HPC Based Acceleration for Optimization of Predictive Models: Lithography Overlay Performance Modeling

Ozan Dogu Tuna

QCE-CE-MS-2019-25
o.d.tuna@student.tudelft.nl

Abstract

This thesis project achieves designing and comparing two parallel implementations for exhaustive grid search along a large model space to find the optimum mapping model for overlay predictions used in ASML lithography machines. The search algorithm leads to an effectively intractable problem as long as sequential implementation is concerned, but a parallel implementation using the technologies provided by ASML High Performance Cluster (HPC) pave the way to tackle the challenge. A number of parallel execution concepts have been developed using different frameworks that are exposed to the ASML HPC developer community by the platform maintainers. Among these concepts, the most promising ones with respect to a defined set of criteria have been chosen to carry on with the implementation effort. It has been shown that a PBS based Lab implementation can scale on HPC with a parallel efficiency of 66%, with most of the efficiency loss stemming from scheduler overhead. A second, Spark based Fab implementation has an increased efficiency of 82%, paving a way for speedup of almost 1700 x for a Spark cluster with 2048 cores. Moreover, It has been shown experimentally that performance scales linearly over the model space dimensions. Baseline sequential implementation is estimated to take, by extrapolation, 2590 hours to execute on a single core for a typical model space use case. Refactoring the sequential implementation to utilize multiple CPU cores through multiprocessing can drive execution down to 115 hours on a 24-core machine. The Fab parallel implementation executes the same use case in 1.6 hours, enabling exploratory and iterative approaches to modeling for data scientists and domain experts.

HPC Based Acceleration for Optimization of Predictive Models

Lithography Overlay Performance Modeling

by

Ozan Dogu Tuna

to obtain the degree of Master of Science

at the Delft University of Technology,

to be defended publicly on Tuesday December 17, 2019 at 16:00 PM.

Student number:	4198034	
Project duration:	September 1, 2018 – December 18, 2019	
Thesis committee:	Dr. ir. Z. Al-Ars,	TU Delft, supervisor
	Dr. R. V. Prasad,	TU Delft
	Ir. F. Valente,	ASML

This thesis is confidential and cannot be made public until December 31, 2020.

TU Delft

Contents

List of Figures v

List of Tables vii

Acknowledgements ix

1 Introduction 1

1.1 Motivation . . . 1

1.2 Thesis Scope and Organization . . . 2

2 Background 3

2.1 Overlay and Wafer Alignment Model . . . 3

2.2 Obtaining the Mapping Model . . . 5

2.3 Grid Search on Model Space . . . 8

3 Literature Survey 11

3.1 PBS . . . 12

3.2 PBS with MPI . . . 14

3.3 CUDA . . . 16

3.4 Spark . . . 19

4 Evaluation of Alternative Solutions 25

4.1 PBS Based Concept . . . 26

4.2 CUDA Based Concept . . . 28

4.3 Spark Based Concept . . . 29

4.4 Comparison Overview . . . 31

5 Implementation 33

5.1 Lab Implementation with PBS . . . 33

5.1.1 Parallel Programming Model . . . 34

5.1.2 Execution Model Avoiding Interprocess Communication . . . 36

5.1.3 Auxilliary Functions . . . 38
5.1.4 Application Profiling . . . 40
5.1.5 Performance Model . . . 48
5.2 Fab Implementation with Spark . . . 51
5.2.1 Parallel Programming Model . . . 51
5.2.2 Execution Model . . . 52
5.2.3 Auxilliary Functions . . . 52
5.2.4 Application Profiling . . . 55
5.2.5 Performance Model . . . 57

6 Results and Comparison **61**
6.1 Model Space Scaling . . . 61
6.2 Dataset Scaling . . . 64

7 Conclusions **67**
7.1 Future Work . . . 68

Bibliography **69**

List of Figures

2.1 Context diagram of sub-functions of wafer align. 4

2.2 A visualization of model space size with respect to the three main methods used to tackle overfitting. 9

3.1 Utilization of HPC with respect to departments within ASML. 12

3.2 HPC software stack. 13

3.3 Cuda hardware model elements in their respective hierarchy. 17

3.4 CUDA execution model along with the associations of its elements to the hardware model. 17

3.5 Architecture of a Spark cluster. 20

3.6 Overview of Spark stage and task breakdown. 22

3.7 An example execution DAG with stages, as shown in Spark WebUI. 23

4.1 Execution concept of PBS based implementation. 26

4.2 Execution concept of MPI-extended PBS implementation. 28

4.3 Execution concept of CUDA and H2O4GPU based implementation. 29

4.4 Execution concept of Spark based implementation. 31

5.1 Data and execution flow of industrialized sequential implementation. 34

5.2 Visualization of the increase in size of candidate model space as input data moves sequentially through modeling pipeline. 35

5.3 Visualization of high level programming model for the Lab implementation. . . 36

5.4 A Jupyter notebook used as a web interface designed for user to handle execution parameters. 39

5.5 Memory usage of single node application with respect to the number of atomic tasks ($|\Phi| = 50$), computed on a single CPU. 42

5.6 Multiprocessing performance of single node application with legacy backend with respect to the number of atomic tasks ($|\Phi| = 50$) computed at each execution by a varying number of CPU cores. 44

5.7 Multiprocessing performance of single node application with loky multiprocessing backend with respect to the number of atomic tasks ($|\Phi| = 50$) computed at each execution by a varying number of CPU cores. 44

5.8 Speedup of the multiprocessing single node the application with respect to the number of CPU cores, with a regression line to indicate the linear trend. 45

5.9 The portion of computation and overhead for a varying sizes of job array in Lab implementation. 48

5.10 Comparison of scaling behavior between using number of nodes per subjob as the scaling parameter and number of CPUs per node as scaling parameter for Lab implementation. 49

5.11 Graph for visualising the effect of overhead on speedup, based on the developed performance model for Lab implementation. 50

5.12 Visualization of high level programming model for the Fab implementation. . . 53

5.13 Visualization of Spark execution DAG for the Fab implementation. 53

5.14 Estimation of task execution time per number of CPU cores, using different sources of data. 59

6.1 Performance scaling of Lab and Fab implementations with respect to the model space size along dimension of cross-validation partitions set. 62

6.2 Performance scaling of Lab and Fab implementations with respect to the model space size along dimension of hyperparameter set. 63

6.3 Scaling results for the Fab implementation, investigating effect of number of rows in the input dataset to execution time, when the cluster size is kept constant. 65

List of Tables

3.1 ASML HPC hardware resources with respect to location. 11

4.1 Comparison of alternative solutions. 32

5.1 Average speedup and execution time of multiprocessing single node application with respect to the number of CPU cores. 45

5.2 The effect of job array size to execution performance for Lab implementation. . 47

5.3 Effect of per-node resource requests to execution concurrency for Lab implementation. 48

5.4 Profiling results of Fab implementation on a subset of the model space with varying numbers of atomic tasks per Spark task. 57

5.5 Profiling results for the Fab implementation, investigating effect of number of CPUs per node to performance, when the total number of CPUs in cluster is kept constant. 57

5.6 Profiling results for the Fab implementation, invesgating effect of number of cores per task (equivalently, executors per node) to performance, when the total number of CPUs in cluster is kept constant. 58

5.7 Table showing average execution time of one atomic task divided by the number of CPU cores used on a single node. 59

6.1 Comparison of performance model estimates and actual execution times for the Lab implementation, scaling in cross-validation dimension. 64

6.2 Comparison of performance model estimates and actual execution times for the Fab implementation, scaling in cross-validation dimension. 64

6.3 Scaling results for the Fab implementation, investigating effect of number of rows in the input dataset to execution time, when the cluster size is kept constant. 65

Acknowledgements

From Eindhoven, I would like mention a few people who helped me accomplish this thesis. I feel genuinely lucky that my thesis supervisor in ASML is Frederico Valente. He has the rare quality of offering his help to people with complete attention and without any expectation, and I was no exception. He was always patient at times when I got stuck with the simplest tasks and he was supportive when things looked grim. I would also like to thank Georgios Tsirogiannis for helping me formulate the problem and lending me the data to work on.

From Delft, I would like to express my sincere gratitude to my supervisor, Dr. Zaid Al-Ars for providing the academic perspective in this project. He works with the brightest young minds all day long, still somehow having the youngest soul in the room with his intelligence and enthusiasm. Besides, he is an excellent lecturer of my favorite courses in the department. This fact probably defined what my career will be about from today onwards.

But none of this would be possible, if not for two modest souls in Ankara who chose to dedicate their lives to their children. To my mom, I thank not only for this time of my life, but for all my times: past and future. This ”achievement” is nothing compared to hers for being hands down the best mother in the world... And I will never forget the time when I told my father my intention to take my comfortable work part time and go back to university to find myself a new path, at the age of 30. His response echoed the perfect father figure in 1970’s Turkish cinema: ”Don’t worry, I would sell my jacket to get you through university”. Many times in adult life, we pamper ourselves with the illusion that we earned our achievements exclusively by our self determination and effort. We are self-made women and men, so they say. But the truth is, we are only the few lucky ones to have a family just like in the movies.

At least I am.

1 Introduction

1.1. Motivation

One of the most critical performance metrics related to photolithography process for chip manufacturing is so-called “overlay”; the accuracy at which certain layers and structures in a chip overlap with each other [1]. This accuracy is critical for getting operational and efficient chips. Overlay is formally defined as the difference between the position vector of substrate geometry and the corresponding point of next mask pattern[2]. Errors in overlay directly impact the manufactured semiconductor devices’ reliability and yield.

As the industry is pushing for smaller structures on chips, higher accuracy (better overlay) becomes more and more important. Overlay performance has become even more critical recently, due to a combination of increasing pattern density and innovative techniques such as multiple patterning. As chip makers move to multiple patterning, the number of layers that have to be aligned is increasing exponentially[3], tightening overlay specifications to an unprecedented accuracy. Meeting these specifications for a combination of various layers and masks is a challenging task.

In modern ASML lithography tool, several techniques for improved patterning accuracy depend on utilization of model based corrections, where the models used result from highly accurate measurements on the wafer and several components of the machine. Wafer alignment, the process of performing the horizontal measurements of the wafer surface on the chuck at the measure side of a dual-stage design machine, wafer stage fiducial taken as a reference, is a crucial one of these techniques. The purpose of wafer alignment is to accurately determine position of the wafer with respect to the wafer stage as well as global shape of the wafer. The measurements are performed on alignment marks of the wafer with a special alignment sensor.

Performing on-product overlay measurements by exposing a wafer with the lithography tool are very expensive, as measurement routines should use wafers that qualify for certain warping specifications, and they require special patterns, exposure settings and recipes. Moreover, these overlay measurements have impact on availability from customer perspective. Therefore, models based on the collected measurement data are also expensive, while requiring high levels of predictive accuracy as they have direct effect on overlay performance. The measurement data should be used as effective as possible to achieve high performance goals.

Taking on the task of maximizing the expected prediction accuracy of a model using popular validation methods suffer from the curse of dimensionality and approaches of this kind lead

to intractable problems as the models (and the candidate model space) gets larger, given the ever-increasing demands on the model in conjunction with the accuracy specifications. Parallel processing techniques not only have the potential to offer a feasible solution to this curse, it can also provide scalability.

1.2. Thesis Scope and Organization

In this thesis, parallel programming design and execution of solutions for the sequentially intractable problem of finding the optimal overlay mapping model by exhaustive search on a model space is sought to be addressed. Two implementations are presented to scale out the modeling pipeline using different cluster solutions, namely PBS and Spark. This thesis focuses on the following research questions:

- Is it possible to scale out overlay mapping model search pipeline in a cluster environment efficiently? Where are the bottlenecks? What is the efficiency that can be achieved?
- What is the most effective framework available on ASML High Performance Cluster platform for scaling out a parallel modeling pipeline, with the shortest path to industrialization?
- How would a larger model space, scaling through orthogonal dimensions, effect the parallel execution time?
- How would input dataset size effect the parallel performance?
- Would it be possible to provide an accurate estimation of parallel execution times using a performance model so that users of application can have an idea beforehand?

The thesis is organized as follows. Chapter 2 provides background information and explains current state-of-art for overlay modeling techniques. A survey study for parallel execution frameworks available on ASML High Performance Cluster is presented in Chapter 3. Out of these frameworks, alternative solutions are developed and offered in Chapter 4, along with a comparison and choices for implementation. Chapter 5 explain the details of two implementations, goes deeper in chosen approaches to development and translates profiling results to a performance model for each implementation. Chapter 6 discusses results with respect to research questions and comparison of two implementations, along with verification of the performance models. Chapter 7 closes the thesis with conclusions and recommendations on future work.

2

Background

2.1. Overlay and Wafer Alignment Model

Wafer alignment consists of three subfunctions: horizontal stage align, wafer align and wafer readout. During horizontal stage alignment, the position of the sensor plate on wafer stage is determined. After horizontal stage alignment, wafer align is performed. Here, initially the wafer shape is captured in a coarse model, after which the final fine wafer alignment model is determined. During exposure, results of horizontal stage alignment and wafer alignment model are used to align image exposure optimally with respect to the layer in which alignment marks were printed, in order to reach optimal on-product overlay. Finally, overlay accuracy of the system (and of alignment function) can be determined by reading out the exposed wafers.

The purpose of wafer align sub-function is to determine the position and deformation of the wafer with respect to the stage, or to be more precise, to determine a wafer grid per exposure with respect to the wafer stage coordinate system. Wafer align is divided in two phases: Coarse Wafer Alignment (COWA) and Fine Wafer Alignment (FIWA).

COWA and FIWA are similar in concept in the sense that both use measurement information gathered from alignment markers located on the wafer. However, they differ in measurement accuracy due to the fact that the number and capture range of alignment markers that are used for FIWA is higher than COWA. COWA essentially takes an ”initial guess” of marker locations based on measurement information gathered while the wafer is placed on the stage, as its input. Similarly, FIWA uses COWA wafer model as a beginning point for its own measurement routine. A context diagram is shown in Figure 2.1 for visualization of the relationship between different steps of the process.

On top of the inter-field wafer model determined during FIWA, a wafer-to-wafer intra-field offset can be determined via intra-field wafer alignment (IFWA). The obtained wafer model is to be utilized on expose side for improved overlay accuracy.

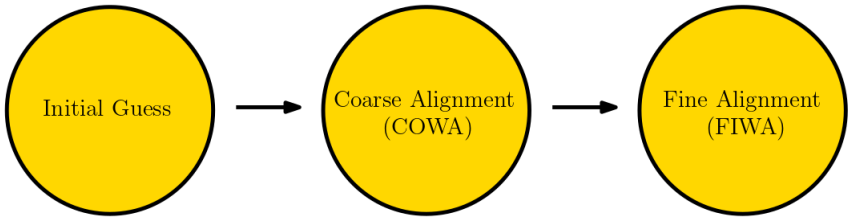


Figure 2.1: Context diagram of sub-functions of wafer align.

However, measurement based WA (wafer align) model contains errors, mainly caused by alignment mark deformation and various process residuals. Using WA model directly as exposure guidance can lead to low overlay accuracy. For this reason, a mapping model is used to translate a given WA model to an expose grid calculation with the objective of minimizing on-product overlay error, namely an OVL (overlay) model. Essentially, the mapping model translates measurement based results to an estimation of the real performance criterion; the overlay.

For a formal definition of mapping model, one has to define WA and OVL models first. High Order Wafer Alignment of third order (HOWA3) is the state of the art description for WA and OVL models, as a third order model is observed to be a good balance of accuracy (small residuals after modeling) and sensitivity to measurement errors.

In HOWA3 description, both WA and OVL models of each measured wafer are basically 10-term, third degree polynomials of nominal positions along each direction on horizontal plane of exposure, namely x and y. For both models, the polynomials in each direction have the following form:

$$x_m = T_x + M_{w,x}x + R_{w,x}y + p_{x(2,0)}x^2 + p_{x(1,1)}xy + p_{x(0,2)}y^2 + \dots + p_{x(0,3)}y^3 \tag{2.1}$$

$$y_m = T_y + M_{w,y}y + R_{w,y}y + p_{y(2,0)}x^2 + p_{y(1,1)}xy + p_{y(0,2)}y^2 + \dots + p_{y(0,3)}y^3 \tag{2.2}$$

where x_m and y_m are measured wafer align and overlay readout positions for WA and OVL models respectively. The transformation from nominal position plane to measured positions can be defined in terms of two isometries, namely translation and rotation, in combination with linear magnification. M_w and $R_{w,x}$ are the magnification and rotation factors for each direction. The linear model is then extended with higher order polynomial coefficients, namely p's with subscripts indicating their associated terms in x and y. The reason for the polynomial extension is to capture the higher order effects that have an increasingly important impact on ever tightening overlay accuracy specifications.

