OPTIMAL AGGREGATION OF BLOCKS INTO SUBPROBLEMS IN LINEAR-PROGRAMS WITH BLOCK-DIAGONAL-STRUCTURE

A THESIS SUBMITTED TO THE GRADUATE DIVISION OF THE UNIVERSITY OF HAWAIʻI AT MĀNOA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

ELECTRICAL ENGINEERING

AUGUST 2015

By

Deepak Ponvel Chermakani

Thesis Committee:

Matthias Fripp, Chairperson

Henri Casanova

Rui Zhang

ACKNOWLEDGMENTS

I thank my advisor Dr. Matthias Fripp, for agreeing to supervise me on this project, his able guidance, the knowledge I learnt on Power Grid problems and modelling them as Linear Programs via courses under him, and also for his continued faith in me by funding me as a Research Assistant. I thank Dr. Henri Casanova for the advanced concepts I learnt in High Performance Computing via a course under him, and also for having earlier agreed to supervise me on a mini-project in that course, on a topic related to this thesis. I thank Dr. Rui Zhang and Dr. Gurdal Arslan for the advanced concepts I imbibed in Algorithms and Dynamic Programming respectively via courses under them. I also thank Josiah Johnston for helping me understand many concepts in the Pyomo programming language.

ABSTRACT

In many large practical planning problems modelled as linear-programs with block-diagonal-structure, it is desirable to minimize wall-clock-time for a solution to the entire set of diagonal blocks. When the number of available independent-processing-units is at least equal to the number of blocks, this wall-clock-time is minimized by completely decomposing the linear-program into as many small-sized subproblems as possible, each block resulting in a separate subproblem. This decomposition strategy does not necessarily work when the parallel processing capability is limited, causing multiple subproblems to be serially solved on the same processing-unit. In such a situation, it might be better to aggregate blocks into larger sized subproblems. The optimal aggregation strategy depends on the computing-platform used. We show that optimal aggregation is NP-hard, when blocks are of unequal size. We also show that when blocks are of equal size, optimal aggregation can be achieved within polynomial-time. Experiments with solvers show substantial reduction of solution-times by using our optimal aggregation strategy, compared with trivial aggregation strategies. This is an important result for linear-programs that have diagonal blocks of the same size, since our approach can be used to minimize the expected solution-time of the set of subproblems in every iteration of any decomposition technique.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS AND NOTATIONS

The following abbreviations are used for this thesis:

IPU	Independent Processing Unit (refers to a “core” since we consider a single machine)
LP	Linear Program
ILP	Integer Linear Program
SPM	Simplex Method
IPM	Interior Point Method
BDS	Block Diagonal Structure
OS	Operating System
HPC	High Performance Computing
DP	Dynamic Programming

We use the following notations for this thesis:

1) By “an LP of size n”, we may consider it to be an LP where the total number of non-negative variables (including slack variables) is n, after conversion of all $\leq$ constraints to equality-constraints.
2) By “optimally aggregating blocks”, we mean that we are trying to do so on a single IPU (i.e. a single core). This follows the assumption that all the IPUs are similar to each other in computing ability having their own separate cache (though RAM may be shared among IPUs), and the assumption that we can efficiently distribute the blocks among the IPUs, such that each IPU is roughly equally loaded.
3) By “computing-platform”, we mean all hardware and software, on which the LP with BDS runs.
4) By an “LP with BDS”, we mean an LP with only diagonal blocks without any common constraints or common variables. This assumes that any common constraints have been relaxed, or any common variables have been fixed, making the blocks separable.
5) *P(x)* is defined in Chapter 2. But for the purpose of our experiments, the definition of *P(si)* is relaxed and is mentioned in section 3.6 of Chapter 3.
6) For an LP with BDS having equal sized blocks, an aggregation strategy depicted by $\{y_1, y_2, ...y_i, ..., y_k\}$ denotes there should be k subproblems, with the i^{th} subproblem having y_i blocks.

CHAPTER 1

INTRODUCTION

1.1 Background

Many planning problems scale to a very large size, depending on the number of scenarios considered. It is often desirable to include as many scenarios as possible, depending on the limits of available processing power. So we would like to solve these as quickly as possible. These problems are usually structured as large linear-programs (LPs) with Block-Diagonal-Structure (BDS) that arises from modelling the constraints of multiple scenarios. The variables for all scenarios are also linked by common constraints. For example, a planning model may have decision variables representing capacity expansion of some resources and other variables representing daily operations of this capacity. Each day is treated as a separate scenario, because operations on each day are independent, creating a number of separate blocks that form a BDS in the LP constraint matrix. However, operations on each day are also constrained by the expansion decisions, creating constraints common to all blocks in the LP constraint matrix. A concrete example of a simple LP with BDS having three blocks and two common constraints is as follows.

Maximize:

$$c_1 x_1 + c_2 x_2 + c_3 x_3 + c_4 x_4 + c_5 x_5 + c_6 x_6$$

Subject to:

$$\begin{array}{ll}
a_{11} x_1 + a_{12} x_2 + a_{13} x_3 + a_{14} x_4 + a_{15} x_5 + a_{16} x_6 & \leq d_1 \\
a_{21} x_1 + a_{22} x_2 + a_{23} x_3 + a_{24} x_4 + a_{25} x_5 + a_{26} x_6 & \leq d_2 \\
b_{11} x_1 + b_{12} x_2 & \leq e_1 \\
b_{21} x_1 + b_{22} x_2 & \leq e_2 \\
\qquad\qquad b_{33} x_3 + b_{34} x_4 & \leq e_3 \\
\qquad\qquad b_{43} x_3 + b_{44} x_4 & \leq e_4 \\
\qquad\qquad\qquad\qquad b_{55} x_5 + b_{56} x_6 & \leq e_5 \\
\qquad\qquad\qquad\qquad b_{65} x_5 + b_{66} x_6 & \leq e_6
\end{array}$$

Here, each of $\{x_1 , x_2 , x_3 , x_4 , x_5 , x_6 \}$ is a non-negative real variable, and each of the coefficients $\{c_i, e_i , d_j , a_{j\,i}, b_{i\,k}$ $\}$ are given integer coefficients for all integers i in [1,6], for all integers j in [1, 2], for all integers k in [1, 6].

BDS in LPs allows decomposition (by either relaxing the common constraints, or fixing the common variables) and hence allows parallelism, i.e. parallel computation. Every decomposition technique is an iterative procedure that runs until a convergence criterion is reached, where each iteration consists of the parallel solution to the set of subproblems, followed by the solution to the master-program. Here, the master problem is usually formed by the constraints or variables common to all blocks, while the blocks usually form the subproblems. In each iteration, the subproblems retain their original LP structure with altered coefficients, but the master-program might get a new LP structure (depending on the decomposition technique used, by the addition of new constraints or variables) along with altered coefficients. The underlying equivalence of many decomposition techniques for BDS has been proved [1], for example between Dantzig-Wolfe decomposition on an LP, Benders decomposition on the LP's dual, and a cutting-plane approach to solve its Lagrangian dual. Figure 1 illustrates the iterative nature of decomposition techniques for an LP with BDS with common constraints, where after relaxation, each block usually becomes a subproblem (SP).

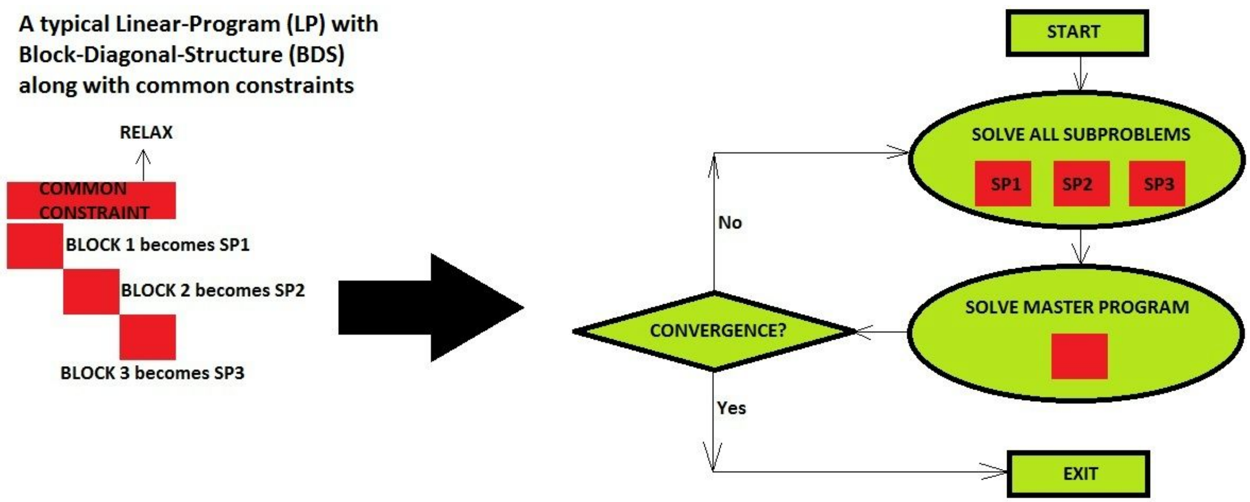

Figure 1. Iterative nature of decomposition techniques for LP with BDS

Parallelism is coarse-grained [7] while executing subproblems, meaning that in this context, a subproblem executes continuously from start to finish on the independent-processing-unit (IPU) to which it is assigned. If one has a large number of IPUs, at least equal to the number of subproblems, then it makes sense to have as many small-sized subproblems as possible, and allow simultaneous execution of each subproblem on a separate IPU, which has been done in [2][3]. But issues [4][5][6][7] have been reported with decomposition, while attempting to solve too many subproblems in parallel, when the number of IPUs is limited (example, on a four-core machine). Some issues are slow convergence, memory-overhead, and communication-overhead.

1.2 Related work

One branch of research [5][6][8][9] into decomposition techniques for LP, focuses on heuristics for row-permutations and column-permutations on an initial sparse constraint matrix, to yield a matrix with specified number of diagonal blocks. The basic approach of these permutation strategies is to represent the non-zero structure of the constraint matrix as a bipartite graph, and reducing the permutation problem to graph-partitioning by vertex separation. These permutation problems are NP-Hard [6] in general, which is why heuristics have been explored for such permutation. However, these permutation-based approaches have not mentioned what would be the number of blocks or the size of such blocks, optimal for the given computing-platform.

A second branch of research [10][11][12][13] has been into the theory of the master program, for example generating more effective cutting-planes in Benders decomposition, or improved column-generation for Dantzig-Wolfe decomposition, or better sub-gradient optimization techniques for Lagrangian relaxation. While this research helps in lowering the number of iterations of decomposition techniques, on any computing-platform, it does not address the question of how to reduce the solution-time of the subproblems in every iteration of any decomposition technique.

A third branch of research [14] focuses on scheduling n subproblems (of unequal size) in parallel on m IPUs ($m<n$), to minimize overall solution time. This problem is equivalent to the the well-known NP-hard problem of job-shop-scheduling. Due to its hardness, many heuristics have been proposed in this area. While this research does attempt to reduce solution-time of subproblems in every iteration of any decomposition technique, it will not perform well in a situation where the number of IPUs is limited, and where the solution-time for an LP on the given computing-platform has a high fixed-component that is independent of LP-size.

A fourth branch of research is into aggregating blocks and treating each aggregate as a separate subproblem. This makes sense when two or more subproblems have been scheduled to execute serially on the same IPU. The idea of aggregation has been done earlier [15] as a part of a heuristic (called the "cascading process") while solving LPs with a staircase structure (similar to BDS, except that adjacent blocks share common variables). Aggregating subproblems in a LP with BDS has been done in real world applications [2][7]. The research in [2][3] concluded that the greater the number of small-sized subproblems in this BDS, the better is the performance, when the number of available IPUs is large. This is however, not the case when the number of IPUs is limited, as agreed by the authors of [5][6][7]. The authors of [5] also gave the idea of performing row and column permutations on a sparse matrix to get a specified number of blocks, but did not specify a strategy to get the optimum number of blocks catered for a given computing-platform.

A fifth branch of research is into clustering (or grouping) of scenarios in LPs [23][24][25], based on the LP coefficients, such that each cluster (or group) of scenarios is treated as a separate subproblem. These authors used heuristics such as k-means-clustering to create a partition or covering of the set of scenarios, so as to hopefully minimize both the running time of the clustered subproblems and the number of iterations of the decomposition technique. It should be noted however that these heuristics are not likely to succeed on all computing platforms. As we will see later, whether or not two or more scenarios (or blocks of the initial LP constraint matrix) should be grouped under the same subproblem, depends on the nature of the LP and nature of the computing-platform used.

It is worth mentioning a sixth branch of research not in the field of LP, but in the specialized fields of linear-algebra and video-processing. This research [27][28] proposes dynamic-programming strategies to obtain the subset of data that should be cached for optimal processing. However, to the best of our knowledge, related ideas have not been applied to the field of LP with BDS.

1.3 Our proposed research

Before mentioning the focus of our research, we illustrate the execution of the master-program and three subproblems (SPs) on a 3-core machine, 2-core machine, and 1-core machine using Figures 2, 3, and 4 respectively (red means the core is busy, while yellow means the core is idle). The focus of our research is on reducing the expected solution-time of subproblems of an LP with BDS, per iteration of any decomposition technique, when the number of IPUs (i.e. cores) is limited to one, using the reasonable assumption that the subproblems may be efficiently distributed among the cores so that each core is roughly equally loaded. We have not found any strategy in literature for LPs with BDS, catered to reduce the solution-time of subproblems on a given computing-platform, giving consideration to the fixed-component and size-dependent-component in the solution-time of an LP on that computing-platform. To the best of our knowledge, our research is the first to focus on this issue. We will first describe aggregation of blocks into subproblems as an approach to deal with this issue. We will then explore the complexity of aggregation. For those cases of aggregation that are difficult, we will try to prove hardness with well-established complexity classes. For those cases of aggregation that allow an easy solution, we will develop an efficient approach to aggregate blocks into subproblems, so that the sum of the solution times of the subproblems is minimized on the computing-platform, in every iteration of any decomposition technique. We will test our aggregation approach using a well-known LP with BDS. We do acknowledge that our approach does not guarantee the decrease of the number of iterations of the decomposition technique, and therefore that our approach could be used in conjunction with strategies that decrease the number of iterations.

