4 = \phi_4(v_1, v_3) = v_1 + v_3$
$+$
$*$
$v_3 = \phi_3(v_2, v_0) = v_2 v_0$
$v_1 = \phi_1(v_{-1}, v_{-2}) = v_{-1}v_{-2}$
$*$
sin
$v_2 = \phi_2(v_{-2}) = sin(v_{-2})$
$v_{-1} = \phi_{-1} = x_{-1}$
x_{-1}
x_{-2}
$v_{-2} = \phi_{-2} = x_{-2}$
x_0
$v_0 = \phi_0 = x_0$

Figure 5.1: The computational graph for function $f(x) = (x_{-1}x_{-2}) + \sin(x_{-2})x_0$.

When using the forward AD, evaluation of the gradient vector requires the algorithm to be run n times, where n is the number of independent variables in the function expression (*i.e.* $n = |x|$). That is, it requires n forward sweeps on the computational graph to evaluate the gradient vector. Otherwise we can modify the algorithm by carrying a vector of size n at each loop iteration to compute gradient in one forward sweep; however, this is not memory efficient. The forward AD algorithm is inefficient in both memory usage and run-time speed.

5.2 Reverse AD Algorithm

The Jacobian matrices of the large scale structured problems we are dealing with are always sparse. In order to fully exploit the sparsity structure of the Jacobian matrices, we have adopted the reverse mode AD in PSMG.

In the reverse mode AD, we should declare a list of adjoint variables $(\bar{v}_{n-1}, \cdots, \bar{v}_l)$ for each node $i \in \{n-1, l\}$ in the computation graph $G(N, A)$. The algorithm

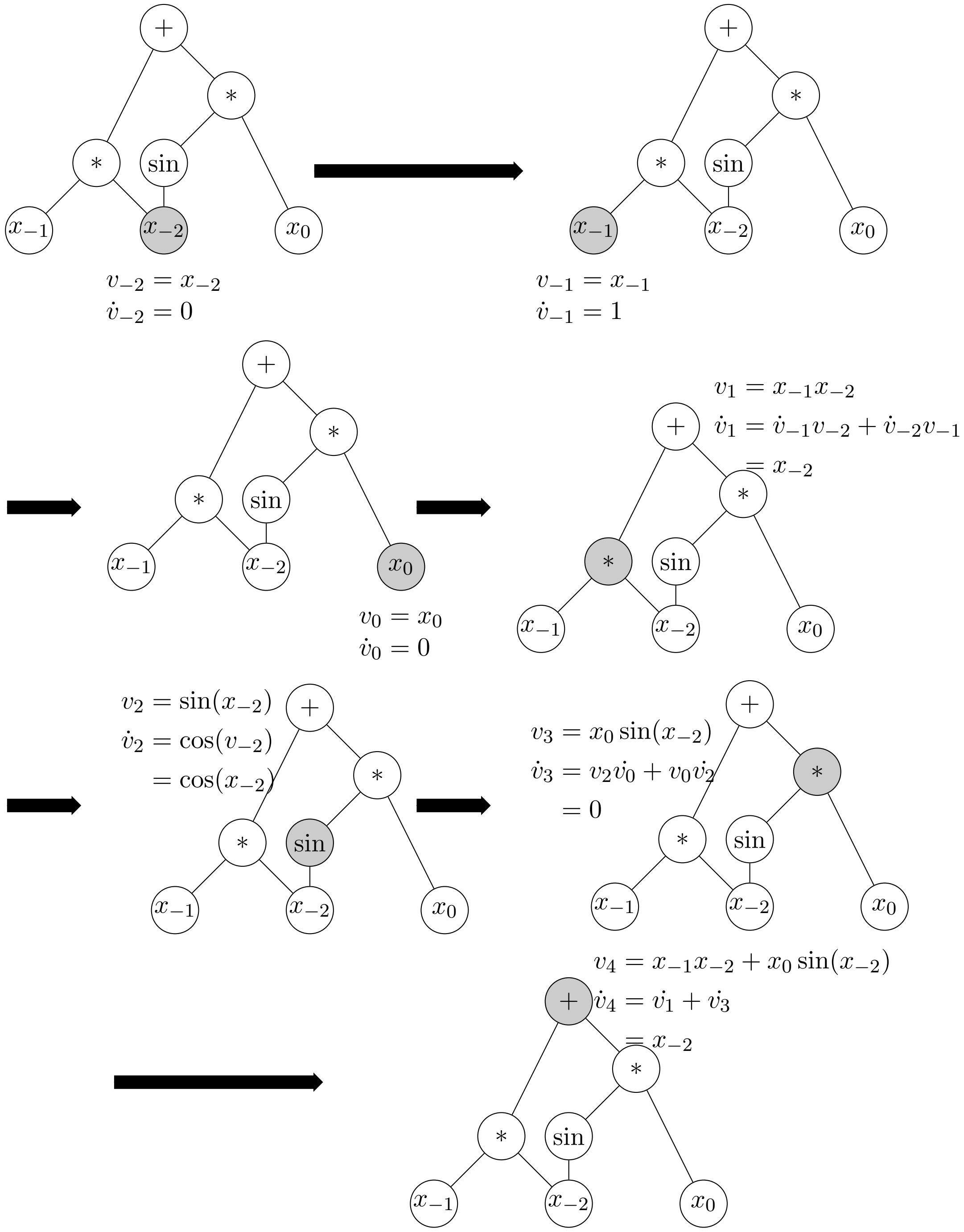


Figure 5.2: Illustration for applying forward AD to evaluate $\frac{\partial f}{\partial x_{-1}}$ of $f(x) = (x_1 x_{-2}) + \sin(x_{-2})x_0$.

Algorithm 1 The forward AD algorithm for computing gradient

```
Input:
    $a$                                          ▷ the seed direction
    $x$                                          ▷ the given function point
Output:
    $\dot{v}_l$                                  ▷ partial derivative for the seed direction
for $i = n-1, \cdots, 0$ do
    $v_i \leftarrow x_i$
    if $i = a$ then
        $\dot{v}_i \leftarrow 1$
    else
        $\dot{v}_i \leftarrow 0$
    end if
end for
for $i = 1, \cdots, l$ do
    $v_i = \phi_i(v_j)_{i \prec j}$
    $\dot{v}_i \leftarrow 0$
    for $i \prec j$ do
        $\dot{v}_i += \frac{\partial \phi_i}{\partial v_j} \dot{v}_j$
    end for
end for
```

for reverse mode AD is presented in Algorithm 2. As we can see that the gradient vector can be evaluated by calling to the reverse AD algorithm once. Only one forward and one reverse sweep on the computational graph is needed in the reverse AD algorithm. In the forward sweep, the value for each ϕ_i is computed and stored in variable v_i. The reverse sweep propagates the adjoints from top to bottom. Once finishing the reverse sweep, the gradient vector is given in the adjoint variables at each leaf node (the independent variable node). Reverse AD can exploit the sparsity structure in the Jacobian matrix. Furthermore, by carefully arranging the evaluation order on the computational graph, reverse AD can be efficiently implemented using a stack [53]. Figure 5.3 and 5.4 demonstrate the computation steps in forward and reverse sweeps for evaluating $\nabla f(x)$ using reverse mode AD algorithm. The computational graph and function expression is illustrated in Figure 5.1. Once the reverse AD finishes the reverse sweep on the computational graph, the gradient vector $(\nabla f(x))^T$ can be obtained from adjoints $(i.e.\ (\frac{\partial f}{\partial x_{-2}}, \frac{\partial f}{\partial x_{-2}}, \frac{\partial f}{\partial x_{-2}}) = (\bar{v}_{-2}, \bar{v}_{-1}, \bar{v}_0) = (x_0 cos(x_{-2}) + x_{-1}, x_{-2}, sin(x_{-2})))$.

5.3 Computing Sparse Hessian

Most optimization solvers require the Lagrangian Hessian matrix to be evaluated by the model generator. The Lagrangian function is the linear combination of constraint and objective functions. Therefore it is crucial for performance reason to have the sparse Hessian matrix evaluated efficiently in our AD implementation. Gebremedhin et al. presented an effective approach of computing the sparse

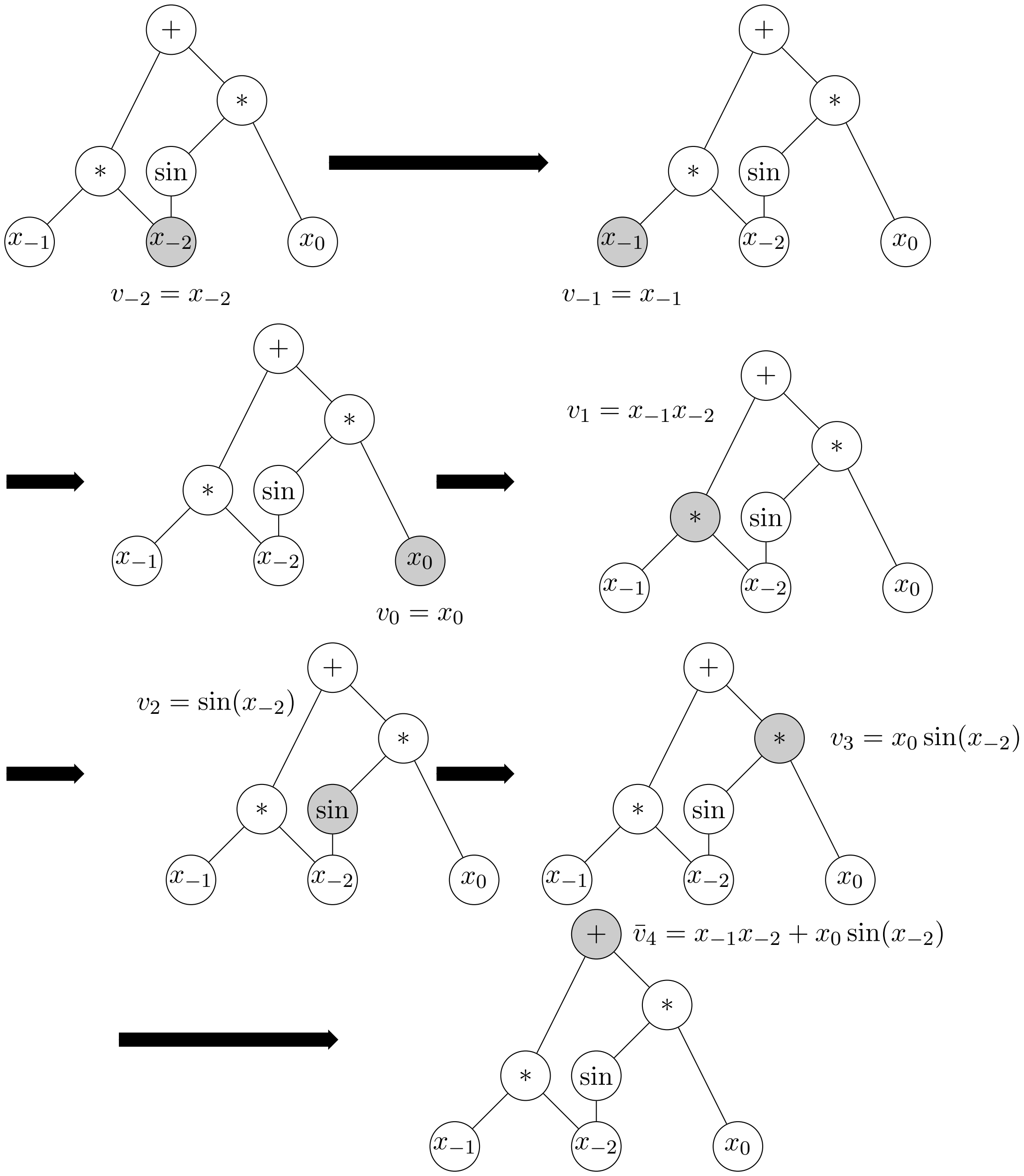


Figure 5.3: Illustrating the forward sweep for applying reverse AD to evaluate $\nabla f(x)$ of $f(x) = (x_{-1}x_{-2}) + \sin(x_{-2})x_0$.

Algorithm 2 The reverse AD algorithm for computing gradient

```
Input: $x$                                        ▷ point on function
Output: $\bar{v}$                                 ▷ gradient of the function at $x$
for $i = n-1, \cdots, 0$ do                       ▷ start forward sweep
    $v_i \leftarrow x_i$
    $\bar{v}_i \leftarrow 0$
end for
for $i = 1, \cdots, l$ do
    $v_i = \phi_i(v_j)_{i \prec j}$
    $\bar{v}_i \leftarrow 0$
end for
$\bar{v}_l \leftarrow 1$
for $i = l, \cdots, 1$ do                         ▷ start reverse sweep
    for $i \prec j$ do
        $\bar{v}_j += \frac{\partial \phi_i}{\partial v_j} \bar{v}_i$
    end for
end for
```

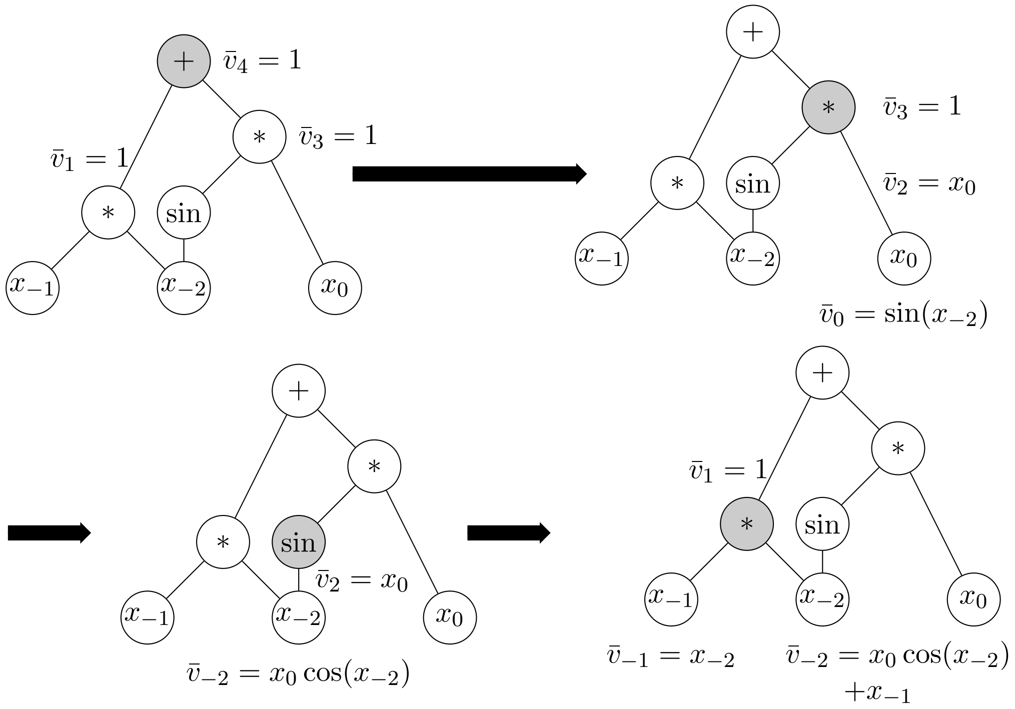


Figure 5.4: Illustrating the reverse sweep for applying reverse AD to evaluate $\nabla f(x)$ of $f(x) = (x_{-1}x_{-2}) + \sin(x_{-2})x_0$.

Hessian by combining graph coloring and the reverse AD algorithm [54]. This approach is also integrated into ADOL-C [55] using the graph coloring library, ColPack [56]. The sparse Hessian computation using graph coloring is a two-step algorithm. In the first step the compressed Hessian matrix $C = HS$ is evaluated, where the *seed matrix* [54] S is obtained according to a coloring mode. Then the Hessian matrix H can be recovered from C using the *seed matrix* obtained in the second step. It seems that this two-step algorithm for sparse Hessian evaluation was the most efficient approach before a recent work published by Gower et al. [57] who presented a "truely" reverse algorithm for sparse Hessian calculation, namely `edge_pushing`. The `edge_pushing` algorithm exploits both the symmetry and sparsity structure in Hessian matrix. It allows the Hessian matrix to be evaluated in one forward and one reverse sweep on the computational graph. According to the result analysis presented in [57, 58], `edge_pushing` algorithm demonstrates more robust performance than the graph coloring algorithm. Therefore we have chosen to implement the `edge_pushing` algorithm in PSMG's AD module.

In addition, most optimization solvers require the model generator to provide the sparsity structure for the Hessian matrix to set up the memory storage before the actual Hessian matrix is evaluated; hence a routine for just returning the Hessian matrix structure is also quite useful for this purpose. Routines for retrieving the non-zero structure of the Hessian matrix are provided in AMPL and their usage is documented in [31]. Walther [59] also presented a forward mode algorithm for evaluating the sparsity structure of the Hessian matrix; and this algorithm is implemented in ADOL-C. However a "truly" reverse algorithm for detecting non-zero structure of the Hessian matrix is recently developed by Gower, namely `edge_push_sp` [60]. We have also implemented this reverse algorithm in PSMG's AD module to return the sparsity structure of the Hessian matrix.

5.4 AutoDiff Library Module

There are various open source libraries for using AD to compute derivatives of a function [61]. These libraries are implemented using either source code transformation or operator overloading method. The AD libraries implemented using both of the methods require the compiler to be used to build the computation graph of a function expression. On the other hand, PSMG employs a user friendly interface design, and requires no compiler intervention after it is deployed a target machine. The modeller only needs to specify the model and data files to use the modelling system. We need either to develop an interface between PSMG and an open-source AD library or to implement our own AD module. For technical reason and also the ability to tailor the AD algorithm for the purpose needed by PSMG, we choose the latter approach.

We should now discuss some design concepts and implementation details in the PSMG's AD module. The AD routines in PSMG are designed as a separate module, and itself is released as an open source library, namely `AutoDiff_Library` [62]. AD is an active research area, therefore the modular design allows PSMG to adopt the latest AD algorithms easily. Unit tests implemented using Boost Testing Framework [63] are also integrated in `AutoDiff_Library` for testing new

AD algorithms.

While implementing the AD library, we have consulted an Object-Oriented implementation from [64]. We have also borrowed the “tape” idea from ADOL-C [55, 65] for storing the computational graph and temporary variables needed in the derivative evaluation. We implemented the tape using a collection of array objects that store the elementary function operators of the function expression and any intermediate values such as $v_i, \bar{v}_i, \dot{v}_i, \frac{\partial \phi_i}{\partial v_j}$, etc. Using tape allows the AD routine to store the computed intermediate values that can be quickly accessed for later use; therefore it is important for an efficient AD implementation. It is worth mentioning that our AD routine implementation is on scalar function, a more robust AD implementation based on vector function as presented in ADOL-C [55]. This could be a future route to go for improving the performance of PSMG.

Our `AutoDiff_Library` provide interface methods for building the constraint functions in the form of directed acyclic graphs, where the leaf nodes represent either variables or parameters, and the non-leaf nodes represent either binary or unary operators. Then the Jacobian and Hessian matrices (in sparse form using `ublas` [66]) can be evaluated using corresponding interface methods from this library.

The `AutoDiff_Library` is implemented in C++ with Object-Oriented design. The computational graph for function expression is built using nodes initiated from classes: `VNode`; `BinaryOpNode`; `UnaryOpNode`; `PIndex`; and `PVal`. Their explanations are listed below. Figure 5.5 presents the class hierarchy in AD module.

- `BinaryOpNode`
  - represents a binary operator node that has two operands as its children.
- `UaryOpNode`
  - represents a unary operator node that has only one operand as its child.
- `VNode`
  - represents a variable node which is also a leaf node.
- `PVal`
  - represents a *fixed* parameter leaf node in the expression tree.
- `PIndex`
  - represents a parameter node whose value can be looked up by index.

In order to use the `AutoDiff_Library`, one should include the header file `autodiff.h` where interface methods for building the computational graph are declared. There are also interface methods for computing the Jacobian and Hessian matrices, and their corresponding non-zero structures. PSMG computes the Jacobian and Hessian in blocks. Therefore, tailor made method calls to return those matrix blocks are provided in the AD module. The solver can decided how to deal with the block matrices for other part of the problem (*i.e.* either communicate the block or discard it). The interface methods and their explanations are attached in Appendix A of this thesis.

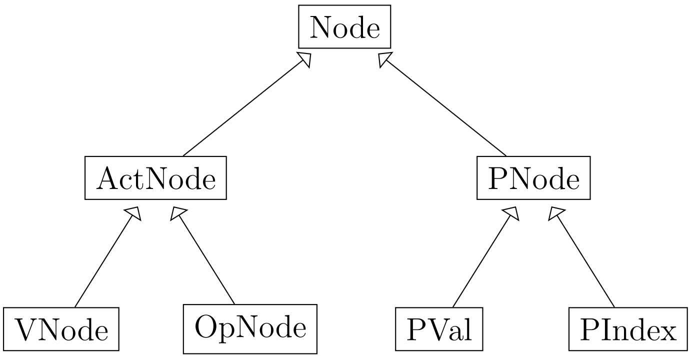


Figure 5.5: The class hierarchy diagram in AutoDiff Library for building function expression tree.

5.5 Future Work in the AD Module

The AD module in PSMG has its own interface and can be used as a standalone library. Because derivative evaluation takes a consideration amount of computation effort in problem generation, it is quite useful to conduct a performance analysis between the PSMG's AD module and other open sourced AD libraries. In this case, we can identify any potential performance issue in the AD evaluation routine and to conduct further improvements to PSMG's AD module implementation.

5.5.1 Some Parallel Consideration

In the reverse sweep of AD algorithm, it traverses the computational graph from root to leaf nodes. At each non-leaf node with more than one child, there is an opportunity to traverse each child in parallel. It is note that Christianson [51] also described the similar approaches for parallel implementation of the reverse gradient algorithm. His method is to use a counter at each node to indicate the number of terms that are waiting to be accumulated in the reverse sweep. When the counter is zero, the adjoint variable at this node is ready. Then, the reverse sweep can be propagated in parallel for its children. Using this approach, the adjoint variables have to be accessed through some synchronization primitives. There are also prototype implementations for both reverse and forward AD algorithms using parallel programming such as using MPI in [67] and OpenMP in [68].

Chapter 6

PSMG Solver Interface

In this chapter, we discuss PSMG's solver interface methods for retrieving problem structure and evaluating the function and derivatives. PSMG offers two categories of interface methods (*distributed* and *local*) for function and derivative evaluations, which provide great flexibility to support different parallel solver algorithms. PSMG can be used for modelling LP, QP and NLP problems with nested structure. The interface methods for structure retrieval are the same for these three problem types, whereas the interface methods for function and derivative evaluation can be different.

6.1 General Problem Formulation

PSMG is designed to model structured problems. A structured problem can be naturally presented using a single-rooted tree as described in Chapter 4, where each child node represents a sub-problem, and the root node represents the master problem. The structured problem can also be nested, which corresponds to a tree with more than two levels. To help us explain PSMG's solver interface design, we should present the general problem formulation for problems modelled in PSMG.

Let us first restate *block separability assumption* (Section 2.4.1) more formally. Given $\mathcal{T}$ be the set of nodes in the tree that represents the structured problem. We should also define a function *anc* that takes a node $i \in \mathcal{T}$ and returns a set of tree nodes containing i's ancestors. Then we can give the generalized *block separability assumption* below.

Block Separability Assumption (Generalized) Let i and j be any two different nodes in $\mathcal{T}$ that is not on the same *branch* – which means there isn't any path on the tree started from root that passes both nodes, i and j. Let the set of nodes $\mathcal{I} = anc(i) \cap anc(j)$, and $\mathcal{F}$ be the set of constraint functions declared in every node $n \in \mathcal{I}$. Let $\mathcal{V}_i$ and $\mathcal{V}_j$ be the sets of variables declared in node i and j respectively. Given variables $v_i \in \mathcal{V}_i$ and $v_j \in \mathcal{V}_j$, and constraint function $f_k, k \in \mathcal{F}$, v_i and v_j must not present in the expression of $\partial_{v_j} f_k$ and $\partial_{v_i} f_k$ respectively. In this case, we say v_i and v_j satisfy the *block separability assumption.*

In above description of the generalized *block separability assumption*, we have relaxed the additively separable requirement for variables v_i and v_j (originally stated in 2.4.1). This assumption holds providing that v_i and v_j are separable with respect to the partial differentiation operator in constraint function f_k. The additively separable relation between v_i and v_j is a special case which is often enough for modelling structured problems. In addition, PSMG provides the `Exp(·)` function (inherited from SML [2]) for modelling stochastic programming problems. Many risk measures, such as variance, conditional value at risk (CVAR), stochastic dominance constraints, etc can be modelled using expectation-like constraints [1, 34, 69].

