ParallelLCA: A foreground aware parallel calculator for Life Cycle Assessment

by

François SAAB

THESIS PRESENTED TO ÉCOLE DE TECHNOLOGIE SUPÉRIEURE IN
PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR A
MASTER'S DEGREE WITH THESIS IN SOFTWARE ENGINEERING
M.Sc.A.

MONTREAL, OCTOBER 25, 2019

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

BOARD OF EXAMINERS (THESIS M.Sc.A.)

THIS THESIS HAS BEEN EVALUATED

BY THE FOLLOWING BOARD OF EXAMINERS

Professor, Alain April, Thesis Supervisor
Department of Software Engineering and IT at École de technologie supérieure

Professor Conrad Boton, President of the jury
Department of Construction Engineering at École de technologie supérieure

Professor Mohamed Faten Zhani, Member of the jury
Department of Software Engineering and IT at École de technologie supérieure

THIS THESIS WAS PRESENTED AND DEFENDED

IN THE PRESENCE OF A BOARD OF EXAMINERS AND THE PUBLIC

OCTOBER 17, 2019

AT ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

ACKNOWLEDGMENTS

I would like first to thank Professor Alain April for his guidance concerning the academic writing of this thesis.

I also want to express my gratitude for the generous scholarship from the École Polytechnique de Montréal, which made it possible for me to pursue the Master's program.

Finally, I would like to thank the library staff at École de technologie supérieure for providing me with all the necessary advice and literature resources.

ParallelLCA: CALCULATEUR PARALLÈLE D'ANALYSE DE CYCLE DE VIE PRENANT COMPTE DE L'AVANT PLAN

François SAAB

RÉSUMÉ

L'analyse du cycle de vie (ACV), qui vise à évaluer les impacts environnementaux au cours du cycle de vie d'un produit (par exemple, la production d'aluminium au Québec) ou d'un système de produits, peut être utilisée pour comparer différents systèmes utilisant différents types de matériaux afin de déterminer celui qui est le moins dommageable pour l'environnement. Le calcul d'ACVreprésente un défi de calcul car il dépend de la taille du système de produit, du nombre d'itérations dans la simulation de Monte-Carlo, et du nombre de variables incertaines dans le système. Tout d'abord, la résolution d'un système linéaire, de dimensions dans l'ordre de 10 000 équations par 10 000 variables inconnues pour le cas de base. Deuxièmement, la construction d'un arbre, de nature itérative, avec des dimensions minimales de 10 000 nœuds. Troisièmement, la simulation de Monte-Carlo nécessitant plusieurs milliers d'itérations pour converger. Finalement, une analyse de sensibilité, qui nécessite le calcul des millions de corrélations vecteur-vecteur, dans laquelle chaque vecteur a une dimension qui est équivalente aux nombres des itérations qui sont effectués lors de la computation de Monte-Carlo.

Pour résoudre au mieux les défis informatiques présents dans l'ACV, la recherche bénéficie des bibliothèques standards pour la résolution de systèmes linéaires creux et du calcul sur des matrices creuses. En outre, la recherche a adoptée des optimisations mathématiques qui ont par exemple supprimé l'inverse matriciel de l'analyse de contribution qui est très coûteuse, ainsi que des optimisations algorithmiques qui ont pu enlever une grande partie d'analyse du calcul matriciel. De plus, la recherche a expérimenté avec des librairies, qui permettent de paralléliser le calcul, telles que OpenMP, MPI, et Apache Spark.

Dans un premier temps, ce mémoire abordera la littérature de ces opportunités de calcul. Deuxièmement, il présentera un calculateur d'ACVpour mettre en œuvre un calcul efficace. Enfin, il décrira les performances du calcul des différentes phases de l'ACV pour différentes dimensions du système (S) et se terminera par des suggestions d'améliorations et de développement futur.

Mots-clés: Analyse de Cycle de Vie (ACV), solveur hybride ACV, ACV parallèle, ACV distinguant l'avant plan

ParallelLCA: A FOREGROUND AWARE PARALLEL CALCULATOR FOR LIFE CYCLE ASSESSMENT

François SAAB

ABSTRACT

Life Cycle Assessment (LCA), which aims to assess the environmental impacts during the life cycle of a system product (S) (e.g., production of aluminum in Quebec), can be used to compare different systems built with different types of materials to determine which is the least harmful to the environment. The calculation in LCA represents a computational challenge as it is dependent on the size of the system, the number of iterations in the Monte-Carlo simulation, and the number of uncertain variables in the system. First, the solving of a linear system of dimensions in the order of 10,000 equations by 10,000 unknown variables is required for the base case. Second, the building of a graph iterative in nature with minimum dimensions of 10,000 vertices. Third, the computing of a Monte-Carlo simulation requiring several thousands of iterations to converge is to be computed. Finally, a sensitivity analysis which requires the computing of millions of correlations between vectors each having a dimension that is proportional to the number of iterations in the Monte-Carlo simulation.

To best solve the computational challenges present in LCA, this research benefits from well-established libraries that solve large sparse linear systems and performs large sparse matrix computing. Also, this thesis adopted mathematical optimizations that removed the matrix inverse step from the contribution analysis module, which is very expensive, as well as other algorithmic optimizations that removed the large and variant part of the LCA supply-chain from the matrix component of the various calculation phases. Furthermore, this research experimented with libraries such as OpenMP, MPI, and Apache Spark to parallelize the computation.

First, the thesis will discuss the literature regarding these computational opportunities. Second, it will present a proposed LCA calculator for implementing an efficient LCA computation. Finally, it will present the performance of computing the different phases of LCA for various dimensions of the system (S) and concludes with suggestions for improvement and future development.

Keywords: Life cycle assessment (LCA), LCA Hybrid Solver, Parallel LCA, Foreground aware LCA

TABLE OF CONTENTS

XIV

LIST OF TABLES

XVI

LIST OF FIGURES

Page

XX

LIST OF ALGORITHMS

LIST OF ABBREVIATIONS

ARPACK	ARnoldi PACKage
Brightway2	Open Source LCA Software in Python
BLACS	Basic Linear Algebra Communication Sub-programs
BLAS	Basic Linear Algebra Subprogram
CCDF	Complementary Cumulative Density Function
CDF	Cumulative Density Function
CIRAIG	Centre International de Reference sur Le Cycle de Vie des Produits
CMLCA	Scientific software for LCA, IOA, EIOA
COO	Coordinate matrix
COLT	Sparse matrix library developed by CERN
CSC	Compresses Sparse Column matrix
CSR	Compressed Sparse Row matrix
CuSPARSE	Cuda Sparse library
Ecoinvent	Proprietary life cycle inventory database
EcoSpold	Life cycle inventory database format
Eigen++	C++ library for linear algebra
GSA	Global Sensitivity Analysis
GSL	GNU Scientific Library
HPC	High-Performance Computing
LCIA	Life Cycle Impact Assessment
LCA	Life Cycle Assessment

LCI	Life-Cycle Inventory
LU	LU matrix decomposition
MCS	Monte-Carlo Sampling
MD	Multi Dissect matrix ordering
METIS	Serial Graph Partitioning and Fill-Reducing matrix ordering
MLlib	Apache Spark's machine learning library
MTJ	Matrix Toolkit for Java matrix library
MUMPS	Multi-Frontal Un-Symmetric Matrices Parallel Solver
NNZ	Non-Zero entries in a matrix
OpenLCA	Open Source LCA Software in Java
OpenMP	Open Multiprocessing
ParallelColt	Parallel version of Colt a sparse linear algebra library in Java
PDF	Probability Density Function
QR	QR matrix decomposition
RDD	Apache Spark's Resilient Distributed Datasets
Revit	Construction modelling software
RNG	Random Number Generator
Scalanlp	library for natural language processing in Scala
ScaLAPACK	Fortran Library for scalable linear algebra package
SCC	Spearman Correlation Coefficient
SOR	Successive Over-Relaxation
Apache Spark	Library for distributed computing on massive datasets
SQL	Structured Query Language

SRS	Software Requirement Specifications
STDLIB	The C++ standard library
SuperLU	Library for linear algebra
SVD	Singular Value Decomposition
UMFPACK	Un-Symmetric Multifrontal Package

LIST OF SYMBOLS

1D   One Dimensional

2D   Two Dimensional

a_{ij} Cell in matrix A at row i and column j

A   Technology matrix

b_{mn} cell in matrix B at row m and column n

B   Intervention Matrix

g   Impact Score vector

(g)   LCA network or graph

h   Inventory vector

P   Parameters vector

p   parameter in the parameters vector

q_{kl} cell in matrix Q at row k and column l

Q   Characterization Matrix

S   LCA Scalars vector

INTRODUCTION

The use of products (e.g. the use of buildings or cars) exchanges materials with the environment either through extraction from nature (e.g., water and minerals) or through emissions into the air (e.g., CO_2). Products that we use daily are created in factories by a chain of manufacturing processes (e.g., energy production, mining, and transport activities). This chain of processes extracts materials from nature, transforms those materials into complexes, and by doing this, generates pollutants that have various impacts in various areas of the environment (e.g., on human health, on climate change).

LCA (Life Cycle Assessment) is an established framework which aims at assessing the environmental impacts of a given product on the different areas in the environment by quantifying the impact of the life cycle of that product on the environment, highlighting impactful underlying industrial processes on a specific impact area. A series of iterative development phases in LCA has provided with LCA databases where industrial activities are represented as processes producing and consuming products through what is called intermediate exchanges. Also, each process exchanges elementary flows with the environment (e.g., CO_2, water).

➢ Research Context

A standard procedure is usually adopted to assess the impact of using a given product. This procedure can consist of:

1. The building of a graph interconnecting the different LCA processes (i.e., Figure 0.1) based on the information provided in the adopted LCA database. This algorithm is iterative and requires database access in each iteration;
2. The traversing of the created graph, which may be cyclic, and the aggregating of the processes individual impact scores to assess the total impact. Alternatively, this step can be replaced by transforming the network into a system of linear equations which if solved

and scaled, can give the total impact scores of the involved activities. When the Matrix Method is used, additional matrix operations, namely matrix-matrix and matrix-vector multiplications, are necessary to replace the graph aggregation operation. This *Calculation Kernel* is the core of this research project;

3. The inverse of a matrix representing the created LCA graph is another type of expensive matrix operation to consider;

4. A Monte-Carlo simulation that requires several, tens, or hundreds of thousands of iterations to converge, wherein each iteration the aforementioned *Calculation Kernel* is executed;

5. A global sensitivity analysis which involves millions of vector-vector correlations to be computed. The central role of the sensitivity analysis is to identify significant contributors to the uncertainty of the output result.
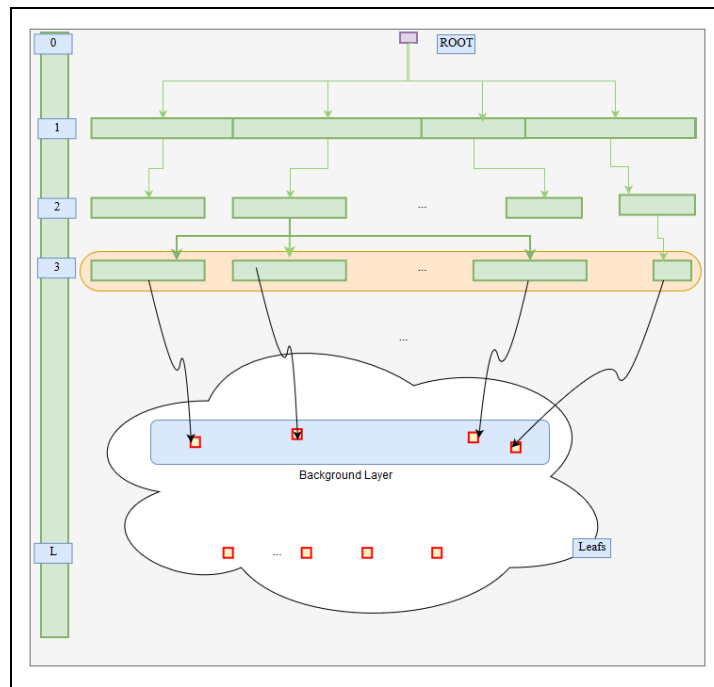
Figure 0.1 LCA Graph

Two kinds of processes interconnections are essential to distinguish in LCA: The *background* layer graph contains processes with cyclic interconnections, and the *foreground* layer (or front layer) graph consists of an acyclic graph of interconnections with mostly no *Elementary*

Flows exchanges. When modelling the lifecycle of "the use of a product" (e.g., phone or building) the different constituents of that product (e.g., screen, keyboard, electronics) are modelled as a hierarchical foreground graph that connects to processes from the background layer.

➢ **Research challenges**

A primary goal in this research is the development of algorithms that provide scientifically correct results (e.g., similar to other standard LCA calculators). In addition to this functional requirement, other non-functional computational challenges make an essential constituent of this research, such as:

1. The efficient instant loading of a database that changes in content and increases in size to up to hundreds of thousands of elements in the foreground layer;
2. The fast computation of the LCA graph which can increase in size. Building the graph (g) is a challenging task because:

    A. The exchanges database can become large enough which makes the queries execution inefficient;
    B. The graph can be very deep (i.e., composed of thousands of layers) and therefore, the corresponding complexity will depend on the number of layers in the graph, the number of activities per layer, and the size of the exchanges database.

3. The implementation of a fast *Calculation Kernel* which consists of the traversal and transformation of the graph (g) into a set of matrices for performing the solving of a linear system (i.e., A x = b) as well as the fast computation of Matrix-Vector product operations which is essential in LCA equations;
4. The fast computation of a matrix inverse with dimensions proportional to the number of processes in the graph (g).

In addition to the aforementioned kernel calculation steps, this research is required to generate reports that provide insights allowing for the interpretation of the foundational LCA results such as:

1. The process-based contribution analysis reports, which allow assessing the contribution of each process to the totals of the inventory and environmental impact results;
2. The process-based upstream contribution analysis reports, which provides the contribution of each activity upstream chain to the total computed impact score vector. This report involves other expensive operations such as the computation of a matrix inverse and the translation of the cyclic graph into an acyclic one;
3. The fast and scalable implementation of a Monte-Carlo simulation consisting of several thousands of iterations. Each of those iterations consists of sampling a large number of objects (i.e., hundreds of thousands), and applying the aforementioned *Calculation Kernel*;
4. The fast and scalable implementation of a global sensitivity analysis (GSA) calculator, which involves the computation of millions of correlations, each involving vectors with dimensions starting from multiples of at least 1,000 elements.

➢ **Methods and thesis plan**

The thesis considered two design decisions for solving these computational challenges. First, it experimented with parallel frameworks that support parallel data processing, parallel task execution, and parallel matrix computing. Second, the thesis adopted a myriad of mathematical and algorithmic optimizations to achieve faster response times.

The thesis presents, in Chapter 1, the literature review and focuses on topics such as computational LCA, parallel computing, and sparse matrix computing. In Chapter 2, the calculator prototype design and implementation decisions are presented. Finally, Chapter 4 describes the different conducted experiments and then concludes with an interpretation of these experiments results.
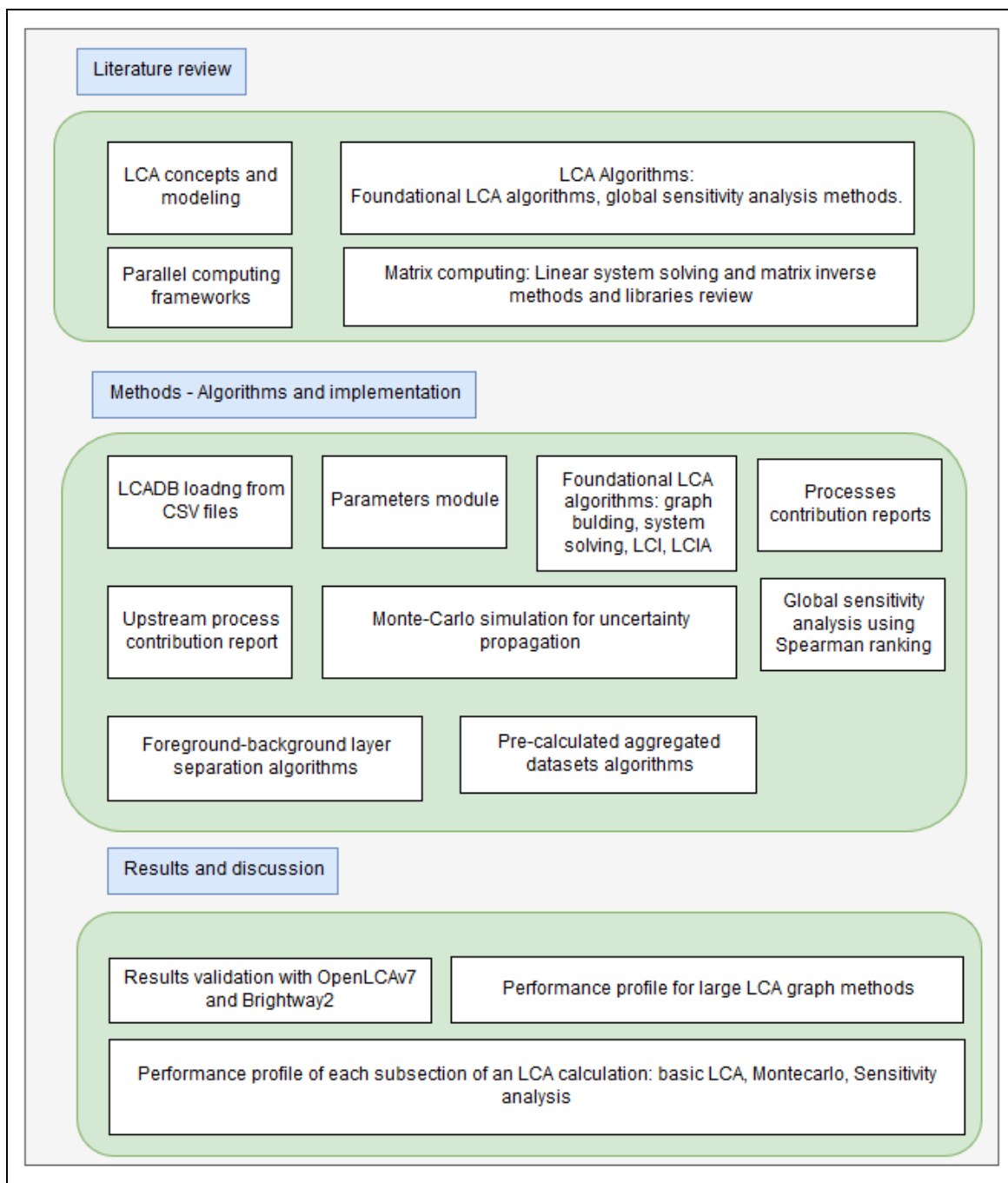
Figure 0.2 Thesis plan

CHAPTER 1

LITERATURE REVIEW

This literature review aims at covering the background behind the different domains of this research project, which includes computational LCA, linear algebra, numerical matrix computing, efficient computing, parallel computing, and parallel computing frameworks. The literature review provides all the necessary background information that is used in the remainder of this thesis with a focus on the practical aspects (i.e., algorithms, programming methods, software frameworks) of the research.

1.1   Life cycle assessment algorithms

1.1.1   Concepts and definitions

LCA (Life Cycle Assessment) is a systemic framework to assess the environmental footprint, from cradle to grave, of the *life cycle* of a product, which includes the raw material acquisition, the product manufacturing, the distribution, the use of the product, and the final disposal. The *lifecycle* of a product can be modelled as a *Network* (or a *Graph*) of connected processes. A *Process*, also called activity, is a step in the life cycle of a product (e.g., material extraction activity during electricity production). Intermediate Flows are the interconnections between the different processes of the lifecycle of a product (e.g., electricity production process connection to the process of crude oil). The interactions between an LCA process and its surrounding *Biosphere* are called *Elementary Flows* (e.g., extracting or emitting substances into nature, like CO_2). An *Inventory* contains the total quantities of *Elementary Flow* exchanges with nature. Impacts on the environment are classified in corresponding *Impact Categories,* based on the concerned damaged area (e.g., water, air, human life), The step of converting the elementary flows inventory into impact categories scores is called *Characterization.* This step uses a list of *Impact Factors* which map each

elementary flow inventory into an impact score in a given impact category. *Impact Methods* represent, in LCA databases, a higher container for *Impact Categories.* Each impact method is responsible for several impact categories (ISO, 2006).

LCA consists of four phases. First, in the scope and definition phase, a *Functional Unit*, which is the definition of the function (or functions in the case of multi-functional LCA) provided by a product life cycle, is established. Second, in the inventory phase or LCI (Life Cycle Inventory), the totals of the different emissions and extractions with the environment (i.e., *Elementary Flows)* are quantified: each elementary flow exchange is assigned a total quantity. Third, in the impact assessment phase or LCIA (Life Cycle Impact Assessment), the inventories (e.g., 1 kg of wood), are characterized into respective impact categories and impact methods using the corresponding *Impact Factors*. Finally, in the interpretation phase, the framework assesses the contribution of each of the activities in the life cycle of a product to the total impacts and the total inventory results. Also, the interpretation phase may assess the uncertainty of the output results through what is called *Uncertainty Propagation analysis*. Following the uncertainty propagation analysis is a sensitivity analysis that can be performed to assess the effect of uncertainty in input parameters on the uncertainty in output results (ISO, 2006).

1.1.2   The Sequential Method for LCA

Jolliet, Soucy, and Houillon (2010) explained that a typical LCA calculation method, also called the *Sequential Method*, follows a series of steps for evaluating the inventory and the environmental impacts of a product lifecycle. It starts by identifying the *Functional Unit* (e.g., production of 1 kg diesel) of the system to analyze. It then identifies the demand-supply interconnections and scales the included processes to meet the requirement of the functional unit. The scaling of the processes scalars consists of propagating the *Functional Unit* from the root process in the downstream direction until reaching the leaf nodes or non-demanding processes.

After scaling the processes, the inventory is calculated as in equation 1.1 and characterized into *Impact Categories* using their *Impact Factor* coefficients as in equation 1.2.

LCI_f , the inventory of an elementary flow f is computed as an aggregation, as given by equation 1.1, over the array of all the LCA processes that exchange that flow with the environment.

$$LCI_f = \sum_{processes} scalar_{process} * Unitary\ quantity_f \qquad (1.1)$$

$LCIA_c$, the impact score of an impact category c can be computed as an aggregation, as given by equation 1.2, over the inventory vector previously obtained from equation 1.1.

$$LCIA_c = \sum_{All\ f,c} LCI_f * Impact\ Factor_{f,c} \qquad (1.2)$$

A significant challenge when using the Sequential Method is the possible presence of feedback loops (i.e., circular connections) in the network interconnecting the different LCA processes of the background layer. Those circular connections will cause the traversal of the graph never to finish. Several solutions to this problem have been proposed, such as:

> "Interrupting a branch after a specified number of loops, interrupting a branch when the last round has added less than a specified amount, replacing process data by corrected process data in which feedback loops have been accounted for, and the use of infinite geometrical progression." (Heijungs and Sun, 2002)

1.1.3 The Matrix Method for LCA

The Matrix Method, first introduced by Heijungs (1994) and further developed in Heijungs and Sun (2002), came to propose a new structure for computational LCA. This new structure

consists of converting the graph traversal and aggregations of equations 1.1 and 1.2 into matrix operations. In the following section, the thesis presents the main components of the Matrix Method and its internal operations.

Matrix **A**, the *Technology Matrix,* contains the unitary quantities of the intermediate exchanges (i.e., rows) associated with the LCA processes (i.e., columns) of a product lifecycle. Matrix **B,** the *Biosphere* Matrix*,* contains the unitary quantities of the elementary flows (i.e., rows) associated with the LCA processes (i.e., columns) of a product lifecycle. Matrix Q, the *Impact Factors*, has in its rows the *Elementary Flows* and in its columns the *Impact Categories*. Vector **f**, the *Demand Vector,* has all its cells set to zeros except one cell set to the functional unit quantity.

Vector **s**, the *Scalars Vector*, results from scaling the intermediate exchanges (i.e., rows in **A**) to produce the desired demand vector **f**. The scalars vector **s** can be computed using equation 1.3.

$$A s = f, s = A^{-1} f \qquad (1.3)$$

Vector **g**, the *inventory* vector*,* contains the *total inventories* of the elementary flows generated or extracted by a given product, and it is computed as in equation 1.4.

$$g = B s = B A^{-1} \qquad (1.4)$$

Vector **h**, the *Impact Scores* vector*,* contains the total impacts of the impact categories belonging to the impact method understudy, and it is computed as in equation 1.5.

$$h = Q g \qquad (1.5)$$

An important variable in matrix-based LCA is the inverse of the Technology Matrix **A**. Su and Heijungs (2007) provided the development of the matrix inverse method in LCA using power series expansion, as shown in equation 1.6.

$$A^{-1} = I + (I - A) + (I - A)^2 + (I - A)^3 + \cdots \qquad (1.6)$$
$$= 1 + Z + Z^2 + Z^3 + \cdots$$

The individual column cells in A^{-1} represent the total output that each unitary process (i.e., matrix cells), in that given column, must produce to satisfy the reference flow of the process associated with the same column in A.

The process-based contribution to the total inventory of producing one unit of a given process P is given by first computing equation 1.7 and then selecting the column, in the resulting matrix, which position is equal to the position of process P in the original matrix A.

$$LCI_{Aggregate} = B\,A^{-1} \qquad (1.7)$$

Similarly, the process-based contribution to the total environmental impact of producing one unit of a given process P is given by first computing equation 1.8 and then selecting the column in the resulting matrix which position is equal to the position of process P in matrix A.

$$LCIA_{Aggregate} = Q\,B\,A^{-1} \qquad (1.8)$$

1.1.4 Uncertainties and Data Quality Indicators (DQI) in LCA databases

We begin our exploration of uncertainties by listing some related definitions:

1. A population is a domain from which one observation is sampled;
2. A Probability Density Function (PDF) is a graphical representation that maps an

observation from the possible domain of observations to a probability value;

3. Uncertainty can be caused by either a Random Variation or a Bias. A Random Variation corresponds to the randomness in the values of a given variable. A Bias is a skewness that was introduced to the observation because of a systemic measurement error.

Table 1.1 Lognormal, normal, uniform, and triangular distributions

$f(x) = \dfrac{1}{x\,\sigma\,\sqrt{2\,\pi}}\; e^{-(\frac{(ln\,x\,-\,\mu)^2}{2\,\sigma^2})}$	$f(x) = \dfrac{1}{x\,\sigma\,\sqrt{2\,\pi}}\; e^{-(\frac{(x\,-\,\mu)^2}{2\,\sigma^2})}$
$f(x) = \dfrac{1}{(B-A)}$	$f(x) = \begin{cases} \dfrac{2(x-A)}{(B-A)(C-A)} & for A \le x \le C \\ \dfrac{2(B-x)}{(B-A)(B-C)} & for C \le x \le B \\ 0 & elsewhere \end{cases}$

Several statistical measures are to be considered when assessing uncertainties. The *arithmetic mean* is the sum of observation values divided by the count of the observations. The error is the deviation of an observation from the mean of its population. The variance is the sum of the squares of the error divided by the population size. The standard deviation is the root square of the variance. The median is the value that splits the population distribution in half. The mode is the most likely occurred observation. A two-sided confidence interval (e.g., 95%) is the central part of the distribution that is obtained by excluding a certain percentage (e.g., 2.5%) from both sides of the distribution.

Current LCA databases use four principal statistical distributions: the uniform, the triangular, the normal, and the lognormal distributions. These distributions are represented in different forms. The mathematical representation will give a formula for the PDF as in Table 1.1. The EcoSpold format provides the following fields for each of the uncertainty types: UncertaintyType, mean value, minValue, maxValue, most likely value, and standardDeviation95. The representation in the EcoSpold format is what is widely used in

current LCA databases.

In addition to the basic uncertainties, current LCA databases such as Ecoinvent provide Data Quality Indicators (DQI) that allows for the fine-tuning of the basic uncertainty to provide a more reliable PDF. Based on the provided DQI, additional uncertainties can be added to the lognormal representation of the initial PDF to provide with a Pedigree transformed PDF.

Table 1.2 Pedigree Matrix, Version 2

Taken from Mutel (2013)

Indicators	Ranks (u_i)				
	1	2	3	4	5
Reliability	1	1.54	1.61	1.69	1.69
Completeness	1	1.03	1.04	1.08	1.08
Temporal Correlation	1	1.03	1.1	1.19	1.29
Geographical Correlation	1	1.04	1.08	1.11	1.11
Further Technological Correlation	1	1.18	1.65	1.08	2.8

The Data Quality Indicators (DQI) are computed based on a ranking system that involves the use of expert domain judges. The Expert judges assess data sources with uncertainties according to five independent characteristics as listed in Table 1.2. Each of the characteristics is ranked, in a system of five quality levels, with a score between one and five. After the DQI scores are attributed based on a given uncertainty source information, a normal uncertainty distribution is computed for each of the DQI characteristics. Each of these distributions has a mean value of zero, and a variance as shown in Table 1.3. The uncertainty of the given source is then computed, as shown in equation 1.9. (Weidema et al., 2013)

$$\sigma^2 = \sigma_b^2 + \sum_i^5 \sigma_i^2$$

(1.9)

Table 1.3 Pedigree Matrix, Variances

Taken from Weidema et al. (2013)

Indicators	Variances (u_i)				
	1	2	3	4	5
Reliability	0	0.0006	0.002	0.008	0.04
Completeness	10	0.0001	0.0006	0.002	0.008
Temporal Correlation	10	0.0002	0.002	0.008	0.04
Geographical Correlation	0	2.5 10^{-5}	0.0001	0.0006	0.002
Further Technological Correlation	0	0.0006	0.008	0.04	0.12

In addition to the pedigree for lognormal distributions, Muller et al. (2016a) provided equations and a procedure to compute the additional variances for distributions other than the lognormal.

1.1.5   Sensitivity Analysis methods for LCA

As explained by Groen, Bokkers, Heijungs, and de Boer (2017), several methods are commonly used for performing sensitivity analysis in LCA. We focus on the exploration of the sampling-based and correlation-based methods as they are part of the functional requirements for this research project.

For the *Correlation-Based* methods, the literature distinguishes two methods, among others: the Pearson Correlation Coefficient and the Spearman rank-order correlation coefficient. The Pearson Correlation Coefficient provides the correlation between two sample vectors p_i and g_j as in equation 1.10 below.

$$r_i = \frac{\sum (p_i - \overline{p})(g_j - \overline{g})}{\sqrt{\left(\sum (p_i - \overline{p})^2 \sum (g_j - \overline{g})^2\right)}} \tag{1.10}$$

An alternative to the Pearson correlation coefficient method is the Spearman rank-order correlation coefficient (SCC) method, which measures the linear dependence between variables p_i and g_j. To compute the ranked correlation coefficient, the variables p_i and g_j are replaced by their respective ranked vectors. Equation 1.11 applied to the ranked vectors gives r_i^{SCC}. The sensitivity index, using this method, would be computed as in equation 1.12.

$$S_j^{SCC} = \left(r_i^{SCC}\right)^2 \tag{1.11}$$

For the *Variance-based methods*, the Sobol method relies on calculating two main indexes, the Sobol Main Effect, and the Sobol Total Effect. The Sobol algorithm in LCA consists of repeating a sampling of three matrices in several iterations followed by the computation of output variable g or h to finally calculate the indexes in equation 1.12 and equation 1.13. The sampling step consists of first generating a matrix P containing the values of all input parameters p_{ij}. A second matrix Q is generated in the same way. Finally, for each column from Q a third matrix R is generated by adopting that column from Q, and all other columns from P. Following this sampling, the output variables g or h are computed for each selection of a column from Q and for several iterations. The Sobol Main Effect S_j^{SME}, equation 1.12, measures the influence, on the output variable g or h, of the event of "fixing the parameter p_{ij} and making all other parameters variant". The Sobol Total Effect S_j^{STE}, equation 1.13, measures the influence on the output variable g of the event of "making all parameters fixed but parameter p_{ij} variable".

$$S_j^{SME} = \frac{var\left(E(g|p_j)\right)}{Var(g)} = \frac{\frac{1}{N}\Sigma_i\, g(Q)\left(g(R^j)_i - g(P)_i\right)}{\frac{1}{N}\left(\Sigma_i\{g(P)_i^2 - (\frac{1}{N}\Sigma_i\, g(P)_i\}\right)^2} \tag{1.12}$$

$$S_j^{STE} = \frac{\frac{1}{2N}\Sigma_i\left(g(p) - g(R^j)_i\right)^2}{\frac{1}{N}\Sigma_i(g(P)_i)^2 - \left(\frac{1}{N}\Sigma_i(g(P)_i\right)^2} \tag{1.13}$$

Other methods that are available are based on the linear regression formulation and First Order-Taylor expansion.

According to the theory of multiple linear regression, the output g can be expressed as in equation 1.14. The coefficients c_0, c_j, and e_j are the intercept, the slope, and the error terms respectively. The sensitivity index of g would be as in equation 1.15.

$$g = c_0 + \sum (c_j \, p_{ij}) + e_j \tag{1.14}$$

$$S_j^{SRC} = \frac{Var(p_j)}{Var(g)} \, c_j^2 \tag{1.15}$$

Finally, for the *Key-issue analysis* methods, a commonly used method is the *First-order Taylor expansion*. Applying this method to LCA will give an expression of the output variable g using Taylor expansion as in equation 1.16. Using Taylor expansion, the sensitivity index would be calculated as in equation 1.17:

$$g_j = g(P_j) = g(\bar{p}_j) + \frac{\partial \, g(\bar{p}_j)}{\partial \, p_j} (p_j - \bar{p}_j) \tag{1.16}$$

$$S_j^{SCC} = \frac{Var(p_j)}{Var(g)} \left(\frac{\partial g}{\partial p_j} \right)^2 \tag{1.17}$$

1.1.6   Uncertainty propagation in LCA

A system can be modelled as a function Y of independent variables ($x_1, x_2, \ldots, x_k$). If there is uncertainty in the independent variables x_i, two main points of concerns are raised. A first question is concerning the amount of uncertainty that is induced in the output variable because of the uncertainty in the input variables. This first question implies another equally

important question, which is how to propagate the uncertainty from input variables to output variables in order to assess the output uncertainty. A second question concerns the contribution of the uncertainty of a given input variable X_i to the total uncertainty of the output variable Y. This section discusses the literature concerning these questions.

1.1.6.1   Analytical uncertainty propagation for LCA

The analytical approach of uncertainty propagation consists of treating the system as a function of input variables and applying differential calculus to propagate the uncertainty. Supposing Z is a function of variables x and y that represent the system; then, as explained in Heijungs and Sun (2002), the variance of Z is calculated as in equation 1.18.

$$V_z \approx \left(\frac{\partial z}{\partial x}\right)^2 . V_x + \left(\frac{\partial z}{\partial y}\right)^2 . V_y + 2 \frac{\partial z}{\partial x} \frac{\partial z}{\partial y} COV\,(x,y) \tag{1.18}$$

In the case where Z is dependent on more than two inputs, and if we ignore the correlation between the input variables, the variance is calculated as in equation 1.19 as a sum of the product of the *Sensitivity Coefficient SC* (equation 1.20) of a given parameter and the variance of that parameter.

$$Y^j = f(X_1, \dots , X_n), \qquad V(Y)_i^j \approx \sum SC_i^{j^2} . V_{input}(X_i) \tag{1.19}$$

$$SC = \frac{\Delta_z}{\Delta_X} \approx \frac{\partial_z}{\partial_X} \tag{1.20}$$

As explained by MacLeod, Fraser, and Mackay (2002), for the particular case of input variables with lognormal distributions, which is very common in the sciences and LCA databases, the propagation of uncertainty can be approximated as in equation 1.21. The quantity GSD^2 is the *Confidence Factor (CF)* characterizing a 95% confidence interval of the

variables x and y. $S_{x,h}$ is the sensitivity of parameter x relative to the output result h and is computed as in equation 1.22.

$$(ln \ GSD_h^2)^2 \ = \sum S_{x,h}^2 \ (\ln GSD_x^2)^2 \tag{1.21}$$

$$S_{x,h} = \frac{\partial h}{\partial x}\frac{x}{h} \tag{1.22}$$

$$K_{x,h} = S_{x,y}^2 \ \frac{\ln GSD_X^2 \ ^2}{\ln GSD_h^2 \ ^2} \tag{1.23}$$

The contribution of the uncertainty in the input variable x (i.e., GSD_X^2) to the uncertainty in the output variable y (i.e., $\ln GSD_Y^2$) is given by equation 1.23.

1.1.6.2   Sampling-based uncertainty propagation methods for LCA

As explained by E. A. Groen, Heijungs, Bokkers and de Boer (2014) and by Peters (2007a), *Monte-Carlo sampling (MCS)*, a widespread sampling method, will first define a compuTable format of the uncertainties in input parameters as PDFs (Probability Density Function). Second, MCS will proceed to the generation of pseudo-random samples for the uncertain cells in A_{ij}, B_{mn}, and Q_{kl} based on their PDF definitions. Third, it will use the generated samples in equations 1.3, 1.4, and 1.5 to propagate the uncertainty from input parameters to the output result.

Algorithm 1.1 Monte-Carlo Sampling.

Based on (Peters, 2007a)

Algorithm: LCA Monte-Carlo - based on (Peters, 2007a)

---

1 **for** *iteration: 1 to N* **do**
2   A = Sample cells of A
3   B = Sample cells of B
4   Q = Sample cells of Q
5   s'= result of solving A's'=f using iterative method
6   h'= B'* s'
7   Q'= Sample impact factors
8   g'= h'* Q'
9   save g'or h'
10 **end**

Latin Hypercube Sampling (LHS) is a variant of the MCS, which employs a stratified sampling approach instead of pseudo-random sampling one. In random sampling, random numbers are picked independently and at random for the different independent variables X_i. As explained in Helton and Davis (2002), LHS will first identify the range of each variable and then divide that range into n LHS intervals with equal probability. Second, an array of random variables is generated with values are computed by randomly sampling one variable from each LHS interval. This process is repeated to select another N values for the independent variable X_{i+1}, and so on. The N values of X_i are paired at random and without replacement with the values of X_{i+1} to create LHS pairs of two variables. This array of pairs will then be combined with the LHS values of a third variable X_{i+2} to form a triplet of LHS values. The process will be repeated for all uncertain independent variables. (Helton and Davis, 2002).

Quasi MCS (QMCS) is another variant of MCS with the difference of not using the default the random number sampling. QMCS uses quasi-random numbers to sample from the distribution functions. As shown in Saltelli et al. (2007), MCS pseudo-number generated samples tend to have clusters and gaps. When using independent variables with *Clusters,* functions that exist in the vicinity of that cluster are overemphasized in statistical analysis. On the other hand, when independent variables have gaps, the function values that are

dependent on these variables will not be sampled. Quasi-random generators produce sequences having the property of near uniformity, where clusters and gaps are eliminated. The process of generating quasi-random numbers consist of generating deterministic sequences of numbers taking into consideration the position of previously sampled points (Tarantola, Becker and Zeitz, 2012).

1.1.7   Aggregated dataset uncertainty propagation for LCA

In a study published by Qin and Suh (2017), unitary process-level exchanges PDFs were sampled. This sampling was repeated 1,000 times for all of the exchanges of each process independently. For each process, 1,000 samples of 1,000 exchanges were selected, which sums up to a million samples per process. This study tries to find the distribution that best fits aggregate LCI at the level of each unitary LCA process. In finding the best fit, the research is trying to find an approximate PDF to represent the unitary processes aggregate LCI. This study found that most of the aggregate LCIs in the Ecoinvent 3.1 database follow a lognormal distribution. Consequently, the study suggested that using lognormal PDF approximations of aggregated LCI, the sampling is only needed on the approximate PDFs and that evaluating the detailed supply-chain of each process can be avoided.

In a reply letter to Qin and Suh (2017) paper, Heijungs, Henriksson and Guinée (2017)) reasoned that in a comparative LCA, uncertainty has to be deduced from samples generated dependently and that using pre-calculated distributions of complete systems generated from samples that are generated independently will lead to a large overestimation when assessing the uncertainty of the final output results.

Qin and Suh (2017) re-visited their study and re-published a related paper. In this revision and according to empirical results, they show that the use of pre-calculated LCIs "leads to a slight underestimation rather than an overestimation." Furthermore, they considered that the error in the GSDs of the LCI arrays generated from either pre-calculated LCIs or full MCS is

negligible and that "in practice, pre-calculated LCIs can be used in understanding the uncertainties of both non-comparative and comparative LCA."

Finally, Lesage et al. (2018) conducted a study comparing the results of using pre-calculated aggregated LCI when using independent and dependent sampling. The research concluded that "independent sampling should not be used for comparative LCA" and that dependently pre-sampled and aggregated LCIs provides quick and correct results. The method of Lesage et al. (2018) consists of first sampling the uncertain cells dependently, second iterating over all the Ecoinvent activities and calculating the quantities g and h. Finally, it saves the results as 2D arrays. The researchers in this paper argue that while this method provides results with a minimal error when compared to full MCS, it also allows performing Monte-Carlo without re-evaluating the Ecoinvent background layer activities, and solving the corresponding linear system for each MCS iteration as it is the case of full MCS shown in algorithm 1.1.

1.2 Parallel computing

Parallel computing provides a set of models, architectures, and frameworks to achieve parallel processing at the data and task levels. This section will present the models and architectures behind parallel computing. In addition, in this section, we will provide a background review of three widely used parallel frameworks: MPI, OpenMP, and Apache Spark. This review will begin by exploring the basic concepts in parallel computing, and then it will switch the focus to more practical programming-oriented aspects which are heavily used in this research project.

1.2.1 Parallel computing models

The ***Fork-Join*** model, implemented in POSIX (Nichols, Buttlar and Farrell, 2013) and OpenMP (Chapman, Jost and Pas, 2008) among others, is a model composed of two phases: *fork* and *join*. In the fork phase, the work is initially divided into smaller tasks that can be

computed independently and therefore executed in parallel. In the join phase, using a synchronization point, results of the parallel tasks are reduced or joined into one result.

