Noname manuscript No.
(will be inserted by the editor)

PSMG–A Parallel Structured Model Generator for Mathematical Programming

Feng Qiang · **Andreas Grothey**

Received: date / Accepted: date

Abstract In this paper, we present PSMG–Parallel Structured Model Generator–an efficient parallel implementation of a model generator for the structure conveying modelling language (SML[4]). Unlike the earlier proof-of-concept implementation presented with SML, PSMG does not depend on AMPL.

The main purposes of PSMG are: to provide an easy to use framework for modelling and generating large scale nested structured problems, including multi-stage stochastic problems; to eliminate a current bottleneck in the problem generation stage by parallelising the problem generation; and also to offer a generic solver interface that can be easily linked with different structure exploiting optimization solvers such as decomposition or interior point based solvers. As far as we are aware, PSMG is the first processor for an algebraic modelling language that is capable of generating the problem in parallel. PSMG's solver interface delegates to the solver the task of deciding how to distribute problem parts to processors thus achieving better data locality and load balancing.

We also report performance benchmark test on two different structured problems of various sizes. The results show that PSMG achieves good parallel efficiency on up to 96 processes. The distributed memory design for PSMG also enables the generation of problems that are too large to be processed on a single node due to memory restriction.

Keywords Algebraic Modelling Language · Stochastic Programming · Parallel Problem Generation · Structure exploitation

F. Qiang
School of Mathematics, University of Edinburgh, Edinburgh, UK
E-mail: F.Qiang@sms.ed.ac.uk

A. Grothey
School of Mathematics, University of Edinburgh, Edinburgh, UK

1 Introduction

Consider a mathematical programming problem in the general form,

$$\min_{x \in X} f(x) \text{ s.t. } g(x) \leq 0, \tag{1}$$

where $X \subseteq \mathbb{R}^n$, $f : \mathbb{R}^n \to \mathbb{R}$, $g : \mathbb{R}^n$ 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 $f(x), g(x), \nabla f(x), \nabla g(x), \nabla^2 f(x)$, and $\nabla^2 g_i(x)$ are required at the current iterate $x \in X$. It is often the case that the solver delegates the the computation work of these values to an algebraic modelling language (AML), such as AMPL[7], GAMS[3], etc. There are many AMLs available either commercially or as open-source [3,1,2,11,10]. By using an AML, the modeller can focus on specifying the underlining mathematical relations of the problem using common algebraic abstraction such as set, parameter, variables and etc., rather than having to explicitly write code for computing those values. A concise and meaningful high level mathematical abstraction can be easily developed and maintained by the modeller.

As the need for accurate modelling increases, the size and complexity of optimization models increases likewise. It is common that large scale, real life problems are 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.

For many years, researchers have been working on using such problem structure to parallelise optimization solvers in order to speed up their solution. Algorithms, such as Dantzig-Wolfe and Benders decomposition, and interior point solvers, such as OOPS[9] and PIPS[12] can take advantage of such structure to speed up the solver, enable the solution of larger problems and facilitate parallelism. To use such techniques the solver needs to be aware of the problem structure. Current modelling languages, however, do not usually have the capabilities to express such structure and convey it on to the solver.

Those modelling system that do offer capabilities to express structure, are either not general approaches (*eg.* specialised to stochastic programming), or they require assembling the complete unstructured model before annotations can be parsed, which is infeasible for large problems. See [4] for a more detailed review of these approaches. On the other hand, SML[4] is designed as a generic AML for describing any structured problem.

The total time required for solving 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. While 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 parallelisation,

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[6].

In this paper we present PSMG, a model generator for the structure conveying modelling language SML[4]. Together with SML, PSMG forms a modelling system that is targeted at the efficient parallel generation of large scale structured optimization problems.

PSMG uses the structure conveying modelling language SML[4] which enables the modeller to describe any nested problem structure using high level abstraction. The initial proof-of-concept generator for SML described in [4]was implemented as an AMPL pre- and post-processor. It used the AmplSolver[8] library for computing the function and derivative evaluations through `*.nl`-files. Thus it used a file system based interface for communicating between the solver and AMPL. This has proven inefficient for parallel processing; in PSMG this dependency on AMPL has been removed.

The elimination of the file system interface has resulted in huge performance gains in PSMG for generating large structured problems both in serial and parallel.

SML describes a structured optimization problems in terms of blocks, sub-blocks and relations between them. These blocks form natural *atoms* of the model that can be used for parallelisation. However the best distribution of these atoms to processors for maximal parallel efficiency depends on many factors such as relative size of blocks, relations between blocks and data locality issues. Some of these factors depend on the algorithm used in the optimization solver. The model generator therefore does not have sufficient information to decide on the optimal allocation of problem components to processors.

Because of these considerations PSMG adopts a *solver driven work assignment* approach: When PSMG parses a model it creates a data structure (the *Expanded Model Tree* that carries minimal information about the number and size of blocks and their relation with each other. This data structure serves as interface to the solver and provides the solver with sufficient information to decide on the allocation of problem parts to processors. Subsequently the solver can initiate the further processing of the model (and later the evaluation of functions and their derivatives) through call-back functions that are executed *only* on those processors where the relevant information is needed.

