PSMG: A Parallel Problem Generator for a Structure Conveying Modelling Language for Mathematical Programming*

Andreas Grothey[1] and Feng Qiang[1]**

School of Mathematics, University of Edinburgh.
Emails: A.Grothey@ed.ac.uk, Feng.Qiang@sms.ed.ac.uk

Abstract. In this paper, we present PSMG – Parallel Structured Model Generator – a parallel implementation of a model generator for the structure conveying modelling language SML[1]. PSMG analyses the structure of an optimization problem given as an SML model file and uses this information to parallelise the model generation process itself. As far as we are aware PSMG is the only algebraic modelling language that can perform parallel problem generation.
PSMG offers an interface that can be linked in parallel with many different categories of structure exploiting optimization solvers such as interior point or decomposition based solvers. One of the features of this interface is that the decision on how to distribute problem parts to processors can be delegated to the solver thus enabling better data locality and load balancing.
We also present performance benchmark result for PSMG. The benchmarking results show that PSMG achieves good parallel efficiency on up to 256 processes. They also show that exploitation of parallelism enables the generation of problems that cannot be processed on a single node due to memory restrictions.

Keywords: modelling language, parallel, mathematical programming, problem generation,structure exploitation

1 Introduction

Mathematical Programming is an important tool for decision makers. As computing power and need for accurate modelling increases, the size and complexity of optimization models increases likewise. For many years, researchers have been working on parallel optimization solvers to speed up their solution.

* This work has made use of the resources provided by the Edinburgh Compute and Data Facility (ECDF). (http://www.ecdf.ed.ac.uk/). The ECDF is partially supported by the eDIKT initiative (http://www.edikt.org.uk).
** Fully supported by a Principal's Career Development Scholarship from the University of Edinburgh

In most cases problems are modelled with an algebraic modelling language (AML), such as AMPL[2], GAMS[3], etc to enable fast development and maintainability. However, for large problems the model generation process itself becomes a bottleneck, especially when the optimization solver is parallelised, but the model generator is not.

Consider a mathematical programming problem in the form

$$\begin{aligned} &\min_{x \in X} f(x) \text{ s.t. } g(x) \leq 0, \\ &\text{where } X \subseteq \mathbb{R}^n,\ f : \mathbb{R}^n \to \mathbb{R},\ g : \mathbb{R}^n \to \mathbb{R}^m \end{aligned} \tag{1}$$

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, values of $f(x), g(x), \nabla f(x), \nabla g(x), \nabla^2 f(x)$, $\nabla^2 g_i(x)$ are required at a given point $x \in X$. It is often the the case that the solver is linked with an AML to whom the computation work of these values are delegated. By using an AML, the modeller can focus on the underlining mathematical relations of the problem rather than the programming work for computing those values. Therefore use of an AML helps to produce an easy-to-understand model in a concise format and improve the maintainability of large optimization models.

For real life problems the number of constraints, m and the number of decision variables, n can become very large: problem sizes in excess of tens or hundreds of millions are not uncommon. Such large scale optimization problems are typically 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. Algorithms, such as Dantzig-Wolfe and Benders decomposition, and interior point solvers, such as OOPS[4] and PIPS[5] can take advantage of such structure to speed up the solution, enable the solution of larger problems and to 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 pass it on to the solver. There has been some research work done to recover the structure information from the constraint matrix[6], however, this is computationally expensive. More importantly we believe this is unnecessary since the problem structure is most likely known to the modeller.

There are structure conveying modelling languages for specific applications such as stochastic programming [7–10]. Alternatively structure information can be provided to the optimization solver by annotation of the unstructured model[11]. These, however, are either not general approaches, or they require assembling the complete unstructured model before annotations can be parsed, which is infeasible for large problems. The Structure-conveying Modelling Language SML[1] was designed to be a generic AML for describing any structured problems by building

models from nested blocks. A similar approach has recently been implemented in Pyomo[12].

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 problems in a massively parallel environment, problem generation can become a significant bottleneck for both memory and execution speed. 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[13].

In this paper we present a parallel model generator for SML, named Parallel Structured Model Generator (PSMG). PSMG can not only convey the problem structure to the solver, but also use it to parallelise the problem generation. PSMG also removes memory limitation of a single node by distributing the problem data. The subsequent sections of this paper are organized as follows: Section 2 reviews the SML syntax and properties of structured problems. Section 3 presents important design considerations in PSMG and explains the interface between PSMG and the parallel solver. Section 4 presents benchmarking results regarding parallel efficiency and memory usage. Finally we present our conclusions in Section 5.