For simplicity, we consider a general two-level structured problem whose mathematical formulation is given in (6.1). This problem has one master problem and n sub-problems. Each q_i represents the part of the objective function declared in i^{th} sub-problem, where $i \in \{1, \cdots, n\}$. q_0 represents the part of the objective function declared in the master problem. The overall objective function of this problem is the linear combination of q_i, $\forall i \in \{0, \cdots, n\}$. g_0^i is the constraint in the master problem which links all the sub-problems' variables. The x_i are the variables declared in the i^{th} sub-problem, $\forall i \in \{1, \cdots, n\}$. x_0 is the complicating variables declared in master problem. Without lost of generality, q_i and g_i can represent either linear or nonlinear vector functions; and x_i can represent either scalar or vector, $\forall i \in \{0, \cdots, n\}$. In this formulation we assume variables $x_i, \forall i \in \{1, \cdots, n\}$ are additively separable in the complicating constraints (that are declared in the master problem). However, the results can be obviously applied for problem formulation that satisfies the generalized version of the *block separability assumption.*

$$\min_{x_i} \quad q_0(x_0) + \sum_{i=1}^{n} q_i(x_i, x_0) \tag{6.1a}$$

$$\text{s.t. } g_i(x_i, x_0) \le 0, \quad i \in \{1, \cdots, n\} \tag{6.1b}$$

$$\sum_{i=1}^{n} g_0^i(x_0, x_i) \le 0 \tag{6.1c}$$

Given the *block separability assumption*, the constraints and objective function can be split into several separable parts according to the operation described in Section 4.4.2. Then (6.1) can be rewritten into an expanded formulation in (6.2). The additively separable part of the constraint expression involves only the complicating variables x_0 that are expressed separately using c_i^0, $\forall i \in \{0, \cdots, n\}$. PSMG always processes a model's constraints and objective function into the expanded formulation; hence, the expanded formulation is used to explain the principle of PSMG's solver interface in the rest of this chapter.

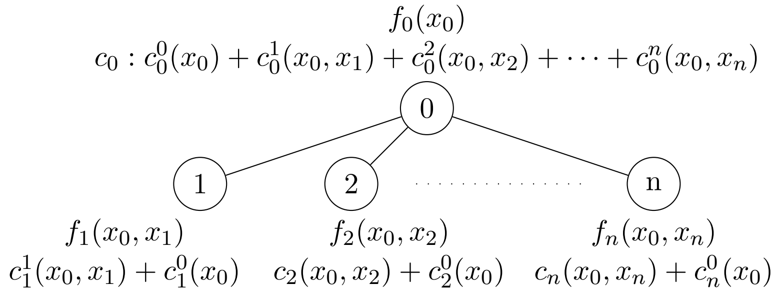

$f_0(x_0)$
$c_0 : c_0^0(x_0) + c_0^1(x_0, x_1) + c_0^2(x_0, x_2) + \cdots + c_0^n(x_0, x_n)$
0
1
2
n
$f_1(x_0, x_1)$
$c_1^1(x_0, x_1) + c_1^0(x_0)$
$f_2(x_0, x_2)$
$c_2(x_0, x_2) + c_2^0(x_0)$
$f_n(x_0, x_n)$
$c_n(x_0, x_n) + c_n^0(x_0)$

Figure 6.1: A general two level structured problem.

$$
\begin{array}{llllll}
\min & f_1(x_0, x_1) + f_2(x_0, x_2) & + \cdots + f_n(x_0, x_n) & +f_0(x_0) & & \text{(6.2a)} \\
\text{s.t.} & c_1^1(x_0, x_1) & & +c_1^0(x_0) \le 0 & & \text{(6.2b)} \\
 & \quad c_2^2(x_0, x_2) & & +c_2^0(x_0) \le 0 & & \text{(6.2c)} \\
 & & \ddots & \vdots \; \vdots & & \text{(6.2d)} \\
 & & c_n^n(x_0, x_n) & +c_n^0(x_0) \le 0 & & \text{(6.2e)} \\
 & c_0^1(x_0, x_1) + c_0^2(x_0, x_2) & + \cdots + c_0^n(x_0, x_n) & +c_0^0(x_0) \le 0 & & \text{(6.2f)}
\end{array}
$$

Figure 6.1 demonstrates the tree structure of this two level structured problem, where the constraint and objective expressions in every sub-problem are annotated next to every node. This tree structure is also the same as the expanded model tree generated by PSMG; hence an expanded model tree node can be used to refer to a sub-problem, where the root node can refer to the master problem.

We can conclude the following key facts in a general problem formulation of a problem modelled by PSMG.

- The overall objective function is declared in its additively separable parts in each expanded model tree node, where f_0 is declared at master problem (root node).

- The objective function part declared in an expanded model tree node can only use variables declared from this node itself and its ancestor nodes.

- The variables declared in an expanded model tree node can only be used in constraints declared in this node itself, or its ancestor or descendant nodes.

- The variables declared in any two nodes from sibling branches are additively separable with each other in the constraints. This is also known as our *block separability assumption.*

This formulation is guaranteed to produce a double bordered block-angular structure (in Figure 6.2) for Jacobian and Hessian matrix of the problem. It is worth mentioning that structure problems demonstrating either primal or dual block-angular structure in their Jacobian matrix are considered as special cases in PSMG. Therefore, it is unnecessary to discuss them separately.

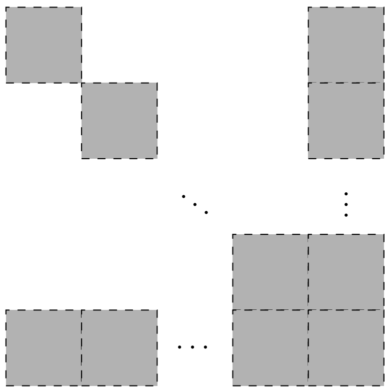


Figure 6.2: Double bordered block-angular structure for a two level structured problem

6.2 Problem Structure Retrieval

We have already explained that the problem structure can be presented using the expanded model tree in PSMG. In order to retrieve the problem structure and dimension information (constraint and variable numbers) for each sub- and master problem, the solver can use the interface methods and properties declared in the `ExpandedModel` class. The root node of the expanded model tree represents the master problem. Therefore, the solver can recursively trace from the root node to obtain the whole problem structure.

The keyword, `this` which is commonly found in Object-Oriented programming languages (*i.e* JAVA, C++, *etc.*) is used throughout rest of this chapter in the explanation of PSMG's solver interface functions. Because PSMG's interface functions are declared as member methods or properties in `ExpandedModel` class, `this` always refers the current expanded model tree node object, which could be either a sub-problem or the master problem (if `this` refers to the root node).

The interface properties and methods can be used to retrieve problem structure are listed below.

- `string name`
  - the name of `this` expanded model tree node.
- `uint numLocalVars`
  - the number of local variables declared in `this` sub-problem. This also corresponds to the number of columns in this sub-block of the full constraint matrix.
- `uint numLocalCons`
  - the number of local constraints declared in `this` sub-problem. This also corresponds to the number of rows in this sub-block of the full constraint matrix.
- `unit nchild()`
  - returns the number of children nodes of `this` expanded model tree node. This also gives the number of sub-problems.
- `ExpandedModel* child(int i)`
  - returns the ExpandedModel pointer which points to the i^{th} sub-problem of the problem represented by `this` expanded model tree node.

6.3 Variable and Constraints Information

Each expanded model tree node can have a list of local variables and constraints belonging to this sub-problem (or master problem if it the root node). The variables and constraints may have static information which stays the same during the solution process. This information could be the upper and lower bounds or ranges, and the constraint and variable names, etc. The interface methods for retrieving information related to variables and constraints are also implemented in `ExpandedModel` class.

It worth noting that interface methods for retrieving variable and constraint names are only for solution reporting or debugging purposes. Those names are dynamically generated but not stored because of memory consideration (illustrated in Section 4.4.1).

In certain cases, the modeller may want to specify default primal and dual variable values in the model as the starting point for solver iteration. PSMG's solver interface also provides methods to obtain these default values.

The interface methods for retrieving above static information are listed below.

- `void get_var_ranges(double** lower, double** upper)`
  - set the `lower` and `upper` pointers to the corresponding lower and upper array. The dimension of the double arrays pointed by `lower` and `upper` need to be agreed with the number of variables declared in `this` expanded model tree node. Positive or negative infinity value is set for a variable that does not present a upper or lower ranges.
- `void get_cons_bounds(double** lower, double** upper)`
  - set the `lower` and `upper` pointers to the corresponding lower and upper array. The dimension the double arrays pointed by `lower` and `upper` need to be agreed with the number of constraint declared in this expanded model tree node. Positive or negative infinity value is set for a variable that does not present a upper or lower bounds.
- `void get_vars_names(vector<string>& vnames)`
  - set the names of the variables declared at this expanded model tree node to the string vector `vnames`.
- `void get_cons_names(vector<string>& cnames)`
  - set the names of the constraints declared at this expanded model tree node to the string vector `cnames`.
- `void get_default_x0(double** x0)`
  - set `x0` pointer to the default value of the primal variable declared at `this` expanded model tree node. The dimension of the array pointed by `x0` needs to be agreed with the number of variables declared in `this` expanded model tree node. `NaN` is set whenever a default value is not presented for a primal variable.
- `void get_default_y0(double** y0)`
  - set `y0` pointer to the default value of the dual variable declared at `this` the expanded model tree node. The dimension of the array pointed by `y0` needs to be agreed with the number of constraints declared in `this` expanded model tree node. `NaN` is set whenever a default value is not presented for a dual variable.

At the end of current solver iteration, the primal and dual variable values are very likely to be updated by the solver, so that the solver can request PSMG to evaluate the function and derivative using the updated values for next iteration. The interface methods for solver to update the primal and dual variable values are also implemented in `ExpandedModel` class and listed below.

- `void update_primal_x(double* vals)`
  - update the primal variable values. The dimension of `vals` is equal to the number of variables declared in `this` expanded model tree node.
- `void update_dual_y(double* vals)`
  - update the dual variable values. The dimension of `vals` is equal to the number of constraints declared in `this` expanded model tree node.

6.4 Local and Distributed Interface Methods

Each expanded model tree node has a list of local variables and constraints declared in it. The constraint function in an expanded model tree node can be declared using not only the local variables but also the variables from its ancestor and descendant nodes, whereas the object function part in an expanded model tree node can be declared using the local variables and variables from its descendant nodes only. These may be allocated on different processors. Therefore, it is required to provide values of variables declared in the dependent nodes before invoking any interface methods for function or derivatives evaluation.

By considering the *block separability assumption* in PSMG's problem formulation, the *dependent node set* is decided in accordance with the type of problem being modelled and the actual interface method being called. PSMG offers two types of interfaces (*local* and *distributed*) for three types of problems (LP, QP and NLP problem). Each type of interface has methods for evaluating entities such as constraint or objective function value, objective gradient vector and blocks of the Jacobian or Hessian matrix based on the problem type. Before calling an interface method for evaluating an entity, the solver first is required to retrieve the corresponding *dependent node set* and to make necessary inter-process communication work so that the variable values from the *dependent node set* are locally available. Then the solver can safely invoking the corresponding interface method.

The *local* interface offered in PSMG allows each above mentioned entity to be evaluated on one processor by a single call to a corresponding method. However, the *dependent node set* could contain every node from the expanded model tree when using *local* interface method, that is to say variable values of the full problem may need to be accessed locally. Therefore the inter-process communication cost and memory usage could be significant for problem with a large number of variables. For example, evaluation of the link constraint values in the root node may require variable values of the entire problem to be locally accessible using the *local* interface method.

By taking further consideration of the additively separable expression for the above mentioned entity, we can derive a more efficient interface type, namely the *distributed* interface. The *distributed* interface methods evaluate only each separable part in the entity requested by the solver. Therefore, the *dependent node sets* are computed differently from the ones used in *local* interface methods. This will be illustrated in later section in accordance with the interface methods. The final result can be obtained by a *MPI Reduce* operation to sum over the value return for each separable part on parallel processors. The *distributed* interface can maximise the parallelism for function and derivative evaluation. It can also reduce the cost of communication and memory usage for problem variable values. The *distributed* interface methods are further demonstrated in rest of this chapter.

6.4.1 Inter-process Communication

Local Interface

Generally speaking, the local interface method calling sequence can be summarised as following.

- Solver requests the *dependent node set* from PSMG according to the interface method to be called.
- Solver makes inter-process communication to make variable values from the dependent node set locally available.
- Solver calls the PSMG's interface method to evaluate the requested entity.

As we can see that the *local* interface method requires communication of the variable values from its corresponding *dependent node set* before the method is invoked.

Distribute Interface

The *distributed* interface calling sequence can be summarised as following.

S.1 Parallel solver processes request the dependent node set from PSMG according to the interface method to be called.

S.2 Each solver process makes necessary communication efforts to make variable values from the dependent node set locally available before invoking the interface methods.

S.3 Each separable parts of the entity is computed by the *distributed* interface methods.

S.4 The final result should be computed by the *MPI Reduce* operation on the corresponding parallel communicator.

It is worth noting that the communication cost (in **S.2**) can be eliminated by allocating variables of the parent node of the expanded model tree on every processes that have the children nodes allocated. This allocation scheme guarantees each separable part of the requested entity can be evaluated on a local process using the *distributed* interface method. OOPS [21] is one of the solver enforced this parallel allocation scheme. The communication work (in **S.4**) requires a parallel communicator to be created. At this time, PSMG does not takes the parallel process allocation results from the solver, therefore the communicator creation should be taking care of when linking PSMG with a parallel solver. Moreover, PSMG could implement helper methods to create such communicator given that the solver provides the process allocation results to PSMG.

6.4.2 Interface Method Summary

A summary of those interface methods is given in Table 6.1, where each row gives the type of the interface methods implemented for one of the problem types that can be modelled using PSMG at this time. The *Jacobian block* column lists the evaluation routines for the Jacobian matrix of a problem's constraints. The Hessian of the Lagrangian matrix is evaluated for NLP problem, whereas the Hessian matrix of the objective function is evaluated for QP problem. They correspond to the *Hessian block* column. In the rest of this chapter, we shall explain the interface methods from Table 6.1 according to the problem types. Without loss of generality, we use the two level problem (in 6.2) as an example to explain PSMG's solver interface methods.

	Interface Methods				
Problem type	**Jacobian block**	**Hessian block**	**constraint value**	**objective value**	**objective gradient**
LP	Same	×	L,D	L,D	Same
QP	Same	Same	L,D	L,D	L,D
NLP	L,D	L,D	L,D	L,D	L,D

Table 6.1: Summary of the local and distributed interface methods for LP, QP and NLP problems. In this table, "L" donates the *local* interface method is implemented for a problem type. "D" means the *distributed* interface is implemented. "Same" means the interface methods are the same for both *local* and *distributed* implementations. This happens when the entity to be evaluated does not depend on the variable values. "×" means the evaluation routine for a problem type is not meaningful.

6.5 LP Problem Interface

In a LP problem, the general two level formulation given in (6.2) can be simplified to (6.3), where $c_i^j \in \mathbb{R}^{p_i \times q_j}$ and $f_j \in \mathbb{R}^{q_j}$. p_i is the number of constraints in sub-problem i, for $i \in \{1, \cdots, n\}$. p_0 is the number of constraints in master problem. q_j is the number of variables in sub-problem j, for $j \in \{1, \cdots, n\}$. q_0 is the number of variables in master problem. The corresponding tree representation for this two level LP problem is given in Figure 6.3. This tree structure is the same as the expanded model tree.

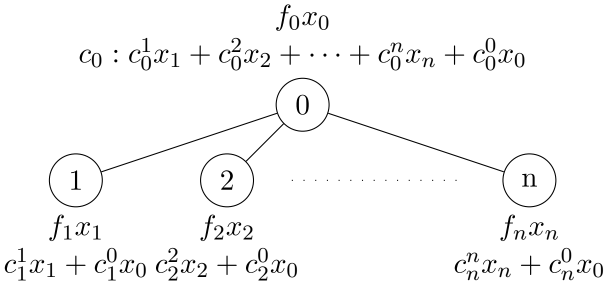


Figure 6.3: A general two level structured problem.

$$
\begin{aligned}
\min \quad & f_1 x_1 && +f_2 x_2 + \cdots && +f_n x_n + f_0 x_0 && && \text{(6.3a)}\\
\text{s.t.} \quad & c_1^1 x_1 && && + c_1^0 x_0 && \le b_1 && \text{(6.3b)}\\
& && c_2^2 x_2 && + c_2^0 x_0 && \le b_2 && \text{(6.3c)}\\
& && \ddots && && \vdots && \text{(6.3d)}\\
& && && c_n^n x_n + c_n^0 x_0 && \le b_n && \text{(6.3e)}\\
& c_0^1 x_1 && +c_0^2 x_2 + \cdots && +c_0^n x_n + c_0^0 x_0 && \le b_0 && \text{(6.3f)}
\end{aligned}
$$

6.5.1 Constraint Matrix Evaluation

PSMG's formulation guarantees the constraint matrix for LP problem demonstrates a double bordered block-angular structure. The corresponding constraint matrix structure for the two-level problem (Figure 6.3) is presented in Figure 6.4.

It is worth noting that blocks in the constraint matrix are identified by the constraints declared at one expanded model tree node indicating the rows, and variables declared at the other expanded model node indicating the columns. Therefore, we should define the following terms to ease our discussion.

- *row-node* – an expanded model tree node that indicates the rows in a Jacobian block.
- *col-node* – an expanded model tree node that indicates the columns in a Jacobian block.

Thus, each pair *(row-node,col-node)* uniquely identifies a block in the Jacobian matrix. It is to note that if *row-node* and *col-node* represent the same node in the expanded model tree, the pair *(row-node,col-node)* identifies a diagonal block in the Jacobian matrix.

We can also observe that the pair *(row-node,col-node)* identifies an non-zero block if and only if *row-node* and *col-node* are on the same *branch*, which means there is a path on the tree started from root that passes both nodes. Therefore, by using the expanded model tree structure, we can easily focus the evaluation of the non-zero blocks of the constraint matrix.

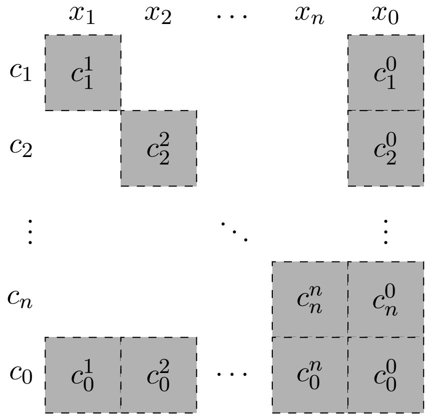


Figure 6.4: Structure of Jacobian matrix for a general two LP problem.

The pairs (i, i), $\forall i \in \{0, \cdots, n\}$ represent the corresponding diagonal blocks in the Jacobian matrix. The pairs *(*$0, i$*)*, $\forall i \in \{0, \cdots, n\}$ represent the bottom border blocks in the Jacobian matrix. And the pairs $(i, 0)$, $\forall i \in \{0, \cdots, n\}$ represent the right-hand side border blocks in the constraint matrix.

Because the constraint matrix is constant and does not depend on the problem variable values, the *dependent node set* for calling this interface method is an empty set. The *local* and *distributed* interface methods are the same for LP problem.

PSMG offers two interface methods for evaluating a block of the constraint matrix, one for returning the sparsity structure of the block, the other one for evaluating the value of the entries. The sparsity structure of a matrix block is returned using an *indicator matrix* object, namely `indicator_matrix` which is illustrated below.

Indicator Matrix
The indicator matrix represents the sparsity pattern of a value matrix. Its underlying data structure uses two integer array `row_index` and `col_start`, where `row_index` stores the row indicies of each nonzero, and `col_start` stores the index of `row_index` array which start a column of the value matrix. This is similar to the *compressed column storage* but without value array.

PSMG returns the block matrix using a sparse matrix storage type, named `col_compress_matrx`. `col_compress_matrix` is an wrapper object, and its underlying data structure uses *compressed column storage* from the Boost ublas library [66]. These two interface methods are declared in the `ExpandedModel` class. They both take an expanded model tree node `emcol` as an input parameter, where `this` expanded model tree node represents *row-node* and `emcol` represents *col-node*. The `this` pointer represents the current object in an Object-Oriented programming context. The two interface methods are listed below.

- `uint nz_cons_jacobs_lp_qp(ExpandedModel* emcol, indicator_matrix& im)`
  - returns the number of non-zero elements in the constraint matrix block identified by the constraints in `this` node and the variables in `emcol` node. This method also fills an indicator matrix `im` to demonstrate the sparsity structure of the block matrix.
- `void cons_jacobs_lp_qp(ExpandedModel* emcol, col_compress_matrix& m)`
  - evaluates the constraint matrix block identified by the constraints in `this` node and the variables in `emcol` node. The result is filled in a sparse matrix m.

6.5.2 Constraint Function Evaluation

A constraint function declared in an expanded model tree node could use variables from both its ancestor nodes and descendant nodes. In particular, the linking constraints declared at the root node could reference every variable from the entire problem. Therefore, the *local* interface method for constraint function evaluation could require all the problem variable values to be accessible locally on one processor. However, with the consideration of the additively separable structure in the constraint formulation, we can produce the *distributed* interface method for evaluating the constraint functions.

Local interface

To ease our discussion, we should now define the following two function terms as listed below.

- *anc* – takes an expanded model tree node as input and returns its ancestors in a set.
- *des* – takes an expanded model tree node as input and returns its descendants in a set.

The *dependent node set* for evaluating constraint function declared at an expanded model tree node i can be evaluated as $anc(i) \cup des(i) \cup \{i\}$. Variable values should be provided by the solver using interface method `update_primal_x` at each corresponding expanded model tree node. Then the solver can employ the *local* interface method to compute values of the constraint functions declared in expanded model tree node i. The `local` interface method for constraint function evaluation is listed below.

- `void cons_feval_local(double* vals)`
  - computes values of the constraint function declared at `this` expanded model tree node, and sets the result values to the double array `vals`. The dimension of `vals` is equal to the number of constraints declared at this expanded model tree node.

Distributed interface

Because the variables in LP problem are always additively separable with each other, the *dependent node set* in this case contains only the node itself. In most case, the variables are allocated on the calling process. Using *distributed* interface method, PSMG can evaluate each of the separable parts of the constraint function declaration in an expanded model tree node (*i.e.* $c_i^T x_i$), and those separable parts can be computed in parallel.

For example, to evaluate the constraint declared at an expanded model tree node i, the separable part of the constraint is identified by the variable in one of the expanded model tree nodes in the set $anc(i) \cup des(i) \cup \{i\}$. Therefore, we can apply the following steps using *distributed* interface to evaluate constraint declared in the expanded model tree node i.

S.1 evaluating the separable part of constraint $v_j, \forall j \in anc(i) \cup des(i) \cup \{i\}$

S.2 obtaining the constraint value $v = \sum_j v_j$

S.1 is the parallel step, and the *dependent node set* for evaluating each v_j is $\{j\}, \forall j \in anc(i) \cup des(i) \cup \{i\}$. Evaluation of each v_j can be done in parallel on the process where each node j is allocated. **S.2** is the communication step, where the constraint function values are obtained by a *MPI Reduce* or *AllReduce* operation with the corresponding parallel communicator.

The *distribute* method maximizes the parallelism for constraint function evaluation and also saves potential memory used for storing variable values from other parallel processors. The *distributed* interface method for constraint function evaluation is listed below.

- `void cons_feval_dist(ExpandedModel* emcol, double* vals)`
  - computes the separable part of the constraint functions in `this` expanded model tree node. The separable part is identified by the variables declared in expanded model tree node, `emcol`. The resulting values are placed into the double array `vals`. The dimension of the double array `vals` is equal to the number of constraints in `this` expanded model tree node.

To illustrate the *distributed* interface, let us consider evaluating the linking constraint declared at root node in Figure 6.3. The constraint can be expressed as $\sum_{i=0}^{n} c_0^i x_i$.

S.1 evaluate $c_0^i x_i$ by calling $\texttt{node}_0\texttt{.cons\_feval\_dist(}i\texttt{, v}_i\texttt{)}$, $\forall i \in \{0, \cdots n\}$.

S.2 obtain the constraint value $v = \sum_{i=0}^{n} v_i$.

S.1 is the parallel step. Before invoking the interface method in **S.1** on each parallel process, the solver need to provide values of the variables in the dependent node set, $\{i\}$. Communication effort may be needed if the values are not available locally. The constraint function value is obtained in **S.2**, where a *MPI Reduce* operation may be invoked with the corresponding parallel communicator to compute, $v = \sum_{i=0}^{n} v_i$.

6.5.3 Objective Function Evaluation

In problems modelled using PSMG, each expanded model tree node may contribute an additively separable part in the overall objective function. The part of the objective function declared in an expanded model tree node can only use variables from this node itself for LP problems.

Local interface

To evaluate the objective function using *local* interface method, the variable values should be provided for every expanded model tree node where an objective function part is declared;hence, the *dependent node set* may include every nodes in the expanded model tree in the worst case. Once the solver provides values of the variables in the corresponding *dependent node set*, the *local* interface method can be called to evaluate the objection function.

This *local* interface method is implemented as a *static* method in `ExpandedModel` class and listed below. *static* keyword has the same meaning as it is in a Object-Oriented programming context.

- `double& obj_feval_local(double& oval)`
  - computes the value of the objective function in the full problem, and sets the objective function value to `oval`. The method also returns the value of the objective function.

Distributed interface

By considering the objective function as a linear combination of the parts declared in each expanded model tree node, the *distributed* interface method can be implemented for objective function evaluation.