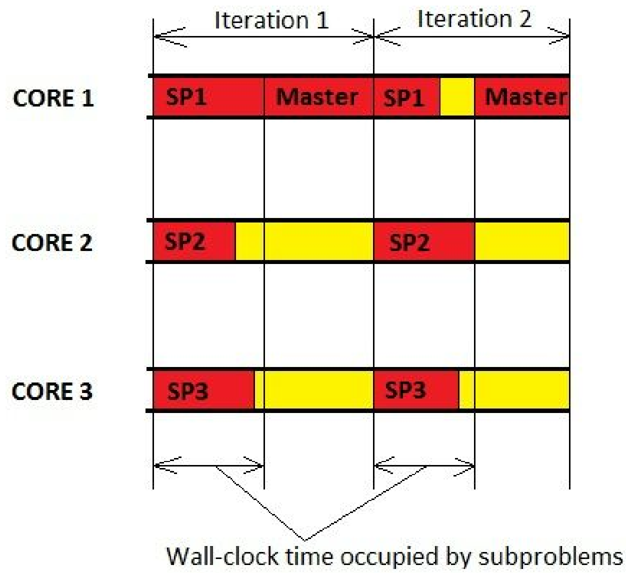

Figure 2. Illustration of wall-clock-time spent in solving 3 subproblems on a 3-core machine

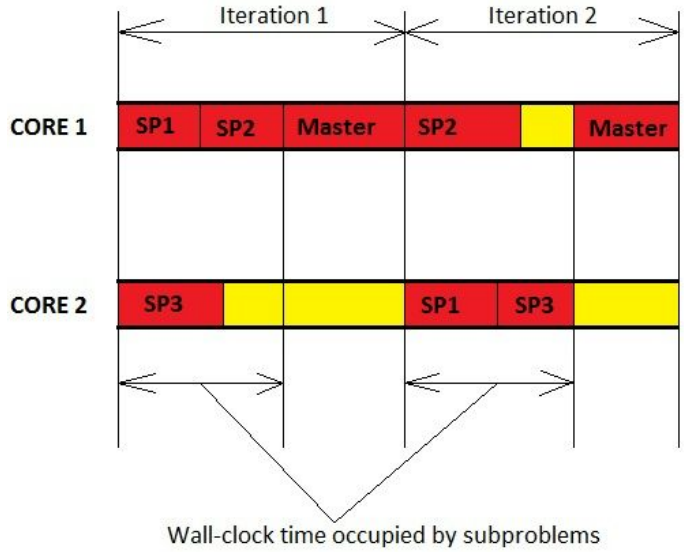

Figure 3. Illustration of wall-clock-time spent in solving 3 subproblems on a 2-core machine

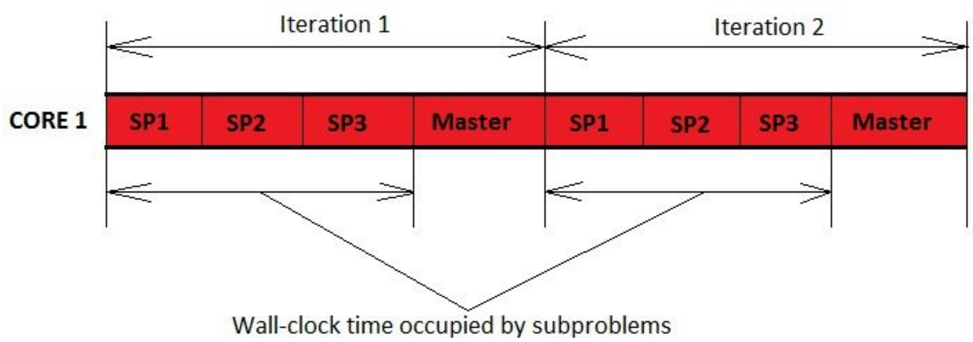

Figure 4. Illustration of wall-clock-time spent in solving 3 subproblems on a 1-core machine

1.4 Organization of this thesis

This thesis is organized as follows. In Chapter 2, we introduce the concept of aggregating blocks into subproblems, explore the complexity of optimal aggregation, and develop approaches for optimal aggregation in LPs with BDS. Chapter 3 describes our experiments, the data we used, the computing-platforms including the LP solver softwares used, how we forced a single core to be used on the multi-core platform, and how we eliminated noise. Chapter 3 first describes experiments called sanity-tests with LP solvers that show that there exist LP instances with BDS, for which none of the two trivial boundary cases of aggregation (i.e. full-aggregation or no-aggregation) is optimal, thereby showing that the theory developed in Chapter 2 will be useful. Chapter 3 then describes experiments implementing the optimal aggregation strategy of Chapter 2 and shows the benefit of the optimal strategy over trivial strategies. Chapter 4 discusses possible reasons why trivial aggregation strategies need not always be optimal, though the exact reasons are beyond the scope of this thesis. Chapter 5 discusses important limitations of our approach, and future work suggested to obviate these limitations. Chapter 6 concludes our thesis.

CHAPTER 2

THEORY OF OPTIMALLY AGGREGATING BLOCKS INTO SUBPROBLEMS

In a LP with BDS, aggregation is the grouping of diagonal blocks into one or more groups, so that each group (or aggregate) is treated as a separate subproblem. The purpose of aggregation is to minimize the solution-time of the set of subproblems. In this full thesis, we aim to minimize solution-time only on one IPU, using the reasonable assumption that the blocks have been distributed among IPUs, giving roughly equal load to each IPU. The assumption is reasonable since there exist efficient greedy-heuristics to distribute n jobs (subproblems in this case) among m machines (cores in this case), where $n > m$, such that the wall-clock-time for processing is less than two times the optimal distribution strategy. Figure 5 illustrates the aggregation of four blocks into two subproblems, in an LP with BDS with complicating constraints.

We define $P(x)$ to be a polynomial-function specific for a computing-platform, such that for any LP of size i, $P(i)$ is the "solution-time" on that computing-platform. If all blocks in the LP with BDS have the same size s, $P(si)$ can also be defined as the "solution-time" of a subproblem with i blocks. In this chapter, by "solution-time", we do not mean exact solution-time (which depends on LP coefficients), but we mean either expected solution-time or worst-case solution-time (whichever is desired to be optimized), over all possible LP coefficients within desired ranges. Depending on the situation, these desired ranges of the LP coefficients can mean one of the following:

1) all integers whose magnitudes are below the largest positive integer that can be represented on the computing-platform. This range may be used, if nothing is known about the type of LP that the computing-platform will frequently deal with.
2) all integers within ranges of the LP that the computing-platform frequently deals with. For example, if the coefficient of the second column and seventh row is known to only vary within [15, 100], then $P(x)$ may be calculated accordingly.

$P(x)$ can be treated as a polynomial-function, since the memory of any computing-platform is finite, and it is well-known that a set of d points can be connected by a polynomial of degree d. For chapter 2, we assume that we already have $P(x)$. Later in chapter 3, we relax the definition of $P(x)$, making it easier to obtain in practice.

We illustrate the motivation for aggregating blocks into subproblems with an example. Consider two blocks of sizes x_1 and x_2, scheduled to run serially on a IPU. Total solution time of subproblems, by not aggregating, is equal to $(P(x_1)+P(x_2))$. Total solution time of subproblems, by aggregating, is equal to $P(x_1 + x_2)$. Hence, whether one should aggregate or not aggregate, depends on whether $P(x_1 + x_2)$ is lesser or greater than $(P(x_1)+P(x_2))$. It is easy to see that $P(x_1 + x_2)$ would be lesser than $(P(x_1)+P(x_2))$ if $P(x)$ has a very large constant term (or fixed component) and if x_1 and x_2 are both small. We now give a precise definition to our problem of optimal aggregation of blocks into subproblems.

2.1 Rigorous definition for the problem of optimal aggregation

Given k diagonal blocks of sizes $\{x_1, x_2, \ldots, x_k\}$ and denote $n = x_1 + x_2 + \ldots + x_k$. The aim of aggregation is to obtain k subproblems of sizes $\{y_1, y_2, \ldots, y_k\}$, where every block is assigned to exactly one subproblem, so that the sum of expected solution-times of the subproblems is minimized. Note that some subproblems might be of zero-size. Let b_{ij} be a binary variable that is 1, if and only if, block i is assigned to subproblem j. Let y_j be a non-negative real variable depicting the size of subproblem j. Let z_j be a binary variable that is 1, if and only if, y_j is positive. We can now define our aggregation problem as the following non-linear-integer-program:

Minimize: $\sum_{j=1}^{k} (P(y_j) - P(0)(1-z_j))$

Subject to: $\sum_{j=1}^{k} b_{ij} = 1$, for each integer i in $[1,k]$

$y_j = \sum_{i=1}^{k} (x_i b_{ij})$, for each integer j in $[1,k]$

$z_j \leq y_j \leq n z_j$, for each integer j in $[1,k]$

$b_{ij} \in \{0,1\}$ and $z_j \in \{0,1\}$ are binary variables, and y_j is a real variable for all integers i and j in $[1,k]$

The first constraint above ensures that every block is assigned to exactly one subproblem. The second constraint assigns the size of each subproblem to the sum of sizes of blocks assigned to that subproblem. The third constraint ensures that the size y_j of each subproblem is positive, if and only if, the binary flag z_j is 1 (clearly, y_j is forced to be equal to 0 if z_j is 0, and y_j is forced to be in $[1,n]$ if z_j is 1). The objective now sums the the solution-times due to each subproblem (clearly, the solution-time out of subproblem j is $P(y_j)$ if $y_j > 0$, and is 0 if $y_j = 0$).

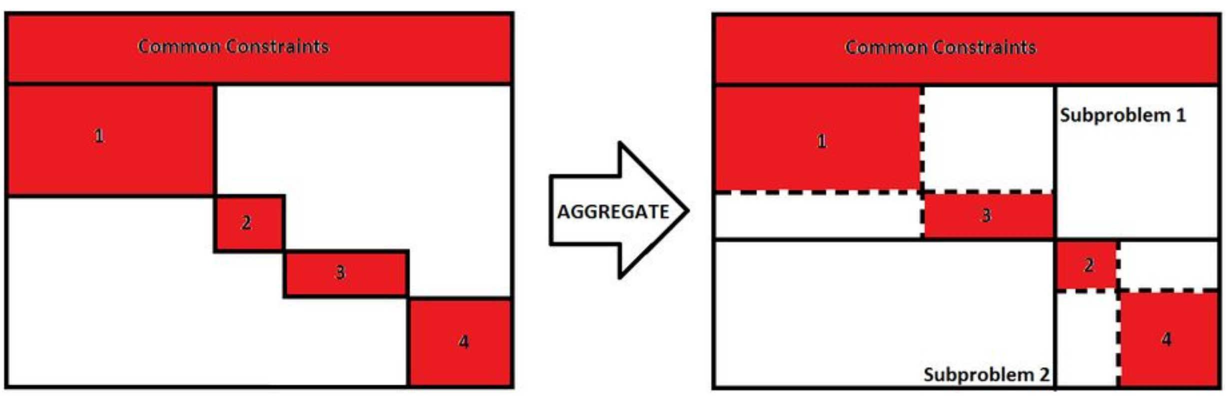

Figure 5. Aggregation of four blocks into two subproblems

In Section 2.2, we show that it is NP-Hard to optimally aggregate blocks of unequal size, into subproblems. In Section 2.3, we show that one can, within polynomial time, optimally aggregate blocks of equal size, into subproblems.

2.2 Optimally aggregating blocks of unequal size into subproblems: An NP-hard problem

Classifying the hardness of a problem is extremely important. If a problem is NP-hard, one can focus on developing efficient heuristics for the problem, rather than an efficient exact method (unless P=NP). Our aggregation problem is NP-hard, which is proved in Theorem 1.

Theorem 1: Optimally aggregating blocks of unequal size, into subproblems, is strongly NP-Hard

Proof: We proceed by showing that any instance of a strongly NP-Complete problem can be converted, within polynomial-time, into a question of whether or not the optimal objective of our aggregation problem can be equal to some value. The strongly NP-Complete problem we use is 3-PARTITION, one of whose versions is to decide whether or not a given set X of positive integers can be partitioned into three subsets, such that the sum of elements in every subset is the same. Create an instance of our aggregation problem with a set of block sizes that is equal to the set of positive integers $X = \{ x_1, x_2, \dots, x_k \}$ from any instance of 3-PARTITION. Define $n = \sum_{i=1}^{k} x_i$. Let our growth polynomial be given by $P(x) = a + x^2$, assuming that one can build a computing-platform with such a growth function in O($Q(log(a))$) time, where $Q(log(a))$ is a polynomial function of $log(a)$. It is clear that if a is very large, we would aggregate all blocks into a single subproblem of size n. It is also clear that if a is 0, we would leave all k blocks without any aggregation. If a is varied between 0 and the very large value, the optimal number of subproblems will vary from k to 1 (need not vary continuously among these integers). Due to the convexity of $P(x)$, for every value of the optimal number of subproblems, the optimal assignment of blocks to subproblems would occur when all subproblems have the same size, if it is possible. We now consider two cases:

CASE 1 (There exists three partitions of X, such the the sum of integers in each partition is equal to ($n/3$)). In this case, we first choose a such that the optimal aggregation strategy is to have 3 subproblems, each of size ($n/3$). By Theorem 2, this aggregation strategy is optimal when $P'(n/3) = (P(n/3) / (n/3))$, which happens when $a = n^2/9$. The optimal objective value of our aggregation problem in CASE 1 is thus equal to $(3(a + (n/3)^2)) = (2n^2/3)$.

CASE 2 (There do not exist three partitions of X, such the the sum of integers in each partition is equal to ($n/3$)). Set $a = n^2/9$. Now, the optimal aggregation strategy can have h subproblems of non-zero size, where h is some integer in [1,k]. Denote the sizes of the h subproblems as $\{y_1, y_2, \dots, y_h\}$. It is impossible for $y_1 = y_2 = \dots = y_h = (n/3)$, due to the condition imposed by CASE 2. Also the least value of $(P(x)/x)$ occurs when $x = (n/3)$. Therefore, $((\sum_{i=1}^{h} P(y_i))/ n) = (\sum_{i=1}^{h} P(y_i) / \sum_{i=1}^{h} y_i) = (\sum_{i=1}^{h} (y_i P(y_i)/y_i) / \sum_{i=1}^{h} y_i)$ is the weighted average of h values, each of which is greater than or equal to $(P(n/3)/(n/3))$, and one of which is strictly greater than $(P(n/3)/(n/3))$. This is due to convexity of $P(x)$. The optimal objective value of our aggregation problem in CASE 2 is thus equal to $((2n^2/3) + \Delta)$, where $\Delta > 0$.

From CASE 1 and CASE 2, by setting $a = (n^2/9)$, the answer to the 3-PARTITION problem is YES, if and only if, the optimal objective to the aggregation problem is equal to $(2n^2/3)$. We have reduced 3-PARTITION, within polynomial-time, into a question of whether or not the optimal objective of our aggregation problem can be equal to some value. This completes the proof of strong NP-Hardness. **Hence Proved**

2.3 Optimally aggregating blocks of equal size, into subproblems, within polynomial time

When all blocks are of equal size, the blocks can be easily divided among the IPUs, such that the difference between the sum of block-sizes on any two IPUs, is not more than the size of a single block. Our assumption that the blocks can be efficiently divided roughly equally among the IPUs holds very well, and hence, we continue to focus on minimizing solution-time on a single IPU. Denote the size of each block as s, the sum of sizes of blocks per IPU as n, and hence the total number of blocks per IPU as (n/s). We first give Theorem 2 for obtaining the optimal subproblem size under some assumptions of continuity.

Theorem 2: The optimal size (denoted as x_{opt}) for a subproblem, assuming x_{opt} and (n/x_{opt}) are both integers, is the value of x for which the least-sloped tangent to $P(x)$ meets the origin.

Proof: One possible candidate for the optimal size, will involve setting all subproblems to have the same size x_{opt}, since if we are able to find the optimal size for one subproblem, then all subproblems can be set to that same size, so that overall solution time is minimized for all subproblems. The sum of sizes of all subproblems is n. Assuming x to be an integer, denote the size of each subproblem as x. So the solution time for each subproblem is $P(x)$, and the number of subproblems is (n/x), where assume (n/x) to also be an integer. Denoting T to be the total solution time for all subproblems on an IPU, we can say that $T = (n/x)\, P(x) = n(P(x)/x)$. Since n is fixed, the global minimum of T has the least value of $(P(x)/x)$. The set of stationary points of T are obtained by setting $dT/dx = 0$ giving $P'(x) = (P(x)/x)$, where $P'(x)$ represents $d(P(x))/dx$. This means x_{opt} is the value of x, at which the least-sloped tangent to $P(x)$ meets the origin. **Hence Proved**

If x_{opt} is an integer multiple of s, and if n is an integer multiple of x_{opt}, we are done by setting the sizes of all subproblems to x_{opt}. However, if x_{opt} is not an integer multiple of s, or, if n is not an integer multiple of x_{opt}, we may follow a Dynamic Programming (DP) approach that gives an optimal aggregation strategy in $O((n/s)^2)$ time. We give Theorem 3, which converts our problem of optimal aggregation of blocks of equal size, into a DP problem, and gives the DP approach.