The ***Message Passing Model****,* implemented in MPI (Geist et al., 1996) for example, is a model where each processor uses a private memory to store its variables, which cannot be accessed by other processors. A processor can exchange messages with other processors using inter-process communication. The two main primitives for messages exchanging are *Send* and *Receive,* which can be done either synchronously or asynchronously.

The ***Data flow*** model, implemented for instance, in Apache Spark (Zaharia et al., 2016), transforms the program tasks into a graph of dependent and independent tasks. Dependent tasks must be executed sequentially (i.e., one at a time) and independent tasks can take advantage of parallel.

1.2.2   Bulk Synchronous Parallel (BSP) model

Bulk Synchronous Parallel (BSP) is an abstract model that was introduced in (Valiant, 1990) to find a hardware-software bridge for the parallel programs similar to what the Von-Neuman model provided for sequential programs. BSP allows for the design of parallel algorithms by dividing the program to compute in several super-steps (i.e., iterations in an iterative program). Each super-step will follow a protocol defined by BSP. First, in the concurrent computation step, *components* of the computation are allowed to run locally in parallel. Second, in the communication step, processes exchange messages through *routers* and share data using various communication primitives. Third, in the synchronization barrier step, processes cannot proceed before all processes have reached the barrier. The synchronization step can happen for each period L, also called the *Periodicity* Parameter.

1.2.3   Parallelism levels and granularity

There are three primary levels of parallelism. The *Instruction Level Parallelism (ILP)*, is where instructions are executed on a Vector of data in parallel in the CPU registers. Instruction level parallelism must be enabled at the hardware level through the availability of CPU architectures, and code compilers need to generate *explicit* vectorized code that can run on such hardware architectures (e.g., matrix-matrix multiplication). The *Data Level Parallelism (DLP)* is found in applications where a large amount of data is available, the data is split into chunks, and a single processor works only on a single chunk instead of the whole dataset (e.g., massive datasets queries). Finally, *Task Level Parallelism (TLP)* is adopted when different tasks need to be run in parallel by different processors on different streams of data (multi-processing programming using MPI).

Also, parallelism can be applied at different frequencies. In **Fine-grained** parallelism, the data transfers are frequent and occur at the instruction level between the involved processors. In **Mid-grained** parallelism, data transfer is less frequent and occurs at a higher level between the program sub-tasks. In **Coarse-grained** parallelism, data transfers only occur when parallel processors finish their work and want to join the individual results into a final result.

1.2.4   Parallel instructions stream

The execution model of a sequential computer is based on the Von Neumann machine (Von Neumann, 1993) and is characterized by a *single instructions stream* in which one instruction is applied on a single data item. However, in parallel computing, a program at the low-level physical layer consists of one or many instruction streams acting on one or many data streams.

Table 1.4 Flynn model for parallel computers

Taken from Flynn (1972)

Instruction / Data	Single	Multiple
Single	SISD (e.g., uniprocessor or sequential computers)	MISD (e.g. systolic arrays)
Multiple	SIMD (e.g., vector and data-parallel architectures)	MIMD (Multi-processes running each on its own data stream)

Flynn (1972) introduced four architectures for parallelism based on how instructions are being applied on data streams, as shown in Table 1.4. These architectures can be described as follows:

- SISD (Single Instruction Single Data): this type of architecture is widespread in traditional CPUs where each instruction is executed on a single stream of data;

- SIMD (Single Instruction Multiple Data): this type of architecture is what operates vector computers, for example. In this type of computers, multiple processors are executing the same instruction on data items spread across multiple CPUs;

- MISD (Multiple Instruction Single Data): implemented in systolic arrays (Kung & Leiserson, 1978);

- MIMD (Multiple Instruction Multiple Data): in this model of computers, different datasets are spread across multiple CPUs where each executes its own stream of instructions.

1.2.5   Architectures for parallel computing

The literature distinguishes between two types of parallel computer architectures: *Symmetric Shared-memory multiprocessor (SMP)* and *Distributed Memory Architecture* (Nielsen, 2016).

SMP architecture considers all cores as independent computing units sharing the same memory. The various types of shared memory being used influence the performance in this model. The following are the different types of memory types from fastest to slowest: 1) processors register memory 2) L1, L2, L3 cache memories, 3) hard-disk drives, and 4) remote disks on a network.

Distributed Memory Architecture is a model where each process stores its variables in a private memory that is not accessible to other processors. Data can be shared only by exchanging messages between processors, and therefore, the network communication properties such as Latency, Bandwidth, and Topology are the most influencers in this model. Latency is the time to initiate message passing. Bandwidth is the rate of exchanging messages. Topology is how the different processes are interconnected (i.e., star, grid, etc.).

1.2.6   The laws of parallelism

Let T_{seq} denote the time for running a program serially, T_P the time taken to run an equivalent parallel program using P processors, and T_1 the execution time of that parallel version using a single processor. Three useful metrics are essential to measuring the performance of a proposed parallel algorithm: 1) Scalability, 2) Speedup, and 3) Efficiency.

Scalability is the ratio between the computation time when running on P' processors versus the computation time when running on P processors where P'< P. This metric gives insights on how the performance changes when adding one processor at a time to the pool of resources running a given program.

The *speedup* is the ratio between the execution time of scenario "B" consisting of running the same parallel version on a single processor denoted by t_1, and the execution time of scenario "A" consisting of running a given parallel version of a program on P processor denoted by t_p. Speedup increases with the addition of new processing units only when t_p continue in decreasing.

$$\text{Speedup(P)} = \frac{t_{seq}}{t_p} . \text{ Often, we have } \frac{t_{seq}}{t_p} = \frac{t_1}{t_p};$$

$$\text{Efficiency} = \frac{\text{speedup(P)}}{P} = \frac{t_{seq}}{P\ t_p}$$

$$\text{Scalability}(P', P) = \frac{t_{p'}}{t_p}$$

Figure 1.1 Scalability, Efficiency, and Speedup

Efficiency is the ratio of the *Speedup* when using P processors over the number of processors P. This number provides an insight on the efficiency of adding more cores to the simulation. Usually, efficiency drops when adding more parallelism due to the overhead of parallelism caused by threads synchronization or processors communication.

1.2.6.1   Amdahl's Law: Fixed-size speedup

The time to run a program in parallel can be modelled as consisting of two portions, a parallel portion denoted by α_p and sequential portion denoted by s. This model is characterized by $\alpha_p + s = 1$. The speedup for this model, as described by the Amdahl law, is calculated as in equation 1.24 (Amdahl, 1967).

$$Speedup(P) = \frac{1}{s + \frac{\alpha_{Par}}{P}} \tag{1.24}$$

Based on this law, the speedup is upper bounded by α_{seq} (i.e., the non-parallelizable code) as in equation 1.25. Hence, the speedup is highly dependent on the sequential part of the program.

$$\lim_{P \to \infty} Speedup(P) \; = \; \frac{1}{\alpha_{seq}} \qquad\qquad (1.25)$$
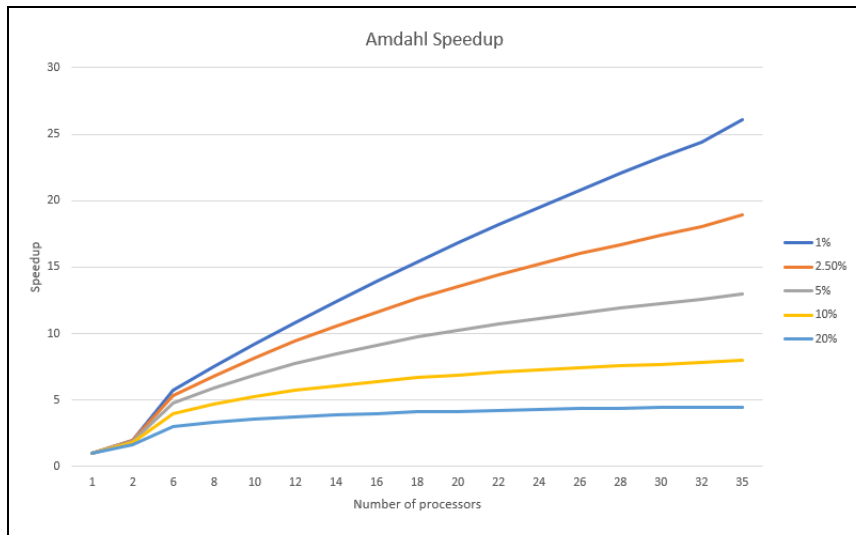
Figure 1.2 Amdahl speedup vs. the number of cores.

Based on equation 1.24

1.2.6.2   Gustafson's Law: Scaled Speedup

Gustafson re-evaluated Amdahl's law by studying the influence of increasing both the number of processors P and the size of data or tasks n being processed by the P processors. This revaluation introduced a new speedup called the scaled speedup, as presented in equation 1.26 (Gustafson, 1988).

$$Speedup_{Ghustafson}(P) = \alpha_{Seq} + P\, \alpha_{Parallel} \qquad\qquad (1.26)$$

The scaled speedup studies practical situations where the software application uses more computing resources when the size of data or tasks increases.

$$\lim_{P \to \infty} Speedup_{Ghustafson}(P) \ = \ P \, \alpha_{Parallel} \qquad (1.27)$$

1.2.7 OpenMP

OpenMP (Dagum and Menon, 1998), is a well-known standard in shared memory parallel programming which promotes code annotations (i.e., compiler directives) that is translated by specialized compilers into parallel code. OpenMP provides the user with compiler directives to configure thread scheduling, to specify parallel and synchronization regions, and to define variables scope (e.g., shared or private).

1.2.7.1 Thread scheduling

OpenMP allows defining *parallel* regions with one of the pre-configured types of scheduling. As explained in Chapman et al. (2008), when the first parallel region is encountered, the master and workers thread is created, and the work is split among the different threads based on the preconfigured type of scheduler. When a parallel region execution is finished, the worker threads switch to *Sleep* mode, and the Master thread remains active. When the next parallel region is encountered, the idle worker's threads are reused instead of created as new threads.

The most common scheduling types can take on the following values:

o **Static**: In this scheduling type, OpenMP divides the work into chunks of size *chunk-size* (i.e., given as a parameter), and schedules the execution of these chunks by the available threads in a circular order;

o **Dynamic**: In this scheduling mode, OpenMP splits the work into chunks of size *chunk-size*, and each thread executes a chunk of the work and then requests additional chunks until the work is done;

o **Guided**: In this scheduling mode, which is similar to the dynamic mode, the chunk size is controlled by the OpenMP framework for better load balancing;

o **Auto**: In the auto-scheduling type, OpenMP delegates the decision of the scheduling mode to the compiler or the runtime system;

o **Runtime**: In the runtime scheduling type, OpenMP defers the decision about the scheduling until runtime.

1.2.7.2 Variables sharing

OpenMP allows for the creation of variables as private to individual threads or as shared among threads. The following is a list of supported variables scopes:

o **private**: A variable is considered as private if it is initialized and owned by one of the OpenMP threads. Each thread can do its modification to its own private variables without the requirement to synchronize with other threads. Private variables cannot be accessed from outside of the OpenMP context;

o **firstprivate**: Similar to private variables, firstprivate variables get their values initialized when copied from the master thread to the inside of an OpenMP context;

o **lastprivate**: Similar to private variables, This type of variables get its value assigned by the last running thread;

o **shared**: shared variables are used as shared storage for the OpenMP team of threads. The OpenMP *CriticalSection* provides exclusive write access in shared variables.

1.2.7.3 OpenMP Execution Constructs

OpenMP provides the following execution constructs:

o "pragma omp parallel": This construct is essential for code to run in parallel. Without this construct, procedures in OpenMP are by default, executed sequentially. When a parallel region is encountered, a team of threads is created by OpenMP runtime where each thread is assigned a unique thread number or thread id. The assigned thread id allows for designing a differed execution plan unique for each thread. At the end of the parallel

region, there is an implied synchronization barrier at which all worker threads will stop running, and only the Master thread will continue executing;

o "#pragma omp for": distribute the for-loop iterations among the available threads;

o "#pragma omp section": distribute independent unit of code, declared by the "section" annotation, among the available threads;

o "pragma omp single": allows configuring sections of the code with single-thread access.

1.2.7.4   OpenMP Synchronization Constructs

OpenMP provides the following synchronization constructs:

o "#pragma omp barrier": At the defined barrier, all threads in the team wait for each other. No proceeding is possible until all threads have reached that point;

o "#pragma omp ordered": allows to execute the configured section in the same order as defined in the for-loop construct;

o "#pragma omp critical": allows to define a section that is executed by only a single thread at a time; all remaining threads are forbidden from accessing this section.

1.2.8   MPI

The Message Passing Interface v2 (MPI) (Geist et al., 1996), is a standard and a programming interface that allows the building of parallel programs requiring the exchange of messages between computer processes.

1.2.8.1   Communication groups

As explained by Nielsen (2016), MPI provides a set of primitives for message communications between different computer processes belonging to specific communication group through various communicators. In MPI, to inter-communicate, computer processes are first added to a *communication group* and then associated with a *rank* in that group.

Communication groups are then connected by *communicators* to allow the communication to take place. *Communication groups* are hierarchical having *MPI_COMM_WORLD* as the top root group, which includes all the processes of an MPI cluster.

MPI provides primitives that allow the retrieval of a computer process's properties inside their communication groups. The **MPI_Comm_size** allows to retrieve the number of processes in a given communication group, and the **MPI_Comm_rank** allows to get a given process rank in that group.

1.2.8.2 Communication primitives

MPI provides four basic communication primitives. *Broadcast* allows for a one to all communication allowing to send a message to all other processes in the same communication group. *Scatter* allows sending a set of messages for each of the processes in the same communication group. *Gather* allows the collecting of a set of messages from each of the processes in the communication group. Finally, *Reduce* allows the aggregation of a set of messages received from the sending processes.
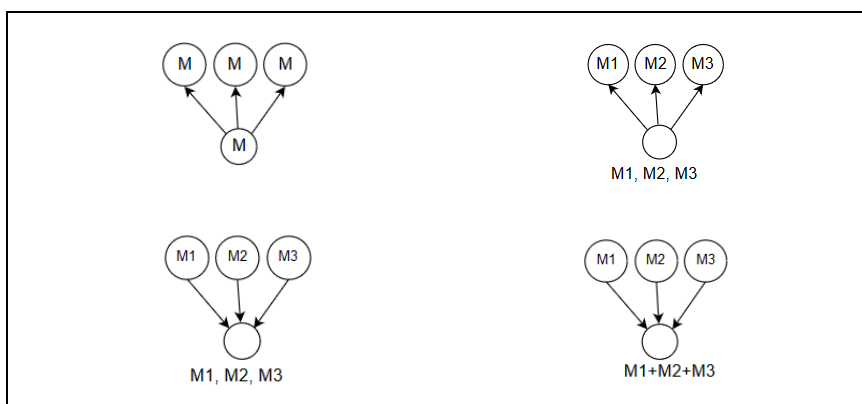
Figure 1.3 MPI Communication primitives

Also, MPI provides the following primitives that allow for inter-processes communication:

o   MPI_SEND: Allows to send an array of elements into a destination process;

- o MPI_RECEIVE: Allows to receive an array of elements from a sending process;
- o MPI_BCAST: Allows the broadcasting of messages from the calling process to the rest processes in the same communication group;
- o MPI_Reduce: Allows the reduction of several data values received by a process into one value using a reduction function;
- o MPI_Scatter: Allows to split an array into chunks and then sends each of the chunks into the processes of the same group;
- o MPI_Gather: Allows for the joining of chunks of data, sent by other processes in the same group, into one array.

1.2.8.3   BOOST::MPI

Boost::MPI (Dawes, Abrahams and Rivera, 1998) is a C++ library, from the Boost family of libraries, that provide access to MPI primitives through OOP functions instead of C/Fortran primitives. BOOST::MPI provides an equivalent function for each of the native MPI primitives (send, receive, reduce, etc.) with the significant ability to use those primitives while operating on custom made C++ class objects instead of only primitives data types (e.g., int, double, array, vector).

The following are the main features BOOST::MPI provides and that this research project is heavily based upon. Using the namespace "*boost*::mpi" the library provides access to both mpi::environment and mpi::communicator. The mpi::communicator class allows access to the process rank through *mpi::communicator::rank()*. Also, the library provides for blocking communication through the functionality of boost::mpi*::send* and boost::mpi*::recv* which support sending and receiving complex C++ datatypes. For non-blocking communications, boost::mpi provides asynchronous sending and receiving using the primitives *boost::mpi::isend* and *boost::mpi::irecv*. The asynchronous primitives allow sending messages in a for loop iteration without waiting for the receiving of the message to happen before passing to the next iteration. The asynchronous communication needs to be synchronized with the primitive *boost::mpi::wait_all*.

```
1   class gps_position
2   {
3   private:
4       friend class boost::serialization::access;
5
6       template<class Archive>
7       void serialize(Archive & ar, const unsigned int version)
8       {
9           ar & degrees;
10          ar & minutes;
11          ar & seconds;
12      }
13
14      int degrees;
15      int minutes;
16      float seconds;
```

Figure 1.4 Example of serialization for BOOST:: MPI.

Taken from Dawes et al. (1998)

Most importantly, "boost::mpi" allows for exchanging complex custom data types by enabling serialization in their class code, as shown in Figure 1.4 (from boost::mpi documentation). A *serialize* procedure needs to be implemented, and that defines how the class attributes will be serialized. This serialization mechanism allows for the serialization of complex data types, including complex nested classes.

1.2.9 Apache Spark

Apache Spark (Zaharia et al., 2016) introduces a new parallel and fault-tolerant framework, based on Resilient Distributed Datasets (RDD), for processing large and massive data sets. Spark is implemented using the Scala programming language and the JVM platform.

1.2.9.1 RDD: Resilient Distributed Datasets

An RDD is a distributed data structure for storing a massive data set of objects in cluster memory. RDD provides a set of transformations that serves as a plan of execution. These transformations are lazy, which means that upon their creation, a plan is created to be executed later on. To materialize the transformations, Spark provides a set of actions on RDDs.

When a transformation is executed, a lineage is created to the previously existing RDD. After a set of transformations are executed, a lineage would have been created connecting all the RDDs from the first to the last current one. When an RDD along that lineage is lost, Spark is able to restore that RDD by recomputing its lineage. Hence the resilient character of RDDs in Apache Spark.

1.2.9.2   Execution model

In Spark, an application program consists of a driver program and executors launched locally or on a Spark cluster. A driver program plays the role of a controller that runs the "*main*" function of the application in which it creates a SparkContext that launches jobs on a cluster of workers. Based on the mode of clustering, a driver can run on a local machine or remotely in the Spark cluster. A worker node hosts the execution of tasks on the Spark cluster with *Spark executors*. An executor is a process launched by an application master on a worker node to run tasks scheduled for execution. Running queries in Spark is implemented by launching *Spark jobs* each time an action (i.e., execution of a transformation) is required to run. A job is then analyzed, and a DAG of tasks is built and possibly split into stages of tasks. Tasks inside stages can run in parallel on different partitions of an RDD. Each stage *pipelines* its input RDD through a series of transformations (i.e., tasks), and the resulting RDD becomes the input data of the next stage (Zaharia et al., 2016).

1.2.9.3   RDD transformations and actions

Apache Spark provides a set of data transformations that can be applied to RDDs. A transformation is a lazy call, which means that when applied, it does not result in executing an operation, but rather, it results in a lineage of RDDs.

As explained in Apache Spark (2017), the framework provides a list of data transformations and actions. The following are the most commonly used transformations:

- **map**: Takes an input RDD of type RDD[T] and applies a function F to each of its elements to convert the input RDD into a new RDD of type *RDD[U]*;

- **filter**: Takes an input RDD of type T and returns a new RDD[T] of the same type containing only the elements that return *True* when F is applied (i.e., *RDD[T] => RDD[T]*);

- **flatMap**: Similar to a *map*, but in *flatMap,* each element of the input RDD is mapped to an array of elements not necessarily from the same type (*RDD[T] => RDD[U]*);

- **mapValues**: In contrast with the *map* transformation, which may change the key of a pair RDD, the mapValues passes each value in the (key, value) pair RDD through *F* without changing the keys (*RDD [(K, V)] => RDD [(K, U)]*);

- **groupByKey**: Takes an RDD of type *RDD[(k,v)]*, groups the tuples (k,v) by the key field k and returns a new RDD of type *RDD[(K, Seq[V])]*;

- **reduceByKey**: Works on pair RDD by grouping the values by keys and applying an aggregate function locally in each executor. The aggregate function transforms an *RDD[(K, V)]* into *RDD[(K, V)]*. This transformation provides a considerable reduction to shuffling as it performs a first aggregation step on the data available on a single executor data partition;

- **cogroup**: For each key in two given pair RDDs of type *RDD[(K, V)] and RDD[(K, W)]* return a new RDD of type *RDD[(K,(Seq[V], Seq[W]))]*, where each entry in that RDD contains a tuple of the key *K* and the sequences of values, having the key field equal to *K,* extracted from both of the input RDDs;

- **Join**: Given two pair RDDs of type *RDD[(K, V)] and RDD[(K, W)]*, return a new RDD of type *RDD[(K, (V, W))]* containing all pairs of elements with matching keys in both RDDs. This transformation may involve a considerable amount of shuffling.

An action must be called on the transformation lineage RDD to execute the transformation associated with that lineage. Below is a list of the actions that Apache Spark provides:

- **reduce**: Reduce all elements of an RDD using the provided function F to one element, and return it to the driver program (i.e., *RDD[T] => T)*;

- **collect**: Convert the given RDD into a local array (i.e., *RDD[T] => Array[T])*;

- **count**: Return the number of elements in the given RDD (i.e., *RDD[T] => Long)*.

1.3 Solving linear systems: General concepts and software libraries

A system of linear equations can be written in matrix form as in equation 1.28 below:

$$A x = b \tag{1.28}$$

where A is a matrix of coefficients, x is a vector of unknowns and b is a right-hand side vector or matrix of multiple columns. The matrix A of the linear system comes with different characteristics such as dense or sparse, triangular or diagonal, invertible or singular, symmetric or un-symmetric, and square or rectangular.

Several methods are presented in the literature that solve systems of linear equations, such as the direct methods for solving a dense linear system, the direct methods for solving a sparse linear system, the iterative methods for solving sparse linear systems, and the method of matrix inverse followed by matrix multiplication (e.g., $x = A^{-1} b$).

In this research, we focus on the exploration of the concepts and the state-of-the-art libraries for sparse matrix computing.

1.3.1 Linear systems solving concepts

In this section, different methods for solving general matrix equations will be discussed. We begin by discussing Gaussian elimination, and later, two matrix factorization techniques that allow for the solving with further processing.

As explained by Walter (2014), solving systems with triangular matrices can be accomplished using backward or forward substitution algorithms. Backward substitution applies to upper triangular systems of the form $U x = b$. Forward substitution, on the other hand, applies to lower triangular systems of the form $L x = b$.

Gaussian elimination transforms the system in equation 1.28 into an upper triangular system $U x = v$. The algorithm replaces each row of both sides of equation 1.28 by a linear combination of other rows applied to both sides of each row in equation 1.28. This triangular system is then solved using the backward substitution algorithm (Walter, 2014).

Similar to Gaussian elimination, LU factorization with full pivoting allows for converting the system in equation 1.28 into $L U x = b$, which can be solved using forward and backward substitution algorithms. LU factorization needs only the matrix A as an input variable in contrast with Gaussian elimination, which needs both inputs A and b. When the LU components are produced, they can be stored and used to solve equation 1.28 even if the right-hand side changes. The triangular system $L U x = b$ can be solved as follows:

1. Forward substitution to find Y by solving $L Y = b$;
2. Back-substitution to find X by solving $U X = Y$.

Another form of LU factorization is LU factorization with *partial pivoting*, which consists of permuting the rows of both A and b to avoid zero pivots using a permutation matrix as in equation 1.29 (Walter, 2014).

$$P A x = P b \tag{1.29}$$

Equation 1.30 presents the forward and backward substitution methods to solve the system in equation 1.29.

$$LU x = P b, \; Ly = Pb \text{ and } Ux = y \tag{1.30}$$

In addition to the LU factorization, the system in equation 1.28 can be rewritten using the QR factorization as in equation 1.31.

$$R\,x = Q^T\,b \qquad\qquad (1.31)$$

In equation 1.31, **Q** is an *(n ×n)* orthonormal matrix, and **R** is an *(n × n)* invertible upper triangular matrix. This new system can be solved by back-substitution. Contrary to LU factorization, QR factorization can be applied to both square and rectangular matrices (Walter, 2014).

1.3.2   Concepts and methods for solving sparse linear systems

As explained in (T. A. Davis, Rajamanickam and Sid-Lakhdar, 2016), four phases characterize direct sparse solvers. First, an *ordering* phase consists of re-ordering the pivots to reduce fill-in and enhance parallelism. Fill-in is a well-known issue that matrix solving algorithms face. It consists of reducing the transitions from zero to non-zero while executing the algorithm by performing row or column permutations. There are specific algorithms that would provide the best re-ordering, such as the minimum degree (MD) and the METIS re-ordering algorithm. Second, a *symbolic* phase analyses the matrix structure and determines a proper pivot sequence that significantly reduces both memory and CPU requirement. Third, a *numerical factorization* phase uses the pivot sequence determined in the symbolic phase to factorize the matrix numerically. Fourth, a *solve* phase that operates on triangular systems using numerical algorithms such as forward elimination and back-substitution is used to solve the system.

The separation of these phases is beneficial when solving practical problems such as the ones found in LCA. A linear system in which the Left Hand Side(LHS) is not changing can be solved by computing the analysis and factorization phases only once; the solve phase is repeated for each Right Hand Side(RHS). An LCA system solver can be designed to build

the system represented by matrix A and apply the phases of analysis and factorization on it. Afterwards, when calculation requests come with different demand vectors b, only the solve phase would need to be executed on each calculation request.

An iterative algorithm is an algorithm that tries to find an approximate solution to the matrix equation 1.28 by minimizing the approximating error on each iteration. As explained in Walter (2014), the general concept relies on first decomposing the matrix A into a sum of two matrices, as shown in equation 1.32. Given that matrix A is invertible, equation 1.32 can be rewritten to give the iterative equation 1.33. The decomposition of matrix A will be optimal if matrices M and v in equation 1.33 result in convergence when k tends to infinity.

$$A = A_1 + A_2, \quad A^{-1} \ exists \tag{1.32}$$

$$x = -A_1^{-1} A_2\, x + A_1^{-1}\, b = M\, x + v, \quad x_{k+1} = M\, x_k + v \tag{1.33}$$

Several methods are presented in the literature that implements this logic, such as Jacobi, Gauss-seidel, and successive overrelaxation. These methods differ by how matrix A is decomposed. In the specific case of Jacobi, the iterative process can be described as in equation 1.34, where D is the diagonal matrix. Equation 1.35 implies that the iteration in equation 1.35 would become as expressed in equation 1.36.

$$x^{k+1} = -D^{-1}\,(L + U)\, x^k + D^{-1}\, b \tag{1.34}$$

$$L + U = A - D \tag{1.35}$$

$$x^{k+1} = (I - D^{-1} A)\, x^k + D^{-1}\, b \tag{1.36}$$

The solution of the system (i.e., x^*), which is equal to $A^{-1}\, b$, cannot be computed directly. Similarly, the error $\delta\, x^k$ which is equal to $x^k - x^*$ cannot be computed. Therefore, to

measure the quality of the solution, we use the residual as in equation 1.37. Normalizing equation 1.36 will give an iterative formulation of the solution as in equation 1.38.

$$r^k = b - Ax_k = -A \, \delta \, x^k \tag{1.37}$$

$$x^{k+1} = x^k + r^k, \qquad r^{k+1} = r^k - A \, r^k \tag{1.38}$$

As explained in Walter, 2014), based on the *Krylov* subspace analysis, the following properties hold:

1. Each subspace K_k is within a higher subspace: K_{k-1};
2. The next larger subspace K_{m+1} can be obtained using a matrix product with previous subspace matrix (i.e., line 6 of Figure 1.5);
3. If $x \in K_m$, then $x = K_m \, Z$ where Z is an approximate solution at iteration *m*.

```
1  k₀ = b
2  while r is not acceptable do
3      z=MIN_{x∈K_m}||b − Ax||₂
4      x=K_m z
5      r=residual(..)
6      K_{m+1} = AK_m
7  end
```

Figure 1.5 *Krylov* solving example

These properties of the Krylov subspace allows for the design of very efficient solving algorithms. Figure 1.5 presents an algorithm to calculate an approximate solution by minimizing a convex function. Using the *Krylov* methods, the solver will start with a small domain k_0, and progressively, through matrix multiplication, it will pass to a larger space to search for a solution. This makes the search space, at any given time, to be restricted to a certain sub-space and therefore, it will faster to compute. Section 1.3.4 will present state-of-

the-art libraries, implementing *Krylov* methods, and that can solve very large systems in a few seconds.

1.3.3   Concepts and methods for computing the matrix inverse

The literature identifies several methods for calculating a matrix inverse, mainly Gauss-Jordan elimination, Cramer's Determinant Cofactor, and matrix factorization methods (Walter, 2014).

An n-by-n square matrix is invertible (also non-singular or non-degenerate) if there exists an *n*-by-*n* square matrix A_{inv} such that equation 1.39 is satisfied. I_n denotes the *n*-by-*n* identity matrix. A_{inv} is uniquely determined by *A* and is called the inverse of *A*, denoted by A^{-1}.

$$A\,A_{inv} = A_{inv}\,A = I_n \tag{1.39}$$

Gauss-Jordan elimination uses pivoting and row or column permutations to transform the original matrix into its Reduced Echelon Form. The algorithm will iteratively operate on the augmented matrix [A|I] to convert it into the identity-inverse augmented matrix [I|A⁻¹].

The Cramer's Determinant consists of calculating the (n+1) matrix determinant of size (n x n) which each takes O (n!) giving a total of O (n x n!). Cramer's method is highly inefficient, and the literature advises against it.

Several methods of matrix decomposition allow for the calculation of the inverse of a matrix such as LU decomposition, Singular Value Decomposition (SVD), and QR decomposition.

The LU decomposition method will decompose the original matrix *A* into a product of two matrices: **L** and **U**, where **L** is a lower triangular matrix, and **U** is an upper triangular matrix. Matrix inverse operations follow this factorization step as in equation 1.40.

$$A = LU, \; A^{-1} = U^{-1}L^{-1} \tag{1.40}$$

The QR decomposition approach first decomposes the matrix A in equation 1.28 into a product of an orthogonal matrix **Q** and an upper triangular matrix **R**, and second, it carries out the inverse and transposes operations as shown in equation 1.41.

$$A = QR, \; A^{-1} = \; R^{-1}\,Q^{T} \tag{1.41}$$

The rank k Singular Value Decomposition (SVD) factorizes the original matrix A of dimensions $m \times n$ into U, V, and σ . Vector U holds the Right-Singular vectors and is of dimensions $m \times k$, V holds the Left-Singular vectors and is of dimension $n \times k$, and σ is a $k \times k$ diagonal matrix, which holds the singular values of matrix A.

$$A = U\,\sigma\,V^{T}, \; A^{-1} = \; V\,\sigma^{-1}\,U^{T} \tag{1.42}$$

After computing the factors, the SVD approach will carry out the transpose and diagonal inverse operations.

1.3.4   Matrix computing libraries

In this section, we provide a review of the state-of-the-art matrix libraries that are currently available in the literature. We omit the libraries that are designed for hardware or devices that we did not have access to (e.g., GPU or Vector machines) or the libraries that require commercial licensing. We focus instead on Open Source libraries that have support for sparse matrix computing.

Spark MLlib (Spark MLlib, 2017), is a library for machine learning built on top of Apache Spark. MLlib comes with support for local and distributed matrix computing. The library performance is enabled by the use of distributed computing for large matrices and by the use of low-level BLAS, through the jblas interface implemented in Breeze, for local matrix

computation. As explained in Spark MLlib (2017), the library supports local and distributed, dense and sparse, matrices and vectors using:

1. A RowMatrix implementation which distributes the matrix rows in RDD [vector];
2. An IndexedRowMatrix implementation which distributes the matrix rows into RDD[IndexedRow] where an IndexedRow is a tuple of (rowIndex, Vector);
3. A coordinate matrix implementation which distributes the matrix entries into an RDD of type RDD[(i: Long, j: Long, value: Double)], where i is the row index, j is the column index, and value is the matrix cell value;
4. A BlockMatrix implementation where the matrix entries are grouped into matrix blocks of type MatrixBlock. The list of blocks is distributed in RDD[Matrix]. Each MatrixBlock is comprised of two components grouped in a tuple of ((Int, Int), Matrix). The (Int, Int) is the index component of the block, and Matrix is the portion of the matrix or sub-matrix that resides at the position given by the 2D index. Each Matrix will have the same size conFigured by two parameters: rowsPerBlock and colsPerBlock.

ParallelColt (Wendykier and Nagy, 2010), from the CERN laboratory, provides support for dense and sparse parallel matrix computation. This library does not provide for advanced sparse matrix direct system solving. However, it provides for wrappers to the Krylov subspace iterative solver methods by using the MTJ (Matrix Toolkit for Java) library. ParallelColt provides wrappers for:

1. Iterative solvers such as BiConjugate Gradients (BiCG), BiConjugate Gradients stabilized (BiCGStab), Conjugate Gradients (CG), Conjugate Gradients Squared (CGS), Generalized Minimal Residual using restart (GMRES);
2. Preconditioners such as Diagonal, Incomplete Cholesky without fill-in (ICC), Incomplete LU without fill-in (ILU), Incomplete LU with fill-in (ILUT), Symmetrical Successive Overrelaxation (SSOR), and Algebraic Multigrid (AMG);
3. Formats such as multi-dimensional arrays in Java, sparse and dense 2D, and 3D matrices, and both CSR and CSC sparse matrix formats.

UMFPACK (T. A. Davis, 2004), is a package which consists of a list of procedures specializing in the solving of unsymmetric sparse linear systems, using an un-symmetrical multi-frontal method written in Fortran. UMFPACK has installation options to use the many versions of the BLAS or no BLAS at all. As explained by the Umfpack user guide (T. A. Davis, 2011), five primary UMFPACK routines are required to factorize *A* or solve *A x = b*:

- *umfpack_di_symbolic* has as a principal function is to pre-order the columns of *A* to reduce fill-in. It returns a pointer of type *void** to the *Symbolic* object. The Symbolic object will next be fed as input into the next phase of solving, which is the numerical factorization;
- *umfpack_di_numeric* takes as input the *Symbolic* object and numerically scales and then factorizes the sparse matrix of equation 1.9 in the product *LU*. In its turn, this procedure will give a *Numeric* object as a pointer of type void* which is fed as input to the solver routine *umfpack_di_solve*;
- *umfpack_di_solve* solves equation 1.28, using the numeric factorization embedded in the *Numeric* object computed by *umfpack_di_numeric*;
- *umfpack_di_freesymbolic* performs a safe freeing of the *Symbolic* object created by *umfpack_di_symbolic*;
- *umfpack_di_freenumeric* allows the user to free the *Numeric* object created by *umfpack_di_numeric*.

Eigen (Gaël and Benoit, 2017) is a C++ library that provides support for dense and sparse matrix operations with a variety of numeric types (integers, complex and custom numeric types) of operations. Eigen++ performance is provided by using BLAS and Lapack subroutines for dense matrix components, explicit vectorization for a range of processor architectures, and the use of BlockMatrix formats which allows for the use of parallel computation on large matrices. Eigen provides the following features used in LCA algorithms:

1. Wrappers for major sparse direct solvers such as Umfpack, SuperLU, and Paradiso;
2. An Implementation of sparse iterative solvers such as BiCGStab, CG, and CGS;
3. An Implementation for matrix decomposition methods such as LU, QR, and SVD;
4. An Implementation for vectorized matrix-matrix and matrix-vector multiplications and additions using VectoXd, Matrix, and SMatrx data types.

Explicit vectorization is provided with graceful fallback to non-vectorized code. Various architectures are supported: SSE, AVX, ARM NEON, PowerPC AltiVec/VSX, ZVector SIMD instruction sets.

MUMPS (L'Excellent, 2017), or Multi-Frontal Massively Parallel Sparse Direct Solver, was developed by the University of Lille, France. It focuses on HPC big sparse matrix computing and provides support for the solution of large symmetric positive definite sparse matrices as well as general symmetric and unsymmetrical matrices. MUMPS provides support for parallelism in factorization and solving phases. The library relies on the Distributed Multi-Frontal Solver with shared memory OpenMP directives, and Dynamic Distributed Scheduling with MPI.

ViennaCL (Rupp et al., 2016) implemented in C++ provides for a wide variety of sparse matrix computing such as matrix-matrix and matrix-vector multiplication, iterative solvers, direct solvers, and a variety of preconditioners. Moreover, it allows access to objects from other libraries such as Eigen, Armadillo, MTL4, uBLAS. It also provides support for CUDA, OpenCL, and OpenMP. When none of the parallel systems is available on the target machine, the library falls to a single-threaded execution. ViennaCL provides for porTable performance by using an internal database of a configuration suitable for each target device.

SuperLU (Li, 2005), is a set of C subroutines for the solving of large sparse symmetric and unsymmetric systems. The algorithms implement a Gaussian elimination algorithm that takes advantage of the sparsity of the matrix and the available parallelism of the test machine. SuperLU comes in three versions:

1. SuperLU is a sequential version without parallelism;
2. SuperLU_MT is designed for Shared Memory Processors which use multi-threading to speed up the computation. Extensions in C are developed to provide parallelism through OpenMP or PThreads;
3. SuperLU_DT uses MPI to disburse the computation on processors distributed across cluster computers.

PETSc (Balay et al., 2014), the Portable Extensible Toolkit for ScientficComputing, is a suite of algorithms and data structures for solving large scale systems using parallel computing. PETSc is implemented in C and therefore accessible from C++. It also provides an interface for Fortran access. PETSc uses MPI for message passing between parallel processes. The library provides a high-level MPI collective: the user does not need to make MPI calls to exchange messages. Some of the functionalities of PETSc includes Krylov subspace methods, distributed matrices, and vectors. Finally, PETSc allows interoperability with major matrix libraries such as MUMPS, Hypre, and Trilinos, among others.

We highlight the following GPU based and intel-based commercial libraries that are at the top of every performance benchmark. These libraries and others in their categories were omitted from our research experimentations due to the research scope:

Paradiso (Schenk and Gartner, 2014), a commercial library developed by Intel, specializes in sparse matrices on CPUs and it allows for the solving of symmetric and unsymmetrical real or complex systems, the LU decomposition with complete pivoting, and the parallelism on SMP (Shared Memory Processors) and distributed SMP.

cuSparse (Naumov, 2011), the Nvidia Cuda Sparse Matrix library supports Blocked CSR sparse matrix and dense CSR, COO, CSC storage formats, levels one, two and three sparse and dense matrix-vector operations and sparse by sparse matrix addition/multiplication operations, sparse triangular solve and finally incomplete factorization preconditioners ilu0 and ic0.

1.4 Libraries and calculators review

In this section, we provide a review of some of the recent experiments conducted on major parallel frameworks such as MapReduce, Apache Spark, MPI, and OpenMP. We focus our review on iterative and matrix computing libraries.

Also, we provide a comparison between the features adopted in ParallelLCA0.1 and other major Open Source calculators.

1.4.1 Parallel framework review

Gopalani and Arora (2015) compared two implementations of K-Means algorithms using MapReduce and Apache Spark MLlib. The experiments were conducted for two datasets of sizes 62MB and 1240MB. The three experiments that were conducted are indicated on the x-axis of Figure 1.6: the 62MB using one node, the 1240 MB using one node, and the 1240MB using two nodes. The research shows that Apache Spark, with its in-memory processing framework, can provide an enhancement of almost two times for the iterative K-Means algorithm when compared to MapReduce.
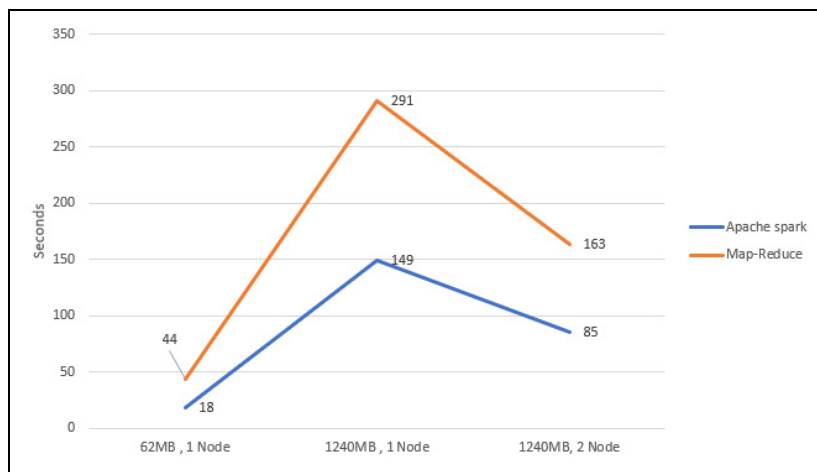
Figure 1.6 Spark vs. MapReduce for K–Means

Taken from Gopalani and Arora (2015)

Reyes-Ortiz, Oneto, and Anguita (2015) compared the performance of two machine learning algorithms when using MPI/OpenMP versus Apache Spark. The algorithms are KNN and Pegasos. Figure 1.7 is an extract of this experiment for the KNN algorithm. Similar conclusions can be found for the Pegasos algorithm.
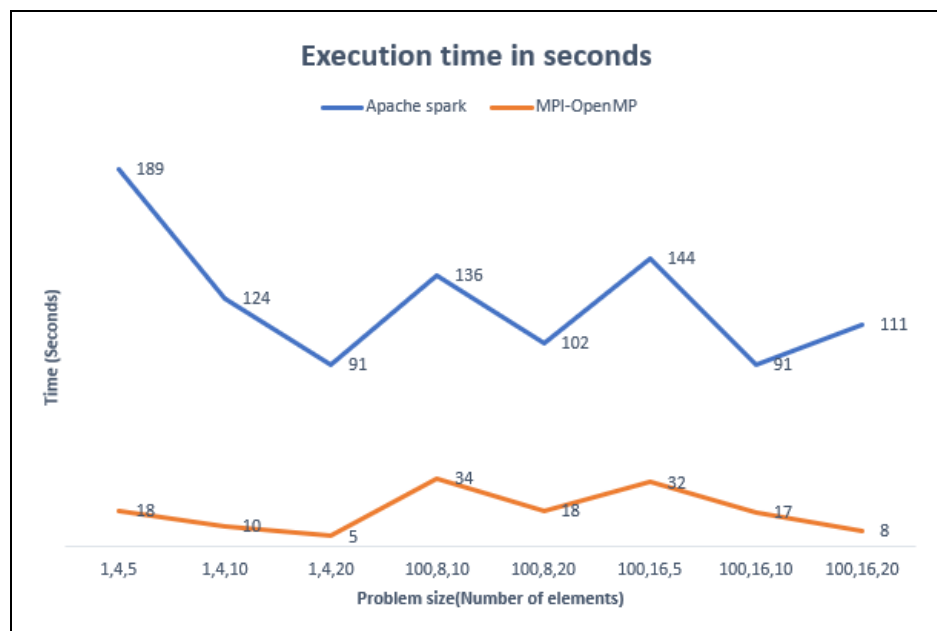
Figure 1.7 MPI vs. Apache Spark for KNN.
Taken from Reyes-Ortiz, Oneto and Anguita (2015)

The experiment concluded that the MPI/OpenMP implementation can be as much as ten times faster than the Apache Spark implementation and that the difference tends to decrease with the increase of dataset size. With smaller dataset sizes, Apache Spark falls way behind. This is shown in Figure 1.7, where MPI/OpenMP overpasses Apache Spark for smaller datasets.