The subsequent sections of this paper are organised as follows: Section 2 presents two examples of structured problems, and also demonstrates the model and data syntax used by SML; Section 3 presents some key design considerations for PSMG to build an internal representation of the problem structure. We also use the examples in Section 2 to explain these concepts in PSMG; Section 4 explains PSMG's parallel solver interface in detail; Section 5 presents benchmarking results regarding parallel efficiency and memory usage for PSMG; Finally we present our conclusions in Section 6.

2 Modelling of Structured Problems in SML/PSMG

In this section, we present two nested structured problems modelled in SML. These also appear in [4] but are repeated here for sake of completeness. The examples we use are Multi-Commodity Survivable Network Design (MSND) and Asset Liability Management with Second Order Stochastic Dominance constraints (ALM-SSD). We will use the examples to showcase how SMLs block modelling tools can be used to express problem structure and also in order to evaluate PSMG's performance.

2.1 Modelling problem with nested-block structure: MSND

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. This problem is fully described in the previous SML paper[4]. For ease of reference, the mathematical formulation for this problem is given below in (2). The sets, parameters and decision variables used in this model formulation are explained below

Sets

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

Parameters

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

Variables

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

$$\min \sum_{l \in \mathscr{E}} c_l s_l \tag{2a}$$

$$\text{s.t. } A^{(n,i)} x_k^{(n,i)} = b_k \qquad \forall k \in \mathscr{C}, i \in \mathscr{N} \tag{2b}$$

$$A^{(e,j)} x_k^{(e,j)} = b_k \qquad \forall k \in \mathscr{C}, j \in \mathscr{E} \tag{2c}$$

$$\sum_{k \in \mathscr{C}} x_{k,l}^{(e,j)} \leq C_l + s_l \qquad \forall j \in \mathscr{E}, l \in \mathscr{E} \tag{2d}$$

$$\sum_{k \in \mathscr{C}} x_{k,l}^{(n,i)} \leq C_l + s_l \qquad \forall i \in \mathscr{N}, l \in \mathscr{E} \tag{2e}$$

$$x \geq 0, s_j \geq 0 \ \ \forall j \in \mathscr{E}. \tag{2f}$$

The constraints (2b) and (2c) are flow balance constraints for each node or edge failure scenario respectively. These two sets of constraints ensure that demand is satisfied for each commodity in the reduced network. The constraints (2d) and (2e) are edge capacity constraints, which ensure the flow passing through each edge does not exceed the capacity limit of the edge.

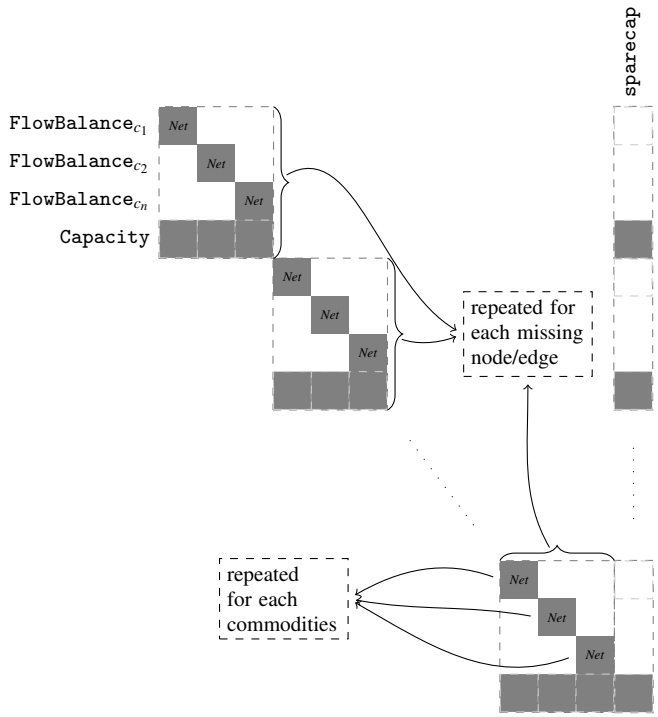

Fig. 1: The constraint matrix structure of MSND problem.

The constraint matrix of this problem is given in Figure 1. As it can be seen the problem displays a nested block structure, a fact that is not immediately obvious from the mathematical description.

The PSMG model for this problem is given in Model 1. This model uses `block`-statements to describe repeated common structures that build up the full problem (refer to [4] for a detailed description of the language syntax). The problem can be constructed by firstly modelling a basic building block, namely the node-edge incident matrix formed by the flow balance constraint. Then, we can repeat these blocks for all commodities. Finally we can repeat the nested block again for each missing edge and node cases to build the full problem: The flow balance constraints in lines 10–11 and 22–23 describe a node-edge incidence matrix for a network with a missing edge or node respectively. The `block`-statements at lines 8 and 20 repeat this structure for each commodity, while the `block`-statements at lines 6 and 18 repeat for each missing edge or node, creating a nesting of subproblem blocks. We also 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.