Theorem 3: Let $P(x)$ be an arbitrary function with domain $[1, n]$. Let all (n/s) blocks be of equal size s. The optimal solution to our aggregation problem can be obtained in $O((n/s)^2)$ time using a Dynamic Programming approach.

Proof: Denoting x_i as the number of subproblems to be formed of size si (i.e. with i blocks), our aggregation problem is equivalent to the following Integer Linear Program (ILP):

Minimize: $\sum_{i=1}^{(n/s)} x_i P(is)$

Subject to: $\sum_{i=1}^{(n/s)} (ix_i) = (n/s)$

x_i is an integer variable in [0, (n/s)], for all integers i in [1, (n/s)]

The above ILP formulation is the well-known Unbounded-Knapsack problem with equality-constraint, for which the following pseudo-polynomial DP algorithm is also well-known [22], by converting into a shortest path problem:

1) Define (n/s) states $\{f_0, f_1, f_2, \dots f_i \dots, f_{(n/s)}\}$, where f_i denotes the optimal solution time of i blocks, each of size s
2) For all $i > j$, $c_{ij} = P((i\text{-}j)s)$
3) $f_0 = 0$
4) $f_i = min_{\, j \in \{ i-1, i-2, \dots , 2, 1, 0\}} (c_{ij} + f_j)$, for all integers i in [1, (n/s)]
5) Initialize $x_i = 0$ for all integers i in [1, (n/s)]
6) Retrace the path backwards starting from $f_{(n/s)}$, where from each state i, we choose to go to state t, where $t{=}argmin_{\, j \in \{ i-1, i-2, \dots , 2, 1, 0\}} (c_{ij} + f_j)$, at the same time updating $x_{i-t} = x_{i-t} +1$

The complexity is clearly O(1 + 2 + 3 + ... + ((n/s) - 1) + (n/s)) = O((n/s)^2). **Hence Proved**

When $P(x)$ is known to be convex or concave, we can use certain properties to obtain algorithms that are less complex than O((n/s)^2). Theorems 4 and 5 describe these properties for convex and concave functions respectively.

Theorem 4: Let $P(x)$ be a convex function in the domain [sr_1, sr_2], where r_1 and r_2 are positive integers. Let all (n/s) blocks be of equal size s. If a feasible solution exists to the aggregation problem, there exists an optimal solution that will consist of not more than two unique subproblem sizes in [sr_1, sr_2], and these two sizes will differ by not more than s.

Proof: Assume there exists an optimal solution having more than two unique subproblem sizes. Iteratively subtract s from the size of the largest subproblem, and add s to the size of the smallest subproblem, causing overall solution time to either reduce or remain the same (due to convexity property). In this case, since our solution was assumed to be optimal, the overall solution time remains the same. We eventually get no more than two unique subproblem sizes, which differ by not more than s. **Hence Proved**

Theorem 5: Let $P(x)$ be a concave function in the domain [sr_1, sr_2], where r_1 and r_2 are positive integers. Let all (n/s) blocks be of equal size s. If a feasible solution exists to the aggregation problem, there exists an

optimal solution that will consist of not more than three unique subproblem sizes, belonging to the set {sr_1, sr, sr_2}, where $r_1 < r < r_2$, and r is some integer.

Proof: Assume that an optimal solution has four or more unique subproblem sizes in [sr_1, sr_2]. Iteratively consider any two of these subproblems, whose sizes are neither equal to sr_1 nor equal to sr_2, and subtract s from the size of the smaller subproblem and add s to the size of the larger subproblem, causing overall solution time to either reduce or remain the same (due to concavity property), until one of their sizes is equal to either sr_1 or sr_2. In this case, since our solution was assumed to be optimal, the overall solution time remains the same. We eventually get not more than three unique subproblem sizes, belonging to the set $\{sr_1, sr, sr_2\}$, where $r_1 < r < r_2$, and r is some integer.

Hence Proved

We now use the property described in Theorem 4, to develop an aggregation approach for convex *P(x)* that is less complex than the O($(n/s)^2$) DP approach. Theorem 6 describes this.

Theorem 6: Let *P(x)* be a convex function with domain [1, *n*]. Let all (*n/s*) blocks be of equal size *s*. Let *rs* be the smallest integer multiple of *s* greater than x_{opt}. The optimal solution to the aggregation problem, can be obtained in:

1) **O(*n/s*) time for all $(n/s) \leq (r^2 - r - 1)$**
2) **O(1) time for all $(n/s) > (r^2 - r - 1)$**

Proof: We consider three cases:

CASE 1 ($x_{opt} \leq 1$): This is a trivial case in which the optimal solution to the aggregation problem is to have no-aggregation (i.e. each block becomes a separate subproblem). Solution time is O(1).

CASE 2 ($x_{opt} \geq n$): This is a trivial case in which the optimal solution to the aggregation problem is to perform full-aggregation (i.e. all blocks are combined into a single subproblem). Solution time is O(1).

CASE 3 ($1 < x_{opt} < n$): Since *P(x)* is convex, Theorem 4 applies. In the optimal solution, denote the two unique sizes of the smaller and larger subproblems as (ks) and $((k+1)s)$ respectively, where k is an integer in [1 , ((*n/s*)-1)]. Hence, for each of the ((*n/s*)-1) values of *k*, we can obtain a separate ILP:

Minimize: $x_1 P(ks) + x_2 P((k+1)s)$

Subject to: $x_1 (k)s + x_2 (k+1)s = n$

Each of $\{x_1, x_2\}$ is a non-negative integer variable $\leq (n/s)$

For each value of *k*, the above ILP can be solved by an algorithm [16][17] within constant time, since the number of variables and constraints is constant. We choose the solution with minimum objective value from amongst the ((*n/s*)-1) ILPs. Overall complexity is thus O(*n/s*).

Due to convexity of *P(x)*, the two integer multiples of *s*, with the least values of ($P(x)/x$), are among the set $\{(r-2)s, (r-1)s, rs, (r+1)s\}$. Applying the formula [21] of the Frobenius number of two coins with denominations *rs* and $(r+1)s$, the equation: $x_1 s(r)+x_2 s(r+1)=n$, or equivalently: $x_1(r)+x_2(r+1)=(n/s)$, has a solution for all values of $(n/s) > (r(r+1)-(r+1)-r) = (r^2 - r - 1)$. The value of $(r^2 - r - 1)$ attains a minimum at r=0.5, meaning that $(r^2 - r - 1)$ increases for all r>1. This also means that for all integers k in [1, r], the general equation $(x_1(k)+x_2(k+1)=(n/s))$ has a solution if $(n/s) > (r^2 - r - 1)$. Using Theorem 4 and for minimizing the weighted average of $(P(x)/x)$, it suffices to consider only three ILPs, the first ILP being with $k=(r-2)$, the second ILP being with $k=(r-1)$, and the third ILP being with $k=r$. Hence, overall complexity is O(1) for all $(n/s) > (r^2 - r - 1)$. **Hence Proved**

We demonstrate a simple example showing that Theorem 6 gives the same optimal aggregation strategy as DP. Consider a convex $P(x) = (x^2 - 16x + 98)$, and s=2 (i.e. block-size is 2). Solving for the least sloped tangent to $P(x)$ from the origin using $P'(x)=P(x)/x$, yields $x_{opt} = 7\sqrt{2} \approx 9.899$. So for Theorem 6, we have rs=10 and r=5 and $(r^2 - r - 1)$=19.

Now let us say n=24, which means (n/s)=12, which is lesser than 19. So we formulate the following 11 ILPs, for each integer k in [1,11], where ILP_k is as follows:

<u>Minimize</u>: $x_1 P(2k) + x_2 P(2k+2)$

<u>Subject to</u>: $kx_1 + (k+1)x_2 = 12$

Each of $\{x_1, x_2\}$ is a non-negative integer variable ≤ 12

The optimal objective values of ILP_1 = 300, ILP_2 = 152, ILP_3 = 102, ILP_4 = 102, ILP_5 = 100, ILP_6 = 100, ILP_{11} = 290. ILP_7 , ILP_8 , ILP_9 and ILP_{10} are infeasible. So we may choose the optimal aggregation strategy from either ILP_5 or ILP_6 , which have the least objective values of 100. This yields two subproblems each of size 12 (or two subproblems each with 6 blocks), which is the optimal aggregation strategy obtained using Theorem 6. DP is verified to yield the same optimal aggregation strategy.

Now let us say n=100, which means (n/s)=50, which is greater than 19. As per the second part of Theorem 6, we need to formulate only the following 4 ILPs, for each integer k in [3,4,5,6], where ILP_k is as follows:

<u>Minimize</u>: $x_1 P(2k) + x_2 P(2k+2)$

<u>Subject to</u>: $kx_1 + (k+1)x_2 = 50$

Each of $\{x_1, x_2\}$ is a non-negative integer variable ≤ 50

The optimal objective values of ILP_3 = 450, ILP_4 = 380, ILP_5 = 380, ILP_6 = 440. So we may choose the optimal aggregation strategy from either ILP_4 or ILP_5 , which have the least objective values of 380. This yields ten subproblems each of size 10 (or ten subproblems each with 5 blocks), which is the optimal aggregation strategy obtained using Theorem 6. DP is verified to yield the same optimal aggregation strategy.

We now use the property described in Theorem 5, to develop an aggregation approach for concave $P(x)$ that is less complex than the $O((n/s)^2)$ DP approach. Theorem 7 describes this.

Theorem 7: Let *P(x)* be a concave function with domain [1, *n*]. Let all (*n/s*) blocks be of equal size *s*. The optimal solution to the aggregation problem, can be obtained in O((*n/s*)) time

Proof: Since *P(x)* is concave, Theorem 5 applies. In the optimal solution, denote the unique size of the subproblem that is between the smallest and largest integer multiples of *s*, as (*ks*), where *k* is some integer in [2, ((*n/s*)-1)]. Hence, for each of the ((*n/s*)-2) values of *k*, we can obtain a separate ILP:

Minimize:

$$x_{SMALLEST}\,P(k_{SMALLEST}\,s) + x\,P(ks) + x_{LARGEST}\,P(k_{LARGEST}\,s)$$

Subject to:

$$x_{SMALLEST}\,(k_{SMALLEST}\,s) + x\,(ks) + x_{LARGEST}\,(k_{LARGEST}\,s) = n$$

$$k_{SMALLEST} = 1$$

$$k_{LARGEST} = (n/s)$$

Each of $\{x_{SMALLEST}, x, x_{LARGEST}\}$ is a non-negative integer variable $\leq (n/s)$

For each value of *k*, the above ILP can be solved by an algorithm [16][17] within constant time, since the number of variables and constraints is constant. We choose the solution with minimum objective value from amongst the ((*n/s*)-2) ILPs. Overall complexity is thus O(*n/s*). **Hence Proved**

We demonstrate a simple example showing that Theorem 7 gives the same optimal aggregation strategy as DP. Consider a concave $P(x) = (1000 + 5000x - x^2)$, and s=2 (i.e. block-size is 2).

Now let us say n=12, which means (n/s)=6. So we formulate the following 4 ILPs, for each integer k in [2,5], where ILP_k is as follows:

Minimize:

$$x_{SMALLEST}\,P(2) + x\,P(2k) + x_{LARGEST}\,P(12)$$

Subject to:

$$x_{SMALLEST} + x\,(k) + 6\,x_{LARGEST} = 6$$

Each of $\{x_{SMALLEST}, x, x_{LARGEST}\}$ is a non-negative integer variable ≤ 6

The optimal objective values of $ILP_2 = ILP_3 = ILP_4 = ILP_5 = 60856$. So we may choose from any of these ILPs, all of which yield a single subproblems of size 12 (i.e. a single subproblems with 6 blocks), which is the optimal aggregation strategy obtained using Theorem 6. DP is verified to yield the same optimal aggregation strategy.

Having given Theorems 6 and 7 that cover the cases when *P(x)* is convex and concave respectively, we now give Theorem 8 that describes the optimal aggregation approach if *P(x)* is linear.

Theorem 8: Let *P(x)* be a linear function with domain [1, *n*]. Let all (*n/s*) blocks be of equal size *s*. The optimal solution to the aggregation problem, can be obtained in O(1) time

Proof: Let $P(x) = a+bx$. The solution-time of all subproblems = $ka+bn$, where k is the number of subproblems formed, and hence k is an integer in [1, (n/s)]. If $a \geq 0$, to minimize the value of $ka+bn$, we minimize the number of subproblems formed (i.e. aggregate all blocks into a single subproblem). If $a < 0$, to minimize the value of $ka+bn$, we maximize the number of subproblems formed (i.e. treat each block as a separate subproblem). The optimal aggregation strategy is thus devised in O(1) time. **Hence Proved**

We demonstrate a simple example showing that Theorem 8 gives the same optimal aggregation strategy as DP. Consider $P(x) = 10+5x$ with n=6 and s=2. Theorem 8 says that we need to aggregate all blocks into a single subproblem, and DP is verified to give the same optimal aggregation strategy. Now consider $P(x) = -1+10x$ with n=6 and s=2. Theorem 8 says that we need to obtain 3 subproblems each of size 2 (i.e. 3 subproblems each with a single block), and DP is verified to give the same optimal aggregation strategy.

We now put together all our theories and develop the flowchart for the optimal aggregation strategy, in Figure 6.

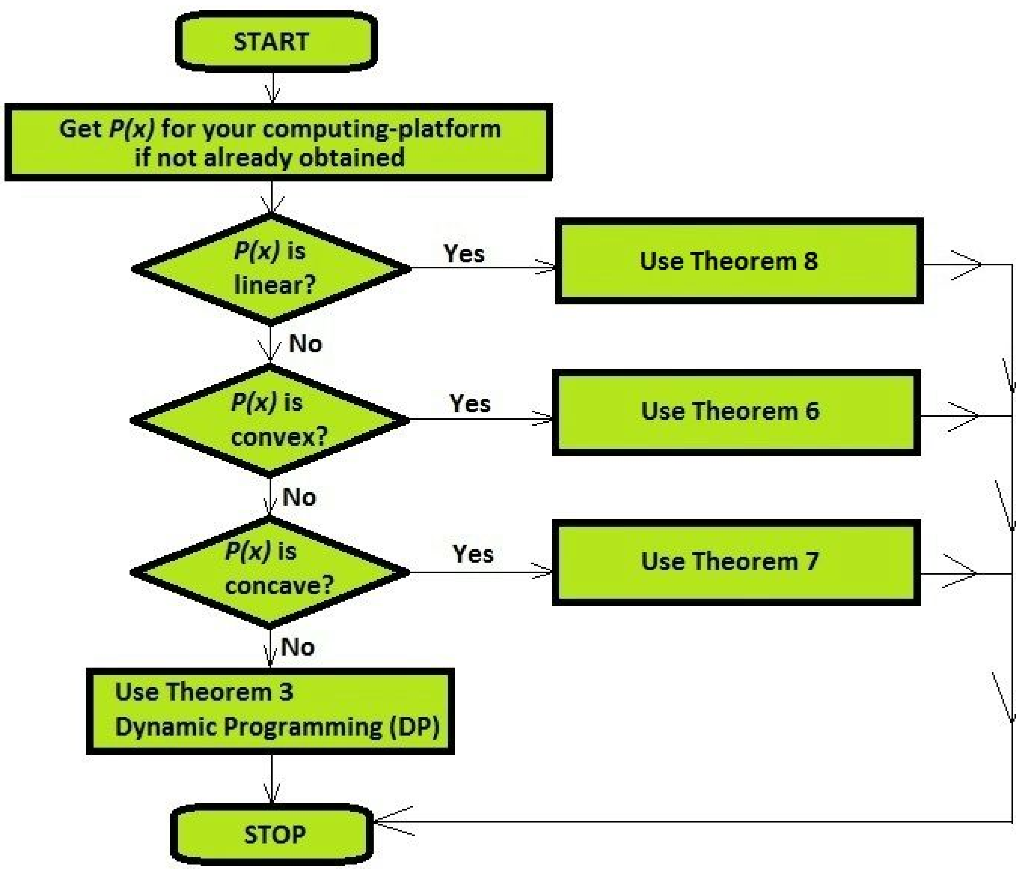

Figure 6. The optimal aggregation strategy for an LP with BDS having equal-sized blocks

Having developed an efficient strategy to optimally aggregate equal-sized blocks into subproblems, we are in a position to implement them on a real LP having BDS. Chapter 3 describes the experiments we conducted.

