Progressive Hedging In Parallel

Michael Somervell
Department of Engineering Science
University of Auckland
New Zealand

Abstract

The Progressive Hedging Algorithm (PHA) is a technique for solving linear discrete stochastic programs. It is of special interest in large-scale problems (which can easily happen with discrete stochastic problems) as it decomposes the problem into smaller problems that can be solved independently. The PHA is implemented in parallel by solving the scenarios in parallel at each stage. Several extensions to this have also been created, most of which are simply heuristics at this stage, as convergency has not been proven. The extensions include asynchronisation within each iteration and not solving all of the scenarios within each iteration. The most significant computational results to date are that the problems still obtain an optimal solution when less than all of the scenarios are solved before updating, and that this leads to a speedup in the solution time.

1 Introduction

Multistage stochastic programs are often too large to be solved via direct solvers. In order to increase the manageability of these problems, they can be decomposed into scenarios as shown in Figure 1. The relationships between the variables at the different time stages and different scenarios are also shown in Figure 1 i.e. the non-anticipativity constraints (the constraints that say that the variables at nodes that coincide at a given time stage must be equal). After decomposition, the non-anticipativity constraints are then able to be relaxed. The augmented Lagrangian relaxation of this is given in (1.1).

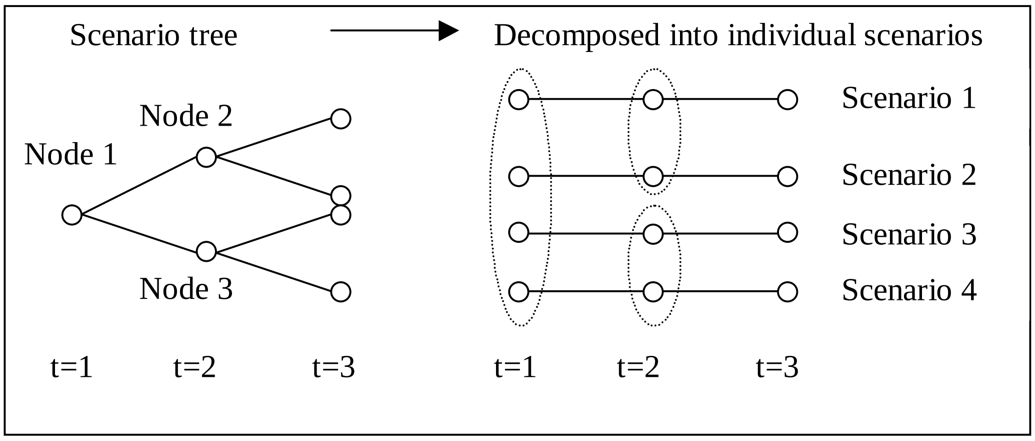

Figure 1

$$\min\ \mathrm{L}\left(\mathbf{x},\mathbf{y},\mathbf{w},\mathbf{r},\overline{\mathbf{x}}^{k-1}\right)=\sum_{s\in S} p_s f\left(\mathbf{x}_s\right)$$
$$+\sum_{t\in T}\sum_{b\in B_t}\sum_{s\in S}\left[w_{t,s}\left(p_s x_{t,s}-p_s \underset{s'\in S}{\mathrm{E}}\left[x_{t,s'}\right]\right)+\frac{1}{2}r_{t,b}\left(p_s x_{t,s}-p_s \underset{s'\in S}{\mathrm{E}}\left[x_{t,s'}\right]\right)^2\right]$$

subject to

$A_i x_i = b_i \qquad i\in S$

where $f\left(\mathbf{x}_s\right)$ is the cost function of the problem before decomposition,

S is the set of scenarios,

p_i is the probability of scenario i occuring, (1.1)

c_i is the cost function associated with scenario i,

x_i is the vector of decision variables $(x_{0,i}, x_{1,i}, \ldots, x_{T,i})^T$,

T is the number of time stages

B_t is the set of scenarios that are indistinguishable up to time t,

w are the Lagrange multipliers of the non-anticipativity constraints,

and r is a penalty term.