In order to generate a problem instance, the modeller needs to provide a data file for the MSND model. A sample data file with 3 nodes and 3 arcs is given in Data 2.

```
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  block MCNFArcs{a in ARCS}: {
7    set ARCSDIFF := ARCS diff {a};
8    block Net{k in COMM}: {
9      var Flow{REMARCS}>=0;
10     subject to FlowBalance{i in NODES}:
11       sum{j in REMARCS:arc_target[j]==i} Flow[j]
12       - sum{j in REMARCS:arc_source[j]==i} Flow[j] = b[k,i];
13   }
14   var capslack{REMARCS} >= 0;
15   subject to Capacity{j in REMARCS}:
16     sum{k in COMM} Net[k].Flow[j] = basecap[j] + sparecap[j] + capslack[j];
17 }
18 block MCNFNodes{n in NODES}: {
19   set REMNODES := NODES diff {n};
20   set REMARCS := {m in ARCS:arc_source[m]!=n and arc_target[m]!=n};
21   block Net{k in COMM}: {
22     var Flow{ARCS} >= 0;
23     subject to FlowBlance{i in REMNODES}:
24       sum{j in REMARCS:arc_target[j]==i} Flow[j]
25       - sum{j in REMARCS:arc_source[j]==i} Flow[j] = b[k,i];
26   }
27   subject to Capacity{j in REMARCS}:
28   sum{k in COMM} Net[k].Flow[j] <= basecap[j] + sparecap[j];
29 }
30 minimize costToInstall: sum{x in ARCS} sparecap[x]*cost[x];
```

Model 1: Model file for MSND problem.

```
1  set NODES := N1 N2 N3 ;
2  set ARCS := A1 A2 A3 ;
3  param: cost basecap arc_source arc_target :=
4    A1 1 4 N1 N2
5    A2 4 7 N1 N3
6    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;
```

Data 2: Sample data file for MSND problem.

2.2 Modelling a multistage stochastic programming problem: ALM-SSD

Here we present an example of a stochastic programming problem, namely an Asset & Liability Management problem with second order stochastic dominance constraints (ALM-SSD) [14].

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

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

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

$$\max \quad \sum_{i \in \mathscr{L}_T} p_i \Big(\sum_{j \in \mathscr{A}} V_j x^h_{i,j} + c_i \Big) \tag{3a}$$

$$\text{s.t.} \quad (1+\gamma) \sum_{j \in \mathscr{A}} (x^h_{0,j} V_j) + c_0 = I, \tag{3b}$$

$$c_i + (1+\gamma) \sum_{j \in \mathscr{A}} (x^h_{i,j}) = c_{\pi(i)} + (1-\gamma) \sum_{j \in \mathscr{A}} (V_j x^s_{i,j}), \qquad \forall i \neq 0, i \in \mathscr{N} \tag{3c}$$

$$(1+r_{i,j}) x^h_{\pi(i),j} + x^b_{i,j} - x^s_{i,j} = x^h_{i,j}, \qquad \forall i \neq 0, i \in \mathscr{N}, j \in \mathscr{A} \tag{3d}$$

$$\sum_{j \in \mathscr{A}} (V_j x^h_{i,j}) + c_i + b_{i,l} \geq N_l, \qquad \forall i \neq 0, i \in \mathscr{N}, l \in \mathscr{F} \tag{3e}$$

$$\sum_{i \in \mathscr{L}_s} p_i b_{i,l} \leq M_{l,b}, \qquad \forall b \in \mathscr{B}, l \in \mathscr{F} \tag{3f}$$

$$x \geq 0,\ b_{i,l} \geq 0 \qquad \forall l \in \mathscr{F}, \forall i \neq 0, i \in \mathscr{N} \tag{3g}$$

The constraint matrix of this problem (illustrated in Figure 3 for the scenario tree in Figure 2) has the nested block angular structure typical for 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.

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

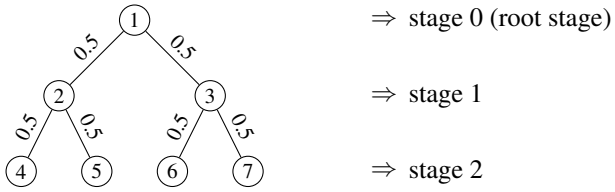

Fig. 2: Scenario tree structure declared in Data 4. The numbers on the tree edges are the transition probability declared in parameter `Probs`.

We can contrast the mathematical description with the model file in SML (Model 3). SML provides a `stochastic`-block (line 15) to describe a model block which is repeated for every node of a scenario tree. The arguments of the `stochastic block` statement are used to describe the shape of the scenario tree. Further `stage`-blocks can be used to specify that some entities should only be present for specific stages. A sample 3-stage scenario tree structure is given in Data 4. The scenario tree declared in this data file is visualised in Figure 2. Note that due to specifying the parameters describing the scenario tree as arguments to the `stochastic block` statement, the abstract mathematical relation among stages can be separated from the shape of the scenario tree.