1.4.2   Matrix libraries performance review

1.4.2.1   Direct solvers review

Considering the sparse LU factorization problem, a comparative study published by Gupta and Muliadi (2000) for a set of state-of-the-art sparse direct solvers, namely *MUMPS, S+, SPOOLES, SuperLU, UMFPACK,* and *WSMP* was published. The experimentation was conducted on twenty sparse matrices generated in different scientific domains. The dimensions of the matrices range from 16,783 to 659,033 number of rows and from 145,149 to 2,374,001 non-zero elements. The results highlighted *WSMP* as the fastest and most memory efficient serial solver; the parallel version was yet not developed. Among the parallel libraries, *MUMPS* was shown to have the best overall performance.

As shown in a paper published by Gould, Hu, and Scott (2005) comparing sparse solvers for thirty matrices with minimum dimensions of 10,000 rows, the clear winners were: MA57 and PARADISO. Of the remaining solvers, BCSLIB-EXT and unsymmetrical MUMPS perform the best. From this study, we ignore MA57 and BCSLIB-EXT as they are designed for symmetric matrices only (i.e., LCA matrices are unsymmetrical), as well as Paradiso as it is commercial. We retain MUMPS.

A study, published by Tracy, Oppe, and Engineer U.S.A (2005), aimed at comparing the performance between a set of sparse matrix solvers. The solvers tested in this study are *SuperLU, UMFPACK*, and other domain-specific libraries (e.g., *Bansol, SSGETRF, SSGETRS, SSTSTRF, SSTSTRS*). This experimentation was tested on matrices, generated out of the CGWave software, which comes in different dimensions: small (130,255) nodes, medium (265,119 nodes), and large (496,286 nodes). This paper concluded that the Umfpack solver outperforms the SuperLU solver.

Table 1.5 Time of direct sparse solvers of the Eigen++ library.

Taken from Gaël and Benoit (2017)

Rows	NNZ	Umfpack (Seconds)	superLU (Seconds)	pastixLU (Seconds)
160,000	1,750,416	3.46381	5.89542	16.1594
19,788	308,232	0.0280053	0.0194402	0.268747
12,855	72,069	0.0288333	0.0225195	0.0750265

A study made by Eigen++, published by Gaël and Benoit (2017), compares major sparse direct solvers supported in the library. An extract of that study is shown in Table 1.5. The results show that the Umfpack library provided better performance than the SuperLU and pastixLU.

1.4.2.2   Iterative solvers review

The literature shows that the fastest algorithms used for unsymmetrical sparse systems solving and implementing the Krylov subspace methods are the Conjugate Gradient Squared (CGS), the Generalized Minimum Residual Method (GMRES), and the Biconjugate Gradient Stabilized (BiCGStab) algorithms. Specialized libraries have been developed over time that implements these Krylov algorithms.

An extract of the experiment published by Eigen++ (2017) comparing several sparse solvers is shown in Table 1.6. It shows that for 160,000 rows and 1,750,416 non-zero elements, BiCGStab was able to solve the system in 1.49 seconds.

Table 1.6 Time of sparse iterative solvers in the Eigen++ library.

Taken from Gaël and Benoit (2017)

Rows	NNZ	BiCGSTAB (Seconds)	BiCGSTABILU (Seconds)	GMRESILUT (Seconds)	CG (Seconds)
160,000	1,750,416	1.49473	3.34948	3.54288	-
19,788	308,232	0.037642	0.0186552	0.0238484	-
12,855	72,069	0.249916	1.39486	1.42741	0.239551

A benchmark comparison between ViennaCL and PETSc was published in Rupp et al. (2016) comparing the solving time of large matrices. Table 1.7 shows the order of magnitude of execution per solver iteration on AMD FirePro W9100 for linear systems starting from 100,000 rows or columns.

Table 1.7 ViennaCL vs. PETSc time per solver iteration benchmark.

Taken from Rupp et al. (2016)

Order/ Iteration Size	CG		GMRES		BiCGStab	
	ViennaCL	PETSc	ViennaCL	PETSc	ViennaCL	PETSc
10^3	10^{-4}	10^{-5}	10^{-4}	10^{-4}	10^{-4}	10^{-4}
10^4	10^{-4}	10^{-3}	10^{-4}	10^{-3}	10^{-4}	10^{-3}
10^5	10^{-4}	10^{-3}	10^{-3}	10^{-3}	10^{-3}	10^{-2}

Similarly, for larger matrices and several types of matrices, Rupp et al. (2016) show that for matrices having at least 11 million entries and at most 37 million entries, the execution time will vary between 0.94 and 8.92 milliseconds per solver iteration. Similar execution times have been shown when using the "Nvidia Tesla K20m" hardware.

Solving a large linear system, as indicated in Table 1.8, requiring tens of iterations may take a few hundred milliseconds to compute or a few seconds when using the many processors available on hardware like the "AMD FirePro W9100" or the "Nvidia Tesla K20m".

Table 1.8 ViennaCL performance for large matrices.

Based on (Rupp et al., 2016)

Name	Rows	Cols	NNZ	Solver	Time(ms) / Iteration
KKT POWER	2,063,494	2,063,494	12,771,361	GMRES	6.17
RM07R	381,689	381,689	37,464,962	GMRES	8.92
KKT POWER	2,063,494	2,063,494	12,771,361	BICGSTAB	6.5
RM07R	381,689	381,689	37,464,962	BICGSTAB	8.05
PWTK	217,918	217,918	11,524,432	CG	1.19
SHIPSEC	140,874	140,874	3,568,176	CG	0.94

1.4.2.3 Matrix inverse libraries review

A paper published by Xiang, Meng, and Aboulnaga (2014), presented a scalable matrix inversion algorithm based on LU decomposition for large size matrices. The experimentation consisted of running the proposed algorithm with matrices of different sizes as in Table 1.9.

Table 1.9 MapReduce matrix inversion experiment setup.

Taken from Xiang, Meng and Aboulnaga (2014b)

Matrix type	Order	Non-zero size (billion)
M1	20,480	0.42
M2	32,768	1.07
M3	40,960	1.68

All experiments were performed on Amazon's Elastic Compute Cloud (EC2 instances), having 3.7 GB of memory, one virtual core with 2 EC2 compute units (i.e., similar to -era 1.0–1.2 GHz AMD Opteron or Xeon processor). The resulting execution time takes 20 minutes, 38.6 minutes, to 1.5 hours for the matrices M1, M2, M3 respectively.
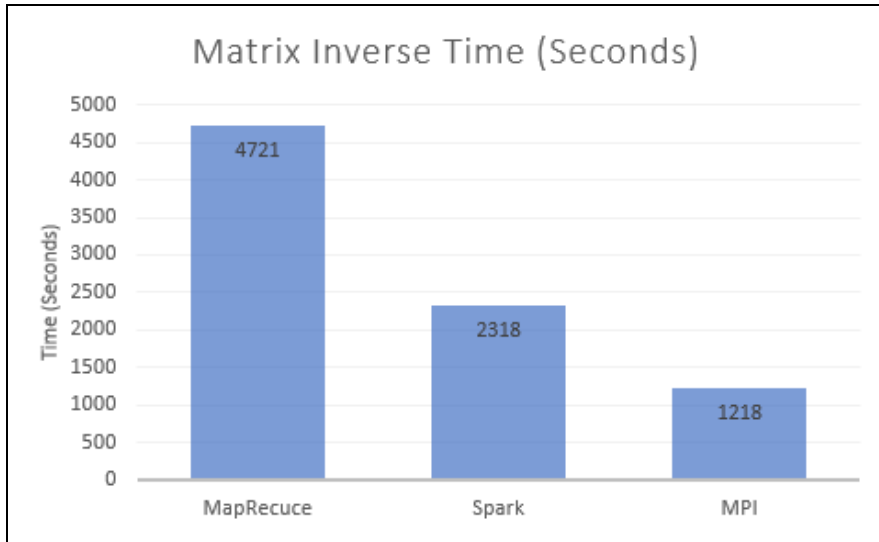
Figure 1.8 Matrix inverse on Spark vs MPI vs MapReduce.

Taken from Yang Liang et al. (2016)

In a paper published by Yang Liang, Liu, Cheng Fang and Ansari (2016) in late 2016, the researchers compared MPI, Apache Spark, and MapReduce when implementing a scalable matrix inversion algorithm based on LU decomposition for large size matrices. In this paper, the experiments were tested on a cluster of commodity servers. Each server had 64GB memory and two 2.1GHz Intel Xeon CPUs, summing to a total of 12 physical cores. The storage used eight 7200 RPM hard disks. The servers were connected by Gigabit switches.

The experiment shows that with respect to performance, MPI, Apache Spark, and MapReduce came first, second, and third, respectively. The difference between MPI and the other two JVM technologies increases when the order of the matrices increases. Figure 1.8 shows the results obtained on a seven-server cluster for a matrix of size 40,960 rows and 31GB in file size.

1.4.3   LCA calculators review

Table 1.10 shows a comparison for linear system solving, with regard to the adopted matrix libraries, among the calculators Brightway2.0, ParallelLCA0.1 and OpenLCA7.

Table 1.10 Calculators—matrix computing libraries

Research/Calculators	Direct solving	Iterative solving
OpenLCA7	Eigen++ SuperLU	Eigen++ BiCGStab
Brightway2	SuperLU – Umfpack - Paradiso	SciPy CGS
ParallelLCA0.1	Umfpack - MUMPS	Eigen++ BiCGStab

Table 1.11 shows a comparison in the adopted methods for computing Monte-Carlo and the GSA algorithms among the calculators Brightway2.0, ParallelLCA0.1 and OpenLCA7. OpenLCA provides OAT (One at A time) sensitivity analysis only.

Table 1.11 Calculators—Monte-Carlo and GSA methods

Research/Calculators	Monte-Carlo	GSA
OpenLCA7	Sequential – Java based	Not available
Brightway2	Parallel - using Multiprocessing Python library	Kendal Tau
ParallelLCA0.1	Parallel - using OpenMP and MPI	Parallel LCA using OpenMP and MPI

Table 1.12 provides a feature comparison between ParallelLCA and other major Open Source LCA calculators. Currently, ParallelLCA0.1 does not support reading directly from the Ecoinvent database or geo-localization.

Table 1.12 Calculators—features comparison.

Comparing ParallelLCA0.1 to Brightway2.0 and OpenLCA7

Research/Feature	OpenLCA7	Brightway2	ParallelLCA0.1
Static basic LCA	✓	✓	✓
Static phase - Hybrid solver	⊘	⊘	✓
Contribution analysis	✓	✓	✓
Monte-Carlo	✓	✓	✓
Monte-Carlo- Hybrid solver	⊘	⊘	✓
HPC Monte-Carlo	⊘	✓	✓
Parallel Monte-Carlo	⊘	✓	✓
Parallel Database	⊘	⊘	✓
GSA	⊘	✓	✓
Pre-calculated aggregated datasets	⊘	✓	✓
Geo-Localization	✓	✓	⊘
Distributed Monte-Carlo	⊘	✓	✓
Ecoinvent support	✓	✓	⊘
Parallel in-memory Database	⊘	⊘	✓
Parallel GSA	⊘	✓	✓
Distributed GSA	⊘	✓	✓
Pre-Sampling Monte-Carlo	⊘	⊘	✓

1.5   Conclusion

This chapter provided a review of LCA algorithms, state-of-the-art parallel computing frameworks, and matrix libraries adopted in the industry.

As explained in section 1.1, two methods are commonly used for LCA calculation. The Sequential Method uses a graph traversal and aggregation approach. The Matrix Method translates the graph interconnection into a system of linear equations, which can be solved with different methods, as explained in section 1.3.

Using matrix-based methods for linear system solving and matrix inverse computing that are designed for generic dense matrices requires the use of a considerate amount of computing resources to reach a reasonable execution time. However, using specialized sparse matrix solvers, direct or iterative, provides the solution within tens or hundreds of milliseconds for mid-size problems and in seconds for large systems.

Apache Spark provides a convenient API for data-parallel operations over large data sets. OpenMP provides for a multi-threading computing framework using a declarative syntax allowing, with minimal effort, to access state-of-the-art parallel computing research. Finally, MPI allows for implementing multi-processes algorithms that can be local or distributed across a cluster of servers.

The research will experiment with Apache Spark and assesses its relevance for LCA algorithms. However, as shown in the literature, the combination of MPI/OpenMP provided much better performance for computing algorithms similar to those found in LCA (i.e., iterative and matrix inverse algorithms).

Current LCA calculators implement their algorithms predominantly by using the matrix only approaches, and they provide limited features using the Sequential Method. When analyzing large systems with hundreds of thousands of LCA processes, the use of GPUs and a cluster of computing resources is inevitable for algorithms based on matrix computing. The research will provide a calculator with parallelism enabled in the different levels of the LCA algorithms. In addition, the research will try to find alternatives that replace the matrix algorithms to minimize the number of resources needed for large systems.

CHAPTER 2

PARALLEL FOREGROUND AWARE LCA

The project began with the requirement to develop an LCA calculator specially designed for the construction industry (i.e., reading Revit digitalized building plan files as an input). Later, a decision was made to extend the scope of work and allow for the processing of text files containing generic LCA data generated by a companion project.

This chapter describes the required functional outcomes and the proposed methods to reach them.

2.1 Prototype Scope and Goals

2.1.1 Research scope

The calculator is tested with data from the OpenLCA database. During the validation of the results, we discovered that OpenLCA has inconsistencies in using the pedigree method for Monte-Carlo sampling. In addition, the test data lacked a precise linking between the demand and supply processes. The validation of the results obtained using our calculator has to be accomplished, taking into consideration this limitation.

2.1.2 Functional requirements

In this section, we present the functional requirements of the prototype by describing the data views that the calculator is required to generate, which are:

1. The *LCI – Process contribution* data report shows a Table with the following fields: Flow Id, Total Value, ProcessId, Process Contribution;

2. The *LCIA - Process contribution* data report shows a Table of the following fields: Impact Id, Impact Score, ProcessId, Process Contribution, Unit;

3. The *Hierarchical Aggregate Contribution* report provides the cumulative contribution of individual activities to the total impact of a given impact category. By cumulative we mean the aggregation over all the processes which are involved in the production of a given process**;**

4. *The LCIA uncertainty report* shows a Table with the following fields: Impact Id, Mean, STD, 2.5 %, 5 %, 25 %, 75%, 95%, 97.5%;

5. The *Global Sensitivity Analysis* report shows, for each impact category, the top 100 input variables that contribute the most to its uncertainty. This report shows a table consisting of the following columns: OutputVariableId, Input Parameter Type, InputVariableId, SROCC (Spearman Rank Order Correlation Coefficient).

2.2   Calculator modules overview

In this section, we present the different modules that a request for calculation traverses from its arrival at the REST service to the exit point, where the calculator returns a JSON message to the calling client program containing the HTTP links of the generated reports.

ParallelLCA provides the following modules:

1. The REST Service module, which consists of a set of web methods (e.g., calculate, GetFile) implemented using the Play framework, presented in section 2.3;

2. The LCA Database module, presented in section 2.4;

3. The Calculator Data loading presented in section 2.5;

4. The LCA parameters module, presented in section 2.6;

5. The graph factory feature for building the LCA processes **(g)** presented in section 2.7;

6. The parsing of the graph **(g)** and the building of the matrices **A, B**, **Q** and demand vector **f**, described in section 2.8;

7. Computing the scalars vector *s*, presented in sections 2.9, 2.10, 2.11, and 2.12;

8. Computing the inventory vector g (equation 1.4), and the impact vector h (equation 1.5);

9. Building contribution reports for LCI and LCIA and the hierarchical upstream LCIA report (described in sections 2.14, 2.15, and 2.16);

10. The Stochastic calculation modules:

    A. An MCS simulator to generate input and output samples;

    B. An Uncertainty Propagation utility to analyze the uncertainty at the output variables (described in sections 2.17, 2.18, and 2.20);

    C. A Global Sensitivity Analysis module to analyze the contribution of input variables uncertainties to the uncertainties in output variables (described in sections 2.17, 2.19, and 2.20).
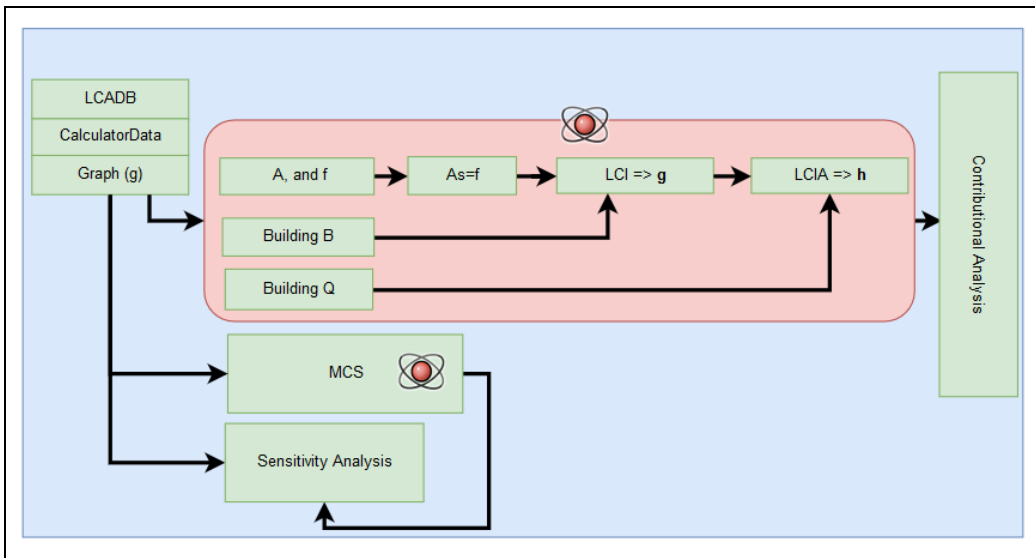
Figure 2.1 ParallelLCA calculator steps.

Vertical steps can run in parallel

The calculator steps, shown in Figure 2.1, are split into four categories that are executed in order:

1. An Initialization step that includes building the LCADB, loading the CalculatorData object, and constructing the graph (g) entities;

2. The *Foundational LCA* kernel step that includes loading matrices *A*, *f*, solving equation 1.3, computing the inventory g vector and the impact scores h vector;

3. The contribution analysis step, which consists of the generation of process-based contribution reports;

4. The stochastic phase step, which includes performing Monte-Carlo Sampling (MCS) followed by the generation of the uncertainty propagation report and the global sensitivity analysis report.

Steps 2, 3, and 4 can be executed in parallel after that step 1 has finished executing. The sensitivity analysis can start computing while MCS is running, but needs the output of MCS to be completed.

Before launching the calculator algorithms implemented in a C++ library, a layer written in Scala communicates with the library functionalities of ParallelLCA. This JVM-C++ communication is enabled using the Java Native Interface(JNI).
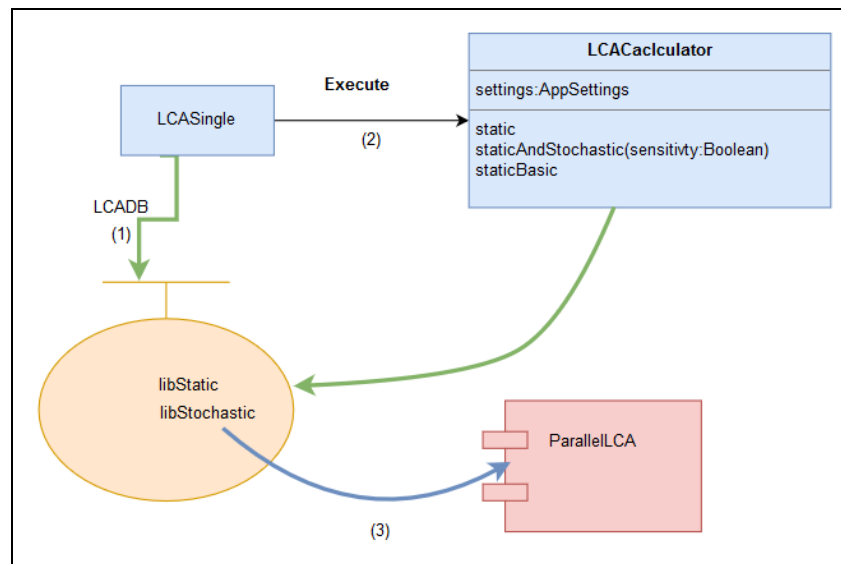
Figure 2.2 The REST service – ParallelLCA communication
barrier architecture

An *AppSettings* object holding the settings of the current application settings is passed created in the Scala layer and passed to ParallelLCA. *LCASingle* is a Scala class that initiates the LCADB creation through a direct JNI call. Also, LCASingle launches the calculation functions in ParallelLCA through the LCACalculator static and stochastic JNI wrappers.

ParallelLCA0.1 implements LCA algorithms in various forms. One example of this variety is found in the implementation of Monte-Carlo sampling, where the calculator provides an implementation using OpenMP and another one using MPI. ParallelLCA0.1 provides each of the implementations in a specialized package such as Caclualtors_MPI, Calculators_OpenMP, Calculators_Foreground, Calculators_Presampling, etc. This modular separation between the various implementations allows the user of ParallelLCA to choose which package to use for a given project and therefore, to minimize the amount of code to manage and maintain.
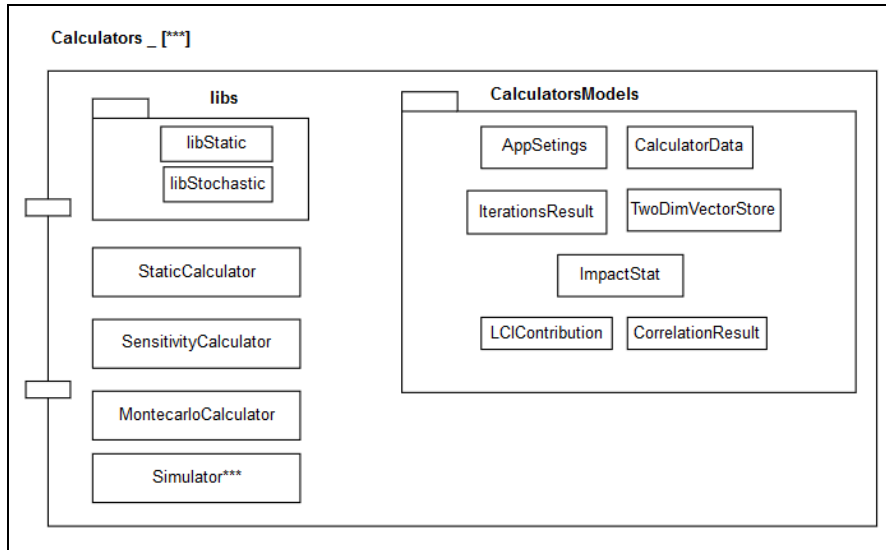
Figure 2.3 Calculator model architecture

The various calculator implementations follow similar architecture, as shown in Figure 2.3. This architecture can be described as follows:

1. The libs package plays the role of interfacing between the REST service and the inner calculator functionalities;
2. The *StaticCalculator* class implements the different phases of static LCA;
3. The *SensitivityCalculator* class defines how the ranking and the correlations operations are parallelized;
4. The *MonteCarloCalculator* class defines how the Monte-Carlo iterations are launched. It is in this class that the type of parallelism is defined to use OpenMP or MPI. The Simulator class defines how a single Monte-Carlo iteration is implemented.

The calculator implements classes for the LCA graph components, the parameter evaluation entities, the different uncertainties representations, and the pedigree method levels. This implementation is shown in Figure 2.4.
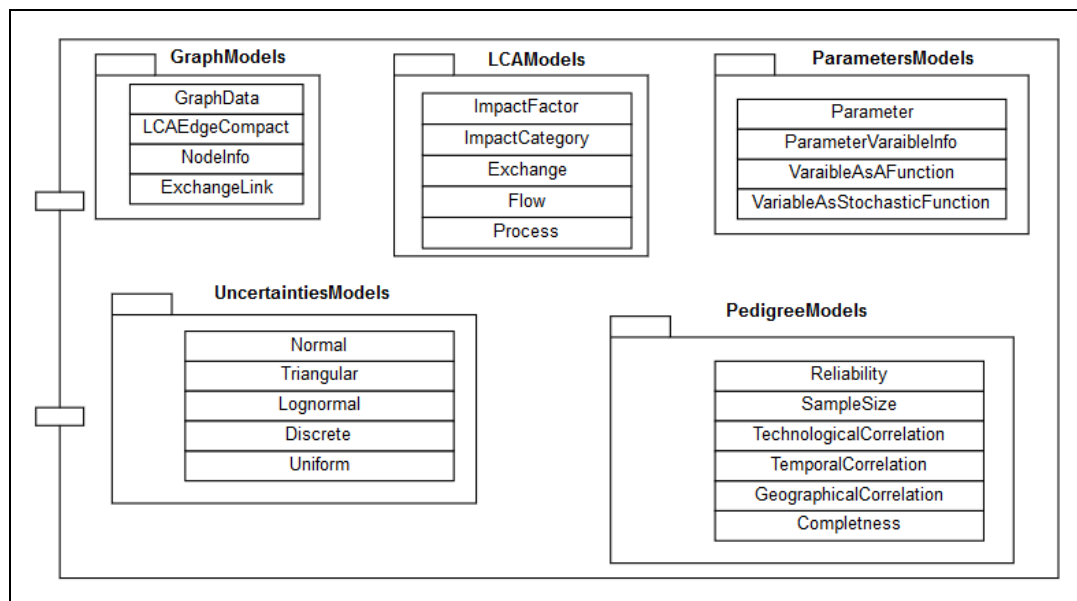
Figure 2.4 Calculator class models

Before the calculation starts, the different calculator implementations create, in an initialization step, objects that are necessary for their functioning. Several factories, as shown in Figure 2.5, have been provided for that purpose, such as for the creation of the LCA graph object, the different matrices in LCA, and the CalculatorData object.

The *LCADB* class represents the database object of the calculator. The objects in LCADB are shown in Figure 2.4 in the LCAModels package. For each of these classes, a corresponding builder class is provided. For example, for the *Exchange* class, an ExchangeBuilder class creates the *Exchange* object and inserts it in a collection of exchanges, as explained in section 2.4. The different builders of LCADB objects are implemented as classes in the DAL (Data Access Layer) package.
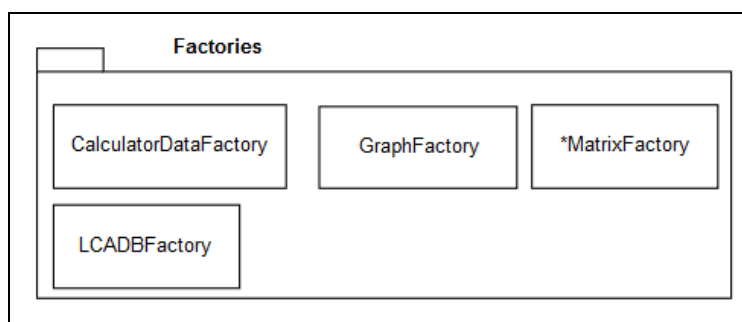
Figure 2.5 Object Factory models

The different factories, reports, and calculators use functionalities implemented in the *Utilities* package classes. Different utilities are provided in this package such as for graph building and traversal, for LCA upstream computing, for pedigree uncertainty computing, and for other functionalities such as file and string manipulation.
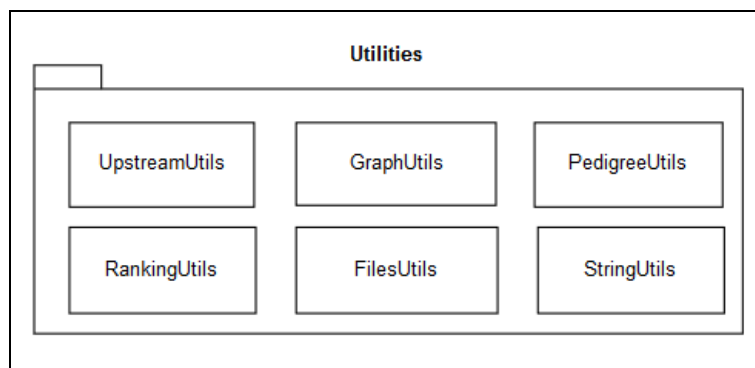
Figure 2.6 Utility functions

After the static and stochastic results are computed as arrays or matrices, specific reports modules are executed to generate CSV files from these arrays and to save the generated reports to disk. Figure 2.7 lists the various reporting classes provided in the calculator.
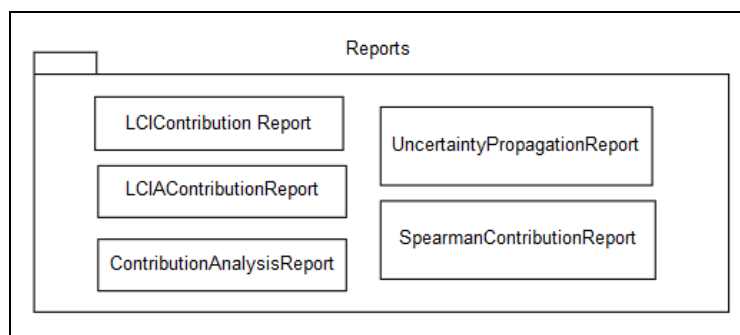
Figure 2.7 Reporting module

2.3 REST Service

The client program (i.e., the front-end) may interact with the proposed LCA calculator through a REST service by sending HTTP GET/POST requests, which allow running a specific calculator functionality (e.g., static LCA, stochastic LCA, etc.).

Upon receiving a request, the REST service will parse its arguments, save its uploaded files, and pass that request to the calculator through the various steps, from step 1 to step 7 as shown in Figure 2.8, until finally returning a response to the calling program as a JSON object containing an array of the generated *CSV* report's HTTP links.

First, in an initialization phase, when the service receives a request for an LCA calculation, it parses the parameters in the HTTP request and creates a *CalculationSettings* class object, which serves as a data bag to be passed to all subsequent function calls. The *CalculationSettings* consists of properties such as the phases that the calculator is required to compute or to skip (i.e., LCIA, Monte-Carlo or Sensitivity), various entities identifiers (i.e., ProjectId, CalculationId, Version, ImpactId, RootProcessId, SolvingMethod, SystemID), and other properties such as Monte-Carlo_Iterations, RootPath, OutputQuantity.
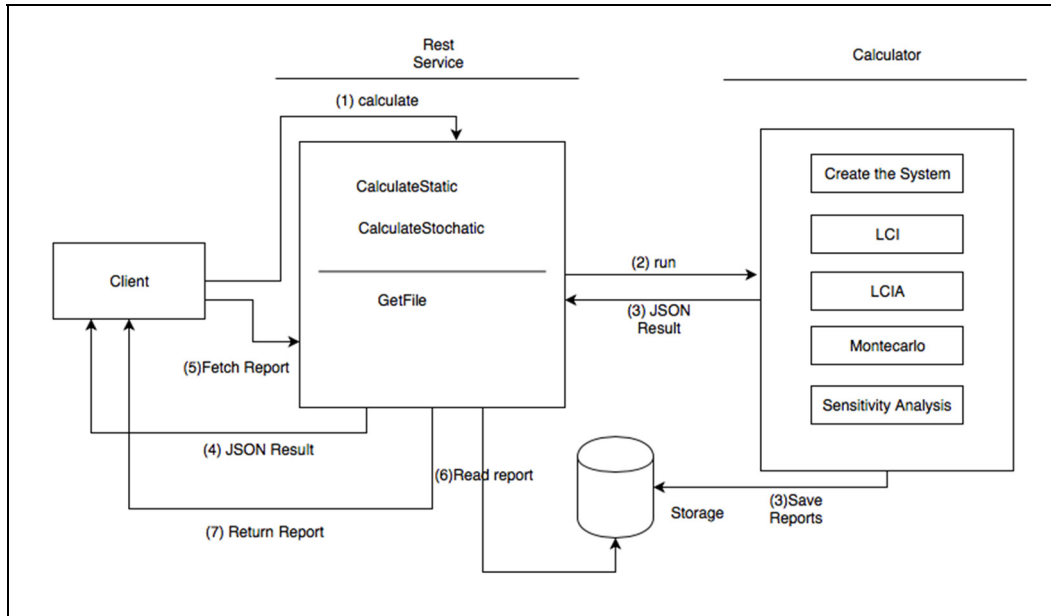
Figure 2.8 Program Flow

Second, the service stores the uploaded differential files in a file structure, shown in Figure 2.9, which consists of a set of folders to organize the storing of the received delta files. After the delta files are stored, the service initiates a call to the calculator to load or update a database in memory, LCADB, with the newly stored delta files as presented in section 2.4.The file structure adopted in ParallelLCA0.1 is described as follows:

- The *DBTemplates* folder contains LCADB templates. A template is a ground database for building an LCA Database. We are currently using a database template extracted from the OpenLCA MySql database;
- The *Projects* folder contains all the projects files. Each project directory contains the different project version files where the delta files are stored. Consequently, to access a delta file, a path of the form "{projected}/DBVersions/{version}/deltas/{delta files}" is adopted;
- The *Calculations* folder contains the results files of each of the calculations. Each calculation directory will contain the generated reports files as well as other files, such as error logs generated during the calculation.
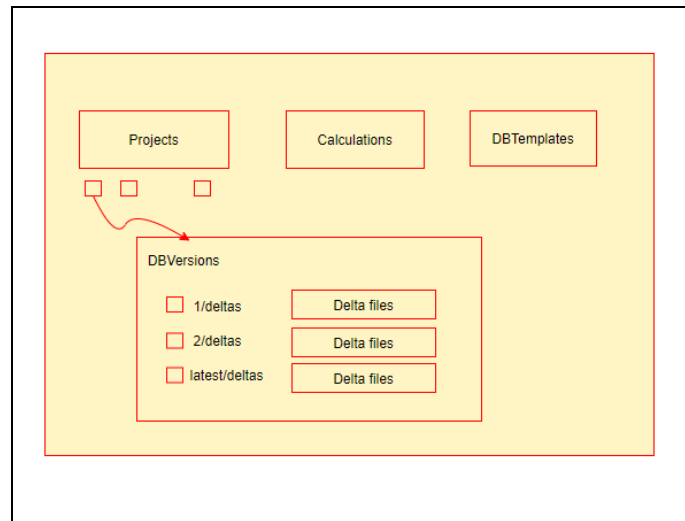
Figure 2.9 Calculator Files Structure

2.4   Parallel LCADB

Before initiating an LCA calculation, the calculator parses the delta files line by line, performs a data processing on each line to generate objects from LCAModels package, and inserts or updates those objects into respective LCADB data structures. The data processing is computed in parallel using a team of OpenMP threads.
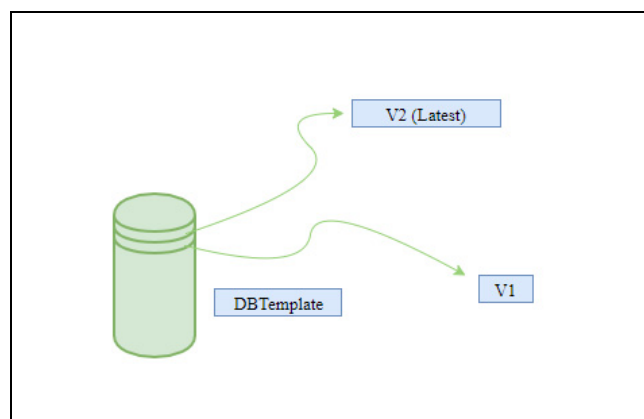
Figure 2.10 LCADB Differential Model

The process of applying different versions of data modifications into LCADB is shown in Figure 2.10. The *DBTemplate* layer is the ground zero layer. The first received delta files

constitute the first layer on top of DBTemplate. Each subsequent new version of delta files comes on top of the previously existing stack of layers.
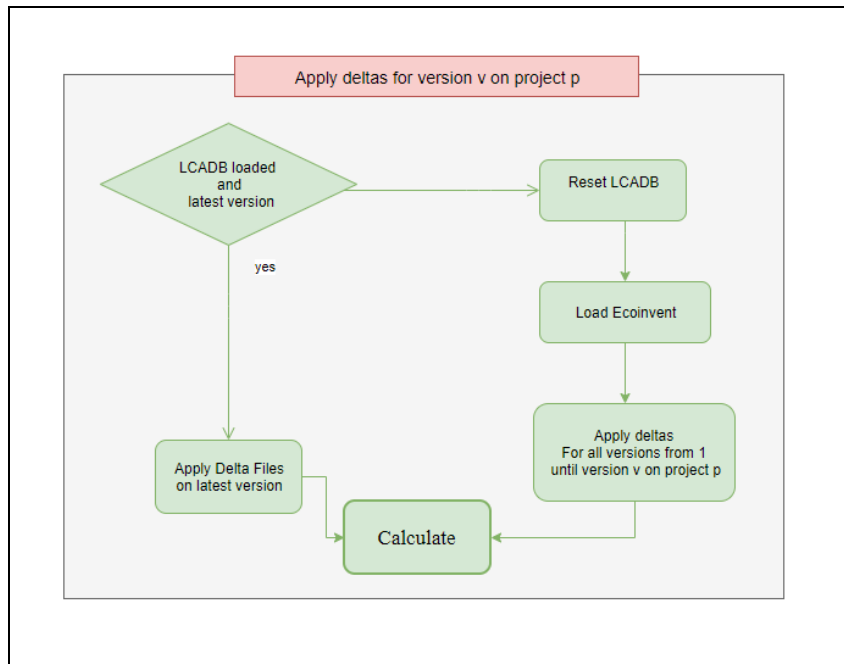
Figure 2.11 Database Loading Workflow

More precisely, as shown in Figure 2.11, if *LCADB* is already loaded with a project version, ParallelLCA applies the received delta files directly on the latest version, and therefore there the reload from the *DBTemplate* can be avoided. If *LCADB* is not loaded, then the algorithm loads *DBTemplate* first and then applies the delta files sequentially starting on top of *DBTemplate*. At any time, if an error occurs, *LCADB* will be reset, and an error message will be returned to the user.

The parsing of delta files into LCADB objects consists of three phases: 1) the building of the objects listed in annex VII from CSV files; 2) the building of indexes for LCADB collections; 3) the building of pedigree-based PDF for the objects.

We designed a parallel algorithm for loading large CSV deltas files such as the exchangesDiff, ProcessesDiff, and FlowsDiff. First, the algorithm loads the large delta file

into memory as a Vector<string>, where each entry represents one line of the file. Second, it launches a team of OpenMP threads and splits large files into a number of chunks equal to the available number of threads. Each thread will be working on a chunk of the large delta file. The processing of each chunk will consist of the aforementioned steps of entity object creation, pedigree-based PDF creation, and indexes loading. Third, when a thread has finished executing, it merges its results into LCADB in an OpenMP critical section.
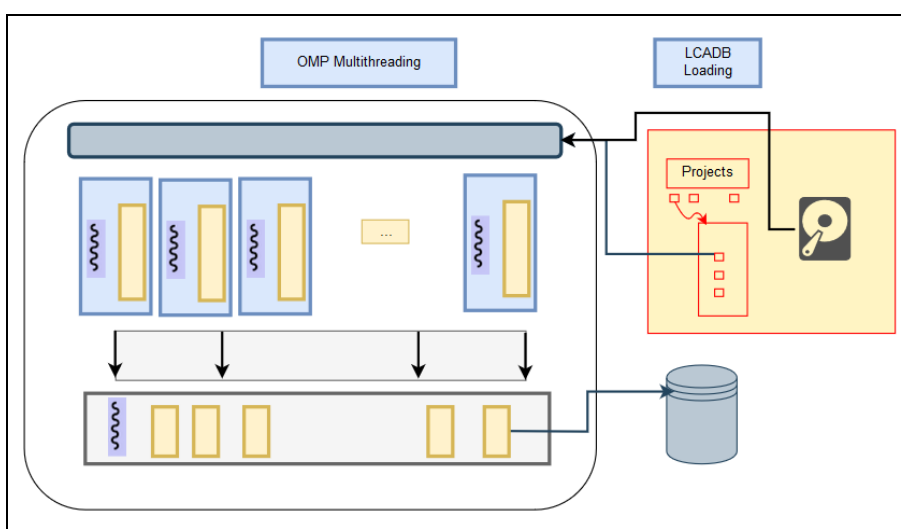
Figure 2.12 Loading LCADB using OpenMP

Table 2.1 LCADB Store Structure

Category	Collection name
Technosphere	map<long, Exchange> exchanges
	map<long, Process> processes
	map<long, Flow> intermediateFlows
Biosphere	map<long, ImpactCategory> impactCategories
	map<long, UnitOfMeasurement> unitOfMeasurements
	map<std::pair<long, long>, CalcImpactFactor> impactFactors
	map<long, long> impatCategoriesUnits
	map<long, vector<long>> impactMethodCategories
	map<long, Flow> elementaryFlows

The program converts the uploaded delta file (in .csv format) line by line into a collection of objects in memory, more precisely in a *Map<Id, Object>*. The Id is of type Long, and the object is saved as a custom class (e.g., *Exchange*, *ImpactFactor*, *Parameter* or *Process*) from the LCAModels package. Table 2.1 shows a list of Maps to store the objects by their Ids.