Using the *distributed* method for objective function evaluation, PSMG computes only the value of the objective function part declared in one expanded model tree node. These additively separable parts declared at every expanded model tree node can be evaluated in parallel. Once they are evaluated, *MPI Reduce* or *AllReduce* operation can be invoked to sum them together to produce the overall objective function value. Similar to the *distributed* method for constraint function evaluation, the *dependent node set* for objective function evaluation only contains this node itself.

The *distributed* interface method again maximizes the parallelism for objective function evaluation and reduces communication and memory cost related for gathering all the variable values locally. This *distributed* interface is implemented as a member method in `ExpandedModel` class and listed below.

- `double& obj_feval_dist(double& oval)`
  - computes the separable part of the objective function declared at this expanded model tree node. The result is assigned to oval and returned.

To demonstrate the *distributed* interface method, we should consider to evaluate the objective function in the two level LP problem formulation in (6.3). The objective function can be expressed using $\sum_{i=0}^{n} f_i x_i$. Each of the $oval_i = f_i x_i$ can be evaluated using the *distributed* interface method by calling `node`$_i$`.obj_feval_dist(oval`$_i$`)` $\forall i \in \{0, \cdots, n\}$. The overall objective value (*i.e.* $oval = \sum_{i=0}^{n} oval_i$) can be evaluated using a *MPI Reduce* operation with the corresponding parallel communicator.

6.5.4 Objective Gradient Evaluation

The objective function is declared in its separable part in each expanded model tree node, therefore the objective gradient is composed of several sub-vectors. Each of these sub-vectors can be evaluated separately in parallel. Because the objective gradient vector is constant and does not depend on the problem variable values, the *dependent node set* is an empty set for this interface method. The *local* and *distributed* interface methods for objective gradient evaluation are the same for an LP problem.

For example, let us consider the objective gradient for the two level LP problem (6.3). Its structure is presented in Figure 6.5. The objective gradient vector is composed of a list of sub-vectors: f_0, f_1, f_2, and $\cdots, f_n$. Each f_i can be evaluated separately on each expanded model tree node $i \in \{0, \cdots, n\}$.

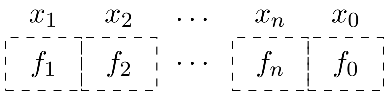


Figure 6.5: The objective gradient for a general two level LP problem.

The interface method for evaluating a sub-vector of the objective gradient is declared in `ExpandedModel` class and listed below.

- `void obj_grad_c_lp(double* vals)`
    - sets the values of gradient sub-vector to the double array `vals`. The sub-vector is evaluated according to the objective function part declared in `this` expanded model node. The dimension of `vals` is needed to agree with the number of variables declared at this expanded model tree node.

6.6 QP Problem Interface

In a QP problem, the general two level formulation given in (6.2) can be simplified to (6.4), where $c_i^j \in \mathbb{R}^{p_i \times q_j}$ and $f_j \in \mathbb{R}^{q_j}$. p_i is the number of constraints in sub-problem i, for $i \in \{1, \cdots, n\}$. p_0 is the number of constraints in master problem. q_j is the number of variables in sub-problem j, for $j \in \{1, \cdots, n\}$. q_0 is the number of variables in master problem. g_i represents a quadratic function, for $i \in \{0, \cdots, n\}$.

Comparing to the LP problem, the QP problem formulation has additional quadratic terms in the objective function part modelled at each expanded model tree node. These quadratic terms are the cross products of the variables declared at this expanded model tree node with respect to the variables declared at itself or its ancestor nodes. In the two-level general QP problem formulation, there is one additional quadratic term $g_i(x_i, x_0)$ in the objective function part for each sub-problem i, $\forall i \in \{1, ...n\}$. $g_0(x_0)$ is the quadratic term that is modelled in master problem. The corresponding tree representation is given in Figure 6.6.

$$
\begin{array}{rllllr}
\min & f_1 x_1 & +f_2 x_2 + \cdots & +f_n x_n + f_0 x_0 + & & (6.4a)\\
 & g_1(x_1, x_0) & +g_2(x_2, x_0) + \cdots & +g_n(x_n, x_0) + g_0(x_0) & & (6.4b)\\
\text{s.t.} & c_1^1 x_1 & & + c_1^0 x_0 & \leq 0 & (6.4c)\\
 & & c_2^2 x_2 & + c_2^0 x_0 & \leq 0 & (6.4d)\\
 & & \ddots & & \vdots & (6.4e)\\
 & & & c_n^n x_n + c_n^0 x_0 & \leq 0 & (6.4f)\\
 & c_0^1 x_1 & +c_0^2 x_2 + \cdots & +c_0^n x_n + c_0^0 x_0 & \leq 0 & (6.4g)
\end{array}
$$

Because LP and QP problems have the exactly same formulation for their constraint formulation, the interface methods for evaluating Jacobian matrix blocks and constraint function values are the same. Thus, the discussion of the corresponding interface methods in LP problem can be applied to QP problem.

In the rest of this section, we discuss the interface methods specifically related to QP problems.

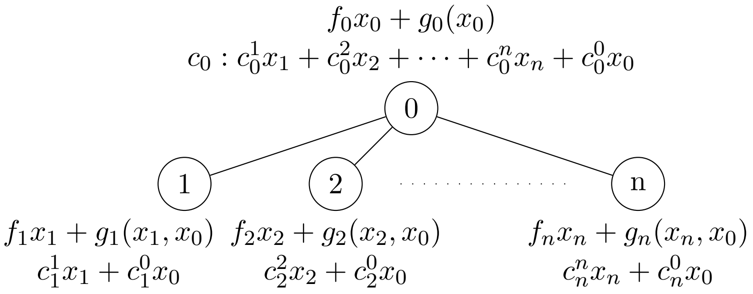


Figure 6.6: A general two level structured problem.

6.6.1 Hessian Matrix Evaluation

Optimization solvers may request the Hessian matrix to be evaluated for QP problems. PSMG's formulation for QP problem also guarantees the objective Hessian matrix demonstrates a double bordered angular-block structure (or nested structure for problems with more than two levels).

For example, the objective Hessian matrix structure for the two-level QP problem formulation in 6.4 presents a double bordered block-angular structure as illustrated in Figure 6.7. The evaluation expression of each non-zero block is also annotated in Figure 6.7.

The objective Hessian matrix is constant and does not depend on problem variable values for QP problems. The *dependent node set* for invoking the Hessian matrix evaluation routine is an empty set. Therefore, the *local* and *distributed* interface methods for Hessian matrix evaluation are the same.

$$\begin{array}{c|ccccc}
 & x_1 & x_2 & \cdots & x_n & x_0 \\
\hline
x_1 & \frac{\partial^2 g_1}{\partial x_1^2} & & & & \frac{\partial^2 g_1}{\partial x_0 \partial x_1} \\
x_2 & & \frac{\partial^2 g_2}{\partial x_2^2} & & & \frac{\partial^2 g_2}{\partial x_0 \partial x_0} \\
\vdots & & & \ddots & & \vdots \\
x_n & & & & \frac{\partial^2 g_n}{\partial x_n^2} & \frac{\partial^2 g_n}{\partial x_0 \partial x_n} \\
x_0 & \frac{\partial^2 g_1}{\partial x_0 \partial x_1} & \frac{\partial^2 g_2}{\partial x_0 \partial x_2} & \cdots & \frac{\partial^2 g_n}{\partial x_0 \partial x_n} & \frac{\partial^2 g_0}{\partial x_0^2}
\end{array}$$

Figure 6.7: The objective Hessian matrix structure for a general two level QP problem.

PSMG also offers two interface methods for evaluating a block of the objective Hessian matrix in QP problem, one for computing the sparsity structure of this block matrix, and the other one for evaluating the matrix entries. An expanded model tree node is taken by both methods as an input parameter. Together with the expanded model tree node represented by `this` pointer, the two expanded model tree nodes indicate the block in the Hessian matrix to be evaluated. These two interface methods are listed below.

- `uint nz_obj_hess_qp(ExpandedModel* emcol, indicator_matrix& im)`
  - returns the number of non-zero entries in a block of the objective Hessian matrix. This block is identified by taking the second order derivative of the non-linear constraint declared in `this` node with respect to the variables declared in `this` and `emcol` expanded model tree node. This method also fills an indicator matrix `im` to demonstrate the sparsity structure of the block matrix.
- `void obj_hess_qp(ExpandedModel* emcol, col_compress_matrix& m)`
  - evaluates a block of the objective Hessian matrix. This block is identified by taking the second order derivative of the non-linear constraint declared in `this` node with respect to the variables declared in `this` and `emcol` expended model tree node. The results are filled in the column compress matrix type `col_compress_matrix`.

6.6.2 Objective Function Evaluation

Similar to the LP problem, the objective function for QP problems can be evaluated using either *local* or *distributed* interface methods.

Local interface

Similar to the *local* interface method discussed for LP problems, the *local* interface method for QP problems requires the variable values to be provided for every expanded model tree node where an objective function part is declared. Therefore, the discussion of the *local* interface method from LP problem can also be applied here. In fact, this interface method has the same method signature as its counterpart for LP problems.

Distributed interface

The *distributed* interface allows the objective part declared in an expanded model tree node to be evaluated separately. The objective function part declared in each expanded model tree node could include a quadratic term. This quadratic term could be the cross product with the variables from any of its ancestor nodes (but not from its descendants), therefore the *dependent node set* for evaluating the objective part declared in an expanded model tree node i can be represented as $anc(i) \cup \{i\}$. Variable values in this dependent node set need to be provided before calling the *distributed* interface method to evaluate the part objective function declared in the expanded model tree node i. Each objective function part can be evaluated in parallel. The overall objective function value can be obtained by a *MPI Reduce* operation with the corresponding parallel communicator. The method signature for this *distributed* interface method is the same as the one for LP problem.

Let us use the two-level QP problem formulation in (6.4) as an example to demonstrate the use of the *distributed* interface method for evaluating the objective function. The corresponding calling sequence is listed below.

S.1 computes the value of the objective function part in each expanded model tree node i, $\forall i \in \{0, \cdots, n\}$. *i.e.* `node`$_i$`.obj_feval_dist(`$v_i$`)`

- $v_i = \begin{cases} v_0 = f_0 x_0 + g_0(x_0), & \text{if } i = 0 \\ v_i = f_i x_i + g_i(x_i, x_0), & \text{otherwise} \end{cases}$

S.2 obtains the objective function value of the problem v.

- $v = \sum_{i=0}^{n} v_i$

In the above calling sequence, **S.1** is the parallel step. Before invoking the interface method in **S.1** on each parallel process, the solver need to provide values of the variables in the dependent node set, $anc(i) \cup \{i\}$. Communication effort may be requested if the values are not available locally. The objective function value is obtained in **S.2**, where a *MPI Reduce* operation may be invokded with the corresponding parallel communicator.

6.6.3 Objective Gradient Evaluation

Similar to the LP problem, PSMG's solver interface methods evaluate the objective gradient sub-vector in accordance with variables declared in an expanded model tree node as well. In QP problems, variables declared in one expanded model tree node can be used in the objective function part declaration of this node itself or its decedents, therefore the objective function parts declared in more than one expanded model tree nodes may be needed to be considered for evaluating one sub-vector of the objective gradient. In general, the set of expanded model tree nodes to be considered for evaluating an objective gradient sub-vector identified by variables declared in an expanded model tree node i is $\{i\} \cup des(i)$.

The objective gradient structure for the two level QP problem in formulation (6.4) is given in Figure 6.8. Evaluation of the objective gradient sub-vector for x_0 requires knowledge of all objective function parts declared in the entire expanded model tree.

$$\begin{array}{c|c|c|c|c|}
x_1 & x_2 & \cdots & x_n & x_0 \\
\hline
\frac{\partial f_1}{\partial x_1} + \frac{\partial g_1}{\partial x_1} & \frac{\partial f_2}{\partial x_2} + \frac{\partial g_2}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} + \frac{\partial g_n}{\partial x_n} & \frac{\partial f_0}{\partial x_0} + \sum_{i=0}^{n} \frac{\partial g_i}{\partial x_0} \\
\hline
\end{array}$$

Figure 6.8: The objective gradient for a general two level QP problem.

Local interface

The *local* interface evaluates an objective gradient sub-vector in a single call. The *dependent node set* for evaluating one such sub-vector identified by variables declared in an expanded model tree node i can be represented as $anc(i) \cup des(i) \cup \{i\}$. The solver is required to provide values for variables declared in this set before invoke the interface method. However, the actual expression to evaluate this sub-vector is created by taking a linear combination of the objective function parts declared from node set $\{i\} \cup des(i)$.

The *local* interface method for evaluating a gradient sub-vector is listed below.

- `void obj_grad_qp_nlp_local(double* vals)`
  - evaluates the objective gradient sub-vector for variable declared in `this` expanded model tree node. The result is assigned to the double array `vals` whose dimension is equal to the number of variables in this expanded model tree node.

To illustrate this interface method, let us consider the objective gradient sub-vector identified by the x_0 variable in the two level general QP problem formulation (6.4). The *dependent node set* is evaluated as $anc(0) \cup des(0) \cup \{0\}$. The work involved in using the *local* interface method for evaluating this sub-vector is summarized below.

S.1 solver provides variable values for dependent node set, $\{0, \cdots, n\}$.

S.2 solver invokes the interface method to evaluate this sub-vector v_0 (*i.e.* by calling `node`$_0$`.obj_grad_qp_nlp_local(`v_0`)`.

In above steps, communication work may be required in **S.1** to make the required variable values locally available to PSMG. In **S.2**, PSMG creates a temporary expression by taking linear combination of the objective function parts declared in node set $\{0\} \cup des(0)$. Then, the sub-vector of the objective gradient is evaluated by taking the partial derivative respect to x_0 on this temporary expression (*i.e.* $\sum_{i=0}^{n} \frac{\partial g_i}{x_0} + f_0 x_0$), and the resulting sub-vector is written to v_0.

As we can see that evaluation of the objective gradient sub-vector for variables declared in the root node may require to access values of the entire problem's variables using the *local* interface method.

Distributed interface

Using the *local* interface method may require accessing the entire problem's variable values in the worst case. By considering the objective function as a linear combination of the parts declared in each expanded model tree node, the *distribute* interface method can be implemented for objective gradient evaluation. We can derive the following steps of using the *distributed* interface method to evaluate an objective gradient sub-vector for variables declared in an expanded model tree node i.

S.1 evaluating the gradient vector v_j of the part of objective function declared in node j, $\forall j \in des(i) \cup \{i\}$.

S.2 obtain the gradient sub-vector $v = \sum_j v_j$.

S.1 is the parallel step, where each v_j can be evaluate using the *distributed* interface method one parallel processes. The *dependent node set* for evaluating each v_j is $anc(j) \cup \{j\}$, which is less than the dependent node set required by the *local* interface method on one processor. **S.2** is the communication step, where the sub-vector of the objective gradient is obtained by a *MPI Reduce* or *AllReduce* operation with the corresponding parallel communicator.

Therefore, the *distributed* interface method maximizes the parallelism and reduces amount of memory used for storing variable values on one processor. This *distributed* interface is listed below.

- `void obj_grad_qp_nlp_dist(ExpandedModel* decol,double* vals)`
  - evaluates a separable part of the objective gradient sub-vector for variable declared at `this` expanded model tree node. The result is assigned to the double array `vals` whose dimension is equal to the number of variables in this expanded model tree node. The separable part is indicated by `decol` node.

For example, evaluating the objective gradient sub-vector for variable declared at the master problem (*i.e.* x_0) in the two level QP problem in formulation (6.4) can be achieved by the following steps of using *distributed* interface method.

S.1 computes each separable part v_i, $\forall i \in \{0, \cdots, n\}$
`node`$_i$`.obj_grad_nlp_dist(0,`v_i`)`

- $v_i = \begin{cases} f_0 & \text{if } i = 0 \\ \frac{\partial g_i}{\partial x_0} & \text{otherwise} \end{cases}$

S.2 obtain the sub-vector v.

- $v = \sum_{i=0}^{n} v_i$

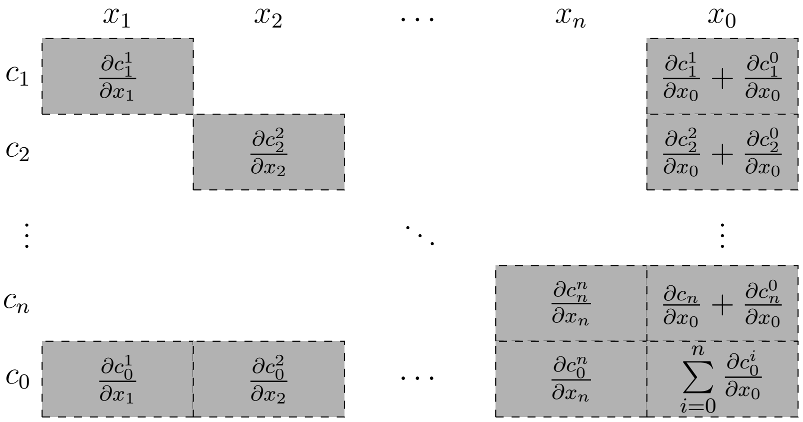
In above example, **S.1** can be performed in parallel. Before invoking the interface method in **S.1** on each parallel process, communication effort may be required for the solver to provide values of the variables in the dependent node set, $anc(i) \cup \{i\}$. The sub-vector is obtained in **S.2**, where a *MPI Reduce* operation may be invoked with the corresponding parallel communicator.

6.7 NLP Problem Interface

The interface methods for function and derivative evaluations are more complicated for NLP problems. Without loss of generality, we use the general two-level problem formulation in (6.2) to demonstrate the interface methods for NLP problem. The tree structure of this problem can be viewed in Figure 6.1. We also assume each of the functions f_i, c_i, and c_0^i are nonlinear functions, $\forall i \in \{0, \cdots, n\}$.

6.7.1 Jacobian Matrix Evaluation

The Jacobian matrix of a NLP problem modelled using PSMG also demonstrates a double bordered block-angular structure. The structure can be nested for problems with more than two levels. The Jacobian matrix structure for the two-level NLP problem in formulation (6.2) is given in Figure 6.9, where each of the non-zero blocks is annotated with the corresponding evaluation expression. Similar to LP problems, each block in the Jacobian matrix can also be identified by a pair *(row-node, col-node)*.



$$
\begin{array}{c|ccccc}
 & x_1 & x_2 & \cdots & x_n & x_0 \\
\hline
c_1 & \frac{\partial c_1^1}{\partial x_1} & & & & \frac{\partial c_1^1}{\partial x_0} + \frac{\partial c_1^0}{\partial x_0} \\
c_2 & & \frac{\partial c_2^2}{\partial x_2} & & & \frac{\partial c_2^2}{\partial x_0} + \frac{\partial c_2^0}{\partial x_0} \\
\vdots & & & \ddots & & \vdots \\
c_n & & & & \frac{\partial c_n^n}{\partial x_n} & \frac{\partial c_n}{\partial x_0} + \frac{\partial c_n^0}{\partial x_0} \\
c_0 & \frac{\partial c_0^1}{\partial x_1} & \frac{\partial c_0^2}{\partial x_2} & \cdots & \frac{\partial c_0^n}{\partial x_n} & \sum_{i=0}^{n} \frac{\partial c_0^i}{\partial x_0}
\end{array}
$$

Figure 6.9: Structure of Jacobian matrix for a general two level NLP problem.

To ease our discussion, we should define a level function as below.

- *level* – takes an expanded model tree node as input and returns its level in the tree, where the level of root node is 0.

In order to evaluate a non-zero block in the Jacobian matrix identified by pair *(row-node,col-node)*, PSMG may need to access the variable values of the dependent nodes. There are two cases to be considered to decide a set of dependent nodes.

Case 1: *level(row-node)=level(col-node)*

When the pair (*row-node,col-node*) indicates a non-zero block in the Jacobian matrix, the condition of this case also implies *row-node* and *col-node* are the same node in the expanded model tree, since sibling nodes can not be on the same tree branch. The pair (*row-node,col-node*) indicates a diagonal block in the Jacobian matrix. For example, pairs *(*0,0*), (*1,1*), (*2,2*),* $\cdots$*, (n,n)* indicate the diagonal blocks in the Jacobian matrix of the two-level NLP problem in Figure 6.1.

Because constraints declared in node i can reference the variables in i or i's ancestor and descendant nodes, in order to evaluate the diagonal Jacobian block identified by pair (i, i), the variable values from node set $anc(i) \cup des(i) \cup \{i\}$ may be accessed.

For example, to evaluate the bottom right corner block indicated by the pair *(0,0)* in Figure 6.9, the values of variables declared in the node set $\{0, 1, \cdots, n\}$ may be accessed.

Case 2: *level(row-node)≠level(col-node)*

In this case, the pair *(row-node,col-node)* identifies either a block in the bottom border of the Jacobian matrix (if *level(row-node)<level(col-node)*) or a block in the right-hand side border of the Jacobian matirx (if *level(row-node)>level(col-node)*). This case implies that one of the *row-node* or *col-node* is the ancestor node of the other one. For example, pairs *(0,1)*, *(0,2)*, $\cdots$, *(0,n)* identify the bottom border blocks in the Jacobian matrix of the two-level NLP problem in Figure 6.1, and *(1,0)*, *(2,0)*, $\cdots$, *(n,0)* identify the right-hand side border blocks in the Jacobian matrix.

Let i be the node *row-node* or *col-node* which has the greater *level* number. The variables declared in node set $anc(i) \cup des(i) \cup \{i\}$ should be considered for evaluating a Jacobian block in this case. For example, to evaluate the bottom block in the Jacobian matrix identified by the pair *(*$0, 1$*)*, the dependent node set is evaluated to $\{0, 1\}$.

We can now discuss the *local* and *distributed* interface methods for Jacobian matrix evaluation of the NLP problems.

Local interface

The *local* interface methods allow each Jacobian matrix block to be evaluated with a single call. Therefore, it requires the solver to provide the variable values from the *dependent node set* before invoking the *local* interface method.

To use the *local* interface method for evaluating the Jacobian block identified by pair (i, j), where $i, j \in \mathcal{T}$, and $\mathcal{T}$ is the set of nodes in the expanded model tree, the *dependent node set* should be represented as $anc(k) \cup des(k) \cup \{k\}$, where

$$k = \begin{cases} i & \text{if } \ level(i) \geq level(j) \\ j & \text{otherwise} \end{cases}$$

The worst case could happen when evaluating the bottom right corner block in the Jacobian matrix. In this case, the *dependent node set* may include every node in the expanded model tree.

For example, to evaluate the bottom right block (identified by pair $(0, 0)$), the *dependent node set* is evaluated as $anc(0) \cup des(0) \cup \{0\} = \{0, 1, \cdots, n\}$. Thus, the entire problem's variable values could be needed. Therefore depending on the parallel allocation strategy, a solver may need to communicate those variable values and provide the values to PSMG. PSMG also needs to allocate memory to store those values. Then the solver can invoke the *local* interface methods for evaluating the block of the Jacobian matrix.

There are three interface methods declared for evaluating the Jacobian matrix block, two for computing the sparsity structure of this block, the other for evaluating this block matrix. These interface methods for evaluating the block of the Jacobian matrix are listed below.

- `uint nz_cons_jacobs_nlp_local(ExpandedModel* emcol)`
  - returns the number of non-zero elements in the block matrix of the Jacobian matrix identified by pair (`this,emcol`).
- `uint nz_cons_jacobs_nlp_local(ExpandedModel* emcol, indicate_matrix& im)`
  - same as the method above. In addition, this method also fills an indicator matrix `im` to demonstrate the sparsity structure of the block matrix.
- `void cons_jacobs_nlp_local(ExpandedModel* emcol, col_compress_matrix& m)`
  - evaluates the the Jacobian matrix block identified by pair (`this, emcol`). The results are filled into the column compress matrix type `col_compress_matrix`.

Distributed interface

Using the *local* interface methods may require values of the entire problem's variables to be available on one processor for evaluating a block in the Jacobian matrix in the worst case. However, by taking consideration of the separability structure within each block in the Jacobian matrix, we can design the *distributed* interface methods for evaluating each separable part in the block. The number of dependent nodes for each calling of the *distributed* interface method is less than the number of dependent nodes required for invoking the *local* interface method. Moreover, each separable part can also be evaluated in parallel. The *distributed* interface methods are listed below.

- `nz_cons_jac_nlp_dist(ExpandedModel* emcol, ExpandedModel* decol, indicator_matrix& im)`
  - returns the number of non-zero element in the separable part within a block in the Jacobian matrix identified by pair `(this, emcol)`. The separable part is identified by `decol`. `decol` is one of the node in set $anc(i) \cup des(i) \cup \{i\}$, where $i = \texttt{this}$ if $level(\texttt{this}) \geq level(\texttt{emcol})$,or $i = \texttt{emcol}$ otherwise. The method also sets the sparsity structure to the indicator matrix `im` for the separable part.
- `cons_jac_nlp_dist(ExpandedModel* emcol, ExpandedModel* decol, col_compress_matrix& m)`
  - computes one of the separable parts within a block in the Jacobian matrix block identified by pair `(this, emcol)`. The separable part is identified by `decol`. `decol` is one of the node in set $anc(i) \cup des(i) \cup \{i\}$, where $i = \texttt{this}$ if $level(\texttt{this}) \geq level(\texttt{emcol})$, or $i = \texttt{emcol}$ otherwise.