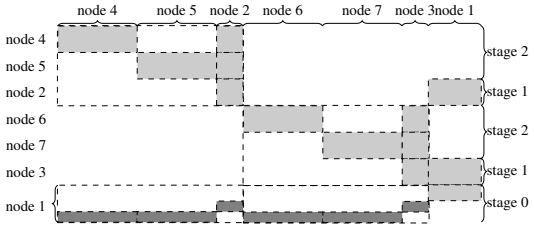

Fig. 3: The constraint matrix structure of ALM-SSD problem based on scenario tree structure declared in Data 4

Constraint (3b) in above formulation is the start-budget constraint and only appears in root stage (line 20 in Model 3). Constraints (3c) and (3d), lines 28–30 in the model file, are cash-balance and inventory constraints respectively. They are repeated in every nodes other than the root stage node. The last two constraints (3e) and (3f) are the linear formulation of second-order stochastic dominance (line 33 and 36 in Model 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). Our benchmarking results in Section 5 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
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 block alm stochastic using (nd in NODES, Parent, Probs, st in TIME): {
16    var x_hold{ASSETS} >= 0;
17    var shortfall{BENCHMARK} >= 0;
18    var cash >=0 ;
19    stages {0}: {
20       subject to StartBudget:
21          (1+Gamma)*sum{a in ASSETS} (x_hold[a]*Price[a]) + cash  =
                InitialWealth;
22    }
23    stages {1..T}: {
24       var x_sold{ASSETS} >= 0;
25       var x_bought{ASSETS} >= 0;
26
27       subject to CashBalance:
28          ancestor(1).cash - (1-Gamma)*sum{a in ASSETS} (Price[a]*x_sold[a])
29          = cash + (1+Gamma)*sum{a in ASSETS} (Price[a]*x_bought[a]);
30       subject to Inventory{a in ASSETS}:
31          x_hold[a] - Return[a,nd] * ancestor(1).x_hold[a] - x_bought[a] +
                x_sold[a] = 0;
32
33       subject to StochasticDominanceSlck{l in BENCHMARK}:
34          sum{a in ASSETS}(Price[a]*x_hold[a]) + cash + shortfall[l] >=
                VBenchmk[l]*InitialWealth;
35
36       subject to StochasticDominanceExp{l in BENCHMARK}:
37           Exp(shortfall[l]) <= HBenchmk[l]*InitialWealth;
38    }
39    stages {T}:{
40       var wealth >= 0;
41
42       subject to FinalWealth:
43          wealth - sum{a in ASSETS} (Price[a]*x_hold[a]) - cash = 0;
44       maximize objFunc: wealth;
45    }
46 }
```

Model 3: Model file for ALM-SSD problem.