For modeling purposes, it is convenient to assume that there is a linear relationship between WA and OVL models. A mapping $\mathcal{M}$ of $(\mathcal{W}, \mathcal{O})$ defines this linear relationship between a given set of WA and OVL model coefficients. For mapping $\mathcal{M}$, The relationship $f(\mathcal{M})$ is defined as:

$$f(\mathcal{W}) = \mathcal{W}\mathcal{M} + \varepsilon(\mathcal{M}) = \mathcal{O}, \tag{2.3}$$

where $\mathcal{W}$ and $\mathcal{O}$ are matrix representations of WA and OVL models respectively, for multiple wafers. The columns of both matrices represent the polynomial coefficients, where each row represents the model extracted from a single wafer:

$$\mathcal{W} = \begin{bmatrix} T_{w1,x,w} & M_{w1,x,w} & R_{w1,x,w} & \cdots & T_{w1,y,w} & M_{w1,y,w} & R_{w1,y,w} & \cdots \\ T_{w2,x,w} & M_{w2,x,w} & R_{w2,x,w} & \cdots & T_{w2,y,w} & M_{w2,y,w} & R_{w2,y,w} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

$$\mathcal{O} = \begin{bmatrix} T_{w1,x,o} & M_{w1,x,o} & R_{w1,x,o} & \cdots & T_{w1,y,o} & M_{w1,y,o} & R_{w1,y,o} & \cdots \\ T_{w2,x,o} & M_{w2,x,o} & R_{w2,x,o} & \cdots & T_{w2,y,o} & M_{w2,y,o} & R_{w2,y,o} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

Unobserved random error with zero mean, that is the difference between the overlay model represented in $\mathcal{O}$, and overlay estimation of a mapping $\mathcal{M}$ is called the residual $\varepsilon(\mathcal{M})$.

For a typical application, both matrices $\mathcal{W}$ and $\mathcal{O}$ are about of size $(4m, m)$, where for HOWA3 type models, $n = 20$. Resulting overdetermined system is likely to only have an approximate solution, which can be obtained using the method of least squares, where the mapping $\mathcal{M}$ is to be estimated with a model based on a particular training dataset, $(\mathcal{W}_{train}, \mathcal{O}_{train})$. Method of least squares yields an optimal solution $\hat{\mathcal{M}}$ of $\mathcal{M}$, for which the cost function $J(\hat{\mathcal{M}})$ of the following form is sought to be minimized:

$$J\left(\hat{\mathcal{M}}\right) = \left\|\mathcal{W}_{train}\hat{\mathcal{M}} - \mathcal{O}_{train}\right\|_2^2 \tag{2.4}$$

Calculating the mapping model $\hat{\mathcal{M}}$ that minimizes $J(\hat{\mathcal{M}})$, is also known as ordinary least squares regression problem.

2.2. Obtaining the Mapping Model

Since the mapping model will be used for making predictions of overlay based on wafer align measurements for future wafers, any “fitted” model on the training dataset should be tested for its accuracy with out-of-sample data; how well the model can predict OVL based on future measurements of WA. Overfitting is a common danger for any predictive model based on regression analysis[4], and the quest of finding the best $\hat{\mathcal{M}}$ in terms of prediction accuracy is no exception. In the context of wafer alignment, a candidate mapping model $\hat{\mathcal{M}}_c$ is considered to be overfit on a training dataset, when it is more accurate in representing the relationship of the known data defined in Equation 2.3 (i.e. low cost function $J(\hat{\mathcal{M}}_c)$), but less accurate in making future predictions, compared to another candidate model.

A better understanding of overfitting can be given using the statistical concepts of bias and variance. For an arbitrary measurement represented in training dataset $(\mathcal{W}_{train}, \mathcal{O}_{train})$ and a mapping model $\hat{\mathcal{M}}$, assume that the function $f(\hat{\mathcal{M}}) = \mathcal{W}\hat{\mathcal{M}}$ estimates the relationship between $\mathcal{W}_{train}$ and $\mathcal{O}_{train}$ in the sense of least squares. Then, the expectation of the cost function on a test dataset $(\mathcal{W}_{test}, \mathcal{O}_{test})$ can be written as [5]:

$$
\begin{aligned}
E\left[J(\hat{\mathcal{M}})\right] &= E\left[\left\|\mathcal{W}_{test}\hat{\mathcal{M}} - \mathcal{O}_{test}\right\|_2^2\right] \\
&= \mathrm{Var}(\left\|\mathcal{W}_{test}\hat{\mathcal{M}}\right\|_2) + \left[\mathrm{Bias}(\left\|\mathcal{W}_{test}\hat{\mathcal{M}}\right\|_2)\right]^2 + \mathrm{Var}\,(\varepsilon)\,,
\end{aligned}
\tag{2.5}
$$

where $E[J(\hat{\mathcal{M}})]$ is the expectation of least squares error; the average error that one would result with if $\hat{\mathcal{M}}$ was to be obtained using a large training dataset $(\mathcal{W}_{train}, \mathcal{O}_{train})$ and tested with $(\mathcal{W}_{test}, \mathcal{O}_{test})$ for accuracy. Moreover, variance and bias in Equation 2.5 are, in fact:

$$
\begin{aligned}
\mathrm{Var}(\left\|\mathcal{W}\hat{\mathcal{M}}\right\|_2) &= E\left[\left\|\mathcal{W}\hat{\mathcal{M}}\right\|_2^2\right] - (E\left[\left\|\mathcal{W}\hat{\mathcal{M}}\right\|_2\right])^2, \\
\mathrm{Bias}(\left\|\mathcal{W}\hat{\mathcal{M}}\right\|_2) &= E\left[\left\|\mathcal{W}\hat{\mathcal{M}} - \mathcal{O}\right\|_2\right]
\end{aligned}
$$

The terms defined in above can also be seen intuitively [6].

Variance is a measure of how much the estimation $f(\hat{\mathcal{M}})$ would change, if $\hat{\mathcal{M}}$ was to be calculated using a different dataset $(\mathcal{W}_{train}, \mathcal{O}_{train})$. Ideally, a model's coefficients should not be varying a lot when computed using different datasets, as the underlying physical principles for wafer align and overlay measurements obtained from a particular lithography tool are not expected to vary from one dataset to the other, apart from process residuals and the element of random noise. A low variance model is thus expected to not overfit on the training data, and to have better predictive capabilities. In practice, simple models often outperform complex models in this regard.

Bias can be viewed as the predictive error caused by the simplifying assumptions of the modeling effort. It is unlikely that any model of finite order can completely capture the true relationship between wafer align and overlay measurements, and some modeling bias will remain while predicting overlay from wafer align data. Generally, complex models have less bias, or equivalently, are said to be less prone to underfitting the training data.

From the intuitive descriptions, it is natural to conclude that having a statistical model that simultaneously has low bias and variance is almost impossible. High-variance, low-bias models tend to overfit the training data as well as the noise. In contrast, algorithms that result in low variance typically aim for simpler models that have high bias and tend to underfit their training data, failing to capture important regularities.

There are a number of methods in literature that try to maximize predictive capabilities of a statistical model. At ASML, a three step procedure has been developed as a baseline algorithm for obtaining the mapping model that has the lowest out-of-sample deviance estimate, given the used methods. The procedure involves:

1. feature selection,
2. linear regression with regularization,
3. cross-validation.

The next sections will explain these three steps in detail.

Feature Selection

Overfitting is often a result of an excessively complicated model, and it can be prevented by fitting a given dataset multiple models with varying complexity and estimating each model's

predictive performance to achieve a good balance of bias and variance. Keeping the model structure intact, one can hard-constraint an arbitrary number of model coefficients to zero during regression, where each set of zero-constrained coefficients results in an associated, reduced order model. Zero-constrained coefficient(s) are effectively the coefficient(s) that are left out of the candidate model, resulting in a decrease of model complexity.

An exhaustive search seeks to build regression models with every possible combination of zero-constrained coefficients and recommend the one which has the best predictive capability with the right model complexity. In general, exhaustive search is the only technique guaranteed to find the model coefficient subset with the best evaluation criterion estimate. It is often the ideal technique when the number of model coefficients is kept small. Note that this number depends to some degree on the computational complexity of evaluating a model coefficient subset. For a model with m independent feature coefficients, there will be $2^m - 1$ regression models to choose from for each target.

Recalling that for HOWA3 type models where $m = 20$, an $m \times m$ mapping model $\hat{\mathcal{M}}$ can be seen as having 20 independent variables for predicting each OVL model coefficient from WA.

The problem with exhaustive search is that it is often a computationally intractable technique. Clearly, exhaustive search is not always practical in a sequential implementation, but a parallelized search algorithm can improve the performance drastically.

Linear Regression with Regularization

Especially in cases where the input data is noisy, a straightforward linear regression model will not generalize well into the future. Regularization methods are typically used to avoid this shortcoming[7]. Regularization is a form of regression, in our case linear, that constraints (ie. regularizes or shrinks) the model coefficient estimates towards a small absolute value. A regularization method typically discourages flexibility of the model to avoid overfitting. A regularized regression problem usually has a cost function of the form as follows:

$$\begin{aligned} &\min(c(J(\hat{\mathcal{M}}), \lambda, f_{reg}(\beta))), \text{ where} \\ &\quad c(J(\hat{\mathcal{M}}), \lambda, f_{reg}(\beta)) = J(\hat{\mathcal{M}}) + \lambda f_{reg}(\beta) \end{aligned} \tag{2.6}$$

For a given λ, a regularization cost function $f_{reg}(\beta)$ on model coefficient set β, and a cost function $J(\hat{\mathcal{M}})$ to measure the discrepancy of estimation $f(\hat{\mathcal{M}})$, the solution of Equation 2.6 yields an optimal regularized model. In a way, a regularization can be thought as putting a constraint on β while a linear regression optimization problem is being solved. Equations of the form $c(J(\hat{\mathcal{M}}), \lambda, f_{reg}(\beta))$ are referred to as constraint functions associated with the regularization method. Constraining behavior $f_{reg}(\beta))$ can be tuned by varying λ, also known as the hyperparameter of regularization.

One can easily see that regularization methods as described above involve the challenge of choosing optimal hyperparameter(s) λ. A traditional way to optimize hyperparameter(s) is a sweep through a manually selected hyperparameter set $(\lambda_1, \ldots, \lambda_l)$, denoted by Λ. This sweep suffers from the curse of dimensionality, but the optimization problem for each hyperparameter candidate is embarrassingly parallel with respect to another candidate.

Cross-validation

In principle, one could use some portion of the data to fit a model, then use another portion to evaluate the predictive capability of the model. Cross-validation is a method to evaluate predictive models by repeatedly partitioning the original sample into training sets to train the

model, validation sets to provide an unbiased tuning of various hyperparameters if applicable, and test sets to estimate its power of prediction.

Cross-validation is commonly used to estimate the prediction capabilities of any statistical model and to define a realistic upper boundary for out-of-sample deviance. In other words, it investigates how the model under question would perform in real life, when predictions have to be made on data that does not belong to the dataset used for building the model.

Cross-validation is particularly useful when one has dataset with a rather limited[8]. Another reason for its popularity is that cross-validation procedure is relatively simple to understand and implement. In fact, almost all popular machine learning libraries have some flavor of cross-validation already implemented for model validation and hyperparameter tuning purposes. The method has repeatedly proven itself to result in low biased models and/or less optimistic (more realistic) estimates of predictive performance of a model than using the whole dataset for both training and validation[9].

There are numerous ways of repeatedly using a dataset for cross-validation. A common way, and the way chosen for this investigation, is to holdout a portion of the dataset for final testing, and repeatedly create train/test splits of the remaining part of dataset and create a cross-validation split set, Φ, via random sampling. The procedure can be outlined as follows:

1. Reserve a predetermined portion of the dataset as test partition, define the rest of the data as cross-validation partition.
2. Randomly split the cross-validation partition into a preset ratio of train and validation subpartitions, for a predetermined number of times.
3. For each split of train and validation subpartitions:
   (a) Train a model on the training partition with a given model structure.
   (b) Evaluate the estimation capabilities of the model structure on the validation subpartition.
4. Calculate the average score of each candidate model structure over all splits and determine the model structure with the highest average score.
5. Train the winning model structure using the whole cross-validation partition.
6. Report the estimation performance of the model on the test partition.

Training and evaluation on each unique split of cross-validation partition is an independent computational process.

2.3. Grid Search on Model Space

The baseline procedure devised to achieve high predictive overlay accuracy effectively creates a space of candidate mapping models. The model space results from mutually independent choice of three methods listed below, all aiming in different ways to minimize out-of-sample deviance:

- constraining the models to different levels of complexity (feature selection)
- hyperparameter optimization for a given choice of regularization method
- using a given dataset partitioning via random sampling for cross-validation

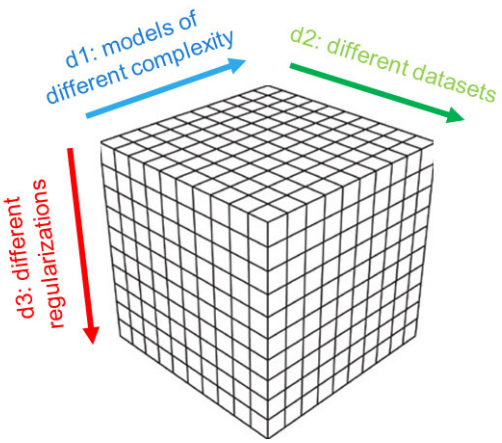


Figure 2.2: A visualization of model space size with respect to the three main methods used to tackle overfitting.

Given the problem at hand is finding the best mapping model $\hat{\mathcal{M}}$ for WA and OVL models defined in terms of HOWA3 polynomial coefficients $\mathcal{W}$ and $\mathcal{O}$, there are up to 2^{20} possible feature selection options for prediction of each 20 OVL model coefficient. Furthermore, for each feature selection option, a typical problem is observed to benefit from up to 10 regularization objectives (with unique hyperparameters) where data is randomly partitioned in 50 sets for cross-validation. The combination of the three aspects lead to a massive model space, see Figure 2.2. For each resulting model in the model space, a small linear system has to be solved.

Looking at the size of the model space of interest, one can either try to reduce the search space by using a set of heuristic rules. However, given the cost of the measurements that constitute the dataset and the accuracy requirements for overlay modeling, a better way is to exploit (almost) embarrassingly parallel nature of a grid search approach on the model space using high performance cluster hardware available in ASML. The next sections will investigate how this intractable problem for serial computing can be solved by parallel computing.

3

Literature Survey

There are a number of parallel programming environments, of which a selected bunch will be discussed in this chapter, that provide a varying range of abstraction and implementation complexity via an execution model for computation and synchronization of different processes among a parallel computation task. These environments potentially range in functionality, performance and low-level implementation access. The following sections investigate a subset of environments already integrated to ASML High Performance Cluster (HPC) in depth.

The current trend is that so-called supercomputers are made up of commodity components. These components may include high end CPU families such as Intel Xeon and computation oriented GPU hardware such as NVIDIA Tesla. The reason is that, the desktop class CPU, memory and GPU are fast enough, given a grid of these modest components are interconnected with low latency, high bandwidth networks. ASML HPC is an example of such a system. The resources of this supercomputer are divided into two locations in Eindhoven, namely Flight Forum and HTC. The key figures of capacity for ASML HPC is outlined in Table 3.1.

ASML HPC at the moment is being used in majority for analysis related use cases. These use cases include computational physics (thermal, electromagnetics, plasma, ray tracing optics) and metrology. It is being utilized in majority by Development Engineering and Research groups. Usage statistics by department is visualized in Figure 3.1.

Naturally, the use cases which have high business value and utilize HPC resources the most are historically prioritized during the deployment and maintenance of the software stack. As seen in Figure 3.2, the data residing in hardware layer is abstracted through a number of storage systems, have it being file based such as HDFS and GPFS or object based such as S3 and SWIFT. The computational resources of the system is accessible through a similar layer for abstraction, where various services are deployed.

Perhaps the most critical layer in the software stack, that influences the range of choices for technologies used in the higher layers such as applications and interfaces, is the resource

Table 3.1: ASML HPC hardware resources with respect to location.

	Flight Forum	**HTC**
CPU	3000+	4000+
GPU	N/A	30+
Storage	400 TB	400 TB

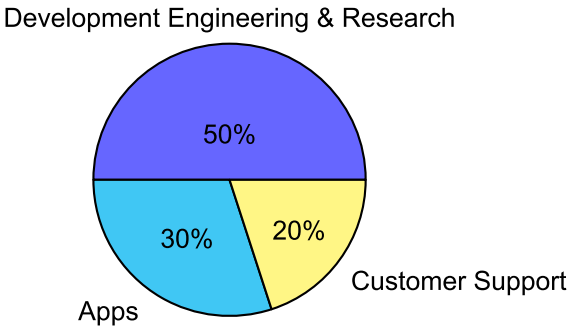


Figure 3.1: Utilization of HPC with respect to departments within ASML.

manager. PBS, that will be explained in detail in section 3.1, is chosen in view of the use cases that are deemed valuable by the departments utilizing HPC the most. PBS offers tight integration with commonly used analysis tools such as Ansys, Comsol, Staccm and MATLAB, offering parallel execution of typical workflows with complete transparency on the end user side. Therefore, there is a clear emphasis on analysis use cases for this decision.

However, data science is an emerging field in development of holistic lithography solutions. Most of the auxiliary hardware and software for holistic lithography are developed by Apps group in ASML, where data science, particularly for predictive performance modeling is used extensively. Various parallel frameworks have been deployed and offered to application developers, granted that the application has to request a set of resources contained as a job in a job queue, whose execution is handled by the resource manager layer. The parallel frameworks of HPC that are most mature in terms of deployment and usage are listed below:

- PBS ad-hoc usage through PBSDSH
- PBS and MPI
- Custom parallel frameworks for analysis applications (MATLAB MDCS, ANSYS HPC, COMSOL Cluster)
- PBS and CUDA
- PBS and Spark
- PBS and Dask with Dask.distributed

Out of the available options and based on the maturity level and previous experience in statistical modeling applications, PBS ad-hoc usage, MPI, CUDA and Spark has been chosen for further investigation. The following sections lay out the findings of the literature survey on these chosen technologies.