CHAPTER 3

EXPERIMENTS

We describe the computing platforms we worked on, the types of experiments we conducted to ensure that optimal aggregation strategies were worth pursuing, how we implemented the theory of chapter 2 for optimal aggregation and the assumptions made while doing so, and how we eliminated noise in our experiments from foreign processes. We also discuss the benefits our optimal aggregation strategy has over the two most trivial aggregation strategies (i.e. full-aggregation and no-aggregation).

3.1 Computing platforms including linear programming solvers used

We used two computing platform hardware for our experiments.

1) A high performance computing (HPC) platform in the Holmes Hall 409 Lab, called REDR.
2) A personal Windows laptop called DPC-Windows-Laptop.

The specifications of the REDR computing platform are as follows:

1) A machine with 12 physical cores, and 24 logical cores. Each core has several layers of cache, and all cores shared more than 60 GB RAM.
2) 64 bit Linux OS
3) Commercial complete version of CPLEX LP solver software installed
4) Free LP solver software installed called GLPK
5) Python software with Pyomo Libraries for scripting and interaction with the LP solvers.

The specifications of the DPC Laptop are as follows:

1) A Laptop with 2 physical cores, and 4 logical cores, each at 2.3 GHz, and 6 GB RAM.
2) 64 bit Windows OS
3) Student version of CPLEX LP solver software installed
4) Python software with Pyomo Libraries for scripting and interaction with the LP solvers.

3.2 Forcing not more than one core to be used at all instants of time, by Python process and LP-solver process

Since our aim was to evaluate our optimal aggregation strategy on a machine with a limited number of IPUs (i.e. cores), and since we made the reasonable assumption that blocks can be efficiently divided roughly equally among the cores, we decided to force the number of cores used at any instant of time to be one. This was accomplished by:

1) Forcing the Python process to send subproblems serially to the solver process (using the “serial” specification shown in the screenshot in Figure 7)

2) Forcing the solver process to use only a single thread (using the "threads" specification shown in the screenshot in Figure 7)

```
opt = SolverFactory("cplex", solver_io="nl")
#opt = SolverFactory("glpk")
opt.options["threads"] = 1
solver_manager = SolverManagerFactory("serial")
```

Figure 7. Forcing not more than a single core to be used at all instants of time

3.3 Types of solution-times measured

3.3.1 Types of times measurable on the REDR (Linux) machine

Since REDR was shared by multiple users, we had to measure CPU-time, rather than wall-clock-time. Since REDR was a Linux machine, we were able to obtain seven subsets of CPU timings:

1) Python user-time, which is the CPU time spent on tasks of the Python process within its address space.
2) Python system-time, which is the CPU time spent on tasks of the Python process outside its address space, and in input or output.
3) Python cpu-time, which is the sum of Python user-time and Python-system-time.
4) LP-solver user-time, which is the CPU time spent on tasks of the LP Solver process within its address. space. This is a measure of performance if the OS and the scripting process (i.e. python in this case) were to occupy negligible time.
5) LP-solver system-time, which is the CPU time spent on tasks of the LP Solver process outside its address space, and in input or output.
6) LP-solver-cpu-time, which is the sum of LP-solver-user-time and LP-solver-system-time.
7) Overall-cpu-time, which is the summation of Python-cpu-time and LP-solver-cpu-time, and which is a measure of true performance on that computing platform.

We use the *os.times()* in-built method of Python [26], which returns a 5-tuple set of timing elements:

1) user-time of process, which corresponds to the user-time of the Python process
2) system-time of process, which corresponds to the system-time of the Python process
3) user-time of children processes that corresponds to the user-time of the LP Solver process
4) system-time of children processes that corresponds to the system-time of the LP Solver process
5) wall-clock-time

The *os.times()* method is invoked prior to, and after the calls to the solver from Python, as shown in Figure 8:

```
start_ostimes = os.times()
solve_all_instances(solver_manager, opt, subproblem_instances, suffixes=['rc','dual'])
end_ostimes = os.times()
```

Figure 8. Measurement of different types of solution-time

With reference to Figure 8:

1) The *solve_all_instances()* method is a Pyomo in-built method, and our obtain the 5-tuple set of times spent in this Pyomo method, by taking the difference of *start_ostimes* and *end_ostimes* variables.
2) The *solver_manager* object is specified to solve the subproblems serially (and not parallely), while *opt* is a pointer (or handle in Python terminology) to the LP Solver instance with a pre-specified flag to indicate that the Solver should have only a single thread (CPLEX explicitly needs the single-thread option to be specified, while GLPK does not since GLPK does not support multiple threads). Thus, both the Python process and the LP Solver process, are iteratively invoked sequentially, and each process is forced to use only a single thread. This satisfies the aim of our research, which is to study the effect of aggregation of blocks into subproblems, where the number of IPUs (i.e. cores in this case) is limited to one.
3) *subproblem_instances* is a pointer to the list of subproblems that have been created for the present iteration.

3.3.2 On the Windows laptop

The only reliable metric on the windows laptop is wall-clock-time, since the other values of CPU-time obtained by *os.times()* do not work on Windows (it only works on Linux). However, wall-clock-time is sufficient for our purpose since the windows laptop is a dedicated machine used only for the purpose of our experiments, so lesser noise affects it.

3.4 Data for linear programs

We use two types of LP with BDS for our study:

3.4.1 A scaled down version of SWITCH

SWITCH [18] is a large LP model, which identifies the least cost strategy for building and operating renewable generators, conventional generators, transmission, and storage, to satisfy demand and also meet emission reduction standards in a large power system, over a multi-decade period. SWITCH is a loose acronym for Solar, WInd, Conventional and Hydroelectric generation and Transmission.

Typically, LP models make decisions which constrain operations under different possible sets of future conditions. In SWITCH, each day is modelled as a separate set of conditions. Operations on many independent days are constrained by a shared set of prior investment decisions.

One way of writing out the constraint matrix in SWITCH, is so that the constraints of each day form a separate diagonal block. The typical LP structure of the original SWITCH model, is shown in Figure 9. By relaxing the annual emission cap constraint, and by fixing the values of the common variables, the SWITCH LP becomes separable into a number of diagonal blocks, each of which corresponds to a day.

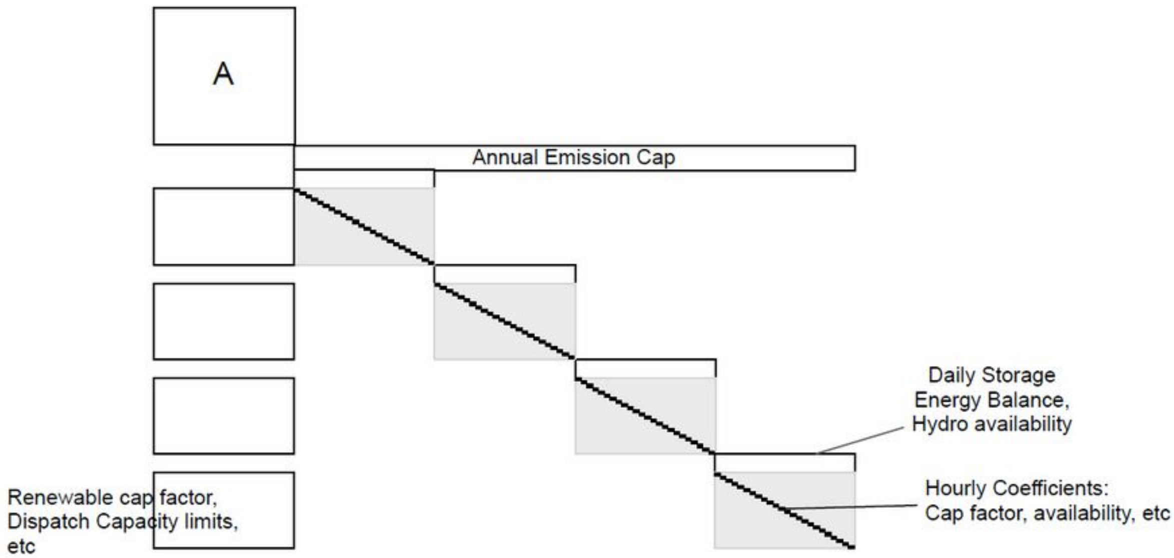

Figure 9. Structure of LP in the original version of SWITCH

The scaled-down-version of SWITCH LP we used, is a subset of the published version of SWITCH without annual emission cap constraints, and is a multi-zonal model with capabilities for gas, renewable energy, storage and transmission. The data (for 18 zones of California, over 96 days with 24 hours per day) used for this scaled down version of SWITCH, is the same data used by the published version of SWITCH.

For the scaled down SWITCH-LP, simulations are repeated for each of 20 cases of different combinations of storage-rate and storage-cost, which relate to parameters of battery storage. The 20 cases are basically for all values of storage-rate in {10000, 5000, 1000, 500, 100} and for all values of storage-cost in {1000, 100, 10, 1}.

3.4.2 Randomly-generated-synthetic LP

In this LP, we first formulate the whole LP with BDS, where the coefficients of each block are initialized using a uniform random generator from integers between 0 and 100000, with an initial seed of 13 for the first column of the first row of the first block. The only purpose of this synthetic LP was to be used in the sanity-test experiments (described in Section 3.5) along with the SWITCH-LP.

For the randomly-generated-synthetic LP, we freeze the whole LP after its formation, and we do not invoke the random generator again, and we proceed on our sanity test by dividing blocks equally among the subproblems. We repeat our sanity experiments for each of 6 cases of the number of rows and columns per block being 10, 20, 40, 100, 200 and 500. As the size of each block is increased during our experiments, the number of blocks are scaled down appropriately to ensure that the memory utilization is well-below 60% of the maximum RAM of the machine we are running on.

3.5 Sanity-test experiments

Using both types of LP mentioned in Section 3.4, we perform a sanity test, to evaluate whether it is worthwhile developing non-trivial strategies for optimal aggregation, or whether a trivial aggregation strategy (for example, full-aggregation or no-aggregation) might always be optimal. We consider various ways of having the blocks divided equally among the subproblems, and obtain the solution-time of the set of subproblems for each of these ways. For example, there are 6 ways of obtaining subproblems from an LP with 12 blocks (1 subproblem with 12 blocks per subproblem, 2 subproblems with 6 blocks per subproblem, 3 subproblems with 4 blocks per subproblem, 4 subproblems with 3 blocks per subproblem, 6 subproblems with 2 blocks per subproblem, and 12 subproblems with 1 block per subproblem). For simplicity in our sanity test, we group adjacent blocks into the same subproblem wherever possible, and do not consider multiple combinations of blocks being assigned to subproblems.

The sanity test consists of checking whether the least cpu-time for solving the set of subproblems occurs when the aggregation strategy is not trivial. The two trivial boundary-case aggregation strategies we consider are zero-aggregation (where the size of each subproblem is equal to the size of a block) and full-aggregation (where all blocks have been aggregated into a single subproblem).

We will be interested in checking whether at least one of the following metrics uncovers aggregation strategies that are better (i.e. give lesser solution-times) than the two trivial boundary case strategies, which will imply that it is worth developing strategies for optimal aggregation.

1) Solver-user-time, since it gives a measure of performance if the OS and scripting process were both perfect (i.e. occupy negligible cpu-times)
2) Solver-cpu-time (sum of Solver-user-time and Solver-system-time), since it gives a measure of performance if the scripting process were perfect
3) Overall-cpu-time (sum of Python-cpu-time and Solver-cpu-time), since it gives a measure of true performance on the current computing platform

3.5.1 Sanity-test with CPLEX, using scaled-down-SWITCH-LP, on REDR machine

Figure 10 plots various aspects of cpu-time of the set of subproblems, versus the number of equal-sized subproblems, for all the 20 cases of storage-rate and storage-cost. The aspects of cpu-time plotted are cplex-user-time (top-left), python-user-time (top-right), cplex-system-time (bottom-left) and overall-cpu-time (bottom-right). Python-system-time was observed to be very similar in behavior to cplex-system-time and hence has been omitted from the figure.

Figure 10 shows that as the number of equal-sized subproblems increase, cplex-system-time (and python-system-time) increases, while cplex-user-time takes a U-shape, giving a U-shape to overall-cpu-time as well. The left-side of the U-shape with overall-cpu-time is mild (but the U-shape is present). It is therefore worthwhile to obtain non-trivial optimal aggregation strategies to minimize cplex-user-time or overall-cpu-time.

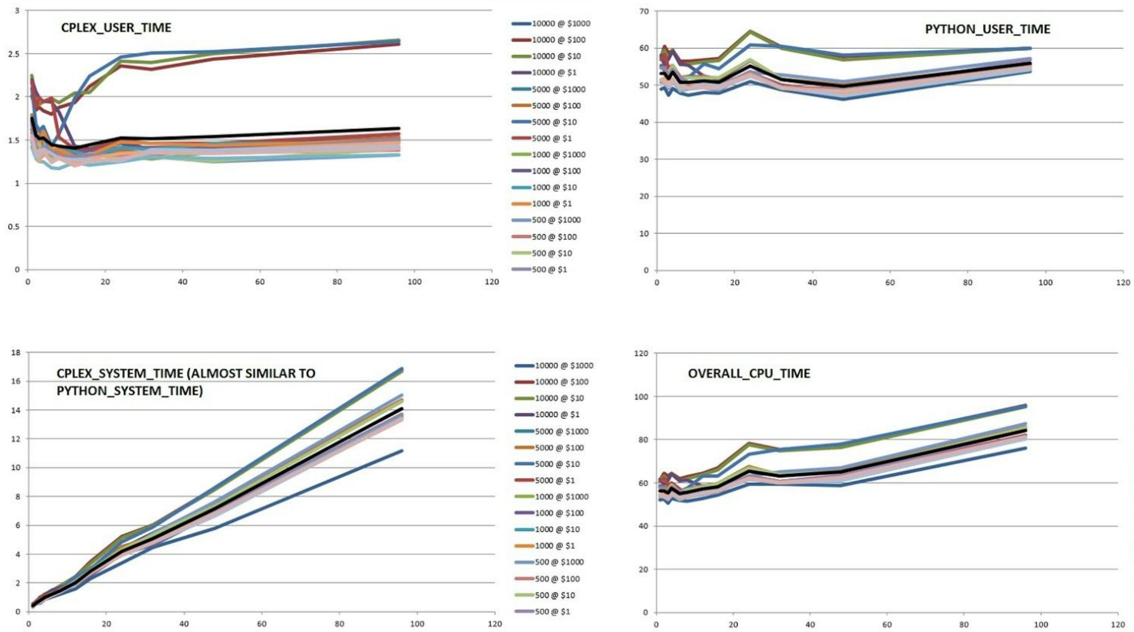

Figure 10. Sanity-test using CPLEX with scaled-down-SWITCH LP

3.5.2 Sanity-test with GLPK, using scaled-down-SWITCH-LP, on REDR machine

Figure 11 plots various aspects of cpu-time of the set of subproblems, versus the number of equal-sized subproblems, for all 20 cases of storage-rate and storage-cost. The aspects of cpu-time plotted are glpk-user-time (top-left), python-user-time (top-right), glpk-system-time (bottom-left) and overall-cpu-time (bottom-right). Python-system-time was observed to be very similar in behavior to glpk-system-time and hence has been omitted from the figure.