Table 2.2 LCA Indexes Structure

Index key	Index variable
ProcessId	map<long,vector<long>> in_exch_Map_p
	map<long,vector<long>>out_exch_Map_p
	map<long,vector<long>>elem_flow_Map_p
OutputIntermediateFlowId	map<long, vector<long> > producer_Map_f

The algorithm then computes the indexes shown in Table 2.2, to allow for *Approximatively Constant Time* of objects retrieval from LCADB. More precisely, the index *elem_flow_Map_p* fetches the elementary flows exchanges by *processId* field, the index *in_exch_Map_p* fetches the intermediate input exchanges by *processId* field, and similarly, *out_exch_Map_p* stores output intermediate exchanges by *processId* field. Finally, the index *producer_Map_f* fetches *producerExchanges* by input *exchangeId*.

Table 2.3 Supported Uncertainties

Uncertainty Type	Uncertainty Fields
Lognormal	Geometric mean, Geometric standard deviation
Normal	Arithmetic mean, standard deviation
Triangular	Minimum, Mode, Maximum
Uniform	Minimum, Maximum

Third*,* the calculator computes a PDF (Probability Density Function) for each of the LCADB objects containing uncertain information. The PDF is created by applying pedigree to the provided uncertainty information.

Applying pedigree alters the entities' variance or standard deviation by using the pedigree Matrix v2 in Table 2.3 and equation 2.1 from Jolliet et al. (2010). A separate script was also developed using the equations in the work of Muller et al. (2016b).

$$SD_{g95} = Exp \sum_{1}^{5} \ln^2 u_i + \ln^2 u_b$$

(2.1)

A final step is to construct a Random Number Generator (RNG) object based on the calculated PDF and store it in each object. ParalellLCA supports lognormal, uniform, normal, and triangular uncertainties types. Table 2.3 lists the data fields for each uncertainty type. We are using the "random" package from C++ stdlib to generate RNGs for each distribution type.

2.5   Calculator data loading

The calculator data loading step is where the data is transferred from LCADB to a format that is best suitable for the calculator algorithms. The calculator will use CalculatorData as a temporary object that exists only for the duration of a calculation request. Building the CalculatorData object aims at preparing and storing vectors and dictionaries of entities for later access in the calculation lifecycle. This phase can be executed on each calculation request, or it can be skipped when a calculation request has no changes to the system.

These objects are classified as vectors containing matrices raw data objects, only static raw data objects, only uncertain raw data objects, matrices indexes, and system parameters. These objects are as follows:

1. The *A_exchanges*, *B_exchanges*, *Q_cells* contain all the matrices raw data objects of the current calculation request
2. The *A_foreground_exchanges* and *A_background_exchanges* provides the raw data for the foreground and background layers separately;

3. The *A_static_exchanges* and *B_static_exchanges* contain the static exchanges of the system;

4. The *A_uncertain_exchanges* and *B_uncertain_exchanges* contain the stochastic exchanges of the system;

5. The *parameter* dictionary implemented as a *map<long, Parameter>* maps a parameter id to its Parameter class instance;

6. The LCAIndexes provides indexes for each of the collection of objects in CalculatorData that is a raw data for matrices loading. The first index is a vector containing the object Ids and the second is a map<long, long> that maps each Id to a matrix column or row index.

2.6   Parameters module using EXPRTK

The purpose of the Parameters module is to manage the loading of parameters into memory, the access of these parameters, and the evaluation of mathematical expressions dependent on parameters. This module allows the LCA analysis of a parametrized system of products. One requirement that this module answers is the evaluation of a mathematical formula $F(x_1, x_2, \ldots, x_n)$ whose variables are both static, stochastic, and dependent on other formulas.

The thesis adopted the library EXPRTK (Arash Partow, 2010) for formula evaluation. EXPRTK relies on data structures such as:

1. The *Expression_t*, that represents an expression to evaluate;
2. The *Symbol_t* that represents a parameter and its value;
3. The *Symbol_Table_t* that represents a collection of *Symbol_t* instances.

Figure 2.13 shows the loading process for parameters and expressions. The parameters or symbols are saved in a symbol table, which is an instance of *symbol_Table_t* type and stored in a map *symbolTableMap< key: Long, Symbol_Table_t>* indexed by "scope_owner" as key. Similarly, the expressions are created as instances of *Expression_t* type, and then they are

registered in an instance of *symbol_Table_t* type. Each instance of *expression_t* is stored by the "Scope_OwnerId" key in the *expressionsTableMap* map, which is an instance of *Map<key, Expression_t >*, where the "scope" is either Global or Process and OwnerId represents the Id of the LCA process owning a parameter. The parameters can be stored per LCA process or globally based on this architecture.
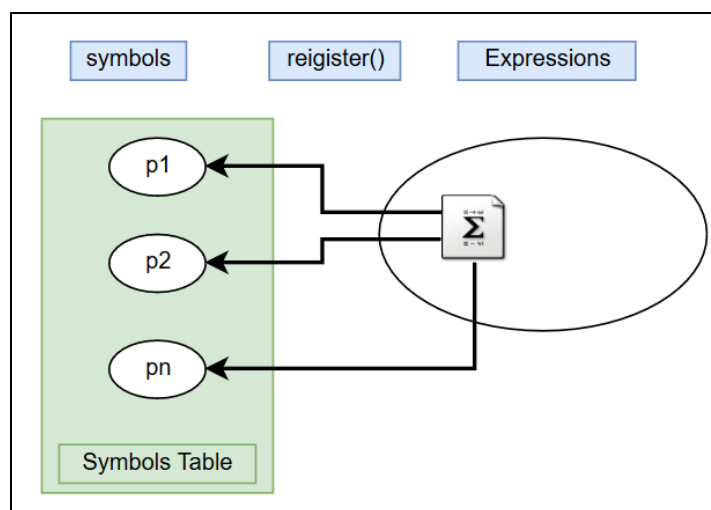
Figure 2.13 Symbols, expression, and symbols Table

When a parameter value is dependent on an expression, the parameter is created as a "*function*" class. When the parameter value is given, it is created as a "*variable*" class or as a "*constant*" class. The evaluation of a parameter expression is achieved in the EXPRTK Eval function either by reading the value of the "*variable*" or "*constant*" based object or by executing the "*function*" class.

The "function" class allows us to define a custom method for the evaluation of expressions that is specific to our research project. When the parameter is stochastic, we use the function *stochasticFunction* that inherits from the "function" class, and that specializes in sampling parameters PDFs based on the corresponding uncertainty information. When the parameter is static and expression-based, we use the function *staticFunction* whose role is to make static parameter expression evaluations. The *staticFunction* class allows for evaluating a chain of expressions (i.e., formulas whose variables are formulas) through recursive.

Our implementation supports the evaluation of parameters with formulas relying on static variables and on stochastic variables. In addition, it allows evaluating parameters with formulas relying on a graph of formulas whose evaluation depends on other static or stochastic variables, as shown in Figure 2.14.
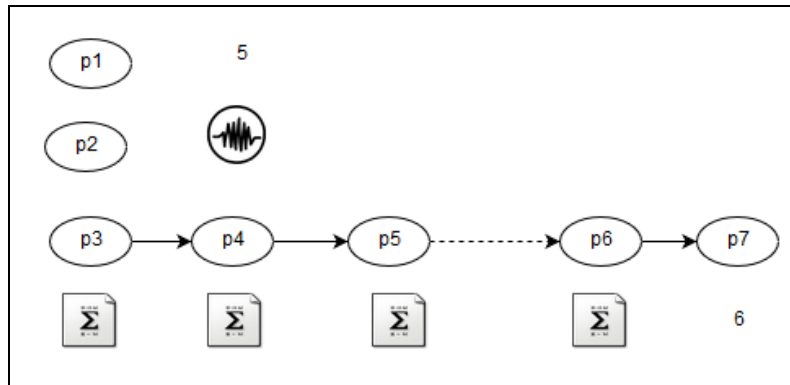
Figure 2.14 Supported expression evaluation

As a result of the supported evaluation modes, when calculating a formula that depends on several other stochastic variables, its independent stochastic variables are first evaluated by sampling, and then the given formula is applied on the generated samples. This evaluation is represented in equation 2.2, where the symbol " ' " represents sampling.

$$F'(x1, x2, ..., xn) = F(x1', x2',..., xn') \qquad (2.2)$$

2.7   Parallel LCA graph building using OpenMP

Our implementation of building the graph **(g)** consists of creating and connecting the client and supply processes, in a layered graph, as shown in Figure 2.15. The resulting algorithm consists of an iterative process that starts at the root node and finishes when it reaches a layer of non-demanding LCA processes. Building each supply layer is implemented by dividing a client layer into chunks of processes that are processed in parallel.

While building (g), an elementary operation or kernel, shown in Figure 2.15, is repeated on each of the LCA activities of each chunk of processes. First, the demands (i.e., input arrows) of each client process are fetched from LCADB, using the indexes in Table 2.2, as a *Vector<InputIntermediateExchange>*. Second, for each of the exchanges fetched in step 1, a list of producers (i.e., supply activities) is computed based on the information available in LCADB and using the indexes in Table 2.2. Third, a unique producer (i.e., red rectangle in Figure 2.16) must be selected for each of the exchanges in step 2. Fourth, a connection (edge) between a parent process and a supply process is created as an Edge object class.
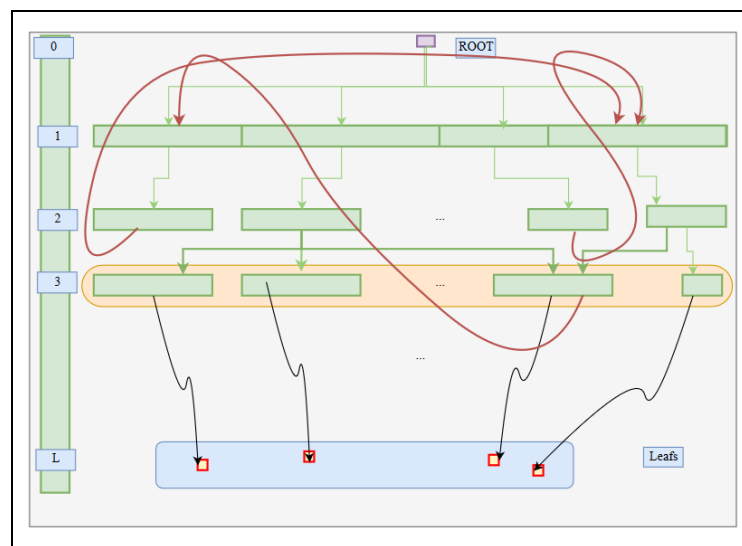
Figure 2.15 Layered Cyclic LCA Graph

The unique producer selection consists of selecting a single producer out of the set of candidate producers fetched from *LCADB* in step 2. As shown in Algorithm 2.1, the program will select a producer that is first the *defaultProvider* of the demanding client activity or second, tagged as *preferred* for the client activity or otherwise has a specific geographical location.

Algorithm 2.1 is designed to use "*#pragma parallel for*" from OMP to build supply layers for large client layers. As shown in Figure 2.17, the algorithm will split each graph layer into batches of activities, then it computes in parallel their producers (i.e., access to LCADB) and

selects unique ones using an OpenMP team of threads. When the parallel threads finish executing, the unique producers of each thread are joined, in a synchronization step, into one vector of unique producers. The step of connecting demand processes to their unique producers is executed sequentially. This process is repeated until the full graph is built.
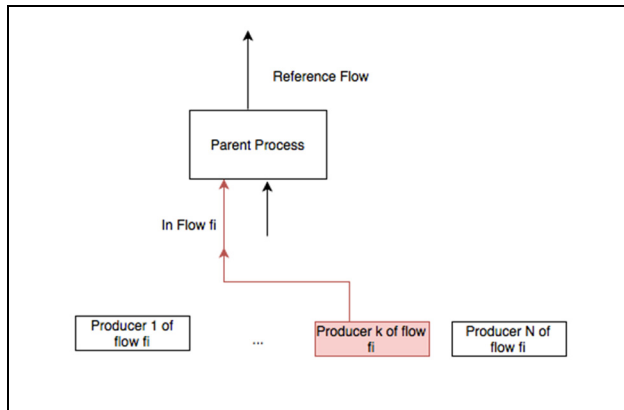
Figure 2.16 Graph building kernel
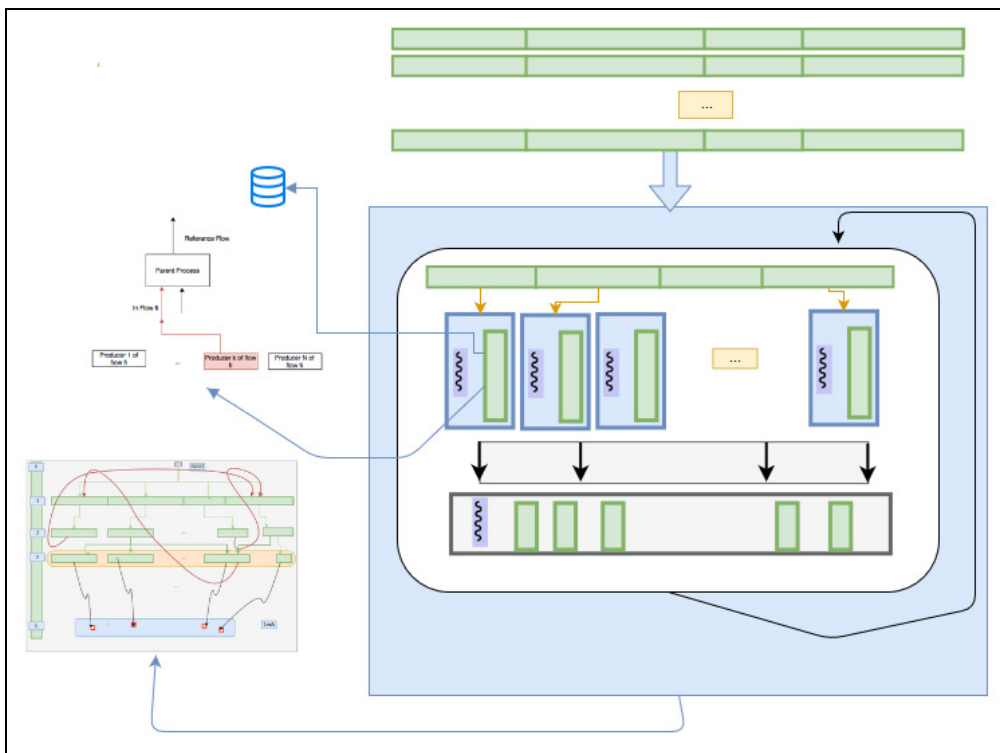
Figure 2.17 Parallel iterative graph building using OpenMP

Algorithm 2.1 Parallel and iterative graph building

Algorithm: Parallel and iterative graph building

   Inputs : demandProcesses as Vector
   Output: (edges , vertices) as Vector

```
1  Function buildGraph(demandProcesses as Vector):
2  foreach demandProcess in demandProcesses do
       Parallel:
3      demandFlows=demandProcess.getDemandingFlows()
4      producers=getProducers(demandFlows)
5      foreach producer in producers do
6          if demandProcess.defaultProvider is equal to the
             producer.ProcessId or producer is preferreType then
7              uniqueProducer = producer
8          end
9      end
10     producersInCountries= filterInContry(producers)
11     if size(producersInCountries) >0 then
12         Return first producer from producersInCountries
13     end
14     uniqueProducer = first(producers)
       End - Parallel
       CriticalSection:
15     uniqueProducers += uniqueProducer
       End - CriticalSection
16 end
17 edges += connect(layer.demands,uniqueProducers)
18 vertices += demand
19 buildGraph(uniqueProducers)
20 return (edges , vertices)
21 End-Function
```

Algorithm 2.1 connects the tuple (*demandProcess*, *uniqueProducer*) by sequentially traversing the *Vector <(demandProcess*, *uniqueProducer)>* and creating a link (i.e., edge) connecting the two nodes *demandProcess*, *uniqueProducer*. When connecting a demandProcess to its supplyProcess, checking is made on whether the producer is already in the graph so that it will be reused; otherwise, a new process node will be created for the supply process. After repeating this step for all the processes of the demand layer, a new layer of supply processes is created and fully connected to the demand layer. The algorithm can use this newly created layer to iterate again to create yet another supply layer. This iterative process is repeated until it reaches a layer with no demand processes.

When the graph is fully built, a GraphData structure will have been built with the properties *edgesListCompact* as a *map<Long, LCAEdgeCompact>* and *nodesInfoMap* as *map<Long, NodeInfo>* representing the edges and the nodes of the graph respectively.

Foreground aware graph building

While building the graph, a set of data structures is created to store the graph nodes and their connections. The building of the graph (g) produces data structures designed to serve as a memory store for later computations. Also, additional data structures are created to describe the foreground-background layer nature of the graph. The graph data structures consist of dictionaries to store the graph information, indexes to store the Ids information of the graph, and dictionaries to describe the barriers connecting the foreground to the background layers.

The graph dictionaries store the list of processes and their interconnections:

1. The nodesInfoMap (i.e., map<long, NodeInfo>) is a dictionary storing information about the processes in the graph. NodeInfo has properties such as processId, scalar, isBackgrounLayer, isBarrier, etc;
2. The edgesListComapct (i.e., map<long, LCAEdgeCompact>) maps an edge id to an LCAEdgeCompact class instance. The LCAEdgeCompact class contains information such as InputExchangeId, ProducerExchangeId, ProcessSrcId, ProcessDestId, InputCalculatedValue, OutputCalculatedValue, InputScalar, OutputScalar, and FlowId.

The graph Indexes store the list of processes residing in either the foreground layer, background layer or in any of their barriers:

1. The fronttLayerNodes (i.e., vector<long>) contain the process ids of the foreground layer processes;

2. The backgroundLayerNodes (i.e., vector<long>) contain the process ids of the background layer processes;

3. The frontLayerBarrierNodes (i.e., vector<long>) contain the process ids of foreground barrier processes;

4. The backgroundLayerBarrierNodes (i.e., vector<long>) contain the process ids of background barrier processes.

The graph barrier dictionaries fully describe the foreground and background barriers:

1. The foregroundNodesScalars (i.e., map<long, double>): map a foreground barrier process to its scalar;

2. The barrier_demand: maps each foreground barrier process to its demand connections from the background layer.

> **Inputs :** supplyExchange, demandExchange, demandProcess,
>   supplyProcess
> **Output:** Nothing
>
> 1 Assign outputflowvalue= Evaluate(suuplyExchange)
> 2 Assign inputflowvalue= Evaluate(demandExchange)
> 3 **If** demand and supply processes are front layer **Then**
> 4 $supplyScalar = \dfrac{inputflowValue * demandProcess.Scalar}{outputflowvalue}$
> 5   verticesScalar[supplyProcess.id] += supplyScalar
> 6 **end**

Figure 2.18 Solving kernel on establishing a connection

While building the graph, the foreground layer can be solved in the step of connecting a demanding process to its supply process. As shown in Figure 2.18, the solving kernel section consists of scaling the supply process scalar to satisfy the total demand. When the graph is built, the foreground would have been solved (i.e., scaled) to satisfy the functional unit.

2.8   Calculator Kernel

2.8.1   Loading the Matrices: A, B, and Q and their indexes

After the graph (g) is built, an *LCAGraph* object is returned, and the loading of matrices *A, B, Q,* and the demand vector *f* can start.

Our proposed algorithm for building the matrices consists of three steps:

1. The Building, for each of the matrices *A, B,* and *Q,* of two kinds of collections: An Index of ids as a *Vector<Id: Long>* and a corresponding Index Indices dictionary as *Map<Id, IdPosition: Long>* which map each id to a matrix row or column position. This dictionary structure will be accessed when assigning values in corresponding matrices cells;

2. The Fetching of the matrices raw data from *LCADB* as a *Vector<(rowIdPosition, columnIdPosition, object)>* or for brevity *Vector<(i, j, object)>*. The *object* represents either *Exchange* (i.e., A or B) or *ImpactFactor* (i.e., Q) entities;

3. The Assigning of *(i, j, object.value)* into matrices cells *(i,j)*. This step involves aggregating cell values pointing to the same position (i, j).

The demand vector *f* is created as a sparse vector that contains the functional unit quantity, at the position of the processId, of the functional unit process (i.e., root node in the graph (g)).
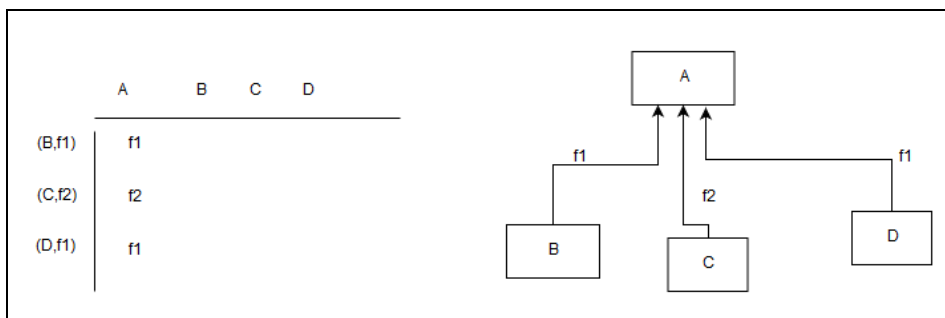
Figure 2.19 matrix A building

When loading matrix A, the rows should correspond to unique tuples of (process, flow); the flowId cannot be used alone otherwise we will get a rectangular matrix as in the case of the example shown in Figure 2.19.

2.8.2 LCA activities scalars computing

At this point in the calculation, all the objects needed for solving equation 1.3 are built, and therefore, the computation of these equations can start. We implemented the system solving equation 1.3 in three methods:

1. Direct solving: using Umfpack, as introduced in section 2.11;
2. Iterative solving: using Eigen++ BiCGStab and ParallelColt CGS, discussed in section 2.11;
3. Hybrid Solver: By solving the foreground layer sequentially and using a Matrix Method (e.g., BiCGStab from Eigen++) to solve the background layer, discussed in section 2.10.

We also experimented with implementations for computing the matrix inverse, adopted by OpenLCA, such as the solving of $A\,A^{-1} = I$ to get A^{-1}, and the use of BLAS primitives to compute the matrix inverse A^{-1}.

2.9 Foreground-Background LCA: Optimization for large systems

The use of the foreground layer (acyclic) connected to processes from the background layer (cyclic) is widespread in LCA systems. We model this pattern using: 1) foreground layer barrier (i.e., yellow band), 2) background layer barrier (i.e., blue band) and 3) the connections of the foreground layer barrier processes to background layer barrier processes.
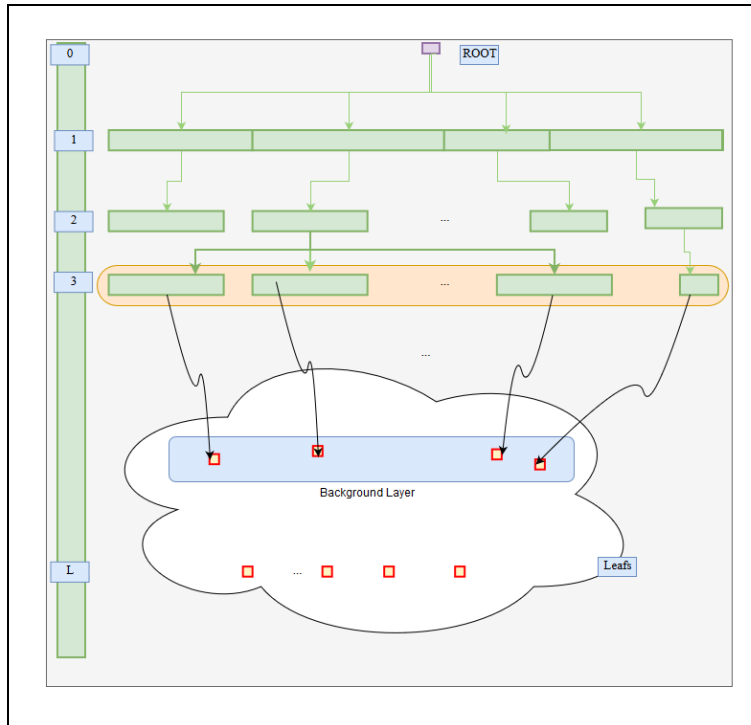
Figure 2.20 Foreground-Background layers connection

Computational LCA should take into consideration the foreground-background layers separation pattern because it allows for the design of several efficient algorithms for system solving by reducing or eliminating the matrix component. We are using this separation patterns in two places in our research:

1. Algorithm 2.2 which solves the foreground layer using graph traversal reducing the matrix component of equation 1.3 to the dimensions of the background layer;
2. An extension of algorithm 2.2 is the algorithm 2.9, which aggregates the background layer and thus allows the complete removal of the matrix component.

2.10 Hybrid Algorithm for solving the Foreground-Background layers

As discussed in section 1.1, there are two methods used to solve LCA systems: the sequential iterative method and the Matrix Method. In this section, we present an equivalency model that serves as a foundation for a hybrid solving algorithm which we are proposing for LCA

systems. This algorithm solves the foreground layer of the graph, which is the large part, using a graph traversal approach (i.e., *Sequential Method*) and the background layer of the graph using the Matrix Method.
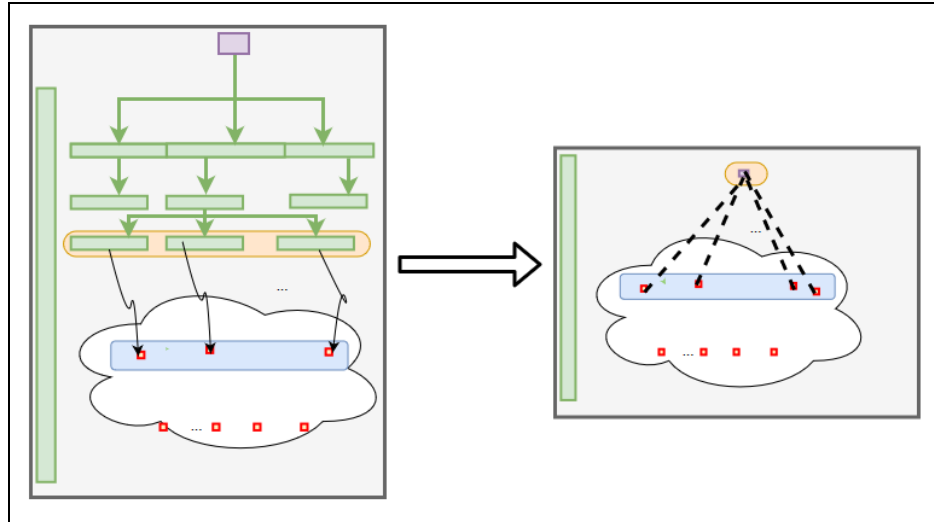
Figure 2.21 Foreground layer collapsing – equivalent LCA system

The *Hybrid Solver* in algorithm 2.2, will solve the system in equation 1.3 by:

1. First, scaling the foreground layer processes and collapsing it to a single node which we call the *Collapsed Foreground*. The *Collapsed Foreground* has demands equivalent to the total demands that the foreground layer requires from the background layer;

2. Second, it loads a new matrix $A_{(CollapsedForeground + BackgroundLayer)}$, containing the background layer and the collapsed foreground layer single node as shown in Figure 2.22;

3. Third, it solves equation 1.3 by replacing matrix A with $A_{(Collapsed\ Foreground + BackgroundLayer)}$ to get the scalars of the background layer;

4. Fourth, it loads the background layer elementary flow exchanges in matrix B, and finally, calculates LCI and LCIA using equations 1.4 and 1.5.

$$N_{\text{collapsed model}} = N_{background\ layer} + 1 \qquad (2.3)$$
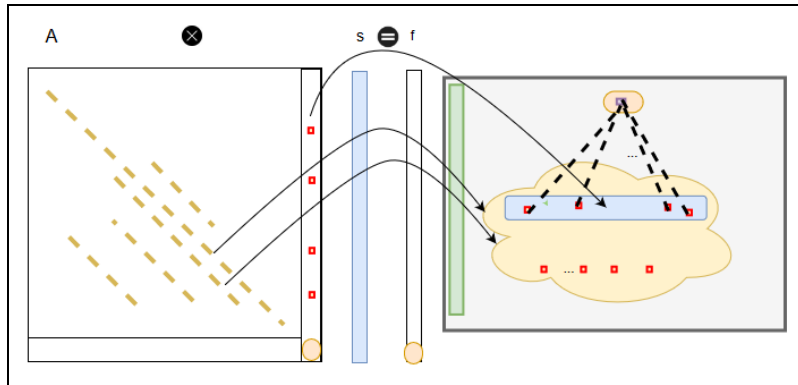
Figure 2.22 The new matrix A (Collapsed Foreground + *Background Layer*)

As a result of this method, the graph in the *Collapsed Foreground* model, in Figure 2.22, is translated into a constant number of equations and unknowns, as shown in equation 2.3. A foreground layer with an increasing size will not affect the matrix component in the calculation as a single node always represents the foreground layer.

Algorithm 2.2 Hybrid solving algorithm.

collapsing the foreground layer

Algorithm: Foreground solving algorithm by Foreground Layer collapsing

Inputs : parentRootId:Long, functionalUnit :Decimal
Output: scalars: Map [(processId, scalar)]

```
1  scalars[parentRootId] = functionalUnit
2  foreach  layer in graph.Layers.where(layer>1) do
3      foreach  supplyProcess in layer do
4          demands= demands[layer-1][supplyProcess]
5          foreach  demandProcess in demands do
6              Solving Kernel:
7              totalDemand= unitaryDemand * demand_scalar
8              outputSupply = supplyReferenceFlow
9              newScalar= (totalDemand / outputSupply)
10             supplyScalar = supplyScalar + newScalar
11         end
12         scalars[supplyProcess] = supplyScalar
13     end
14 end
15 Load the new matrix Acollapsed
16 Solve Acollapsed s_(background+collapsedforeground) = f
17 Load background layer elementary flows exchanges into B
18 Calculate LCI and LCIA using equations 1.4 and 1.5
```

2.11   Matrix-based solving scripts

2.11.1   Direct and iterative sparse system solving

Solving the LCA system of equation 1.3 is implemented in two methods: one method consists of loading the whole system into a matrix, the technological matrix *A*, and solving equation 1.3. This method is presented in this section. A second method consists of using a combination of the *Sequential Method* and the *Matrix Method,* which we call the *hybrid solver*. The *hybrid solver* is presented in section 2.10.

Two state-of-the-art libraries for sparse matrix solving, or sparse matrix inverse computation, are being adopted. The first, using *UMFPACK*, implements a *direct* sparse solving approach, the second, using *Eigen++ BiCGSTAB*, implements an *iterative* sparse solving approach.

For UMFPACK, the solving script first transforms the raw exchanges data into the CSC (Compressed Sparse Column) format consisting of three arrays of rows: offset Ap, column indices Ai, and values Ax. Second, it factorizes the matrix in an analyze phase. Finally, it will find the system solution in the "*solve"* phase of UMFPACK. A sample code used for Umfpack is provided in Annex VI.

```
1  void bicgstab(int dim, int n, long *rowsA, long *colsA, double *dataA,
2                         double *b, double *x)
3              {
4                  SMatrix m(dim, dim);
5                  fillSparseMatrix(&m, n, rowsA, colsA, dataA);
6                  VectorXd xVec(dim), bVec(dim);
7                  for (int i = 0; i < dim; i++)  bVec(i) = b[i];
8                  BiCGSTAB<SparseMatrix<double> > solver;
9                  solver.compute(m);
10                 solver.setTolerance(0.0001);
11                 xVec = solver.solve(bVec);
12                 for (int i = 0; i < dim; i++) x[i] = xVec(i);
13             }
```

Figure 2.23 Eigen++ *BiCGSTAB* solving implementation.

Based on Gaël and Benoit (2017)

For the Eigen++ *BiCGSTAB*, the calculator will traverse the matrices raw data and generate three arrays with the COO sparse format: the *RowIndices*, the *ColumnIndices,* and the *values* arrays. Those arrays are fed as parameters into the function in Figure 2.23. The function will first build the matrices from the COO arrays, and then an *analysis* phase will be performed, followed by a *solve* phase that will solve the system iteratively.

2.11.2   Matrix inverse experimentation

The literature highly recommends against computing the matrix inverse and encourages finding alternatives for it. Our research did not implement scripts for computing the matrix inverse, but instead, it developed an algorithm to avoid computing the matrix inverse. However, the research has experimented with the code developed by OpenLCA to compute the matrix inverse, including:

o   The solving of $A\,A^{-1} = I$, which requires solving a set of linear systems of size equal to the number of processes in the graph (GreenDelta, 2017b);
o   The use of the low-level BLAS primitives for matrix inverse (GreenDelta, 2017a).

2.12   LCI and LCIA vectors

2.12.1   The Traditional Matrix Method

This method is essential for cases where the background layer changes between calculation requests. After computing the scalars vector *s*, the calculator will proceed to calculate the inventory vector *g* and the impact scores array *h* using a matrix product as shown in equations 1.4 and 1.5.

We have implemented matrix multiplication using the methods below:
1. Multithreaded matrices multiplication using Parallel-Colt;
2. Distributed Spark-based matrices multiplications;

3. Vectorized SIMD-enabled matrices multiplication using Eigen++ which provided the best performance for the examples in this project.

2.12.2 The Hybrid computing of LCI and LCIA

This method consists of collapsing the foreground layer and analyzing LCA based on a new system, the *Collapsed Foreground*, explained in section 2.7. The matrix component in equation 1.4 and 1.5, using this method, is reduced to only include the background layer plus one node representing the *Collapsed Foreground* layer.

2.12.3 The Hybrid-Aggregation method

This method is beneficial for cases where the background layer is not changing, and its details can be omitted from the LCA reports. With this method, when a request for calculation hit the calculator, only the foreground layer get solved, and the background layer unitary inventories and scores are read from memory.
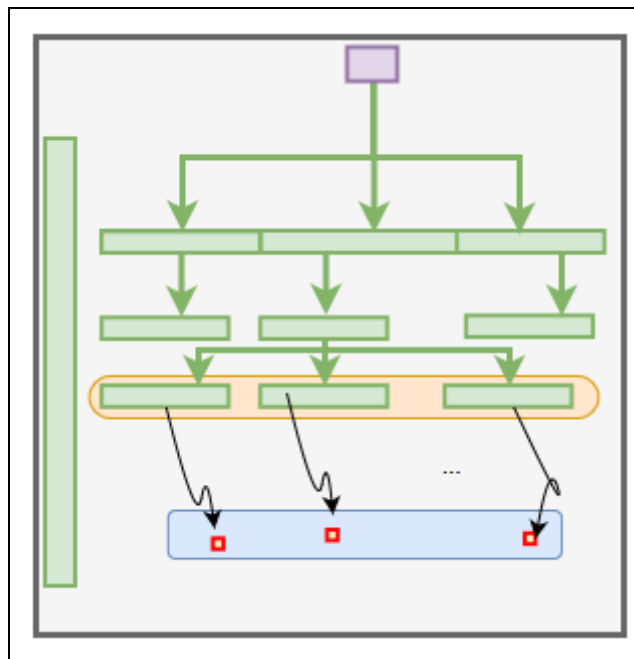
Figure 2.24 Aggregated background layer.

The method, in a first initialization step (i.e., not during the calculation), computes and stores the unitary inventory and scores of the individual background layer processes, as shown in equation 1.7 and 1.8. In a second step, when the foreground layer is solved, the scalars of the foreground barrier and the unitary demands of this barrier are read to compute the total inventory vector g and total impact scores h as shown in equations 2.4 and 2.5.

$$h = QBA^{-1} * Scalar_{FgBarrierActivity} * demand_{FgBarrierActivity} \qquad (2.4)$$

$$g = BA^{-1} * Scalar_{FgBarrierActivity} * demand_{FgBarrierActivity} \qquad (2.5)$$

Algorithm 2.3 LCIA scores for the collapsed-aggregated system

Algorithm: LCA scores fore the collapsed-aggreagted system

   Inputs : FGBarrier, demandsMap, backgroundBarrierScores
   Output: lcia: Vector of Double

1 backgroundBarrierScores = Read(/pathTo/BGBarrierScores)
2 Solve the foreground layer sequentially // algorithm 2.2, lines 1-15
3 **foreach** *process in FGBarrier* **do**
4   **foreach** *demandSupply in demandsMap[process]* **do**
5   **foreach** *impactCategory in impactMethod.impactCategories()* **do**
6   Lcia[impactCategory]+=
                backgroundBarrierScores[demandSupply.supplyId] *
                demandSupply.demandScalar *
                demandSupply.demandAmount
7   **end**
8   **end**
9 **end**
10 **return** lcia

As shown in Figure 2.25, the foreground layer is collapsed to a single node, and the background layer is represented only by the processes to which the foreground layer connects. The impact scores of this new system are computed using the algorithm 2.3 by aggregating the scores over the two layers of supply and demand.
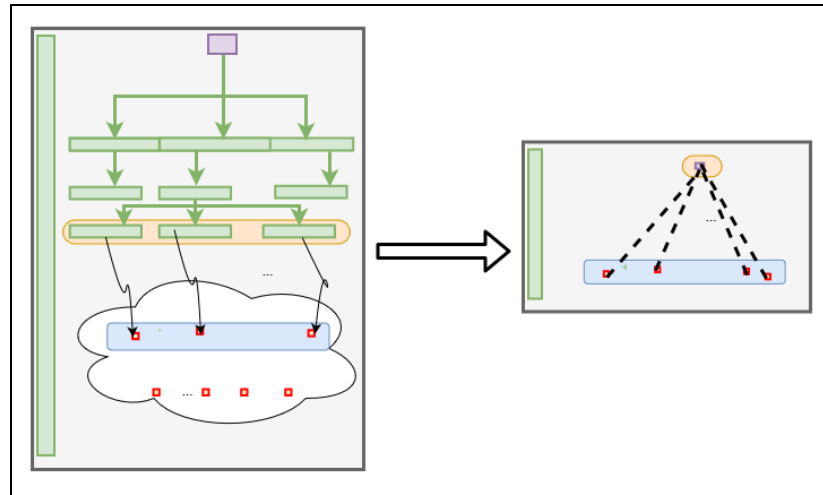
Figure 2.25 Collapsed-Aggregated LCA system

2.13   Aggregate upstream computation

2.13.1   Aggregate upstream reformulation: matrix inverse avoidance and dimensionality reduction

Figure 2.26 shows a reformulation of aggregate LCA given by equation 1.8, in which the matrix inverse is removed. Instead of computing matrix inverse, which is very expensive, the upstream quantities can now be calculated by solving a set of sparse linear systems as in equation 2.6.

$$x = QBA^{-1} \rightarrow$$
$$x^t = (A^{-1})^t \, (QB)^t \rightarrow$$
$$x^t = (A^t)^{-1} \, (QB)^t \rightarrow$$
$$A^t x^t = A^t \, (A^t)^{-1} \, (QB)^t = I \, (QB)^t = (QB)^t \rightarrow$$
$$A^t x^t = (QB)^t$$

Figure 2.26 Aggregate LCIA Re-formulation.

Matrix inverse removal

Solving the system in equation 2.6 is equivalent to solving a linear system with multiple RHS (Right Hand Side), which in turn can be computed by solving the LHS (Left Hand Side) for each column in the RHS independently (i.e., in parallel) and then stacking each obtained solution vector in a shared 2D matrix variable.

The transformation to equation 2.6 provided two significant optimization opportunities. First, the number of systems to solve in equation 2.6 is equal to the number of impact categories required in the calculation (e.g., 18 impact categories for Recipe midpoint 2008 (I)). Second, these independent systems can be solved in parallel.

$$A^t x^t = (QB)^t \tag{2.6}$$
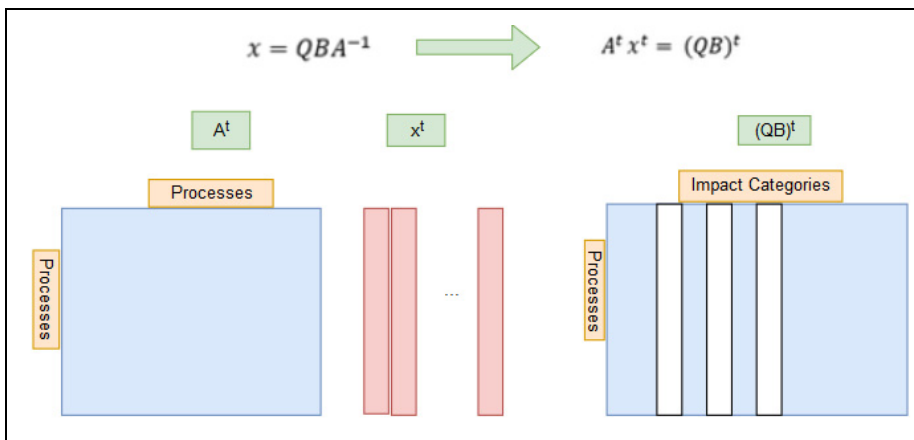
Figure 2.27 Upstream aggregation optimization.
Eliminating the need for matrix inverse computation

The number of impact categories is the smallest dimension in an LCA problem. The dimensions in LCA problems are, from the largest to the smallest as follows: 1) the number of intermediate exchanges in the supply chain, 2) the number of supply chain processes, 3) the number of elementary flows that are exchanged with the biosphere, and 4) the number of impact categories to analyze. Therefore, equation 2.6 reduces the problem dimensions proportionally to the number of impact categories we are studying.