The problem with this is that it is non-separable across scenarios (due to the cross product in the squared term), and as such this does not help reduce the problem size. However, the progressive hedging algorithm eliminates this problem by approximating the expected value term by the expected value at the previous iteration of the algorithm. After simplification [1] the objective then becomes the PHA as shown in (1.2) (this condition is present if the Lagrange multipliers are equal to zero in the first iteration as the condition is inherited throughout the algorithm.)

$$\mathrm{L}_{\mathrm{PHA}}\left(\mathbf{x},\mathbf{y},\mathbf{w},\mathbf{r},\overline{\mathbf{x}}^{k-1}\right)=\sum_{s\in S} p_s\left[f\left(\mathbf{x}_s,\mathbf{y}_s\right)+\sum_{t\in T}\left(w_{t,s}x_{t,s}+\frac{1}{2}\rho_{t,b}\left(x_{t,s}-\overline{x}_{t,b}^{k-1}\right)^2\right)\right]$$
$$\text{where } \rho_{t.b}=\sum_{s\in b} p_s r_{t,b} \qquad (1.2)$$

This is a fully separable objective function so the problem can be solved (at each iteration) by solving each of the scenario sub-problems and then combining them to get the solution. After each iteration, the Lagrange multipliers, the average term, and possibly the penalty term are updated and the problem is solved again. This continues until convergence is achieved as shown in Algorithm 1. Convergence of this algorithm has been proven by Rockafellar and Wets [4].

```
(1) $k \leftarrow 0$
(2) $\rho^0_{t,b} > 0$, for all $b \in B_t$ and $t \in T_t$
(3) $w^0_{t,s} \leftarrow 0$, for all $s \in S$ and $t \in T$
(4) $\bar{\mathbf{x}}^0_{t,b} \leftarrow \mathop{\mathrm{E}}_{s' \in b}\left[\mathbf{x}^0_{t,s'}\right]$, for all $b \in B_t$ and $t \in T$, where $\mathbf{x}^0_s$ is the scenario $s$
    component of the solution to the relaxed deterministic equivalent.
(5) Repeat
    a) $k \leftarrow k+1$
    b) Solve min $\mathrm{L}_{\mathrm{PHA}}\left(\mathbf{x}_s, \mathbf{y}_s, \mathbf{w}^{k-1}_s, \mathbf{p}^{k-1}_s, \bar{\mathbf{x}}^{k-1}_s\right)$, subject to $\mathbf{A}_s\mathbf{x}_s = \mathbf{d}_s$, for all
       $s \in S$, where $\rho_s = \rho_b$ and $\bar{\mathbf{x}}_s = \bar{\mathbf{x}}_b$, for all $s \in S$, $b \in B_t$, and $t \in T$
    c) $\bar{\mathbf{x}}^k_{t,b} \leftarrow \mathop{\mathrm{E}}_{s' \in b}\left[\mathbf{x}^k_{t,s'}\right]$, for all $b \in B_t$ and $t \in T$
    d) $w^k_{t,s} = w^{k-1}_{t,s} + \rho_{\mathrm{t,b}}\left(x^k_{t,s} - \bar{x}^k_{t,s}\right)$, for all $s \in S$, $b \in B_t$, and $t \in T$
(6) Until convergence, i.e. $\delta^k_b \left( = \left[ \left| \bar{x}^k_{t,s} - \bar{x}^{k-1}_{t,s} \right|^2 + \mathop{\mathrm{E}}_{s \in b}\left[ \left( x^k_{t,s} - \bar{x}^k_{t,s} \right)^2 \right] \right]^{\frac{1}{2}} \right) \leq$ some
    tolerence (say $10^{-4}$)
```

Algorithm 1

The rest of this paper is organised as follows: In Section 2, extensions of the PHA algorithm to parallel via the Message Passing Interface (MPI) libraries (see [3]) and creating modifications to it which are designed to speed the algorithm's convergence, along with the theoretically best combination of methods, the combination of the asynchronisation method with the unconverged scenarios method and ensuring that at least half of the available scenarios are processed per iteration. Several possibilities for extensions are discussed in Section 3. Finally, in Section 4 some initial conclusions for each of the algorithms mentioned in Section 2 will be presented.

2 Different Algorithms

2.1 Simple extensions to parallel

2.1.1 Synchronous method

The synchronous parallel version of the PHA extends the PHA to a multi-processor environment. Instead of solving each of the scenario sub-problems one after another on a single processor, they are given to a number of sub-processors to solve.