3.1. PBS

Sometimes, an embarrassingly parallel task with little to no interprocess communication requirements can be performed without the need of using sophisticated, higher level parallel computing frameworks, and a job scheduler such as PBS can be used instead in a simplistic Single Program Multiple Data (SPMD) fashion.

A job scheduler is an operating system level software component which typically allows specifying and scheduling the use of computing resources that will be allocated for executing the

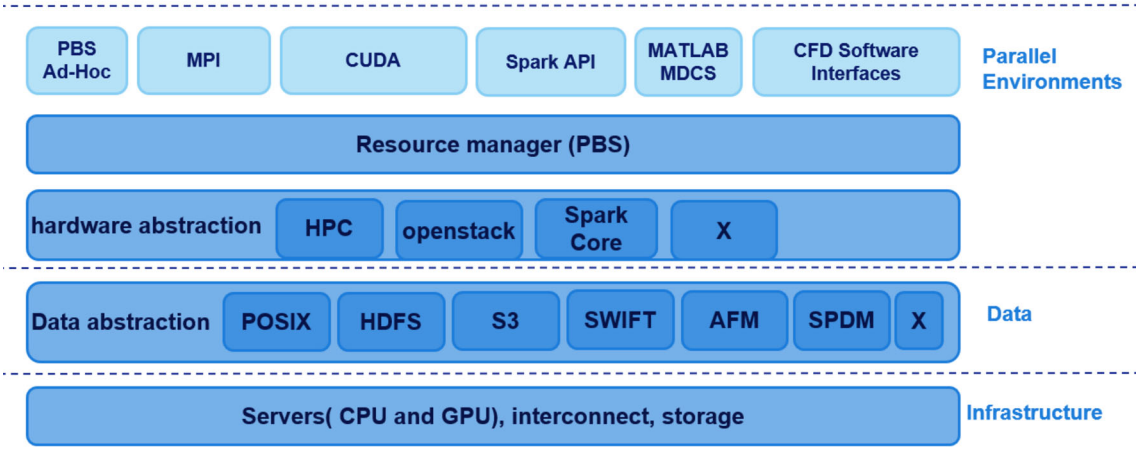


Figure 3.2: HPC software stack.

computing job that is requested. Most job schedulers also provide various functionality on the running job such as monitoring the execution and reporting back the outcome in standardized output or error messages. Implementations which exploit massive parallelism are typically run as batch jobs, if the job scheduler provides support for it. For most of the job scheduler software, starting the scheduling process for a job involves the following steps:

- preparing a submission script in a syntax compatible with the job scheduler,
- submitting the job to the scheduler for execution.

Portable Batch System (PBS) is a popular job scheduling software used in UNIX cluster environments. It optimally schedules and allocates a series of computations, also known as jobs, among a defined set of computing resources within a cluster. PBS is originally developed to manage aerospace computing resources at NASA[10]. Since its release to the public domain, PBS has been adopted in a wide variety of HPC applications and became a de facto standard in workload management of computing resources on Linux clusters[11]. PBS is also deployed and commonly used in ASML HPC.

Traditionally, UNIX cluster systems were used in an interactive manner where background jobs are merely processes with their input disconnected from the terminal. In time, as UNIX moved onto larger and larger clusters, the need to be able to efficiently schedule tasks based on available resources with maximum utilization increased in importance. With the progress in multiple fronts such as networked architectures and performance of commodity hardware, distributed parallel systems began to gain importance. At this point, UNIX alone began to be seen as insufficient for handling complex scheduling requirements presented by parallel computations running on clusters. NASA, driven by these requirements decided funding a development of a new resource management system that can span requirements for a range of parallel computing applications. A new task scheduling standard, named Portable Batch System (PBS) was developed[12].

The power of PBS emerges from three main focus points on its design[13]:

- Scalability: PBS supports millions of cores with fast job dispatch and minimal latency, and its scaling capabilities have already been tested on clusters made up of over 50000 nodes.

- Policy-Driven Scheduling: PBS enables system operators and application developers to optimize resource usage via policy definitions, by offering an environment that simplifies management of balancing job turnaround time and utilization with optimal job placement.
- Resiliency: PBS provides automatic fail-over architecture with no single point of failure. Jobs are never lost, and they continue to run despite failures.

In order to use the HPC compute nodes, one must first login to one of the nodes and submit a PBS job. The `qsub` command is used to submit a job to the PBS queue and to request necessary resources. The `qstat` command is used to check on the status of a job already in the PBS queue. To simplify submitting a job, one can create a PBS script and use the `qsub` and `qstat` commands to interact with the running job and PBS queue. A running job can be prematurely terminated using the `qdel` command.

3.2. PBS with MPI

The combined CPU power of a cluster can be harnessed using a dedicated parallel computing framework which supports sophisticated process architectures with fast interprocess communication routines. MPI is a cross-platform framework that is designed to facilitate communication and synchronization between multiple computers running a parallel program across a distributed memory model. It specifies message passing concepts and semantics, with goals of high performance, scalability and portability[14]. MPI is one of the most popular technologies for high performance computing applications[15].

An MPI library typically provides various functionalities such as:

- definition functions for a Cartesian or graph-like process topology,
- retrieving information on process topology (such as neighboring processes of an active process),
- definition of standard primitive data types for communication,
- point-to-point data send and receive operations,
- collective communication operations,
- synchronization between compute nodes,
- gather and reduce operations on partial results.

MPI programming model is built around the message passing paradigm, as the name suggests. Programs can be written in one of many popular languages that have native MPI bindings such as C and C++, or in languages such as Java or Python where various (non-standard) MPI implementations are available. A program written in one of these languages launch processes in traditional manner, where a partition of data is private to the process and resides in its local memory. These processes communicate data using MPI subroutines, over a network or locally on the same machine. This way, concurrency can be established.

The programming model provides a virtual process topology as well as communication and synchronization functionality between a set of processes that are mapped to computation nodes in a cluster environment. The processes that communicate via MPI can have a Single Program Multiple Data (SPMD) execution model, where same algorithm is executed by every process, but on a different part of data. However, these processes may also execute different programs. Hence, the MPI programming model allows multiple program multiple data (MPMD) applications too.

Due to the nature of its programming model, most MPI applications are built around a fixed the number of processes. These processes, all executing on different nodes of computation, communicate over the MPI Message Passing Environment. Therefore, understanding the capabilities and limitations of the communication environment provided by MPI is crucial to build high performing MPI applications. The following section will investigate this subject in detail.

MPI Message Passing Environment

MPI provides an environment for different processes on different compute nodes, possibly executing in different machines, to communicate and synchronize with each other. One can see an MPI application as a group of processes in isolation, executing independently and communicating via calls to MPI messaging subroutines. Most commonly used subroutines are the ones which facilitate point to point communications, collective communications and synchronization.

Point to Point Communication

Point to point communication happens between two processes, one specified as a sender and the other as receiver, explicitly. Point to point communications are often facilitated by the functions `MPI_Send()` and `MPI_Recv()`.

In MPI, point to point communications via send and receive calls start by the sender process initiating a message passing session to the receiver process. The sender process buffers up all the data into a single message that is to be sent to the receiver process. These data buffers are sometimes called envelopes, as a reference to a letter being packed inside an envelope before posting. After the data is buffered, it is passed to the network that is responsible for routing the message to the receiver, which is explicitly specified by the receiver process rank.

Collective Communication

Collective communication functions of MPI enable participation of all processes in a communicator. Under the hood, collective communications can be seen as a collection of point to point communications across all the processes in a communicator environment, that can be invoked in one subroutine call. Most widely used collective communication subroutine of MPI uses the broadcasting technique, and is called `MPI_Bcast()`.

Broadcast means that one sender process sends out the same piece of data to all processes in an MPI communicator. Broadcast is generally used to distribute configuration parameters, initial conditions and user input among the processes.

Another potentially useful scheme for collective communication is to use `MPI_Scatter()` to dispatch a subset of work from one node to a cluster of surrounding nodes, and then to use `MPI_Gather()` to collect the results. This is especially useful if the implementation concept involves numerous nodes acting as root processes where a subset of execution is orchestrated, rather than one node exclusively being responsible for collective communication handling.

Synchronization

A collective communication usually implies that more than two processes that are involved with the communication should synchronize among each other. In other words, all the processes should reach the same point in their execution, send and/or receive data through a collective communication routine and then continue execution. MPI has a dedicated function for synchronizing processes: `MPI_Barrier()`.

As the name implies, the function acts as a “barrier” where no single process in the communicator can pass the barrier until all processes in the communicator reach the same barrier. This way, synchronization can be achieved.

3.3. CUDA

While PBS and MPI are deployed in ASML HPC for CPU based applications, a number of nodes in the cluster are also equipped with multiple high-end graphics cards. General Purpose Computation on Graphics Processing Units (GPGPU) is a fast growing field of parallel computing with a lot of research interest. GPGPU paradigm strives to find high granularity parallelism in algorithms, as Graphics Processing Units (GPU) are designed to process massive amount of computation concurrently. Recent developments in GPGPU allow multiple GPUs to be used in harmony on the same task, increasing the concurrency even more. Within GPGPU research, implementing various machine learning and deep learning algorithms is an important subfield.

CUDA was launched by NVIDIA, one of the leading companies of GPU design and manufacturing, in 2007. Since then, CUDA is used for a variety of research topics, applications and industries to benefit processing power of GPUs in executing parallel algorithms. Examples include computer vision[16], high performance signal processing[17] and molecular simulations[18].

CUDA programming patterns are designed around a high level abstraction model of hardware, execution and memory. In order to understand how parallel computation is organized in a CUDA based implementation for GPU acceleration, one has to have a clear understanding of these models, as well as the associations between them[19]. The following sections will explain the details of CUDA hardware, execution and memory models.

Hardware Model

CUDA provides an abstraction of hardware depicted in Figure 3.3. Hardware model allows to write programs that are independent of the specific hardware architecture of the target GPU. This means CUDA programs that are written at present are expected to be compatible with future GPUs (although future architecture and resource improvements may allow further optimizations that can only be achieved by a change in code). Hardware model, in this sense, provides an platform that allows the programmer to express an execution schema, and therefore is tightly coupled to the execution model, see Figure 3.4.

The terms associated with the hardware model, in hierarchically ascending order, are as follows:

CUDA Cores

The CUDA cores, sometimes also referred as scalar processors (SP), are the primary building blocks in a GPU hardware grid. They are capable of performing basic integer, floating point, comparison and type conversion operations. Each CUDA core contains various functional compute units, and has fully pipelined architecture.

Streaming Multiprocessors

The association of code execution and CUDA hardware model is built around a scalable array of Streaming Multiprocessors (SMs), which are made up of CUDA cores. During the runtime of a CUDA program, the host CPU invokes a grid of GPU executed functions (kernels). Fixed size chunks of the function grid (blocks) are enumerated and distributed to SMs that have

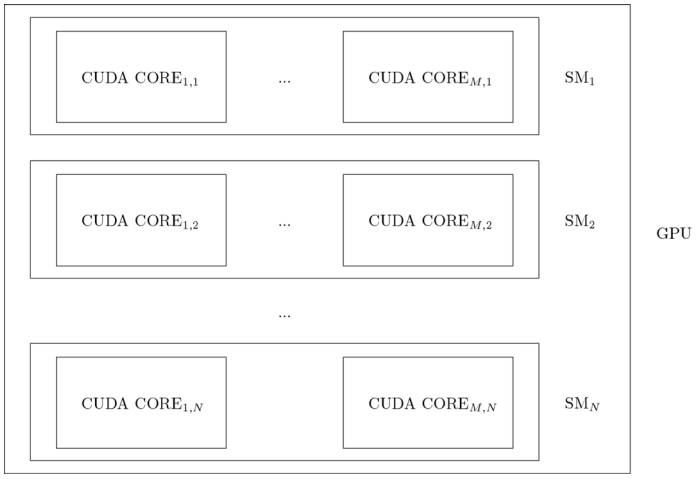


Figure 3.3: Cuda hardware model elements in their respective hierarchy.

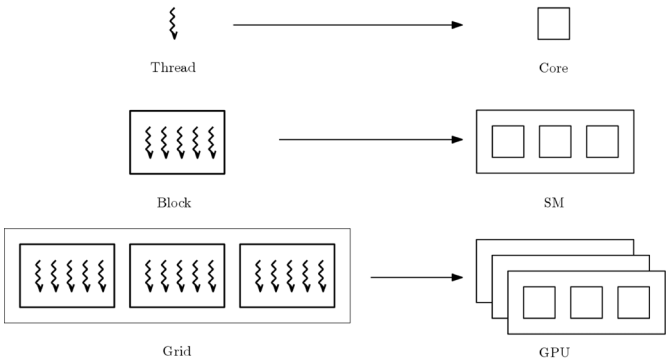


Figure 3.4: CUDA execution model along with the associations of its elements to the hardware model.

their resources available for computation. Multiple copies of a GPU executed function run concurrently on an SM. As blocks of functions terminate, new blocks are launched on the vacated multiprocessors.

GPU

A GPU is a collection of SMs. Each GPU has a unique identifier, where a CUDA program can specify the communication and execution between multiple GPUs through the use of these unique identifiers.

Execution Model

The execution model of CUDA provides an abstraction that partitions a given computation into multiple pieces, to enable performance and scalability of CUDA programs. This model, in conjunction with the hardware model, maps the inherent parallelism within an application to the highly granular architecture of the GPGPU hardware.

The functions that can be executed by the GPU are called kernels. Each kernel function is executed in a grid of threads; a fixed size partition of computation. A grid of threads is further divided into blocks, also known as thread blocks, and each block contains a set number of threads.

Thread

Each time a kernel is called in the GPU, the same kernel executes in parallel N times. This number is specified during the kernel call, and each execution of the kernel with an associated index is called a thread. Via this index, each thread can execute the kernel on a specific part of an indexed problem or dataset. Together, a group of threads can process the problem or dataset in parallel.

Block

In CUDA context, a group of threads are called a block. The size of a block is pre-defined along up to three dimensions, namely `blockDim.x`, `blockDim.y` and `blockDim.z`. Furthermore, every thread in a block is associated with an index in these dimensions. It is not under the control of the programmer to schedule execution of threads in a block. However, `_syncthreads()` can be used to make sure each thread in a block stops at a certain point until all threads of the block reach that point in execution.

Grid

A grid is a group of blocks. The size of a grid is fixed in two dimensions `gridDim.x` and `gridDim.y`. There is no way in CUDA to synchronize blocks of a grid.

Memory Model

A CUDA thread can access data in multiple memory spaces, following a hierarchy defined by the CUDA memory model. The performance of a CUDA program is highly dependent on its efficient usage of memory, as bandwidth between host CPU and target GPU is a common bottleneck. Thus understanding the specifics and constraints of the memory model is crucial.

Global memory

This memory is accessible to all units of a GPU. Any thread, regardless of which core it is running on has read and write access to the whole address space of the global memory.

Texture Memory

Within GPU, there is the texture memory (also called texture cache) that can be accessed by threads in read-only mode. Texture memory can be used in different address modes and can provide data filtering for some data formats.

Constant Memory

Similar to the texture memory, this is a read-only space within each GPU. Constant and texture memory spaces are optimized for different memory usage scenarios.

Shared memory

This is a small memory within each SM that can be read/written by any thread in a block assigned to that SM.

Local Memory

Each thread has private local memory. As opposed to higher memory spaces in the memory model hierarchy, local memory spaces are not persistent across kernel launches by the same application.

3.4. Spark

Spark is a general purpose distributed computation framework that was initially developed in 2012, with the aim to overcome certain limitations arising from the computational model of Hadoop MapReduce. Being the most popular cluster computing paradigm at the time, Hadoop MapReduce suffers from executing many disk operations for resilience[20], as it relies on a model that roughly consists of reading input data from persistent storage, run the data through mapping and reducing functions in order and write results back to the storage on each step. In contrast, Spark carries out consecutive operations on memory and ensures resilience by keeping a lineage of the operations performed on its data model. Furthermore, Spark provides a better framework for iterative algorithms such as predictive model training algorithms that are relevant to this thesis [21]. Spark ships with comprehensive libraries for SQL, machine learning, graph computation and stream processing, therefore is suitable for a wide range of use cases. The core functionality is written mainly with Scala and Java. Nevertheless, popular data science programming languages, namely Python and R, are also supported.

Apache Spark has as its architectural foundation, the resilient distributed dataset (RDD): a read-only multiset of data items distributed over a cluster of machines. RDDs are maintained in a fault tolerant way while providing a scheme similar to distributed shared memory, for high performance in-memory operations. This feature renders Spark fundamentally different than MapReduce, where dataflow involves writing the results back to disk every time a map operation is performed. Over time, Spark library has added higher levels of abstraction over RDDs, such as Dataframes and Datasets[22]. Nevertheless, underlying these APIs are still RDDs, and they are in a sense the heart of Spark. RDDs are distributed along the cluster by splitting the data into partitions that are ideally small enough to store in-memory on cluster nodes.

Two types of operations can be performed on an RDD: transformations and actions. Transformations map an existing RDD to a new dataset (also an RDD), where actions return a value to the driver program as a result of a computation performed on the dataset. Spark employs lazy execution strategy, meaning that (a series of) transformations are not executed until an action is instructed by the driver program.