Consequently, this dimensionality reduction is reflected as a reduction in the number of threads and cores needed in the parallelism of solving the independent linear systems. If we are to parallelize $A\,A^{-1} = I,$ we will need a number of threads that is equal to the number of LCA processes in the graph (g). However, in our method, and because of equation 2.6, we need a degree of parallelism equal to the number of threads in the system.

2.13.2   Parallel aggregate upstream: MUMPS-MPI and Eigen++-OMP

o   Using Eigen++ BiCGStab and OpenMP:

To parallelize the tasks of solving the aforementioned independent systems in section 2.13.1, we have implemented a parallel version that computes equation 2.6 using OpenMP. In this implementation shown in Figure 2.27, the linear systems solving tasks are executed in parallel in an OpenMP FOR loop, and the stacking of the solution columns are computed sequentially. The solving of the system for each column is computed using Eigen++ BiCGStab.
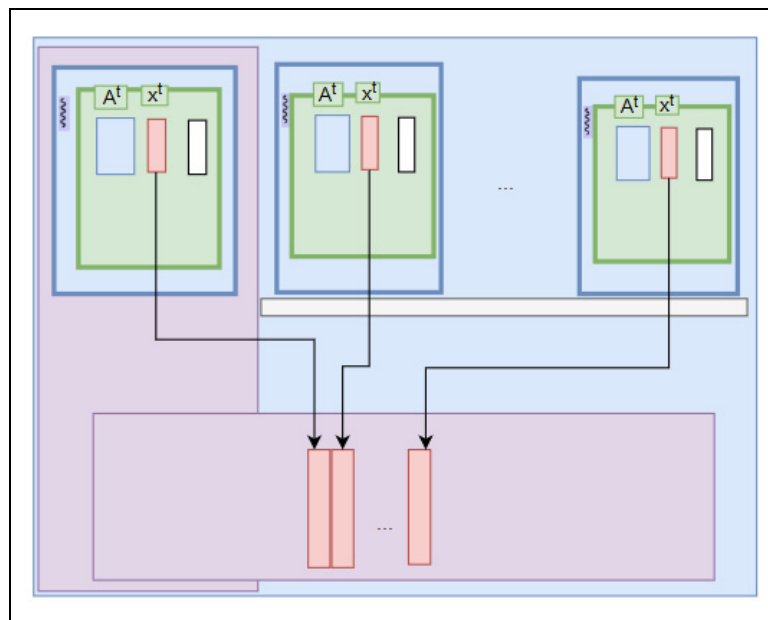
Figure 2.28 Parallelizing QBA^{-1} using OpenMP

In Figure 2.28, each green box is solving equation 2.6 for one of the columns in its RHS. When the different solvers finish their work, they join their solutions in a shared 2D matrix variable that will give x^T. A transpose is applied to x^T, which gives the solution matrix x.

o   Using MUMPS transpose solving and MPI:

MUMPS allows for the solution of equations similar to equation 2.6 with the form $A\,x\ \ = \ B^t$ and with special configurations. The MUMPS solver starts by initializing a data structure representing the current operations as a *"DMUMPS_STRUC_C id."* The *"DMUMPS_STRUC_C"* data structure allows accessing configuration flags to set up various properties of the calculation. The configuration starts by setting up id.job as -1 to indicate an initialization phase, id.par to indicate serial or parallel computing using MPI, and id.sym=0 for unsymmetric solving. At this step, the configuration is sent to the MUMPS using *"dmumps_c(&id)."*

Next, an ordering algorithm needs to be chosen among metis (5), port (4), AMD (0), AMF (2), and QAMD (6). We chose the Metis ordering. As we are computing *QBA⁻¹, id.icntl[8]* needs to be set to "0". To specify a sparse RHS, the flag *id.icntl(20)* needs to be set to "1".

After the initialization phase, we started by loading the LHS and RHS. The LHS is loaded in coordinate format and the RHS in CSC (Compressed Sparse Column). Finally, a solve command is triggered by setting *id.job* to "6". The result will be x^t which needs to be transposed to get the aggregated scores or QBA^{-1}.

2.14   Process contribution to LCI and LCIA

In addition to the inventory and impact scores arrays, the calculator generates analytical reports that allow assessing the process contribution information.

The LCI – Contribution report provides the contribution of each process in the LCA graph to the total inventory of each elementary flow. To generate the LCI - Process Contribution report, the prototype proceeds first to aggregate (i.e., grouping followed by summing) the elementary flows exchanges of each process by (*processid, flowid*) which yields a vector of *<(processid, flowid), aggregatedValue>*, then the resulting *aggregatedValue* is multiplied by the process scalar which yields the contribution of each process to each elementary flow *<(processid, flowid*), TotalProcessContribution>*.

The LCIA – Contribution report provides the contribution of each process in the LCA graph to the total impact scores. The process-to-flow contribution represented by *<(processid,flowid), contribution>* is mapped into an environmental impact category by multiplying the *contribution* value of a given category *characterization factor* to yield *<(processid,impactid), CharacterisedContribution>*. Second, the characterized contribution is then aggregated by *(processid, impactid)* to yield the total impact contribution of a process to a particular impact category represented by *<(processid,impactid), TotalCharacterisedContribution>*.

2.15   Parallel LCIA upstream for multiple impact targets using OpenMP

The upstream LCIA report provides, for a given process *p***,** the cumulative contribution of all the processes in its supply chain. Conceptually, computing the upstream quantities of a process "*p*" and an impact category *"c"* consists of traversing and aggregating the impact scores of "*c*" on all the upstream paths originating from the leaf nodes of the graph and arriving at the process *p*. This approach works well for an acyclic graph. However, it has a significant computational challenge when applied to the background layer because of the presence of feedback loops, which makes the upstream traversal circular.

Our approach to computing this report will proceed as follows, similar to the OpenLCA implementation (GreenDelta, 2017c):

1. Compute the upstream quantities for all the processes and all the impact categories of the given impact method;
2. Next, the cyclic graph needs to be transformed into an acyclic graph (or network) as shown in Figure 2.30, to meet the requirement presented in section 2.1.2;
3. The upstream quantities computed in step 1 will be applied to the acyclic graph computed in step 2. A final phase is the rendering of the acyclic graph with upstream quantities in a CSV file.
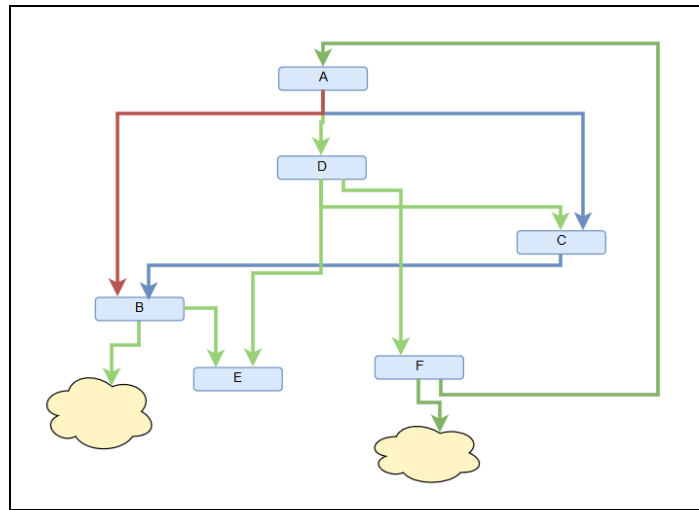
Figure 2.29 Cyclic LCA Graph

We followed the same methods implemented in OpenLCA to get the same results, which is a requirement. However, the following sections will show how we implemented these methods using tools and optimizations, which significantly enhances performance.

2.15.1   Computing cyclic upstream

A matrix-based formulation that computes the upstream quantities is presented in (Heijungs and Sun, 2002) and (Qin and Suh, 2017) and shown in equation 2.6. Equation 2.6 contains a matrix inverse operation, which is very expensive. We designed an algorithm that will compute the same aggregate upstream but using a different formula presented in equation

2.6. Also, we have parallelized the computation of equation 2.6 using OpenMP and MPI, as explained in section 2.13.2.

2.15.1.1 Hybrid aggregate upstream

The matrix-based upstream implementation tends to give an exponentially increasing time delay when tested with large foreground layers. The Hybrid aggregate algorithm, presented in this section, comes to solve this problem. It uses the Matrix Method for the background layer portion of the graph and uses reverse graph propagation of LCIA scores for the foreground layer.

Algorithm 2.4 Upstream aggregate LCIA by reverse graph propagation

Algorithm 2.4 is an iterative algorithm, which starts from the foreground barrier layer and aggregates the LCIA scores per process while moving in the upstream direction toward the root process.

The scores of the background layer, using this method, are first computed (lines 1 to 5) and then propagated through the foreground layer barrier to reach the root node (lines 6 to 19). When on the foreground barrier, the upstream of the background barrier processes is transferred to its process in line 13.

2.15.2   Cyclic to acyclic graph transformation

In this section, we describe the step of transforming the cyclic graph (g) into an acyclic graph. The acyclic target graph consists of a graph of linear branches with no cyclic connections. Algorithm 2.5 has been designed to transform the cyclic graph, which is built in section 2.71 into an acyclic graph, as shown in Figure 2.30.
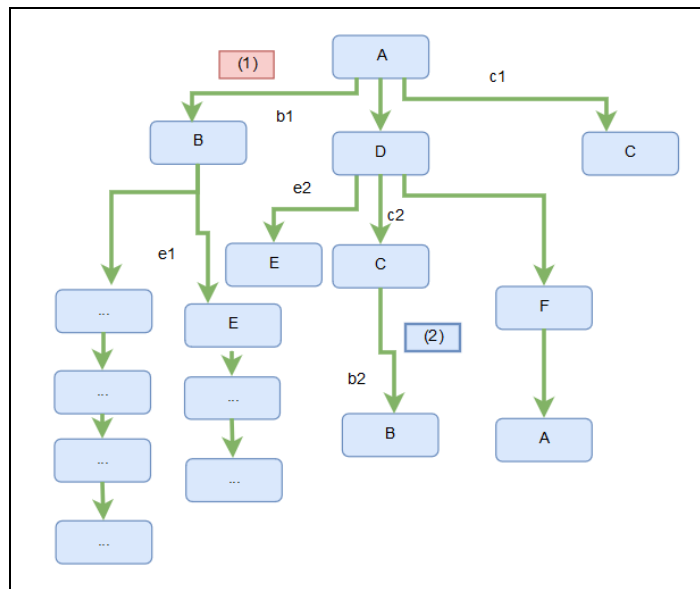
Figure 2.30 Non-repetitive Acyclic Network

Algorithm 2.5 iterates over the graph processes starting from the left side of the graph and going in the downstream direction. It has two major concepts:

o Non-cyclic linking: When creating a link to an existent process in the acyclic graph, a copy of the existing process node is added to the graph;
o Non-repetitive expanding: To meet the requirement in section 2.1.2, if the same process is met for a second time, then it will be included in the graph, but it will not be expanded for a second time.

Algorithm 2.5 Acyclic non-repetitive graph building

Algorithm: Cyclic to acyclic non-repetitive graph building
Inputs : rootNode
Output: Vector [NodeInfo]

```
1  tmpStack+= rootNode
2  while tmpStack ≠ empty do
3      childs=createChildrens(stack.front)
4      foreach child in childs do
5          if child ∉ handled then
6              nodesList+=child
7              tmpStack +=child
8              handled[child]=true
9          else
10             nodesList+=child
11         end
12     end
13 end
14 return nodeList
```

Line 5 of algorithm 2.5, checks whether the node is already added to the acyclic graph. In case it is already handled (i.e., added previously to the graph), a new copy of that process will be added to the graph in line 12; otherwise, it will be added, in line 7, to a queue for further processing.

2.15.3   Acyclic upstream LCIA

The total upstream on the acyclic graph can be given by equation 2.7 below:

$$TotalUpstream_{(p,c)} = UnitaryUpstream_{(p,c)} * Scalar_p \qquad (2.7)$$

To transform the total upstream on the acyclic graph to a total upstream, the mathematical development of equations 2.8, 2.9, and 2.10, taken from GreenDelta (2017c), would need to be applied.

The share of a branch connecting a supply node B to a demand node A is equal to the fraction of supply node B that it is responsible for.

$$Share_{nd} = \frac{scalar_{child} * supply_{child}}{scalar_{parent} * demand_{parent}} , \qquad Share_{root} = 1 \qquad (2.8)$$

The *CompoundShare* of a given node *Nd* is a quantity that results from multiplying all node shares from the root node until reaching the *Nd*. A compound share gives the fraction of supply that a branch segment is providing.

$$CompoundShare_{node=E} = Share_{(root=A)} * b1 * e1 \qquad (2.9)$$

The total upstream of a branch segment would be equal to multiplying the segment fraction (i.e., *CompoundShare*) by the *TotalUpstream* of a process p.

$$TotalUpstream_{(nd,c)} = TotalUpstream_{(p,c)} * CompoundShare_{(p \to nd)} \qquad (2.10)$$

This arithmetic calculation must be applied on each node in the acyclic network and for each impact category of the given impact method. We designed a parallel algorithm that will apply the aforementioned formulas for the different impact categories, as shown in Figure 2.31.

Each thread will apply the formulas on the acyclic network for a chosen impact category and will generate and save a corresponding CSV file.
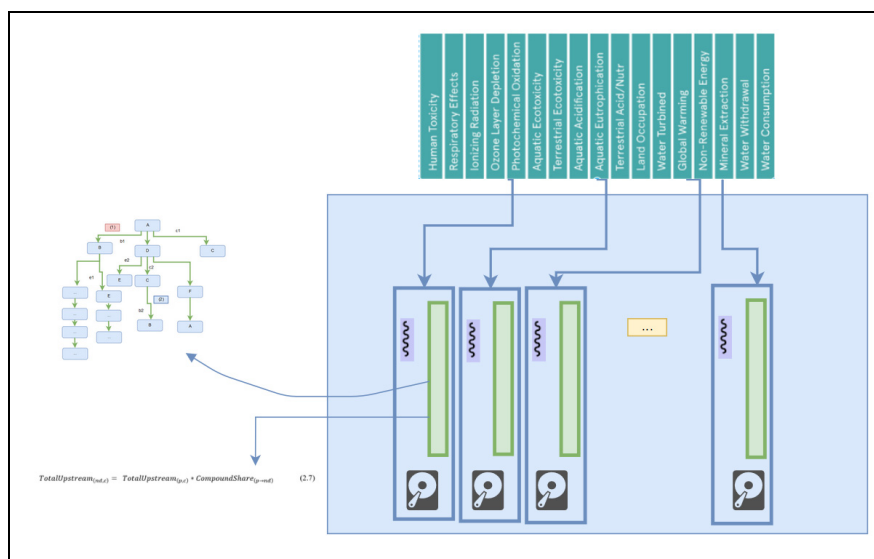
Figure 2.31 Parallel upstream aggregated hierarchical graph

2.16   Stochastic prototypes

Figure 2.32 presents the architecture of the stochastic module. The call to run the stochastic reports originates from a "libStochastic" class. If OpenMP is to be used, the call is routed directly to the respective class. If MPI is to be used, the call needs first to pass through a "main" class compiled as a computer process.
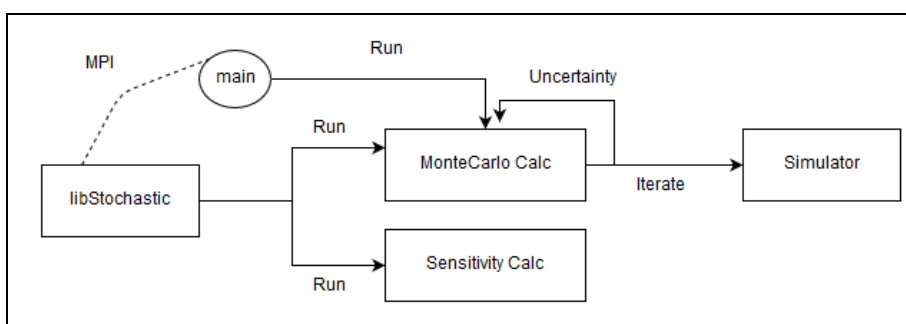
Figure 2.32 Stochastic module architecture

2.16.1   Main Features of the Prototypes

The research designed and developed a variety of prototypes for computing Monte-Carlo and GSA. These prototypes are built around four main types of features described below.

A. Parallelism and Data Exchanges

The simulation is split into a batch of iterations executed in parallel. The iterations inside each batch run sequentially. The parallelism is provided by either using the OMP PARALLEL FOR loops feature or by using the multi-processes MPI (i.e., master-slave topology) feature.
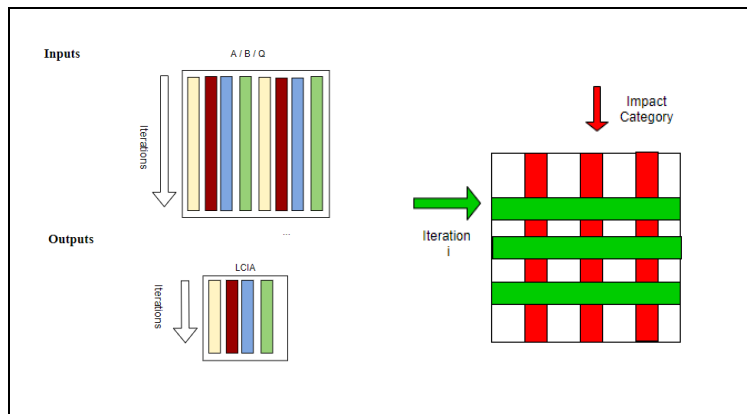
Figure 2.33 MCS input and output samples

At the end of each batch of iterations, the resulting input or output variables are exchanged with a master node (i.e., thread or process) to form a 2D vector (Figure 2.33) in which columns are the aforementioned variables, and rows are the different iterations.

In our OpenMP-based implementation, each thread enters a critical section when it finishes execution, and where it exchanges its 2D vector of input or output variables with a Master thread. Each exchanged 2D vector is stacked in a globally shared 2D vector, which will contain the samples from all the iterations and for all the variables of each variable type.

Each type of output or input variable has a global 2D vector, which makes a total of three 2D vectors for A, B, and the output LCIA. In our MPI-based implementation, the processes exchange their input or output 2D vector variables with the Master process asynchronously when they finish execution.
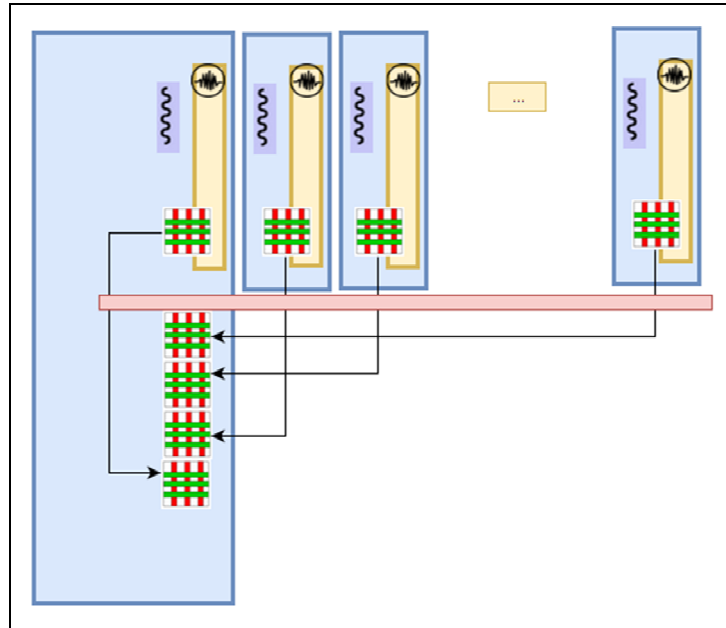
Figure 2.34 2D Vectors exchanging in OpenMP/MPI

Another form of data exchange is letting each batch of iterations save its results using the grid shown in Figure 2.33. The size of rows and columns can be configured dynamically.

B. In-sampling vs. pre-sampling

The pre-sampling of an LCA system consists of, in an initializing step, dependently sampling its exchanges in a Monte-Carlo simulation. Those samples are stored in memory, or on disk, along with their ranked vectors for later calculation requests.
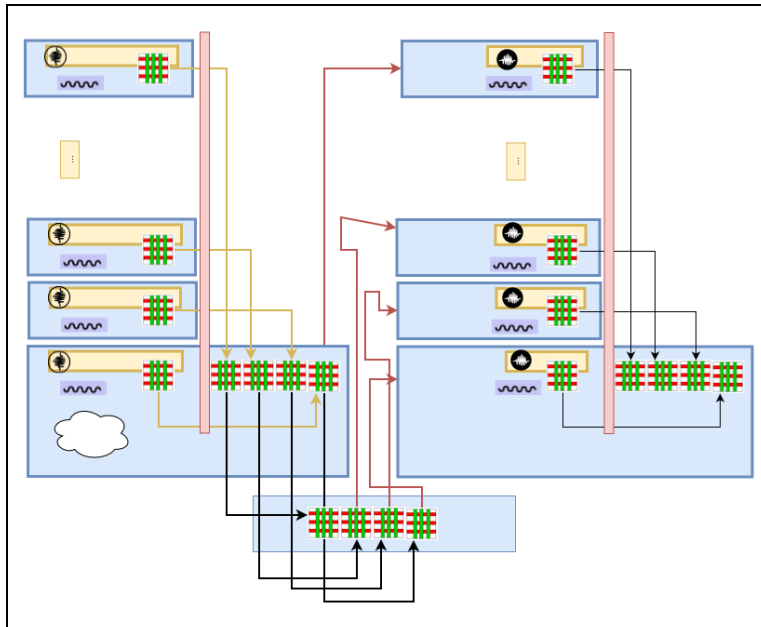
Figure 2.35 Pre-sampling parallelism

When a new calculation request for a Monte-Carlo simulation arrives, the samples are read from memory instead of sampled randomly. If sensitivity analysis is needed, the ranked vectors in memory are read instead of re-computed. The avoidance of ranking the system exchanges for each calculation request is very beneficial.

In an initialization step, the cluster on the left of Figure 2.35 performs the sampling and ranking of the system and sends the sampled 2D vectors to the "*main*" application process. When a calculation request is received, the stored 2D vectors will be sent to the individual threads or processes of the calculation cluster on the right of Figure 2.35.

C. Solving methods

Two main solving methods are present:

o The Matrix Method, where the whole system is loaded into a matrix object and equation 1.3 would be solved using Eigen++ BiCGStab;
o The *Hybrid Solver*, where only the background layer is solved using Matrix Method.

D. Background layer representations

There are two representations of the background layer:

➢ Des-aggregated background layer where the details of the background layer are taken into consideration while solving and reporting;

➢ Aggregated background layer where the background layer impact scores are aggregated and stored by *impactId* and *processId*.

The aggregated background layer feature consists of two designed modules the "*AggregateSampler* (MPI)" and the "*CalculatorsAggregated* (OpenMP)". The *AggregateSampler*, explained in section 2.12, will pre-sample and pre-rank the input variables, and aggregate the impact scores for the background layer processes. The *CalculatorsAggregated*, explained in section 2.12, will read the aggregated scores generated by the *AggregateSampler* and use those scores in the Monte-Carlo iterations to skip analyzing the background layer.

2.16.2   Prototypes for Parallel Monte-Carlo and Parallel GSA

The different prototypes are:

1. Calculators_OpenMP: an in-sampling Monte-Carlo with parallelism using OpenMP;
2. Calculaors_MPI: an in-sampling Monte-Carlo with parallelism using MPI;
3. Calculators_PreSampling: uses pre-sampling with MPI and Monte-Carlo;
4. Calculators_Foreground: allows the use of a *hybrid solver* and OpenMP features;
5. AggregateSampler and *CalculatorsAggregated* prototypes: use the pre-calculated aggregated datasets feature.

2.17 Monte-Carlo sampling (MCS) and uncertainty propagation

This section presents the in-sampling and pre-sampling MCS prototypes using the des-aggregated background layer features. Both in-sampling and pre-sampling can work with either the Matrix Method or the Hybrid Solver methods.

2.17.1 In-sampling MCS

In LCA, as per Peters (2007b), uncertainty propagation through MCS can be accomplished by sampling the input matrices A, B, and Q by using RNGs created based on the uncertainty information in the matrices, then computing the LCA output variables **g** and **h** and finally, saving the output variables for **N** iterations for later statistical analysis. After MCS is complete, the uncertainty is propagated and presented to the output variables. The uncertainty on the output variables can be assessed by computing statistical metrics such as the median, variance, and percentiles. Furthermore, the output variable samples can be drawn to visualize the shape of the PDF at the output of MCS.

Our proposed implementation of MCS is described in Algorithm 2.6. The implementation proceeds as follows: create static only matrices (i.e., A, B, Q) outside of the iterations loop; second, create, inside the loop, the sample matrices (i.e., A', B') containing uncertain cells; third, merge the static matrices and the sampled matrices using vectorized matrix addition and finally, compute equations 1.3, 1.4 and 1.5 to generate vectors g and h. The samples of output and input variables are accumulated (Figure 2.33) in 2D arrays for each of the matrices A, B, and Q.

The evaluation of *Exchanges* or *ImpactFactors* in the static phase is equivalent to directly reading static values or interpreting formulas dependent on a tree of parameters. On the other hand, the evaluation in the stochastic phase is equivalent to directly sampling a known PDF. In both situations and when formulas are present, EXPRTK is adopted as described in section 2.6.

After the simulation is finished, the different threads or processes running the simulation will exchange their 2D matrices of output variables with uncertainties into a global 2D matrix. The columns of this global matrix, which are impact categories, will be the subject of statistical analysis, as described earlier in this section.

Algorithm 2.6 Monte-Carlo proposed implementation

Algorithm: LCA Monte-Carlo proposed algorithm

Inputs : data:CalculatorData

Output: result:IterationsResults

1 $A = $ Eval (Static Exchanges of A) //Loop
2 $B = $ Eval (Static Exchanges of B) //Loop
3 $Q = $ Eval (impact factors) //Loop
4 **for** *iteration: 1 to $N_{ComputeNode}$* **do**
5      $A'= $ Sample (Exchanges of A) //Loop
6      $A'= A'+A$
7      $B'= $ Sample (Exchanges of B) //Loop
8      $B'= B'+B$
9   Calculate Section:
10     $s'= $ result of solving $A's'=f$
11     $h'= B'* s'$
12     $g'= h'* Q$
13   End - Calculate Section
14   Accumulate g', A', B' in 2D matrices
15   Reset A', B'
16 **end**
17 *Return the 2D representing g', A', B'*

This statistical analysis is developed in parallel using OpenMP. The different columns are spread across the available threads, and each thread will perform its statistical analysis and store it in ImpactStat object. At the end of each thread work, an ImpactStat object will be exchanged and joined into a final vector (i.e., vector[ImpactStat]) containing the statistical metrics of all the impact categories under study.

2.17.2   Pre-sampling MCS

When adopting the pre-sampling feature, lines five and seven now read from memory instead of entities sampling. Also, lines 14 and 17 would consist of accumulating and saving only the output variable g.

2.18   SCC based GSA

To perform GSA using SCC, an algorithm implemented in most scientific libraries may proceed as follows: first, it will iterate over all the input and output variables, and for each (*input, output*) instance, it will compute SCC using the library provided functions. Second, it will compute the contribution coefficients based on the computed correlations. Finally, it will sort the contributions and select top contributors.

Our first implementation of the SCC-based GSA relied on using statistical libraries (e.g., MLlib Spearman, gsl Spearman, etc.) to perform the Spearman correlations. These libraries are designed to make individual correlations, which requires that those libraries are fed with a cartesian product of (*input, output*) variables. This cartesian product results in a re-computation of the Spearman ranks of input variables for every output variable. In LCA, the number of output variables is in the order of magnitude of 10th of elements (e.g., 18 variables for Recipe Midpoint I 2008), and the input variables are in order of magnitude of $100,000^{th}$ of elements (e.g., 273,000 for "Production of aluminum in Quebec"). This ordinality resulted in the hundreds of thousands of input variables being re-ranked tens of times.

2.18.1   In-ranking SCC

To overcome the cartesian ranking, we designed and implemented our custom version of SCC in algorithm 2.7, that mainly separates the ranking step from the correlation step. This separation results in a ranking that happens only once for each of the input and output

variables. The algorithm uses gsl Spearman rank for the ranking, and we have implemented our version of Pearson correlation, as shown in Figure 2.36.

In the ranking section of algorithm 2.7, each input and output vector is ranked only once using *OMP For* loops. In the correlation section, the input and output variables are cross-correlated. Each of lines 3 to 5 is a call to a gsl ranking function. Lines 7 to 9 are executed serially for joining the computed ranks into 2D shared memory. In the contribution section, the contributing factors are computed, sorted, and the top 100 contributors are selected.

Algorithm 2.7 SCC proposed algorithm

Algorithm: SRCC proposed algorithm

  Inputs : Asamples:2DVector, Bsamples:2DVector,
          LCIASamples:2DVector
  Output: Correlations: vector [(inputid,outputid,correlation)]

1 RankingSection;
2 OMP **for** *Asample, Bsample, LCIASample in Asamples, Bsamples,*
    LCIASamples **do**
3   Arank = SpearmanRank(Asample)// from gsl library
4   Brank = SpearmanRank(Bsample)// from gsl library
5   LCIARank = SpearmanRank(LCIASample) //from gsl library
6   Critical Section;
7   Aranks += Arank
8   Branks += Brank
9   LCIARanks += LCIARank
10 **end**
11 CorrelationsSection;
12 Initiate correlations as vector [(inputid,outputid,correlation)]
13 OMP **for** *outputrank in LCIARanks* **do**
14   **for** *inputrank in Aranks and Branks* **do**
15   Correlations+ = pearsoncorrelate(inputrank, outputrank)
16   **end**
17 **end**
18 ContributionSection;
19 correlations'= correlations^2
20 correlationsSorted= correlations'.sort()
21 topContributions"= $\dfrac{correlationsSorted.top(100)}{\sum correlations^2}$
22 **return** topContributions

Our implementation of the Pearson correlation uses implicit vectorization to provide efficient computation, as shown in Figure 2.36.

```
1   double pearson(double *in, double inavg, double *out, double outavg, int n)
2       {
3           double sum = 0, sum_sq_in = 0, sum_sq_out = 0;
4           for (int i = 0; i < n; i++)
5           {
6               double in_diff = 0, out_diff = 0;
7               in_diff = in[i] - inavg;
8               out_diff = out[i] - outavg;
9
10              sum += (in_diff) * (out_diff);
11              sum_sq_in += pow(in_diff, 2);
12              sum_sq_out += pow(out_diff, 2);
13          }
14          return sum / sqrt(sum_sq_in * sum_sq_out);
15      }
```

Figure 2.36 Pearson implementation – implicit vectorization

2.18.2   Pre-ranking SCC

When adopting the pre-ranking feature, the ranks are pre-calculated, and therefore, the "RankingSection" of algorithm 2.7 will consist of ranking the output variables.

2.19   Aggregated datasets based MCS

The solving of an LCA system by using pre-calculated aggregated datasets is based on the model shown in Figure 2.37. Based on this model, solving an LCA system is accomplished by first solving the foreground layer using the Sequential Method; then reading the unitary pre-calculated scores of the background layer barrier processes from pre-calculated arrays and finally, scaling the background layer barrier processes unitary aggregated scores by the total demands of the scaled foreground layer barrier processes.
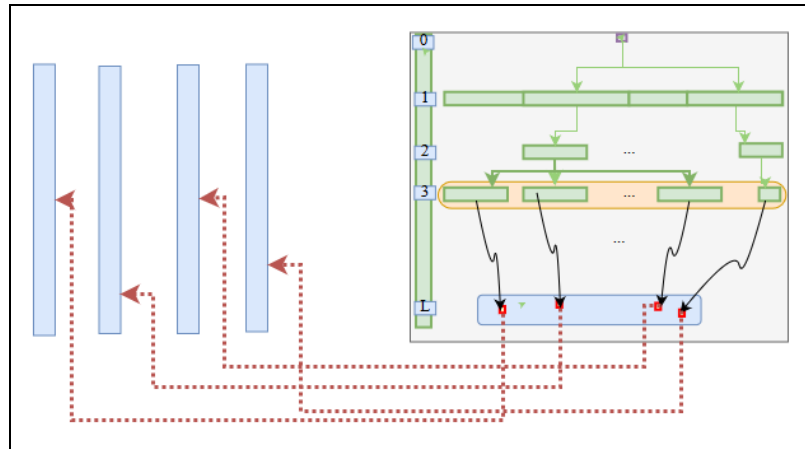
Figure 2.37 Aggregated background layer graph model

To accomplish this model, we designed algorithm 2.8 to aggregate each background layer process, which allows for replacing its LCA graph with a single node. Figure 2.38 presents the process of generating an aggregated score for each process and impact category. The algorithm consists of creating a system of one root process, demanding all the background layer processes and then iterating a Monte-Carlo simulation to generate arrays of aggregated LCI or LCIA scores for each background layer process.
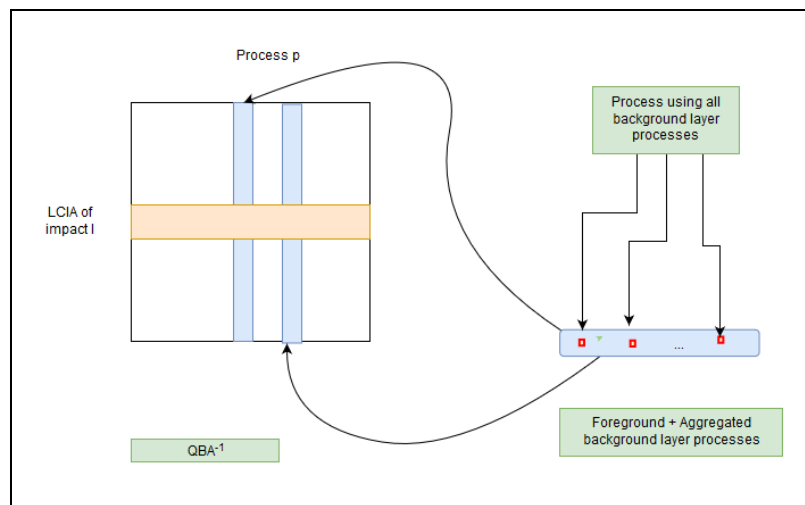
Figure 2.38 Aggregating background layer

Algorithm 2.8 Background layer scores aggregation

Algorithm: LCA Monte-Carlo Iteration - aggregating the background layer

Inputs : ImpactMethodId, parallelism, number of iterations, Apath, Bpath, CalculatorData

Output: IterationsResults arrays stored in files

1 $A = $ load samples of A
2 $B = $ load sample of B
3 Load Q;
4 LCIAScores$= QBA^{-1}$
5 **if** *lciIsNeeded* **then**
6 $\quad$ LCI$= BA^{-1}$
7 **end**
8 Store the LCIAScores by /root/impactMethodId/processId;
9 StoreIfNeeded the LCIScores by /root/elementaryflowid/processId;

Algorithm 2.9 will first load the system in Figure 2.32 into matrices A, B and Q. Second, it computes QBA^{-1} in each iteration to generate the arrays of aggregated scores for each impact category and LCA process. Finally, it saves the computed arrays to a store (i.e., in memory, on disk, or SS3).

The research adopted algorithm 2.9 to use the pre-calculated aggregated scores arrays in a Monte-Carlo simulation. This algorithm first traverses the background layer barrier to read its LCIA scores. Second, it solves the foreground layer to calculate the total demands at the foreground barrier. Finally, it aggregates the total LCIA scores. This algorithm provides considerable performance enhancement over algorithm 2.6 because it eliminates the step of sampling the background layer exchanges, and the step of solving the background layer matrix equation in each iteration. Instead, it solves the foreground layer (i.e., using the *Sequential Method*) and scales the aggregated scores of the background layer barrier processes.

Algorithm 2.9 LCIA scores for the collapsed-aggregated system

---

Algorithm: Monte-Carlo using aggregated scores

Inputs : CalculatorData, FGBarrier, FGDemandsMap

Output: lcia: Vector of Double

1 backgroundBarrierScores = Read(/pathTo/BGBarrierScores)
2 **for** *iteration: 1 to $N_{ComputeNode}$* **do**
3   **if** *fg_uncertain_exchanges_A.size()>0* **then**
4   **foreach** *layer in graph.foregroundLayers.where(layer>1)* **do**
5   **foreach** *supply in layer* **do**
6   *demands= demands[layer-1][supply]*
7   **foreach** *demand in demands* **do**
8   *Solving Kernel:*
9   *totalDemand= sample(demand) * demand.Scalar*
10   *outputSupply = sample(supplyReferenceFlow)*
11   *newScalar= (totalDemand / outputSupply)*
12   *supplyScalar = supply.Scalar + newScalar*
13   **end**
14   *scalars[supply.ProcessId] = supply.Scalar*
15   **end**
16   **end**
17 **end**
18 **foreach** *process in FGBarrier* **do**
19   **foreach** *demandSupply in FGDemandsMap[process]* **do**
20   **foreach** *impactCategory in impactMethod.impactCategories()* **do**
21   *totalDemand= backgroundBarrier-Scores[demandSupply.supplyId][impactCategory] * demandSupply.demandScalar*
22   *Lcia[impactCategory]+= totalDemand * sample(demandSupply.demand)*
23   **end**
24   **end**
25 **end**
26 *Accumulate g', A', B'and Q'in 2D matrices*
27 *Reset A', B', Q'*
28 **end**
29 *Return the 2D representing g', A', B', and Q'*

Algorithm 2.9 allows the complete removal of the matrix component by replacing the sampling and solving of the background layer by a read from an array of pre-calculated data. Figure 2.25 shows the equivalent model provided by algorithm 2.9, where the foreground

layer is solved and collapsed into a single node. The background layer barrier scores are read from the pre-calculated arrays generated in algorithm 2.7.

2.20   Conclusion

This chapter presented the algorithms that we have adopted to implement the requirements of this research project.

The program starts by loading the database, which involves loading the DBTemplate, second it applies the newly received differential files, and third, it loads and stores RNG for each of the uncertain entities in the system. After loading the database, the program proceeds to build the LCAGraph interconnecting the different LCA processes. Serial and parallel versions were implemented for this step.

After building the graph, the prototype proceeds to load the matrices and their indexes. The prototype then computes equations 1.3, 1.4, and 1.5. We experimented with several Matrix Methods, namely UMFPACK, for direct solving and Eigen++ BiCGSTAB for iterative solving. Also, a hybrid algorithm was proposed to solve the foreground layer using the Sequential Method and the background layer using the Matrix Method.

The research implemented uncertainty propagating using Monte-Carlo sampling followed by uncertainty assessment. Two main optimizations were proposed. First, algorithm 2.6 consists of separating certain and uncertain cells and accumulating output results in memory. Second, algorithm 2.9 allows performing the sampling and the aggregation of the background layer before running the simulation. This allows Monte-Carlo to avoid the sampling and the solving of the background layer in each iteration of the simulation.

The prototype, to perform GSA using SCC, proposes an algorithm which consists of separating the ranking from the correlation steps. This proposed algorithm allows for the ranking, in parallel, of a series of variables to be determined only once.

CHAPTER 3

LIFE CYCLE ASSESSMENT ON APACHE SPARK

This project started by studying the opportunities provided by using the Apache Spark data-parallel distributed computing framework for implementing LCA calculation algorithms. While experimenting with some of the Spark features, and by applying them to our project functionalities, we found that the Spark features are not aligned with what we are trying to implement. Also, we have found that the programming methodologies provided by traditional programming languages, as explained in Chapter 2, provide much better performance gains. In this chapter, we speak about our experimentation with Apache Spark and provide recommendations for its use in the context of an LCA project.

3.1 Apache Spark static LCA kernel implementation

The implementation of the calculator kernel on Apache Spark is based on using lazy transformations over CSV files loaded into Spark Dataframes/RDDs. A series of transformations are adopted to transform the CSV files into various LCA objects such as LCA graph and LCA matrices (i.e., A, B, and Q). The objects creation is then followed by an LCA calculation step. The schemas of the CSV files are provided in Annex VII.

> **Graph building**

Prior to building the graph (g), an initialization phase is necessary for loading the CSV files into Spark data frames. The data frames that we will be using are:

1. df_exchanges is a Dataframe containing the exchanges CSV file;
2. df_exchangesIn is a Dataframe containing the input exchanges CSV file;
3. df_exchangesOut is a Dataframe containing the output exchanges CSV file.