We should now demonstrate using of the *distributed* interface methods for evaluating the Jacobian matrix block identified by pair *(i, j)*, where $i, j \in \mathcal{T}$, and $\mathcal{T}$ is the node set of the expanded model tree.

Let

$$k = \begin{cases} i & \text{if } \ level(i) \geq level(j) \\ j & \text{otherwise} \end{cases}$$

Then, the following steps can be applied for evaluating the Jacobian matrix block (i, j)

S.1 evaluating the separable part of this block m_p,
$\forall p \in anc(k) \cup des(k) \cup \{k\}$
i.e. by invoking $\texttt{node}_i\texttt{.cons\_jac\_nlp\_dist(j,p,}m_p\texttt{)}$

S.2 obtaining this block matrix m
$m = \sum_p m_p$

S.1 is the parallel step, where each m_p can be evaluated in parallel using the *distributed* interface. The *dependent node set* for invoking each interface method to evaluate m_p is $anc(k) \cup anc(p) \cup \{k\}$. This set can be further simplified to $anc(p) \cup \{p\}$ if $level(p) \geq level(k)$, or $anc(k) \cup \{k\}$ otherwise. Depending on the solver's parallel allocation scheme, inter-process communication work may be required to make the values of the variables in the dependent node set available locally before the interface method is invoked to evaluate each separable part on a process. **S.2** is the communication step, where the Jacobian block is obtained by a *MPI Reduce* or *AllReduce* operation with the corresponding parallel communicator.

The dependent nodes set for evaluating each m_p using the *distributed* interface is less than the dependent node set for evaluating m directly using the *local* interface method. Therefore, using the *distributed* interface for evaluating blocks of the Jacobian matrix could maximize the parallelism and also reduce the amount of memory needed for storing variable values on one processor.

For example, let us considering the right bottom block identified by the pair $(0, 0)$ in the Jacobian matrix of the two level NLP problem in Figure 6.9. The block can be evaluated by following sequence using the *distributed* interface method.

S.1 computing each separable part of the matrix block, m_p for all $p \in \{0, \cdots n\}$
`node`$_0$`.cons_jac_nlp_dist(0,`p`, `m_p`)`

- $m_p = \frac{\partial c_0^p}{\partial x_0}$, for all $p \in \{0, \cdots n\}$.

S.2 obtaining the matrix block $m = \sum_{p=0}^{n} m_p$.

It is worth noting that **S.1** is the parallel step. Before calling the interface method in **S.1** to evaluate m_p on each parallel process, values of variables from node set $\{0, p\}$, $\forall p \in \{0, \cdots, n\}$ should be available locally.

For computing the sparsity structure of the block identified by the pair *(*0, 0*)*, let us define the binary operator $||$ for the indicator matrix using the component-wised logical *or* operator (*e.g.* $0||1 = 1$). The *nnz* can be obtained by counting the number of non-zero elements the indicator matrix. This sparsity pattern can be evaluated by the following sequence using the *distributed* interface method.

S.1 computing the indicate matrix, m_p for all $p \in \{0, \cdots n\}$
`node`$_0$`.nz_cons_jac_nlp_dist(0,`p`, `im_p`)`

- im_p indicates the sparsity structure of $\frac{\partial c_0^p}{\partial x_0}$, for all $p \in \{0, \cdots n\}$.

S.2 obtaining the sparsity structure of the matrix block $im = im_0||im_1...||im_n$. The non-zero number $nz = nnz(im)$.

In above steps, **S.1** is the parallel step, and **S.2** is achieved by a *MPI Reduce* or *AllReduce* operation with the corresponding parallel communicator. It is worth noting that variable values are not required for evaluating the sparsity pattern of a block.

6.7.2 Hessian of the Lagrangian Matrix Evaluation

Most optimization solvers for solving NLP problems require the Hessian of the Lagrangian matrix to be evaluated. The Hessian of the Lagrangian matrix also demonstrates a double bordered angular-block structure for the NLP problem modelled in PSMG. The structure can be nested for problems with more than two levels. For the NLP problem in formulation (6.2), the Lagrangian $\mathcal{L}$ is given below.

$$\begin{aligned}\mathcal{L}(x,y) =& f_0(x_0) + \sum_{i=1}^{n} f_i(x_i,x_0) + \sum_{i=1}^{n} y_i c_i^i(x_0,x_i) + \sum_{i=1}^{n} y_i c_i^0(x_0) \\ &+ y_0 \sum_{i=1}^{n} c_0(x_0,x_i) + y_0 c_0^0(x_0)\end{aligned} \tag{6.5}$$

where y_i represents the Lagrangian multipliers for constraints declared at node i. The Hessian of the Lagrangian matrix structure for this two level NLP problem is presented in Figure 6.10. It demonstrates a double bordered angular-block structure, where each of the non-zero block is annotated with the corresponding evaluation expression.

We should now redefine the *row-node* and *col-node* in the following to represent a block in the Hessian of the Lagrangian matrix.

- *row-node* – an expanded model tree node indicating the rows in a block of the Hessian of the Lagrangian matrix.
- *col-node* – an expanded model tree node indicating the columns in a block of the Hessian of the Lagrangian matrix.

Therefore each pair *(row-node,col-node)* uniquely identifies a block in the Hessian of the Lagrangian matrix. Because of the symmetry of the Hessian matrix, the block matrix identified by the pair *(row-node, col-node)* is equal to the transpose of the block matrix identified by pair *(col-node, row-node)* . We can also observe that the pair *(row-node, col-node)* identify a non-zero block if and only if *row-node* and *col-node* on the same branch of the expanded model tree. Therefore, using the expanded model tree structure, we can easily work out the structure of the Hessian of the Lagrangian matrix and focus on the evaluation of the non-zero blocks.

Depending on the relative *levels* of *row-node* and *col-node*, there are three cases to be considered.

Case 1: ***level(row-node)=level(col-node)***
The non-zero blocks identified by the pairs *(row-node,col-node)* are the diagonal blocks in the Hessian of the Lagrangian matrix. *row-node* and *col-node* represent the same node in the expanded model tree. The variables declared in *row-node* (or *col-node*) can be used in the constraint declaration of a node from the set *anc(row-node)* ∪ *des(row-node)* ∪ {*row-node*}. The nonlinear terms for variables in *row-node* node could be presented in the constraint declarations of a node from this set.

Case 2: ***level(row-node)<level(col-node)***
The non-zero blocks identified by these pairs are the bottom border in the Hessian of the Lagrangian matrix. The nonlinear terms respect to variables in *row-node*

$$\begin{array}{c|ccccc}
 & x_1 & x_2 & \cdots & x_n & x_0 \\
\hline
x_1 & y_1 \frac{\partial^2 c_1^1}{\partial x_1^2} + y_0 \frac{\partial^2 c_0^1}{\partial x_1^2} + \frac{\partial^2 f_1^2}{\partial x_1^2} & & & & y_1 \frac{\partial^2 c_1^1}{\partial x_1 \partial x_0} + y_0 \frac{\partial^2 c_0^1}{\partial x_1 \partial x_0} + \frac{\partial^2 f_1}{\partial x_1 \partial x_0} \\
x_2 & & y_2 \frac{\partial^2 c_2^2}{\partial x_2^2} + y_0 \frac{\partial^2 c_0^2}{\partial x_2^2} + \frac{\partial^2 f_2^2}{\partial x_2^2} & & & y_2 \frac{\partial^2 c_2^0}{\partial x_2 \partial x_0} + y_0 \frac{\partial^2 c_0^2}{\partial x_2 \partial x_0} + \frac{\partial^2 f_2}{\partial x_2 \partial x_0} \\
\vdots & & & \ddots & & \vdots \\
x_n & & & & y_n \frac{\partial^2 c_n^n}{\partial x_n^2} + y_0 \frac{\partial^2 c_0^n}{\partial x_n^2} + \frac{\partial^2 f_n}{\partial x_n^2} & y_n \frac{\partial^2 c_n^0}{\partial x_n \partial x_0} + y_0 \frac{\partial^2 c_0^n}{\partial x_n \partial x_0} + \frac{\partial^2 f_n}{\partial x_n \partial x_0} \\
x_0 & y_1 \frac{\partial^2 c_1^1}{\partial x_1 \partial x_0} + y_0 \frac{\partial^2 c_0^1}{\partial x_1 \partial x_0} + \frac{\partial^2 f_1}{\partial x_1 \partial x_0} & y_2 \frac{\partial^2 c_2^2}{\partial x_2 \partial x_0} + y_0 \frac{\partial^2 c_0^2}{\partial x_2 \partial x_0} + \frac{\partial^2 f_2}{\partial x_2 \partial x_0} & \cdots & y_n \frac{\partial^2 c_n^n}{\partial x_n \partial x_0} + y_0 \frac{\partial^2 c_0^n}{\partial x_n \partial x_0} + \frac{\partial^2 f_n}{\partial x_n \partial x_0} & y_0 \sum_{i=0}^{n} \frac{\partial^2 c_0^i}{\partial x_0^2} + \sum_{i=1}^{n} y_i \left(\frac{\partial^2 c_i^i}{\partial x_0^2} + \frac{\partial^2 c_i^0}{\partial x_0^2} \right) + \sum_{i=0}^{n} \frac{\partial^2 f_i}{\partial x_0^2}
\end{array}$$

Figure 6.10: Structure of Hessian of the Lagrangian matrix for a general two level NLP problem.

and *col-node* could present in the constraint declaration of a node from the set *anc(col-node)* $\cup$ *des(col-node)* $\cup$ $\{$*col-node*$\}$.

Case 3: *level(row-node)>level(col-node)*
The non-zero blocks identified by these pairs are the right hand side border in the Hessian of the Lagrangian matrix. The nonlinear terms respect to variables in *row-node* and *col-node* could be presented in the constraint declaration of a node from the set *anc(row-node)* $\cup$ *des(row-node)* $\cup$ $\{$*row-node*$\}$. Because of symmetry in the Hessian matrix, we can interchange *row-node* and *col-node*, and then evaluate the transpose of the block matrix as per our discussion in case 2.

Furthermore, we can condense above three cases. Let us consider a block in the Hessian of Lagrangian matrix identified by a pair (i, j), where $i, j \in \mathcal{T}$, and $\mathcal{T}$ is the node set of the expanded model tree.

Let

$$k = \begin{cases} i & \text{if } \ level(i) \geq level(j) \\ j & \text{otherwise} \end{cases}$$

Then, the nonlinear terms respect to variables from node i and node j could be presented in the constraint declaration of a node from the set $anc(k) \cup des(k) \cup \{k\}$.

Local interface

The *local* interface method evaluates a block in the Hessian of the Lagrangian matrix in a single call. The values of the primal and dual variables declaration in the *dependent node set* should be made available locally before invoking the interface method. The primal and dual variable values can be provide to PSMG using interface methods `update_primal_x` and `update_dual_y` respectively on each of the expanded model tree node.

There are two *local* interface methods declared for evaluating a block in the Hessian of the Lagrangian matrix, one for computing the sparsity structure of this block, the other one for evaluating the block matrix. These interface methods are listed below.

- `uint nz_lag_hess_nlp_local(ExpandedModel* emcol, indicator_matrix& im)`
    - computes the sparsity structure of the block in the Hessian of the Lagrangian matrix. The block is identified by the pair (`this`, `emcol`) of expanded model tree nodes. The sparsity structure is presented by the indicator matrix `im`. This method also returns the number of nonzero elements in this block.
- `void lag_hess_nlp_local(ExpandedModel* emcol, col_compress_matrix& m)`
    - computes the matrix block in the Hessian of the Lagrangian matrix. The block is identified by the pair (`this`, `emcol`) of expanded model tree nodes. The block matrix is filled into the column compressed matrix `m`.

It is worth mentioning that the sparsity structure evaluation of a block does not require the problem variable values.

Assuming the *local* interface method is called for evaluating a block in the Hessian of the Lagrangian matrix identified by a pair (i, j), the cross-product terms that contributed in this block are from the constraint declarations of the node from node set $anc(k) \cup des(k) \cup \{k\}$, where k is defined above. PSMG constructs a temporary expression by taking the linear combination of those cross-product terms and evaluate the entries in this block matrix. The *dependent node set* for evaluation this temporary expression is also $anc(k) \cup des(k) \cup \{k\}$. The values of the primal and dual variables declared in the node from this *dependent node set* should be provided to PSMG before invoking the *local* interface method.

For example, to evaluate bottom right corner block identified by the pair $(0, 0)$ in Figure 6.10, the *dependent node set* is evaluate as $anc(0) \cup des(0) \cup \{0\} = \{0, 1, \cdots, n\}$, the full node set of the expanded model tree (in Figure 6.1). After providing the primal and dual variable values declared in the node of the dependent node set, solver can obtain the required matrix block by calling `node`$_0$`.lag_hess_nlp_local(0,m)`. The result is filled in the sparse matrix object m.

In the worst case (*i.e.* bottom right corner block), using the *local* interface to evaluate a block in the Hessian of the Lagrangian matrix may require values of all the problem's primal and dual variables to be available locally. It could cause a heavy communication and memory cost.

Distribute interface

By considering the additively separable structure within the block, the *distributed* interface methods can be designed for evaluating each separable part in the block. Then, each of the separable part can be evaluated on parallel processes, and the number of dependent nodes for evaluating each separable part is less than the number of dependent nodes for evaluating the full block using *local* interface method. Therefore, the memory used for storing the variable values is also reduced compare to the memory usage for *local* interface method.

There are two *distributed* interface methods declared for evaluating an additively separable part of the block in the Hessian of the Lagrangian matrix, one for computing the sparsity structure of this block, and the other one for evaluating the block matrix.

- `void nz_lag_hess_nlp_dist(ExpandedModel* emcol, ExpandedModel* decol, indicator_matrix& im)`
  - computes the sparsity structure for the separable part in the block in the Hessian of the Lagrangian matrix. The block is identified by pair *(this,emcol)*. The separable part is further indicated by `decol`. The sparsity pattern will be returned in an indicator matrix `im`.
- `void lag_hess_nlp_dist(ExpandedModel* emcol, ExpandedModel* decol, col_compress_matrix& m)`
  - computes a separable part of the block in the Hessian of the Lagrangian matrix. The block is identified by pair *(this,emcol)*. The separable part is further indicated by `decol`. The result will be filled into the column compressed matrix `m`.

We should now demonstrate how to use the *distributed* interface to evaluate a block in the Hessian of the Lagrangian matrix identified by pair (i, j). The nonlinear terms that contribute in this block are from the constraint declarations of the node from node set $anc(k) \cup des(k) \cup \{k\}$, where k is defined above. Instead of building a temporary expression from the nonlinear terms using the node set and evaluating the block locally, the *distributed* methods evaluate these additively separable terms on parallel processes. The work involved in evaluating the block identified by pair (i, j) is summarized below.

S.1 evaluating the separable part of this block m_p
$\forall p \in anc(k) \cup des(k) \cup \{k\}$
i.e. by invoking `node`$_i$`.lag_hess_nlp_dist(j,p,`m_p`)`

S.2 obtaining this block matrix m
$m = \sum_p m_p$

S.1 is the parallel step, where each m_p can be evaluated in paralle using distributed interface. The *dependent node set* for invoking each interface method to evaluate m_p is $anc(k) \cup anc(p) \cup \{k\}$. Similar to the discussion in the Jacobian matrix evaluation for the NLP problem, the set can be further simplified to $anc(p) \cup p$ or $anc(k) \cup \{k\}$. Inter-process communication work may be required to make values of the primal and dual variables in the *dependent node set* available locally before the interface method is invoked on one process. **S.2** is the communication step, where each of the evaluated separable parts are summed together by a *MPI Reduce* or AllReduce operation with the corresponding parallel communicator.

Similar to the argument in the Jacobian matrix evaluation for the NLP problem, the *distributed* interface method for evaluation Hessian of the Lagrangian matrix could maximize the parallelism and also reduce the amount of memory needed for storing variable values on one processor.

To further demonstrate the *distributed* interface methods, let us consider the right bottom corner block identified by the pair $(0, 0)$ in the Hessian of the Lagrangian matrix of the two level NLP problem in Figure 6.10. The block can be evaluated by the following sequence using *distributed* interface method.

S.1 computing each separable part in the matrix block, $m_i, \forall i \in \{0, \cdots, n\}$.
node$_0$.lag_hess_nlp_dist(0,i,m_i)

- $m_i = \begin{cases} y_0 \frac{\partial^2 c_0^0}{\partial x_0^2} + \frac{\partial^2 f_0}{\partial x_0^2}, & \text{if } i = 0 \\ y_0 \frac{\partial^2 c_0^i}{\partial x_0^2} + y_i(\frac{\partial^2 c_i^i}{\partial x_0^2} + \frac{\partial^2 c_i^0}{\partial x_0^2}) + \frac{\partial^2 f_i}{\partial x_0^2}, & \text{otherwise} \end{cases}$

S.2 obtaining the matrix block $m = \sum_{i=0}^{n} m_i$.

In **S.1**, each m_i can be evaluated in parallel given the values of primal and dual variables from node set $\{0, i\}$ is available locally before each calling of `lag_hess_nlp_dist` method, $\forall i \in \{0, \cdots, n\}$. **S.2** is the communication step, where a *MPI Reduce* or *AllReduce* operation is involved to obtain the matrix block.

To demonstrate the sparsity pattern evaluation of the bottom right corner block identified by the pair $(0, 0)$, the following calling sequence can be applied for using *distributed* interface methods. It is worth mentioning that the variable values are not required to evaluate the sparsity pattern.

S.1 computing the indicator matrix, im_i for all $i \in \{0, \cdots, n\}$
node$_0$.nz_hess_nlp_dist(0,i,im_i)

- im_i indicates the sparsity structure of $y_0 \frac{\partial^2 c_0^0}{\partial x_0^2} + \frac{\partial^2 f_0}{\partial x_0^2}$, if $i = 0$,
- otherwise im_i indicates the sparsity structure of $y_0 \frac{\partial^2 c_0^i}{\partial x_0^2} + y_i(\frac{\partial^2 c_i^i}{\partial x_0^2} + \frac{\partial^2 c_i^0}{\partial x_0^2}) + \frac{\partial^2 f_i}{\partial x_0^2}$.

S.2 computing $im = im_0 || im_1 \cdots || im_n$.
The non-zero number $nz = nnz(im)$.

6.7.3 Constraint Function Evaluation

PSMG can also evaluate the constraint function of an NLP problem using either the *distributed* or *local* interface method.

Local interface

Because the constraint declared in an expanded model tree node i can reference variables from i itself and its ancestor or descendant nodes, the *dependent node set* for evaluating the constraint functions in node i using *local* interface method is *anc(i)* ∪ *des(i)* ∪ $\{i\}$. Therefore, in the worst case the whole problem variable values are needed for constraint function evaluation (*e.g.* when i is the root node of the expanded model tree).

The method has the same signature as described in LP problem section. Possible communication work may be needed to make values of the variables from the *dependent node set* available locally before invoking the interface method for constraint function evaluation.

Distributed interface

Considering the additively separable structure in the constraint function, we can design the *distributed* interface method for evaluating the constraint function.

The method has the same signature as described in LP problem section. We can derive the following steps for using the *distributed* interface method to evaluate constraint function declared in an expanded model tree node i.

S.1 compute the separable part v_j,
$\forall j \in anc(i) \cup des(i) \cup \{i\}$

S.2 obtain the constraint function value v
$v = \sum_j v_j$

S.1 is the parallel step, where each of the separable part $v_j, \forall j \in anc(i) \cup des(i) \cup \{i\}$ can be evaluated on separate process. The dependent node set for computing each v_j is $anc(i) \cup anc(j) \cup \{i\}$. The set can be further simplified to $anc(i) \cup \{i\}$ if $level(i) \geq level(j)$, $anc(j) \cup \{j\}$ otherwise. Values of variables for the dependent node set should be provided before invoking the interface method for evaluating each separable part in the constraint function. Then a *MPI Reduce* or *AllReduce* operation is performed on the corresponding parallel communicator to obtain the constraint function values. The dependent node set for calling *distributed* interface method is less than the node set used in *local* interface method. The *distributed* interface method again maximizes the parallelism and reduce amount of memory used for storing variable values on one processor.

For example, evaluating the constraint functions in the master problem (at node 0) of the two level NLP problem in formulation (6.2) can be achieved by the following steps using *distributed* interface method.

S.1 compute each separable part v_i , $\forall i \in \{0, \cdots, n\}$
`node`$_0$`.cons_feval_dist(`i`,`v_i`)`

- $v_i = \begin{cases} c_0^0(x_0) & \text{if} \quad i = 0 \\ c_0^i(x_0, x_i) & \text{otherwise} \end{cases}$

S.2 obtain the constraint function value $v = \sum_{i=0}^{n} v_i$

S.1 is the parallel step to evaluate each v_i. Communication work may be incurred for making variable values available locally before invoking the interface method. The resulting constraint value v is obtained in **S.2**, where a *MPI Reduce* or *AllReduce* operation may be performed with the corresponding parallel communicator.

6.7.4 Objective Function Evaluation

Similar to the LP and QP problem, the objective function for the NLP problem can also be evaluated using either *distributed* or *local* interface method.

Local interface

Similar to the discussion for LP and QP problems, the *local* interface method for the NLP problem requires variable values to be provided for every expanded model tree nodes where an objective function part is declared. Therefore, the discussion of the *local* interface method from the LP and QP problem also applies here. The *local* interface method for objective function evaluation of the NLP problem has the same signature as the one declared for the LP and QP problems. The *dependent node set* contains all the expanded model tree node that has an objective function part declaration.

Distributed interface

The *distributed* interface method evaluates the objective function part declared in an expanded model tree node. Since the objective function part declared in an expanded model tree node can use variables from this node itself and its ancestors, the *dependent node set* for evaluating the objective function part declared in an expanded model tree node i can be represented as *anc(i)* $\cup \{i\}$.

The *distributed* interface method for objective function evaluation of the NLP problem also has the same signature as the one declared for the LP and QP problems.

The following calling sequence can be applied for using the *distributed* method interface to evaluate the objective function for an NLP problem.

S.1 computing each objective function part o_i, $\forall i \in \mathcal{T}$, where $\mathcal{T}$ is the node set of an expanded model tree.
i.e. by calling `node`$_i$`.cons_feval_dist(`o_i`)`

S.2 obtaining the objective function value $o = \sum_{i \in \mathcal{T}} o_i$

S.1 is the parallel step, where the *dependent node set* for evaluating each o_i is $anc(i) \cup \{i\}$. Values of variables from this node set should be provided before each calling of the interface method to evaluate o_i. Then, a *MPI Reduce* or *AllReduce* operation is performed with the corresponding parallel communicator for obtaining the overall objective function value v in **S.2**. Again, the *distributed* interface method for objective function evaluation maximizes the parallelism and reduce the amount of memory for storing variable values on one processor.

For example, to evaluate the objective function value of the two level NLP problem in formulation (6.2) can be achieved by the following steps of using *distributed* interface method.

S.1 computes each objective function part o_i, $\forall i \in \{0, \cdots, n\}$
`node`$_i$`.obj_fval_nlp_dist(`o_i`)`

- $o_i = \begin{cases} f_0(x_0) & \text{if} \quad i = 0 \\ f_i(x_0, x_i) & \text{otherwise} \end{cases}$

S.2 obtains the objective function value, $o = \sum\limits_{i=0}^{n} o_i$

The *dependent node set* in **S.1** for evaluating o_i is $\{0\}$ for $i = 0$, $\{0, i\}$ otherwise.

6.7.5 Objective Gradient Evaluation

Similar to the QP problem, the objective gradient for the NLP problem is also evaluated by the sub-vectors in accordance with variables declared in an expanded model tree node using either *local* or *distributed* interface method. In fact, the method signatures for objective gradient evaluation of the NLP problem are the same as their counterparts for the QP problem. Therefore, the discussion from the objective function evaluation section for the QP problem can be easily generalised for handling the NLP problem.

6.8 Summary

PSMG can be used to model three type of problems (*i.e.* LP, QP and NLP problem). In this chapter, we have discussed the PSMG's solver interface for retrieving the problem structure and evaluating a list of entities from the problem model. Those entities to be evaluated using PSMG's interface methods could be constraint function values, objective function values, blocks in the Jacobian matrix, blocks in the Hessian of the Langrangian matrix, blocks in the objective Hessian matrix and objective gradient sub-vectors depending on the problem type.

The interface methods for structure retrieval are the same for the three type of problems, whereas the interface methods for evaluating those entities mentioned above could be different according to the problem type. Because of the similarity in the LP and QP problem formulation, they share most of the interface methods. However, the NLP problem formulation is the most general case handled by PSMG.

PSMG also offers two types of interfaces for evaluating the entities mentioned above, namely the *local* and *distributed* interfaces. We have discussed them separately for three problem types. The *local* interface allows the required entity to be evaluated in a single call to the corresponding interface method. However, we demonstrate that the *local* interface methods may have a significant cost for inter-process communication and memory usage in the worst case scenario. On the other hand the *distributed* interface considers the additively separable structure in the entity to be evaluated. The *distributed* interface allows each separable part in each entity to be evaluated separately in parallel. Therefore, it reduces the amount of memory usage and inter-process communication needed for problem variable values. The *distributed* interface can maximize parallelism for evaluating each entity according to different problem types. However, if using the *distributed* interface method, a corresponding parallel communicator should be created to perform a *MPI Reduce* or *AllReduce* operation to sum over the values returned on each parallel processor. This could introduce more linkage work between PSMG and the parallel solver.