2 Structured problems and review of SML

As an example of a structured problem we consider the Multi-commodity Survivable Network Design (MSND) problem[1]. In this problem the objective is to install additional capacity on the links 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 pattern of the constraint matrix for the MSND problem is shown in Figure 1. The matrix features network constraints (*Net* blocks) that are repeated over commodities and joined by link capacity constraints. These sub-blocks again are repeated for each missing link or node. All is joined together by the column at the right hand side representing additional capacity variables. The problem is highly structured of a from that can be exploited by parallel solvers such as OOPS.

The SML model corresponding to the MSND problem is given in Model 1.1. SML syntax is based on AMPL with additional keywords such as `block`, `stochastic`, etc. The model features a description of the network block in lines 9–11 and 21–23, which is then repeated by the `block`-statements in lines 8/20 and 6/17 over commodities and missing links/nodes respectively. Each block defines a scope which can contain variables, constraints, or further blocks. Variables can be referenced from outside of the scope with a syntax borrowed from object-oriented programming (lines 15 and 26). Note the strong correspondence of the structure of the constraint matrix and the structure implied by the nesting of `block`-statements.

```
set NODES, ARCS, COMM;
param cost{ARCS}, basecap{ARCS}, arc_source{ARCS}, arc_target{ARCS};
param comm_source{COMM}, comm_target{COMM}, comm_demand{COMM};
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;
var sparecap{ARCS}>=0;
block MCNFArcs{a in ARCS}: {
  set ARCSDIFF := ARCS diff {a};
  block Net{k in COMM}: {
    var Flow{ARCSDIFF}>=0;
    subject to FlowBalance{i in NODES}:
      sum{j in ARCSDIFF:arc_target[j]==ord(i)} Flow[j] - sum{j in ARCSDIFF:arc_source[j]==ord(i)} Flow[j] =
            b[k,i];
  }
  var capslack{ARCSDIFF} >= 0;
  subject to Capacity{j in ARCSDIFF}:
    sum{k in COMM} Net[k].Flow[j] = basecap[j] + sparecap[j] + capslack[j];
}
block MCNFNodes{n in NODES}: {
  set NODESDIFF := NODES diff {n};
  set ARCSDIFF := {m in ARCS:arc_source[m]!=ord(n) and arc_target[m]!=ord(n)};
  block Net{k in COMM}: {
    var Flow{ARCS} >= 0;
    subject to FlowBlance{i in NODESDIFF}:
      sum{j in ARCSDIFF:arc_target[j]==ord(i)}Flow[j] - sum{j in ARCSDIFF:arc_source[j]==ord(i)}Flow[j] = b[
            k,i];
  }
  subject to Capacity{j in ARCSDIFF}:
  sum{k in COMM} Net[k].Flow[j] <= basecap[j] + sparecap[j];
}
minimize costToInstall: sum{x in ARCS} sparecap[x]*cost[x];
```

Model 1.1: SML model for MSND problem.

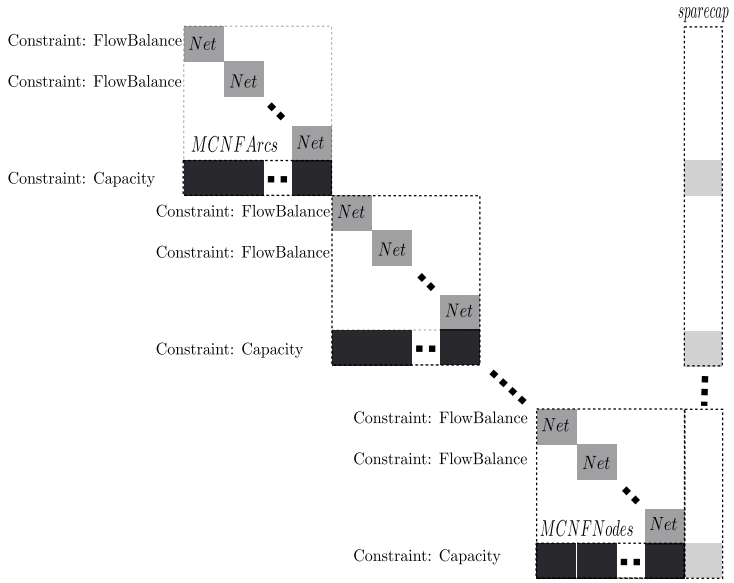

Fig. 1: The block angular structure of the constraint matrix for MSND problem.