Figure 11 shows that as the number of equal-sized subproblems increase, glpk-user-time decreases, glpk-system-time (and python-system-time) increases, giving a U-shape to overall-cpu-time. It is therefore worthwhile to obtain non-trivial optimal aggregation strategies to minimize overall-cpu-time.

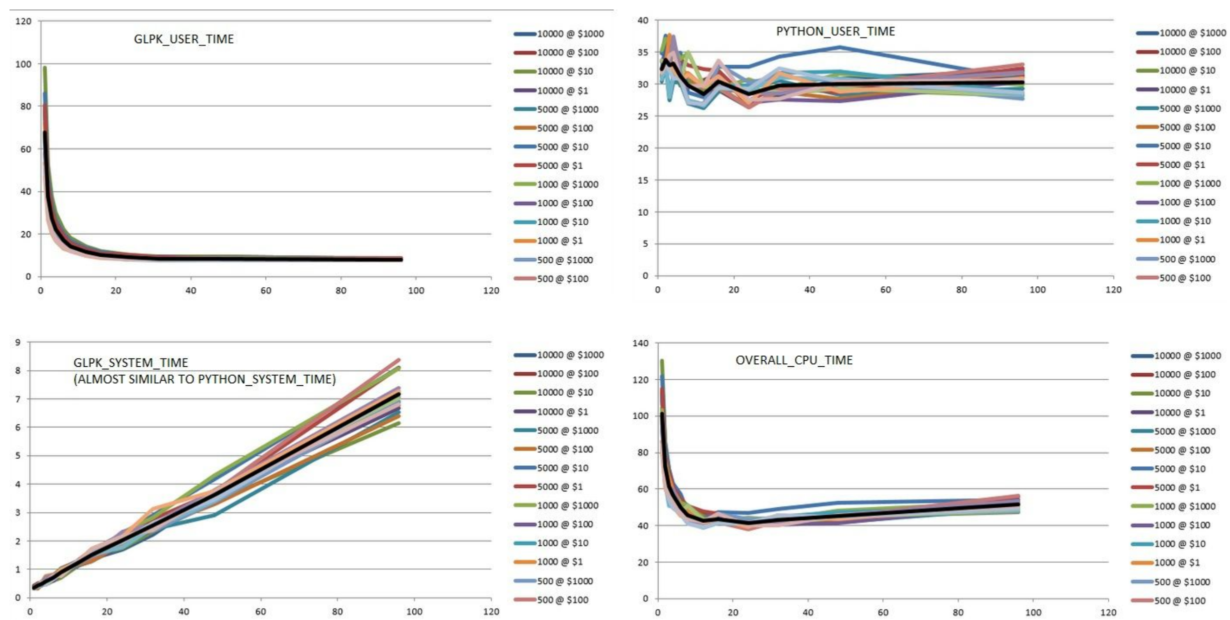

Figure 11. Sanity-test using GLPK with scaled-down-SWITCH-LP

3.5.3 Sanity-test with CPLEX, using randomly-generated-synthetic LP, on REDR machine

Figure 12 plots various aspects of cpu-time of the set of subproblems, versus the number of equal-sized subproblems. The aspects of cpu-time plotted are cplex-user-time (top-left), python-user-time (top-right), system-times of both cplex and python (bottom-left) and overall-cpu-time (bottom-right).

Figure 12 shows that as the number of equal sized-subproblems increases, the user-times of both cplex and python take on a U-shape, while the system-times of both cplex and python increase, giving a U-shape to overall-cpu-time. It is therefore worthwhile to obtain non-trivial optimal aggregation strategies to minimize cplex-user-time or overall-cpu-time.

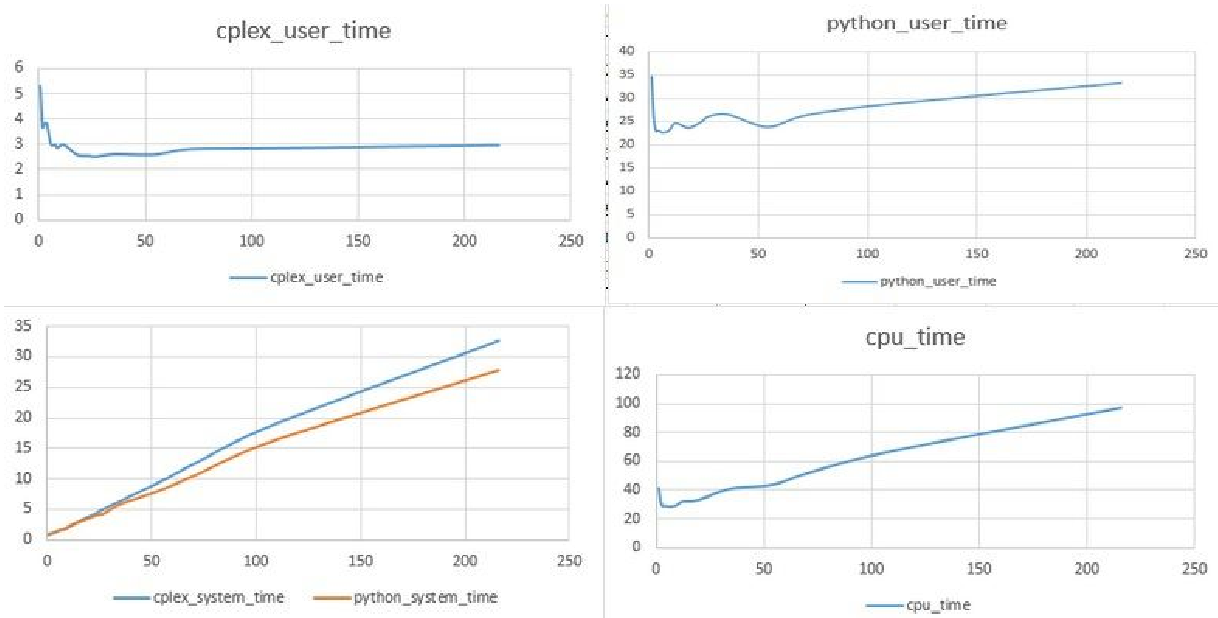

Figure 12. Sanity-test using CPLEX with randomly-generated-synthetic LP (216 blocks each with 200 rows and 200 cols)

3.5.4 Sanity-test with GLPK using randomly-generated-synthetic LP, on REDR machine

Figure 13 plots various aspects of cpu-time of the set of subproblems, versus the number of equal-sized subproblems. The aspects of cpu-time plotted are glpk-user-time (top-left), python-user-time (top-right), system-times of both glpk and python (bottom-left) and overall-cpu-time (bottom-right).

Figure 13 shows that as the number of equal sized-subproblems increases, the glpk-user-time decreases, the system-times of both glpk and python increase, the python-user-time takes on a U-shape, giving a U-shape to overall-cpu-time. It is therefore worthwhile to obtain non-trivial optimal aggregation strategies to minimize overall-cpu-time.

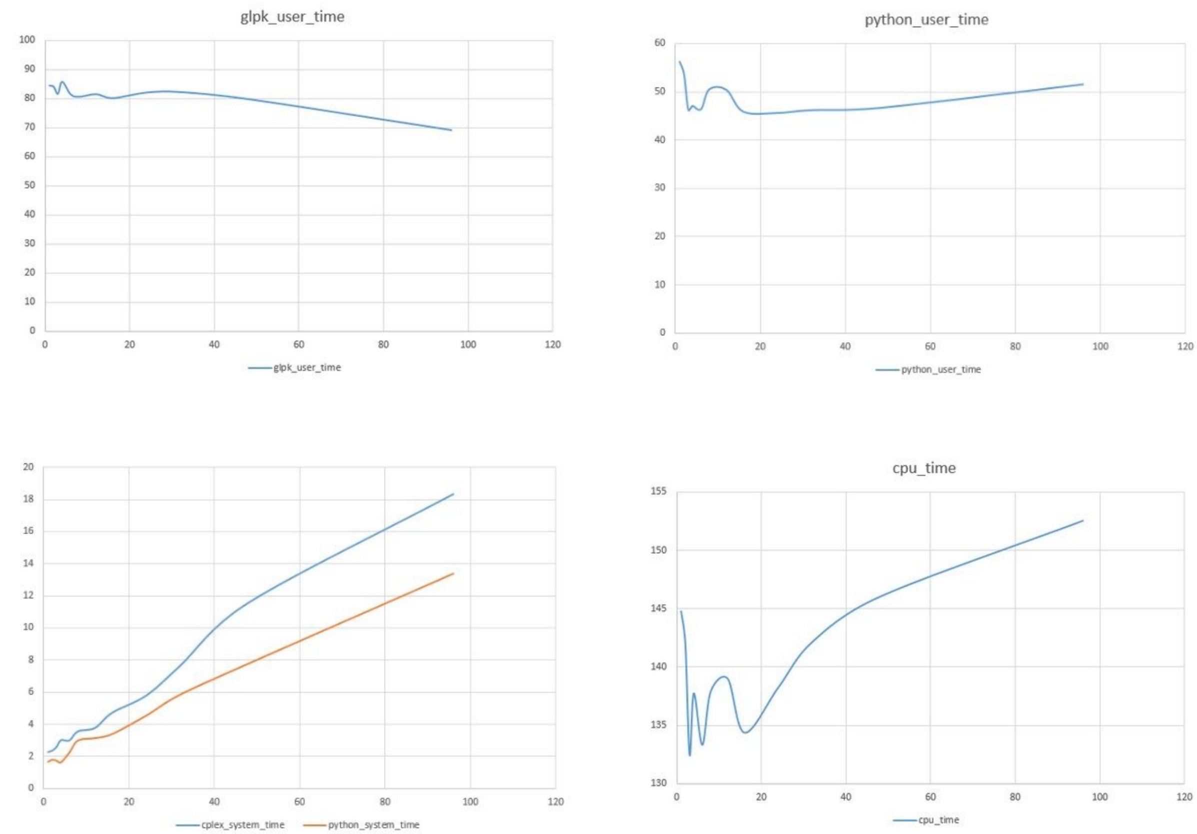

Figure 13. Sanity-test using GLPK with randomly-generated-synthetic LP (96 blocks each with 500 rows and 500 cols)

3.5.5 Sanity tests on the DPC-Windows-Laptop

On the Windows-Laptop, we were not able to test LP instances as big as those on REDR. However, there were some medium-sized LPs, which demonstrated non-trivial optimal aggregation strategies on Windows-Laptop. Table 1 shows consistent results with CPLEX, using the Windows Laptop for a randomly generated synthetic LP with 4 blocks, each block having 100 rows and 1000 cols. Since wall-clock-time was the only reliable metric, we first took precautions (mentioned in Section 3.8.2) to avoid noise. Next, to ensure that there was no garbage accumulation that could alter wall-clock-timings, we used two rules:

1) Without restarting the Laptop, we would close and re-open the command prompt (from which Python was invoked) and re-run the experiments in all 6 sequences, each sequence being a different permutation of the number of equal-sized subproblems, i.e 1, 2, and 4.
2) We re-ran the experiment, after restarting the Laptop, once for each aggregation strategy.

In Table 1, an aggregation strategy depicted by $\{y_1, y_2, ...y_i, ..., y_k\}$ denotes there should be k subproblems, with the

i^{th} subproblem having y_i blocks. Table 1 clearly shows that the optimal aggregation strategy is {2,2}, which does not belong to any of the two trivial boundary cases of aggregation.

Table 1. Wall-clock-timings on the Windows-Laptop, with CPLEX

	{1,1,1,1}	**{2,2}**	**{4}**
Rule 1 (sequence of subproblem-sizes is 1, 2, 4)	2.10599	1.87199	2.21499
Rule 1 (sequence of subproblem-sizes is 1, 4, 2)	2.15299	1.88800	2.40300
Rule 1 (sequence of subproblem-sizes is 2, 1, 4)	2.15299	1.82599	2.16899
Rule 1 (sequence of subproblems is 2, 4, 1)	2.01100	1.76300	2.1210
Rule 1 (sequence of subproblems is 4, 1, 2)	2.12100	1.82500	2.13700
Rule 1 (sequence of subproblems is 4, 2, 1)	1.93499	1.85599	2.04290
Rule 2 (restarting the Laptop each time)	2.028	1.85599	2.16900

3.5.6 Sanity tests with LPs having very small-sized blocks (no U-shaped curves)

In the experiments in Subsections 3.5.1-3.5.5, the blocks of the LPs were large and complex. This is what probably contributed to the U-shaped plots, possible reason for which we cover in Section 3.9. However, we found that when the blocks of LPs were very small and simple (less than 10 rows and cols per block), then on both REDR and on DPC-Laptop, the solvers (both CPLEX and GLPK) tended to show that the trivial strategy of full-aggregation was optimal. The possible reason for this is also covered in Section 3.9. Figures 14 and 15 show the sanity tests conducted with CPLEX and GLPK respectively, on REDR. Both these figures plot of various aspects of cpu time of the set of subproblems, versus the number of equally-sized subproblems. The aspects of cpu-time shown in both figures are solver-user-time (top-left), python-user-time (top-right), system-times of both solver and python (bottom-left) and overall-cpu-time (bottom-right).