Chapter 7

Linking With Parallel Solvers

Different parallel solver algorithms may employ different parallel allocation strategies by considering the load balancing and the inter-process communication cost. Because of the solver driven work assignment approach taken in PSMG's solver interface design, it is quite convenient to link PSMG with different parallel solver algorithms. In this chapter, we discuss the work involved in linking PSMG's solver interface with two types of parallel solver algorithms that were presented in Chapter 3.

7.1 Structure Exploiting Interior Point Method

The solvers that implements the structure exploiting interior point method usually builds the augmented system matrix internally using the structure from A and Q matrices, where A represents the Jacobian matrix of constraints, and Q represents the Hessian of the objective function (for QP problems) or the Lagrangian (for NLP problems) (see Section 3.1.2).

Therefore, PSMG is responsible for passing the structure information of A and Q matrices to the solver. The expanded model tree built after the structure building stage in PSMG can fulfil this requirement, as each block from the A and Q matrices can always be identified by the intersection of two expanded model tree nodes (as illustrated in Chapter 6).

7.1.1 Building Matrix Structure

To explain the algorithm for building the matrix structure from the expanded model tree, we should introduce the following data type abstractions in an Object-Oriented programming context as illustrated in Figure 7.1. These data type abstractions are enough for us to describe algorithm for building a nested double bordered block-angular matrix using the expanded model tree from PSMG. Without loss of generality, we assume both A and Q are both double bordered block-angular matrices. Data abstractions for other matrix structures can be easily added as a sub-type of the `Matrix` interface.

- `Matrix`

  - the matrix interface.

- `MatrixSparse`
  - represents a simple matrix block whose value can be retrieved using PSMG's solver interface methods. This matrix block is always created by two expanded model tree nodes, where one node represents the row and the other represents the column in its containing matrix.
- `DbBordDiagMatrix`
  - represents a double bordered block-angular matrix (possibly nested).
- `DenseMatrix`
  - represents a bottom or right-hand-side border matrix block.

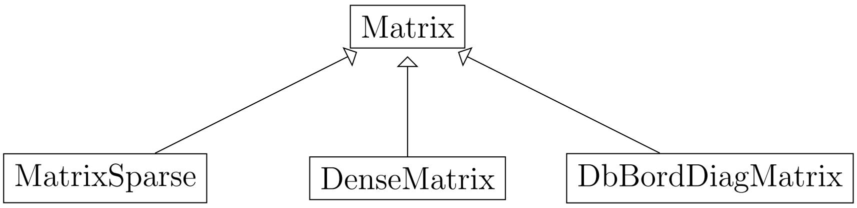


Figure 7.1: A potential matrix data type hierarchy diagram adopted by solver implementation.

For each of the data types above (except the `Matrix` interface), we should also define their corresponding constructors as following.

- `MatrixSparse(ExpandedModel row,ExpandedModel col)`
  - Creates a matrix block using two expanded model tree nodes. The dimension of this matrix block is obtained by `row.numLocalCons` × `col.numLocalVars` if it is a sub-block of the constraints Jacobian matrix. If this matrix is a sub-block of the Hessian (of the objective function or Lagrangian), the dimension is obtained by `row.numLocalVars` × `col.numLocalVars`.
- `DbBordDiagMatrix(int nchild,Matrix* D,Matrix* B,Matrix* R)`
  - Creates a double bordered block-angular matrix, where `D`, `B` and `R` are arrays of `Matrix` pointers for the diagonal blocks, the bottom border blocks, and the right-hand-side blocks respectively. `nchild` is the size of the `B` and `R` arrays. The dimension for the `D` array is equal to `nchild+1`, because the right bottom corner block is also store in the `D` array. This matrix can have nested structure when the `Matrix` pointer in `D` is a `DbBordDiagMatrix` type. In this case, the pointers from corresponding index of the `B` and `R` `Matrix` arrays should be `DenseMatrix` type.
- `DenseMatrix(int nblocks,Matrix* B)`
  - Creates a complex matrix block of the bottom or right-hand-side border. Each `Matrix` pointers in the `B` array represents a sub-block of this

complex border matrix. It is used to represent the border blocks of matrices with nested block-angular structure. The dimension of the `B` array is equal to `nblocks`.

The pseudocode for building the nested double bordered block-angular matrix is given in Algorithm 7.1.1. The main procedure of this algorithm takes the root of an expanded model tree as the input parameter and returns the double bordered block-angular matrix constructed. The main procedure in this algorithm also calls other two procedures for building the bottom and right-hand-side border matrix blocks correspondingly. It is worth noting that this algorithm and its sub-procedures are called recursively to build the corresponding matrix structure for nested structured problem. Moreover, Algorithm 7.1.1 can be applied for constructing both A and Q matrices.

Figure 7.2 demonstrates the matrix constructed using an expanded model tree of a general two level structured problem in Figure 6.1. Each nonzero block of the matrix is indicated by a pair of expanded model tree nodes.

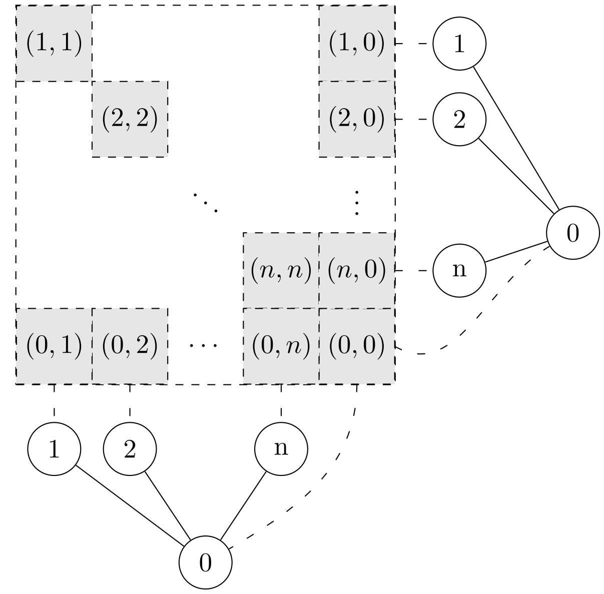


Figure 7.2: The double bordered angular-block structure matrix created using an expanded model tree of a two level problem. Each nonzero block of the matrix is indicated by a pair of expanded model tree nodes.

Algorithm 7.1.1: CREATEDBBORDDIAGM(*ExpandedModel node*)

```
main
  $Matrix\ M;$
  $int\ nchild =\ node.nchild();$
  if $nchild == 0$
    then $M = new\ MatrixSparse(node, node);$
    else
      $Matrix\ D[nchild + 1];$
      $Matrix\ B[nchild];$
      $Matrix\ R[nchild];$
      for $i \leftarrow 0$ to $nchild$
        do
          $D[i] \leftarrow$ CREATEDBBORDDIAGM$(node.child(i))$
          $B[i] \leftarrow$ CREATEBOTTOMBLOCK$(node, node.child(i))$
          $R[i] \leftarrow$ CREATEBORDERBLOCK$(node.child(i), node)$
      $M = new\ DbBordDiagMatrix(nchild, D, B, R);$
  return $(M)$

procedure CREATEBOTTOMBLOCK($ExpandedModel\ row, ExpandedModel\ col$)
  comment: Create the bottom block.$Matrix\ M;$

  $int\ nchild =\ row.nchild();$
  if $nchild == 0$
    then $M = new\ MatrixSparse(row, col)$
    else
      $Matrix\ B[nchild + 1]$
      for $i \leftarrow 0$ to $nchild$
        do {$B[i] \leftarrow$ CREATEBOTTOMBLOCK$(row, col.nchild(i))$
      $B[nchild] = new\ MatrixSparse(row, col)$
      $M = new\ DenseMatrix(nchild + 1, B)$
  return $(M)$

procedure CREATEBORDERBLOCK($ExpandedModel\ row, ExpandedModel\ col$)
  comment: Create the right-hand-side border block.$Matrix\ M;$

  $int\ nchild = col.nchild();$
  if $nchild == 0$
    then $M = new\ MatrixSparse(row, col)$
    else
      $Matrix\ R[nchild + 1]$
      for $i \leftarrow 0$ to $nchild$
        do {$R[i] \leftarrow$ CREATEBOTTOMBLOCK$(row.nchild(i), col)$
      $R[nchild] =\ new\ MatrixSparse(row, col)$
      $M = new\ DenseMatrix(nchild + 1, R)$
  return $(M)$
```

7.1.2 Building Vector Structure

The solver in this category may also need to know the structure of the primal and dual vectors of the structured problem. The primal vector structure can be used for storing primal variables and their bounds, objective gradient, etc. The dual vector structure could be used for storing constraint values and ranges, etc. This structure information can be easily obtained by tracing the expanded model tree. To explain the algorithm used for building such structured vectors, we should introduce the following data type abstractions in Object-Oriented programing context as illustrated in Figure 7.3.

- `Vector`
  - The vector interface.
- `VectorSparse`
  - Represents a sparse vector whose value can be retrieved by PSMG's solver interface methods.
- `VectorDense`
  - Represents the vector with nested structure.

Again, the corresponding constructors for each of the data types above (except `Vector` interface) are listed below.

- `VectorSparse(ExpandedModel node)`
  - Creates a sparse vector using the expanded model tree node. The dimension of this vector is equal to `node.numLocalVars`, if the vector has primal structure. If the vector has a dual structure, the dimension is equal to `node.numLocalCons`.
- `VectorDense(int nvec, Vector* V)`
  - Creates a dense vector to represent the (nested) structure of either the primal or dual vector. The dimension of the array `V` is equal to `nvec`.

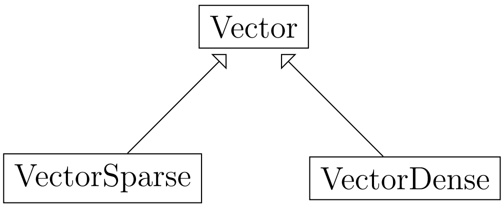


Figure 7.3: A potential vector data type hierarchy diagram adopted by solver's implementation.

The pseudocode for building the structured vector is given in Algorithm 7.1.2. This algorithm takes the root node of an expanded model tree as the input parameter and returns the structured vector. This algorithm applies for constructing vectors in either primal or dual structure. It is worth noting that this algorithm is called recursively to create the corresponding structured vector for nested structured problems.

Figure 7.4 demonstrates the primal and dual vector structures constructed using an expanded model tree of a general two level structured problem in Figure

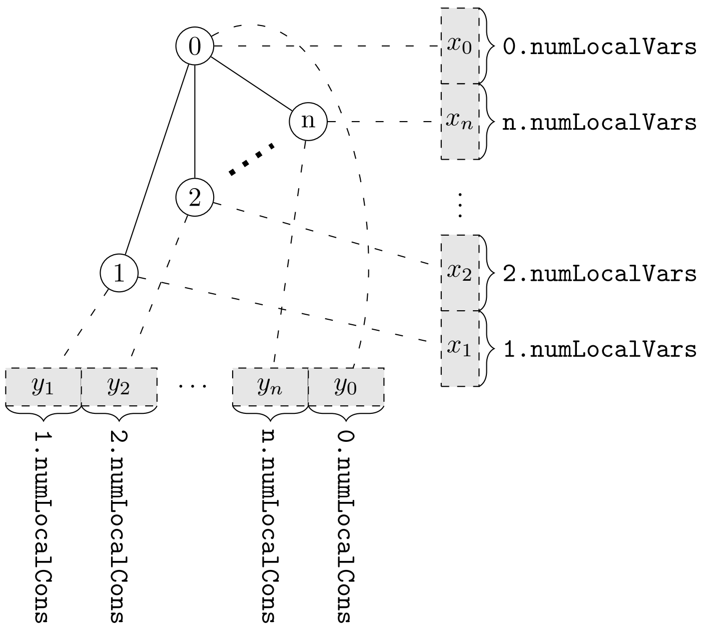


Figure 7.4: This figure demonstrates the expanded model tree and its corresponding vector structure. The dimension of each sub-vector can be obtained by accessing the interface properties of the expanded model tree node.

6.1. The dimension of each sub-vector is indicated on the figure.

Algorithm 7.1.2: CREATESTRVECTOR(*ExpandedModel node*)

```
comment: Creating a structured vector
main
  int nchild = node.nchild();
  Vector v;
  if nchild == 0;
    then v = new SpraseVector(node);
    else { Vector subv[nchild];
           for i ← 1 to nchild
             do { subv[i] = CREATESTRVECTOR(node.child(i));
           v = new DenseVector(nchild, subv);
  return (v)
```

7.1.3 Parallel Process Allocation

Once PSMG passes the structures of the required matrices and vectors to the solver, the solver allocates the matrix and vector blocks among parallel processes. This parallel allocation is performed on the augmented system matrix, and the related linear algebra background is explained in Chapter 3. The primal and dual structured vectors are also allocated correspondingly among parallel processes.

Figure 7.5 demonstrates the parallel allocation result by OOPS solver on n parallel processes of a general two level problem with n sub-problems. Each block in the augmented system matrix is composed of corresponding blocks from Q and A matrices, and each sub-vector is also composed of the corresponding sub-vectors from primal and dual vectors. In order to minimize the inter-communication cost, OOPS allocates the right bottom corner block on every process. The primal and dual sub-vectors of the top level problem are also allocated on all processes. The numbers annotated on top of each block in Figure 7.5 indicate the processes where the block or sub-vector is allocated.

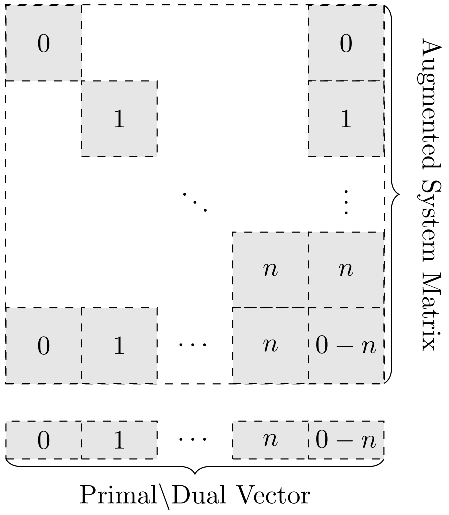


Figure 7.5: Parallel processor allocation of the rearranged augmented matrix and corresponding vector in OOPS.

7.1.4 Parallel Problem Generation

According to the discussion in the previous section, each matrix block is created by the `MatrixSparse` constructor with two expanded model tree nodes, and each sub-vector is created by the `VectorSparse` constructor with one expanded model tree node. It is sufficient to evaluate each matrix block and sub-vector using the interface methods in the `ExpandedModel` class (as illustrated in Chapter 6).

Call-back interface design with OOPS

While building the matrix structure, PSMG also sets up a `CallBackInterfaceType` object for each matrix block. At a later time, when PSMG is asked to evaluate the matrix block by invoking the callback function, PSMG can decided which action to take according to the content of `CallBackInterfaceType` instance. The declaration of the `CallBackInterfaceType` is given in Listing 7.1.

The possible actions that PSMG could take are the following.

- returns number of non-zero elements in the block requested;
- returns an indicator matrix in compressed column storage (CCS) format to indicate the sparsity pattern of the matrix sub-block;

- returns the actual block matrix in CCS format.

In LP and QP problems, the matrices A and Q stay the same at each iteration. Each block matrix is evaluated in two calls to PSMG. The first call to evaluate the number of non-zero elements, and solver uses this number to allocate memory. Then, the matrix block is evaluated in the second call.

For NLP problems, OOPS requires the indicator matrix of a block to be evaluate first, to set up the indices for symbolic factorisation. The indicator matrix can be requested by invoking the call-back function with `nlp_str = true`, whereas the actual block matrix is evaluated when setting `nlp_str = false`.

```
class OOPSBlock {
public:
  ExpandedModel *emrow;    // Expanded Model giving row information
  ExpandedModel *emcol;    // Expanded Model giving col information
  OOPSBlock(ExpandedModel *rowmod, ExpandedModel *colmod);
};
typedef struct {
  /* INPUT: */
  void *id;      // points to a OOPSBlock
  bool nlp_str; // evaluates the indicator matrix if true,
                //otherwise evaluates the actual block matrix.
  /* OUTPUT */
  int nz;
  int *row_nbs;
  int *col_nbs;
  int *col_beg;
  int *col_len;
  double *element;
} CallBackInterfaceType;
```

Listing 7.1: Call-back interface in OOPS

Distributed vs. Local interface

With the above call-back interface design, it is straightforward to link the PSMG's *local* interface methods with OOPS. Each block or sub-vector can be generated locally on the calling process, as long as the variable values from the dependent node set are provided before each invoking of the PSMG's interface method. It is also obvious that each block in the A and Q matrices for LP and QP problem are constant and independent from the problem's variable values. Only the constraint and objective function values are dependent on the current problem's variable values. However the constraint and objective function values are not required by OOPS from PSMG. Therefore the *local* interface methods are sufficient to offer an efficient parallel problem generation strategy for LP and QP problems.

For NLP problem, OOPS need to provide variable values for the dependent node set before invoking PSMG's interface methods. Therefore inter-process communication costs for sending and receiving variable values may incur. We have linked PSMG's local interface methods with OOPS for demonstrating PSMG's performance. The benchmark results are given in next chapter. It is worth mentioning that the *distributed* interface could offer better performance for generating a general NLP problems. However it requires considerable amount of linkage work to take advantage of the *distributed* interface methods.

7.2 Benders' Decomposition

PSMG can also be linked with parallel solver that implements a decomposition algorithm, such as Benders' Decomposition. We have already discussed the approach of using Bender's decomposition scheme for solving a general nonlinear structured problem in Chapter 3. A Benders' Decomposition solver requires PSMG to provide the problem in terms of master and sub-problem structure. The expanded model tree constructed after PSMG finishes the structure building stage can be used for this purpose. The root node of the expanded model tree represents the master problem, and each child node represents one of the sub-problems. The number of constraints and variables for master and sub-problems can be easily accessed using the interface properties from the `ExpandedModel` class as illustrated in Chapter 6.

In Section 3.2.2, we have suggested a possible parallel allocation strategy for a Benders' Decomposition solver which is to allocate each sub-problem on one process, and the master problem on every processes. Then, the sub-problems can be solved in parallel. The master problem could also be solved using a corresponding parallel solver algorithm. Since values of the master problem variables are needed to solve each sub-problem at every iteration, allocating the master problem on every processes could efficiently reduce inter-process communication cost for values of the variables declared in the master problem. For example, the value of x_0 is needed to solve the sub-problem whose formulation is given in (3.27).

7.2.1 Parallel problem generation

Once the solver finishes allocation of the problems on parallel processors, the solver can request PSMG to generate the required function and derivative values for the master and sub-problems. While linking PSMG with a decomposition solver, similar callback interface as illustrated in 7.1.4 can also be designed for accessing PSMG's solver interface methods conveniently.

Now, let us demonstrate how PSMG's solver interface is called to generate the master and sub-problems in formulation (3.28) and (3.27) respectively. The master and sub-problems are derived by applying the Benders' Decomposition scheme on a two level general structure problem in formulation (3.25). The corresponding expanded model tree for this problem is illustrated in Figure 3.2.

Sub-problem generation

To solve the i^{th} sub-problem in formulation (3.27), the solver is likely to request the following entities to be evaluated by PSMG.

- constraint function value, $v = c_i^i(x_0, x_i)$
  - *i.e.* node$_i$.`cons_feval_dist(0,v)`
- right hand side value, $v = c_i^0(x_0)$
  - *i.e.* node$_0$.`cons_feval_dist(`i`,v)`

- objective function value, $v = f_i(x_0, x_i)$
  - *i.e.* `node`$_i$`.obj_feval_dist(v)`
- Jacobian matrix, $m = \frac{\partial c_i^i}{\partial x_i}$
  - *i.e.* `node`$_i$`.cons_jacobs_nlp_local(0, m)`
- objective gradient vector, $v = \frac{\partial f_i}{\partial x_i}$
  - *i.e.* `node`$_i$`.obj_grad_qp_nlp_local(v)`
- Hessian of the Lagrangian, $m = \frac{\partial^2 f_i}{\partial x_i^2} + y_i \frac{\partial^2 c_i^i}{\partial x_i^2}$
  - *i.e.* `node`$_i$`.lag_hess_nlp_local(0,m)`

It is worth mentioning that the *dependent node set* for each above PSMG interface calls is evaluated as $0, i$. By allocating the master problem variables on every process, each sub-problem can be generated in parallel without any inter-process communication.

Master problem generation

PSMG's solver interface methods can be used to generate the following entities of the master problem.

- part of the objective function value, $v = f_0(x_0)$
  - *i.e.* `node`$_0$`.obj_feval_dist(0,v)`
- objective gradient vector, $v = \frac{\partial f_0}{\partial x_0}$
  - *i.e.* `node`$_0$`.obj_grad_qp_nlp_local(v)`
- Hessian of $m = f_0(x_0)$
  - *i.e.* `node`$_i$`.lag_hess_nlp_dist(0,0,m)`

While doing the linkage work between a Benders' Decomposition solver and PSMG, v_i and $g_i^{(j)}$ values from each sub-problem i should be communicated to processes where the master problem is solved. Based on the suggested parallel allocation strategy described above, a *MPI AllReduce* operation could be used to make the values available on all parallel processes using *MPI_COMM_WORLD* communicator. If other parallel allocation scheme is used, the corresponding parallel communicator should be created.

Chapter 8

PSMG Performance Benchmark

We have linked PSMG's *local* interface methods with OOPS. In this chapter, we discuss the performance of PSMG's problem generation by problem type. For LP and QP problems, the problem matrices are constant, therefore the solver only requires PSMG to provide the matrix blocks once. On the other hand, the solver is likely to require PSMG to provide values of the problem matrix blocks at every iteration for NLP problems. Additionally, based on the allocation scheme used by OOPS, each matrix block in LP and QP problem can always be evaluated with only local information, whereas evaluation of a matrix block in NLP problems may depend on variable values of other nodes. Therefore, the NLP problem may have different performance results from LP and QP problems.

8.1 LP and QP Problems

8.1.1 Test Problem Sets

We have used three test problem sets for our experiments in this section. The first one is a set of MSND problems (described in Listing 2.1) based on a network of 30 Nodes and 435 Arcs, corresponding to a complete graph. The number of commodities varies between 5 and 50. The number of constraints and variables in these problems increases linearly with the number of commodities. The largest problem of this MSND problem set has over 10.2 million variables and 0.8 million constraints.

The second test problem set is a series of large scale portfolio management problems. We have chosen 10 random stock symbols from Nasdaq and used the Nasdaq-100 as a benchmark to generate a set of 3-stage problem instances for the ALM-SSD problem (described in Listing 2.3) for a range of scenarios and benchmark realisations. The largest problem of this ALM-SSD problem set has over 10.1 million constraints and 20.5 million variables.

The above two are sets of LP problems. The third one is a set of QP problems based on Markovitz mean-variance model [70] (henceforth referred as ALM-VAR). The mathematical formulation is given in (8.1). The sets, parameters and variables represented by $\mathcal{A}, \mathcal{L}, \mathcal{L}_{\mathcal{T}}, \gamma, V_j, p_i, c_i, x^h_{i,j}, x^s_{i,j}, x^b_{i,j}, r_{i,j}$ have the same meaning as the notations used in ALM-SSD formulation. However, this model does not have the second order stochastic dominance constraints. Instead, it maximizes the expectation of wealth μ and its negative variance with risk-aversion parameter ρ. $s(i)$ represents the stage of a node i; hence, $L_{s(i)}$ represents the liability needed to be satisfied at its corresponding stage. The model file for this problem is given in Appendix C. We have generated a set of 3-stage problem instances for this model by choosing 50 random stock symbols and a range of scenarios. The largest problem of this ALM-VAR problem set has over 2.5 million constraints and 7.3 million variables.

$$
\begin{aligned}
\max \quad & \mu - \rho \left[\sum_{i \in \mathcal{L}_T} p_i \left[\sum_{j \in \mathcal{A}} V_j x^h_{i,j} + c_i \right]^2 - \mu^2 \right] \\
\text{s.t.} \quad & (1+\gamma) \sum_{j \in \mathcal{A}} (x^h_{0,j} V_j) + c_0 = I, \\
& c_{\pi(i)} + (1-\gamma) \sum_{j \in \mathcal{A}} (V_j x^s_{i,j}) - L_{s(i)} = c_i + (1+\gamma) \sum_{j \in \mathcal{A}} x^h_{i,j}, \quad \forall i \neq 0, i \in \mathcal{L} \\
& (1 + r_{i,j}) x^h_{\pi(i),j} + x^b_{i,j} - x^s_{i,j} = x^h_{i,j}, \qquad \forall i \neq 0, i \in \mathcal{L}, j \in \mathcal{A} \\
& \sum_{i \in \mathcal{L}_T} p_i (\sum_{j \in \mathcal{A}} V_j x^h_{i,j} + c_i) = \mu
\end{aligned}
\tag{8.1}
$$

8.1.2 Comparison Analysis with SML

The proof-of-concept model generator for SML [1] was implemented as a pre- and post-processor for AMPL; henceforth we use SML-AMLP to refer to it. After analyzing the problem structure as defined by the block-statements, SML-AMPL would create a self-contained sub-model file for each block. Then the sub-model files are processed by AMPL together with the data file to create `*.nl`-files for every block. The solver interface can access these files through the `AmplSolver` library [31] for function and derivative values.