In order to run Spark in a cluster, one would need a cluster manager and a distributed storage system already deployed. Spark comes with its own cluster manager, although a number of other popular cluster managers such as YARN and Mesos are also supported. For the scope of this work the built in cluster manager of Spark is used, due to the sheer reasons of availability and seamless integration. For a distributed storage system, ASML HPC provides HDFS, which is also the storage solution suggested for high performance Spark applications.

Standalone Mode Cluster Management Architecture

Spark ships with a simple cluster manager that makes the setup of a cluster very straightforward and reduces dependency to external code. Running a Spark cluster with the built-in cluster manager is commonly known as ”standalone mode”. When running in Standalone mode, Spark follows a generic master-slave paradigm with two types of execution daemons (driver and worker) and a separate cluster manager (master) deamon. Each deamon is a Java Virtual Machine (JVM) instance. A spark cluster has a single driver and master, and an arbi-

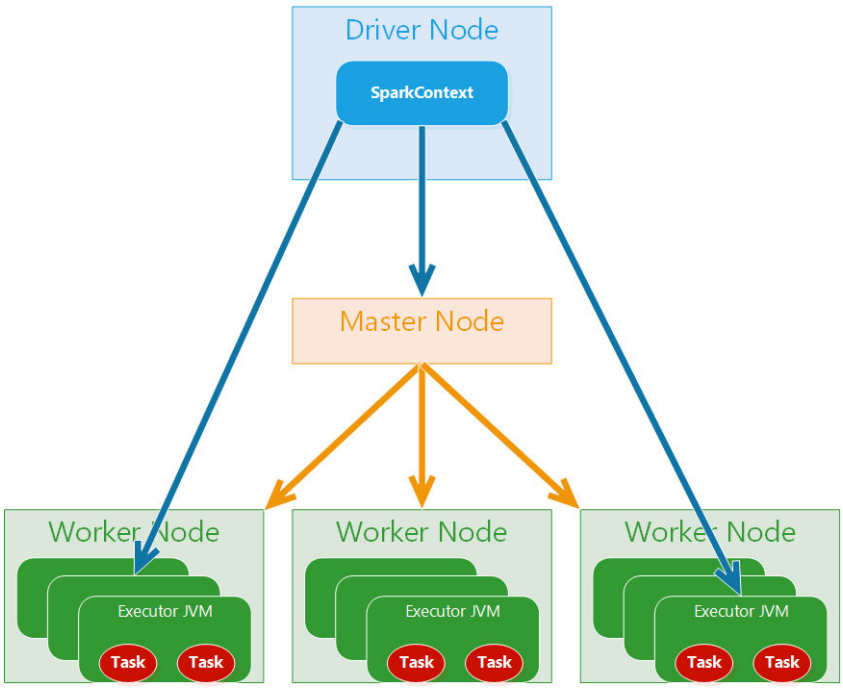


Figure 3.5: Architecture of a Spark cluster.

trary number of workers. The driver JVM instance can run locally on a random worker node of the cluster (cluster deploy mode) or on the machine where Spark application is launched (client deploy mode).

Figure 3.5 visualizes the architecture of a Spark cluster. The following sections will explain the function of the nodes as well as relation between them for a standalone mode Spark cluster in detail.

Driver Node

Driver node takes the role of the master during execution of a Spark application. It runs the driver JVM process with `SparkContext`, which is an abstraction for entry point to every functionality that Spark provides. An application is a Spark application if and only if it refers to a `SparkContext` at some point during its execution.

Apart from `SparkContext`, the driver node provides various services for orchestration of the job and task execution on the cluster, such as `DAGScheduler` and `TaskScheduler`. The driver node is also responsible for executing `main()` function of the application, which implicitly means that each action on an RDD populates the results on the driver node.

Worker Nodes

Worker nodes run (possibly multiple) executor JVM processes, and therefore are responsible for execution of computational tasks. They are essentially the compute nodes of the Spark cluster. Executor processes are responsible for data read/compute/write operations, while storing the compute results in-memory with an option to leak into disk.

By default, Spark employs static resource allocation to executor processes, which means that the workers spawn executors, bind them to their resources and keep them alive until the driver exits the application. However, dynamic resource allocation of executor processes

is also supported, given that the Spark is configured to execute an external service where shuffle files are preserved, so that executors can be spawned and terminated on demand.

Master Node

Despite the name, the master node's responsibility is not related to the execution model of Spark, for that role is fulfilled by the Driver node. Master node is the cluster manager in Spark Standalone mode, and is responsible for allocating resources to Spark applications that are submitted through a script.

To launch a Spark Standalone cluster, configuration files which containing the address of worker should be created in the Spark directory. It is possible to specify no worker nodes in the configuration, in which case the application launches on `localhost` as a single machine application. This mode is particularly useful for testing and debugging of an application before trying it on a real cluster.

Execution Model

The execution of a Spark application depends on a number of concepts related to how Spark achieves in-memory computation and fault tolerance while performing data transformations. For describing an execution model for Spark, these concepts should be well understood. The following sections will lay out the preliminary knowledge that leads to Spark's execution model, and then will go into details of the main subject.

DAG Scheduling of Transformations

The main reason for introduction of a directed acyclic graph (DAG) based scheduling in Spark is the limitations of Hadoop MapReduce that Spark has embarked to tackle. Computation in MapReduce goes, roughly as follows:

1. Read data from HDFS.
2. Apply map and reduce operations.
3. Write the result back to HDFS.

Each map and reduce cycle is treated independently, where the order and interrelation of each cycle is not taken into account during execution. However, for example, a map and reduce operation can be only one identical iteration in a bunch. It may be unnecessary, even wasteful in terms of resource usages of disk and memory, and may cost performance to write back the results of one cycle immediately after execution. Spark tackles this shortcoming of MapReduce by introducing a scheduling concept based on usage of directed acyclic graphs to describe the execution flow.

In Spark, DAG represents the sequence of computation steps performed on an RDD that is partitioned across the cluster, where each node in the graph is an individual RDD partition and an edge is a transformation performed on this partition. The graph is directed in the sense that the transformations shift the state of the data partition they act on from one form to another, and acyclic, hence a transformation on data cannot map to a past state.

This graph abstraction provides the basis for the execution model that replaces Hadoop MapReduce multistage execution model. In multistage execution model, the jobs are executed in steps of map and reduce operations, where each step waits for the previous one to finish and write its output to disk. As a result, a complex computation can require long

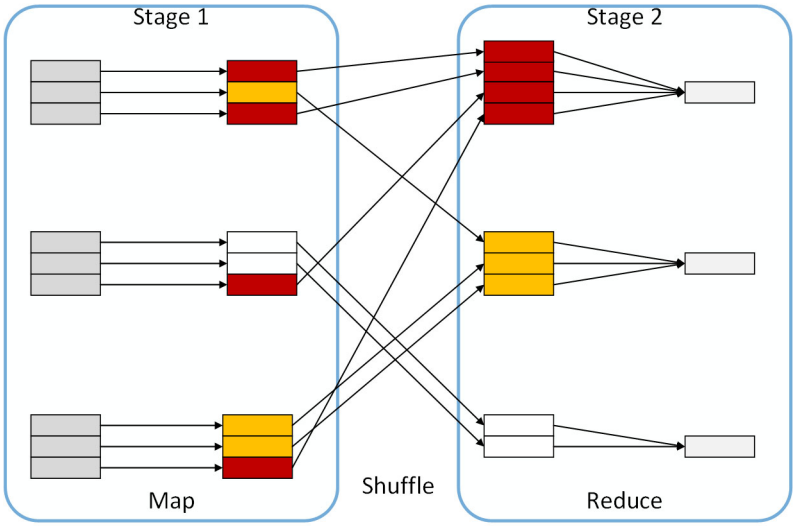


Figure 3.6: Overview of Spark stage and task breakdown.

time with small data volume. In contrast, Spark can apply various optimization routines on DAG to optimize an execution plan by minimizing disk read and writes as well as data shuffle between cluster nodes. Similar optimization techniques can be applied only manually in MapReduce paradigm.

To minimize shuffle and disk access, the notions of stages and tasks have been introduced in Spark execution model of a computational job. The next section investigates these notions.

Stages, Tasks and Jobs

In Spark, jobs of an application are executed in stages, where each stage is composed of identical tasks acting on the partitions of an RDD. It is important to note that Spark transformations come in two flavors, narrow and wide, and this characteristic of a transformation define how a job is supposed to be divided into multiple stages.

Remembering that transformations are edges of a DAG where the nodes are RDD partitions, a transformation is expected to create a new RDD from an existing one. Narrow transformations are the ones where moving data between RDD partitions is not necessary; a resulting RDD partition has only one parent. Therefore, computation phase does not have to move data between partitions when creating a new dataset from an existing one. Multiple narrow transformations can be performed on a dataset in memory by pipelining them in order, making narrow transformations very efficient.

Wide transformations, on the other hand, have resulting RDD partitions that depend on more than one parent partition, therefore data have to be moved between partitions (and maybe nodes) when creating the new dataset as the result of the transformation. Moving the data between partitions is also called shuffling. Shuffling almost always results the data to be sent across the network between clusters, causing network usage. Shuffles are costly operations and should be avoided as much as possible.

In Spark, a pipelined set of narrow transformations is called a stage. The end of a stage is defined by call for a wide transformation operation. Execution of a Spark job is effectively executing the composing stages of the job, sequentially. During the execution of each stage, a pipelined set of transformations are executed on partitions of an RDD that reside on one Executor node, in parallel. Each parallel execution of a pipelined transformation set is called

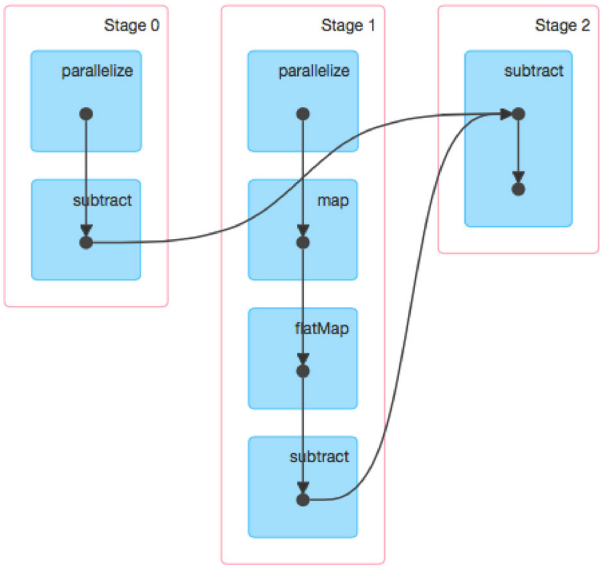


Figure 3.7: An example execution DAG with stages, as shown in Spark WebUI.

a task. In essence, parallel execution of tasks together make up the execution of a stage, and sequential execution of stages make up a Spark job. An example consisting of two stages have been visualized in Figure 3.6.

Execution of a Spark Job

Execution of a Spark job starts by a submit script communicating with the Master node that a certain application code is asked to be run by the cluster. Master node populates the Driver and Worker nodes, and passes the code on to the Driver. The Driver converts the transformations and actions instructed by the code into a DAG. The DAG is further broken down into stages, where narrow transformations are pipelined and shuffle operations are laid out into a stage execution plan. The stages are further divided into tasks and bundled together to be sent to the executors that are instantiated in the Worker nodes.

The driver program then talks to the Master Node and negotiates for resources. The Master node launches executor JVMs on the Worker nodes on behalf of the Driver Node. At this point, Driver node sends tasks to the cluster manager based on data locality. Before executor instances begin their execution, they register themselves with the application so that the driver has a holistic view of all executors. Now, executors start processing various tasks assigned by Driver node. At any point of time when the Spark application is running, the Driver node will monitor the set of executors that run. Driver node also schedules future tasks based on data placement by tracking the location of cached data. When `main()` method of the code exits or when it calls the `stop()` method of the `SparkContext`, the Driver will terminate all the executors and release the resources to the Master node.

The DAG, stages of a job and realtime monitoring of execution can be accessed through WebUI of Spark. An example of how an execution plan of stages of a job looks in Spark WebUI can be seen in Figure 3.7.

4

Evaluation of Alternative Solutions

ASML HPC offers a number of alternative technologies that could be used to facilitate a parallel implementation as a solution to the sequentially intractable problem at hand. The alternatives that have been investigated, and laid out in chapter 3 need to be compared with respect to a set of criteria, before making a final decision on how to continue with the implementation. For this purpose, insights from the survey study have been utilized to design different execution concepts for a parallel implementation. The target for the designs have been determined as to rely on strategies that would bring out the advantages of respective alternatives. Then, the concepts have been compared with each other according to the following criteria:

- Expected (scalable) performance, in relation to achievable parallelism. The solution proposed for finding the optimal mapping model involves aggregation of scores from each cross-validation partition constructed for a given model. The amount of data dependency is variable, depending on the size of the cross-validation dimension chosen by the user. A concept can have varying degree of emphasis facilitating the communication necessary to distribute this dependency across multiple computing resources, resulting in varying degrees of theoretically achievable parallelism. Performance barriers in terms of communication and resource scheduling overhead are also considered in estimation of performance, as well as how this performance is expected to scale. For each technology, a different trade off has to be made in this regard.

- Feasibility of implementation, i.e. availability of necessary data science and machine learning functionality (regularized regression, cross-validation,etc.) built-in or through mature libraries. An implementation method and associated technology is preferred to support functionality via reliable, high-level API choices. A concept is not preferred if realization of it involves implementing low-level functionality such as least square drivers, linear solvers and so forth.

- Ease of an extensible, maintainable implementation. Modular concepts work better than monolithic approaches. Code complexity is sought to be reduced.

- Expected cost of operation. The required hardware for a unit of speedup is a target to be minimized.

Comparison of alternatives and presentation of the results of comparison are the topics of this chapter.

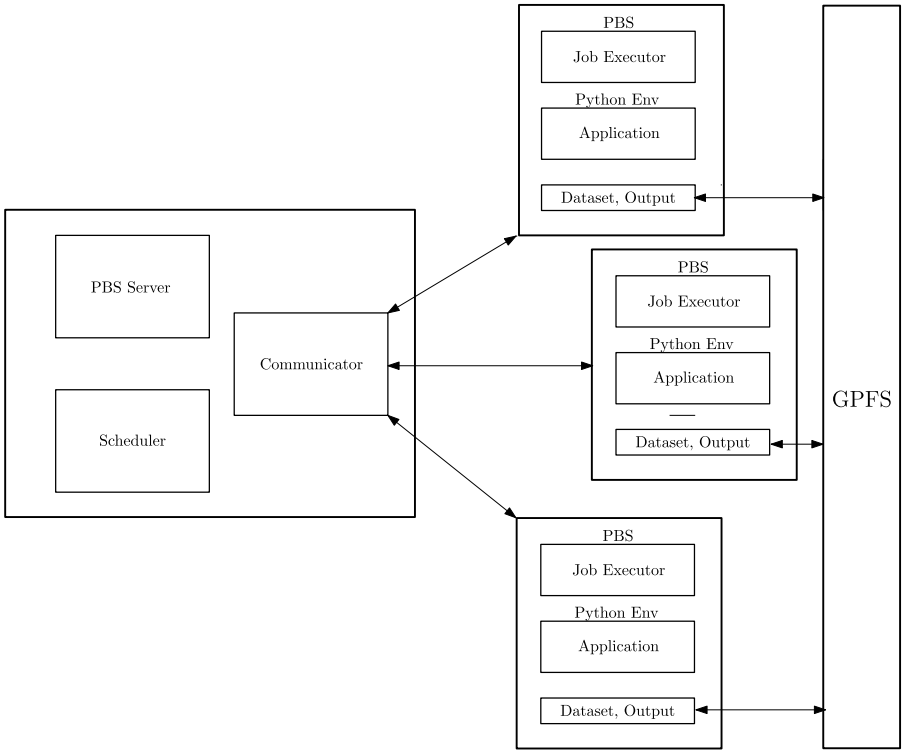


Figure 4.1: Execution concept of PBS based implementation.

4.1. PBS Based Concept

A PBS job scheduler based approach is, for the most part, independent of the development and runtime environment. Any application that is callable via the remote shell (a functionality that PBS already provides) could be scheduled among a configurable choice of hardware resources. However, as there is no specific communication protocol associated with PBS, a strategy for dataset dispatch and collection of results is needed.

For sharing dataset and execution output, the network file system of the cluster, which is based on General Parallel File System (GPFS), can be used, as network bandwidth and disk performance requirements for reading in the dataset and writing out the results initially are not expected to have a detrimental load on the file system. GPFS is an enterprise file system developed by IBM that has support for both local disks on nodes of a cluster as well as storage area networks. GPFS can provide logical isolation where seperate file systems can be used, or physical isolation where where seperate storage pools are made to work together. It is scalable to thousands of nodes and petabytes of storage, where data is replicated on multiple nodes with no single point of failure. File updates can happen synchronous and asynchronous, depending on the configuration.

However, data needs to be communicated between the processes which compute a given model's score on a cross-validation partition, as the individual scores have to be averaged to obtain one cross-validation score per model. For the same reason, if a node is given the task of computing all the scores for a given model, the interprocess communication need is bypassed at the expense of lost parallelism. The concept of PBS implementation exploits the fact that the problem is still embarrassingly parallel on other dimensions of the model space, as long as the whole cross-validation scoring for each individual model is executed at only one node. This strategy results in the concept that is visualized in Figure 4.1.