The graph building implementation in Apache Spark, as shown in Figure 3.1, consists of applying a series of transformations in each iteration of the graph building algorithm.
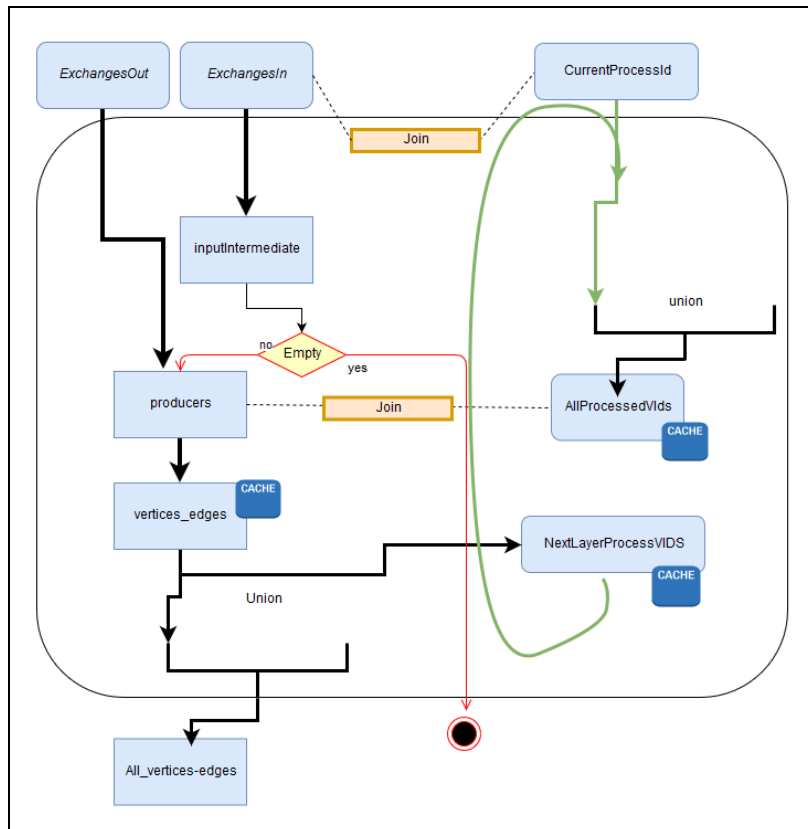
Figure 3.1 LCA Graph building on Apache Spark.
Each iteration represents a graph layer

Each iteration will consist of the following RDDs transformations:

1. For a given layer, distribute its processes in an RDD, named current*ProcessesVIDS*;

2. Append, using a union transformation, the current*ProcessesVIDS* to a global variable storing all the created processes ids, *AllProcessesVids*;

3. Compute the demands of current*ProcessesVIDS* as another RDD named *inputIntermediateRDD*. The input demand exchanges are computed by executing a *Join* on the *exchangesIn RDD* with the current*ProcessesVIDS* RDD;

4. Compute the producers of the input demands by executing a join between the *ExchangesOut RDD* and the *inputIntermediate* RDD;

5. Connect the demand and producers' layer: Join the *allProcessedVIds* RDD and the *producers* RDD followed by a MapPartitions to create an RDD Vertices and Edges named *vertices_edges*. The *vertices_edges* RDD will be appended, after the graph is built, to a global RDD *AllVerticesEdges* using a Union operation;

6. Step 5 is followed by an additional MapPartition to convert the *vertices_edges* RDD into NexProcessLayer RDD.

When joining with a small *processIds* dataframe, a *BroadcastHashJoin* is used to broadcast the relatively small Dataframes to the running executors to avoid shuffling while executing join operations. Also, when calculating producers which involve two data frames with the same key, a pre-partition by the key will re-distribute the dataframe in a way that the joining will occur inside the executor memory without the need to shuffle data with other executors spread across the cluster. Also, the algorithm caches the entities *NextLayerProcessesVIds*, *AllProcessedVIds*, and *vertices_edges* because the *NextLayerProcessesVIds* RDD represents the entity on which the next iteration will process, second, the *AllProcessedIds* is used in all subsequent iterations, and third, the *vertices_edges* RDD represents the result of processing each layer.

➢ **Matrices Loading**

Figure 3.2 shows the loading of the Technosphere matrix in Apache Spark. The algorithm uses the *All_vertices_edges* RDD, output of the graph building algorithm, to construct the raw data of matrix A. The algorithm in Figure 3.2 constructs the output and input intermediate exchanges as *tpls_out* and *tpls_in* RDDs using *MapPartitions* transformations. These two RDDs will be merged using the *union* transformation to form the *tplsA* RDD or the full exchanges of matrix *A*. The exchanges in *tplsA* may contain entries with similar keys; those will be aggregated using the reduceByKey Spark transformation to give tplsA_merged

RDD. Following this, the algorithm will group the entries in the *tplsA_merged* RDD by their rows index to create the IndexedRows RDD representing the rows of matrix A.
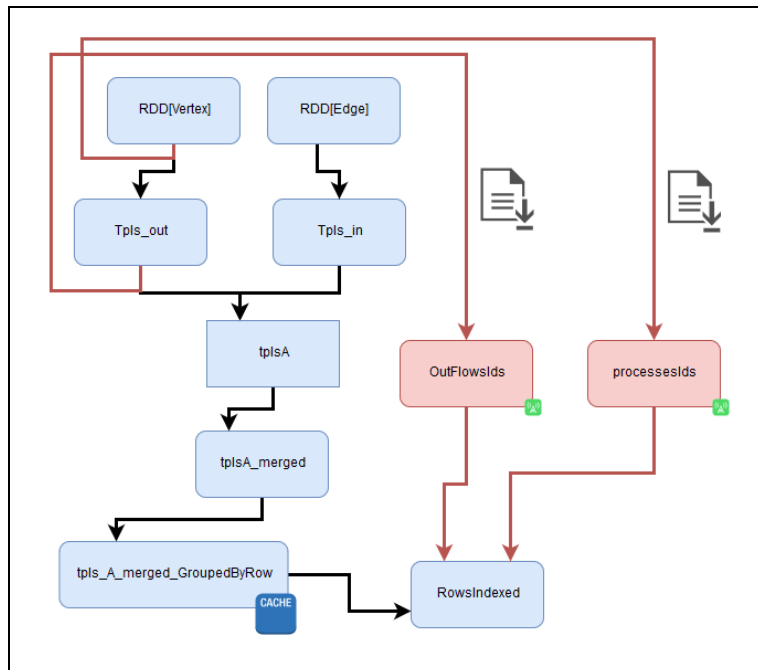
Figure 3.2 Loading A in Apache Spark

Figure 3.3 shows the loading of the biosphere matrix in Apache Spark. The algorithm uses the exchanges dataframe and the *processesIds* vector to construct the raw data of matrix B. The algorithm for building matrix B starts by creating the *elementaryFlows* RDD, which is the result of filtering the *exchanges* dataframe by the *exchangeFlowType* and *processId* fields. The *elementaryFlows* RDD is later followed by an initial MapPartition transformation to build the *elementaryFlowsIds* dataset and a *reduceByKey* to build the *elementaryFlowsMerged* RDD. The merged dataset is finally transformed using a reduceByKey transformation by its rows indices to create the *IndexedRow* dataset from which an *IndexedRowMatrix* can be formed.

The LCI and LCIA steps are computed by matrix multiplication using the Apache Spark BlockMatrix distributed matrix multiplication feature.

To solve the system of linear equations in equation 1.3, we experimented with the following two methods:

- Parallel LU factorization on Apache Spark to solve a linear system (Apache_Spark, 2017);
- SVD decomposition, followed by matrix inverse (StackOverflowCommunity, 2017).
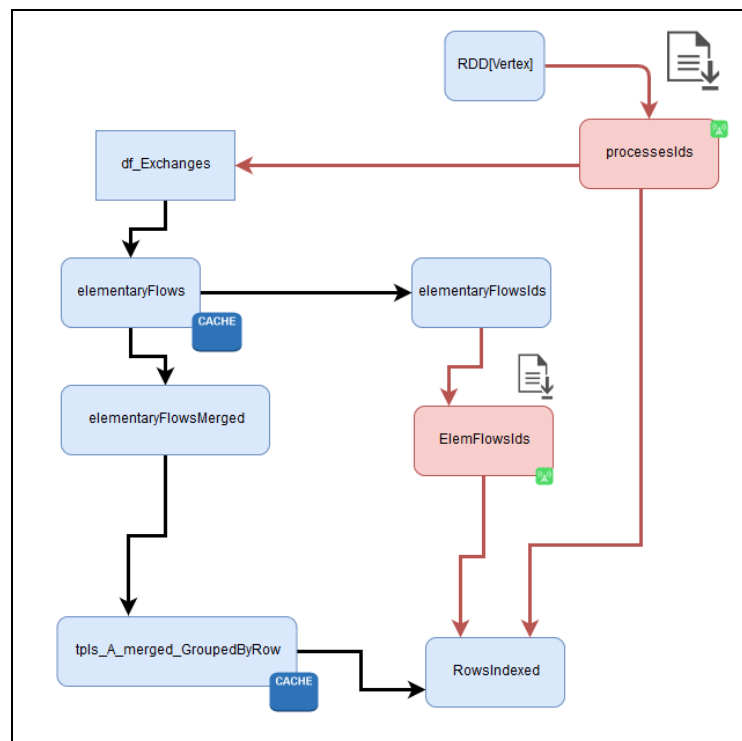
Figure 3.3 Loading B in Apache Spark

3.2   The validity of Apache Spark for LCA projects

The research analysis shows that in an LCA project, there are very few places for Apache Spark to be efficient. In this section, we discuss the opportunities where Apache Spark can be adopted for the different modules of this research project.

➢ **Graph and Matrices Building**

We found that Spark is most useful when building the graph (g) and matrices A and B based on an extensive database. Such a database would contain, for example, activities and exchanges for a large number of system products or projects from different places on the planet (i.e., *UniversalLCADB*).

To build (g), an algorithm based on treating LCADB as a set of Spark data frames needs to access the DB to build the different activities and their connections iteratively. Loading the matrices A and B may also require accessing *UniversalDB*, and therefore querying the database using Apache Spark may be useful.

When trying to select producers for a given demand, a data query may need to be executed on a large centralized dataset containing a large dataset of producers. This would be an excellent opportunity for BigData applications. Accessing a large dataset of producers would need to be done during the building of the graph and not at an initialization step, and therefore graph building would be implemented using traditional methods, but the queries that get producers would be executed on Apache Spark.

➢ **LCA for multiple systems is a task-parallel and not a data-parallel computation**

Even in the extreme case of *UniversalLCADB*, the different underlying projects or system of products are usually registered under a certain projectID or a certain systemId, and therefore, each project can be extracted using the Spark *Filter* transformation. This filtered RDD would then be collected on the Spark driver and transferred to calculators like the ParallelLCA0.1 calculator for LCA computation. LCA algorithms do not have to be designed for Apache Spark when the required data can be extracted and separated from the whole database. Apache Spark should only be used for cases where the subset of data cannot be extracted from the whole store. Also, the step of loading matrices can be avoided because once the

graph (g) is built, it can hold all the necessary information required to build matrices A and B.
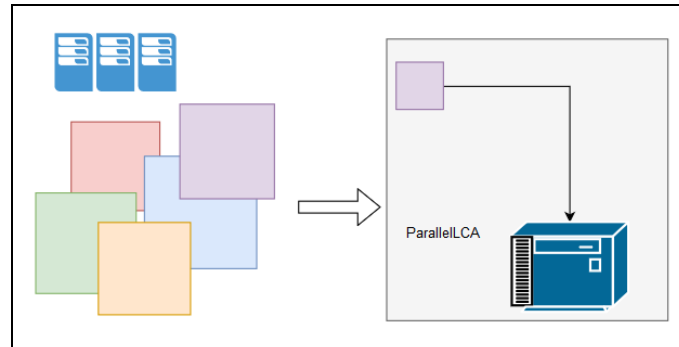
Figure 3.4 Subset extraction

The case of analyzing the buildings of a large city or the set of buildings across the globe is a task-expensive problem and not a BigData problem. The buildings can be analyzed independently, and therefore, the analysis would consist of analyzing the different buildings (i.e. which are not BigData models) in parallel, where each building analysis can be done with calculators like ParallelLCA0.1.

➢ **Updating LCADB**

Applying deltas on the main DBTemplate is a process that requires inserting new records as well as also updating and deleting records in-place. Apache spark does not provide an in-place operations for the update/delete operations. The way Apache Spark introduces changes on datasets (i.e., RDDs, Datasets, Dataframes) is by applying transformations on the initial datasets. For the delete scenario, cells in the RDD need to be excluded using the *Filter* transformation. As for the update operation, a Map transformation is commonly used.

As we saw in Chapter 2, using traditional programming methods like C++ HashMap, the cells can be simply deleted from memory without transforming the whole HashMap. Also, an update operation could consist of simple memory access and value alteration.

➢ **Calculation phase**

Given the fact that a given project can be extracted from the BigData store and sent to ParallelLCA0.1 for processing, the following modules are better developed in specialized scientific computing libraries rather than with Apache Spark:

1. Parsing the graph (g) to solve the foreground layer, solving the background layer, and computing LCI, LCIA;
2. Computing the LCI and LCIA contribution reports;
3. Computing the upstream contribution report;
4. As a result of points one and two, the calculation kernel in Monte-Carlo.

➢ **Sensitivity analysis**

The dimensions of variables at the output of Monte-Carlo simulation, which are the input to the sensitivity analysis, are not BigData. Based on Jolliet et al. (2010), in the context of LCA Monte-Carlo simulations, LCA systems need between 1,000 and 300,000 iterations to converge. Also, by consulting the published research papers with respect to Monte-Carlo simulation, we found that the maximum number of iterations that are being used do not exceed tens of thousands of iterations. The upper limit of 300,000 iterations is equivalent to a vector of numbers of size 2.4 megabytes, which is a fraction of the smallest partition size in the Hadoop/Spark default configurations (i.e., 128 MB is the default size of the atomic data partition). If we extend the number of maximum iterations to one million, this will result in a vector of eight megabytes, which is still far from being considered an application for Apache Spark.

The following are the operations applied to the aforementioned variables:

1. Uncertainty propagation: median, variance, and percentile are computed independently on each of the output variables. This can be computed using multithreading, or parallel hardware;

2. Global sensitivity analysis using Spearman ranking: this operation consists of the ranking of input and output variables independently and then correlating output to input variables. It is operating independently on vectors of maximum sizes of a few megabytes, which is better accomplished using sequential or multithreaded programs on single machines.

3.3 Conclusion

This chapter discusses our experimentation with Apache Spark in implementing traditional LCA algorithms. The research found that, in most of the cases, LCA problems are not Big Data issues even when analyzing a large number of systems. The opportunities for Big Data arise when access to large environmental databases is needed, and Apache Spark and the Hadoop ecosystem can be of better use.

When exploring the capabilities of Spark, we found that it lacks the support for specialized matrix computing. Spark provides matrix factorization and solving algorithms for mainly dense matrices.

CHAPTER 4

RESULTS AND DISCUSSION

As indicated in the introduction chapter, the goals of this research are the scientific validity of results and the implementation of scalable algorithms that can benefit from the availability of a pool of resources.

This chapter will present in section 4.1.1 the validation of the calculation on processes from Ecoinvent 3.3 and in section 4.1.2 the validation of calculation processes representing examples of custom LCA examples (e.g. building models) on the foreground layer processes chain. The performance and the scalability of each phase of the calculation will be presented in section 4.2.

4.1 Results Validation - Comparison with OpenLCA and Brightway

A significant accomplishment of this research is to be able to provide results similar to other LCA software, namely OpenLCA and Brightway.

4.1.1 Use Case of "Aluminium Production in Quebec" from Ecoinvent 3.3 Database

In this section, we compare the thesis calculator (i.e., ParallelLCA) results with OpenLCA7 and Brightway2 results for the static and stochastic phases of an LCA calculation.

4.1.1.1 Static phase validation with OpenLCA7

Figure 4.1 presents the ratio of the LCIA results obtained when using ParallelLCA vs. OpenLCA to analyze the process "aluminum production, primary, ingot | aluminum, primary, ingot | cut-off, U–CA-QC" from Ecoinvent 3.3. This experiment shows a 100 % accordance

when comparing LCIA scores to OpenLCA7. The raw data for this report is provided in Annex III.
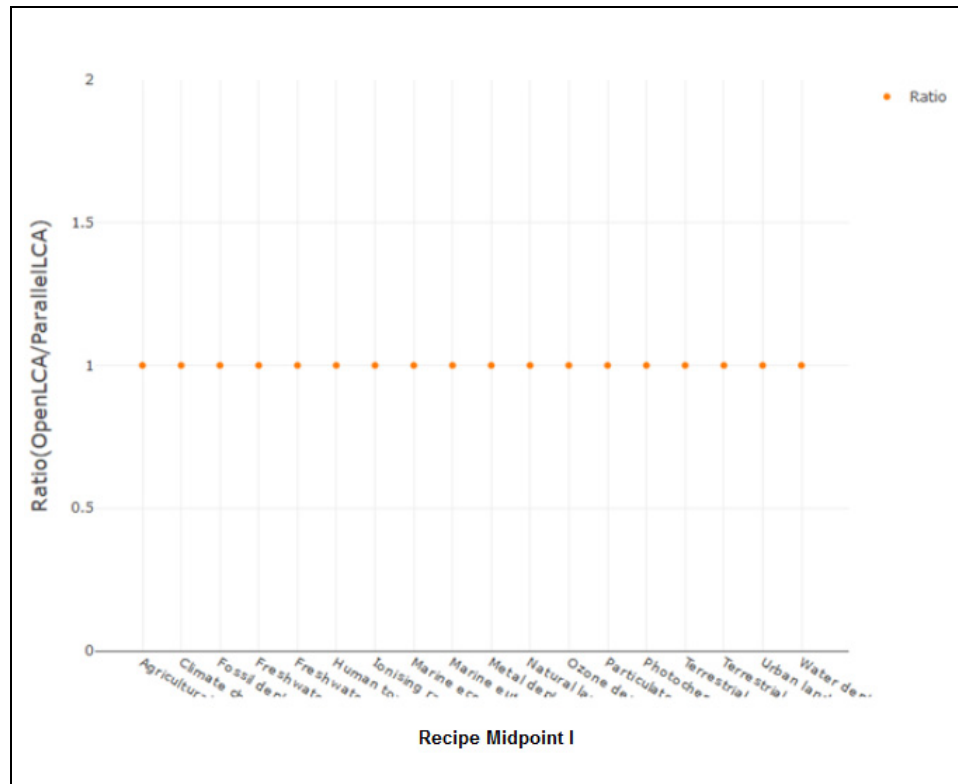
Figure 4.1 Ratio of LCIA scores for
OpenLCA7 vs. ParallelLCA

4.1.1.2 Stochastic phase validation with Brightway2

The validation of the uncertainty propagation report uses Brightway2 due to an incorrect implementation of the Pedigree approach in OpenLCA7.0. However, we are still using the data generated from an OpenLCA7.0 database. An OpenLCA database is created by importing an EcoInvent3.3 into OpenLCA software.

Figure 4.2 shows the error (i.e., Difference) in the PDF statistical metrics when comparing BrightWay2 to the ParallelLCA. Annex IV provides the raw experimental data for both

calculators. In this experiment, we highlight a majority of impact categories that are in accordance.
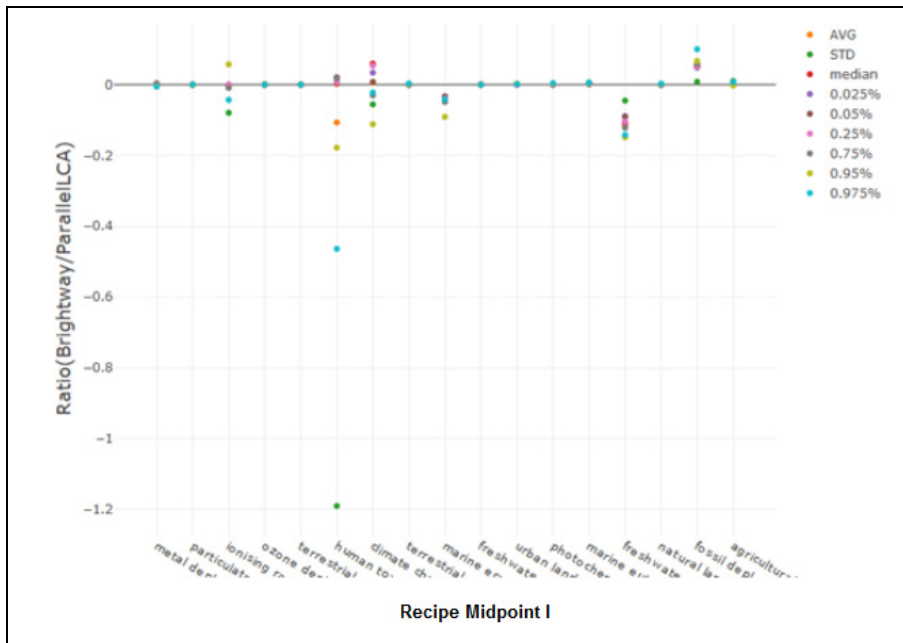
Figure 4.2 Error in the uncertainty of Thesis calculator.

Brightway2 vs. Thesis calculator for RECIPE Midpoint I

We validated the Spearman rank results by comparing and making sure of the similarity of the correlations obtained using our implementation in algorithm 2.7 with those obtained when using an equivalent Spark.MLlib implementation.

4.1.2   Use Case of Front layer Added to the Background Layer

In this section, we test the calculator on examples having a foreground layer that is using processes from EcoInvent3.3. Table 4.1 shows two models that the calculator received as test models.

We conducted the validation, in section 4.1.2.1, for only the static phase given that the foreground layers in the test examples are all static. Also, the models in Table 4.1 are only

available in the OpenLCA7 database, and therefore, the validation will be restricted to OpenLCA7.

Table 4.1 Foreground examples

Descriptor	Description
Poly	Foreground layer, connected to one background process
Habitations	Foreground layer, connected to many background processes

In addition to the validation in the static phase, we will show in section 4.1.2.2 a study comparing algorithm 2.6 (i.e., Full Monte-Carlo sampling) and algorithm 2.9 (i.e., pre-calculated aggregated datasets).

4.1.2.1   Static phase validation for systems with foreground layer with OpenLCA7

Figure 4.3 below shows a comparison of the obtained LCIA results when using ParallelLCA *Hybrid Solver* from algorithm 2.2 against OpenLCA7 implementation. As shown in Figure 4.3, the Hybrid Solver provides the same results as those provided when using the Matrix Method of OpenLCA.
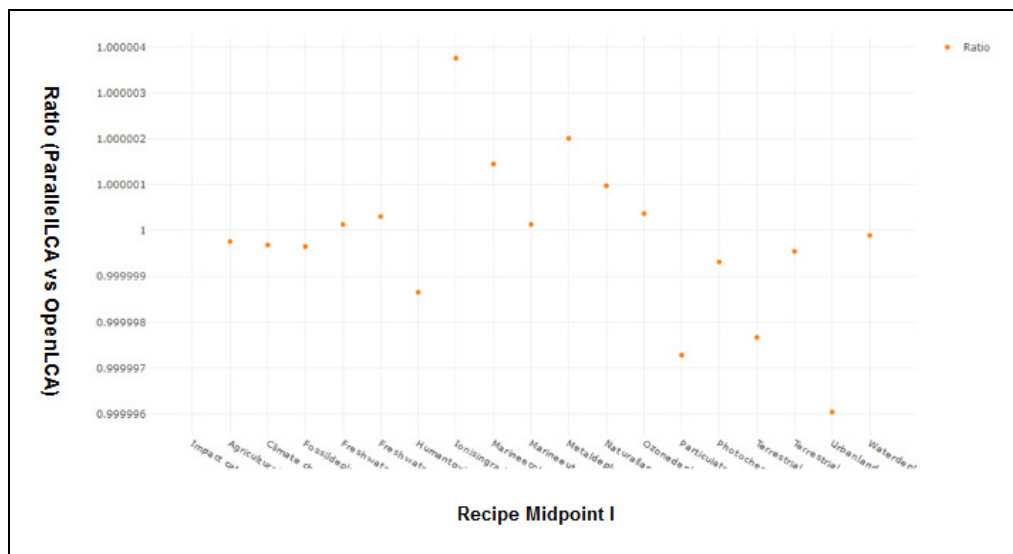
Figure 4.3 Ratio of LCIA scores: OpenLCA7 vs. ParallelLCA Hybrid Solver

for the "poly" example

4.1.2.2 Stochastic LCA comparison of algorithm 2.8 and algorithm 2.5

In Figure 4.4, we compare the results obtained when using algorithm 2.9 (pre-sampled aggregated datasets), instead of algorithm 2.6 (Full Monte-Carlo simulation). We ran Monte-Carlo for 6,000 iterations.
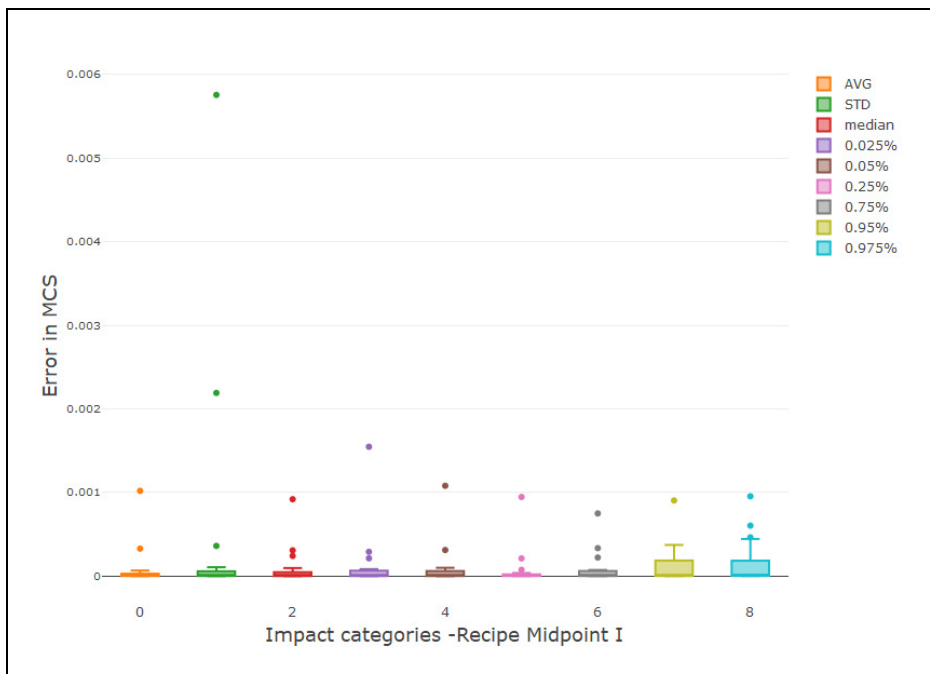
Figure 4.4 Error in the LCIA scores for 6,000 iterations for
Aggregated vs. Full Monte-Carlo of the "Habitations" example.

The experiment shows that by using the Aggregated Scores for the background layer, the median error in all the metrics is in the order of 10^{-6}. Annex V provides the raw data for this experiment. Figure 4.4 shows that our implementation of algorithm 2.9 can provide an uncertainty propagation with a small error when compared to Full Monte-Carlo sampling.

4.2   Calculator Performance Characteristics

In the following sections, the performance profile of the different modules in ParallelLCA will be presented. In addition, we will present the scalability of the calculator with a variable number of Monte-Carlo iterations, a system with increasing size, and a variable number of cores to run the computation.

4.2.1   Simulation Setup, Benchmarking tools, and Graphics generation

The research has developed benchmark scripts that can be re-run to reproduce the results presented here. The scripts, written in Scala, make calls through a JNI interface to the calculator API functions written in C++. To reproduce the research, the user will need to follow the instructions in the GIT links provided in Annex II to install the calculator environment before running the Scala benchmark scripts.

The tests were conducted on two test servers whose characteristics are shown in Table 4.2.

Table 4.2 Test Servers Characteristics

	Server characteristic label	**Server characteristic value**
Leda	Host OS - Docker OS	Linux Centos 7 - Linux Ubuntu
	CPU - RAM	64 GB, 16 cores @ 2.4 GHZ
AWSC-4.9-XLARGE	Host OS - Docker OS	Linux Ubuntu - Linux Ubuntu
	CPU - RAM	64 GB, 36 cores @ 2.9 GHZ

The raw execution time data can be obtained by running the benchmark scripts provided in Annex II.

4.2.2   Performance of the Calculator for Ecoinvent 3.3 Database

In this section, the performance of the calculator in the main phases of static and stochastic LCA for randomly chosen processes from the Ecoinvent 3.3 database will be presented.

4.2.2.1   Parallel Graph Building

The thesis started by developing an algorithm for graph building on Apache Spark, as described in Chapter 3. An average execution time of 500 milliseconds is needed to build a single layer. A graph in Ecoinvent 3.3 takes a total build execution time of 20 seconds using our implementation with Apache Spark. Based on the Spark performance, a decision was made to develop the graph building algorithm using traditional parallel frameworks that are used in HPC.
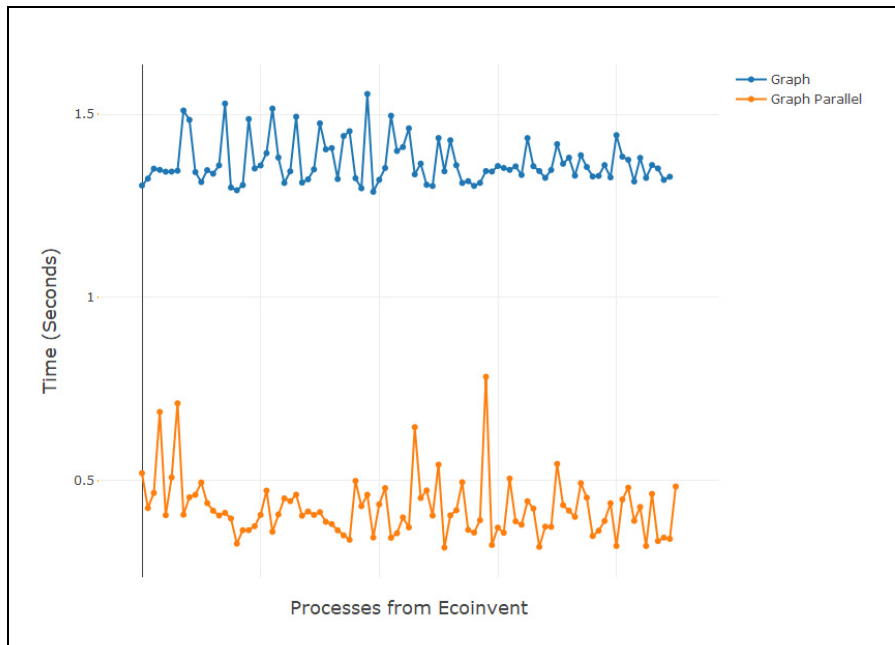
Figure 4.5 Building (g) in Memory, serial vs. parallel versions

Figure 4.5 shows the performance of our implementation, using OpenMP to parallelize the graph building. For 100 randomly chosen processes from Ecoinvent 3.3, we show the

execution time for the parallel and sequential versions of the graph building algorithm. The parallel version, presented in section 2.7, provides a gain of almost three times that of the serial version.

4.2.2.2   Static Phase

The next experiment consists of running foundational static LCA over 100 randomly chosen processes from *EcoInvent3.3*. Figure 4.6 presents a breakdown of the execution time, in milliseconds, for the foundational static phase. The most expensive part is the system solving amounting for a median of 14 milliseconds using the *BiSGStab* solver. The total execution time accounts for a median of 28 milliseconds.
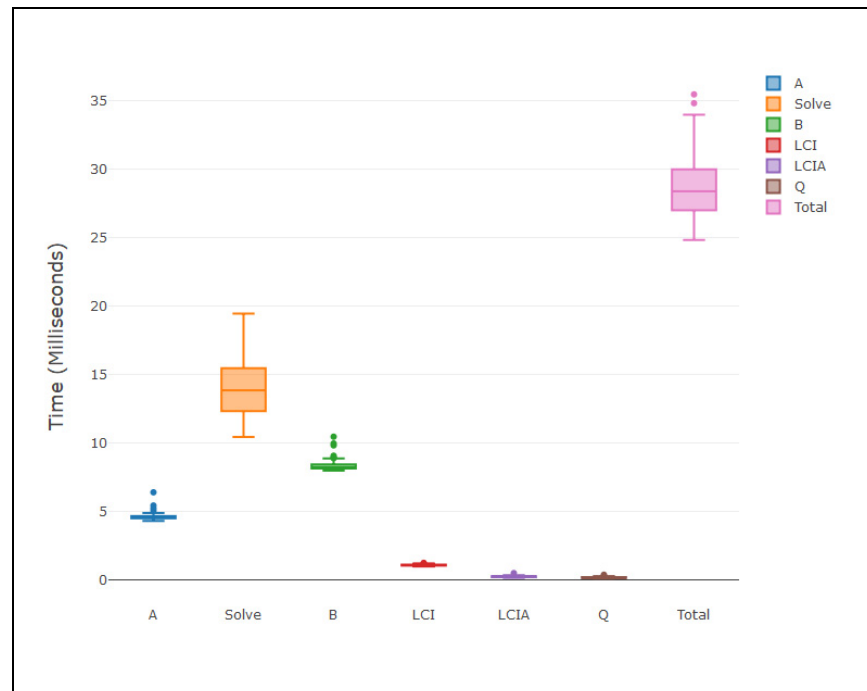
Figure 4.6 Foundational LCA performance profile

Figure 4.7 shows the performance of the static LCA contribution reports for "*Recipe Midpoint (I) 2008*" and 100 selected processes from *Ecoinvent* 3.3. The computation takes

500 milliseconds, 670 milliseconds, and 715 milliseconds for the computation of LCI, LCIA, and upstream LCIA contributions report respectively.
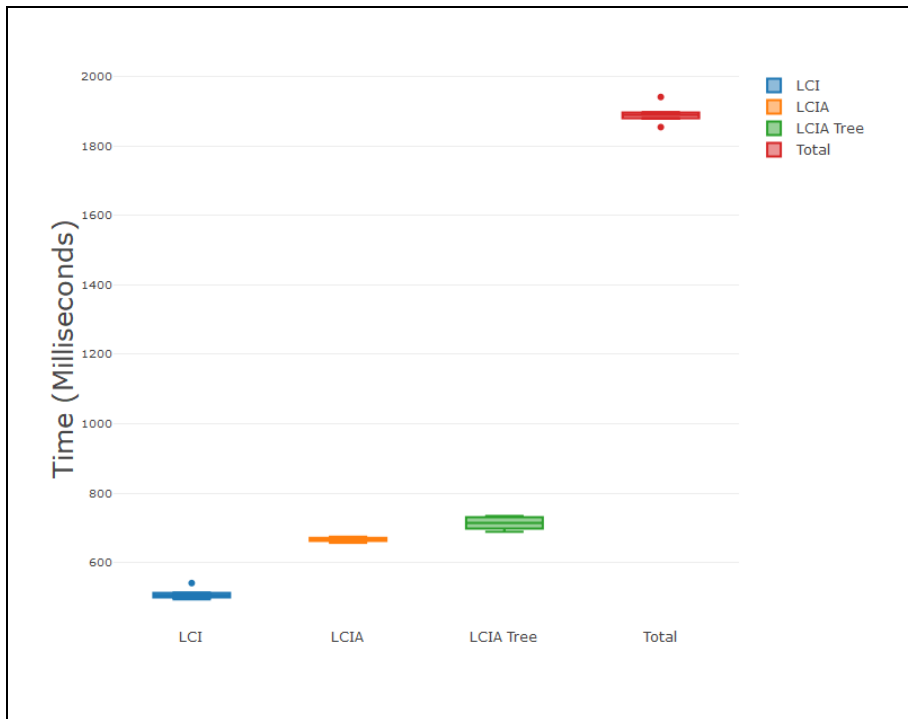
Figure 4.7 Contribution reports performance

4.2.2.3   Stochastic Phase

➢ **Monte-Carlo simulation**

Figure 4.8 shows the execution time for running sequential Monte-Carlo; the number of iterations, on the x-axis, is variable between 1,000 and 100,000 iterations. For 1,000 iterations the calculator needs up to 46 seconds to finish executing at a rate of 21 iterations per second. For 100,000 iterations the calculator finishes the computation in 82 minutes.

To better understand the numbers provided by Figure 4.8, we provide a detailed stochastic performance profile in Figure 4.9. Figure 4.9 presents the execution time in milliseconds for the stochastic kernel running inside each Monte-Carlo (i.e., Algorithm 2.6) iteration. The

experiment consists of running the stochastic kernel serially over 1,000 iterations for the same example of the production of aluminum in Quebec.
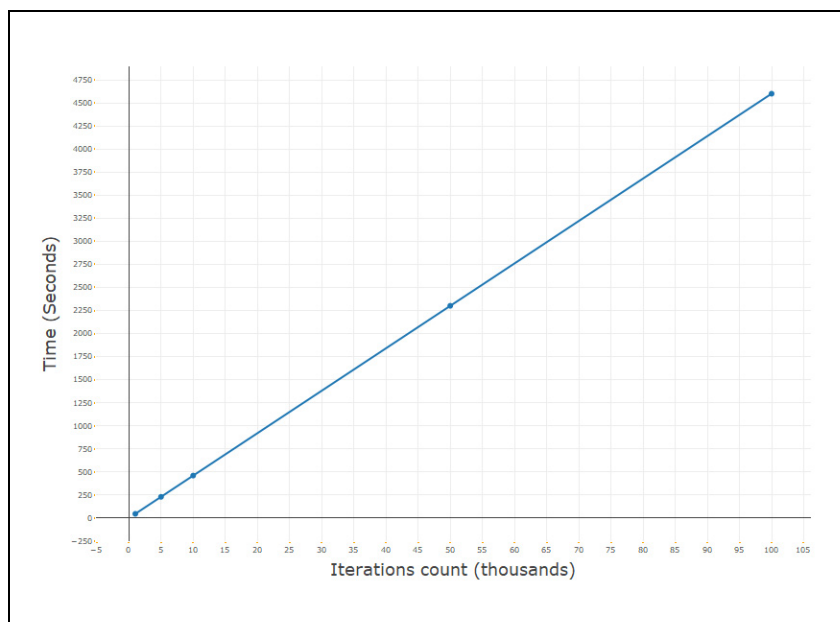
Figure 4.8 Execution time for serial Monte-Carlo of the activity production of Aluminum in Quebec
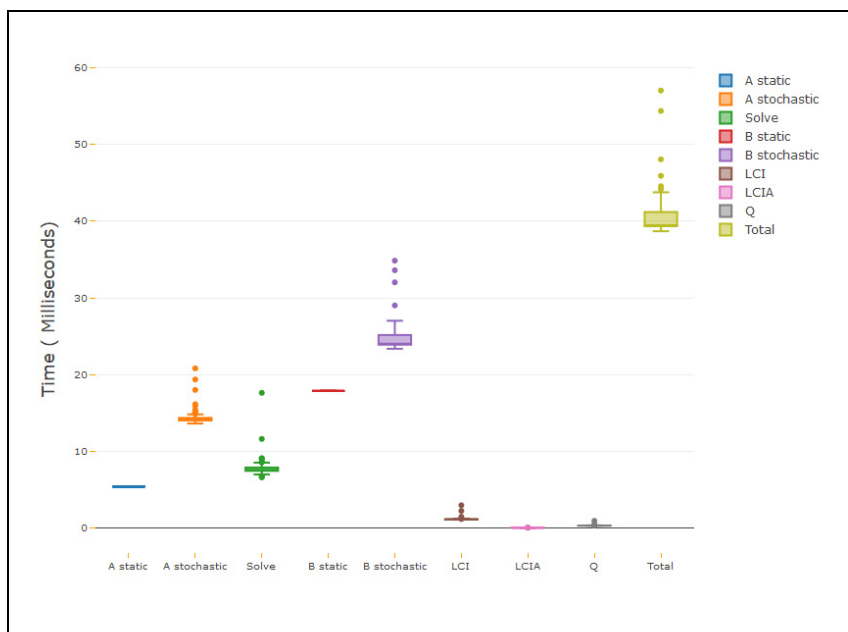
Figure 4.9 Stochastic LCA kernel performance profile

As shown in Figure 4.10, the solving amounting for a median of 8 milliseconds for Eigen++ *BiCGSTAB* (i.e., with initial guess). The total execution time is up to 40-45 milliseconds.
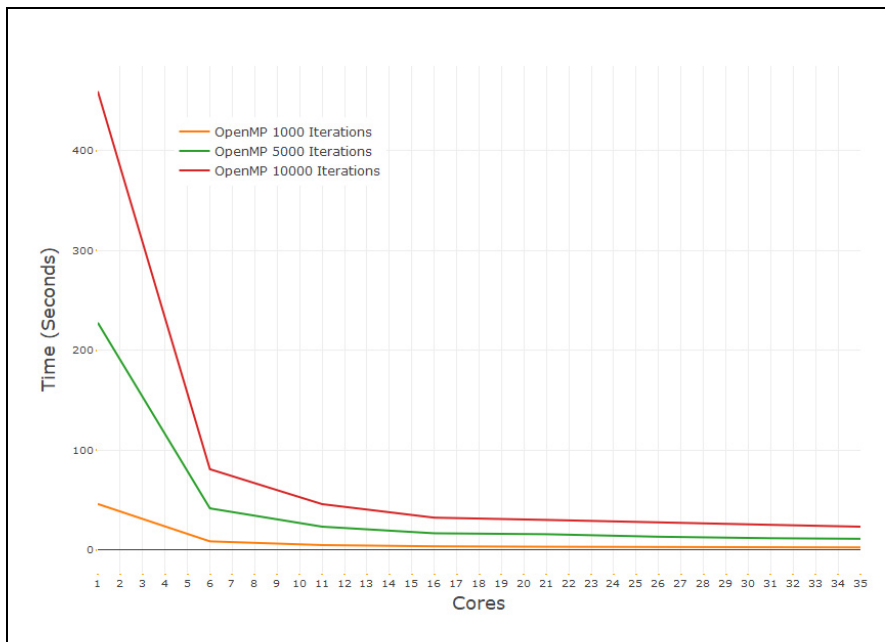
Figure 4.10 Execution time for parallel Monte-Carlo using OpenMP.

Table 4.3 Execution time for Monte-Carlo OpenMP of Figure 4.10

Number of iterations	Time (Seconds)
1,000	2.8
5,000	11.4
10,000	23.5

Figure 4.10 shows the trend of the execution time of parallel Monte-Carlo using OpenMP when varying both the number of iterations and the number of cores. We distinguish in Figure 4.10 between three main phases concerning the slope of the trend. In the first phase, the gain is at its peak with the steepest slope of the curve. In the second phase, the performance keeps on enhancing but loses its initial slope with the addition of more cores. In the third phase, the gain is at its minimum.

A second observation that can be made is concerning the time delay for the simulations with a varying number of iterations. For 1,000 iterations, the calculator needs up to 2.8 seconds to finish executing at a rate of 357 iterations per second. For 5,000 and 10,000 iterations, the calculator finishes execution in 11.4 seconds, and 23.5 seconds respectively.