The way that this is implemented with MPI is to have a master processor which assigns the next scenarios to be calculated to the sub-processors, collects back the solutions from the sub-processors, allocates the next lot of scenarios, and so on until there are no more scenarios to be solved for that particular iteration. The master processor (which is now in possession of all of the solution vectors for that iteration) then proceeds to update the variables and calculate convergence, and, if not converged, starts the process of passing the scenarios out to the sub-processors again. This is shown in Figure 2

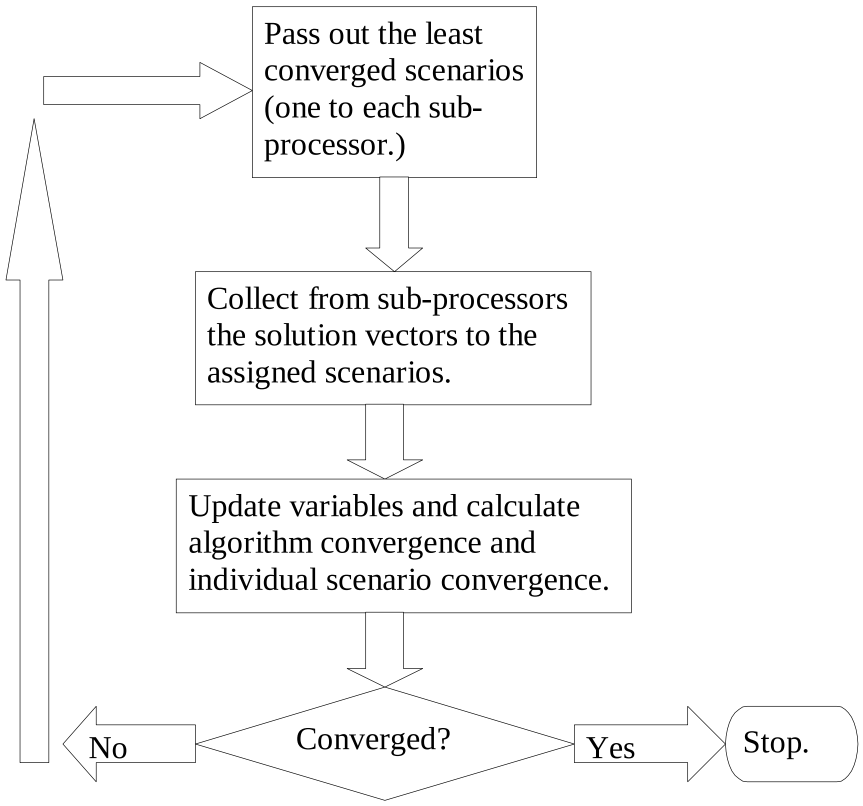
Pass out the least converged scenarios (one to each sub-processor.)
Collect from sub-processors the solution vectors to the assigned scenarios.
Update variables and calculate algorithm convergence and individual scenario convergence.
Converged?
No
Yes
Stop.

Figure 2

2.1.2 Asynchronous method

The asynchronous version of the parallel PHA is similar to the synchronous version except that within each iteration, instead of sending out a number of scenarios and waiting for them all to return, it is constantly sending and receiving solutions as soon as each sub-processor is finished with them. This can allow for significant gains in performance, as it is no longer necessary to have a number of sub-processes that is a factor of the number of scenarios to eliminate idle time spent by sub-processes. For example, if there are nine scenarios and three sub-processes, then there would be no needless idle time as each sub-process would have to calculate the same number of scenario solutions (three each.) However, if there were four sub-processes, three of them would only have to calculate two scenarios, and one sub-processor would have to calculate three. Since the synchronous version must wait for all scenarios to be returned before assigning new scenarios, this will mean an equal amount of CPU time as with the three sub-process case (assuming all the sub-processes have equal performance and that each of the scenarios can be solved in similar amounts of time.) The asynchronous version attempts to mitigate this drawback, as it waits until a scenario has a solution, receives the solution and, if there are scenarios left to calculate in that iteration, assigns the next available scenario to the sub-processor that has just given it the solution. However, there will usually be some amount of time wasted due to waiting in any case.

The synchronous version will give a total time per iteration as being the sum of the maximum times per allocation of scenarios to sub-processors, whereas the asynchronous version will have as its total time the maximum sub-processor time within each iteration. See Figure 3 for an explanatory diagram.