```
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;
```

Data 4: Sample data file for ALM-SSD problem.

3 Model processing in PSMG

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 (in the form of the *prototype* and *expanded model tree* described below) of the information describing the problem with minimal processing. This skeletal description is restricted to information that the solver needs to decide on the distribution of problem components to processors, namely: the structure of the problem *ie.* number and type of blocks and their relationship, and the size of each block. The indexing set declared for the constraints and variable will not be expanded, but only the cardinality of the sets are counted and stored as the problem's dimension. This phase is done in unison on every processor and the result (the expanded model tree) is passed to the solver. Due to the minimal processing involved, this phase is very fast. According to our tests, it takes less than a second to process problems of more than a million variables and constraints.

The second stage is the parallel problem generation stage, where the solver requests the specific blocks or sub-problems to be generated on the processor where they are needed. Major work in processing the problem, such as the expansions of

indexing expressions used in the definition of variables and constraints and the evaluation of the constraint Jacobian entries is happening at this stage.

In the rest of this section, we discuss the main design components involved in the structure building stage. Issues regarding the parallel processing stage are discussed in next section (Section 4).

3.1 The Prototype & Expanded Model Tree

The prototype model tree is PSMG's minimal internal representation of the problem's high level structure as defined in the model file. The tree structure is a natural representation for the nested `block`-statements declared in the model file. 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, constraints or sub-blocks declared at the corresponding block. At this point the description of the entities is very much as described in the model file: they are analysed with respect to their syntactical components (*ie.* 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. The prototype model tree encodes the problem and sub-problems dependencies as described by the block statements in the model file, while each node of the tree serves as a prototype description of the entities in this block that are to be made concrete at the later (parallel) processing stage. Each tree-node is also associated with an indexing expression which will be used for generating the expanded model tree. Figure 4 demonstrates the prototype tree structure for the MSND problem. The tree indexing expression associated with each of the tree nodes is also shown in Figure 4.

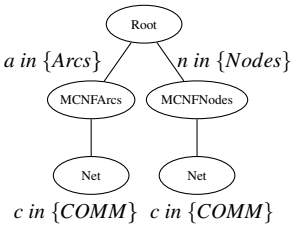

Fig. 4: The prototype model tree for MSND problem specified in Model 1.

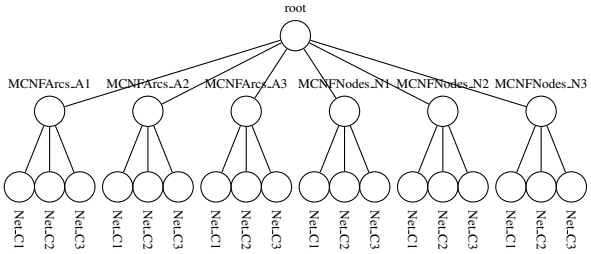

Fig. 5: The expanded model tree of an MSND problem instance constructed by PSMG using data file from Data 2.

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 submodels in the prototype model tree according to the associated indexing expression. For example, the indexing expression '`a in {Arcs}`' in the prototype tree (Figure 4) is expanded into three nodes (`MCNFArcs_A1`, `MCNFArcs_A2` and `MCNFArcs_A3`) in the expanded model tree (Figure 5). Other indexing set, such as the one used in the

variables declarations (line 5,9, 13, etc in Model 1), parameter declarations (line 4 in Model 1) and constraints declaration (line 10, 11, 14, 15, etc in Model 1) are not expanded and remain as high level template in the prototype tree node. However the cardinality of the total of variables and constraints of each sub-models are counted to provide the problem's dimension. 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 also has a pointer to the corresponding prototype 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 of the prototype model tree.

Figure 5 demonstrates an expanded model tree instance constructed by PSMG after reading the data file from Data 2 for the MSND model in Model 1. This expanded model tree is generated by repeating `MCNFArcs, MCNFNodes` and `Net` blocks for each set value of their corresponding indexing set.

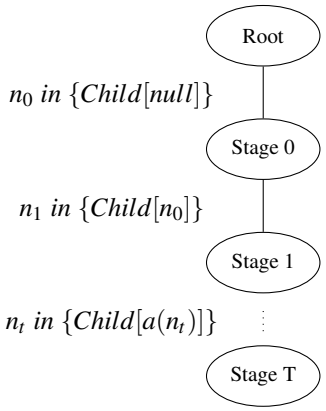

Fig. 6: The prototype model tree for ALM-SSD problem specified in Model 3. The indexing expression associated with each tree node is implicitly constructed by PSMG.

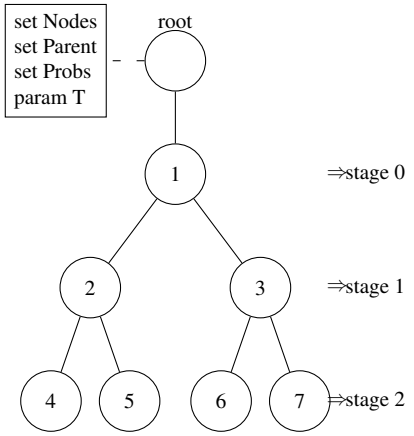

Fig. 7: Expanded model tree for ALM-SSD problem with scenario tree given in Data 4.