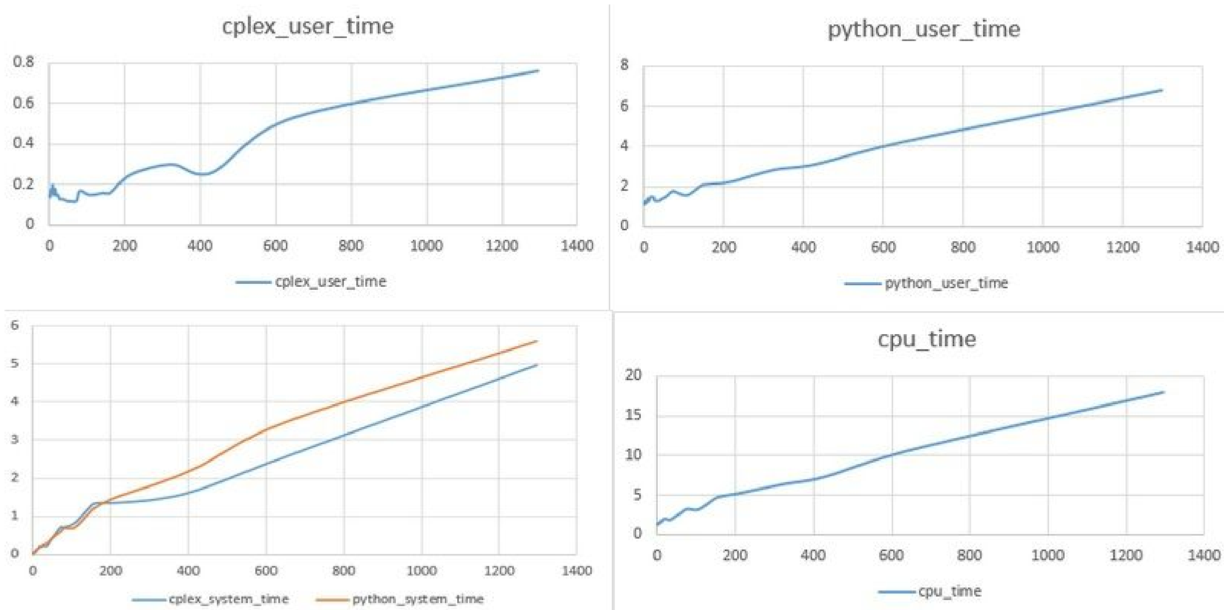

Figure 14. Sanity-test using CPLEX with randomly-generated-synthetic LP (1296 blocks each with 10 rows and 10 cols)

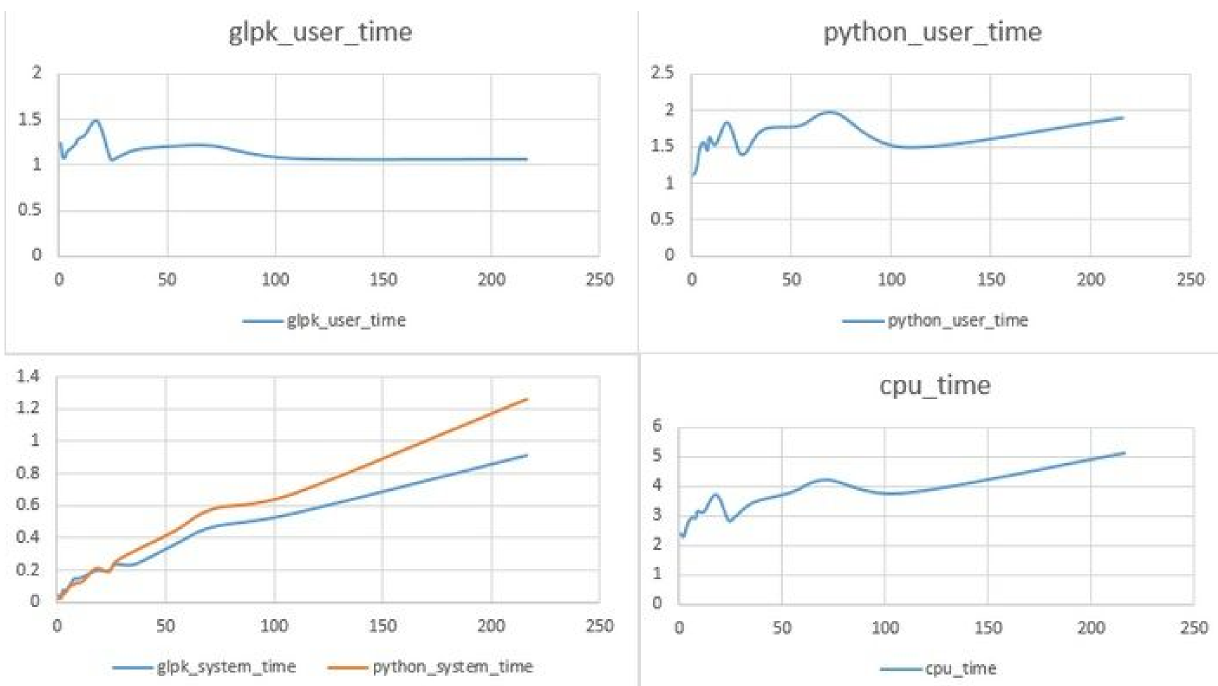

Figure 15. Sanity-test using GLPK with randomly-generated-synthetic LP (216 blocks each with 40 rows and 40 cols)

From these experiments, it is clear that there do exist some LPs with BDS with trivial optimal aggregation strategies. However, the aim of Subsections 3.5.1-3.5.6 together, is to show that there exists a large family of LPs with BDS (especially those with larger sized complex blocks) for every computing platform, whose optimal aggregation strategies are non-trivial.

Given the results of these sanity-test experiments, we have satisfied ourselves that there is a need to identify the optimal aggregation strategy and we proceed to develop, implement and test our optimal aggregation strategy. We use the procedure mentioned in Figure 6 in Chapter 2, to generate the optimal aggregation strategy. For our experiments, however, we relax the definition of *P(x)* that is is easier to obtain.

3.6 Relaxed definition of *P(x)*

For our experiments, we relax the definition of *P(si)* to be the "heuristic" average-case-solution-time of a subproblem consisting of *i* blocks of the given LP with BDS, where the LP size is *n* and each block is of the same size *s.* Here the "heuristic" we use is that for every value of *i*, we average the solution-times of *ceiling((n/(si)))* subproblems, each subproblem consisting of *i* randomly selected blocks. Section 3.7 describes our results we obtained by implementing the optimal aggregation strategy with the scaled-down SWITCH LP on the REDR machine, using CPLEX and GLPK.

3.7 Results of the optimal aggregation strategy

We used the optimal aggregation strategy mentioned in Figure 6, along with the relaxed definition of *P(x)* in Section 3.6, to minimize (average-case) overall-cpu-times. Subsections 3.7.1 and 3.7.2 show the results of our optimal aggregation strategy.

3.7.1 Results with CPLEX, using the scaled-down SWITCH LP

Using CPLEX, we experimentally obtained 20 growth-functions (for all values of storage-rate in {10000, 5000, 1000, 500, 100} and for all values of storage-cost in {1000, 100, 10, 1}). The average-case solution times of the subproblems were obtained for each subproblem whose size varied from 1 block to 96 blocks, in steps of 1. Figures 16 and 17 show the twenty growth functions, and their average respectively.

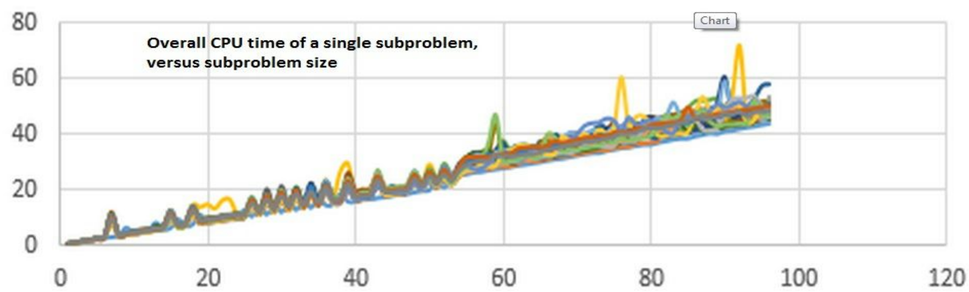

Figure 16. All twenty plots of *P(x)* using CPLEX with scaled-down-SWITCH-LP

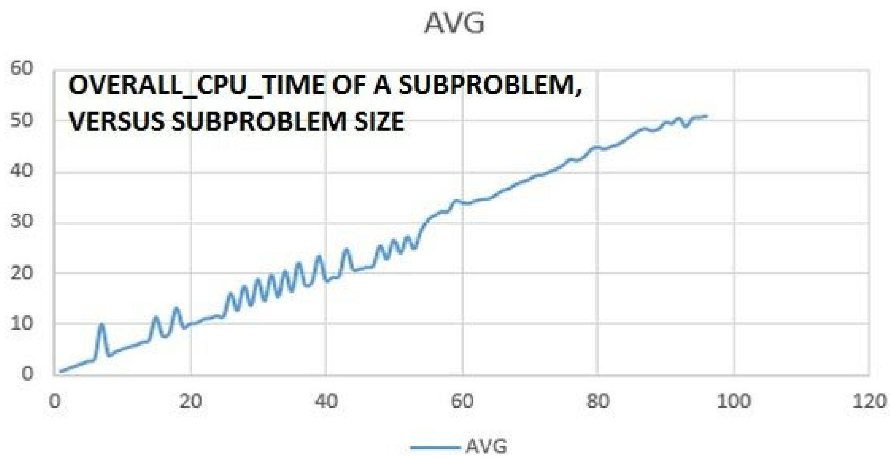

Figure 17. Average plot of *P(x)* using CPLEX with scaled-down-SWITCH-LP

Since, all 20 growth functions were non-convex, we had to only rely on the DP approach for the optimal aggregation strategy.

For each of these 20 cases, we then ran the DP algorithm, which generated theoretically optimal aggregation strategies. Once we obtained the theoretically optimal aggregation strategies for each of these 20 cases, we once again obtained the solution times for all the obtained aggregation strategies with each of the 20 cases, in addition to the trivial boundary aggregation strategies.

Table 2 shows the results of the DP approach with CPLEX, and compares results of three aggregation strategies (the optimum strategies generated by DP, those found so far by experiments, and the trivial boundary case aggregation strategy). In Table 2, an aggregation strategy depicted by $\{y_1, y_2, ...y_i, ..., y_k\}$ denotes there should be k subproblems, with the i^{th} subproblem having y_i blocks.

Table 2. Comparison of overall-cpu-timings of optimal strategy with trivial strategy, using CPLEX

Storage rate @Storage cost	**Theoretical optimum (by DP algorithm)**	**Experimental optimum (so far)**	**Best trivial boundary strategy**
10000 @1000	{2,47,47} 43.74 secs	{20, 38, 38} 41.33 secs	Full aggregation 41.69 secs
10000 @100	{23, 23, 25, 25} 42.93 secs	{3, 23, 35, 35} 42.52 secs	Full aggregation 43.92 secs
10000 @10	{2,47,47} 44.17 secs	{16, 16, 16, 16, 16, 16} 42.64 secs	Full aggregation 50.01 secs
10000 @1	{2,47,47} 44.92 secs	{20, 38, 38} 42.01 secs	Full aggregation 51.32 secs
5000 @1000	{42, 52} 45.67 secs	{32, 16, 16, 16, 16} 45.38 secs	Full aggregation 49.96 secs
5000 @100	{12, 42, 42} 44.36 secs	{20, 38, 38} 42.46 secs	Full aggregation 50.53 secs
5000 @10	{44, 52} 47.21 secs	Matches theoretical	Full aggregation 49.77 secs
5000 @1	{21, 21, 27, 27} 43.4 secs	{20, 38, 38} 42.46 secs	Full aggregation 50.33 secs
1000 @1000	{44, 52} 45.16 secs	Matches theoretical	Full aggregation 48.47 secs
1000 @100	{25, 31, 40} 49.93 secs	{20, 38, 38} 42.6 secs	Full aggregation 50.51 secs
1000 @10	{44, 52} 46.73 secs	Matches theoretical	Full aggregation 48.98 secs
1000 @1	{44, 52} 47.94 secs	Matches theoretical	Full aggregation 51.31 secs
500 @1000	{44, 52} 47.51 secs	Matches theoretical	Full aggregation 50.46 secs
500 @100	{20, 38, 38} 42.5 secs	Matches theoretical	Full aggregation 50.54 secs
500 @10	{44, 52} 46.09 secs	Matches theoretical	Full aggregation 47.85 secs

500 @1	{44, 52} 48.46 secs	Matches theoretical	Full aggregation 50.67 secs
100 @1000	{44, 52} 46.94 secs	Matches theoretical	Full aggregation 47.4 secs
100 @100	{44, 52} 47.98 secs	Matches theoretical	Full aggregation 50.28 secs
100 @10	{44, 52} 47.23 secs	Matches theoretical	Full aggregation 49.26 secs
100 @1	{44, 52} 47.82 secs	Matches theoretical	Full aggregation 50.47 secs

The time taken to run the DP algorithm (written in C and run on the Windows Laptop), for each of these 20 cases, is negligible (less than 0.001 secs).

With CPLEX, the average percentage saving of time by employing DP algorithm, instead of a trivial boundary case, over all 20 cases in Table 1 is equal to (983.73 - 920.69) / 920.69 = 6.8%.

3.7.2 Results with GLPK, using scaled-down-SWITCH LP

We followed a similar approach with GLPK, as followed for CPLEX. Figures 18 and 19 show the plots of *P(x)* for the twenty cases, and for their average, respectively.

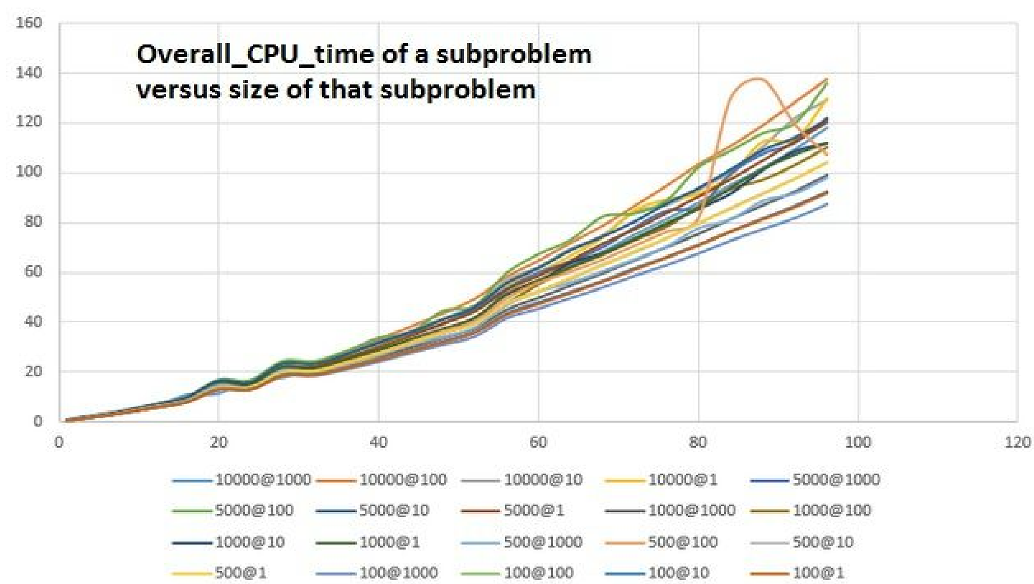

Figure 18. All twenty plots of *P(x)* using GLPK with scaled-down-SWITCH LP

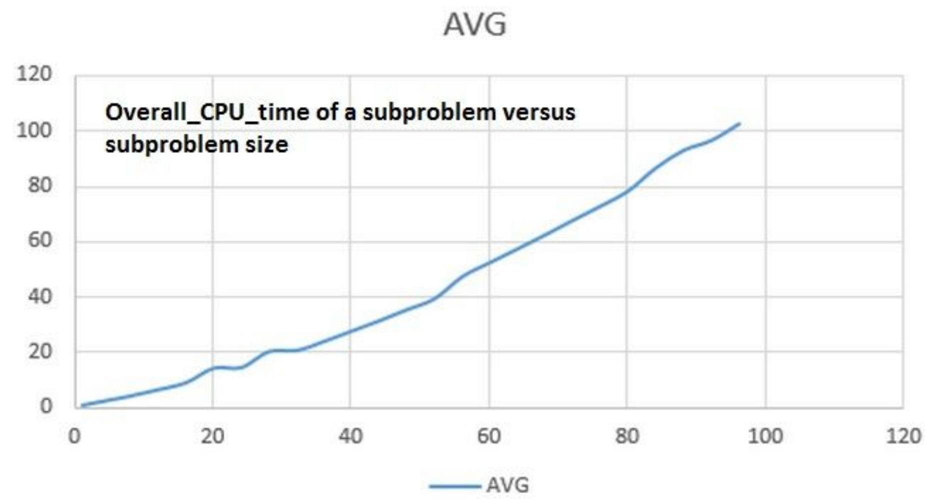

Figure 19. Average plot of *P(x)* using GLPK with scaled-down-SWITCH LP

Since all 20 growth functions were non-convex, we had to once again only rely on the DP approach for the optimal aggregation strategy. Table 3 shows the results of the DP approach with GLPK, and compares results of three aggregation strategies (the optimum strategies generated by DP, those found so far by experiments, and the trivial boundary case aggregation strategy). In Table 3, an aggregation strategy depicted by $\{y_1, y_2, \ldots y_i, \ldots, y_k\}$ denotes there should be k subproblems, with the i^{th} subproblem having y_i blocks.