The advantages of the PBS concept mostly emerge from the fact that the task distribution

strategy is conceptually decoupled from the application development. This means that application can be implemented using a wide variety of programming language and development environment options. High level statistical computation libraries can be used with minimal issues on the scheduling side. Moreover, the development essentially boils down to devising a high performance single node application, whose execution is repeated many times, orchestrated by PBS. This makes the application code more easily readable, as well as highly extensible and maintainable. Naturally, all this comes for the tradeoff of losing a whole dimension of parallelism at the expense of avoiding interprocess communication and relying on the network drive infrastructure for data sharing. Moreover, as a large number of nodes should be utilized to drive the execution time down, the cost of operation for a PBS based implementation is relatively high.

Extension of PBS Concept with MPI

An MPI communication based approach to increase achievable parallelism in PBS based concept asserts certain limitations on the development environment, as the application should be built on a language that provides a comprehensive library for MPI based communication and other functionality. For Python, a common choice among data scientists and machine learning programmers, MPI4Py is an unofficial, but popular package that provides bindings for the MPI standard.

MPI execution model is optimized for interprocess communication. This fact shapes the associated strategy for data sharing and interprocess communication of the associated extension. For storing dataset and execution output, the network file system can still be used. For the interprocess communication, message passing environment of MPI is the obvious choice. As far as interprocess communication goes, only nodes that are responsible for cross-validation of a given model need to communicate with each other, and only for the output scores. A sensible strategy is to individually compute scores of a given model for a cross-validation partition, and then use MPI gather to collect all scores in a “central” node to compute the average across all partitions. The “central” nodes for each model can then pass their results to a single analysis node that is responsible for evaluating the model with the best average score. The concept is shown in Figure 4.2.

Although interprocess communication is handled by MPI, the concept still utilizes PBS for resource allocation. However, because the model space does not have to be flattened across the cross-validation axis anymore, individual nodes will need less computational resources. More nodes can be assigned to a each subjob, where subjobs are this time not characterized by the task that they perform, but the central nodes which are responsible to compute the average of cross-validation partition results.

The advantage of the MPI extension is that it exploits the parallelism of the model space to the full extent, using fast and time tested interprocess communication routines. MPI-PBS hybrid approach to orchestration also means that the resolution of resource allocation can be decreased, which means that less scheduling overhead can be expected from PBS based task allocation part of the concept, in comparison to a purely PBS based approach. Lessened emphasis on resource allocation during execution is also expected to reduce overhead coming from multiple instances of startup and teardown of execution contexts.

The exploitation described above comes at the expense of introducing communication overhead to the execution. The overhead is determined by the network throughput between the nodes, which becomes a variable in the performance model. Furthermore, MPI communication schemes should be explicitly defined within the application code, which asserts a certain rigidity to the execution strategy. The application code becomes more convoluted and less readable, due to the fact that a hybrid approach is taking place: slicing of the model space and parallel execution is handled within the app by MPI routines, whereas the actual computation is done in a high level language environment, with use of libraries dedicated to

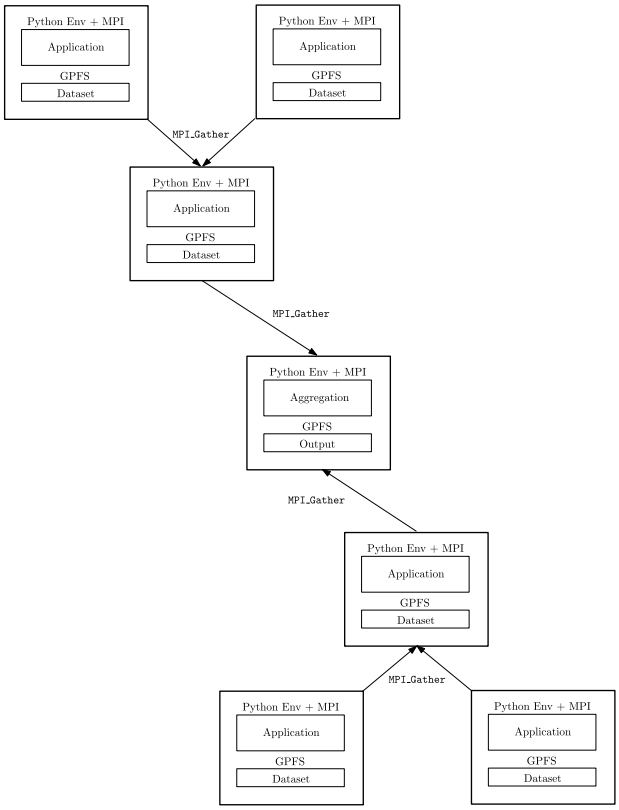


Figure 4.2: Execution concept of MPI-extended PBS implementation.

machine learning and statistical methods. This hybrid approach also makes the code harder to maintain and extend. As for cost of operation, MPI is expected to be on a similar level to an alternative based on PBS.

4.2. CUDA Based Concept

CUDA approach is different in the way that, rather than using the CPU power of every node in the cluster, it utilizes a subset of nodes where GPUs are available, driving down the cost of operation from hundreds of nodes to only one node with a few GPUs. This, along with the compatibility requirements of the development environment are the defining constraints for the CUDA application concept. A CUDA application can be based on already existing library such as LaPACK, at low level linear algebra routines that provide the basis for regression. However, higher level functionality such as model scoring algorithms, different approoaches to creation of cross-validation partitions and flavors of constrained regression would either have to be manually implemented. Alternatively, an intermediary software layer between a high level development environment which provides necessary functionality and CUDA has to be used. This intermediary layer can take code that is written via calls to high level libraries integrated to the development environment, convert the data structures into CUDA friendly forms and utilize GPU-accelerated solvers. The concept is depicted in Figure 4.3.

Scikit-learn, a Python based statistical computing library already provides needed high level functionality, thus the effort of designing a CUDA based application initially is focused on a search for an intermediary layer between Scikit-learn and CUDA. H2O4GPU project is found to provide the necessary GPU solvers, and its Python API is built around Scikit-learn API. H2O4GPU is meant to be used as a drop-in replacement for Scikit-Learn and at the time of this study, regularized regression and cross-validation was in scope of its development team.

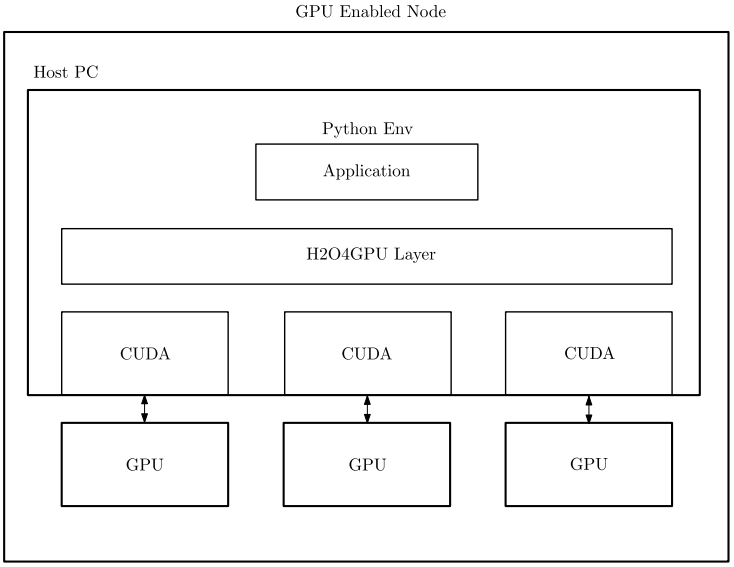


Figure 4.3: Execution concept of CUDA and H2O4GPU based implementation.

H2O4GPU indeed supports two of the most important high level functionality, namely performing a grid search on a cross-validation task for regularized regression. For this purpose, H2O4GPU is integrated into a Python runtime environment in a single node with GPU. The effort involved compiling the code against a shared base of libraries, integrating it into a development environment and doing initial testing which consists of reading in a sample dataset in array structure, and using H2O4GPU as the mediary layer for calling a simple modeling function in Scikit-learn.

Further on, the practice of implementing the concept has proven to be a challenge because of two main reasons: gaps in implementation and various small scale bugs in the already implemented parts of H2O4GPU. For example, because H2O4GPU (at the time of the study) has not implemented regularized regression classes defined by Scikit-learn completely adhering to the interface specifications, a wrapper class had to be written to pass on the argument set in complete and correct order. At a later stage, a software bug has been caught, where hyperparameters of the regression were not correctly passing on to the GPU solvers if the solver function is called with a predefined set Λ. A small code change solved the problem and H2O4GPU team has been notified about the bug. However, the biggest functionality gap is observed as the intermediate layer's lack of support for regression problems with multiple target variables. As the problem at hand is inherently multiple target, a comprehensive workaround to this functional gap was essential. Due to the time constraints, further efforts are descoped.

4.3. Spark Based Concept

Apache Hadoop Ecosystem is chosen as the production environment for ASML HPC and Spark is the high performance compute engine for this ecosystem. Most straightforward industrialization for a proof of concept delivered as a result of this thesis will be possible if that deliverable is written for a Spark cluster. Therefore, it is essential to investigate Spark as a potential alternative for implementation phase.

There are two ways to go about developing a Spark based concept: one using the native machine learning libraries based on the newest, high level Dataset API (Spark ML), and another one using PySpark to enable usage of well known machine learning libraries in Python. De-

velopment of the Spark based concept started by investigating both approaches applied on a restricted subset of the actual problem. The observations regarding both ways are listed below:

- Spark ML is based on a machine learning pipeline approach, where computations on the input data is broke up into stages. Not to be confused with the stages concept related to Spark execution model, nevertheless, the pipeline concept fits very well to the execution model of Spark. A pipeline can be linear or non-linear (similar to a DAG execution flow), and consists of two types of components: transformers and estimators. For a restricted concept, `VectorAssembler()` suits the task of preprocessing the data into features and targets compatible with SparkML norms, and `VectorSlicer()` is found to be the transformer component needed to exhaustively select different combinations of features from a given set. As an estimator component, `LinearRegression()` comes with built in support for ridge and elastic-net regularization techniques. It can be said at a high level, Spark ML provides the functionality that is needed for the task in a seamless manner.
- PySpark allows the usage of popular machine learning libraries (Scikit-learn, mlxtend and similar). These libraries support a machine learning pipeline abstraction. In fact, Spark developers describe Scikit-learn as the inspiration for their machine learning pipeline concept. Scikit-learn does not ship with the data transformers that are needed for the job, mlxtend provides exhaustive feature selection functionality with an interface that is compatible with Scikit-learn. Although this functionality is not exactly what is needed, it provides a basis for a customized implementation of exhaustive search in the feature space needed for the task.
- Spark ML is based on the assumption that data used for the modeling is large enough to be considered big data. Afterall, Spark is a big data processing engine. Therefore, the modeling implementations rely on distributed algorithms such as Stochastic Gradient descent. The assumption is that, the dataset that is composed of features and targets of a modeling effort is too big to be processed on one computer, and can benefit from being distributed among the nodes of a cluster. However, the problem in hand suffers from curse of dimensionality in the candidate model space, where dataset is small enough to easily fit to the memory of a single node. First efforts of implementation performed on a restricted subset of the model space has shown that Scikit-learn for computations with PySpark used as the distribution engine may fit to a solution better than using native Spark ML libraries.

In Spark, a dataset that is small enough to cause minimal network overhead and can fit into a reasonable memory space can be distributed among the Worker Nodes using broadcast functionality. The model space can be parameterized in the driver program and then parallelized into an RDD so that each node in the cluster has partitions that contain the information for constructing a subset of the model space. Each executor would then run the Scikit-learn model fitting and cross-validation algorithms using the parameterization contained in the RDD partitions and return the results to the Driver.

Similar to the PBS concept, it makes sense that Spark concept also exploits the fact that problem is still embarrassingly parallel on a model space which is flattened across the cross-validation axis. That is, cross-validation scoring for each individual model is executed at only one node. This cross-validation task can be further accelerated via multicore support of Scikit-learn, as long as the model class being used is compliant. The high level idea is visualized in Figure 4.4.

Spark concept, although very similar to PBS concept in philosophy, is not a perfect decopuling of task distribution and application. Developer of such a concept is constrained to use languages that are supported by Spark, i.e. Scala, Java or Python. However, Spark provides an excellent platform to handle execution in a cluster; in many ways better than PBS. Some of the advantages of using Spark for task distribution are as follows:

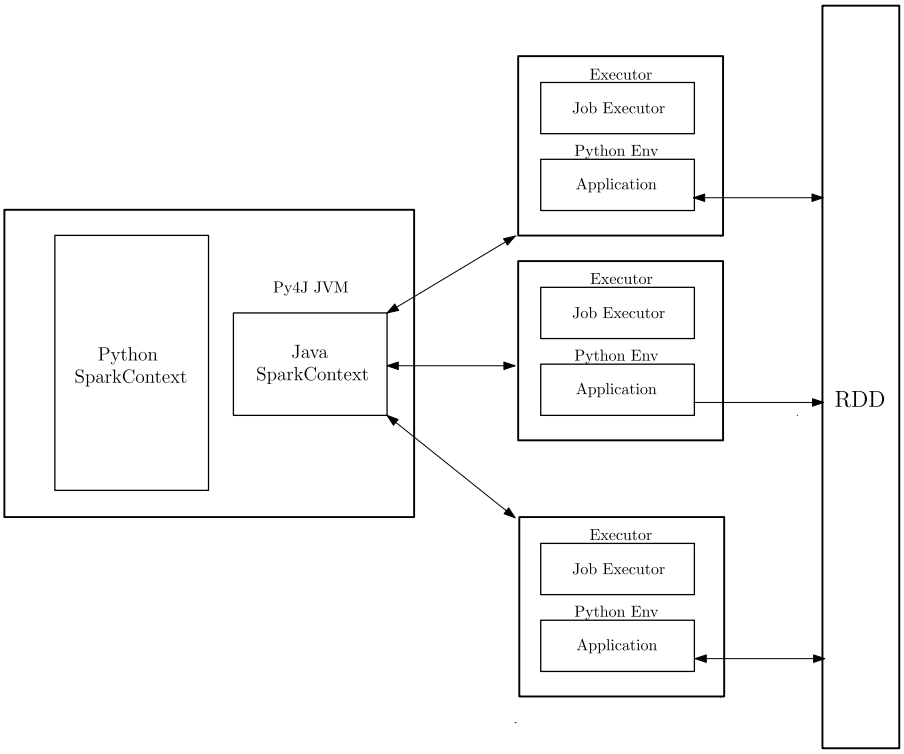


Figure 4.4: Execution concept of Spark based implementation.

- During the trial runs for concept development, scheduling overhead of Spark is consistently observed to be less than PBS.
- Task scheduling behavior of Spark is not as opaque as PBS to the user. There are a number of configuration parameters that can be fine-tuned for the given problem and cluster.
- Spark provides functionality to broadcast and distribute to the nodes across the network in an easy and consistent manner, whereas it gets complicated very quickly to achieve such functionality with the distributed shell of PBS.
- Spark has sophisticated measures for fault tolerance and handling the output of faulty executions. PBS can only provide retrial of a failed job in a rudimentary level.
- Spark provides an extensive WebUI to monitor the cluster during execution. Furthermore, monitoring functionality can be expanded by plugins. PBS has no monitoring functionality for execution of an application during runtime in individual nodes and only can provide monitoring of the whole of the submitted job.

4.4. Comparison Overview

Table 4.1 shows an overview for comparison of all the alternatives that are discussed in this section. As a CUDA based approach does not have the required feasibility of implementation, the choice is focused between PBS and Spark based approaches.

Although achievable parallelism suffers, it is important to note that the granularity of parallelism of the problem at hand is much higher than the number of nodes available in ASML HPC, and interprocess communication that MPI provides does not justify the communication overhead, as well as difficulties in extensibility and maintainability. Thus, a pure PBS based

approach is chosen as a ”lab level” implementation, to have a working application framework that is suitable for exploratory work and facilitates a ”bring your own tool” approach for the developer who would like to extend it. For a full fledged ”fab level” implementation, Spark becomes the most worthy alternative to proceed with, given that it satisfies performance related criteria, and although it constrains the toolset for development, the application would be compliant with the production environment of ASML HPC that is based on the Hadoop Ecosystem.

Table 4.1: Comparison of alternative solutions.

	PBS	PBS + MPI	CUDA + H2O4GPU	Spark
Achievable Parallelism	+	+++	++	+
Communication Overhead	-	- -	-	-
Scheduling Overhead	- -	- -	N/A	-
Feasibility of Implementation	+	+	- - -	+
Extensibility	+++	+	+	++
Maintainability	++	+	+	+++
Cost of Operation	$$	$$	$	$$

5

Implementation

‘In this chapter, technical details of two different implementations are laid out. These two implementations, namely Lab implementation with PBS and Fab implementation with Spark, share considerable common ground at a conceptual level. In essence, both implementations are about distributing the task of performing an exhaustive search on a model space composed of millions of candidates among the nodes of a cluster. Furthermore, both implementations utilize the same machine learning libraries with multiprocess support at single node level, to perform the modeling effort. The main differences between the implementations are:

- the parallel framework that is used,
- how the dataset is broadcasted to compute nodes,
- how individual modeling parameters are communicated to each node,
- how results are collected,
- the scheduling engine used for execution.

The following sections go deeper in investigation, development and profiling phases of the effort for both implementations.