3 Model Generator Design Issues

3.1 Solver driven work assignment approach.

PSMG parallelises both the problem generation and the function evaluates routines. 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 we note that only the solver can judge subproblem complexity (ie. computation work involved in the solution process) in order to achieve load balancing.

This leads us to a design 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.

We now describe some important components of our design.

3.2 Prototype model tree

The prototype model tree (Figure 2) is PSMGs internal representation of the nested block dependencies defined in the model file. Every node in the tree corresponds to one block declared in the model. It contains a list of entities declared in this block of the model file in generic form. Each node is also associated with an indexing expression which will be expanded when generating the expanded model tree. For example, in the prototype model tree for MSND problem, the node *Root_MCNFArcs* has an indexing expression of *l in Arcs*.

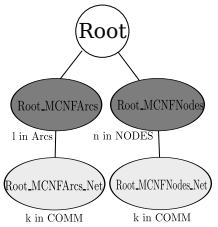

Fig. 2: The prototype model tree for MSND problem specified in Model 1.1.

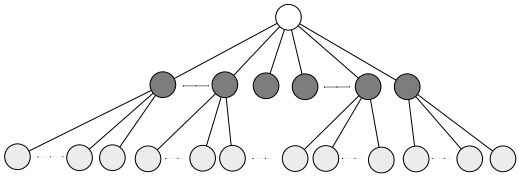

Fig. 3: The expanded model tree for the MSND problem.

3.3 Expanded model tree

The expanded model tree (Figure 2 represents an instance of the problem which is generated after reading the problem data. It is obtained by creating copies of

each node in the prototype tree according to the associated indexing expression. The information stored in each node is restricted to the size of the subproblem (numbers of constraints and variables declared locally) and a pointer to the corresponding prototype tree node. Once the expanded model tree is generated by PSMG, it is passed to solver. The solver will be able to traverse the expanded model tree recursively to retrieve the structure information and set up the problem.

3.4 Model context tree and memory consideration.

The model context tree stores additional information for each node of the expended model tree. This data includes names for the local variables and constraints and values for the locally declared sets and parameters, mapping information for dummy variables etc; namely the context in which to interpret the generic model of the prototype tree. The model context tree is set up at the same time as the expanded model tree but only populated at the request of the solver initiated call-back for function evaluation. In particular the model context tree will only be populated on those processors that have requested the data. This "lazy" approach of data computation guarantees that memory is only used when and where it is necessary. The model context tree also provides hierarchical lookup: for example, in the MSND problem (defined in Model 1.1), all the lookup requests for the set value(s) of the COMM set will fall back to the root level context (where the COMM set is defined).

3.5 Solver interface.

Figure 4 illustrate the overall workflow between PSMG and the solver. After processing of the model and data file on every PSMG processes the prototype tree, expanded model tree and an empty model context tree will be generated on all processes. Every processes will thus have the size and structure information of the entire problem. The time and storage required for these common procedures are very small compared to the function and derivation evaluation routine invoked later. Once this information is set up on every parallel processor, there is no further need for any communication among the PSMG processes. Then the structure information (in form of the prototype and expanded model trees) will be handed over to solver. 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 will be able to request the function and derivative values of each block in parallel from PSMG.

The design of the solver interface allows PSMG to be linked with any structure exploiting parallel solver, not only interior point solvers (such as OOPS[4]), but also decomposition based solvers.

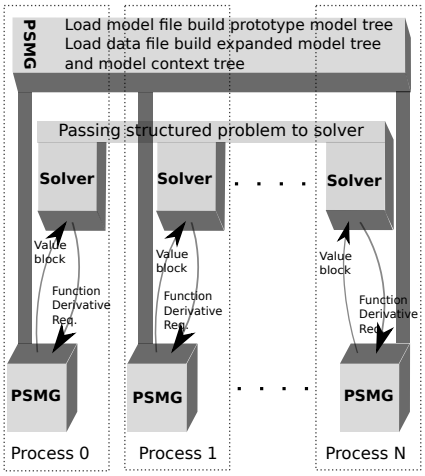

Fig. 4: The PSMG workflow with a parallel optimization solver.

4 Performance evaluation

4.1 Serial Performance

We are aware that our implementation will not match the performance of a commercial model generator such as AMPL on a single node but believe this will be offset by the advantage generated from exploiting parallelisation. In particular, we remove the memory limitation of a single node, and will be able to generate larger problems than could be handled by AMPL on a single node with limited memory.