Figure 4.11 shows the trend of the execution time of our implementation of full MCS when using MPI compared to OpenMP. Both implementations follow a similar trend for a small number of cores. However, when the degree of parallelism increases, MPI takes more time to finish executing the same number of iterations.
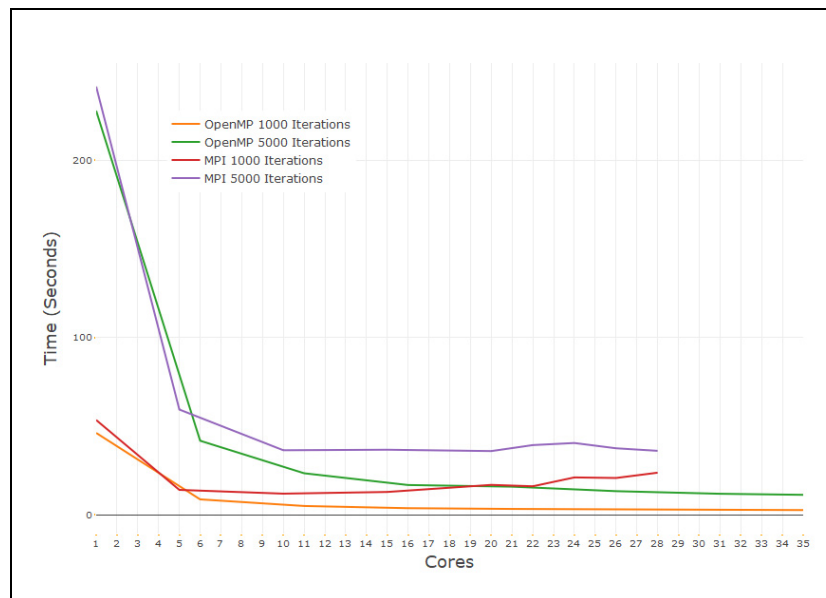
Figure 4.11 MPI vs. OpenMP for full MCS On the Leda server

Next, the thesis will study the system when using OpenMP for a higher number of iterations, 50,000 and 100,000 iterations, and for a varying number of executing cores. Figure 4.12 shows that for 50,000 iterations, and by adding the first 6 cores to the simulation; 82.5 % of the initial serial execution time is being reduced. Furthermore, by adding 16 cores, 92.5 % of the initial time is being reduced. Finally, when adding the full 35 cores, only 4.8% of the initial serial time remains, amounting for 108.6 seconds.

Also, Figure 4.12 shows that for 100,000 iterations, there is a similar reduction from the serial execution time. By adding the first 6 cores to the simulation, 82.5 % of the initial serial execution time is being reduced. Furthermore, after adding 16 cores, 93.11 % of the initial time is being reduced. Finally, when adding the full 35 cores, only 4.67% of the initial serial time remains amounting to 115 seconds.

Table 4.4 Execution time for Monte-Carlo on OpenMP of Figure 4.12

Number of iterations	Time (Seconds)
50,000	108.06
100,000	215.127
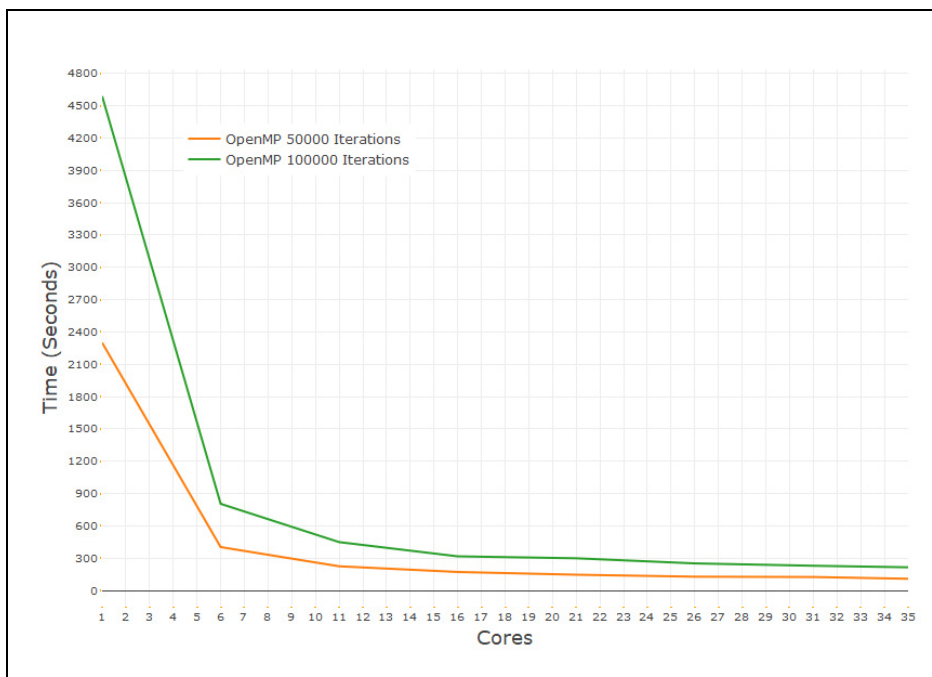
Figure 4.12 Monte-Carlo using OpenMP for a large number of iterations.

In the next study, we consider the performance obtained when using algorithm 2.9. The corresponding time delays, shown in Table 4.5, are in order of a fraction of a second or real-time.

Table 4.5 Execution time for pre-aggregated datasets Monte-Carlo

Number of iterations	Time (Seconds)
1,000	0.088
3,000	0.24
6,000	0.36
10,000	0.87

➢ **Spearman rank correlation GSA**

In Figure 4.13, the thesis presents the trend of the GSA execution time when the number of MCS iterations increases. In this experiment, we measure the time taken by ranking 217,800 uncertain input vectors and 18 uncertain output vectors, in addition to performing 3,920,400 Pearson correlations. The experiment is conducted on the full server capacity (i.e. 35 cores) The number of iterations on the x-axis of Figure 4.14 increases for up to 5,000 iterations. The ranking executing time appears to increase linearly with the number of iterations.

Figure 4.14 shows the evolution of the execution time of the ranking phase when increasing the number of cores. The ranking phase is showing a similar trend to the Monte-Carlo sampling phase of the calculation.

The ranking is further optimized by using pre-sampling and pre-ranking features of the unchanged uncertain cells. In this case, the output variables are ranked, but the un-changed input exchanges ranks are instead read from memory. This reduces the ranking time of Figure 4.13 to less than 200 milliseconds.
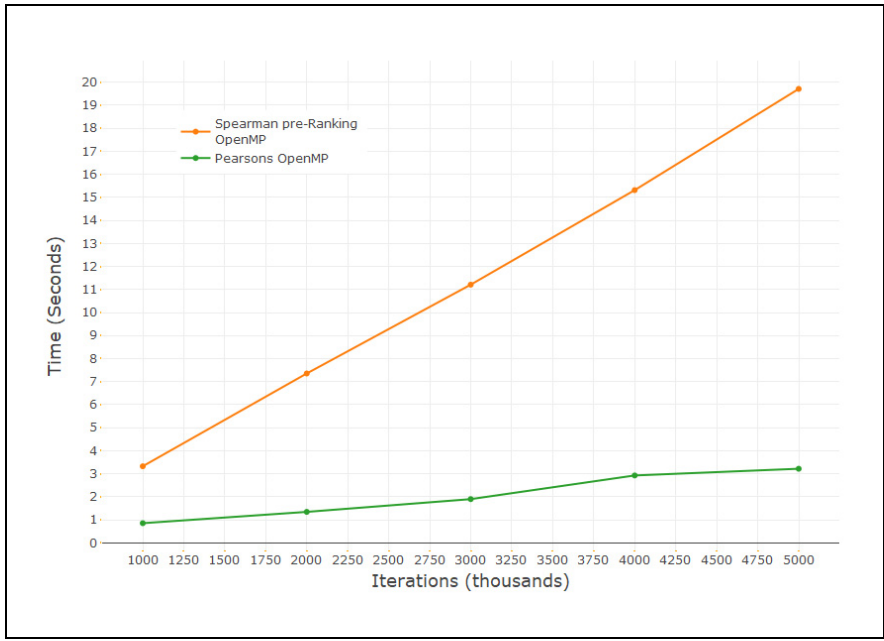
Figure 4.13 Spearman ROCC vs iteration size
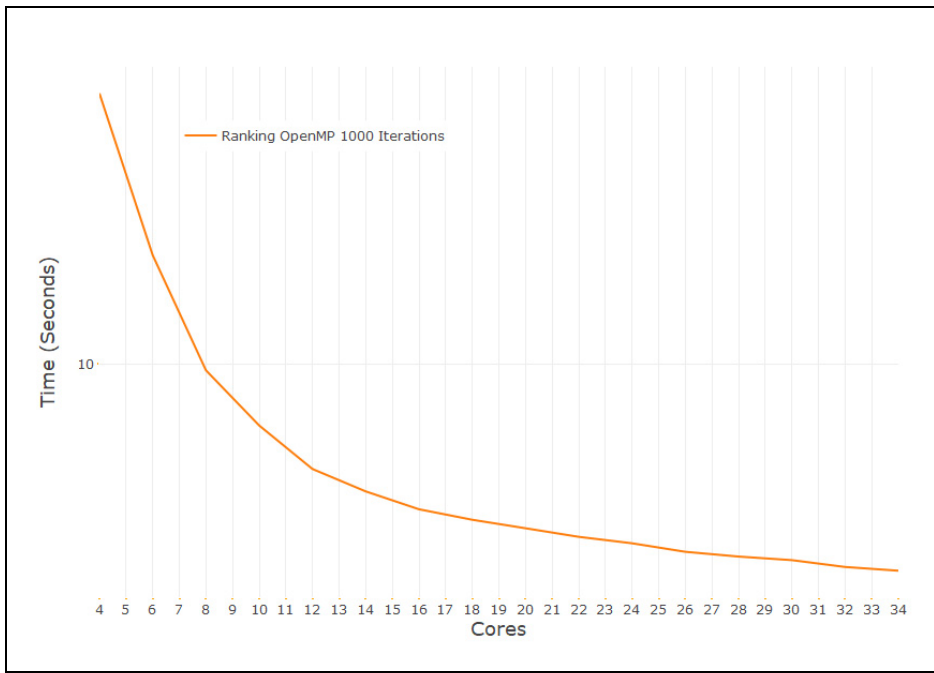
Figure 4.14 Spearman ranking OpenMP on multicore

4.2.3   Variant Size Performance Benchmark

In this section, the procedure for generating LCA systems with variant size foreground layer is discussed. Also, the benchmarks of our different algorithms for these systems is presented. We conducted all the tests in this section on the LEDA server.

4.2.3.1   Foreground generation Procedure

The variant size benchmark consists of testing the different components of the system over a range of foreground layer sizes. This experimentation consists of:

1. Generating different process delta files where each corresponds to a certain number of processes in the foreground layer as well as generating corresponding exchanges and flows delta files;

2. From these delta files, a graph is built with a foreground layer on top of a background layer. The size of the background layer is invariant;

3. Computing all LCA phases based on the graph built-in step 2.

The generation of delta files, as presented in step 1, can be described as follows:

1. The algorithm starts at layer two as current layer **l**. Layer **1** is the root node layer;

2. For each previous layer **l-1**, generate **n** processes for layer **l-1** and **k** input intermediate flows for each of the processes in layer **l-1**;

3. For each of the **k** input intermediate flows in layer **l-1,** generate **k** output intermediate flows where each has a flow-id that matches one of the input intermediate flows in layer **l-1**;

4. Increase layer count and go back to step 2.

The last layer connecting to the background layer has its connections with uncertainties. The rest of the exchanges in the foreground layer are static.

4.2.3.2   Pre-calculation phases

Before the calculation takes place, an initialization phase is computed, which consists of three steps: 1) the LCADB loading, 2) the CalculatorData loading, and 3) the graph building. Figure 4.15 shows the trend of execution time in milliseconds for these phases. The LCADB scales very well with the increase of the size of the foreground layer. However, the graph building and the CalculatorData loading execution time increases with the increase in the size of the foreground layer. The total computation time for one million processes in the foreground layer is up to nine seconds.

The total time is spread into 5.8 seconds for loading the CalculatorData object, one second for loading LCADB and up to 2.8 seconds for the graph building. Each of the graph building and the CalculatorData loading tasks was under one second for up to 162 layers in the foreground layer (i.e., 162,000 LCA processes).
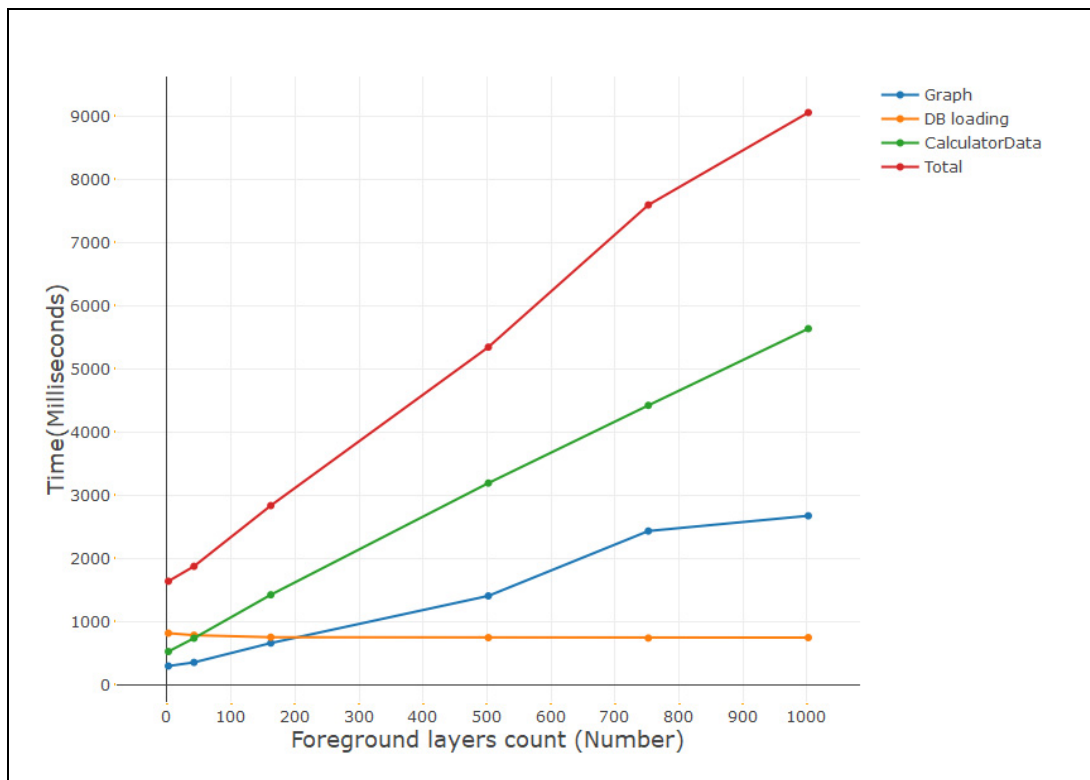
Figure 4.15 Pre-calculation phases Time (milliseconds) for variant size foreground

4.2.3.3 Static Phases

In this section, we are studying the influence on the *Calculation Kernel* performance profile when using large foreground layers. Figure 4.16 shows the solving step, and Figure 4.17 shows the remaining phases. The solving step is not scaling with the addition of layers to the foreground layer. The execution time is up to 32 milliseconds for two layers (i.e. 2000 processes in the foreground layer), 2.3 seconds for 162 layers. Also, Figure 4.14 shows the total execution time, which is dominated by the solver execution time.
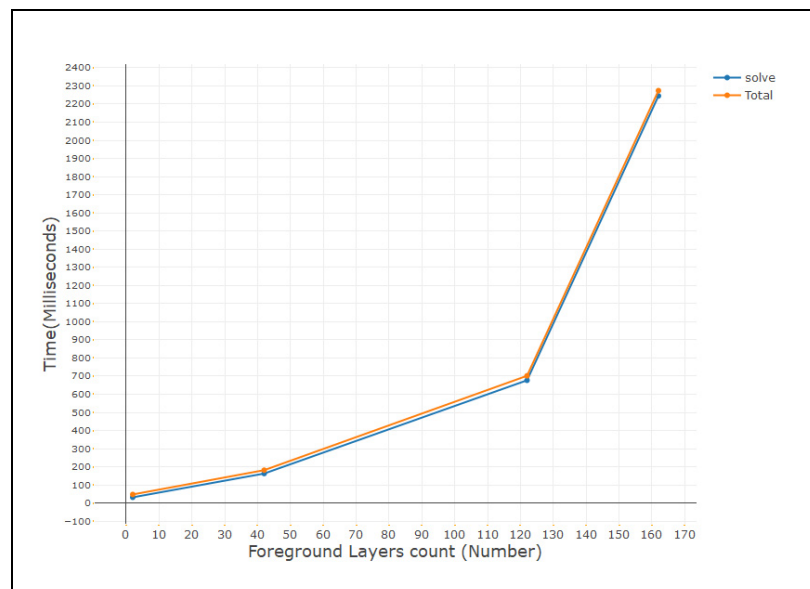
Figure 4.16 Performance for variant size foreground layer
for the Calculation Kernel using the Matrix Method

Figure 4.17 shows the other phases of the calculation, in which it appears that loading A is the most expensive part, and the only affected part by the increase of foreground layer sizes. All other phases scale well or are not influenced by the addition of large foreground layers. Matrix A loading starts with five milliseconds for two layers of foreground processes and finishes with up to 18 milliseconds for 162,000 processes in the foreground layer.
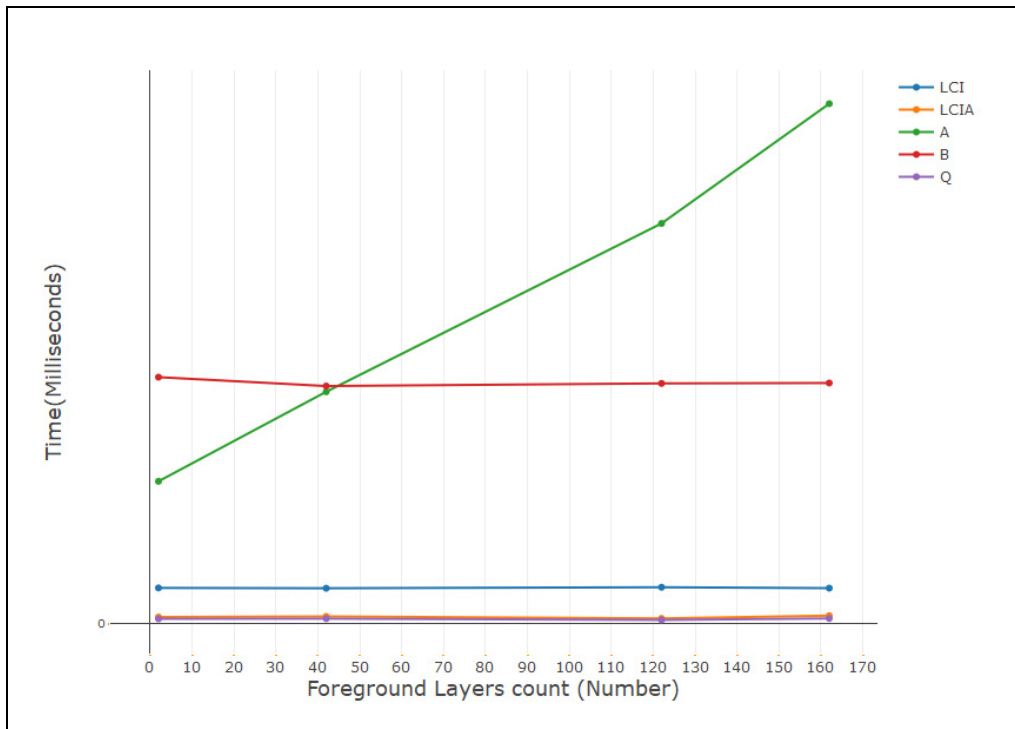
Figure 4.17 Other phases of the kernel solving using the Matrix Method

The trend caused by the solver phase renders the matrix-based kernel useless for Monte-Carlo, where the solving phase needs to be repeated thousands of times. To solve the delay of the solving phase, algorithm 2.2 proposes a Hybrid Solver that solves the foreground layer, which is the large part of the graph, using a graph traversal. Figure 4.18 shows the performance of the Calculation Kernel when using this new solving algorithm. The trend of execution time spans the interval between 38 milliseconds for two foreground layers and 128 milliseconds for 1,000 layers (i.e. one million processes in the foreground layer). The total execution for one million process accounts for 150 milliseconds. Also, as it is shown in Figure 4.18, the remaining phases are invariant with the size of the foreground layer. A difference from Figure 4.15 is that for the *Hybrid Solver* algorithm, the increase of the foreground layer size does not influence the phase of loading the matrix A.
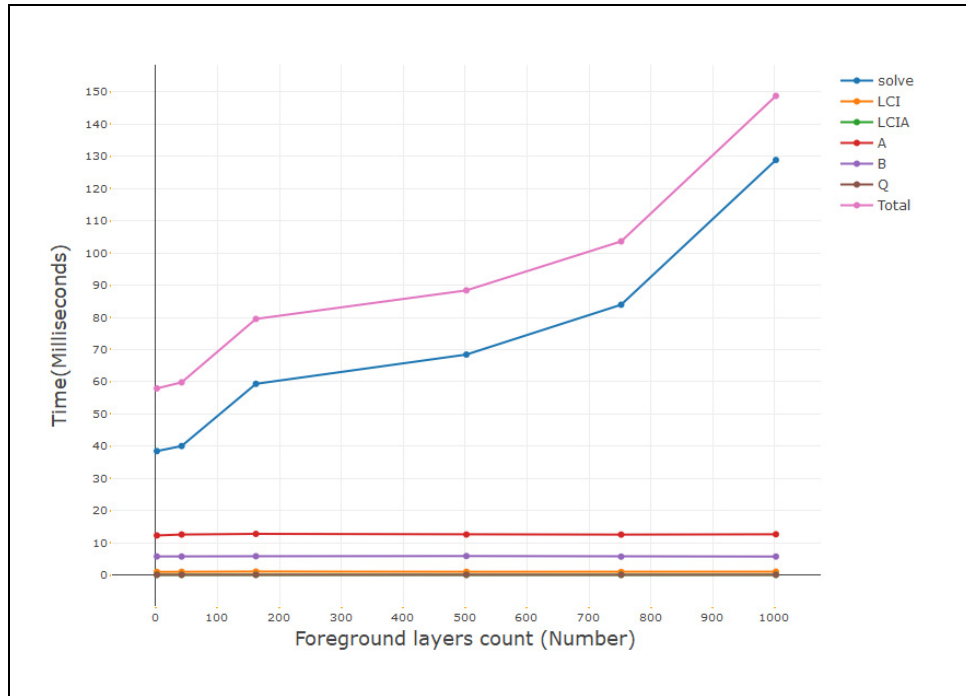
Figure 4.18 Variant size foreground layer performance - the Hybrid Solver

Another significant module which is influenced by the foreground layer size is the aggregated upstream presented in section 2.15. Figure 4.19 shows the performance of that module for a variant size foreground layer.
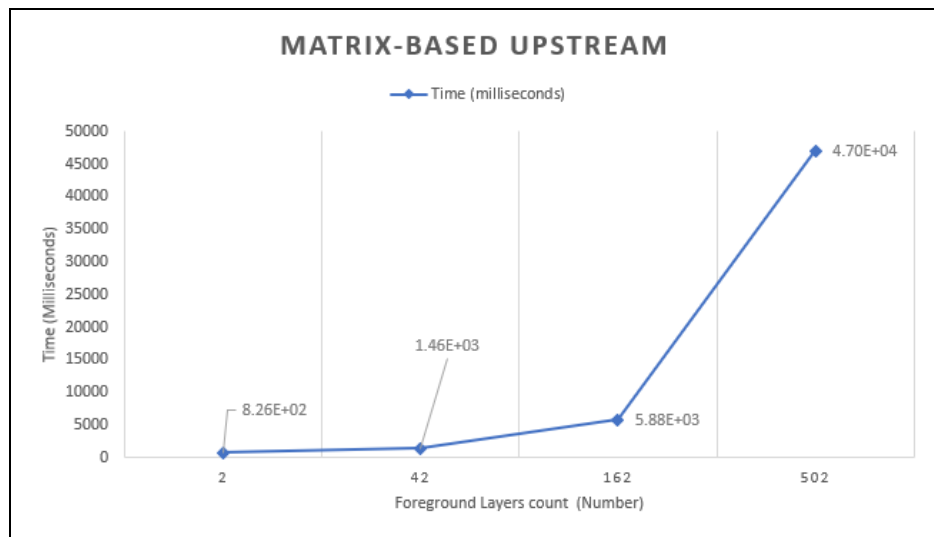
Figure 4.19 Variant size foreground layer influence on the performance
of the matrix-based upstream module

The execution time starts with a few seconds for up to 42,000 processes in the foreground layer. However, when adding half a million processes to the foreground layer, the execution time increases to 47 seconds.

Algorithm 2.4 allows reducing the matrix component of computing equation 2.9. When testing it on the same foreground sizes, a considerate reduction of execution time is observed. For instance, it provides with a reduction of the execution time from 47 seconds to 9.3 seconds for 500 foreground layers.

4.2.3.4   Stochastic phases

The next study assesses the influence on Monte-Carlo performance when varying the size of the foreground layer. Figure 4.20 shows the stochastic kernel performance for variant foreground layer size. Each iteration takes 105 milliseconds to compute. The main phase influencing the time delay is the solving phase. The size of the foreground layer is not influencing the remaining phases.
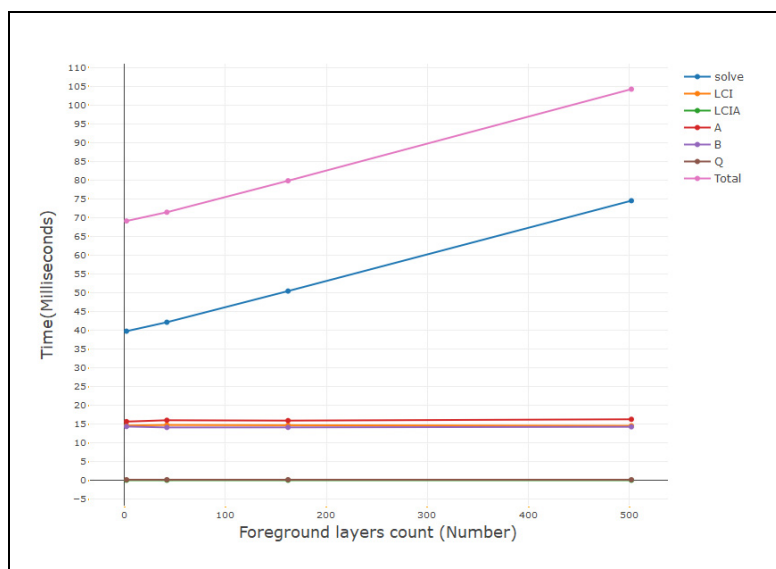
Figure 4.20 The influence of a variant size foreground layer
on Monte-Carlo performance using algorithm 2.9

4.3   **Discussion**

Table 4.6 lists the execution times for the analysis of processes from the Ecoinvent 3.3 database, and 1,000 iterations.

Table 4.6 Phase performance for Ecoinvent 3.3 process.

Production of Aluminium in Quebec

Phase	Time
LCADB loading	1.5 seconds
CalculatorData loading	1.5 seconds
Graph building	500 milliseconds
foundational static LCA	30 milliseconds
Contribution reports	2 seconds
Parallel Monte-Carlo – full sampling	1,000 iterations / 2.8 seconds. 48 milliseconds/iteration
Parallel SCC (1,000 iterations, 217,800 input, 18 output)	2 seconds

Table 4.7 Stochastic Phases performance for Pre-calculated background layer.

1,000 uncertain exchanges on the barrier

Phase	Time
Monte-Carlo	1 second
SCC (The background layer ranks are pre-loaded instead of ranked)	(1 second for ranking the barrier) + (1 second for the correlations)
Total	3 seconds

Table 4.7 shows the performance profile of pre-aggregated datasets, which provided an almost real-time performance for Monte-Carlo and SCC.

Table 4.8 shows the performance of the different phases, with 0.5 million activities added in the foreground layer. The sensitivity analysis for 1,000 iterations is almost invariant with the addition of the 1,000 uncertain cells of the foreground layer. The phases that are most influenced by the size of the foreground layer are Monte-Carlo and aggregate upstream reports.

Table 4.8 Performance for Variant size graph, 1,000 iterations, and
0.5 million nodes in the foreground layer

Group	Phase	Time
Initialization	LCADB loading	1seconds
	CalculatorData loading	5.5 seconds
	Graph building	3 seconds
Static	foundational static LCA	150 milliseconds
	LCI and LCIA Contribution reports	~2 seconds
	LCIA Aggregate Contribution reports	~ 10 seconds / 18 impact categories
Stochastic	Monte-Carlo	~ 20 seconds
	Parallel SCC (1,000 iterations, 217,800 inputs (background layer) + (1,000 from barriers connections), 18 outputs)	5 seconds

4.3.1   Algorithms validation

As shown in Figure 4.1, the calculator was able to provide with results similar to those of OpenLCA7. Also, Figure 4.3 shows the validation of our algorithms when using a foreground layer on top of the background layer. The validation is done with OpenLCA7, that uses the matrix method to perform the computation. This proves the validity of the following steps presented in section 4.3.1:

1. The Graph building;
2. The Graph traversal and the matrices loading;
3. The Matrix computing of equations 1.3, 1.4 and 1.5;
4. The LCI and LCIA calculation.

As shown in Figure 4.2, the stochastic phase is similar to Brightway2.0 except for some outliers. The difference is explained by the fact that using THE OpenLCA7 database will lead to some differences at the level of adopted impact factors and the uncertainty parameters that are being used.

By using algorithm 2.9, we can get statistical analysis that is with error in the order of 10^{-6}, which is a typical error that can occur when running two different Full Monte-Carlo simulations. This study confirms that our implementation can be used for uncertainty propagation based on pre-calculated aggregated scores. As will be shown later in this chapter, algorithm 2.9 can finish up to 10,000 iterations in less than one second. Having this performance enhancement; Figure 4.4 comes to show the validity of using algorithm 2.9.

4.3.1   Algorithms performance

➤ **Upstream report**

Figure 4.7 shows the Upstream contribution report taking 2.4 seconds to complete for EcoInvent3.0 processes. However, all traditional methods of computing matrix inverse show an execution time in order of minutes. Our solution provides much better performance than OpenLCA that implement the same report. Table 4.6 shows experimentations when running the scripts used by OpenLCA7 to compute the matrix inverse. To provide additional confirmation on the measured execution time, we developed a script written in SciPy python, and that computes the matrix inverse using "Scipy.linalg.in."

Table 4.9 Traditional A^{-1} scripts performance

	OpenLCA7		SciPY
Time \ Method	**A A^{-1} = I**	**BLAS**	**Scipy.linalg.inv**
Time (minutes)	5-10	3.5	3

A significant reduction in the execution time was brought using the optimization in equation 2.9, OpenMP parallelism, and the Eigen++ BiCGStab solver, as shown below in Table 4.10 and Table 4.9. Table 4.10 shows our continuous optimization for this problem.

Table 4.10 Thesis QBA^{-1} solvers performance for Recipe Midpoint I

	MUMPS	MUMPS-MPI	Eigen++ BICGStab	BICGStab-OpenMP
T [RHS]	4 s		28 ms	
T [18 RHS]	1 m	20 seconds	2 s	170 ms

As explained in section 2.2.8, our approach to computing this report consists of the following numbered steps shown on the x-axis of Figure 4.21:

1. Computing QBA^{-1} in parallel using OpenMP and BiCGStab; denoted by "mt" in Figure 4.21;

2. Transforming the cyclic graph into a non-repetitive acyclic graph sequentially; denoted by "lg" in Figure 4.21;

3. For each of the impact categories, applying the values of step 1 on the acyclic graph, generating and saving the report in parallel using OpenMP; denoted by "shares" in Figure 4.21.
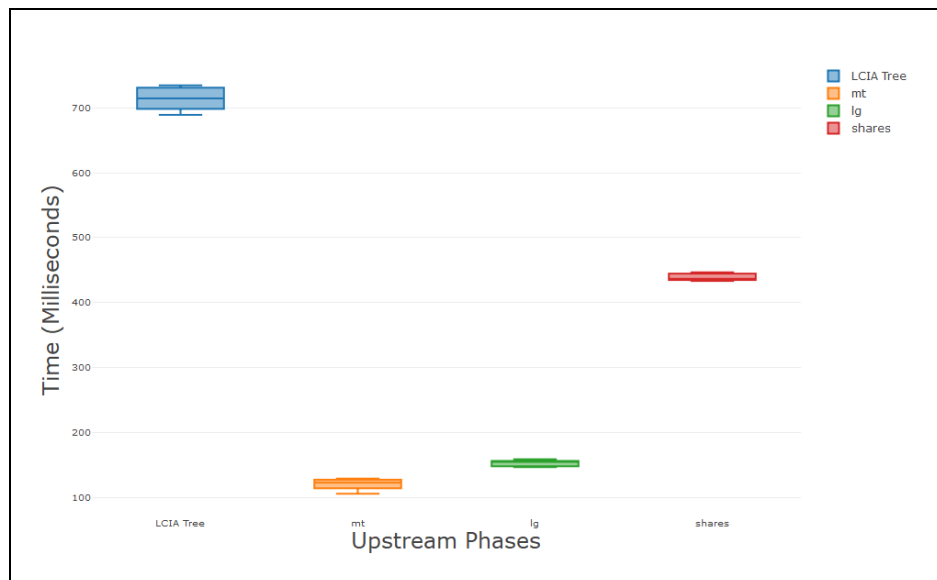
Figure 4.21 Upstream report sub-sections

To analyze the obtained performance, we first provide a breakdown of the Upstream report phases, as shown in Figure 4.21. The parallel algorithm and the optimization in equation 2.6 allowed for a reduction of performance from minutes (Table 4.10) to 120 milliseconds. This performance is attained because of the dimensionality reduction brought by equation 2.9.
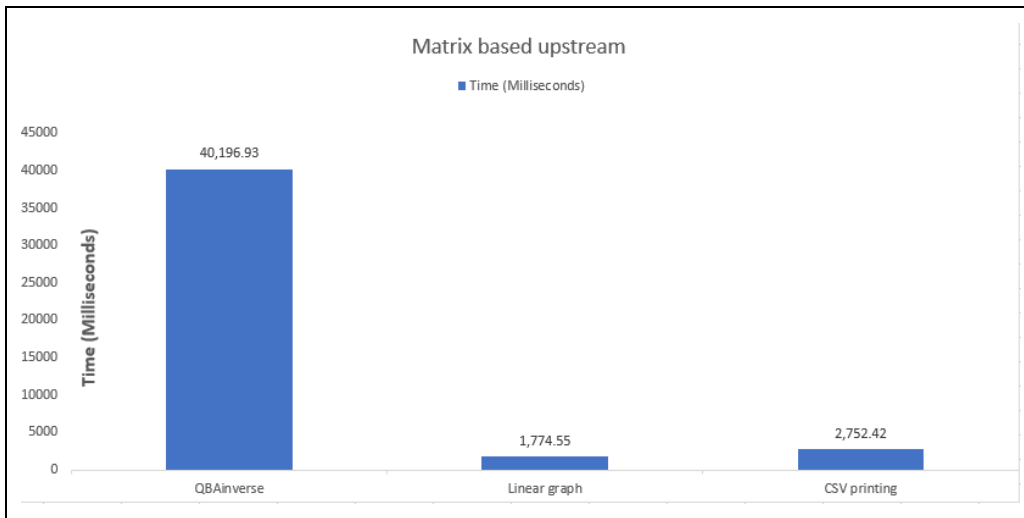
Figure 4.22 Aggregate upstream subsections

for a large foreground layer

Figure 4.19 shows a performance of 47 seconds for the upstream report when using a foreground layer of 502 thousand LCA processes. To understand this performance, Figure 4.22 provides a dissection of execution time over the report subsections. As it can be seen in this dissection, the phases that need optimization is the phase of computing QBA^{-1}. The time delay seen in Figure 4.22 is directly related to the matrix component that is still present in phase 1. With the increase of the foreground layer, the matrix component delay of equation 2.6 increases exponentially, which explains the jump of phase 1 from 170 milliseconds to 40 seconds.

The graph-based upstream method, in algorithm 2.24, removes the foreground layer from the matrix component in equation 2.9 by reverse propagating the scores on the graph. Algorithm 2.4 computes the upstream of a 0.5 million processes foreground layer in up to 4.8 seconds. The remaining phases remain unchanged. The total execution time of all the phases amounts for 9.3 seconds.

Monte-Carlo

To analyze the performance of the Monte-Carlo OpenMP implementation, we present below the scalability in Figure 4.23, the speedup of the computation in Figure 4.24, and the efficiency in Figure 4.25.

As explained in section 1.2.6, the *Scalability* is the ratio between the computation time when running on P' processors versus when running on P processors (i.e., P' is strictly less than P). In Figure 4.23, it is observed that when adding more threads, the scalability decays from an average of 5.5 (maximum scalability) to almost one (minimum scalability). This decay is due to the exchange a higher number of matrices by the MCS simulators with the increase of number of cores being utilized.
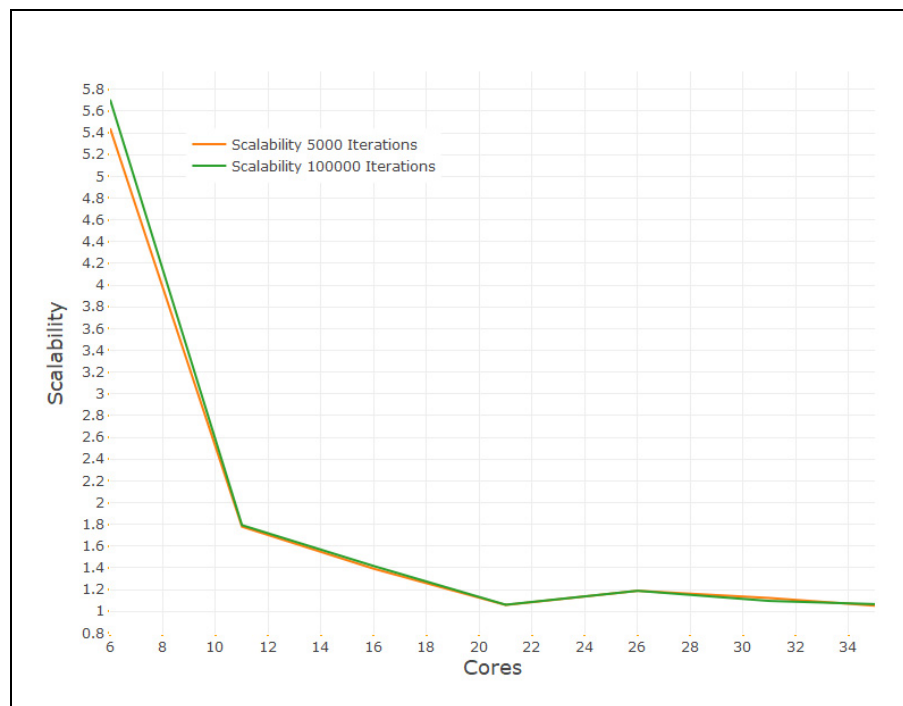
Figure 4.23 Scalability of Monte-Carlo OpenMP

The *Speedup* is the ratio between the execution time of scenario "A" running on P processors and the execution time of scenario "B" running on a P=1 processor. Figure 4.24 shows the

speedup of several Monte-Carlo simulations for a different number of iterations. Using our implementation of Parallel Monte-Carlo and when running on 35 cores 2.9 GHz server, we were able to get a speedup of up to 22 times faster for 100,000 iterations and 16 times faster for 1,000 iterations. From this experience, we observe that the speedup follows an initial linear trend and then, in a second phase, its acceleration deviates slightly from the first linear acceleration.