In the case of a *stochastic problem*, the creation of the prototype model tree requires some processing of the information from the model and data files. The number of stages (*ie.* the depth of the prototype tree) is part of the problem data and potentially given by an expression that needs expansion. Further the list of entities that are associated with every stage need to be extracted from the model description. This translation technique is also used in the implementation of SML[4]. PSMG again adds an indexing expression for each node of the prototype tree corresponding to the

children of a particular node in the scenario tree (shown in Figure 6). This indexing expression is used to generate the expanded model tree.

Figure 6 demonstrates the prototype tree structure for the ALM-SSD problem in Model 3.

The prototype 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.

The process of constructing the expanded model tree from the prototype tree is the same for both `block` and `stochastic block` models.

Figure 7 demonstrates an expanded model tree instance constructed by PSMG with the scenario tree structure specified from Data 4, where n_0 *in* $\{Child[null]\}$ is expanded to the node 1, $Child[1]$ is evaluated to a node set $\{2,3\}$, $Child[2]$ is evaluated to node set $\{4,5\}$ and $Child[3]$ is evaluated to node set $\{6,7\}$.

Once the expanded model tree is constructed by PSMG, it is passed to the solver. The solver can traverse the expanded model tree recursively to retrieve the structure information and set up the problem for parallel generation. The expanded model tree thus acts as an interface between PSMG and the solver.

4 Parallel Solver Interface

In order to avoid unnecessary communication it is evident that function and derivative evaluation routines for a particular part of the problem (and by extension the generation of the necessary data), should be performed on the processor that is also assigned to this part of the problem by the solver. In addition, allocation of sub-problems to processors should take into account load balancing and minimize 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 will 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.

4.1 On demand problem generation

Figure 8 illustrates the overall work-flow between PSMG and the solver.

The steps of in the work-flow diagram in Figure 8 can be summarised as below:

1. PSMG reads model file and data file.
2. PSMG extracts the prototype tree and builds the expanded model tree to describe the problem structure information (as described in Section 3.1).

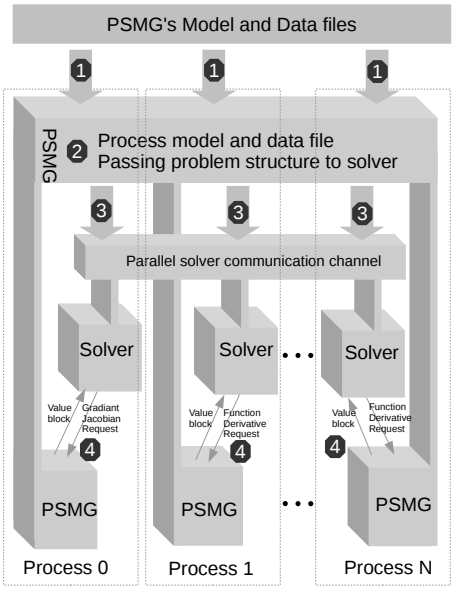

Fig. 8: The PSMG workflow with a parallel optimization solver.

3. PSMG passes the expanded model tree to the parallel solver. The solver will then decide how to distribute the problem parts, *ie.* 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. 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.

4.2 Callback function interface with solver

PSMG uses the expanded model tree as the interface for communicating the problem structure to the solver. PSMG's solver interface offers C and C++ call-back functions (implemented as member functions or properties of the expanded tree node object) for retrieving information such as function and derivative values for the model components.

Further details about PSMG's solver interface can be found in the technical document [13].

4.3 On-demand sub-model processing

When PSMG is first asked to evaluate function values or derivatives of a sub-block or sub-problem, additional processing needs to take place. This includes expansion of the indexing set used in the constraint and variables declaration, expansion of the indexing sets of sum expressions, evaluation of the set and parameter declarations, and initialisation of the variables.

This also results in additional memory having to be allocated for storing variables, constraint expressions and temporary set and parameter values.

Such processing is only done when initiated by a call-back on a specific processor. This "lazy" approach of data computation guarantees that processing power and memory is only used when and where it is necessary.

The names for the local variables and constraints and values are not stored in the data memory, but rather can be dynamically generated upon request from the solver.

5 Performance evaluation

5.1 Serial performance

We have used two test problem sets for our experiments. The first is a set of MSND problems 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 second one is a series of large scale stochastic programming 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 Model 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.

All tests are performed on a machine with Intel i5-3320M CPU and a solid state drive. The problem sizes and their corresponding generation time using PSMG are listed in Table 1. 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 1. For all the MSND and ALM-SSD problem instances, PSMG is able to build the problem structure within 1 second.