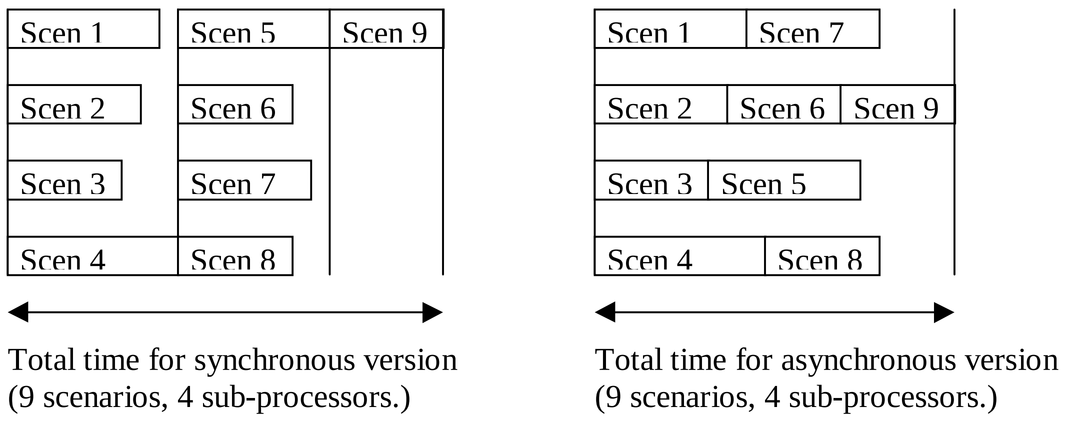
Scen 1
Scen 5
Scen 9
Scen 2
Scen 6
Scen 3
Scen 7
Scen 4
Scen 8
Total time for synchronous version
(9 scenarios, 4 sub-processors.)
Scen 1
Scen 7
Scen 2
Scen 6
Scen 9
Scen 3
Scen 5
Scen 4
Scen 8
Total time for asynchronous version
(9 scenarios, 4 sub-processors.)

Figure 3

2.2 Block-Cyclic Methods

2.2.1 Naïve block cyclic

The number of scenarios that are processed per iteration in the naïve block cyclic method are specified by the user beforehand. For example, if there are nine scenarios and seven are to be processed per iteration, the first iteration 1,2,3,4,5,6 and 7 would be processed and the relevant updates performed, then 8,9,1,2,3,4, and 5 would be processed etc. This method can be used to process larger numbers of scenarios than available processes by processing them in batches exactly the same way that the synchronous and asynchronous parallel versions work.

2.2.2 Least converged scenarios block cyclic

This evolves the idea of the naïve block-cyclic update by attempting to make an intelligent choice for the scenarios to update. After the first iteration, a convergence measure is calculated for each scenario based on the convergence measure used to terminate the algorithm. In the next iteration, the scenarios with the highest of these measures are allocated to the sub-processes until there are no more available sub-processes. The solution is then obtained and an update and convergence check is performed, along with a calculation of the convergence measure for each scenario. If not converged, then the scenarios with the highest convergence measures are chosen and solved, and so on. This is illustrated in Figure 4

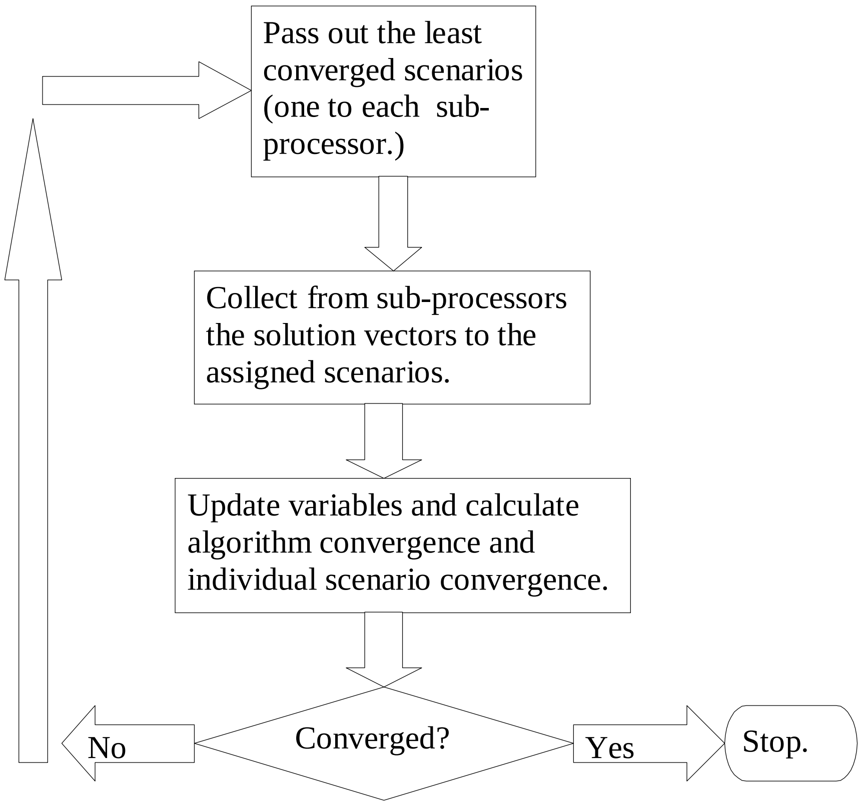
Pass out the least converged scenarios (one to each sub-processor.)
Collect from sub-processors the solution vectors to the assigned scenarios.
Update variables and calculate algorithm convergence and individual scenario convergence.
Converged?
No
Yes
Stop.