Table 3. Comparison of overall-cpu-timings of optimal strategy with trivial strategy, using GLPK

Storage rate @Storage cost	Theoretical optimum (by DP algorithm)	Experimental optimum (so far)	Best trivial boundary strategy
10000 @1000	24 subproblems each with 4 blocks 39.75 secs	Matches theoretical	No aggregation 43.05 secs
10000 @100	4 subproblems each with 4 blocks, and remaining subproblems each with 5 blocks 41.1 secs	Matches theoretical	No aggregation 51.63 secs
10000 @10	32 subproblems each with 3 blocks 38.83 secs	Matches theoretical	No aggregation 47.36 secs
10000	24 subproblems each	4 subproblems each with	No aggregation

@1	with 4 blocks 41.21 secs	4 blocks, and remaining subproblems each with 5 blocks 41.1 secs	44.83 secs
5000 @1000	24 subproblems each with 4 blocks 40.39 secs	32 subproblems each with 3 blocks 38.66 secs	No aggregation 46.56 secs
5000 @100	24 subproblems each with 4 blocks 41.26 secs	Matches theoretical	No aggregation 46.56 secs
5000 @10	24 subproblems each with 4 blocks 41.37 secs	32 subproblems each with 3 blocks 39.98 secs	No aggregation 44.51 secs
5000 @1	24 subproblems each with 4 blocks 40.74 secs	4 subproblems each with 4 blocks, and remaining subproblems each with 5 blocks 40.24 secs	No aggregation 45.09 secs
1000 @1000	24 subproblems each with 4 blocks 39.88 secs	Matches theoretical	No aggregation 50.85 secs
1000 @100	24 subproblems each with 4 blocks 41.86 secs	4 subproblems each with 4 blocks, and remaining subproblems each with 5 blocks 39.71 secs	No aggregation 45.37 secs
1000 @10	24 subproblems each with 4 blocks 40.34 secs	4 subproblems each with 4 blocks, and remaining subproblems each with 5 blocks 40.5 secs	No aggregation 45.39 secs
1000 @1	24 subproblems each with 4 blocks 41.03 secs	Matches theoretical	No aggregation 46.65 secs
500 @1000	12 subproblems each with 9 blocks 43.04 secs	32 subproblems each with 3 blocks 38.64 secs	No aggregation 45.68 secs
500 @100	1 subproblem with 5 blocks, and remaining subproblems each with 7 blocks	4 subproblems each with 4 blocks, and remaining subproblems each with 5 blocks	No aggregation 45.43 secs

	43.05 secs	38.76 secs	
500 @10	24 subproblems each with 4 blocks 42.37 secs	32 subproblems each with 3 blocks 39.38 secs	No aggregation 45.49 secs
500 @1	24 subproblems each with 4 blocks 40.44 secs	4 subproblems each with 4 blocks, and remaining subproblems each with 5 blocks 40.19 secs	No aggregation 45.64 secs
100 @1000	24 subproblems each with 4 blocks 39.85 secs	32 subproblems each with 3 blocks 38.4 secs	No aggregation 45.11 secs
100 @100	16 subproblems each with 6 blocks 42.57 secs	4 subproblems each with 4 blocks, and remaining subproblems each with 5 blocks 38.72 secs	No aggregation 45.44 secs
100 @10	24 subproblems each with 4 blocks 41.38 secs	4 subproblems each with 4 blocks, and remaining subproblems each with 5 blocks 39.21 secs	No aggregation 45.88 secs
100 @1	2 subproblems each with 6 blocks, and remaining subproblems each with 7 blocks 42.62 secs	4 subproblems each with 4 blocks, and remaining subproblems each with 5 blocks 39.15 secs	No aggregation 46.44 secs

With GLPK, the average percentage saving of time by employing DP algorithm, instead of the best trivial boundary strategy, over all 20 cases is equal to (922.96 - 823.08) / 922.96 = 10.8%.

3.8 Elimination of noise

3.8.1 Elimination of noise on the REDR machine

We ensured that our solution-time observations were not much affected by noise (example: OS processes, or other users simultaneously using the REDR machine), by using two rules:

1) We plotted the difference between overall-cpu-time and wall-clock-time, by using the fact that not more than one core was employed at any instant of time. Experimental observations were discarded if the difference between these two timing measurements was found to be outside [+1, -1] seconds. Figures 20

and 21 show the noise for the observations we accepted while doing the sanity-tests on REDR for scaled-down-SWITCH using CPLEX and GLPK respectively, which were found to be mostly periodic and within +0.6 and -0.6. This rule eliminated cpu-interference by foreign processes.

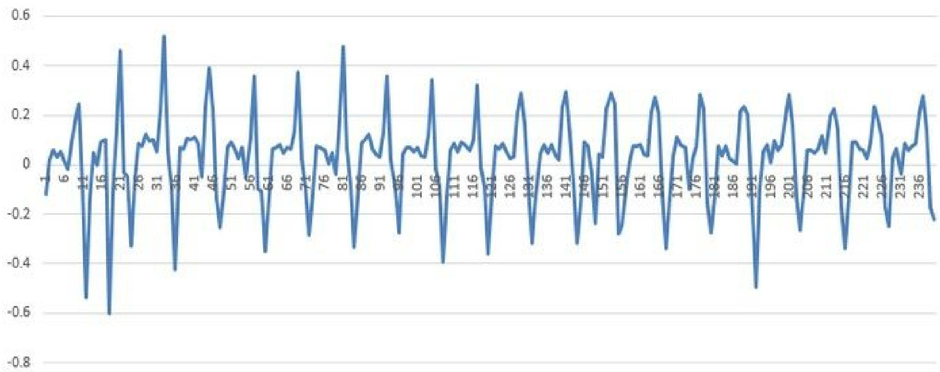

Figure 20. Plot of the difference between overall-cpu-time and wall-clock-time, for CPLEX, on REDR

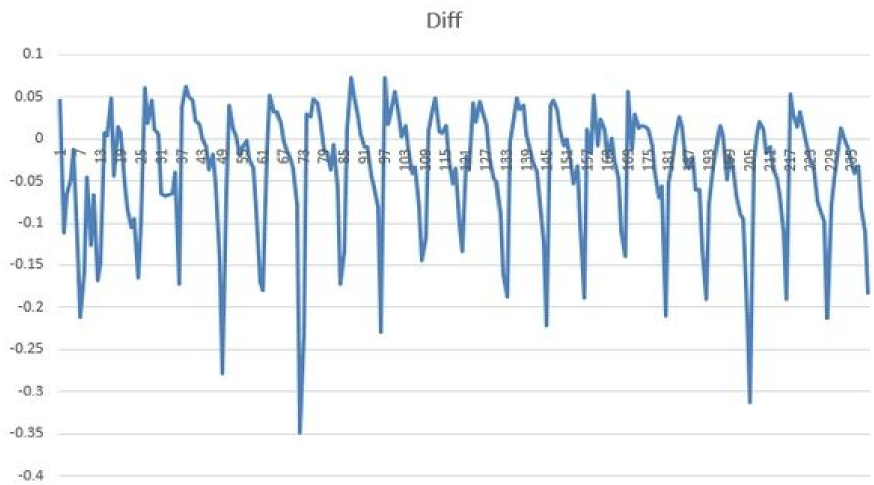

Figure 21. Plot of the difference between overall-cpu-time and wall-clock-time, for GLPK, on REDR

2) The first rule above eliminated cpu-interferences from foreign processes, but does not eliminate the possibility of shared-cache interference. So we repeated our experiments five times (when no other user process was seen in the “top” command), during different times during the week, for each of the 20 cases of storage-cost and storage-rate, with CPLEX and GLPK. With the randomly-generated-synthetic LP, our experiments were repeated ten times with CPLEX and GLPK. In each of our repetitions, the observations (that were accepted based on the first rule) confirmed that the non-trivial optimal aggregation strategies were better than the two trivial boundary cases of aggregation (i.e. full-aggregation or no-aggregation) by at least more than one second (i.e. safely greater than the maximum observed difference between overall-cpu-time and wall-clock-time).

3.8.2 Elimination of noise on the Windows Laptop

It was easier to eliminate noise on the Windows Laptop, since it was a personal dedicated machine. Though wall-clock time was the only metric that could be obtained here, we ensured it was accurate by repetition of experiments at least five times, by removing the internet connection, and disabling unwanted scheduled processes (such as anti-virus scanning software).

3.9 Discussion on the benefits of our optimal aggregation approach

1) While the best trivial boundary case aggregation strategy for CPLEX is full-aggregation, the best trivial strategy for GLPK is no aggregation. Also, the best trivial strategy of GLPK tends to beat the best trivial strategy of CPLEX. This shows that GLPK is good at handling small-sized blocks.
2) The worst trivial strategy of CPLEX (no-aggregation) tends to beat the worst trivial strategy of GLPK (full-aggregation). This shows that GLPK is probably bad at handling large sized LPs.
3) The benefit (in terms of percentage saving of overall-cpu-time) of using DP with GLPK is ten percent over the best trivial boundary case of no-aggregation. In the case of CPLEX, the benefit is six percent over the best trivial boundary case of full-aggregation. Since the time spent in executing the DP algorithm is negligible, and assuming that we are already given *P(si)*, we conclude that it is worth using DP to gain substantial benefits in solution-time in time-critical LP applications.
4) The strategy produced by DP is good but not optimal in many cases as shown by the experimental results in Tables 2 and 3. There are two main reasons for this. Firstly, we have relaxed the definition of *P(si)* as mentioned in Section 3.6. Secondly, even if the definition of *P(si)* was not relaxed, the DP produces the best aggregation strategy optimized for average-case, which may not be the best for the current LP. Thirdly, the optimal aggregation strategy produced by DP only gives the optimal set of subproblem sizes, not an optimal assignment strategy of blocks to subproblems.
5) Comparing Figures 16 and 18, the growth-functions of CPLEX tends to have more hiccups (i.e. minor ups-and-downs) than the smooth growth-functions of GLPK. We concluded that these hiccups were not due to noise from foreign processes using the procedure mentioned in Section 3.8. On the other hand, since the plot of the average-growth-function and each individual 20 plots of the growth-function using CPLEX had most of these hiccups in the same places, we believe that these hiccups were probably due to cache-misses resulting from the algorithmic strategy of CPLEX.
6) If we compare only the user-times of GLPK and CPLEX on REDR, the winner is CPLEX.
7) When the blocks are very small and simple, both CPLEX and GLPK tend to show that it is better to aggregate all blocks into a single subproblem (i.e. the trivial full-aggregation strategy is favored). This is probably because all the blocks of the LP fit inside the cache, making a full-aggregation strategy optimal.

On the other hand, when blocks are large and complex, a full-aggregation strategy might result in more cache misses, and hence more CPU time.

Chapter 4 deals with more possible reasons why trivial aggregation strategies are not always optimal in general.

CHAPTER 4

POSSIBLE REASONS WHY TRIVIAL AGGREGATION STRATEGIES ARE NOT ALWAYS OPTIMAL

From a mathematical perspective, we understood from Chapter 2, that it is the shape of *P(x)* that decides the optimum aggregation strategy, and that a trivial boundary-case aggregation strategy (i.e. full-aggregation or no-aggregation) is not necessarily always optimal. Experiments showed that for both CPLEX and GLPK, when the blocks were large and complex, non-trivial aggregation strategies provided by DP were usually optimal. Experiments also showed that for both CPLEX and GLPK, when the block sizes were very small and simple, the optimal strategy is the trivial case of full-aggregation.

Some important factors contributing to the shape of *P(x)* are the hardware and software present in the computing-platform, study of which are beyond the scope of this research. Still we can try to list some possible reasons, which we mention below:

1) Multiple caches with different sizes and speeds: If the CPU had unlimited amounts of cache, then probably full-aggregation of all blocks into a single subproblem, will yield lesser solution-times. But when the cache size is limited, then we have two cases. The first case is when all blocks are small and simple, which allows all blocks of the LP to fit inside the cache, making a full-aggregation strategy optimal (which is confirmed by experiments). The second case is when blocks are large and complex, making a full aggregation strategy result in more cache misses, and hence more CPU time. In this second case, a trivial strategy of aggregation (or heuristic), might be to choose the subproblem sizes as close as possible to (and just below) the size of the cache. However, the decision becomes more complicated when there are multiple layers of cache of different sizes and different speeds. In addition to that, the lesser the aggregation, the longer it takes for transferring the entire set of objects among the memory units present in the computing-platform, as a result of which a no-aggregation strategy might cause more delays (more CPU cycles and more wall-clock-time) when the blocks are of medium size. Figure 22 is a logical diagram showing the different types of memory-units in the computing-platform, their sizes and speeds [29]. These complicating factors require a DP approach to compute an optimal solution to our problem of optimally aggregating blocks into subproblems.
2) RAM-size: If the combined size of all the blocks exceeds the size of the RAM, then a full aggregation strategy might greatly increase both CPU time (due to thrashing) and wall-clock time since accessing virtual memory on hard-disk is thousands of times slower than accessing RAM.
3) Solver trying to identify BDS: Even if the cache-size is unlimited, the solver might still try to first identify the diagonal blocks (we are not sure whether all solvers will try to do this), which could increase CPU-time. Identifying the diagonal blocks in a LP of size *n*, requires effort above O(*ns*) where all blocks are dense and

s is the size of the biggest block.

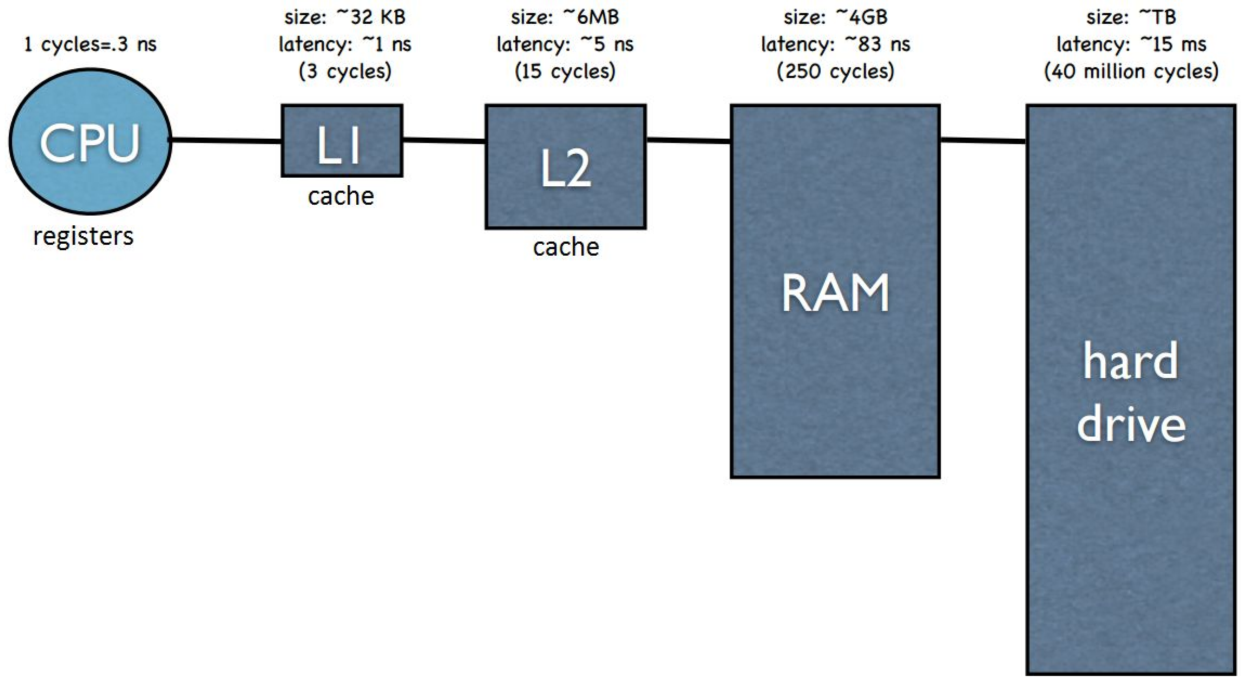

Figure 22. Logical diagram showing speeds and sizes of memory-units

To verify the extent to which cache-misses can affect the shape of *P(x)*, we used the *perf-stat* tool of Linux to plot the cache-misses (of different cache layers) of a subproblem using CPLEX with SWITCH (storage-rate of 5000 and storage-cost of 10), for different sizes of that subproblem. We observed that:

1) The cache-misses of the LLC (Last-Level-Cache) had a sudden spike, when the subproblem-size reached a range of 50-60 blocks (refer Figure 23), which is probably the reason why *P(x)* too had a sudden upward shift in this range (refer Figure 16).
2) The hiccups (i.e. minor ups-and-downs) were prevalent for all values of subproblem-sizes (refer Figure 23), which is probably the reason why *P(x)* too had hiccups throughout its range (refer Figure 16).