5.1. Lab Implementation with PBS

At the time of when Lab implementation efforts were started, there was already a sequential implementation in a development environment that does not trivially scale to a parallel implementation. Thus, the first efforts are concentrated on refactoring the existing code into a single node application with a future parallel implementation in mind. Performance and resource usage of the single node application has been optimized. The single node application is then integrated into a PBS based parallel execution model and scaled to run on ASML HPC.

Lab implementation analysis follows modular structure of the software, breaking down the explanation into functional modules. Although the functional description of the modules are sometimes straightforward, a number of challenges have been encountered during their implementation and integration. These challenges are presented in respective points throughout the text, along with proposed solutions.

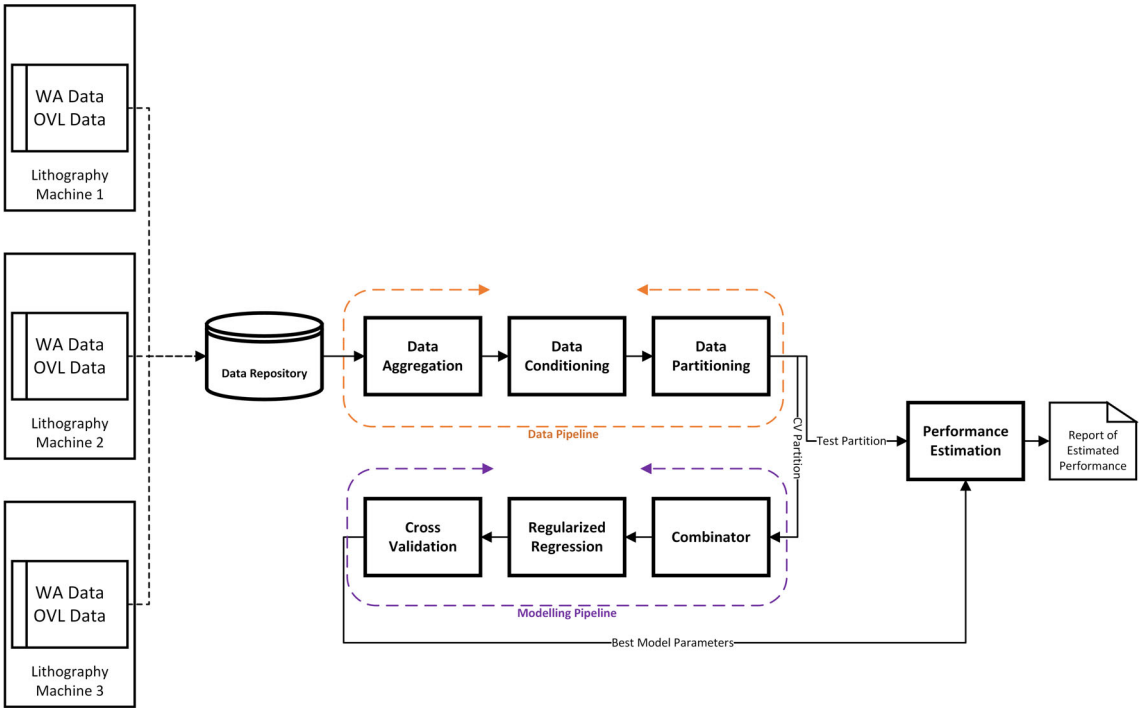


Figure 5.1: Data and execution flow of industrialized sequential implementation.

The section begins by explaining how a parallel programming model is constructed by examining the flow of data in the implementation that already exists, identifying its bottlenecks, parallelizable functionality and synchronization points. This investigation is then further used to develop the parallel programming model. An overview of the steps of parallelization have been given, with a specific emphasis on the task decomposition and assignment strategy that is built to minimize the need for synchronization and thereby simplifying orchestration. This strategy culminates into a parallel execution model.

While some parts of the software are functional during parallel execution, some others fulfill auxiliary roles such as facilitating server communication, taking in user input and visualizing the output. These modules are explained out of the context of the parallelization effort.

After laying out the architectural aspects, toolchain choices are motivated. A single node application built with the toolchain is profiled and optimized. Finally, the scaling behavior of single node application into multiple nodes is investigated, leading to a performance model that guides the full-model-space experiments.

5.1.1. Parallel Programming Model

Parallel programming model takes the data and execution flow of the sequential implementation in MATLAB. This implementation is divided into two main parts, namely data ingestion pipeline and modeling pipeline. The pipelines are further divided into modules that provide specific functionality and transform the data along its flow path, as shown in Figure 5.1.

The primary source of data for overlay modeling are the results of wafer align and overlay measurements performed at multiple lithography machines. Measurement data is internally used by metrology software to fit coefficients of a HOWA3 polynomial, i.e. each measurement set is represented as a row in a $n_{wafer} \times 20$ matrix as well as metadata that carries the information on exposed layers, exposure parameters and wafer history.

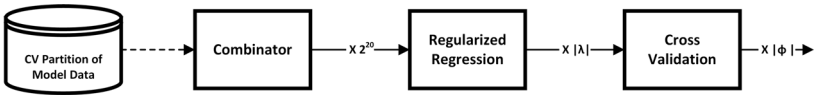


Figure 5.2: Visualization of the increase in size of candidate model space as input data moves sequentially through modeling pipeline.

Measurement data is extracted from the machine into a data repository manually. First transformation that is applied to data is aggregation, that involves filtering the data portion of interest depending on the type of overlay specification to be modelled, then combining wafer align and overlay measurement data of interest into one dataset, outputted as a .mat (MATLAB data storage format) file.

Moving through the data ingestion pipeline, the aggregated data is further reprocessed by normalization, dimension reduction (in geometric coordinate system) and outlier detection, if chosen by the user. When data is deemed ready for statistical analysis, it is divided (by random selection with or without replacement) into test and cross-validation partitions. At this point, the data is ready to enter the modeling pipeline.

modeling pipeline has the biggest potential in performance gain by parallel programming, as the output of data pipeline (input of the modeling pipeline) is relatively small. The main motivation and steps of the modeling effort have been discussed in section 2.2. However it is important to investigate how the execution becomes more compute intensive as data flows through the modules of the modeling pipeline, to understand the benefits of a parallel programming approach to the problem at hand.

The cross-validation partition enters the modeling pipeline as a simple matrix representation with size $n_{wafer} \times 20$. The so-called combinator, calculates all possible combinations of features, scaling the number of models to be obtained with 2^{20}. The regularized regression module scales that number further by the number of regularization hyperparameters specified by the user. At this point, there is no data dependency between model computations of the regularized regression module, therefore the regression effort for all outputs of the combinator scaled with the hyperparameter, leading to $2^{20} \times |\Lambda|$ candidates, is embarrassingly parallel.

The cross-validation module is the point where data dependency is introduced to the problem, as multiple validation sets modelled using the same complexity and hyperparameters should be scored and the scores should be aggregated into a statistically relevant value. One can consider the scoring sub-function of the cross validation module as the only synchronization point along the modeling pipeline. The number of processes that need to synchronize are equal to $|\Phi|$, for each complexity and hyperparameter combination.

Another synchronization point is introduced in the performance estimation module where aggregated scores of cross-validated models should be compared with each other, however the majority of the computational intensity is already left behind at this point.

The parallel execution model, illustrated in Figure 5.3, focuses on the part of the sequential implementation that can benefit the most, namely the modeling pipeline. It assumes that an aggregated dataset is already loaded in a shared persisent location of the cluster, where one node performs the necessary data conditioning and partitioning on it, writing the partitioned dataset back into the network storage, to have it accessible by model compute nodes. In a way, the network storage is set up to be used for the broadcasting step of the parallelization.

Once the input data for the modeling pipeline is guaranteed to exist in the parallel storage, a PBS script is constructed by the PBS Script Call module, which instructs the PBS server about how to perform the decomposition, assignment and orchestration for the parallel pro-

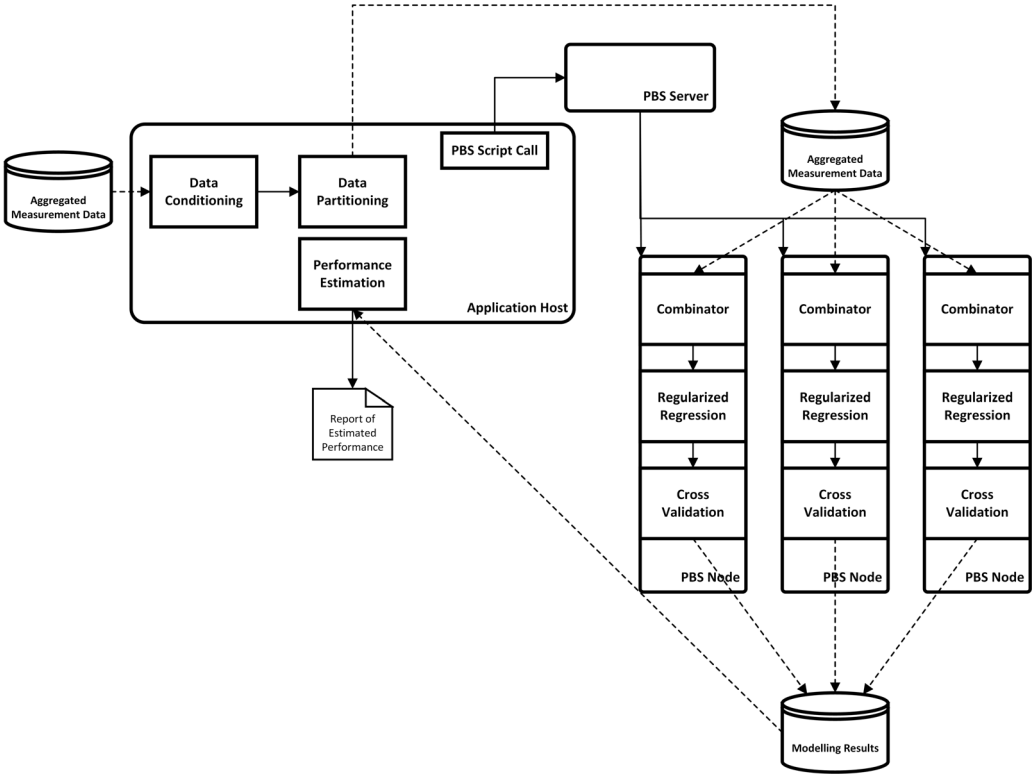


Figure 5.3: Visualization of high level programming model for the Lab implementation.

gram. PBS server executes the job received from the task, allocating the requested resources in terms of nodes and binding them to tasks. Each task involves a similar pipeline to the sequential implementation: computing the feature combinations, performing regularized regression and cross-validating the obtained mapping models. The crucial difference is that rather than executing on the whole model space, each nodes executes on a subset of it. The task in each each node composed of multiple processes. These processes are spawned and executed outside of the PBS scope. In this sense, the model follows a loose integration strategy with PBS.

5.1.2. Execution Model Avoiding Interprocess Communication

Execution model for Lab implementation is designed with targets of scalability, ease of implementation and extensibility. The model relies on dividing the model space (hence the problem) into equal-sized chunks to have a homogeneous array of jobs, then mapping them into a fixed size of resources and schedule the execution using PBS.

The execution model starts with partitioning the model space in all directions. A model's complexity (the number of model coefficients that are not zero constrained) can be represented as a vector that encodes the model coefficients that are to be constrained with a 0 and the rest with a 1. 2^m possible combinations of a model which has m coefficients can be represented in an m-dimensional vector space. For example:

$$\gamma_n = [111 \dots 0] \in \mathbb{R}^{1xm}, 1 \leq n \leq 2^m \in \mathbb{N} \tag{5.1}$$

would encode that only the last coefficient among m coefficients of a model will be zero-constrained.

Given the description of complexity in equation 5.1, each individual model in the model space can be defined in terms of the model's complexity γ_n, the regularization hyperparameter λ_l and the cross-validation set used ϕ_k

$$\mathcal{M}(\gamma_n, \lambda_l, \phi_k), \text{ where } \lambda_l \in \Lambda, \phi_k \in \Phi \tag{5.2}$$

and computation of each model $\mathcal{M}(\gamma_n, \lambda_l, \phi_k)$ is independent.

However, the problem involves not only computation of the models, but averaging the cross-validation score for all $\mathcal{M}(\gamma_n, \lambda_l, .)$ that have been fitted with the same regularization and complexity constraints. It is important to keep in mind that there is data dependency along the cross-validation axis, as the models have to be compared with each other in cross-validation metric score to find the optimum among a given bunch.

Hence, it makes sense to "flatten" the model space along its cross-validation direction to avoid excessive communication between nodes, at the expense of increased computational complexity ($\mathcal{O}(|\Phi|)$) on a given node. Having each node compute a complete subset of model scores along the cross-validation axis also enables a more straightforward implementation, as Scikit-learn already offers a (thread parallelized) function for computing all the models and cross-validation scores on a hyperparameter grid.

Based on this model space partitioning strategy, an "atomic" task, $\tau(\gamma_n, \lambda_l)$ of the execution model can be defined as performing cross-validation routine to a subset of the model space constrained by γ_n in model complexity and regularization hyperparameter λ_l. A task that is going to be mapped to each node, $\mathrm{T}(p)$, is a set of p atomic subtasks. Thus, each node's task involves the following steps:

1. Load the dataset from the input data path.
2. Construct the train and validation subpartitions from cross-validation partition.
3. Identify the model subspace to be worked on.
4. Construct the hyperparameter and complexity tuples to be used to calculate the models.
5. For each subpartition set, fit a model on train subpartition for each hyperparameter and complexity tuple. Compute the score of the model on decided loss metric using the validation subpartition.
6. Compute the average of the scores across all of the validation subpartitions.
7. Report the model with the highest average score, and write it back into a file.

PBS provides the functionality of defining and queuing a set of jobs as a job array. This way, almost-identical jobs such as the tasks $\mathrm{T}(p)$ defined for the execution model of this implementation can be queued in batch. PBS can process a large job array more efficiently than it can process the same number of individual jobs [11].

In PBS terminology, each element in a job array is called a subjob. Each subjob in a job array has a unique index identifier. A subjob runs on an array of identical computational resources. Each element of the array is called a node and each node can be specified to PBS by its physical hardware resources such as CPU and memory size. Each node in a subjob can be individually communicated by their node number via PBS distributed shell, can admit shell commands to run a Python program and pass command line options.

Thus, a unit of computational resource that a PBS job array provides can be referenced by its subjob array index `PBS_ARRAYINDEX` and node number `PBS_VNODENUM` within that

subjob, namely R(PBS_ARRAYINDEX, PBS_VNODENUM). Each time a computation is started, a Bash script is generated to orchestrate PBS in such a way that each task T(p) is assigned to a computational resource R(PBS_ARRAYINDEX, PBS_VNODENUM), exactly once. The results collected in a network hard drive are aggregated,processed by a single, high-performance node and statistically relevant results are displayed utilizing plotting functionality provided by Jupyter interactive environment.

5.1.3. Auxilliary Functions

User Interface

Observing the way that industrialized sequential implementation is designed, one of the goals of both parallel implementations is to take the tool out of the exclusive use of a few domain experts by making it easier to use. There are a number of parameters regarding the execution of the parallel program, most of which are not supposed to be exposed to the end-user, but to be optimized in the background. That said, a data scientist as an end user would prefer to be able to try different modeling approaches, tweak certain modeling parameters and employ different statistical metrics to find the best mapping model. This is even more true, when speedup and scalability achieved by the parallel program allows the end-user to have an iterative approach to achieve the best modeling practices.

Upon discussions with a number of domain experts, the set of execution parameters that are to be exposed by a user interface are specified as follows:

- Modeling options:
  - Path to aggregated dataset.
  - Resampling method for dataset partitioning.
  - The ratio of dataset partitioning.
  - Regularization hyperparameter range and resolution.
  - Number of cross-validation sets.
  - Model selection metric and criterion for cross-validation.
- Parallel execution options:
  - Number of concurrent jobs.
  - Number of nodes assigned per job.
  - Hardware resources per node, in terms of number of processor cores and amount of memory.

The execution frontend can be seen in Figure 5.4.

PBS Script Call

The resulting script queries a set of nodes, with specified amount of resources, for a specified amount of time. Once these nodes are created, they are instructed to activate the runtime environment for the program and to redirect the output of the execution into a debug file.