PSMG is designed specifically to model and generate structured problems with an in-memory solver interface design. On the other hand, SML-AMPL has the same design goal but uses AMPL as a backend, and can only run in serial. Therefore, we have compared the serial performance of PSMG to SML-AMPL. The problem generation times of using PSMG and SML-AMPL for generating the problems in MSND test problem set are presented in Table 8.1 and Figure 8.1.

For SML-AMPL we report the time taken up by the pre-processing stage which finishes when all `*.nl`-files are written (Column 7) and the time taken to fill each matrix block in the problem with numerical values (Column 8). The total time for SML-AMPL to generate the problem is listed in Column 6. For all of our test problems SML-AMPL can barely finish the pre-processing stage in the time it takes by PSMG to generate the whole problem. An even more extensive amount of time is needed by SML-AMPL to generate the values for the matrix blocks. Furthermore, SML-AMPL's performance also highly depends on the disk access speed. We have used a solid state drive which provides a fast file system access speed for our test problems, and SML-AMPL is likely to run even slower on a system with a slower disk.

It turns out that the pre- and post-processing steps take a significant amount of time and does not scale well. This is mainly because for a large scale structured problem, thousands of `*.nl`-files and auxiliary files need to be created on the file system. Those files also need to be read and processed in order to evaluate the sub-blocks of the Jacobian matrix. Furthermore, the file system interface used due to the reliance on `*.nl`-files is unsuitable for parallelization. PSMG was developed to address these issues.

The benchmarking results demonstrate that PSMG's in-memory implementation is much preferred to the pre-/post-processing approach using SML-AMPL.

MSND Problem Instance		PSMG Problem Generation Time(s)			SML-AMPL Problem Generation Time(s)		
Number of commodities	Number of variables	Total	Structure Setup	Problem Generation	Total	Pre-processing	Post-processing
5	1,206,255	12	≤ 1	12	123	16	107
10	2,211,105	23	≤ 1	23	317	29	288
15	3,215,955	34	≤ 1	34	567	42	525
20	4,220,805	48	≤ 1	48	890	55	835
25	5,225,655	57	≤ 1	57	1318	70	1248
30	6,230,505	67	≤ 1	67	1855	81	1774
35	7,235,355	79	≤ 1	79	2619	94	2525
40	8,240,205	91	≤ 1	91	3507	111	3396
45	9,245,055	102	≤ 1	102	4699	126	4573
50	10,249,905	114	≤ 1	114	6022	141	5881

Table 8.1: Comparison of PSMG with SML-AMPL for generating MSND problem instances in serial execution.

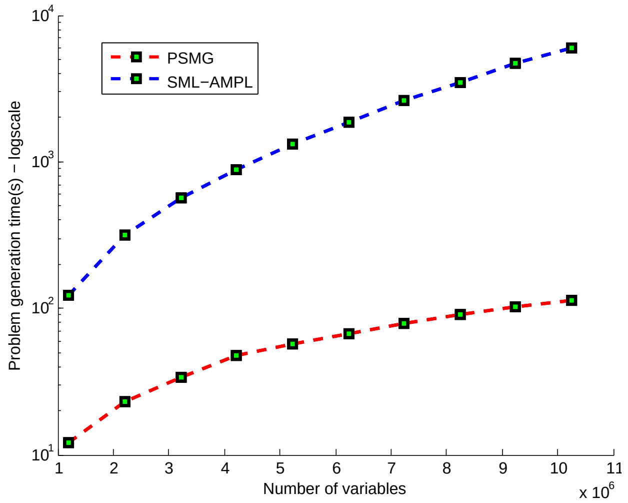


Figure 8.1: Semi-log plot for generating MSNG problem instances using PSMG and SML-AMPL. The data is from Table 8.1. It clearly demonstrates that the performance of PSMG is far superior to SMP-AMPL.

8.1.3 Serial Performance

The serial performance test is conducted using the three problem sets described in section 8.1.1 on a machine with Intel i5-3320M CPU. Table 8.2 presents the problem sizes and their corresponding generation time for the three problem sets.

The problem generation time in PSMG is composed of the two stages mentioned in the previous section: the structure building stage, which finishes by passing the expanded model tree to the solver; and the parallel problem generation stage. We report these separately in addition to the total problem generation time in Table 8.2.

Moreover, in Figure 8.2, 8.3 and 8.4 we plot the total problem generation time against the problem size (represented by the number of variables) for the three problem sets. The plots indicate that PSMG scales almost linearly for these problem set as the problem size increases.

	MSND Problem Instance					PSMG Problem Generation Times(s)		
	Number of commodities	Number of variables	Number of constraints		Number of nonzeros	Total	Structure setup	Problem generation
	5	1,206,255	270,570		3,416,490	12	≤ 1	12
	10	2,211,105	340,170		6,431,040	24	≤ 1	24
	15	3,215,955	409,770		9,445,590	36	≤ 1	36
	20	4,220,805	479,370		12,460,140	48	≤ 1	48
	25	5,225,655	548,970		15,474,690	59	≤ 1	59
	30	6,230,505	618,570		18,489,240	69	≤ 1	69
	35	7,235,355	688,170		21,503,790	79	≤ 1	79
	40	8,240,205	757,770		24,518,340	93	≤ 1	93
	45	9,245,055	827,370		27,532,890	103	≤ 1	103
	50	10,249,905	896,970		30,547,440	119	≤ 1	119
	ALM-SSD Problem Instance							
Number of scenarios	Number of benchmarks							
441	21	34,262	15,268		168,817	0	≤ 1	0
1681	41	196,442	91,308		1,115,457	3	≤ 1	3
4096	64	665,803	316,225		4,034,571	15	≤ 1	15
7056	84	1,428,263	685,525		8,924,171	34	≤ 1	34
11236	106	2,767,777	1,338,463		17,669,787	67	≤ 1	67
16384	128	4,755,851	2,311,809		30,810,123	114	≤ 1	114
21904	148	7,233,511	3,528,469		47,322,123	182	≤ 1	182
28900	170	10,814,561	5,290,911		71,336,091	287	≤ 1	287
36864	192	15,415,883	7,559,617		102,346,763	432	≤ 1	432
44944	212	20,591,783	10,115,157		137,362,443	616	≤ 1	616
	ALM-VAR Problem Instance							
	Number of scenarios			Number of nonzero (Jacobian)	Number of nonzero (Hessian)			
	1,600	249,292	85,242	169,817	1,600	3	0	3
	3,600	556,312	190,262	1,115,457	3,600	9	0	9
	4,600	984,932	336,882	4,034,571	6,400	14	0	14
	10,000	1,535,152	525,102	8,924,171	10,000	25	1	24
	14,400	2,206,972	754,922	17,669,787	14,400	38	2	36
	19,600	3,000,392	1,026,342	30,810,123	19,600	51	2	49
	25,600	3,915,412	1,339,362	47,322,123	25,600	66	4	62
	32,400	4,952,032	1,693,982	71,336,091	32,400	86	6	80
	40,000	6,110,252	2,090,202	102,346,763	40,000	104	8	96
	48,400	7,390,072	2,528,022	137,362,443	48,400	128	12	116

Table 8.2: Problem sizes and PSMG problem generation time for MSND, ALM-SSD and ALM-VAR problem instances.

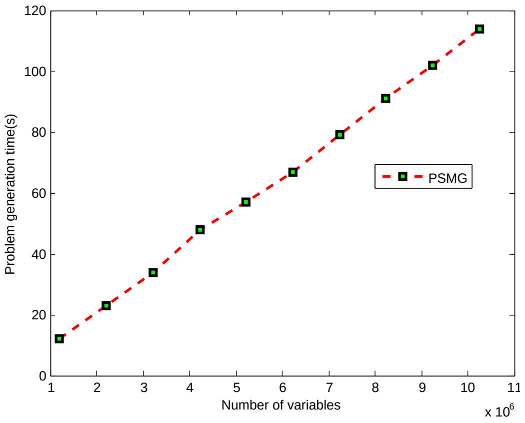


Figure 8.2: Plot for PSMG's problem generation time for the MSND problems in Table 8.2.

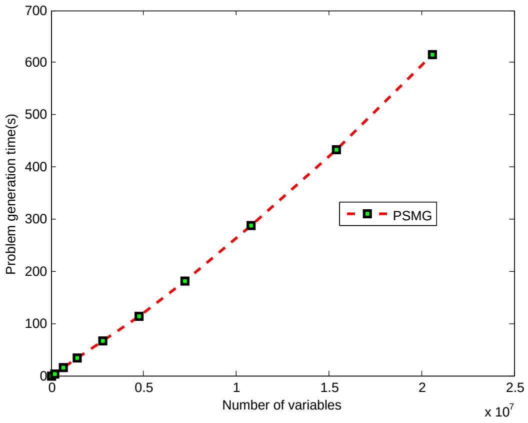


Figure 8.3: Problem generation time for the ALM-SSD problems in Table 8.2

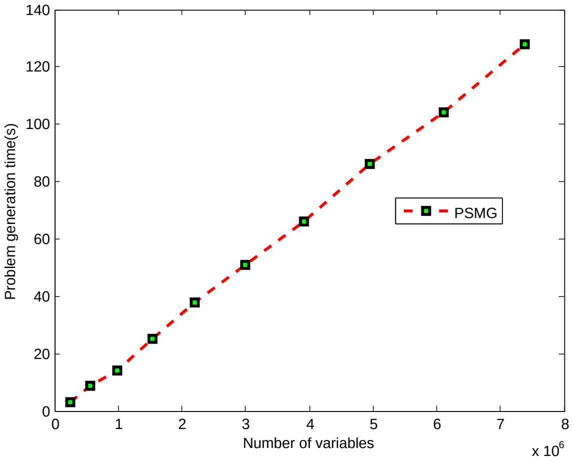


Figure 8.4: Problem generation time for the ALM-VAR problems in Table 8.2

8.1.4 Parallel Performance

On a node with 4GB memory, plain AMPL will not be able to generate a MSND problem with 100 commodities on a network comprising 30 nodes and 435 edges due to running out of memory. This problem (*msnd30_100* - twice as large as the largest MSND problem in Table 8.2) has $20,298,405$ variables and $1,592,970$ constraints. Because PSMG distributes the problem data among processors, this problem can be generated in parallel using PSMG.

We have also performed parallel scaling benchmarks for a large ALM-SSD problem (the largest problem in Table 8.2). This problem has 44944 scenarios, $20,591,783$ variables and $10,115,157$ constraints.

Tables 8.3 and 8.4 show the problem generation time, the resulting speedup and parallel efficiency for these problem when generated on up to 96 processes in parallel. The parallel efficiency is also plotted against the number of processors for both problems in Figures 8.5 and 8.6. These parallel experiments were performed on the parallel cluster at the Edinburgh Compute and Data Facility (also known as Eddie). It comprises 156 worker nodes, each of which is an IBM iDataplex DX360 M3 server with two six core Intel E5645 CPUs and 26GB of RAM. A single core on Eddie is slower than the Intel i5-3320M CPU used for our earlier serial experiments, so these benchmarking times are not directly comparable.

We observe that PSMG obtains excellent speed-up (> 0.9) on up to 24 processes and still a respectable speed-up in excess of 0.7 on 96 processes. The main reason preventing even higher speed-up for both problem sets (MSND and ALM-SSD) is the lack of perfect load balancing (note that the number of sub-blocks in both examples is not divisible by 96). Another reason is OOPS allocates the bottom right corner blocks of the Jacobian matrix on every parallel process based on its parallel allocation scheme. Therefore, the work to evaluate these blocks is repeated on every parallel process, but not distributable.

Number of parallel processes	Finishing Times(s)	Speedup	Efficiency
1	532	NA	NA
2	270	1.97	0.99
4	135	3.94	0.99
8	68	7.82	0.98
12	45	11.82	0.99
24	23	23.13	0.96
48	12	44.33	0.92
96	7	76	0.79

Table 8.3: PSMG speedup and parallel efficiency for a large MSND problem.

8.1.5 Memory Usage Analysis

We have also measured the per processor and total memory usage of PSMG for generating *msnd30_100* problem. The memory usage in each PSMG process is

Number of parallel processes	Finishing Times(s)	Speedup	Efficiency
1	1085	NA	NA
2	564	1.92	0.96
4	281	3.85	0.96
8	140	7.7	0.96
12	93	11.55	0.96
24	48	22.16	0.92
48	27	38.79	0.81
96	15	67.88	0.71

Table 8.4: PSMG speedup and parallel efficiency for the largest ALM-SSD problem.

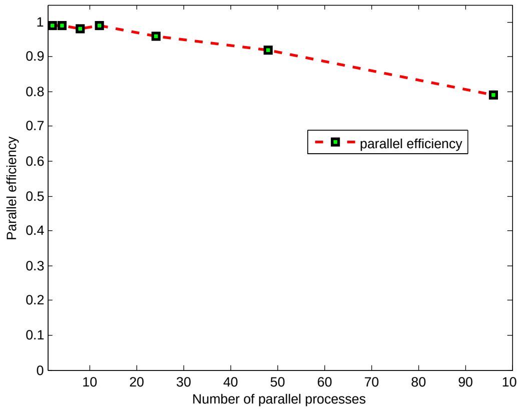


Figure 8.5: Parallel efficiency plot for MSND problem.

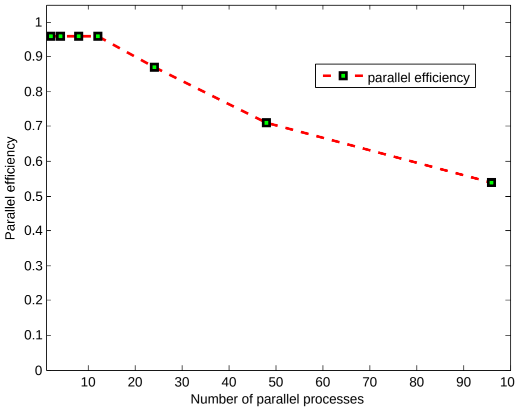


Figure 8.6: Parallel efficiency plot for ALM-SSD problem.

composed of the memory used to store the problem structures (prototype model tree and expanded model tree) and the problem data required to compute the function and derivative evaluation. Recall that the problem data in the expanded model tree will be distributed over all the parallel processes, whereas the problem structure information is not distributable and has to be repeated on every process. We define the memory overhead to be this (*i.e.* the non-distributable) part of the total memory usage.

This memory usage data is presented in Table 8.5. Columns 4 and 5 give the total and per-processor memory requirements respectively. The total memory is broken down in columns 2 and 3 into memory used for problem structure and data required for function and derivative evaluations. Column 6 gives memory overhead as a percentage of total memory usage. We also plot the total memory usage and the average memory usage in Figure 8.7 and 8.8 correspondingly.

For this example the non-distributable part requires about 40MB. This consists of memory used for the prototype model tree (24.5 KB) and the expanded model tree (39.93MB). Note that this problem has 46500 nodes in the expanded model tree; on average, each node in the expanded model tree thus uses about 900 bytes to store the problem structure. This memory is repeated on every processor. The remaining part of 1.46 GB is distributable over processes.

Thus we are able to distribute the vast majority of the memory required, enabling the generation of problems that can not be generated on a single node. The overhead in non-distributable memory is mainly due to making the prototype and expanded model tree available on every processor. This, however, is crucial to enable the solver driven processor assignment and to achieve on-demand parallel problem generation at later stages, so we maintain that it is a worthwhile use of memory.

Number of parallel processes	Total memory by problem structure (GB)	Total memory by problem data (GB)	Total memory (GB)	Memory per process (MB)	Structure memory overhead
1	0.04	1.46	1.50	1532.7	2.6%
2	0.08	1.46	1.54	786.3	5.1%
4	0.16	1.46	1.61	413.2	9.7%
8	0.32	1.46	1.77	226.6	17.6%
12	0.48	1.46	1.93	164.4	24.3%
24	0.96	1.46	2.39	102.2	39.1%
48	1.92	1.46	3.33	71.1	56.2%
96	3.84	1.46	5.21	55.5	72.0%

Table 8.5: Parallel processes memory usage information for generating problem `msnd30_100`.

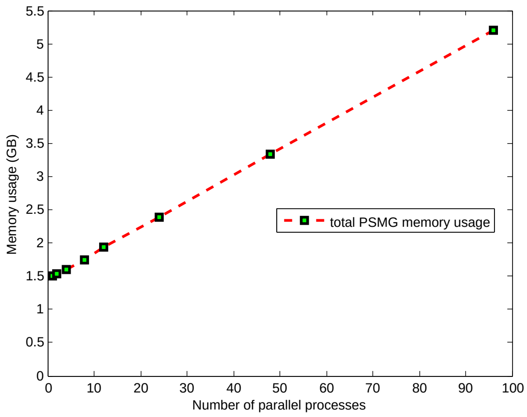


Figure 8.7: Total memory usage plot for generating problem *msnd30_100*.

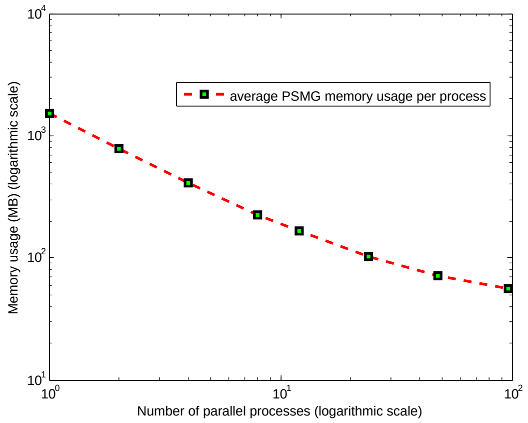


Figure 8.8: Per processor memory usage for generating problem *msnd30_100*.

8.2 NLP Problems

Using the *local* interface for PSMG to generate NLP problems may have a significant parallel performance penalty in any solver. For example, let us consider the bottom right corner block in the Hessian of the Lagrangian matrix for the complicating variables declared in the master problem. We should also refer to this block as the *complicating block* in the Hessian of the Lagrangian matrix.

If the complicating variables in the master problem are forming nonlinear relations in constraints from its sub-problems, the evaluation of the *complicating block* in the Lagrangian Hessian matrix may involve every constraint in all of the sub- and master problems. According to OOPS' parallel problem allocation scheme, this *complicating block* is being allocated on every parallel process. Therefore, if the *local* interface methods are used for parallel generation problem for OOPS, every processor may need to evaluate this *complicating block* locally. Thus, the entities in every expanded model tree node of the entire problem need to be expanded on all the processors. Not only does the solver need to provide the variable values of the entire problem on every processor, but also the memory needs to be allocated for storing the expansion of the entire problem's constraints. It essentially defeats the purpose of parallelization of the problem generation in this case. For NLP problems with this structure, it is quite expensive to compute the *complicating block* in the Hessian of the Lagrangian matrix. As a result, the *distributed* interface method should be considered, which allows the evaluation of the *complicating block* to be distributed on parallel processes according to its separable structure in PSMG's problem formulation.

For NLP problems, OOPS requires the sparsity pattern of the Jacobian and Hessian matrix block to be evaluated in the first iteration to set up the memory storage and symbolic factorisation before requesting the actual values for the blocks. Therefore, the cost of problem generation in the first iteration is more expensive than the cost of function evaluation for each subsequent iteration. In a subsequent iteration, OOPS only requires PSMG to update values in the matrix blocks whose sparsity structure is already known. Therefore, we report the per-iteration problem generation time for the first iteration and subsequent iterations separately in the test results.

8.2.1 SCOPF Problem

To demonstrate PSMG for the generation of NLP problems, we have used a set of *Security Constrained Optimal Power Flow* (AC-SCOPF) problems.

In the SCOPF problem, we are seeking a long-term operation plan which allows demands and power system constraints still to be satisfied when any predefined equipment failure occurs on the power system network. This problem can be viewed as a two level structure problem. The master problem has a set of demand and power system constraints for the full power system network, whereas the similar set of constraints should be repeated for each of the contingency cases on the reduced network.

To satisfy the demands on the reduced network for each contingency case, the real power generation can be modified on a reference bus to allow correction actions. The power and voltage on the real power generator buses should remain the same for all the contingency cases, which introduces linking variables between the master and sub-problems. We have modelled these restrictions with linear constraints in the sub-problem (*i.e.* `BusVolCons` and `BusGenCons` at line 422 and 424 in Appendix B). The full model file of the SCOPF problem in PSMG is given in Appendix B. For the concern of this thesis, we will not provide the detailed mathematical formulation for the SCOFP problem (which can be found in [71]), but rather provide the following key facts about SCOPF problem formulation in PSMG.

- The complicating variables from the master problem do not occur in the nonlinear constraint of the sub-problems.
- The objective function is formulated using variables from master problem only.
- Variables from a sub-problem are not referred to in the master problem's constraints, and there is no linking constraint in the master problem.

According to the above facts observed from SCOPF's model in PSMG, we can infer that the Hessian of the Lagrangian presents a block diagonal matrix structure, and each block can be evaluated with local information without any inter-process communication efforts. As a result, it eliminates the potential issue of inefficiency caused by the *local* interface methods for evaluating Hessian of the Lagrangian matrix blocks.

Furthermore, OOPS allocates the master variables on every parallel processors. Therefore every Jacobian matrix block and constraint function can also be evaluated using local information already available on every parallel processor.

For this reason, the SCOPF problem can be efficiently generated using *local* interface methods on OOPS. Moreover, PSMG provides good parallel results for generating the SCOPF problems using its *local* solver interface linked with OOPS.

8.2.2 Serial Performance

We have generated a set of SCOPF problem instances on a random energy network of 100 buses, 125 generator, 500 lines and 10 transformers. The number of contingency cases are between 20 and 510 for these problems. The problem sizes and their corresponding generation time are presented in Table 8.6.

The serial performance test is conducted on a machine with Intel i5-3320M CPU and a solid state drive. The results are presented in Table 8.6, where Column 6 and 7 provide the per-iteration costs for problem generation in the first and each subsequent iteration.

In addition, we plot the per-iteration generation times against the problem sizes (represented by the number of variables) in Figure 8.9 and 8.10 for the first and subsequent iteration respectively. Similar to LP and QP problems, the plots also indicate the problem generation time in PSMG scales almost linearly as the problem size increases.

8.2.3 Parallel Performance

The special problem structure and variable dependencies among master and sub-problems presented in the SCOPF problem allow each derivative and function

SCOPF Problem Instance					Problem Generation Time(s)		
Number contingency	Number of constraints	Number of variables	Number of nonzeros (Jacobian)	Number of nonzeros (Hessian)	Structure setup	Problem generation (first iteration)	Problem generation (subsequent)
20	73,301	73,590	338,703	110,445	≤ 1	64	49
30	108,321	108,630	500,909	162,029	≤ 1	98	76
40	143,421	143,670	662,487	215,251	≤ 1	125	102
50	178,461	178,710	823,753	265,281	≤ 1	153	128
90	318,621	318,870	1,472,377	477,011	≤ 1	277	222
170	598,771	599,190	2,765,469	893,187	≤ 1	535	392
330	1,158,921	1,159,830	5,344,913	1,699,109	≤ 1	1027	772
510	1,790,301	1,790,550	8,261,941	2,659,715	≤ 1	1544	1283

Table 8.6: Problem sizes and PSMG problem generation time for SCOPF problem instances.

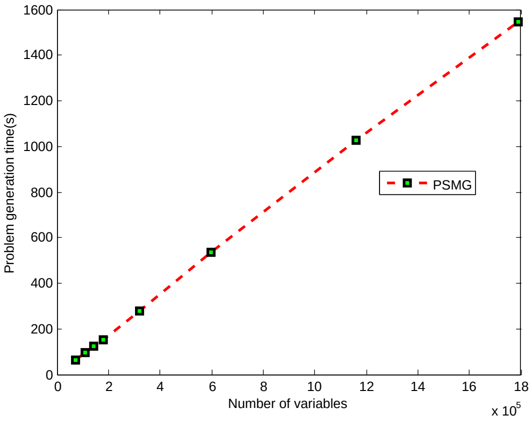


Figure 8.9: Plot for PSMG's problem generation time for the first iteration of the SCOPF problems in Table 8.6.

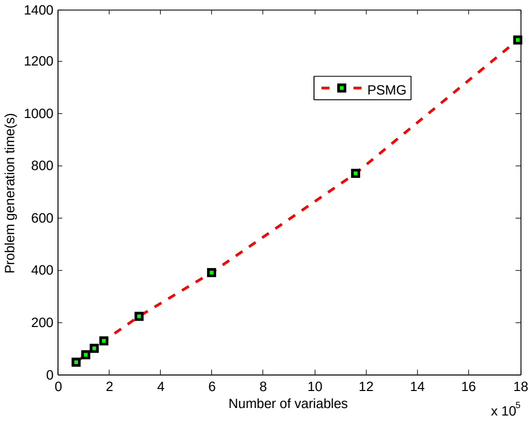


Figure 8.10: Problem generation time for the subsequent iteration of the SCOPF problems in Table 8.6.

evaluation to be performed locally without any inter-process communication effort. Therefore, good parallel efficiency can be expected for generating SCOPF problems using PSMG's *local* interface methods linked with OOPS.