In Figure 10 we additionally plot the total problem generation time against the problem size (represented by the number of variables) for both 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 Time(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	23	≤ 1	23
	15	3,215,955	409,770	9,445,590	34	≤ 1	34
	20	4,220,805	479,370	12,460,140	48	≤ 1	48
	25	5,225,655	548,970	15,474,690	57	≤ 1	57
	30	6,230,505	618,570	18,489,240	67	≤ 1	67
	35	7,235,355	688,170	21,503,790	79	≤ 1	79
	40	8,240,205	757,770	24,518,340	91	≤ 1	91
	45	9,245,055	827,370	27,532,890	102	≤ 1	102
	50	10,249,905	896,970	30,547,440	114	≤ 1	114
ALM-SSD Problem Instance							
Number of Scenarios	Number of benchmarks						
441	21	34,262	15,268	169,817	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

Table 1: Problem sizes and PSMG problem generation time for MSND and ALM-SSD problem instances.

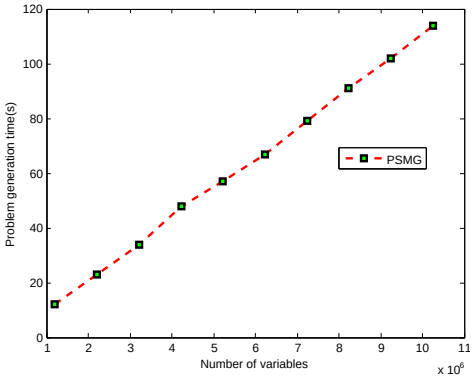

Fig. 9: Plot for PSMG's problem generation time for the MSND problems in Table 1.

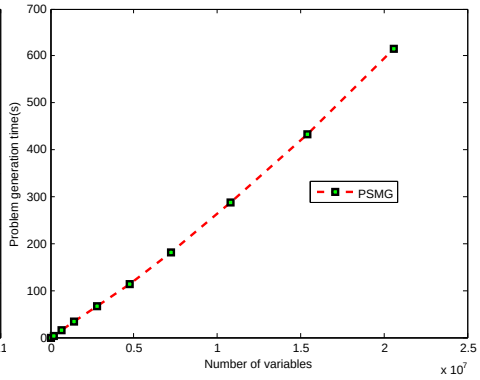

Fig. 10: Problem generation time for the ALM-SSD problems in Table 1

5.2 Comparison with SML-AMPL

We are aware that our implementation will 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 realised from exploiting parallelisation.

The proof-of-concept implementation of a model generator for SML described in [4] (henceforth referred to as SML-AMPL), was implemented as a pre- and post-processor for AMPL. After analysing the problem structure as defined by the block-statements, SML-AMPL would create a self-contained submodel file for each block and process this by AMPL creating a `*.nl`-file for every block. The solver interface would then access these files through the `AmplSolver` library [8].

It turns out that the pre- and post-processing step takes 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 parallelisation. PSMG was developed to address these issues.

PSMG is designed specifically to model and generate structured problems with an in-memory solver interface design. The previous SML implementation (SML-AMPL) has the same design goal but uses AMPL as an backend. Therefore, we compared the serial performance of PSMG to SML-AMPL.

In Table 2 and Figure 11 we compare the problem generation time of PSMG with SML-AMPL for the MSND test problem set. For SML-AMPL we report the time taken up by the pre-processing stage which finishes when all `*.nl`-files are written (Column 9) and the time taken to fill each matrix block in the problem with numerical values (Column 10). The total time for SML-AMPL to generate the problem is listed in Column 8. For all of our test problems SML-AMPL can barely finish the pre-processing stage in the time it takes PSMG for the whole problem generation. An even more extensive amount of time is needed to generate the values for the matrix blocks. 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 running even slower on a system with a slower disk. 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	Structure Setup	Problem Generation
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 2: Comparison of PSMG with SML-AMPL for MSND problem instances.

5.3 Parallel Efficiency

On a node with 4GB memory, plain AMPL will not be able to generate an MSND problem with 100 commodities on a network comprising 30 nodes and 435 edges due running out of memory. This problem (*msnd30_100* - twice as large as the largest MSND problem in Table 1) 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.

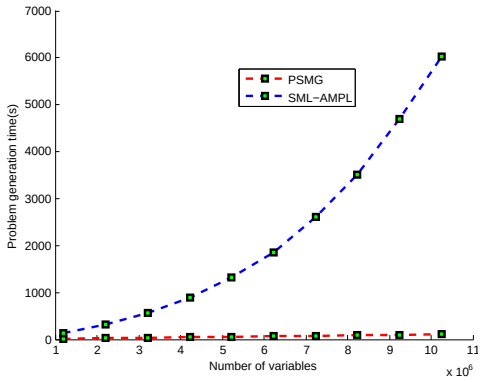

Fig. 11: Problem generation time of PSMG vs. SML-AMPL. for MSND problems in Table 2

We have also performed parallel scaling benchmarks for a large ALM-SSD problem (the largest problem in Table 1). This problem has 44944 scenarios, $20,591,783$ variables and $10,115,157$ constraints.

Tables 3 and 4 show the problem generation time and the resulting speed-up 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 12 and 13. 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 that 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 an 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 lack of perfect load balancing (note that the number of sub-blocks in both example is not divisible by 96).

5.4 PSMG Memory Usage Analysis

We have also measured the per processor and total memory usage of PSMG for generating problem *msnd30_100*. The memory usage in each PSMG process is composed of the memory used for storing the problem structures (prototype model tree and expanded model tree) and the problem data that required for computing 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 processes. We define the memory overhead to be this (*ie.* the non-distributable) part of the total memory usage.

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

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

This memory usage data is presented in Table 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 14 and 15 correspondingly.

For this example the non-distributable part requires about 40 MB.