The cache-misses in the LLC cache layer tend to have as a more profound impact on *P(x)* compared to cache-misses in the L1 cache layer, since LLC cache-misses mean that the CPU is starting to access data from RAM, which is much slower than cache. The total cache-misses (bottom right plot in Figure 23) have spikes in the same range where the LLC had spikes in cache-misses.

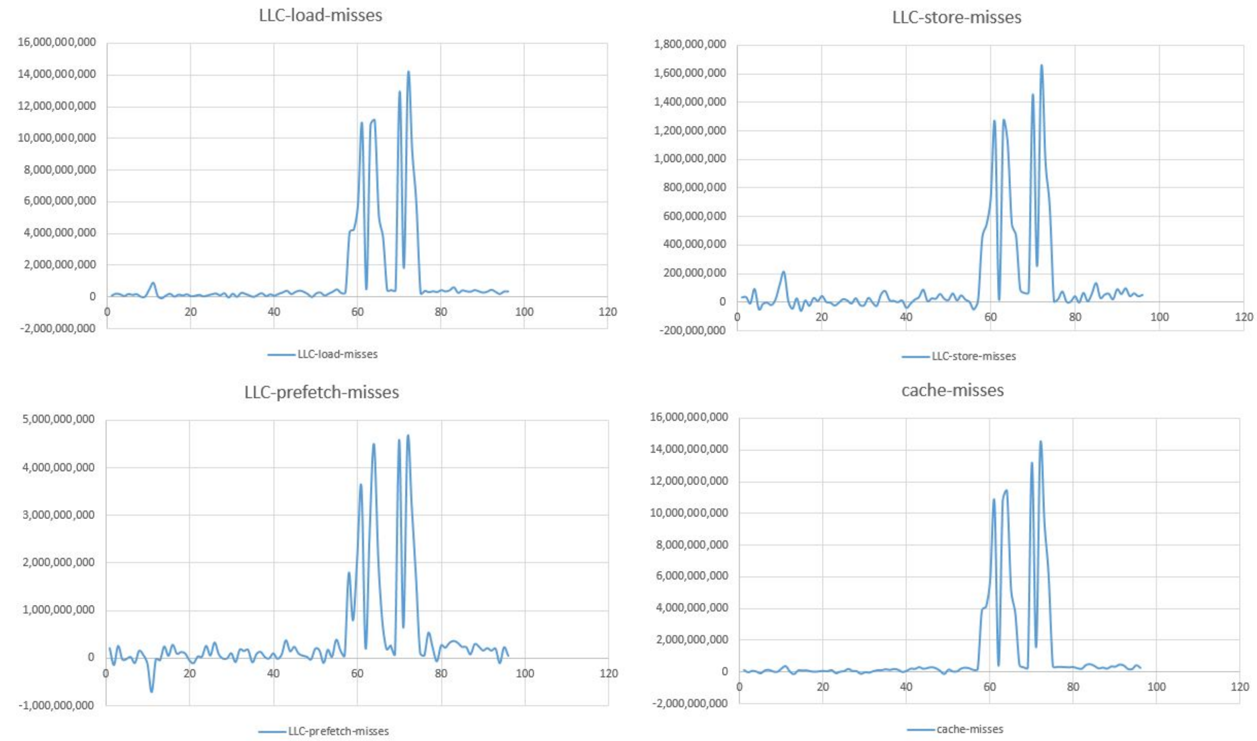

Figure 23. Plot of cache-misses of a subproblem, versus subproblem-size, using CPLEX

Now that we have a better understanding of the factors affecting the shape of *P(x)*, and that cache-misses play a significant role in this, we proceed to discuss important limitations of our approach in Chapter 5.

CHAPTER 5

LIMITATIONS OF OUR AGGREGATION APPROACH AND FUTURE WORK

Further study is needed on important limitations of our approach, which are as follows:

5.1 Complexity of accurately measuring *P(si)*

Our approach essentially uses the DP algorithm with *P(x)*, to obtain the optimal aggregation strategy that minimizes the metric measured by *P(x)*. If *P(x)* measures the growth of average-case running-time, the DP algorithm gives the optimal aggregation strategy that minimizes average-case running-time. If *P(x)* measures the growth of worst-case running-time, the DP algorithm gives the optimal aggregation strategy that minimizes worst-case running-time. The accuracy of our approach is limited by the accuracy of *P(x)*.

In our experiments, we approximated *P(si)* as the average-case solution-time for a subproblem with *i* blocks from the given LP itself. We further approximate the calculation of *P(si)*, as described in Section 3.6. A more accurate approach would be to assign *P(si)* the value of the average of solution-times of all combinations of *i* blocks from the LP. So if the LP has (*n/s*) blocks, there would be (((*n/s*)!) / (*i* !)) combinations for every value of *i*, which grows exponentially. Even though this might be a one-time effort for a given computing-platform for a given range of LP coefficients, the effort is huge.

5.2 Difficulty of interpolating and extrapolating *P(si)*

Our experimentation so far, uses a naive inefficient approach of plotting *P(si)* by considering all integers *i* from 1 to (*n/s*). Though naive, plotting for all values of *i* seems necessary since sharp bends in the plots of *P(si)* have been observed in experiments, probably due to cache size being or not being an integral multiple of block size.

The complexity for plotting *P(si)* needs to be lowered, by using fewer samples of *i*. One might wish to explore how to use a neural-network approach to learn this over time. Once learning has progressed to a substantial extent, accurate interpolation and extrapolation of *P(si)* may be done using fewer sampled values of *i*.

5.3 Difficulty of updating *P(si)* at every iteration of a decomposition technique

In our experiments, we have focussed on only a single iteration of any decomposition technique (like Dantzig-Wolfe, Benders, Lagrangian, etc), wherein by fixing the common variables or by relaxing the common constraints, the LP becomes separable into blocks. As iterations progress in any decomposition technique, the values of the common variables or common multipliers for constraints, keep changing, which might cause the coefficients of the blocks to shift into a domain not covered by the current *P(si)*. At this point, one has three options, before the next iteration of the decomposition technique. First, use the original *P(si)* at the risk of sub-optimal aggregation strategies. Second, take the inefficient approach of once again plotting *P(si)*. Third, take an efficient approach of updating certain coefficients of the polynomial *P(si)* using a heuristic.

5.4 Complexity of the dynamic programming algorithm

The complexity of the DP algorithm grows proportionally to the square of the number of blocks in the LP. The experiments so far suggest that the DP algorithm takes negligible time to execute, which may not be the case when the number of blocks is large. A convex approximation could be used to *P(si)*, enabling the use of the O(1) time algorithm suggested in Theorem 6.

CHAPTER 6

CONCLUSION

We proposed aggregation of blocks into subproblems, as a strategy to minimize average-case solution-time of an LP with BDS, on a computing-platform with a limited number of IPUs (cores in this case, since we consider a single machine). Assuming that the blocks can be equally divided among the cores of the machine, the focus of our work was optimal aggregation on a single IPU.

We experimentally verified that with a single core, there exist LP instances with BDS, for which there are non-trivial aggregation strategies that give lower solution times than the two trivial boundary cases of aggregation (i.e. full-aggregation or no-aggregation). These experiments showed that it is worth developing theories for optimal aggregation.

We also showed that though optimal aggregation is NP-hard in general, blocks of equal size can be optimally aggregated in polynomial-time. The DP algorithm produces the optimal aggregation strategy using a polynomial-growth-function $P(x)$ that is specific for the computing platform. $P(x)$ is the average-case solution-time (over all coefficients whose magnitudes are within some positive integer) for an LP of size x on that computing-platform. In our experiments implementing the DP approach with the LP with blocks of equal size s, we relaxed the definition of $P(si)$ to become an approximation of the average-case solution-time of i blocks each of size s, from that LP alone, on the given computing-platform.

Experiments with LP solvers show substantial improvements of overall-cpu-time of all subproblems, by using our optimal aggregation strategy. Since the DP algorithm takes very little time to execute and since plotting of $P(si)$ is a one-time activity, we conclude that our aggregation strategy is worth using, in preference to trivial boundary cases of aggregation.

We then discussed possible reasons why trivial aggregation strategies (i.e. full-aggregation or no-aggregation) do not always give optimal solution-times on a machine with a limited number of IPUs, though the exact reasons are beyond the scope of this research, since our aim is to only obtain $P(x)$ by treating the computing-platform as a black-box. We finally gave some important limitations of our aggregation approach, and ways to obviate these limitations, which need further study.

Our results will be useful to researchers with limited budgets, and to applications in resource constrained computing environments, where it becomes necessary to use hardware machines with limited amounts of cache and limited number of cores. Our results will also be useful in HPC systems solving large LPs, where the ratio of the number of blocks to the number of available cores, is high.

REFERENCES

[1] C. Lim, *Relationship among Benders, Dantzig-Wolfe, and Lagrangian Optimization*, Wiley Encyclopedia of Operations Research and Management Science, Feb 2011.

[2] J. Rios and K. Ross, *Massively Parallel Dantzig-Wolfe Decomposition Applied to Traffic Flow Scheduling*, Journal of Aerospace Computing, Information and Communication, Vol 7, pp 32-45, Jan 2010.

[3] J. Rioss, *Algorithm 928: A General, Parallel Implementation of Dantzig–Wolfe Decomposition*, ACM Transactions on Mathematical Software, Vol 39, No 3, Article 21, Apr 2013.

[4] M.E. Lubbecke, *Column Generation*, Wiley Encyclopedia of Operations Research and Management Science, Jul 2010.

[5] A. Pinar et al, *Decomposing linear programs for parallel solution*, Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science Lecture Notes in Computer Science, Springer Publishers, Vol 1041, pp 473-482, 1996.

[6] J. R. Tebboth, *A Computational Study of Dantzig-Wolfe Decomposition*, PhD Thesis, University of Buckingham, Dec 2001.

[7] J.M. Latorre et al, *Analysis of the Performance of Benders-Based Distributed Decomposition Methods for Linear Stochastic Programming*, Internal Document Ref No: IIT-12-016A, Institute for Research in Technology, Feb 2012.

[8] R.L. Weil and P.C. Kettler, *Rearranging Matrices to Block Angular Form for Decomposition and Other Algorithms*, Management Science, Vol 18, No 1, pp 98-108, Sep 1971.

[9] C. Aykanat et al, *Permuting Sparse Rectangular Matrices into Block-Diagonal Form*, SIAM Journal on Scientific Computing, Vol 25, Issue 6, pp 1860-1879, 2004.

[10] M. Fischetti et al, *A Note on the Selection of Benders Cuts*, Mathematical Programming, Vol 124, Issue 1-2, pp 175-182, Jul 2010.

[11] T. L. Magnanti and R. T. Wong, *Accelerating Benders Decomposition: Algorithmic Enhancement and Model Selection Criteria*, Operations Research, Vol 29, No 3, pp 464-484, Jun 1981.

[12] J. N. Sawaya and S. Elhedhli, *An interior-point Benders based branch-and-cut algorithm for mixed integer programs*, Annals of Operations Research, Vol 210, Issue 1, pp 33-55, Nov 2013.

[13] F. Vanderbeck, M.W.P. Savelsbergh, *A generic view of Dantzig-Wolfe decomposition in mixed integer programming*, Operations Research Letters, Vol 34, Issue 3, pp 296-306, May 2006.

[14] R. Entriken, *A Parallel Decomposition Algorithm for Staircase Linear Programs*, Report No ORNL/TM-11011, Oak Ridge National Lab, Dec 1988.

[15] M.D. Jayakumar and R.V. Ramasesh, *A Solution-Cascading Approach to the Decomposition of Staircase Linear Programs*, The Journal of the Operational Research Society, Vol 45, No 3, pp 301-308, Mar 1994.

[16] R. Kannan, *A Polynomial Algorithm for the two variable Integer Programming Problem*, Journal of the

Association for Computing Machinery, Vol 27, No 1, pp 118-122, Jan 1980.

[17] H. W. Lenstra, *Integer Programming with a fixed number of Variables*, Mathematics of Operations Research, Vol 8, No 4, pp 538-548, Nov 1983.

[18] M. Fripp, *Switch: A Planning Tool for Power Systems with Large Shares of Intermittent Renewable Energy*, Environmental Science and Technology, American Chemical Society, Apr 2012.

[19] F.A.Potra, *Interior Point Methods, Twenty Years Later*, http://www.math.umbc.edu/~potra/talk0930.pdf, Sep 2003.

[20] Basu, Pollack and Roy, *Algorithms in Real Algebraic Geometry*, Second Edition, ISBN 3-540-33098-4.

[21] J.J. Sylvester, *Mathematical questions with their solutions*, Educational Times 41, 171–178, 1884.

[22] R. Andonov et al, *Unbounded knapsack problem: Dynamic programming revisited*, European Journal of Operational Research, Vol 123, Issue 2, pp 394–407, Jun 2000.

[23] T.G. Crainic et al, *Scenario Clustering in a Progressive-Hedging-based Metaheuristic for Stochastic Network Design*, CIRRELT-2012-41, Aug 2012.

[24] T.G. Crainic et al, *Scenario Grouping in a Progressive-Hedging-based Metaheuristic for Stochastic Network Design*, CIRRELT-2013-52, Aug 2013.

[25] P.L. Carpentier et al, *Optimal Scenario Set Partitioning for Multistage Stochastic Programming with the Progressive Hedging Algorithm*, CIRRELT-2013-55, Sep 2013.

[26] Online Documentation of Python, *https://docs.python.org/2/library/os.html*

[27] T. Fukaya et al, *A dynamic programming approach to optimizing the blocking strategy for the Householder QR decomposition*, pp 402-410, IEEE International Conference on Cluster Computing, 2008.

[28] C. Guo et al, *Optimal caching algorithm based on dynamic-programming*, Proceedings of SPIE 4519, Internet Multimedia Management Systems II, 285, 2001.

[29] http://www.bowdoin.edu/~ltoma/teaching/cs345/spring11/Handouts/memory.pdf