In the following section we compare the performance of PSMG with the previous serial AMPL based SML implementation from [1] (SML-AMPL), and plain AMPL for the equivalent unstructured model on a series of test problems. The problem generation times are composed of two parts: first parsing the model and setting up the structures and secondly function and derivative evaluations. Note that AMPL does a complete expansion of all indexing expressions (in variable and constraint declarations as well as in `sum` expressions) when parsing the model. PSMG on the other hand, in order to minimize the time until control is passed to the solver, defers these expansions until the first time the automatic differentiation routines are called. As a result in AMPL the majority of time is spend in parsing the model, whereas for PSMG it is in the function and derivative evaluations. This design allows PSMG to distribute its most costly work, namely function and derivative evaluations among the parallel processes in order to speed up the problem generation time. Both AMPL and SML-AMPL use nl-files to communicate the model to the solver. Therefore problem generation times are also dependent on the file-system speed.

We have generated a set of random instances for the MSND model (Model 1.1). The data is based on a network of 20 Nodes and 190 Arcs, corresponding

to a complete graph. The number of commodities varies between 1 and 256. The number of constraints and variables in these problems increase linearly with the number of commodities; the largest problem in the sequence has 10.2 million variables and 1.1 million constraints. The problem generation times are presented in Table 1 and Figures 5, 6 and 7.

We can observe that PSMG and AMPL both achieve linear scaling. However, this is not the case for SML-AMPL. Therefore PSMG significantly improves the performance of the previous SML implementation. The serial performance figures, however, also show that AMPL still has a much better performance in the serial case than PSMG. However the purpose for PSMG is not to beat AMPL but to attain parallelisable model generation for structure exploiting solvers.

Table 1: Problem generation time for MSND problem with increasing problem size on a compete graph of a network of 20 nodes and 190 arcs. Number of variables and constraints in the problem increases linearly with commodities

Number of Commodities	PSMG (s)		AMPL (s)		SML-AMPL (s)	
	Structure Setup	Function and Derivative Evaluation	Structure Setup	Function and Derivative Evaluation	Structure Setup	Function and Derivative Evaluation
1	0.21	5.56	0.8	0.16	1	5
2	0.23	10.15	1.33	0.2	2	9
4	0.23	19.25	2.31	0.31	3	19
8	0.24	37.71	4.11	0.52	5	45
16	0.27	74.27	7.72	0.9	9	127
32	0.31	145.9	15.66	1.65	17	404
64	0.42	285.99	31.62	3.16	35	1405
128	0.58	583.53	64.62	5.9	70	5480
256	1.01	1166.6	140.7	11.89	196	25340

4.2 Parallel Efficiency

On a node with 4GB memory, the plain AMPL will not be able to generate the MSND problem that has 55 commodities on a network of complete graph of 30 vertex and 435 arcs because of not enough memory. This problem (*msnd30_55*) has $11,30,795$ variables and $967,410$ constraints. Now we will be able to generate this problem in parallel by PSMG.

Table 2 and Figure 8 show the parallel benchmarking results for this problem on up to 256 processes. We observe that PSMG obtains excellent speed-up on 16 processes and still an respectable speed-up of 184 on 256 processes, corresponding to a parallel efficiency of 0.72. The main reason preventing even higher speed-up is lack of perfect load balancing.

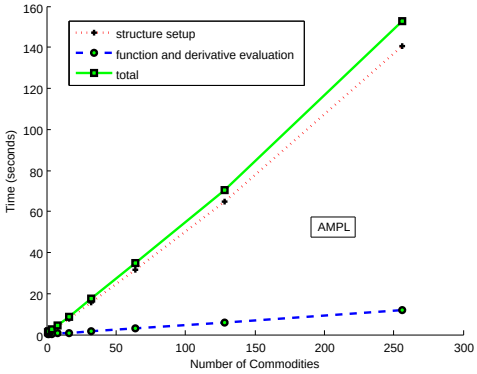

Fig. 5: Problem generation time for AMPL running on a single processor. The data plotted corresonds to Table 1.