But more importantly, PBS script defines the decomposition of the sequential program and explicitly instructs the PBS server about asignment of the tasks to the nodes. It also uses the distributed shell functionality of PBS to instruct each node about the specific task that they need to perform. To further facilitate the understanding of how this module works, an example output script is provided below:

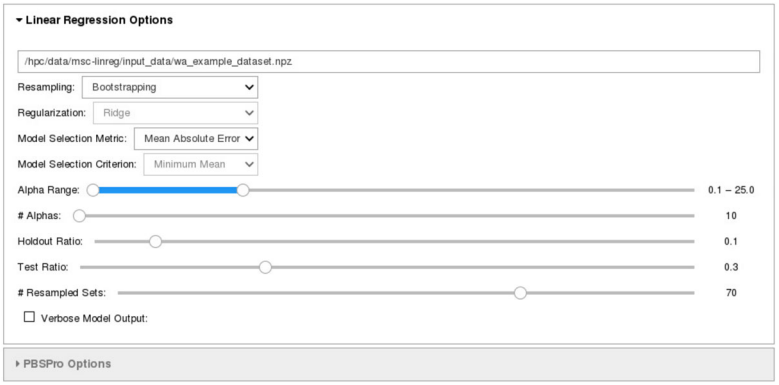


Figure 5.4: A Jupyter notebook used as a web interface designed for user to handle execution parameters.

```
#!/bin/bash
#PBS -N my_job
#PBS -l walltime=00:59:00
#PBS -l select=4:ncpus=2:mem=8gb
#PBS -J 0-1023
#PBS -o /hpc/data/msc-linreg/logs/my_job.out
#PBS -e /hpc/data/msc-linreg/logs/my_job.err
module load pbspro
cd /hpc/data/msc-linreg/

pbsdsh -n 0 -- /hpc/data/msc-linreg/pythonscriptrun.sh
/hpc/data/msc-linreg/nodescript_bootstrapping.py
--n_nodes 4 --pbs_arrayindex ${PBS_ARRAY_INDEX} --pbs_vnodenum 0
pbsdsh -n 1 -- /hpc/data/msc-linreg/pythonscriptrun.sh
/hpc/data/msc-linreg/nodescript_bootstrapping.py
--n_nodes 4 --pbs_arrayindex ${PBS_ARRAY_INDEX} --pbs_vnodenum 1
pbsdsh -n 2 -- /hpc/data/msc-linreg/pythonscriptrun.sh
/hpc/data/msc-linreg/nodescript_bootstrapping.py
--n_nodes 4 --pbs_arrayindex ${PBS_ARRAY_INDEX} --pbs_vnodenum 2
pbsdsh -n 3 -- /hpc/data/msc-linreg/pythonscriptrun.sh
/hpc/data/msc-linreg/nodescript_bootstrapping.py
--n_nodes 4 --pbs_arrayindex ${PBS_ARRAY_INDEX} --pbs_vnodenum 3
```

Toolchain and Development Environment

The attempt at implementing a solution to the problem at hand explicitly uses the fact that the effort of finding the best mapping model $\hat{\mathcal{M}}$ is composed of many similar, but individual (and rather small) computational tasks, which can be also seen as a batch of almost identical PBS jobs. PBS based implementation relies on running the same application on many individual nodes of a cluster and conforms to SPMD model. For a SPMD execution model, the effort of implementation starts with developing an application that can be run by a single node. The development language for this application has been chosen as Python, as there are a number of established high level scientific computing libraries available for it. Particularly, Scikit-learn and NumPy provide a popular library suite in Python for tasks that involve linear algebra and data science.

Moreover, these libraries can be integrated in a Jupyter development environment for providing the users of the application a HTML based, easy to use web interface as well as various visualizations on the output.

NumPy provides the necessary array-like data structures and basic operations that are used in the single node application of PBS implementation, among other things, to:

- represent the dataset and the model subspace of interest in a multidimensional array form,
- represent the computed models and process the results,
- store the results efficiently, in a file format that can be easily read across multiple architectures.
- aggregate and post-process results using memory efficient data structures.

Scikit-learn is a data science library that provides high level functions for a number of supervised and unsupervised learning algorithms via a consistent programming interface. It is built on SciPy for underlying scientific computation routines and its data representation is largely based on NumPy array structures. Scikit-learn has a number the building blocks to perform the task at hand, including:

- pre-processing the dataset,
- partitioning the dataset for cross-validation using random sampling of the dataset with or without replacement,
- performing linear regression modeling on data,
- utilizing various regularization techniques,
- performing grid search on a multidimensional hyperparameter space for regularization,
- cross-validating a set of regression models using a variety of scoring metrics that are commonly used in statistics.

The combination is also powerful in the sense that although the underlying linear algebra libraries of NumPy are not optimized for multi core performance by default [23], Scikit-learn offers process parallelism at regression analysis and cross-validation grid search level [24]. Process parallelism, together with linear algebra libraries tuned for Intel CPUs is potentially useful in ensuring maximum utilization of multi-core architecture of cluster nodes. This gives a chance to utilize a multi-core selection of nodes more efficiently during execution of a single job, while still achieving the majority of parallelism via task scheduling numerous jobs, while minimizing node-to-node communication.

5.1.4. Application Profiling

In accordance with the spirit of the Lab implementation environment, an experimental and iterative approach is followed during the development of the application, where development and profiling went hand in hand. Certain aspects of implementation are motivated by continuously running experiments to identify resource usage and performance. This section offers a summary to this process, along with the experimental findings and the decisions that they led to.

Single Node Profiling

Initially, a single node application has been implemented. This application can process a subset of the whole model space, characterized by the number of (γ_n, λ_l) tuples that will be

cross-validated by the single node, for a chosen collection of cross validation sets Φ. At the end of computation, the results are written into a file in HPC shared storage. Each tuple (γ_n, λ_l) executed on the complete cross validation set Φ constitutes an atomic task τ, and each node task $\mathrm{T}(p)$ is composed of p such atomic tasks, as defined by the execution model. This implementation approach allows for simultaneously obtaining a building block for the computational aspects of parallel implementation, paves way to devise a communication scheme, and most importantly, evaluate single node performance to gain insight on optimization opportunities as well as possible bottlenecks.

Once implemented, single node performance has been investigated experimentally, utilizing nodes of varying memory and processing power. The main motivations behind the experiments are:

- To have an informed decision on scheduling of the hardware resources for the parallel implementation and to predict the effects of choices in this regard to the parallel implementation performance, guided by the performance model at hand.
- To estimate a sequential computation time for the whole problem. This sequential time is to provide a baseline for assessing speedup obtained by parallel implementation.

The experiments are divided into two parts:

1. The amount of required memory by different stages of single node execution, with respect to the size of the processed model subspace.
2. Scaling of performance with respect to the number of processing cores assigned to an individual node ($\#_{cores,node}$), and the effectiveness of multiprocessing approaches to the single node application.

Single Node Memory Usage For analysis of memory usage of the single node application with respect to the the number of atomic tasks τ that the application completes, the popular memory-profiler package has been used. The package is listed in Python Package Index (PyPI), and it is supports IPython, making it easy to integrate to the Jupyter development environment. The line-by line analysis provided by memory-profiler has been used to track the amount of memory used by the complexity-independent part of the code, as well as the part which scales in memory usage with the number of atomic tasks, namely the object that holds the cross-validated models corresponding to each $\tau(\gamma_n, \lambda_l)$. The number of cross validation sets, $|\Phi|$ is chosen as 50 for each task as a default value inherited from baseline serial application. Each test has been conducted 3 times and averaged to ensure repeatibility of results, on a single node with a single CPU core. Figure 5.5 illustrates the tests conducted with different number of atomic tasks as data points, each point being a power of two $(16, 64, 256, 512, 1024)$. The reason for slicing the problem this way is that, the model space along the model complexity axis is always expected to be a (rather high) power of two. Dividing the problem along this axis in equal slices ensures a homogeneous distribution of computational load on the nodes, independent of the choice for Λ and $|\Phi|$. A linear approximation of trend gives insight about various aspects of the memory usage.

Memory profiling results give information on how the memory requirements scale with increased task load on one node. The majority of lines and function calls in the application consumes about 97MB of baseline memory, independent of p for $\mathrm{T}(p)$. Each model in the modelset, along with it's computing parameters, cross-validation results and average scores for $|\Phi| = 50$, consumes about 0.29MB of memory. This means that, for task load of up to 2048 model computations per node, allocating 1GB per processor for the application alone, is a safe choice that can scale to higher task loads if needed. Along with the memory requirements of the operating system, PBS module and the runtime environment, the minimum

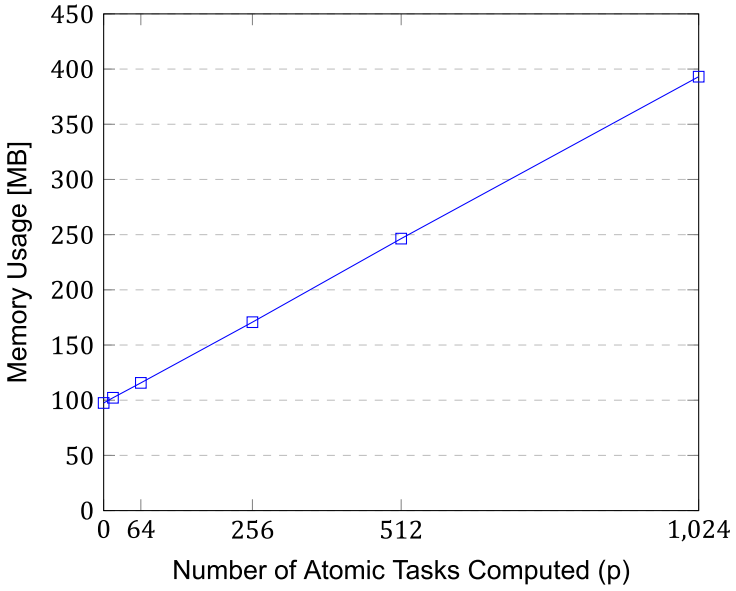


Figure 5.5: Memory usage of single node application with respect to the number of atomic tasks ($|\Phi| = 50$), computed on a single CPU.

memory per node is defined as 4GB, with 8GB being a more favorable choice if the cluster capacity allows for it.

Single Node Multiprocessing Performance Second part of the single node experiments investigate multiprocessing performance with a varying number of cores for a single node and a varying number of task load sizes.

There are a number of ways in which the single node application can execute on multiple cores in parallel, the most straightforward way being initially thought as a direct utilization of multiprocessing support built into Scikit-learn's cross-validation methods. At the point of using Scikit-learn multiprocessing support straight out of the box, it has been discovered that a bug causes nodes scheduled via PBS report the physical number of cores of the machine where the virtual node lands when queried using the sys module of Python, rather than the actual number of cores assigned to the virtual node by PBS. A workaround has been implemented by manually calculating the correct parameter for multiprocessing backend, using the physical and virtual number of cores as an input.

In theory, the library supports computation along cross-validation partitions using a various choice of multiprocessing backends. However, an investigation of the source code reveals that the Scikit-learn implementation of cross-validation is optimized for a k-fold scheme, where $k > 1$. That is to say, a new process is associated with each fold, specified as a parameter during the function call, to exploit the parallelism of individual fold computations in a k-fold validation technique. In contrast, the single node implementation follows the validation scheme that is described in section 2.2, where a cross-validation method is invoked repeatedly with each call executing on only one randomly sampled fold for each set, i.e $k = 1$.

Therefore, in order to achieve multiprocessing, a less trivial approach is needed, where built in support in Scikit-learn falls short. Given the execution model, trivial parallelism of each cross-validation instance should be exploited and each method call associated with a single fold should be encapsulated in its own process. For this, Parallel module of Python joblib library has been used, executing each cross-validation computation within an embarrassingly parallel for loop.

The joblib module allows for selection of a multiprocessing backend and a process dispatch strategy. By default, loky backend is used by the module, which is also considered the next generation solution as opposed to the legacy multiprocessing backend. By loky, all processes are started using fork and exec that are inherent to process management of POSIX systems. This ensures safer interactions with third party libraries. Multiprocessing backend, on the other hand, uses fork without exec, which often causes interference with other acceleration libraries [25]. Furthermore, overhead and data serialization related optimizations of loky usually results in better performance in generic Python applications. However, the performance of each backend should be experimentally compared.

A common and suggested method to accurately measure the running time of Python code is the Python modules cProfile. The module provides some basic profiling support in the form of summarized function-level or line-level timing information. Through this summary, cProfile module can be used for an overview of time taken during the various parts of the program, along with the number of calls to these parts. An important advantage of cProfile is that it is a C extension with less overhead [26]. An individual function can be profiled by importing the cProfile module and invoking `Profile.run(<function>)` instead of `<function>` within any Python code. CProfile can also be invoked as a script to profile a Python application by adding -m cProfile to the shell running command. The generated statistics can be formatted into simple text reports via the pstats module. In a parallel execution, the output is generated for each process and the output is intermingled, which may require some additional post-processing to provide meaningful results.

A comprehensive set of tests have been conducted with multiprocessing-enabled single nodes with varying $\#_{cores,node}$. Different backend choices along with various process pre-dispatch techniques are profiled to maximize the multiprocessing performance. Figure 5.6 and Figure 5.7 provide a comparison of execution time on nodes with a varying number of CPU cores assigned, with respect to the number of atomic tasks p with 50 cross-validation sets for each task, where different backends (namely legacy and loky) have been used. The chosen number of tasks are, once more, varying powers of two. As expected, the effect of multiple cores on the execution time is more profound as the number of atomic tasks that are computed by the single node application increases.

From the results, it is clear that the backend choice between legacy and loky multiprocessing does not influence performance for a reasonable number of atomic tasks per task, and loky is only marginally better. For having a more future-proof implementation, the modern and better supported loky is taken as a baseline for further executions, and inferences are based on its results. Based on the experiments, two inferences can be made:

1. The speedup that can be achieved using multiple cores on each node, as a function of the number of cores administered, with respect to a single node, single core implementation. The mean and standard deviance of this speedup can be measured among multiple runs to infer a repeatable estimate of performance gain by the parallelization effort.

2. An extrapolated estimation of execution time, if the whole model space were to be processed by a single node with varying number of cores.

The outcome of the inferences are laid out in Table 5.1.

The speedup and estimate for execution time are averaged over the results that have been collected by a varying number of atomic tasks computed to account for variances in experimental results. The average estimate execution time is at minimum 115 hours, when a CPU with 24 cores is used, which is the maximum number of cores available to any single node in the cluster. For comparison against multi-node results, performance for a unit of hardware resource (single core CPU) is taken as a baseline and such a resource would be able to process the whole model set in 2590 hours; an execution time that is practically unfeasible.

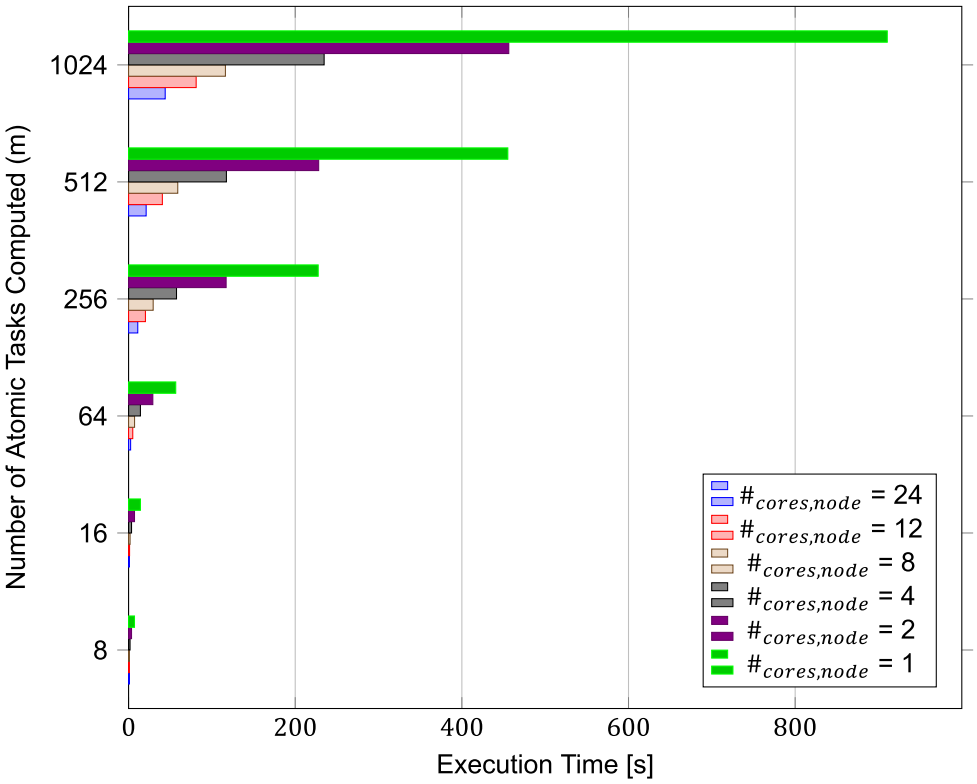


Figure 5.6: Multiprocessing performance of single node application with legacy backend with respect to the number of atomic tasks ($|\Phi| = 50$) computed at each execution by a varying number of CPU cores.