We have conducted the parallel scaling benchmark tests for the largest SCOPF problem in Table 8.6. This problem has over 1.7 million of variables and constraints and 510 contingencies (or sub-problems). The parallel experiments were also executed on Eddie. The per-iteration speedup and parallel efficiency for first and each subsequent iteration are reported in separate columns in Table 8.7. In addition, the per-iteration parallel efficiency is also plotted in Figure 8.11 and 8.12 for first and each subsequent iteration respectively.

We can observe that PSMG obtains good speed-up for both first and subsequent iterations on and up to 96 processes. Similar to the results from LP problem, there are two reasons preventing even higher speed-up. Firstly it is lack of perfect load balancing (note that the number of sub-problem in this SCOPF problem is not divisible by 12). Secondly, OOPS allocates the bottom right corner blocks of both Jacobian and Hessian matrices on every parallel process, therefore the work required for the evaluation of these blocks is repeated on every parallel process, but not distributable. However, extra communication cost may be needed if allocating the top level problem on a dedicated master processes.

Number of	First time setup			Subsequent update		
parallel processes	Finishing Time(s)	Speedup	Efficiency	Finishing Time(s)	Speedup	Efficiency
1	1965	NA	NA	1453	NA	
12	186	10.56	0.88	191	7.61	0.63
24	96	20.47	0.85	97	14.98	0.62
36	67	29.33	0.81	68	21.37	0.59
48	50	39.30	0.82	51	28.49	0.59
60	41	47.93	0.80	43	33.79	0.56
72	37	53.11	0.74	35	41.51	0.58
84	30	65.50	0.78	30	48.43	0.58
96	28	70.18	0.73	28	51.89	0.54

Table 8.7: PSMG speedup and parallel efficiency for largest SCOPF problem in Table 8.6.

8.3 Modelling and Solution Time

8.3.1 Serial Analysis

In order to compare PSMG's modelling time to the total execution time, we have modelled and solved a set of MSND problems based on a network of 20 Nodes and 190 Arcs with various commodity sizes. Those MSND problems are modelled and solved in serial execution on a machine with Intel E7-4830 CPU. The problem sizes and their corresponding serial performance results are presented in Table 8.8. The modelling time represents the time spent for PSMG to generate the problem and pass it to the solver. The solving time is the number of seconds for the solver (OOPS in this case) to solve the corresponding model to reach optimality. We have also computed the modelling time as a percentage of the

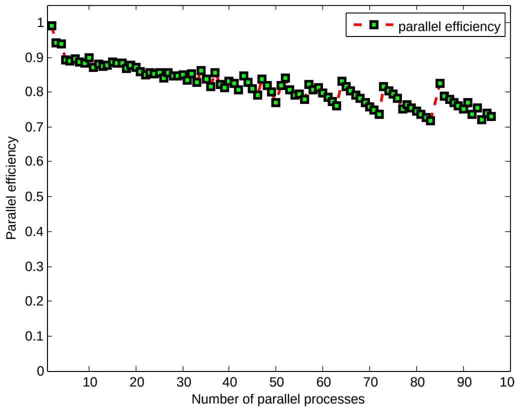


Figure 8.11: Per-iteration parallel efficiency plot for problem generation in the first iteration of the largest SCOPF problem in Table 8.6.

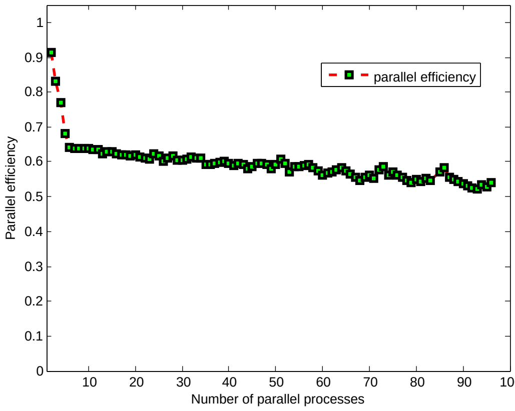


Figure 8.12: Per-iteration parallel efficiency plot for problem generation in the subsequent iterations of the largest SCOPF problem in Table 8.6.

total execution time, which is presented in the last column of Table 8.8. We can observe that the modelling time percentages for all of the MSND problem instances (given in Table 8.8) are between 4.04% and 7.28%.

The modelling time percentage is also dependent on how hard it is for the solver to solve a particular problem. Since a harder problem may require more solver iterations to reach optimality, the modelling time percentage could vary depending on not only the size, but also the hardness of the problem. Therefore, the modelling time percentage could be highly dependent on the complexity of an optimization problem.

MSND Problem Instance						
Number of commodities	Number of variables	Number of constraints	Number of nonzeros	Modelling Time	Solving Time	Modelling Time (%)
5	236,170	60,230	668,610	15	296	4.82%
10	432,820	81,130	1,258,560	31	433	6.68%
15	629,470	102,030	1,848,510	46	586	7.28%
20	826,120	122,930	2,438,460	59	1038	5.38%
25	1,022,770	143,830	3,028,410	77	1346	5.41%
30	1,219,420	164,730	3,618,360	90	1219	6.88%
35	1,416,070	185,630	4,208,310	106	1627	6.12%
40	1,612,720	206,530	4,798,260	122	2012	5.72%
45	1,809,370	227,430	5,388,210	138	2435	5.36%
50	2,006,020	248,330	5,978,160	152	3615	4.04%

Table 8.8: PSMG's modelling time for a set of MSNG problems and their corresponding solving time.

8.3.2 Parallel Analysis

To demonstrate PSMG's parallel modelling time with respect to the parallel solving time, we have solved the largest MSND problem from Table 8.8. The results of those parallel experiments are presented in Table 8.9. Those parallel experiments were performed on a parallel computing server with Intel E7-4830 CPUs. Because of the limited availability of the computing resource at the time for conducting those parallel experiments, we only be able to run up-to 32 parallel processes. The results demonstrate that modelling (problem generation) contributes less than 4% of total execution time for all the parallel executions. Again, the modelling time percentage is highly dependent on the hardness of the problem, therefore we may not be able to generalize the result universally for other problems.

Number of parallel processes	Modelling time	Solving time	Modelling time (%)
1	152	3615	4.04%
2	80	1903	4.03%
4	41	985	4.00%
8	21	539	3.75%
16	11	286	3.70%
32	6	160	3.61%

Table 8.9: PSMG's modelling time the MSND problem.

8.3.3 Discussion

It is worth noting that we have used only LP problems to assess the percentage of the modelling time with respect to the total execution time. LP problems only require their problem matrices to be evaluated once, and matrices remain constant for rest of the solver iterations. For NLP problems, modelling could take more proportion of the total execution time, since at each sub-sequent solver iteration PSMG may be invoked again for evaluating the updated problem matrices. Since the modelling and solving time are highly related to the hardness of an optimization problem, the percentage of modelling time may also differ for different problems.

The purpose of the results in this section are intended to provide a basic guideline for the performance and design efficiency of PSMG.

8.4 Discussion

In this chapter, we have demonstrated the performance for PSMG's problem generation by linking the *local* interface methods with OOPS. The results in this chapter can be generalised for solvers that implements the structure exploiting IPM algorithm (such as PIPS [24],etc).

Based on our performance results, PSMG achieves good parallel efficiency for generating large scale LP and QP problems. It is worth mentioning that PSMG implements the solver driven work assignment approach, therefore those parallel performance results are produced by adopting the parallel problem allocation scheme of the OOPS solver. This parallel problem allocation scheme allows each of the blocks in the Jacobian and objective Hessian matrices to be evaluated using local information without inter-process communication effort.

We have also achieved satisfactory parallel performance for generating the AC-SCOPF problem, however the result cannot be applied to general NLP problems. For a general NLP problems modelled in PSMG, evaluation of the blocks in Jacobian and Hessian of the Lagrangian matrix may require variable values from its dependent nodes. In addition, if the variables in the master problem have nonlinear relation in the constraints of its sub-problems, the evaluation of the *complicating block* in the Hessian of the Lagrangian matrix could be quite expensive using PSMG's *local* interface methods. In this case, the *distributed* interface methods may need to be used.

PSMG is very likely to have similar performance for linking with a decomposition solver, since the same interface methods are used.

We are aware that our implementation may not match the performance of a commercial model generator such as AMPL on a single node but we believe this will be offset by the advantage realized from exploiting parallelization.

Chapter 9

Conclusions

9.1 Research Summary

In this thesis, we have presented PSMG –a model generator for structured problems– which is capable of not only conveying the problem structure to the solver, but also parallelising the problem generation process itself. PSMG uses the modelling language SML which offers an easy-to-use syntax to model nested structured optimisation problems and stochastic programming problems.

Unlike its predecessor, an AMPL based SML implementation (SML-AMPL), PSMG is independent of AMPL. It can efficiently extract the problem structure and convey it to the solver within seconds for large scale structured problems with millions of variables and constraints. Based on our serial performance benchmarks, PSMG's runtime performance is far superior to that of SML-AMPL.

We have also explained PSMG's parallel problem generation design in detail. PSMG generates a structured problem in two stages. In the first stage, PSMG builds a representation of the problem structure with only minimal work required. This representation is also used as the solver interface. The majority of the work involved in problem generation is at the second stage, where the necessary data structures are built for constraint function and derivative evaluations. Any temporary set and parameter values evaluated in the second stage are also stored in memory for future use. PSMG evaluates the first and second order derivatives using state-of-the-art AD algorithms implemented in a separate module. This design allows PSMG to adopt different AD algorithms easily in the future. Additionally, PSMG is able to use the problem structure information to parallelize the second stage of the problem generation process. As far as we know, PSMG is the only model generator for an algebraic modelling language that can pass the structure information to the solver and use this information to parallelize the model generation process itself.

In this thesis, we have also discussed PSMG's novel parallel interface design, namely solver driven work assignment. This design enables the parallel solver to make parallel allocation decisions to achieve load balancing and data locality, and to minimize the amount of data communications among parallel processes. We have illustrated that by paying a small memory overhead to store the problem structure on every parallel process, PSMG can implement solver driven problem distribution to achieve the on-demand problem generation, and further eliminate inter-process communication in both the model generation and function evaluation stages.

PSMG offers two types of solver interfaces for generating LP, QP and NLP problems. The *local* interface methods evaluate each function and derivative request locally. Therefore, possible inter-process communication efforts are needed for passing variable values from the dependent nodes. Since LP and QP problems have constant Jacobian and Hessian matrices whose results does not depend on variable values, the *local* interface methods are sufficient to provide an excellent parallel efficiency for generating LP and QP problems. We have also illustrated that the *local* interface methods may have a significant performance issue for generation NLP problems with border blocks in their Hessian of the Lagrangian matrix. On the other hand, the *distributed* interface methods should be used in this case to achieve better parallel performance.

PSMG's solver interface can be linked with different parallel solvers and algorithms. In particular, we have demonstrated how to link PSMG's solver interface with a parallel structure exploiting IPM solver (*i.e.* OOPS) and solvers that implement Benders' Decomposition scheme.

To evaluate the performance of PSMG, we have linked PSMG's *local* solver interface methods with OOPS and generated several different structured problems with various sizes. The performance evaluation of PSMG shows good parallel efficiency in terms of both speed and memory usage for generating LP and QP problems. We demonstrate that PSMG is able to handle much larger mathematical programming problems that could not be generated by AMPL due to memory limitation on a single node.

PSMG also achieves good parallel efficiency for generating the SCOPF problems. Since the Hessian of Lagrangian matrix of the SCOPF problem presents a block-diagonal structure with no borders, PSMG's *local* interface methods are able to provide an efficient solution for generating this problem in parallel for OOPS. We suspect the parallel efficiency benchmark result can not be generalized for a general NLP problem, where the Hessian matrix could have border blocks.

9.2 Future Work

When using PSMG for generating a problem, a considerable amount of time is spent on expansion of constraints and variables. At this time PSMG does not save the expanded constraints and variables, therefore, if the same problem is solved again with different algorithms or solvers, this expansion work needs to be repeated. It should be quite useful for PSMG to offer ability to store the expanded constraints and variables on disk based on the problem's structure and its parallel allocation so that when the problem is solved again, PSMG can load the expanded constraints and variables directly from disk. This feature is analogous to AMPL's ability to save the generated problem in NL-file format. However, AMPL doesn't deal with structured problem, and stores only a single NL-file to represent the problem. PSMG should address this feature in a parallel and structured fashion.

We have linked PSMG's *local* solver interface with OOPS for demonstration purpose at this time. This allows us to efficiently generate any LP and QP problems in parallel. It can also efficiently generate the NLP problems with block-diagonal structure in their Hessian of the Lagrangian matrix. However, to achieve efficient parallel problem generation for a general NLP problem, the *distributed* interface methods should be used. Linking PSMG's *distributed* interface methods with parallel solvers can also be a potential future work.

PSMG's parallel problem generation efficiency can depend on how the solver allocates the problem on parallel processes. Different solvers may adopt different parallel allocation schemes. Therefore, it is a good idea to test PSMG with different parallel allocation schemes and to compare the performance results among them. This will also help us to identify any bottle-neck in the current solver interface design and to improve the flexibility of PSMG's solver interface.

Appendices

Appendix A

AutoDiff_Library Interface Methods

A list of interface methods for building the computational graph of a function expression.

- `Node* create_var_node(uint& idx)`
  - creates a variable node with index for value lookup.
- `Node* create_param_node(double value)`
  - creates a parameter node with a fixed value.
- `Node* create_param_node(uint& idx)`
  - creates a parameter node with index for value lookup.
- `Node* create_unary_op_node(OPCODE code, Node* left)`
  - creates a unary operator node with its OPCODE and operand pointed by `left`.
- `create_binary_op_node(OPCODE code, Node* left, Node* right)`
  - creates a binary operator node with its OPCODE and two operands pointed by `left` and `right`.

A list of interface methods for evaluating gradient and Hessian matrices and their non-zero structures.

- `double eval_function(Node* root)`
  - returns the function value for function expression pointed by `root`.
- `uint nzGrad(Node* root)`
  - returns the number of non-zero element in the function expression pointed by `root`.

- `double grad_reverse(Node* root, vector<double>& grad)`
  - returns the function value and fills the gradient vector in `grad`. The dimension of `grad` should agrees with the dimension of the gradient vector of the function pointed by `root`.
- `void nonlinearEdges(Node* root, EdgeSet& edges)`
  - implements the `edge_pushing` algorithm for computing a list of nonlinear edges in the function expression pointed by `root`.
- `uint nzHess(EdgeSet& edges, col_compress_imatrx& im)`
  - returns the number of non-zero element in the Hessian matrix. The sparsity structure is filled in the Column Compressed Matrix, `im`.
- `double hess_reverse(Node* root, col_compress_matrix& hess)`
  - returns the funtion value and fills the Hessian matrix in Column Compress Matrix format in `hess`.

A list of tailor-made interface method calls for PSMG in order to generate Jacobian and Hessian matrices in sub-blocks.

- `uint nzGrad(Node* root, boost::unordered_set<Node*>& vnodes)`
  - returns the number of non-zero element in gradient sub-vector defined by variables from `vnodes` of the function expression pointed by `root`.
- `nzGrad(Node* root, vector<Node*>& vlist, col_compress_imatrix_row& rgrad)`
  - returns the number of non-zero element in gradient sub-vector defined by variables from `vnodes` of the function expression pointed by `root`. This methods also fills the gradient sparsity in `rgrad` row vector. The dimension of `rgrad` is equals to size of `vlist`.
- `double grad_reverse(Node* root, vector<Node*>& nodes, col_compress_matrix_row& rgrad)`
  - returns the function value and fills the gradient vector in `rgrad`.
- `uint nzHess(EdgeSet& edgeset,boost::unordered_set<Node*>& set1, boost::unordered_set<Node*>& set2)`
  - returns the number of non-zero elements in sub-block of a Hessian matrix. The variables from `set1` and `set2` identifies the Hessian sub-block. The `edgeset` contains a set of nonlinear edges computed using `edge_pushing` algorithm.
- `uint nzHess(EdgeSet& edgeset,boost::unordered_map<Node*,uint>& colvMap, boost::unordered_map<Node*,uint>& rowvMap,col_compress_imatrix& im)`

- same as above, and this method also fills the sparsity structure in CCS matrix format.

- `double hess_reverse(Node* root, vector<Node*>& nodes, col_compress_matrix& hess)`
  - returns the value of the function expression pointed by `root` and fills the Hessian matrix block identified by variables from `nodes`.

- `double hess_reverse_ep(Node* root,boost::unordered_map<Node*,uint>& colvMap, boost::unordered_map<Node*,uint>& rowvMap,col_compress_matrix& hess)`
  - returns the value of the function expression pointed by `root` and fills the Hessian matrix block identified by variables from `colvMap` and `rowvMap`.