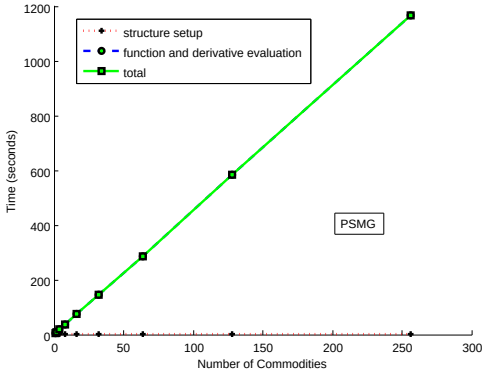

Fig. 6: Problem generation time for PSMG running on a single processor. The data plotted corresonds to Table 1.

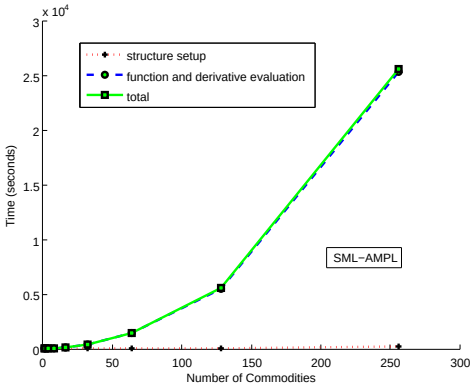

Fig. 7: Problem generation time for SML-AMPL running on a single processor. The data plotted corresponds to Table 1.

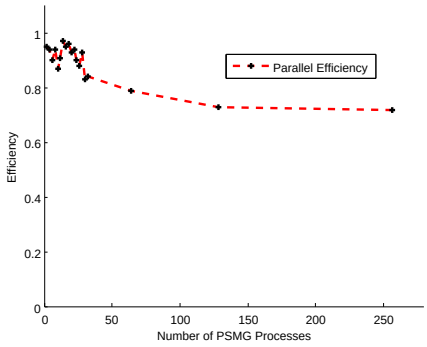

Fig. 8: Parallel efficiency plot for PSMG for problem *msnd30_55*.

Table 2: PSMG speedup and parallel efficiency for problem `msnd30_55`. Note that the Finishing Time column is the maximum time taken for problem generation among the parallel processes.

Number of parallel PSMG processes	Finishing Time(s)	Speedup	Efficiency
1	1911.17	NA	NA
2	1009.79	1.89	0.95
4	509.71	3.75	0.94
8	254.39	7.51	0.94
16	125.78	15.19	0.95
32	71.06	26.9	0.84
64	37.61	50.82	0.79
128	20.53	93.09	0.73
256	10.37	184.3	0.72

4.3 PSMG Memory Usage Analysis

We have also measured per processor and total memory usage of PSMG for generating problem *msnd30_55*. 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 data in the model context tree. Recall that the problem data in the model context 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.

This memory usage data is presented in Table 3. Columns 5 and 6 respectively give the total and per-processor memory requirements. The total memory is broken down in columns 2–4 into memory for the prototype tree, expanded model tree and model context tree. Column 7 gives memory overhead as a percentage of total memory usage. We also plot the total memory usage and the average memory usage in Figure 9 and 10 correspondingly.

The results in Table 3 show that the memory required by PSMG to generate problem *msnd30_55* consists of a non-distributable part used for storing the structure of about 9.59 MB that is repeated on every processor and a remaining part of 4.20 GB that is distributable over processes. Thus we are able to distribute the vast majority of the storage requirements, 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, so we maintain that it is a worthwhile use of memory.

Table 3: Memory usage information for generating a Problem *msnd30_55*

Number of parallel PSMG processes	Total Memory Prototype Tree (MBytes)	Total Memory Expanded Tree (MBytes)	Total Memory Context Tree (GBytes)	Total Memory (GBytes)	Memory per Process (MBytes)	Structure Memory Overhead
1	0.05	9.54	4.20	4.21	4309.87	0.22%
2	0.10	19.07	4.20	4.22	2160.32	0.44%
4	0.20	38.14	4.20	4.24	1085.01	0.88%
8	0.40	76.28	4.21	4.28	548.20	1.75%
16	0.81	152.56	4.21	4.36	279.01	3.44%
32	1.61	305.12	4.22	4.52	144.48	6.63%
64	3.23	610.24	4.23	4.83	77.24	12.41%
128	6.46	1220.49	4.25	5.45	43.62	21.97%
256	12.92	2440.97	4.31	6.70	26.81	35.75%

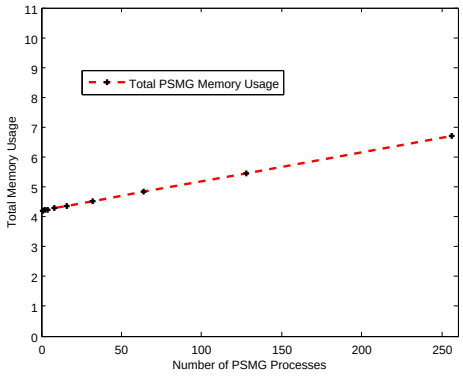

Fig. 9: Total memory usage plot for generating problem *msnd30_55*.