This consists of memory used for the prototype model tree (24.5 KB) and the expanded model tree (39.93 MB). 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 for storing 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 enabling the

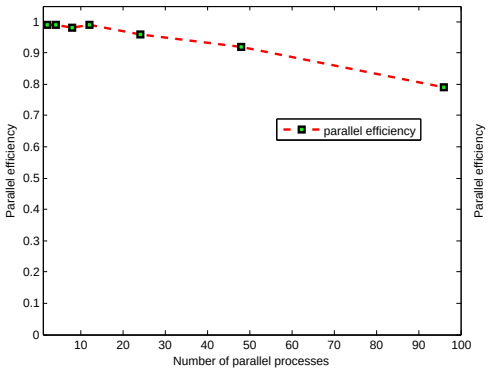

Fig. 12: Parallel efficiency plot for MSND problem.

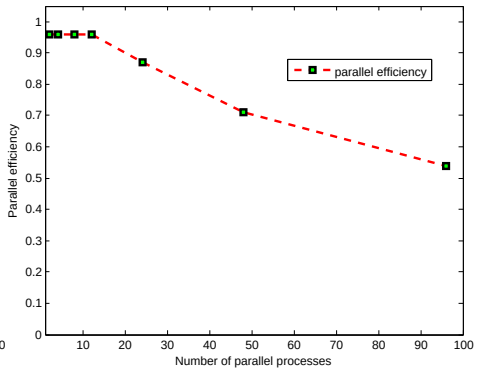

Fig. 13: Parallel efficiency plot for ALM-SSD problem.

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

6 Conclusions

In this paper, 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 optimization problems and stochastic programming problems.

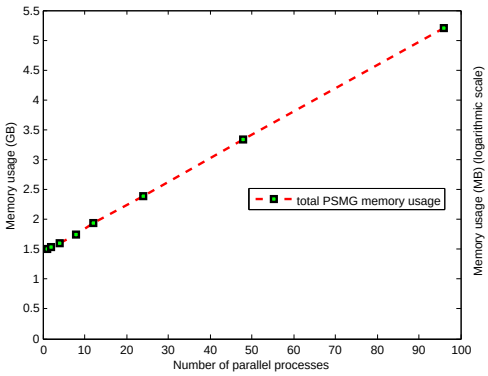

Fig. 14: Total memory usage plot for generating problem *msnd30_100*.

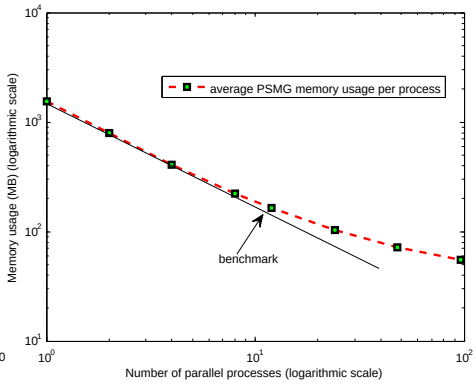

Fig. 15: Per processor memory usage for generating problem *msnd30_100*.

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. 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. The majority of the work of the problem generation is at the second stage – evaluation of constraint functions and derivatives and building the necessary data structures. PSMG is able to use the problem structure information to parallelise this second stage of the problem generation process. As far as we are aware PSMG is the only model generator for an algebraic modelling language that can pass the structure information to the solver and uses this information to parallelise the model generation process itself.

PSMG also features a novel parallel interface design: solver driven work assignment, that enables the parallel solver to achieve load balancing and data locality in order to minimize the amount of data communications among parallel processes. We have illustrated that by paying a small memory overhead, PSMG can implement solver driven problem distribution to achieve the on-demand problem generation and further eliminate inter-processor communication in both the model generation and function evaluation stages.

The performance evaluation of PSMG shows good parallel efficiency both in terms of speed and memory usage. 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.

In this paper, the examples we used are linear programming (LP) problems. We used LP examples to demonstrate the design concept and performance benchmark of PSMG. PSMG's interface design can also be extended to support Quadratic Programming (QP) and Nonlinear Programming (NLP) problems. This requires addi-

tional work for tracking cross-product between variables among every sub-blocks or sub-problems, whereas each sub-blocks in the constraint matrix of a LP problem are always separable (*ie.* independent from variables of other blocks). The extension to NLP and QP would also requires Hessian evaluation routines. However the discussion of this extension is beyond the scope of this paper and is left for a follow-up paper.

References

1. GLPK: Gnu linear programming kit. http://www.gnu.org/software/glpk/
2. Bisschop, J., Entriken, R.: AIMMS the modeling system. Paragon Decision Techonology (1993)
3. Brook, A., Kendrick, D., Meeraus, A.: GAMS, a user's guide. SIGNUM Newsl. **23**(3-4), 10–11 (1988). DOI 10.1145/58859.58863
4. Colombo, M., Grothey, A., Hogg, J., Woodsend, K., Gondzio, J.: A structure-conveying modelling language for mathematical and stochastic programming. Mathematical Programming Computation **1**, 223–247 (2009)
5. Dentcheva, D., Ruszczynski, A.: Optimization with stochastic dominance constraints. SIAM Journal on Optimization **14**(2), 548–566 (2003). DOI 10.1137/S1052623402420528
6. European Exascale Software Initiative: Final report on roadmap and recommendations development. `http://www.eesi-project.eu` (2011)
7. Fourer, R., Gay, D.M., Kernighan, B.W.: AMPL: A Modeling Language for Mathematical Programming. Duxbury Press (2002)
8. Gay, D.M.: Hooking your solver to AMPL. Tech. rep., Bell Laboratories, Murray Hill, NJ (1997)
9. Gondzio, J., Grothey, A.: Exploiting structure in parallel implementation of interior point methods for optimization. Computational Management Science **6**, 135–160 (2009). 10.1007/s10287-008-0090-3
10. Hart, W.E.: Pyomo: Python optimization modeling objects. `https://software.sandia.gov/trac/coopr/wiki/Pyomo`
11. Hultberg, T.H.: FlopC++ an algebraic modeling language embedded in C++. In: K.H. Waldmann, U. Stocker (eds.) Operations Research Proceedings 2006, *Operations Research Proceedings*, vol. 2006, pp. 187–190. Springer Berlin Heidelberg (2007). DOI 10.1007/978-3-540-69995-8_31
12. Petra, C., Anitescu, M., Lubin, M., Zavala, V., Constantinescu, E.: The solver - PIPS. `http://www.mcs.anl.gov/~petra/pips.html`
13. Qiang, F., Grothey, A.: PSMG design and solver interface (2014). URL `http://www.maths.ed.ac.uk/~fqiang/papers/psmg_interface.pdf`
14. Yang, X., Gondzio, J., Grothey, A.: Asset liability management modelling with risk control by stochastic dominance. Journal of Asset Management **11**(2-3), 73–93. DOI 10.1057/jam.2010.8