Figure 4

2.3 Least converged methods

2.3.1 Least converged node

This method is an alteration of the PHA, as it potentially solves fewer scenarios per iteration than the PHA. The least converged node method only solves the scenarios that are in the node that has converged the least (by the convergence measure for that node from the previous iteration.) The update steps are only done on those scenarios that have been processed in that particular iteration.

This is very similar to the least converged scenario algorithm, except that instead of calculating convergence measures for each of the individual scenarios, a convergence measure is calculated for each node, and the node with the highest convergence measure has all the scenarios incident upon it solved in the next iteration. As an example for this case, take a simple two stage two branch tree (four scenarios in all, arranged as shown in Figure 5.

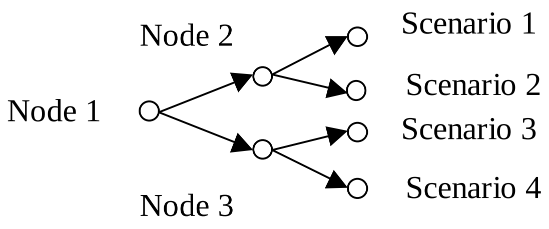

Figure 5

All four scenarios would be solved for the first iteration, it is then determined that node 2 had the greatest convergence measure, so on iteration two, scenarios 1 and 2 are processed, and all the information is updated. For the next iteration, node 1 has the highest convergence measure, so all four scenarios have to be solved for the next iteration and so on until convergence.

2.3.2 Least converged scenario

This alteration to the PHA is similar to the least converged node, except that only one scenario is processed per iteration.

2.4 Unconverged methods

2.4.1 Unconverged scenarios

This method is an attempt to get the problem to govern how many scenarios should be processed per iteration, similar to the least converged methods, as the number dynamically changes depending on how many of the scenarios are “converged”, as opposed to the block cyclic methods in which a set number of scenarios get processed per iteration. This is achieved by only processing those scenarios which have a convergence measure greater than the convergence tolerance divided by the number of scenarios. If there are none, then the problem must have converged.

2.4.2 Unconverged nodes

This method is similar to the unconverged scenarios method in that it attempts to dynamically determine the number of scenarios to solve per iteration, but it uses the convergence values at the scenario tree nodes and processes the scenarios incident on the unconverged nodes in the following iteration.

2.5 Fully asynchronous method

The fully asynchronous method attempts to take advantage of the parallel computer by performing the update and convergence test immediately upon the return of a scenario. For example, 2 scenarios out of 5 are sent to processes 1 and 2 to be solved. Scenario 2 returns first, and the Lagrange multipliers and average values are immediately calculated along with the convergence test, which shows that the problem is not converged. Scenario 3 is now sent to the process that scenario 2 had come from. Next, scenario 1 returns, the updates and convergence test are calculated, and scenario 4 is sent to the process that scenario 1 came from, etc.

2.6 Other possible methods

Many combinations of the above methods are possible, such as combining the asynchronisation method with the unconverged scenarios method and ensuring that at least half of the available scenarios are processed per iteration. This combination should converge (due to processing over half of the scenarios per iteration), should take less time to converge (as it processes the least converged scenarios), and should take less time in the communication phase (due to the inclusion of the asynchronisation step) and is the recommended method at this stage.