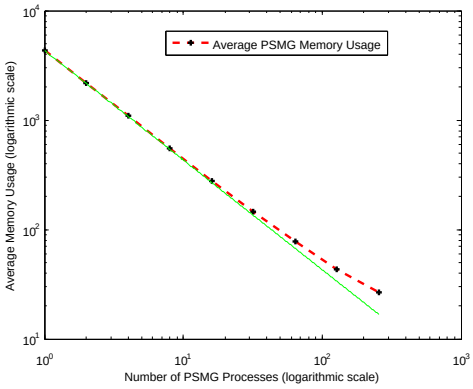

Fig. 10: Per processor memory usage for generating problem *msnd30_55*.

5 Conclusions

In this paper, we have presented a parallel structure-conveying model generator (PSMG) for the structure conveying modelling language SML. Firstly, PSMG retains the advantages of SML, which are: offering an easy-to-use syntax to model optimization problems that are composed of nested sub-problems, and being able to pass the problem structure information to the optimization solver. PSMG is able to use the structure defined in the model to parallelise the problem generation process itself. PSMG also features a novel parallel interface design that enables the parallel solver to achieve load balancing and data locality in order to minimize the amount of data communications among parallel processes. As far as we are aware SML/PSMG is the only parallel algebraic modelling language that can pass the structure information to the solver and uses this information

to parallelise the model generation process. We have presented some key design decisions that have influenced our implementation. We have illustrated that by paying a small memory overhead, PSMG can implement solver driver problem distribution 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 problem which could not be generated on a single node before.

Currently PSMG only supports linear programming. The obvious extension to nonlinear programming would require Hessian evaluation routines and keeping trace of cross products between sub-model components in the constraint functions. We leave this for future work.

References

1. Colombo, M., Grothey, A., Hogg, J., Woodsend, K., Gondzio, J.: A structure-conveying modelling language for mathematical and stochastic programming. Mathematical Programming Computation **1** (2009) 223–247
2. Fourer, R., Gay, D.M., Kernighan, B.W.: AMPL: A Modeling Language for Mathematical Programming. Duxbury Press (2002)
3. Brook, A., Kendrick, D., Meeraus, A.: Gams, a user's guide. SIGNUM Newsl. **23**(3-4) (December 1988) 10–11
4. Gondzio, J., Grothey, A.: Exploiting structure in parallel implementation of interior point methods for optimization. Computational Management Science **6** (2009) 135–160 10.1007/s10287-008-0090-3.
5. Petra, C., Anitescu, M., Lubin, M., Zavala, V., Constantinescu, E.: The solver - PIPS. `http://www.mcs.anl.gov/~petra/pips.html`
6. Ferris, M.C., Horn, J.D.: Partitioning mathematical programs for parallel solution. Mathematical Programming **80** (1994) 35–62
7. Valente, C., Mitra, G., Sadki, M., Fourer, R.: Extending algebraic modelling languages for stochastic programming. INFORMS J. on Computing **21**(1) (January 2009) 107–122
8. Fourer, R., Lopes, L.: StAMP: A filtration-oriented modeling tool for multistage stochastic recourse problems. INFORMS Journal on Computing **21**(2) (2009) 242–256
9. Buchanan, C., McKinnon, K., Skondras, G.: The recursive definition of stochastic linear programming problems within an algebraic modeling language. Annals of Operations Research **104**(1-4) (2001) 15–32
10. Murtagh, B.A.: Advanced Linear Programming : Computation and Practice. New York ; McGraw-Hill International Book Co. (1981)
11. Fragniere, E., Gondzio, J., Sarkissian, R., Vial, J.P.: Structure exploiting tool in algebraic modeling languages. Management Science **46** (2000) 1145–1158
12. Siirola, J.D., Watson, J.P., Woodruff, D.L.: Leveraging block-composable optimization modeling environments for transmission switching and unit commitment, FERC Technical Conference, "Increasing Market and Planning Efficiency through Improved Software", Washington DC (June 25-27, 2012)
13. European Exascale Software Initiative: Final report on roadmap and recommendations development. `http://www.eesi-project.eu` (2011)