Appendix B

Security Constrained Optimal Power Flow Model

```
param baseMVA;
param demScaleDemand;
param demScaleFlow;
param RefBus symbolic;

#BUS
set Bus;
param DemandP{Bus};
param DemandQ{Bus};
param MinVoltage{Bus} >=0;
param MaxVoltage{Bus} >=0;
param GS{Bus};
param BS{Bus};

#Line
set Line;
param LStartBus{Line} symbolic;
param LEndBus{Line} symbolic;
param LReactance{Line};
param LResistance{Line};
param LBsh{Line};
param LMaxFlow{Line} >=0;
param LContMaxFlow{Line} >=0;
param LNminus1{Line};
param LG{l in Line}:=LResistance[l]/(LReactance[l]^2+LResistance[l]^2);
param LB{l in Line}:=-LReactance[l]/(LReactance[l]^2+LResistance[l]^2);

#Transformer
set Transformer;
param TStartBus{Transformer} symbolic;
param TEndBus{Transformer} symbolic;
param TReactance{Transformer};
param TResistance{Transformer};
param TBsh{Transformer};
param TMaxFlow{Transformer} >=0;
param TContMaxFlow{Transformer} >=0;
param TNminus1{Transformer};
param TB{l in Transformer}:=TResistance[l]/(TReactance[l]^2 + TResistance[l]^2);
param TG{l in Transformer}:=-TReactance[l]/(TReactance[l]^2 + TResistance[l]^2);
param ExTap{Transformer} >=0;

# Generator
set Generator;
param Location{Generator} symbolic;
param MinGenP{Generator} >= 0;
param MaxGenP{g in Generator} >= MinGenP[g];
param MinGenQ{Generator};
param MaxGenQ{g in Generator} >= MinGenQ[g];
param GenConst{Generator};
param GenLin{Generator};
param GenQuad{Generator};

#computed parameters
set LUT = Line union Transformer;
param StartBus{lu in LUT}:= if (lu in Line) then LStartBus[lu] else TStartBus[lu] symbolic;
param EndBus{lu in LUT}:= if (lu in Line) then LEndBus[lu] else TEndBus[lu] symbolic;
param MaxFlow{lu in LUT}:= if(lu in Line) then demScaleFlow*LMaxFlow[lu] else demScaleFlow*TMaxFlow[lu];
param ContMaxFlow{lu in LUT}:= if(lu in Line) then demScaleFlow*LContMaxFlow[lu] else demScaleFlow*TContMaxFlow[lu];
param Bsh{lu in LUT} := if (lu in Line) then LBsh[lu] else TBsh[lu];
param G{lu in LUT}  := if (lu in Line) then LG[lu] else TG[lu];
param B{lu in LUT} := if (lu in Line) then LB[lu] else TB[lu];
set PVBUS within Bus = setof {g in Generator: MaxGenP[g]>0} Location[g];
set PQBUS = Bus diff PVBUS;
param TapRatio{t in Transformer} := ExTap[t];
set Contingencies within LUT = setof {r in Line: LNminus1[r] == 1} r union setof {t in Transformer: TNminus1[t] == 1} t;

################################################################
#root stage
###############################################################
#root level variables
var Voltages{b in Bus} >= MinVoltage[b], <= MaxVoltage[b];
var PpowerGen{g in Generator } >= MinGenP[g], <= MaxGenP[g];
var QpowerGen{g in Generator} >= MinGenQ[g], <= MaxGenQ[g];
var FlowStartP{l in LUT};
var FlowStartQ{l in LUT};
var FlowEndP{l in LUT};
var FlowEndQ{l in LUT};
var VoltageAngles{Bus};
var slackStart{l in LUT} <= MaxFlow[l]^2;
var slackEnd{l in LUT} <= MaxFlow[l]^2;

#KCL + P power
subject to KCLP{b in Bus}:
  sum{g in Generator: Location[g] == b} (PpowerGen[g])
  - sum{l in LUT: StartBus[l] == b} ( FlowStartP[l])
  - sum{l in LUT: EndBus[l] == b} (FlowEndP[l])
  - GS[b]*(Voltages[b]^2)
  =
  demScaleDemand*DemandP[b]
  ;

#KCL + Q power
subject to KCLQ{b in Bus}:
  sum{g in Generator: Location[g] == b} (QpowerGen[g])
  - sum{l in LUT: StartBus[l] == b } (FlowStartQ[l])
  - sum{l in LUT: EndBus[l] == b } (FlowEndQ[l])
  + 0.5*( Voltages[b]^2)
    *
    ( sum{l in Line: LStartBus[l] == b } (LBsh[l])
    +   sum{t in Transformer: TStartBus[t] == b} (TBsh[t]/TapRatio[t]^2)
    +   sum{l in LUT: EndBus[l] == b} (Bsh[l])
    )
  + BS[b]*(Voltages[b]^2)
  =
  demScaleDemand*DemandQ[b]
  ;

#KVL +P Power at start bus of lines
subject to PKVLStart{l in Line}:
  FlowStartP[l]
  - G[l]*(Voltages[StartBus[l]]^2)
  + Voltages[StartBus[l]] * Voltages[EndBus[l]]
    *
    (
      G[l] * cos(VoltageAngles[StartBus[l]] - VoltageAngles[EndBus[l]])
      +
      B[l] * sin(VoltageAngles[StartBus[l]] - VoltageAngles[EndBus[l]])
    )
  =
  0
  ;

#KVL +Q power at Start bus of lines
subject to QKVLStart{l in Line}:
  FlowStartQ[l]
  + B[l]*(Voltages[StartBus[l]]^2)
  +
  Voltages[StartBus[l]]*Voltages[EndBus[l]]
  *
  (
    G[l] * sin(VoltageAngles[StartBus[l]] - VoltageAngles[EndBus[l]])
    -
    B[l] * cos(VoltageAngles[StartBus[l]] - VoltageAngles[EndBus[l]])
  )
  =
  0
  ;

#kVL +P power at End bus of lines
subject to PKVLEnd{l in Line}:
  FlowEndP[l]
  - G[l]*(Voltages[EndBus[l]]^2)
  + Voltages[StartBus[l]] * Voltages[EndBus[l]]
  *
  (
    G[l]* cos(VoltageAngles[EndBus[l]] - VoltageAngles[StartBus[l]] )
    +
    B[l]* sin(VoltageAngles[EndBus[l]] - VoltageAngles[StartBus[l]] )
  )
  =
  0
  ;

#KVL +Q power at End bus of lines
subject to QKVLEnd{l in Line}:
  FlowEndQ[l]
  + B[l]*(Voltages[EndBus[l]]^2)
  + Voltages[StartBus[l]] * Voltages[EndBus[l]]
  *
  (
    G[l]* sin(VoltageAngles[EndBus[l]] - VoltageAngles[StartBus[l]] )
    -
    B[l]* cos(VoltageAngles[EndBus[l]] - VoltageAngles[StartBus[l]] )
  )
  =
  0
  ;

################# KVL Transformer
#KVL +P power at start bus of transformer
subject to tPKVLStart{l in Transformer}:
  FlowStartP[l]
  - G[l]*(Voltages[StartBus[l]]^2)/(TapRatio[l]^2)
  + Voltages[StartBus[l]]*Voltages[EndBus[l]]/TapRatio[l]
  *
  (
    G[l]*cos(VoltageAngles[StartBus[l]] - VoltageAngles[EndBus[l]] )
    +
    B[l]*sin(VoltageAngles[StartBus[l]] - VoltageAngles[EndBus[l]] )
  )
  =
  0
  ;

#KVL +P power at end bus of transformer
```

```
subject to tPKVLEnd{l in Transformer}:
  FlowEndP[l]
  -G[l]*Voltages[EndBus[l]]^2
  + Voltages[StartBus[l]]*Voltages[EndBus[l]]/TapRatio[l]
  *
  (
    G[l]* cos(VoltageAngles[EndBus[l]] - VoltageAngles[StartBus[l]]    )
    +
    B[l]* sin(VoltageAngles[EndBus[l]] - VoltageAngles[StartBus[l]]   )
  )
  =
  0
  ;

#KVL +Q power at start bus of transformer
subject to tQKVLStart{l in Transformer}:
  FlowStartQ[l]
  + B[l]*(Voltages[StartBus[l]]^2)/(TapRatio[l]^2)
  + Voltages[StartBus[l]]/TapRatio[l]
  *
  (
    G[l]*sin(VoltageAngles[StartBus[l]] - VoltageAngles[EndBus[l]])
    -
    B[l]*cos(VoltageAngles[StartBus[l]] - VoltageAngles[EndBus[l]])
  )
  =
  0
  ;

#KVL +Q power at end bus of transformer
subject to tQKVLEnd{l in Transformer}:
  FlowEndQ[l]
  + B[l]*(Voltages[EndBus[l]]^2 )
  + Voltages[StartBus[l]]*Voltages[EndBus[l]]/TapRatio[l]
  *
  (
    G[l]*sin(VoltageAngles[EndBus[l]] - VoltageAngles[StartBus[l]])
    -
    B[l]*cos(VoltageAngles[EndBus[l]] - VoltageAngles[StartBus[l]])
  )
  =
  0
  ;

#reference bus
subject to RefBusZero:
  VoltageAngles[RefBus] = 0;

#############Line and Transformer Thermal Limits
#Flow limit at Start bus of each line
subject to FlowLimitStart{l in LUT}:
  FlowStartP[l]^2 + FlowStartQ[l]^2 - slackStart[l] = 0;

subject to FlowLimitEnd{l in LUT}:
  FlowEndP[l]^2 + FlowEndQ[l]^2 - slackEnd[l] = 0;
######################################################################
# 2nd Stage
# Contingency cases
######################################################################
block Contingency{c in Contingencies}: {
  set LUTDIFF = LUT diff {c};
  set LineDIFF= Line diff {c};
  set TransformerDIFF = Transformer diff {c};
  var cVoltages{b in Bus} >= MinVoltage[b], <= MaxVoltage[b];
  var cPpowerGen{g in Generator } >= MinGenP[g], <= MaxGenP[g];
  var cQpowerGen{g in Generator} >= MinGenQ[g], <= MaxGenQ[g];
  var cFlowStartP{l in LUTDIFF};
  var cFlowStartQ{l in LUTDIFF};
  var cFlowEndP{l in LUTDIFF};
  var cFlowEndQ{l in LUTDIFF};
  var cVoltageAngles{Bus};
  var cslackStart{l in LUTDIFF} <= MaxFlow[l]^2;
  var cslackEnd{l in LUTDIFF} <= MaxFlow[l]^2;

  #KCL + P power
  subject to KCLP{b in Bus}:
    sum{g in Generator: Location[g] == b} (cPpowerGen[g])
    - sum{l in LUTDIFF: StartBus[l] == b} (cFlowStartP[l])
    - sum{l in LUTDIFF: EndBus[l] == b} (cFlowEndP[l])
    - GS[b]*(cVoltages[b]^2)
    =
    demScaleDemand*DemandP[b]
    ;


  #KCL + Q power
  subject to KCLQ{b in Bus}:
    sum{g in Generator: Location[g] == b} (cQpowerGen[g])
    - sum{l in LUTDIFF: StartBus[l] == b } (cFlowStartQ[l])
    - sum{l in LUTDIFF: EndBus[l] == b } (cFlowEndQ[l])
    +
    0.5*( cVoltages[b]^2)
    *
    ( sum{l in LineDIFF: LStartBus[l] == b } (LBsh[l])
    + sum{t in TransformerDIFF: TStartBus[t] == b} (TBsh[t]/TapRatio[t]^2)
    +   sum{l in LUTDIFF: EndBus[l] == b} (Bsh[l])
    )
    + BS[b]*(cVoltages[b]^2)
    =
    demScaleDemand*DemandQ[b]

    ;

  #KVL +P Power at start bus of lines
  subject to PKVLStart{l in LineDIFF}:
    cFlowStartP[l]
    - G[l]*(cVoltages[StartBus[l]]^2)
    + cVoltages[StartBus[l]] * cVoltages[EndBus[l]]
    *
    (
      G[l] * cos(cVoltageAngles[StartBus[l]] - cVoltageAngles[EndBus[l]])
      +
      B[l] * sin(cVoltageAngles[StartBus[l]] - cVoltageAngles[EndBus[l]])
    )
    =
    0;

  #KVL +Q power at Start bus of lines
  subject to QKVLStart{l in LineDIFF}:
    cFlowStartQ[l]
    + B[l]*(cVoltages[StartBus[l]]^2)
    + cVoltages[StartBus[l]]*cVoltages[EndBus[l]]
    *
    (
      G[l] * sin(cVoltageAngles[StartBus[l]] - cVoltageAngles[EndBus[l]])
      -
      B[l] * cos(cVoltageAngles[StartBus[l]] - cVoltageAngles[EndBus[l]])
    )
    =
    0
    ;

  #kVL +P power at End bus of lines
  subject to PKVLEnd{l in LineDIFF}:
    cFlowEndP[l]
    - G[l]*(cVoltages[EndBus[l]]^2)
    + cVoltages[StartBus[l]] * cVoltages[EndBus[l]]
    *
    (
      G[l]* cos(cVoltageAngles[EndBus[l]] - cVoltageAngles[StartBus[l]] )
      +
      B[l]* sin(cVoltageAngles[EndBus[l]] - cVoltageAngles[StartBus[l]] )
    )
    =
    0
    ;
  #KVL +Q power at End bus of lines
  subject to QKVLEnd{l in LineDIFF}:
    cFlowEndQ[l]
    + B[l]*(cVoltages[EndBus[l]]^2)
    + cVoltages[StartBus[l]] * cVoltages[EndBus[l]]
    *
    (
      G[l]* sin(cVoltageAngles[EndBus[l]] - cVoltageAngles[StartBus[l]] )
      -
      B[l]* cos(cVoltageAngles[EndBus[l]] - cVoltageAngles[StartBus[l]] )
    )
    =
    0
    ;

  #KVL +P power at start bus of transformer
  subject to tPKVLStart{l in TransformerDIFF}:
    cFlowStartP[l]
    -G[l]*(cVoltages[StartBus[l]]^2)/(TapRatio[l]^2)
    +
    (cVoltages[StartBus[l]]*cVoltages[EndBus[l]]/TapRatio[l])
    *
    (
      G[l]*cos(cVoltageAngles[StartBus[l]] - cVoltageAngles[EndBus[l]] )
      +
      B[l]*sin(cVoltageAngles[StartBus[l]] - cVoltageAngles[EndBus[l]] )
    )
    =
    0
    ;

  #KVL +P power at end bus of transformer
  subject to tPKVLEnd{l in TransformerDIFF}:
    cFlowEndP[l]
    -G[l]*cVoltages[EndBus[l]]^2
    +
    cVoltages[StartBus[l]]*cVoltages[EndBus[l]]/TapRatio[l]
    *
    (
      G[l]* cos(cVoltageAngles[EndBus[l]] - cVoltageAngles[StartBus[l]]   )
      +
      B[l]* sin(cVoltageAngles[EndBus[l]] - cVoltageAngles[StartBus[l]]   )
    )
    =
    0
    ;

  #KVL +Q power at start bus of transformer
  subject to tQKVLStart{l in TransformerDIFF}:
    cFlowStartQ[l]
    + B[l]*(cVoltages[StartBus[l]]^2)/(TapRatio[l]^2)
    + cVoltages[StartBus[l]]/TapRatio[l]
    *
    (
      G[l]*sin(cVoltageAngles[StartBus[l]] - cVoltageAngles[EndBus[l]])
      -
      B[l]*cos(cVoltageAngles[StartBus[l]] - cVoltageAngles[EndBus[l]])
    )
    =
    0
    ;

  #KVL +Q power at end bus of transformer
  subject to tQKVLEnd{l in TransformerDIFF}:
    cFlowEndQ[l]
    + B[l]*(cVoltages[EndBus[l]]^2 )
    + cVoltages[StartBus[l]]*cVoltages[EndBus[l]]/TapRatio[l]
```

```
    *
    (
      G[l]*sin(cVoltageAngles[EndBus[l]] - cVoltageAngles[StartBus[l]])
      -
      B[l]*cos(cVoltageAngles[EndBus[l]] - cVoltageAngles[StartBus[l]])
    )
    =
    0
    ;

  #reference bus
  subject to RefBusZero:
    cVoltageAngles[RefBus] = 0;

  #Line and Transformer Thermal Limits
  subject to FlowLimitStart{l in LUTDIFF}:
    cFlowStartP[l]^2 + cFlowStartQ[l]^2 - cslackStart[l] = 0;
  subject to FlowLimitEnd{l in LUTDIFF}:
    cFlowEndP[l]^2 + cFlowEndQ[l]^2 - cslackEnd[l] = 0;

  ##########################################
  #Linking constraints to first stage
  ##########################################
  subject to BusVolCons{b in PVBUS}: cVoltages[b] - Voltages[b] = 0;
  #noteA: duplicate temporary set created for {g in Generator: Location[g]!=RefBus and (Location[g] in PVBUS)}
  subject to BusGenCons{g in Generator: Location[g]!=RefBus and (Location[g] in PVBUS) }: cPpowerGen[g] - PpowerGen[g] = 0;
}

minimize Total_Cost: 0.000001* sum{ g in Generator} (
    GenConst[g]
    + baseMVA*GenLin[g]*PpowerGen[g]
    + (baseMVA^2)*GenQuad[g]*(PpowerGen[g]^2)
);
```

Listing B.1: Security Constrained Optimal Power Flow Model

Appendix C

Asset Liability Management Model with Mean-Variance

```
param T;
set TIME ordered = 0..T;
set NODES;
param Parent{NODES} symbolic;
param Probs{NODES};
set ASSETS;
param Price{ASSETS};
param Return{ASSETS, NODES};
param Liability{TIME};
param InitialWealth;
param Gamma;  # transaction costs
param Lamda;  # risk-aversion parameter
var mu >=0;
block alm stochastic using (nd in NODES, Parent, Probs, st in TIME): {
  var x_hold{ASSETS} >= 0;
  var cash >= 0;
  stages {0}: {
    subject to StartBudget:
      (1+Gamma) * sum{j in ASSETS} (x_hold[j] * Price[j]) + cash = InitialWealth
          ;
  }
  stages {1..T}: {
    var x_bought{ASSETS} >= 0;
    var x_sold{ASSETS} >= 0;
    subject to Inventory{j in ASSETS}:
      x_hold[j] - (1+Return[j,nd]) * ancestor(1).x_hold[j] - x_bought[j] +
          x_sold[j] = 0;
    subject to CashBalance:
      cash + (1+Gamma) * sum{j in ASSETS} (Price[j] * x_bought[j]) - ancestor(1)
          .cash - (1-Gamma) * sum{j in ASSETS} (Price[j] * x_sold[j] ) = -
          Liability[st];
  }
  stages {T}: {
    var wealth >= 0;
    subject to FinalWealth:
      wealth - sum{j in ASSETS} (Price[j] * x_hold[j]) - cash = 0;
  subject to ExpPortfolioValue:
    Exp(wealth) - mu = 0;

    maximize objFunc: wealth - Lamda * ((wealth*wealth) - mu*mu);
  }
}
```

Listing C.1: Asset Liability Management Model with Mean-Variance

Bibliography

[1] Marco Colombo et al. “A structure-conveying modelling language for mathematical and stochastic programming”. In: *Mathematical Programming Computation* 1 (4 2009), pp. 223–247. ISSN: 1867-2949. URL: http://dx.doi.org/10.1007/s12532-009-0008-2.

[2] Marco Colombo et al. *Structure-Conveying Modeling Language User’s Guide*. Feb. 2011. URL: http://www.maths.ed.ac.uk/ERGO/sml/userguide.pdf.

[3] Robert Fourer, David M. Gay, and Brian W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press, 2002. ISBN: 0534388094. URL: http://www.worldcat.org/isbn/0534388094.

[4] Cornelis Antonius Christiaan Kuip. “Algebraic languages for mathematical programming”. In: *European Journal of Operational Research* 67.1 (1993), pp. 25–51.

[5] Emmanuel Fragniere and Jacek Gondzio. “Optimization modeling languages”. In: *Pardalos, P., Resende, M. Handbook of Applied Optimization* (2002), pp. 993–1007.

[6] Bruce A Murtagh. *Advanced linear programming: computation and practice*. Vol. 1. McGraw-Hill New York/London, 1981.

[7] William Orchard-Hays. “History of mathematical programming systems”. In: *Annals of the History of Computing* 6.3 (1984), pp. 296–312.

[8] Anthony Brook, David Kendrick, and Alexander Meeraus. “GAMS, a user’s guide”. In: *SIGNUM Newsl.* 23.3-4 (Dec. 1988), pp. 10–11. ISSN: 0163-5778. DOI: 10.1145/58859.58863. URL: http://doi.acm.org/10.1145/58859.58863.

[9] J. Bisschop and R. Entriken. “AIMMS the modeling system”. In: *Paragon Decision Techonology* (1993).

[10] Yves Colombani and Susanne Heipcke. “Mosel: an extensible environment for modeling and programming solutions”. In: *Proceedings of the Fourth International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR’02)*. Ed. by Narendra Jussien and Franc C. Laburthe. Le Croisic, France, Mar. 2002, pp. 277–290.

[11] *Gnu Linear Programming Kit*. http://www.gnu.org/software/glpk/.

[12] Tim Helge Hultberg. "FlopC++ An Algebraic Modeling Language Embedded in C++". English. In: *Operations Research Proceedings 2006*. Ed. by Karl-Heinz Waldmann and UlrikeM. Stocker. Vol. 2006. Operations Research Proceedings. Springer Berlin Heidelberg, 2007, pp. 187–190. ISBN: 978-3-540-69994-1. DOI: 10.1007/978-3-540-69995-8_31. URL: http://dx.doi.org/10.1007/978-3-540-69995-8_31.

[13] Miles Lubin and Iain Dunning. "Computing in Operations Research using Julia". In: *arXiv preprint arXiv:1312.1431* (2013).

[14] William E. Hart. *Pyomo: Python Optimization Modeling Objects*. https://software.sandia.gov/trac/coopr/wiki/Pyomo.

[15] William E. Hart, Jean-Paul Watson, and David L. Woodruff. *Pyomo: Modeling and Solving Mathematical Programs in Python*. Tech. rep. Sandia National Laboratories, 2012.

[16] E Weinan. *Principles of multiscale modeling*. Cambridge University Press, 2011.

[17] Christian Wernz and Abhijit Deshmukh. "Multiscale decision-making: Bridging organizational scales in systems with distributed decision-makers". In: *European Journal of Operational Research* 202.3 (2010), pp. 828–840.

[18] George B. Dantzig and Philip Wolfe. "The Decomposition Algorithm for Linear Programs". English. In: *Econometrica* 29.4 (1961), pp. 767–778. ISSN: 00129682. URL: http://www.jstor.org/stable/1911818.

[19] George B Dantzig and Philip Wolfe. "Decomposition principle for linear programs". In: *Operations research* 8.1 (1960), pp. 101–111.

[20] Jacques F Benders. "Partitioning procedures for solving mixed-variables programming problems". In: *Numerische mathematik* 4.1 (1962), pp. 238–252.

[21] Jacek Gondzio and Andreas Grothey. *Object Oriented Parallel Solver*. http://www.maths.ed.ac.uk/~gondzio/parallel/solver.html. July 2003.

[22] Jacek Gondzio and Andreas Grothey. "Exploiting structure in parallel implementation of interior point methods for optimization". In: *Computational Management Science* 6 (2 2009). 10.1007/s10287-008-0090-3, pp. 135–160. ISSN: 1619-697X. URL: http://dx.doi.org/10.1007/s10287-008-0090-3.

[23] Jacek Gondzio and Andreas Grothey. "Direct solution of linear systems of size 109 arising in optimization with interior point methods". In: *Proceedings of the 6th international conference on Parallel Processing and Applied Mathematics*. PPAM'05. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 513–525. ISBN: 3-540-34141-2, 978-3-540-34141-3. DOI: 10.1007/11752578_62. URL: http://dx.doi.org/10.1007/11752578_62.

[24] Cosmin Petra et al. *The solver - PIPS*. http://www.mcs.anl.gov/~petra/pips.html.

[25] Christian Valente et al. “Extending Algebraic Modelling Languages for Stochastic Programming”. In: *INFORMS J. on Computing* 21.1 (Jan. 2009), pp. 107–122. ISSN: 1526-5528. DOI: 10.1287/ijoc.1080.0282. URL: http://dx.doi.org/10.1287/ijoc.1080.0282.

[26] Robert Fourer and Leo Lopes. “StAMP: A Filtration-Oriented Modeling Tool for Multistage Stochastic Recourse Problems”. In: *INFORMS Journal on Computing* 21.2 (2009), pp. 242–256.

[27] Emmanuel Fragniere et al. “Structure Exploiting Tool in Algebraic Modeling Languages”. In: *Management Science* 46 (2000), pp. 1145–1158.

[28] Michael C. Ferris and Jeffrey D. Horn. “Partitioning Mathematical Programs for Parallel Solution”. In: *Mathematical Programming* 80 (1994), pp. 35–62.

[29] European Exascale Software Initiative. *Final report on roadmap and recommendations development.* http://www.eesi-project.eu. 2011.

[30] Joey Huchette, Miles Lubin, and Cosmin Petra. “Parallel algebraic modeling for stochastic optimization”. In: *Proceedings of the 1st First Workshop for High Performance Technical Computing in Dynamic Languages.* IEEE Press. 2014, pp. 29–35.

[31] David M. Gay. *Hooking Your Solver to AMPL.* Tech. rep. Bell Laboratories, Murray Hill, NJ, 1997.

[32] Feng Qiang and Andreas Grothey. *PSMG–A Parallel Structured Model Generator for Mathematical Programming.* Tech. rep. Technical report, School of Mathematics, University of Edinburgh, 2014.

[33] Xi Yang, Jacek Gondzio, and Andreas Grothey. “Asset liability management modelling with risk control by stochastic dominance”. In: *Journal of Asset Management* 11.2-3 (), pp. 73–93. ISSN: 1470-8272. DOI: 10.1057/jam.2010.8. URL: http://dx.doi.org/10.1057/jam.2010.8.

[34] D. Dentcheva and A. Ruszczynski. “Optimization with Stochastic Dominance Constraints”. In: *SIAM Journal on Optimization* 14.2 (2003), pp. 548–566. DOI: 10.1137/S1052623402420528. eprint: http://dx.doi.org/10.1137/S1052623402420528. URL: http://dx.doi.org/10.1137/S1052623402420528.

[35] Andreas Grothey Jacek Gondzio. “Parallel interior-point solver for structured quadratic programs: Application to financial planning problems”. In: *Annals of Operations Research* volume 152.1 (2007), 319–339(21).

[36] Arthur M Geoffrion. “Generalized benders decomposition”. In: *Journal of optimization theory and applications* 10.4 (1972), pp. 237–260.

[37] Andreas Grothey. “Decomposition Methods for Nonlinear Nonconvex Optimization Problems”. PhD thesis. University of Edinburgh, 2001.

[38] Robert J Vanderbei and David F Shanno. “An interior-point algorithm for nonconvex nonlinear programming”. In: *Computational Optimization and Applications* 13.1-3 (1999), pp. 231–252.

[39] Bora Tarhan, Ignacio E Grossmann, and Vikas Goel. “Stochastic programming approach for the planning of offshore oil or gas field infrastructure under decision-dependent uncertainty”. In: *Industrial & Engineering Chemistry Research* 48.6 (2009), pp. 3078–3097.

[40] Diana Cobos-Zaleta and Roger Z Ríos-Mercado. “A MINLP model for minimizing fuel consumption on natural gas pipeline networks”. In: *Proceedings of the XI Latin-Ibero-American conference on operations research*. 2002, pp. 90–94.

[41] Xiang Li et al. “Stochastic pooling problem for natural gas production network design and operation under uncertainty”. In: *AIChE Journal* 57.8 (2011), pp. 2120–2135.

[42] Manuel Soler et al. “En-route optimal flight planning constrained to pass through waypoints using MINLP”. In: *Proceedings of 9th USA/Europe Air Traffic Management Research and Development Seminar, Berlin*. 2011.

[43] Tom Schouwenaars et al. “Mixed integer programming for multi-vehicle path planning”. In: *European control conference*. Vol. 1. 2001, pp. 2603–2608.

[44] Xiang Li, Asgeir Tomasgard, and PaulI. Barton. “Nonconvex Generalized Benders Decomposition for Stochastic Separable Mixed-Integer Nonlinear Programs”. English. In: *Journal of Optimization Theory and Applications* 151.3 (2011), pp. 425–454. ISSN: 0022-3239. DOI: 10.1007/s10957-011-9888-1. URL: http://dx.doi.org/10.1007/s10957-011-9888-1.

[45] *OpenMPI*. www.open-mpi.org.

[46] *Flex*. http://www.gnu.org/software/flex/.

[47] *Bison*. http://www.gnu.org/software/bison/.

[48] Michael Bartholomew-Biggs et al. “Automatic differentiation of algorithms”. In: *Journal of Computational and Applied Mathematics* 124.1 (2000), pp. 171–190.

[49] R.L. Burden and J.D. Faires. *Numerical Analysis*. Brooks/Cole, Cengage Learning, 2011. ISBN: 9780538735643. URL: http://books.google.co.uk/books?id=KlfrjCDayHwC.

[50] Andreas Griewank et al. “On automatic differentiation”. In: *Mathematical Programming: recent developments and applications* 6 (1989), pp. 83–107.

[51] Bruce Christianson. “Automatic Hessians by reverse accumulation”. In: *IMA Journal of Numerical Analysis* 12.2 (1992), pp. 135–150. DOI: 10.1093/imanum/12.2.135. eprint: http://imajna.oxfordjournals.org/content/12/2/135.full.pdf+html. URL: http://imajna.oxfordjournals.org/content/12/2/135.abstract.

[52] Laurence Dixon. “Automatic differentiation: calculation of the Hessian Automatic Differentiation: Calculation of the Hessian”. In: *Encyclopedia of Optimization*. Springer, 2009, pp. 133–137.

[53] Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* Second. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2008. ISBN: 0898716594, 9780898716597.

[54] Assefaw H Gebremedhin et al. "Efficient computation of sparse Hessians using coloring and automatic differentiation". In: *INFORMS Journal on Computing* 21.2 (2009), pp. 209–223.

[55] Andreas Griewank, David Juedes, and Jean Utke. "ADOL-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++". In: *ACM Trans. Math. Softw.* 22.2 (June 1996), pp. 131–167. ISSN: 0098-3500. DOI: `10.1145/229473.229474`. URL: `http://doi.acm.org/10.1145/229473.229474`.

[56] Assefaw H Gebremedhin et al. "ColPack: Software for graph coloring and related problems in scientific computing". In: *ACM Transactions on Mathematical Software (TOMS)* 40.1 (2013), p. 1.

[57] Robert Mansel Gower and Margarida P Mello. *Hessian matrices via automatic differentiation.* Universidade Estadual de Campinas, Instituto de Matemática, Estatística e Computação Científica, 2010.

[58] RM Gower and MP Mello. "A new framework for the computation of Hessians". In: *Optimization Methods and Software* 27.2 (2012), pp. 251–273.

[59] Andrea Walther. "Computing sparse Hessians with automatic differentiation". In: *ACM Transactions on Mathematical Software (TOMS)* 34.1 (2008), p. 3.

[60] Robert Mansel Gower and Margarida Pinheiro Mello. "Computing the sparsity pattern of hessians using automatic differentiation". In: *ACM Transactions on Mathematical Software (TOMS)* 40.2 (2014), p. 10.

[61] *Community Portal for Automatic Differentiation.* `www.autodiff.org`. URL: `www.autodiff.org`.

[62] Feng Qiang. *AutoDiff Library.* `http://www.autodiff.org/?module=Tools&tool=AutoDiff_Library`.

[63] Gennadiy Rozental. *Boost Test Library.* `http://www.boost.org/doc/libs/1_34_1/libs/test/doc/components/utf/index.html`.

[64] R. Neidinger. "Introduction to Automatic Differentiation and MATLAB Object-Oriented Programming". In: *SIAM Review* 52.3 (2010), pp. 545–563. DOI: `10.1137/080743627`. eprint: `http://epubs.siam.org/doi/pdf/10.1137/080743627`. URL: `http://epubs.siam.org/doi/abs/10.1137/080743627`.

[65] Andrea Walther and Andreas Griewank. "Getting started with ADOL-C". In: *Combinatorial Scientific Computing* (2012), pp. 181–202.

[66] Joerg Walter and Mathias Koch. *Basic Linear Algebra Library.* Tech. rep. URL: `http://www.boost.org/doc/libs/1_57_0/libs/numeric/ublas/doc/index.html`.

[67] Paul Hovland and Christian Bischof. “Automatic differentiation for message-passing parallel programs”. In: *Parallel Processing Symposium, International*. IEEE Computer Society. 1998, pp. 0098–0098.

[68] Christian Bischof et al. “Parallel reverse mode automatic differentiation for OpenMP programs with ADOL-C”. In: *Advances in Automatic Differentiation*. Springer, 2008, pp. 163–173.

[69] R Tyrrell Rockafellar and Stanislav Uryasev. “Conditional value-at-risk for general loss distributions”. In: *Journal of banking & finance* 26.7 (2002), pp. 1443–1471.

[70] Harry Markowitz. “Portfolio selection”. In: *The journal of finance* 7.1 (1952), pp. 77–91.

[71] Nai-Yuan Chiang. “Structure-Exploitng Interior Point Methods for Security Constrained Optimal Power Flow Problems”. PhD thesis. University of Edinburgh, 2013.