3 Other avenues of investigation

There are many other areas that are not well understood in PHA that would benefit from further investigation. Amongst these are parameter investigation, such as dynamically varying penalty parameters as in Mulvey and Vladimirou [2].

A variant on this is increasing the penalty parameter for individual nodes/scenarios if the primal convergence (the difference of the $\bar{\mathbf{x}}$'s in the convergence check in (1.1)) is not below a tolerance value and decreasing it if the primal check is satisfied but the dual check (the difference of the $\mathbf{x}$'s in (1.1)) is not. This approach is recommended as a large penalty parameter has been shown to increase speed of primal convergence, and a low penalty parameter has been shown to increase speed of dual convergence (Broad [1]).

Another area that could be looked into is the convergence check itself, such as using the lower bound convergence check put forward by Broad [1]. This would be a better convergence measure, as it measures the distance between the upper and lower bounds of the objective value of the problem being solved. Termination of the algorithm could then occur with confidence that one was within a certain distance of the true optimal value.

4 Results/Conclusions to date

4.1 Timing difficulties

Ideally, to time each of these algorithms, one would use a single-user machine, i.e. one that is fully devoted to the problem at hand. However this is not the case with the machine that these algorithms are to be run on as it is a multi-process algorithm on a multi-user, multi-processor computer. There are many problems with this configuration from a timing point of view. First and foremost of these is the problem that the processes do not necessarily all get processed at the same time. This potentially increases the communication time. Secondly assigning the processes to processors is never done in exactly the same way twice, so timing will have to take account of this somehow. For a more detailed explanation of these issues, see Somervell [5].

4.2 Conclusions so far

The serial version (Algorithm 1) does indeed tend to converge with every problem that was attempted (although the convergence can be very slow for “bad” parameter values.)

The synchronous method (Section 2.1.1) does indeed speed up the algorithm compared to the serial version, but the extent of this speedup is still to be shown. The solutions for this method are exactly the same as for the serial method.

The asynchronous method (Section 2.1.2) creates a definite speedup over the synchronous method which cannot as of yet be quantified. The solutions for this method are also the same as for the serial method.

In the naïve block cyclic method (Section 2.2.1) it appears that if half or more of the available scenarios are processed per iteration the algorithm always converges (usually faster than compared to processing all of the scenarios per iteration) whereas processing fewer than half of the scenarios per iteration does not tend to converge. This would account for the least converged scenario method not converging as it only processes one scenario per iteration, the same is true of the least converged node methods as they would usually be processing less than half of the scenarios per iteration. This would account for it occasionally converging on the rare occasions that it does.

The least converged scenarios block cyclic method (Section 2.2.2) seems to be even better than the naïve bock cyclic as it often converges faster, and also it has a better chance of converging when fewer than half of the scenarios are processed per iteration. However, the best speed is generally achieved when around half of the scenarios are processed.

The least converged node method (Section 2.3.1) does not necessarily converge on problems that are reasonably large in size, but does tend to work on small problems. This is probably due to the phenomenon observed in the naïve block cyclic method.

The least converged scenario method (Section 2.3.2) only converges on trivially small problems, and this is probably due to the observations made in the naïve block cyclic case.

The unconverged scenarios method (Section 2.4.1) works extremely well in some cases (a speedup of around three times in some problems) but slower in others.

The unconverged node method (Section 2.4.2) is slightly faster than the unconverged scenarios method, yet shares the same drawbacks.

The fully asynchronous method (Section 2.5) fails to converge even on very small problems.

References

[1] Kevin Broad. *Power Generation Planning Using Scenario Aggregation*. Masters thesis, Dept. of Engineering Science, University of Auckland, (1996)

[2] J. M. Mulvey and H. Vladimirou. *Solving Multistage Stochastic Networks: An Application of Scenario Aggregation.* Networks 21 (1991) 619-643.

[3] P. Pancheo. *Parallel programming with MPI.* Morgan Kaufmann Publishers, San Francisco (1997).

[4] R.T. Rockafellar and R.J.-B Wets. *Scenarios and policy aggregation in optimization under uncertainty*. Math. Operations Res. 16 (1991) 1-29.

[5] M. B. J. Somervell. *Progressive Hedging in Parallel*. Masters thesis, Dept. of Engineering Science, University of Auckland, (1998)