The speedup is 16 to 22 times faster for 1,000 and 100,000 iterations respectively. This speedup is associated with very good theoretical speedups, as shown in Figure 1.2 of the Amdahl speedup (Amdahl, 1967). Also, our experimental results are in accordance with the Gustafson law of accelerated speedup, which says that the speedup of an algorithm increases with an increase of the workload (i.e., the number of iterations) when using more computing resources. As shown in Figure 4.25, the higher it is the number of iterations, the higher it will be the speedup.
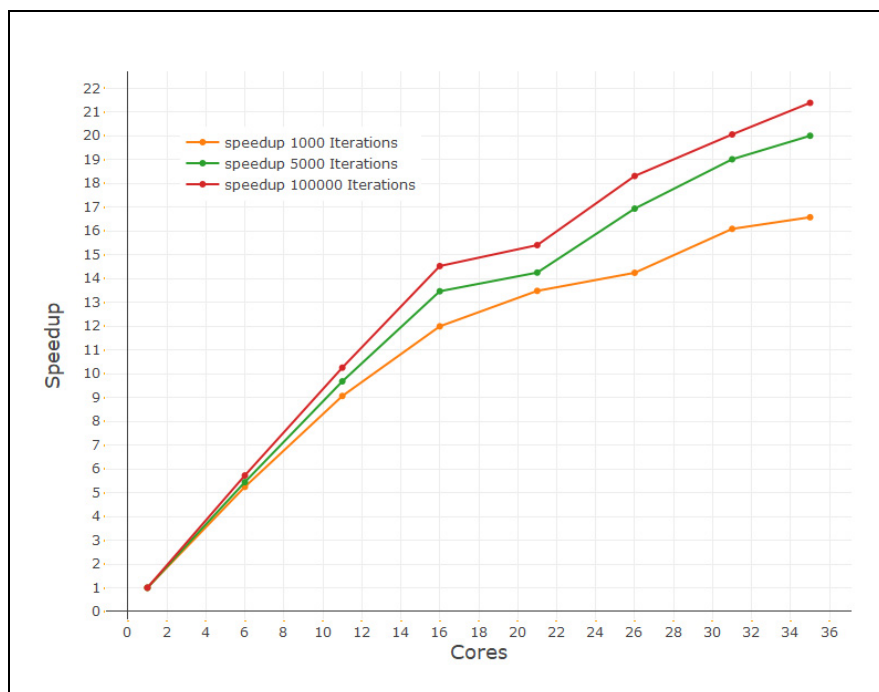
Figure 4.24 Speedup of Monte-Carlo OpenMP.
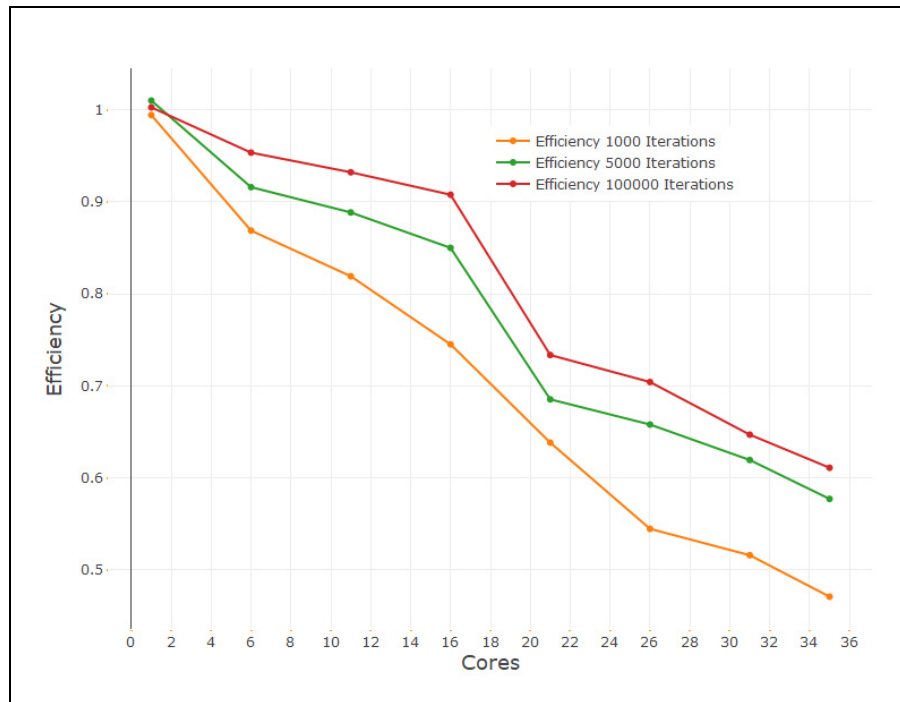
Figure 4.25 Efficiency of Monte-Carlo OpenMP

The *Efficiency* is the ratio of the speedup, when using P processors, over the number of processors P. Figure 4.25 shows that the speedup is at most reduced by half with the addition of more computing resources. For less than 16 cores, efficiency is the closest to one due to the linear speedup in that same interval, as shown in Figure 4.24.

➢ **Spearman rank-order sensitivity analysis**

Figure 4.12 shows the performance of the sensitivity analysis. Algorithm 2.6 takes a total time of 35 seconds for the Spearman rank GSA of the example of "Production of aluminum in Quebec, Recipe Midpoint I, 5,000 iterations". This total time is split between six seconds for the correlations of 3,920,400 vectors and 27.5 seconds for the ranking of 217,800 vectors with the size of 5,000 iterations each.

The ranking execution time is parallelized using OpenMP. The correlations are computed serially by applying equation 1.10 using the implementation in Figure 2.32 on the ranked

vectors. This implementation can compute a Pearson correlation of two vectors of 5,000 length in 1.5 microseconds (i.e. 6 Seconds / 3,920,400 Pearson correlations).

➢ **Variant size foreground layer: the use of the hybrid solver**

Algorithm 2.6 presented a method to solve an LCA system using a hybrid solver, reducing the matrix component in equation 1.3. As shown in Figure 4.16, this new algorithm provides an order of magnitude enhancement in the performance while solving the linear system in equation 1.3. When analyzing a system with 0.5 million processes in the foreground layer, the solving phase is computed in under 110 milliseconds. This solver allows the solving of extensive LCA systems using single threads and providing with similar execution times using matrix libraries that use multi-processes or GPU to reduce the execution time.

➢ **Pre-calculated aggregated datasets: the use of the hybrid solver**

Table 4.5 shows execution times for running algorithm 2.9 for processes from the background layer. If the foreground layer is static, the Monte-Carlo simulation will consist of solving the foreground layer only once and reading the background layer barrier scores in each iteration. This makes the Monte-Carlo simulation consist of mainly reading background layer barrier scores from memory and scaling the barrier processes, as explained in lines 3-9 of algorithm 2.8. When using algorithm 2.8, the computation complexity of each iteration will be equal to $T_{solving\ foreground\ layer} + T_{scaling\ bgl\_barrier}$. Scaling the background layer complexity will be as in equation 3.1 below.

$$N_{FLBarrier} * N_{FLBarrierDemands} * N_{ImpactCategories} * T_{Scaling\ Processe\ Supply} \qquad (3.1)$$

For processes from the background layer, equation 3.1 will become equal to $N_{impact\ categories} * T_{reading\ bgl\ process\ score}$. This operation is computed in order of hundreds of hundreds of microseconds or a factor of milliseconds. Having that this operation

needs to be repeated Monte-Carlo iterations times, this will lead to the execution time in the order shown in Table 4.5.

4.4   Conclusion

This chapter presented the results of the research, which are summarized as follows:

1. A graph parallel vs. a sequential implementation for building the graph (g) shows an enhancement of two factors when using parallelism with OpenMP. The graph is built in under one second for EconIvent 3.3 processes;

2. The foundational LCA for Ecoinvent 3.3 can be solved in less than 30 milliseconds using Eigen++ BiCGSTAB;

3. The aggregate upstream report is computed in under one second for Ecoinvent 3.3 processes;

4. The experiment of running Monte-Carlo in parallel using OpenMP shows a reduction of execution time by 95% when using the full capacity of the server;

5. SCC takes up to five seconds for Recipe MidPoint I with vectors of 1,000 iterations in size;

6. Using our Hybrid Solver algorithm, a Monte-Carlo simulation of 1,000 iterations, and for up to 0.5 million processes in the foreground layer, is computed in 20 seconds;

7. Using our implementation of MCS based on pre-calculated datasets, it can take up to one second for the stochastic analysis of activities from the background layer.

CONCLUSION

We summarise below our methods and their outcomes:

1. Design of a parallel algorithm for graph building that can build EcoInvent 3.0 database activities network in up to 500 milliseconds;
2. Design of a static calculation kernel that can return in up to 30 milliseconds for EcoInvent 3.0 database activities (i.e., an iterative matrix solving algorithm is being used);
3. Design of scalable upstream calculation that computes in up to two seconds for EcoInvent 3.0 database activities. This algorithm contains a mathematical optimization which allows for the removal of matrix inverse;
4. Design of scalable Monte-Carlo simulation that computes in up to 2.8 seconds for 1,000 iterations;
5. Design of scalable sensitivity analysis using Spearman rank that computes in up to 5 seconds for 1,000 iterations.

A unique component in our research is the design of a *Hybrid Solver,* which while solving the scalars of the graph (g), treats the foreground layer differently than the background layer. This newly designed algorithm solves the foreground layer using graph traversal and the background layer using an iterative or direct sparse solving library. This algorithm allows the removal of the foreground layer from the matrix computing component of the *calculation kernel*.

Having that the foreground layer is the component that increases in size and the background layer is the component that is always constant, the *Hybrid Solver* is providing with a solution for the problem of solving large LCA networks using traditional matrix methods. It returns in up to 110 milliseconds for one million LCA activity in the foreground layer using single threading.

This *HybridSolver* is equally used in all the other aforementioned modules. For basic LCI and LCIA, the inventory and scores of the foreground layer are aggregated while building the graph. Similarly, for the contribution reports, the calculator aggregates the inventory and impact scores per LCA activity while building the foreground layer. As for the Upstream contribution report, the foreground layer is removed from the matrix inverse, and instead, the cumulative scores are propagated by reverse graph traversal.

This single-threaded algorithm, the *hybrid solver*, allows for using the available computing resources to parallelize Monte-Carlo iterations, using multi-threading or multiprocessing, instead of for solving large linear systems. The research also used the *Hybrid Solver* algorithm along with recent research (Lesage et al., 2018) to design a newly proposed Monte-Carlo simulation algorithm based on pre-calculated aggregated datasets. This new algorithm can return in less than one second for EcoInvent 3.0 database activities and for several thousands of iterations.

ANNEX I

Simulation environment: Detailed CPU Information

LEDA lscpu:

Architecture: x86_64
CPU op-mode(s):   32-bit, 64-bit
Byte Order: Little Endian
CPU(s):   32
On-line CPU(s) list: 0-31
Thread(s) per core: 2
Core(s) per socket: 8
Socket(s):   2
NUMA node(s):   2
Vendor ID: GenuineIntel
CPU family: 6
Model: 63
Model name: Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz
Stepping: 2
CPU MHz: 1201.218
BogoMIPS: 4798.91
Virtualization: VT-x
L1d cache: 32K
L1i cache: 32K
L2 cache: 256K
L3 cache: 20480K
NUMA node0 CPU(s):   0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30
NUMA node1 CPU(s):   1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31

AWSC-4.9XLARGE : lscpu

Architecture: x86_64

CPU op-mode(s):   32-bit, 64-bit

Byte Order: Little Endian

CPU(s):   36

On-line CPU(s) list: 0-35

Thread(s) per core: 2

Core(s) per socket: 9

Socket(s):   2

NUMA node(s):   2

Vendor ID: GenuineIntel

CPU family: 6

Model: 63

Model name: Intel(R) Xeon(R) CPU E5-2666 v3 @ 2.90GHz

Stepping: 2

CPU MHz: 3259.475

CPU max MHz: 3500.0000

CPU min MHz: 1200.0000

BogoMIPS: 5800.18

Hypervisor vendor: Xen

Virtualization type: full

L1d cache: 32K

L1i cache: 32K

L2 cache: 256K

L3 cache: 25600K

NUMA node0 CPU(s): 0-8,18-26

NUMA node1 CPU(s): 9-17,27-35

ANNEX II

Calculator setup, code, and demos

Please refer to one of the following code source repositories for the calculator setup, code, and the thesis defence demos.

Table-A II-1 Calculator source code

Github (Permanent)	https://github.com/fsaab/ParallelLCA

ANNEX III

Static LCIA Raw Data

Table-A III-1 LCIA of Aluminium Production in Quebec.

using Ecoinvent 3.3 and for Recipe Midpoint I 2008

Impact Category	ParallelLCA	OpenLCA Results	Unit
Agricultural land occupation	0.132708	0.132708	m2*a
Climate change	5.434604	5.434604	kg CO2 eq
Fossil depletion	0.896965	0.896965	kg oil eq
Freshwater ecotoxicity	0.171363	0.171363	kg 1,4-DB eq
Freshwater eutrophication	0.001265	0.001265	kg P eq
Human toxicity	0.514999	0.514999	kg 1,4-DB eq
Ionising radiation	0.068914	0.068914	kg U235 eq
Marine ecotoxicity	0.109736	0.109736	kg 1,4-DB eq
Marine eutrophication	0.000864	0.000864	kg N eq
Metal depletion	0.214003	0.214003	kg Fe eq
Natural land transformation	0.00911	0.00911	m2
Ozone depletion	4.29E-07	4.29E-07	kg CFC-11 eq
Particulate matter formation	0.013761	0.013761	kg PM10 eq
Photochemical oxidant formation	0.018496	0.018496	kg NMVOC
Terrestrial acidification	0.040403	0.040403	kg SO2 eq

Table-A III-1 (Follows)

Impact Category	ParallelLCA	OpenLCA Results	Unit
Terrestrial ecotoxicity	0.00029	0.00029	kg 1,4-DB eq
Urban land occupation	0.042176	0.042176	m2*a
Water depletion	119.9292	119.9292	m3

Table-A III-2 LCIA of BuildingMD example raw data

Impact category	Open LCA	Thesis	Unit
Agricultural land occupation	88243.58	88243.6	m2*a
Climate change	3207.859	3207.86	kg CO2 eq
Fossil depletion	890.2247	890.225	kg oil eq
Freshwater ecotoxicity	28.174	28.174	kg 1,4-DB eq
Freshwater eutrophication	0.868838	0.868838	kg P eq
Human toxicity	151.9568	151.957	kg 1,4-DB eq
Ionising radiation	131.9615	131.961	kg U235 eq
Marine ecotoxicity	18.32533	18.3253	kg 1,4-DB eq
Marine eutrophication	0.925293	0.925293	kg N eq
Metal depletion	209.6124	209.612	kg Fe eq
Natural land transformation	10.45321	10.4532	m2
Ozone depletion	0.000362	0.000362	kg CFC-11 eq
Particulate matter formation	10.70907	10.7091	kg PM10 eq
Photochemical oxidant formation	21.65869	21.6587	kg NMVOC
Terrestrial acidification	15.80366	15.8037	kg SO2 eq
Terrestrial ecotoxicity	0.971627	0.971627	kg 1,4-DB eq
Urban land occupation	1074.446	1074.45	m2*a
Water depletion	9823.229	9823.23	m3

ANNEX IV

Stochastic LCIA Raw Data

Table-A IV-1 Aluminium Quebec - Uncertainty 1000 iterations.

Thesis Calculator

Impact category	Mean	SDV	Median
Marine ecotoxicity	0.170753489	0.15231693	0.05359716
Freshwater eutrophication	0.002147747	0.001554231	5.55E-04
Climate change	5.882832511	0.69429764	4.248995303
Natural land transformation	0.010669925	0.005567146	0.001861205
Urban land occupation	0.056444958	0.011248932	0.032664483
Ionising radiation	0.167214782	0.309454863	0.01338972
Terrestrial acidification	0.04577539	0.006661772	0.030965991
Metal depletion	0.253858314	0.039375393	0.157817017
Photochemical oxidant formation	0.022895299	0.006739409	0.011938956
Ozone depletion	5.70E-07	1.91E-07	2.67E-07
Terrestrial ecotoxicity	4.15E-04	1.04E-04	2.17E-04
Fossil depletion	1.009617571	0.147184932	0.663442434
Freshwater ecotoxicity	0.245909241	0.155172162	0.110190731
Particulate matter formation	0.015679701	0.001879907	0.011322448
Marine eutrophication	0.001044466	2.19E-04	5.41E-04
Water depletion	120.8707739	9.620452755	93.76178327
Human toxicity	0.887839723	1.56483666	0.301459297
Agricultural land occupation	0.19083448	0.062691394	0.088950498

Table-A IV-2 Uncertainty Aluminium QC - Percentiles - 1000 iterations.

Thesis Calculator

Impact category	2.5%	5%	25%	75%	95%	97.5%
Marine ecotoxicity	7.17E-02	7.67E-02	1.02E-01	1.90E-01	3.92E-01	4.64E-01
Freshwater eutrophication	7.70E-04	8.83E-04	1.29E-03	2.47E-03	4.93E-03	6.14E-03
Climate change	4.71E+00	4.87E+00	5.40E+00	6.33E+00	7.10E+00	7.31E+00
Natural land transformation	3.45E-03	4.29E-03	6.71E-03	1.33E-02	2.16E-02	2.56E-02
Urban land occupation	3.94E-02	4.12E-02	4.88E-02	6.26E-02	7.62E-02	8.39E-02
Ionising radiation	2.42E-02	2.78E-02	5.08E-02	1.80E-01	4.73E-01	8.16E-01
Terrestrial acidification	3.62E-02	3.77E-02	4.13E-02	4.89E-02	5.79E-02	6.10E-02
Metal depletion	1.87E-01	1.97E-01	2.26E-01	2.75E-01	3.24E-01	3.40E-01
Photochemical oxidant formation	1.50E-02	1.57E-02	1.88E-02	2.51E-02	3.50E-02	3.87E-02
Ozone depletion	3.34E-07	3.55E-07	4.40E-07	6.61E-07	9.39E-07	1.02E-06
Terrestrial ecotoxicity	2.79E-04	2.95E-04	3.43E-04	4.58E-04	6.14E-04	6.66E-04
Fossil depletion	7.63E-01	7.92E-01	9.03E-01	1.10E+00	1.27E+00	1.32E+00

Table-A IV-2 (Follows)

Impact category	2.5%	5%	25%	75%	95%	97.5%
Freshwater ecotoxicity	1.32E-01	1.38E-01	1.73E-01	2.71E-01	4.61E-01	5.70E-01
Particulate matter formation	1.28E-02	1.31E-02	1.44E-02	1.67E-02	1.89E-02	1.97E-02
Marine eutrophication	7.14E-04	7.56E-04	8.90E-04	1.16E-03	1.45E-03	1.60E-03
Water depletion	1.04E+02	1.06E+02	1.14E+02	1.28E+02	1.37E+02	1.40E+02
Human toxicity	3.61E-01	3.91E-01	5.35E-01	9.08E-01	1.61E+00	2.13E+00
Agricultural land occupation	1.13E-01	1.19E-01	1.49E-01	2.15E-01	3.11E-01	3.53E-01

Table-A IV-3 Aluminium Quebec - Uncertainty 1000 iterations.

Brightway2

Impact category	Mean	SDV	Median
water depletion	0.00612882	0.00127021	0.00593436
metal depletion	0.253632	0.0379391	0.249927
particulate matter formation	0.0156401	0.00194614	0.0153995
ozone depletion	5.73E-07	1.92E-07	5.32E-07
terrestrial ecotoxicity	0.000411882	0.000105904	0.000394007
human toxicity	0.781327	0.379053	0.690922
climate change	5.88899	0.638525	5.85579
terrestrial acidification	0.0458141	0.00732057	0.0444359

Table-A IV-3 (Follows)

Impact category	Mean	SDV	Median
marine ecotoxicity	0.122843	0.108319	0.0931855
freshwater eutrophication	0.00212048	0.00149014	0.00170793
urban land occupation	0.0571928	0.0120781	0.0552276
photochemical oxidant formation	0.0233231	0.00733524	0.0215735
marine eutrophication	0.00503042	0.0010372	0.00484095
freshwater ecotoxicity	0.130003	0.110021	0.100005
natural land transformation	0.0108157	0.00635225	0.00917953
fossil depletion	1.06587	0.155183	1.05053
agricultural land occupation	0.193462	0.0727514	0.178731
ionising radiation	0.161193	0.229809	0.0869785

Table-A IV-4 Uncertainty Aluminium QC - Percentiles - 1000 iterations.

Brightway2

Name	0.025	0.05	0.25	0.75	0.95	0.975
water depletion	4.23E-03	4.43E-03	5.25E-03	6.82E-03	8.40E-03	8.97E-03
metal depletion	1.91E-01	1.99E-01	2.26E-01	2.77E-01	3.18E-01	3.34E-01
particulate matter formation	1.27E-02	1.30E-02	1.43E-02	1.66E-02	1.93E-02	2.01E-02
ionising radiation	2.31E-02	2.69E-02	5.16E-02	1.71E-01	5.30E-01	7.73E-01
ozone depletion	3.28E-07	3.57E-07	4.54E-07	6.43E-07	9.00E-07	1.04E-06
terrestrial ecotoxicity	2.73E-04	2.87E-04	3.45E-04	4.52E-04	6.00E-04	6.41E-04
human toxicity	3.78E-01	4.11E-01	5.42E-01	9.26E-01	1.43E+00	1.67E+00
climate change	4.74E+00	4.88E+00	5.45E+00	6.30E+00	6.99E+00	7.29E+00
terrestrial acidification	3.64E-02	3.75E-02	4.09E-02	4.88E-02	5.94E-02	6.41E-02

Table-A IV-4 (Follows)

Name	0.025	0.05	0.25	0.75	0.95	0.975
marine ecotoxicity	3.74E-02	4.36E-02	6.29E-02	1.41E-01	3.00E-01	4.22E-01
freshwater eutrophication	7.33E-04	8.24E-04	1.28E-03	2.47E-03	4.54E-03	5.81E-03
urban land occupation	3.98E-02	4.14E-02	4.88E-02	6.36E-02	7.97E-02	8.52E-02
photochemical oxidant formation	1.48E-02	1.58E-02	1.86E-02	2.58E-02	3.74E-02	4.30E-02
marine eutrophication	3.47E-03	3.68E-03	4.32E-03	5.58E-03	6.87E-03	7.46E-03

Table-A IV-5 Uncertainty Aluminium QC - Percentiles - 1000 iterations.

Brightway2

Name	0.025	0.05	0.25	0.75	0.95	0.975
freshwater ecotoxicity	4.29E-02	4.74E-02	6.86E-02	1.49E-01	3.12E-01	4.27E-01
natural land transformation	3.23E-03	3.82E-03	6.54E-03	1.32E-02	2.36E-02	2.82E-02
fossil depletion	8.11E-01	8.44E-01	9.55E-01	1.16E+00	1.33E+00	1.42E+00
agricultural land occupation	1.13E-01	1.21E-01	1.52E-01	2.14E-01	3.09E-01	3.61E-01

Table-A IV-6 Uncertainties metrics Difference.

Brightway2 vs. Thesis calculator for the "Water Depletion" category

Impact Category	AVG	STD	Median	2.5	5	25	75	95	97.5
water depletion	121	9.62	121	104	106	114	128	137	140

ANNEX V

Aggregated Scores Error

Rows are the impact categories, X-Axis of Figure 4.4.

Table-A V-1 Error in using aggregated datasets Monte-Carlo
Raw data of Figure 3.4

AVG	STD	Median
1.3010e-05	3.12310e-05	2.8960e-05
9.4000e-07	7.69400e-07	9.3100e-07
2.3200e-11	6.12240e-11	2.8920e-11
4.1710e-06	8.35700e-06	4.0720e-06
1.8000e-06	5.70000e-05	3.0870e-05
1.0190e-03	2.19110e-03	9.1900e-04
5.6270e-07	7.90270e-06	8.8800e-08
7.5100e-07	2.11100e-07	8.7000e-07
6.5600e-05	3.60460e-04	2.4080e-04
1.5437e-07	4.94960e-07	1.0735e-07
3.2800e-04	3.41000e-05	3.0740e-04
3.9730e-08	4.22900e-08	6.1560e-08
1.1600e-05	2.86900e-05	9.5500e-05
2.7720e-05	1.05410e-04	9.2900e-06
1.7010e-06	1.98070e-06	1.5990e-06
3.2500e-09	1.36702e-07	2.9540e-08
8.6600e-06	5.75750e-03	1.4520e-06
3.8560e-05	1.55200e-05	4.6100e-05

Table-A IV-2 Error in using the aggregated datasets algorithm for Monte-Carlo.

Raw data of Figure 4.4

2.5 %	5 %	25 %	75 %	95 %	97.5 %
4.9280e-06	8.5600e-07	1.712e-05	2.5770e-05	1.4990e-05	2.5200e-06
1.0341e-06	1.0288e-06	3.745e-07	8.4600e-07	1.5030e-06	2.7170e-06
1.7180e-11	5.6400e-12	2.082e-11	3.4440e-11	3.8450e-11	5.2900e-12
2.9560e-06	3.1050e-06	3.349e-06	2.7660e-06	2.1860e-06	2.5130e-06
8.2660e-05	5.9440e-05	7.458e-05	3.3940e-05	1.2420e-04	9.3400e-05
1.5470e-03	1.0800e-03	9.460e-04	7.4900e-04	9.0400e-04	6.0300e-04
2.4590e-07	2.8290e-07	2.426e-07	3.4280e-07	4.9780e-07	2.1388e-06
1.4240e-07	5.3750e-07	2.285e-07	1.1360e-06	1.2140e-06	2.0120e-06
2.1200e-04	9.8500e-05	8.200e-06	2.2140e-04	3.7220e-04	1.1130e-04
6.4181e-08	9.3460e-08	3.509e-08	1.3563e-07	1.7020e-07	6.8640e-07
2.8970e-04	3.1140e-04	2.109e-04	3.3350e-04	1.8540e-04	4.4370e-04
1.1700e-09	5.8000e-10	4.188e-08	5.3650e-08	8.3300e-08	6.3720e-08
6.3800e-05	3.9800e-05	1.940e-05	7.2400e-05	1.3670e-04	1.8360e-04
1.1982e-06	3.5600e-08	1.742e-06	7.8300e-06	2.9555e-04	4.6253e-04
2.0000e-08	8.1500e-07	1.118e-06	1.6870e-06	4.7460e-06	7.8480e-06
4.4116e-08	4.0218e-08	1.006e-08	3.2320e-08	4.8300e-09	3.4850e-08
7.2000e-07	3.3330e-07	2.147e-06	5.9930e-05	2.8947e-04	9.5420e-04
5.4620e-05	9.2760e-05	3.676e-05	2.3710e-05	1.0831e-04	5.1180e-05

ANNEX VI Matrix solving code samples

```
1  static double* solve_umfpack(int dim, int Alength, int *rowsA,
2  int *colsA, double *dataA,double *b)
3
4
5  {
6    SMatrix m(dim, dim);
7    fillSparseMatrix(&m, Alength, rowsA, colsA, dataA);
8  long nz=m.nonZeros();
9  int* Ap=new int[dim+1];
10 int* Ai=new int[nz];
11 double* Ax=new double[nz];
12 int k = 0; int aPi=0;int prevcol=-1;
13 for (int j = 0; j < dim; j++)
14 {
15     for (int i = 0; i < dim;i++){
16         double v=m.coeff(i,j);
17         if(v!=0){
18             if(j!=prevcol){
19             prevcol=j; Ap[aPi] = k; aPi++;
20             }
21             Ai[k] = i;
22             Ax[k] = v;
23             k++;
24         }
25     }
26 }
27   Ap[aPi]=nz;
28   int i;
29   double *null = ( double * ) NULL;
30   void *Numeric;
31   int status;
32   void *Symbolic;
33   double* x= new double[dim];
34   status = umfpack_di_symbolic ( dim, dim, Ap, Ai, Ax, &Symbolic,
35   null, null );
36   status = umfpack_di_numeric (Ap, Ai, Ax, Symbolic, &Numeric,
37   null, null );
38   umfpack_di_free_symbolic ( &Symbolic );
39   status = umfpack_di_solve ( UMFPACK_A, Ap, Ai, Ax, x, b, Numeric,
40   null, null );
41   umfpack_di_free_numeric ( &Numeric );
42   return x;
43 }
```

Figure-A VI-1 Umfpack solving code

ANNEX VII CSV Files and LCA model schemas

Exchange {
 long _exchangeId,long _processId ,long _flowId ,bool _input , double _conversionFactor,double _amount ,string _amountFormula ,int _uncertaintyType,double _parameter1 ,double _parameter2 , double _parameter3 ,string _parameter1Formula,string _parameter2Formula,string _parameter3Formula,int _flowType ,long _defaultProviderId,string processType_,bool _isBackground, int _unitid
}

Flow {long _flowId,int _flowType,string _flowUid }

ImpactCategory {
int ICId; string ICName; string ICDescription; string ICReferenceUnit; int ICImpactMethod;
}

CalcParameter{
int param_id_,string ref_id_,string name_,long f_owner_,string scope_,bool input_, conversionFactor_,double value_, string formula_, int uncertaintyType_, double parameter1_, double parameter2_,double parameter3_, string parameter1Formula_,string parameter2Formula_,string parameter3Formula_
}

CalcImpactFactor{
long id; int imactCategoryId; long flowId; double conversionFactor; double amount; string amountFormula;
int uncertaintyType; double parameter1; double parameter2; double parameter3; string parameter1Formula; string parameter2Formula; string parameter3Formula;
int impactMethod = 0; int unitid = 0;
 }

Process {
long ProcId;string Name; string Process_Type;string ref_id; bool IsBackgroundLayer;
}

```
struct UnitOfMeasurement {

int UnitId; string UnitRefId; string SourceUnitName;double Factor; int DestinationUnitId;
string DestinationUnitRefId; string DestinationUnitName;

}
```

ANNEX VIII LCAIndex schema

LCAIndexes{

vector<long> ElementaryFlowsIndex;
unordered_map<long, long> ElementaryFlowsIndexIndices;

vector<long> IntermediateFlowsIndex;
unordered_map<long, long> IntermediateFlowsIndexIndices;
vector<long> ProcessesIndex;

unordered_map<long, long> ProcessesIndexIndices;
vector<long> FrontLayer_ProcessesIndex;

unordered_map<long, long> FrontLayer_ProcessesIndexIndices;
vector<long> BackgroundLayer_ProcessesIndex;
unordered_map<long, long> BackgroundLayer_ProcessesIndexIndices;
vector<long> FrontLayer_IntermediateFlowsIndex;
unordered_map<long, long>FrontLayer_IntermediateFlowsIndexIndices;
vector<long> BackgroundLayer_IntermediateFlowsIndex;
unordered_map<long, long>BackgroundLayer_IntermediateFlowsIndexIndices;
vector<long> FrontLayerBarrier_ProcessesIndex;
vector<long> BackgroundLayerBarrier_ProcessesIndex;

vector<long> ImpactCategoryIndex;
unordered_map<long, long> ImpactCategoryIndexIndices;

 }

BIBLIOGRAPHY

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference on - AFIPS '67 (Spring)* (p. 483). New York, New York, USA: ACM Press. https://doi.org/10.1145/1465482.1465560

Apache_Spark. (2017). BlockLU on Spark. Retrieved from https://issues.apache.org/jira/browse/SPARK-3434

Apache Spark. (2017). "Documentation Apache spark." Retrieved from https://spark.apache.org/docs/latest/

Arash Partow. (2010). EXPRTK library. Retrieved from http://www.partow.net/programming/exprtk/

Balay, S., Brown, J., Buschelman, K., Eijkhout, V., Gropp, W., Kaushik, D., … Zhang, H. (2014). *PETSc Users Manual Revision 3.4*. Argonne, IL (United States). https://doi.org/10.2172/1178104

Chapman, B., Jost, G., & Pas, R. van der. (2008). Using OpenMP: portable shared memory parallel programming. *Choice Reviews Online*, *46*(02), 46-0930-46–0930. https://doi.org/10.5860/CHOICE.46-0930

Dagum, L., & Menon, R. (1998). OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, *5*(1), 46–55. https://doi.org/10.1109/99.660313

Davis, T. A. (2004). Algorithm 832. *ACM Transactions on Mathematical Software*, *30*(2), 196–199. https://doi.org/10.1145/992200.992206

Davis, T. A., Rajamanickam, S., & Sid-Lakhdar, W. M. (2016). A survey of direct methods for sparse linear systems. *Acta Numerica*, *25*(April), 383–566. https://doi.org/10.1017/S0962492916000076

Davis, T. a. (2011). UMFPACK User Guide. *Engineering*, 1–140.

Dawes, B., Abrahams, D., & Rivera, R. (1998). "BOOST_MPI Documentation." Retrieved from https://www.boost.org/doc/libs/1_64_0/doc/html/mpi/

Flynn, M. J. (1972). Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, *C–21*(9), 948–960. https://doi.org/10.1109/TC.1972.5009071

Gaël, G., & Benoit, J. (2017). "Eigen - a C++ template library for linear algebra." Retrieved from http://eigen.tuxfamily.org

Geist, A., Gropp, W., Huss-Lederman, S., Lumsdaine, A., Lusk, E., Saphir, W., … Snir, M. (1996). MPI-2: Extending the message-passing interface (pp. 128–135). https://doi.org/10.1007/3-540-61626-8_16

Gopalani, S., & Arora, R. (2015). Comparing Apache Spark and Map Reduce with Performance Analysis using K-Means, *113*(1), 8–11.

Gould, N. I. M., Hu, Y., & Scott, J. A. (2005). *A numerical evaluation of sparse direct solvers for the solution of large sparse, symmetric linear systems of equations.*

GreenDelta. (2017a). OpenLCA BLAS. Retrieved from https://github.com/GreenDelta/olca-modules/blob/master/olca-core/src/main/cpp/olca_blas.cpp

GreenDelta. (2017b). OpenLCA matrixinverse. Retrieved from https://github.com/GreenDelta/olca-modules/blob/master/olca-core/src/main/cpp/olca_eigen.cpp

GreenDelta. (2017c). OpenLCA Usotream report. Retrieved from https://github.com/GreenDelta/olca-modules/tree/ca2c4a6f565af1fd8ed0653597ead3e77c278ddc/olca-core/src/main/java/org/openlca/core/results

Groen, E.A., Heijungs, R., Bokkers, E. A. M., & de Boer, I. J. M. (2014). Methods for uncertainty propagation in life cycle assessment. *Environmental Modelling & Software*, *62*, 316–325. https://doi.org/10.1016/j.envsoft.2014.10.006

Groen, Evelyne A., Bokkers, E. A. M., Heijungs, R., & de Boer, I. J. M. (2017). Methods for global sensitivity analysis in life cycle assessment. *The International Journal of Life Cycle Assessment*, *22*(7), 1125–1137. https://doi.org/10.1007/s11367-016-1217-3

Gupta, A., & Muliadi, Y. (2000). An experimental comparison of some direct sparse solver packages. In *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001* (Vol. 00, pp. 1901–1908). IEEE Comput. Soc. https://doi.org/10.1109/IPDPS.2001.925182

Gustafson, J. L. (1988). Reevaluating Amdahl's law. *Communications of the ACM*, *31*(5), 532–533. https://doi.org/10.1145/42411.42415

Heijungs, R. (1994). A generic method for the identification of options for cleaner products. *Ecological Economics*, *10*(1), 69–81. https://doi.org/10.1016/0921-8009(94)90038-8

Heijungs, R., Henriksson, P. J. G., & Guinée, J. B. (2017). Pre-calculated LCI systems with uncertainties cannot be used in comparative LCA. *International Journal of Life Cycle Assessment*, *22*(3), 461. https://doi.org/10.1007/s11367-017-1265-3

Heijungs, R., & Sun, S. (2002). The computational structure of life cycle assessment. *The International Journal of Life Cycle Assessment*, *7*(5), 314–314. https://doi.org/10.1007/BF02978899

Helton, J. C., & Davis, F. J. (2002). Illustration of sampling-based methods for uncertainty and sensitivity analysis. *Risk Analysis*, *22*(3), 591–622. https://doi.org/10.1111/0272-4332.00041

ISO. (2006). ISO 14040. *Environmental Management - Life Cycle Assessment - Principles and Framework*, (2006), 1–28. https://doi.org/10.1136/bmj.332.7550.1107

Jolliet, O., Soucy, G., & Houillon, G. (2010). *Analyse du cycle de vie : comprendre et réaliser un écobilan. Science & Ingénierie de l'Environnement* (2nd ed, Vol. 1).

Kung, H. T., & Leiserson, C. E. (1978). Systolic {A}rrays for {VLSI}. *{S}parse {M}atrix {S}ymp. {SIAM}*. Retrieved from https://apps.dtic.mil/dtic/tr/fulltext/u2/a066060.pdf

L'Excellent, J.-Y. (2017). *Multifrontal Massively Parallel Solver (MUMPS 5.1.2) Users' guide. Computer Methods in Applied Mechanics and Engineering*. Retrieved from mumps.enseeiht.fr/doc/userguide_5.1.2.pdf

Lesage, P., Mutel, C., Schenker, U., & Margni, M. (2018). Uncertainty analysis in LCA using precalculated aggregated datasets. *International Journal of Life Cycle Assessment*, *23*(11), 2248–2265. https://doi.org/10.1007/s11367-018-1444-x

Li, X. S. (2005). An overview of SuperLU. *ACM Transactions on Mathematical Software*, *31*(3), 302–325. https://doi.org/10.1145/1089014.1089017

MacLeod, M., Fraser, A. J., & Mackay, D. (2002). Evaluating and expressing the propagation of uncertainty in chemical fate and bioaccumulation models. *Environmental Toxicology and Chemistry*, *21*(4), 700–709. https://doi.org/10.1897/1551-5028(2002)021<0700:EAETPO>2.0.CO;2

Muller, S., Lesage, P., Ciroth, A., Mutel, C., Weidema, B. P., & Samson, R. (2016a). The application of the pedigree approach to the distributions foreseen in ecoinvent v3. *International Journal of Life Cycle Assessment*, *21*(9), 1327–1337. https://doi.org/10.1007/s11367-014-0759-5

Muller, S., Lesage, P., Ciroth, A., Mutel, C., Weidema, B. P., & Samson, R. (2016b). The application of the pedigree approach to the distributions foreseen in ecoinvent v3. *The International Journal of Life Cycle Assessment*, *21*(9), 1327–1337. https://doi.org/10.1007/s11367-014-0759-5

Mutel, C. (2013). *The new pedigree matrix numbers : do they matter ?* Retrieved from https://chris.mutel.org/images/Pedigree-matrix-poster-LCA-orlando.pdf

Naumov, M. (2011). Incomplete-LU and Cholesky preconditioned iterative methods using

CUSPARSE and CUBLAS. *Nvidia White Paper*, (May), 1–16.

Nichols, B., Buttlar, D., & Farrell, J. (2013). *PThreads Programming. O'Reilly* (1st ed.). O'Reilly Media. Retrieved from http://shop.oreilly.com/product/9781565921153.do

Nielsen, F. (2016). *Introduction to HPC with MPI for Data Science*. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-21903-5

Peters, G. P. (2007a). Efficient algorithms for Life Cycle Assessment, Input-Output Analysis, and Monte-Carlo Analysis. *The International Journal of Life Cycle Assessment*, *12*(6), 373–380. https://doi.org/10.1065/lca2006.06.254

Peters, G. P. (2007b). Input-Output and Hybrid LCA (Subject Editor : Sangwon Suh) Efficient Algorithms for Life Cycle Assessment , Input-Output Analysis , and Monte-Carlo Analysis, *12*(6), 373–380.

Qin, Y., & Suh, S. (2017). What distribution function do life cycle inventories follow? *The International Journal of Life Cycle Assessment*, *22*(7), 1138–1145. https://doi.org/10.1007/s11367-016-1224-4

Reyes-Ortiz, J. L., Oneto, L., & Anguita, D. (2015). Big Data Analytics in the Cloud: Spark on Hadoop vs MPI/OpenMP on Beowulf. *Procedia Computer Science*, *53*, 121–130. https://doi.org/10.1016/j.procs.2015.07.286

Rupp, K., Tillet, P., Rudolf, F., Weinbub, J., Morhammer, A., Grasser, T., … Selberherr, S. (2016). ViennaCL---Linear Algebra Library for Multi- and Many-Core Architectures. *SIAM Journal on Scientific Computing*, *38*(5), S412–S439. https://doi.org/10.1137/15m1026419

Saltelli, A., Ratto, M., Andres, T., Campolongo, F., Cariboni, J., Gatelli, D., … Tarantola, S. (2007). *Global Sensitivity Analysis. The Primer*. *Global Sensitivity Analysis. The Primer*. https://doi.org/10.1002/9780470725184

Schenk, O., & Gartner, K. (2014). Parallel Sparse Direct Solver PARDISO — User Guide. *PARDISO Project*, *0*, 0–65. Retrieved from http://pardiso-project.org/manual/manual.pdf

Spark MLlib, A. (2017). Apache Spark MLlib documentation. Retrieved from https://spark.apache.org/docs/latest/ml-guide.html

StackOverflowCommunity. (2017). Apache spark SVD matrix inverse. Retrieved from https://stackoverflow.com/a/29969522/9983159

Suh, S., & Heijungs, R. (2007). Input-Output and Hybrid LCA (Subject Editor : Sangwon Suh) Power Series Expansion and Structural Analysis for Life Cycle Assessment. *International Journal*, *12*(1997), 381–390. https://doi.org/10.1065/lca2007.08.360

Tarantola, S., Becker, W., & Zeitz, D. (2012). A comparison of two sampling methods for global sensitivity analysis. *Computer Physics Communications*, *183*(5), 1061–1072. https://doi.org/10.1016/j.cpc.2011.12.015

Tracy, F. T., Oppe, T. C., & Engineer, U. S. A. (2005). A Comparison of Several Direct Sparse Linear Equation Solvers for CGWAVE on the Cray X1, 1–6.

Valiant, L. G. (1990). A bridging model for parallel computation. *Communications of the ACM*, *33*(8), 103–111. https://doi.org/10.1145/79173.79181

von Neumann, J. (1993). First draft of a report on the EDVAC. *IEEE Annals of the History of Computing*, *15*(4), 27–75. https://doi.org/10.1109/85.238389

Walter, É. (2014). *Numerical Methods and Optimization*. (M. Al-Baali, L. Grandinetti, & A. Purnama, Eds.). Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-07671-3

Weidema, B. P., Bauer, C., Hischier, R., Mutel, C., Nemecek, T., Reinhard, J., … Wernet, G. (2013). *Data quality guideline for the ecoinvent database version 3*. Retrieved from http://www.ecoinvent.org/database/methodology-of-ecoinvent-3/methodology-of-ecoinvent-3.html

Wendykier, P., & Nagy, J. G. (2010). Parallel Colt. *ACM Transactions on Mathematical Software*, *37*(3), 1–22. https://doi.org/10.1145/1824801.1824809

Xiang, J., Meng, H., & Aboulnaga, A. (2014a). Scalable matrix inversion using MapReduce. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing - HPDC '14* (pp. 177–190). New York, New York, USA: ACM Press. https://doi.org/10.1145/2600212.2600220

Xiang, J., Meng, H., & Aboulnaga, A. (2014b). Scalable matrix inversion using MapReduce. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing - HPDC '14* (pp. 177–190). New York, New York, USA: ACM Press. https://doi.org/10.1145/2600212.2600220

Yang Liang, Liu, J., Cheng Fang, & Ansari, N. (2016). Spark-based large-scale matrix inversion for big data processing. In *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)* (Vol. 2016-Septe, pp. 718–723). IEEE. https://doi.org/10.1109/INFCOMW.2016.7562171

Zaharia, M., Franklin, M. J., Ghodsi, A., Gonzalez, J., Shenker, S., Stoica, I., … Venkataraman, S. (2016). Apache Spark. *Communications of the ACM*, *59*(11), 56–65. https://doi.org/10.1145/2934664