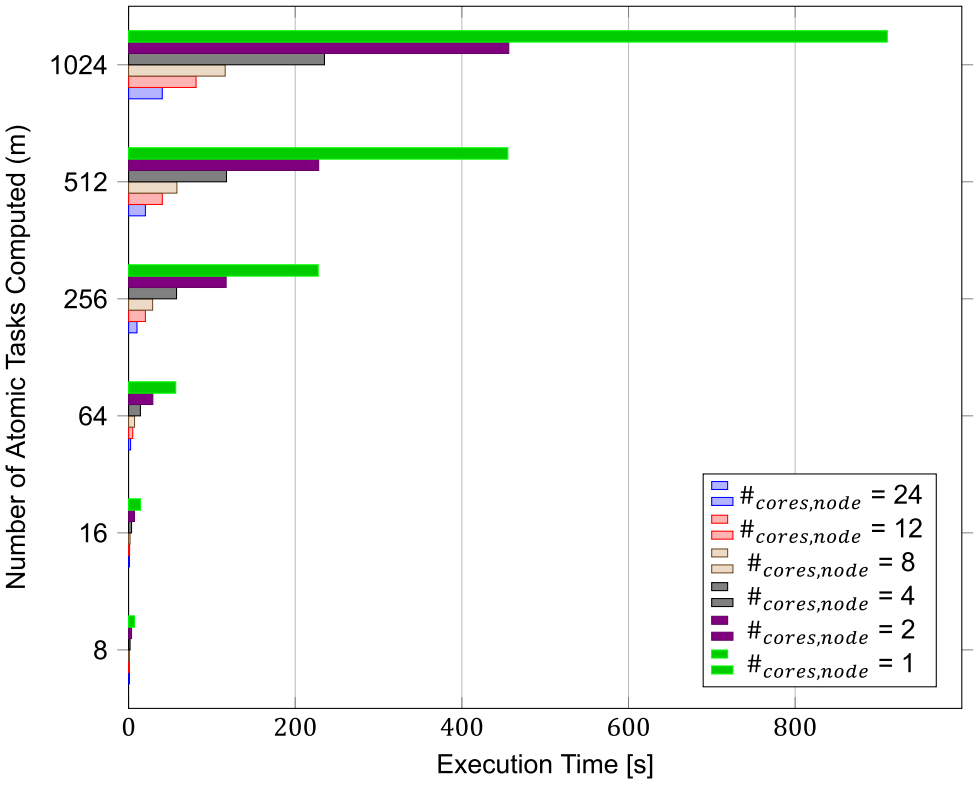


Figure 5.7: Multiprocessing performance of single node application with loky multiprocessing backend with respect to the number of atomic tasks ($|\Phi| = 50$) computed at each execution by a varying number of CPU cores.

Table 5.1: Average speedup and execution time of multiprocessing single node application with respect to the number of CPU cores.

$\#_{\mathbf{cores,node}}$	**1**	**2**	**4**	**8**	**12**	**24**
Mean Speedup	N/A	1.98	3.95	7.89	10.83	19.05
Mean Efficiency	N/A	0.99	0.99	0.99	0.91	0.80
Speedup Standard Deviation	N/A	0.01	0.02	0.05	0.30	1.67
Mean Execution Time [hours]	2590	1298	669	330	231	115

This observation alone supports the motivation of having a parallel implementation to drive the execution time down to a value that can be deemed feasible by ASML availability and throughput needs.

Also, the results show that adding CPU cores to a node lead to a linear speedup trend with a sufficiently large model space as it can be seen in Figure 5.8, which motivates further experimentation to find whether best distributed computation strategy involves “fat” nodes (with many CPU cores) with reduced achievable job concurrency or “skinny” nodes with potentially higher achievable job concurrency.

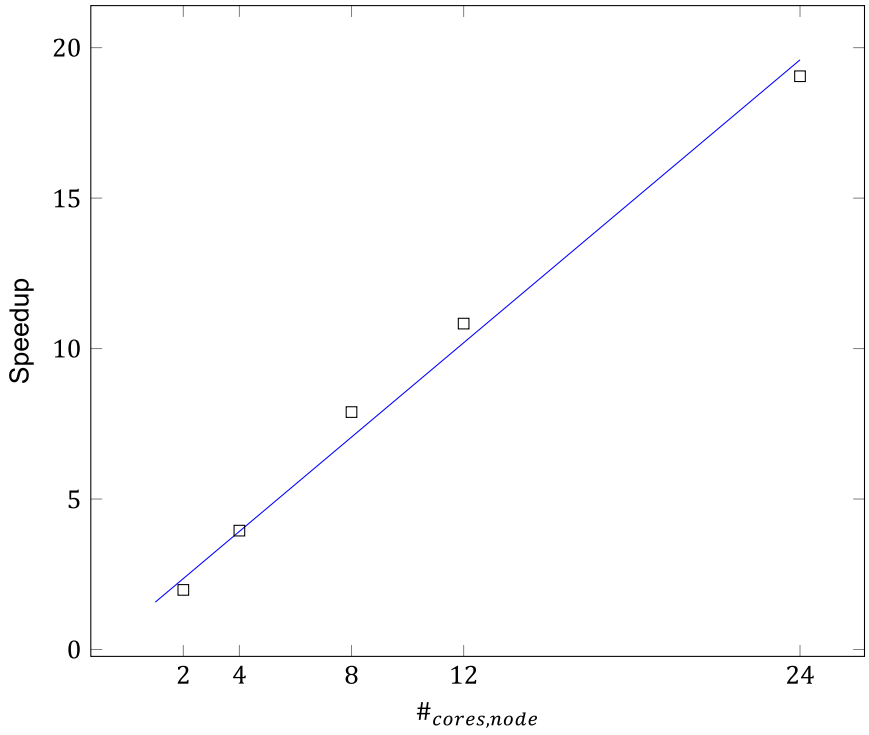


Figure 5.8: Speedup of the multiprocessing single node the application with respect to the number of CPU cores, with a regression line to indicate the linear trend.

Multi-node Profiling

After the single node application has reached a certain maturity in functionality and performance, a multi-node version has been put into works. For a multi-node execution, PBS server should be called to submit a job array, where each subjob is an instance of the single node application. The server call should specify resource allocation details such as number of nodes that should execute a given subjob, along with CPU and memory allocation for each node. It should also define job array properties such as size of the array, allowed maximum wall time per job and extra commands that should be passed on each call. The whole process

is automated by writing a simple wrapper application which prepares a PBS job submit script with desired properties and a front-end UI which accepts resource specification and dataset location from the user.

Similar to the single node profiling, an experimental approach has been taken to investigate multi-node performance, constructing PBS jobs by only varying one performance influencing parameter at a time while keeping the rest of the parameters constant. The intentions behind the experiments are:

- To have an initial measure of achievable speedup by multi-node implementation.
- To evaluate the effect of different parameters of resource configuration to the multi-node performance, which is mainly driven by the concurrency that PBS server determines as a function of the resource configuration.
- To determine the size and scaling properties of parallelization overhead in relation to the computation time.
- To evaluate a strategic choice between dividing the execution among many 'skinny' (low CPU core count) nodes or a few 'fat' (high CPU core count) nodes, through the evaluation of multi-node, multiprocessing execution.
- To put the outcomes together in order to achieve a performance model of the parallel implementation.

Three experiments, all using a model subspace constructed with identical parameters (one hyperparameter and 50 cross-validation executions for each complexity and hyperparameter tuple), have been devised:

1. Running PBS job arrays of varying size $|\mathcal{J}|$ where each subjob allocates $\#_{nodes,job} = 1$, $\#_{cores,node} = 1$ and $\#_{ram,node} = 8$.
2. Allocating a single node with varying $\#_{cores,node}$ and $\#_{ram,node} = 8$ per subjob, $|\mathcal{J}| = 2048$.
3. Allocating a varying number of nodes per subjob, where each node has a single core cpu and 8 GB of memory, while job array size is kept constant at $|\mathcal{J}| = 2048$.

First, the effect of job array size to performance has been put to test. During the initial runs, it has been found out that PBS server in ASML cluster is configured to admit a maximum job size of $|\mathcal{J}| = 10000$. Adhering to the familiar logic of slicing the problem space on the dimension of complexity, powers of two are used as possible selections of job array size, 8192 setting the higher limit. From this number, the number of jobs are scaled down and an accompanying varying number of atomic tasks per node have been chosen to ensure the whole model space is being processed at each run. Average concurrency has been determined by observing the number of subjobs that are executed concurrently throughout the run. Average execution time has been calculated from the start and end times of the batch job, as reported by PBS.

As it can be seen from Table 5.2, size of the batch job array does not influence concurrency. Given identical node resource configurations, achievable concurrency is dictated only by cluster availability. Moreover, total time lost on overhead increases with the increased ratio of number of jobs to concurrency. Average overhead is found by dividing the total calculated overhead to the number of batch subjob runs per execution. It has been found that average overhead is fairly stable among different runs.

During job runs, it has been consistently observed that the scheduler spends some time assigning subjobs to resources, before concurrency limits are reached and assigned subjobs

Table 5.2: The effect of job array size to execution performance for Lab implementation.

|J|	**2048**	**4096**	**8192**
$\#_{\mathbf{concurrency,avg}}$	1578	1535	1573
Average Execution Time [mins]	14.85	17.38	21.13
Average Overhead [mins]	1.80	1.56	1.55
Total Overhead [mins]	3.60	4.68	9.30

start executing in batch. Each subjob exhibit single node execution characteristics, and exit in batch. This behavior repeats, until the whole job array is exhausted.

Based on this observation, average and total overhead has been calculated by using a simple modeling of parallel execution time $t_{\mathcal{J},par}$ from the ratio of job array to concurrency, the sequential execution time of each node instance for processing p atomic tasks $t_{\mathrm{T}(p)}$, and assuming fixed overhead:

$$t_{\mathcal{J},par} = \lceil \frac{|\mathcal{J}|}{s_{conc}} \rceil (t_{\mathrm{T}(p)} + t_{overhead}) \tag{5.3}$$

The results derived from the model is further visualized in Figure 5.9 to emphasize the fact that overhead penalty from PBS job allocation and resource optimization routines are in fact making up a considerable amount of the total execution time. In order to minimize overhead, concurrency should be close in size to the job array size so that allocation routines are not repeated multiple times during a run.

Keeping the job array size fixed, the amount of computing power allocated to each job is determined by the number of CPU cores per node and the number of nodes per subjob. A number of runs have been executed with a varying value for both these parameters and the average concurrency as well as total execution time is recorded. The results are summarized in Table 5.3. The main observations are:

1. Within the boundaries of cluster availability, the relationship with the number of nodes per subjob and concurrency is inversely proportional and can be assumed as linear.

2. The concurrency goes down as the number of CPU cores per node goes up, almost in a linear fashion. Bearing in mind that multiprocessing performance has been observed to scale in a sublinear fashion, the total execution time per run suffers from reduced concurrency despite multicore performance gains.

Another observation is that a skinny node strategy, where each subjob gets a number of nodes with low CPU core count performs worse than a fat node strategy, where each subjob gets one node with a high CPU core count. Figure 5.10 visualizes the effect of both distribution parameters on the execution time. Inspecting the runtime, this behavior is due to the added overhead for acquiring a large number of nodes in the cluster is more than the single node performance loss due to sublinear gains of having multiprocessing with many cores. This further shows that the single-node application has been tuned successfully for multi-core performance.

The observed behavior from all multi-node profiling paves way to a model that aims to capture the combined effect of PBS configuration parameters on performance.

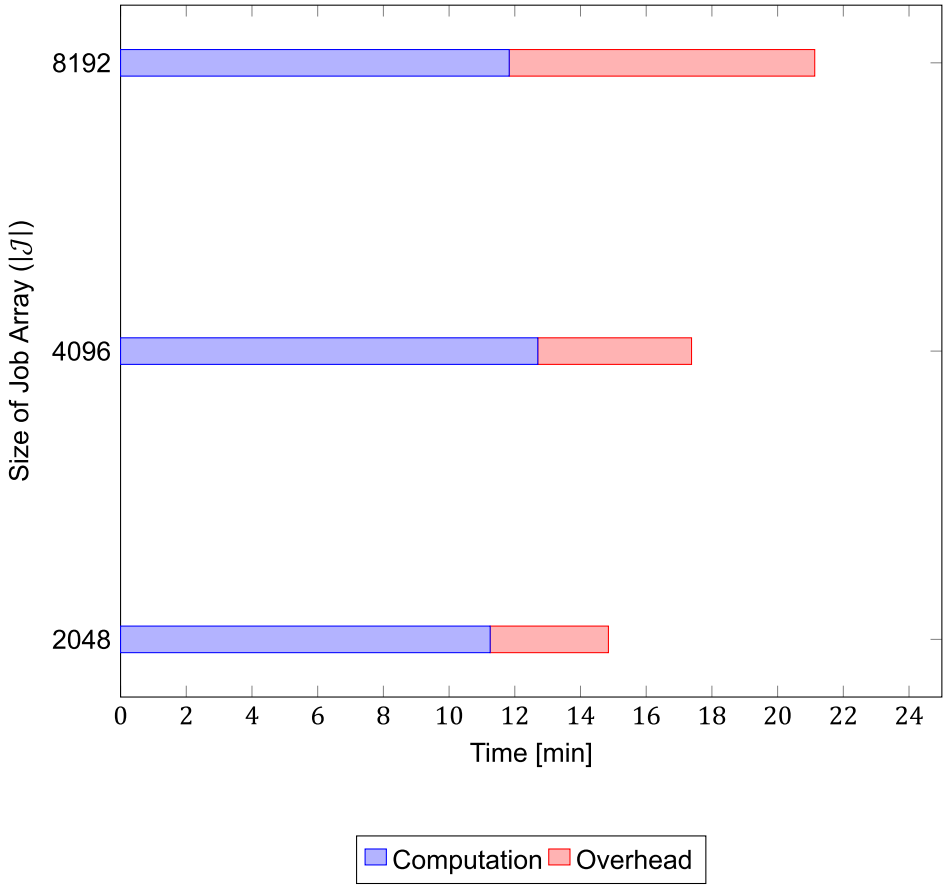


Figure 5.9: The portion of computation and overhead for a varying sizes of job array in Lab implementation.

5.1.5. Performance Model

The defining constraints on performance model are the task scheduling principles of PBS. In PBS, an array of subjobs are run in batches of concurrency. That is to say, PBS uses an opaque internal algorithm to decide how many subjobs are going to be run concurrently depending on the amount of available computational resources, queue properties and task priority. The behavior of the algorithm can be manipulated via a number of configuration parameters, but the dynamic scheduling algorithm cannot be explicitly commanded for a fixed size of concurrent subjob batches.

Based on the execution model, one can calculate the required size of the job array $|\mathcal{J}|$ for a model space of size P, given execution parameters for number of atomic tasks to be executed by a single node task p and number of nodes assigned for each subjob $\#_{nodes,job}$ as follows:

Table 5.3: Effect of per-node resource requests to execution concurrency for Lab implementation.

	$\#_{cores,node}$ ($\#_{nodes,job} = 1$)			$\#_{nodes,job}$ ($\#_{cores,node} = 1$)		
	1	**2**	**4**	**2**	**4**	**8**
Average Concurrency	1710	861	394	1710	752	374

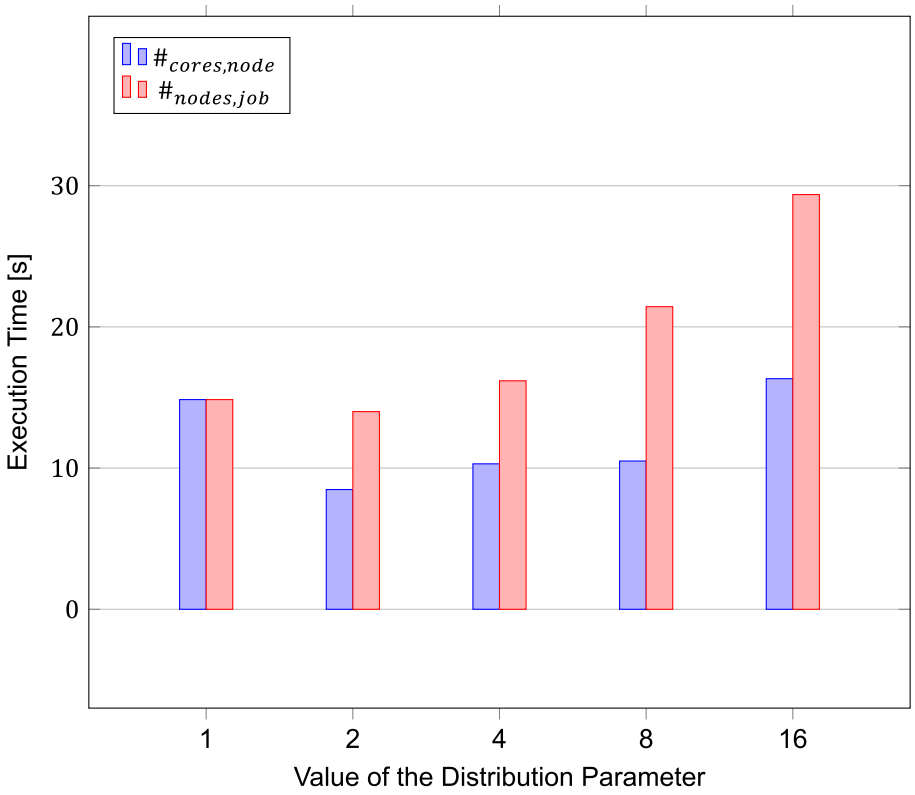


Figure 5.10: Comparison of scaling behavior between using number of nodes per subjob as the scaling parameter and number of CPUs per node as scaling parameter for Lab implementation.

$$|\mathcal{J}| = \frac{P}{p * \#_{nodes,job}} \tag{5.4}$$

The job array whose size is determined by equation 5.4 is executed by PBS in parallel. Most important thing to note about the achievable concurrency is that, there are natural theoretical limits to it, namely the total number of CPU cores in the cluster and size of the PBS job array.

$$s_{conc,max} \leq s_{jobarray},$$
$$s_{conc,max} \leq \#_{cores,cluster}$$

Given the cluster is occupied by other tasks at any given time, this theoretical limit is almost never reached. However, the concurrency at any time can be maximized by an informed selection of execution parameters. As subsection 5.1.4 outlines, a number of tests have been conducted to understand the effect of execution parameters, particularly $|\mathcal{J}|$, $\#_{nodes,job}$ and $\#_{cores,node}$ on the PBS scheduling algorithm. The tests suggest that that there is a linear, inverse relationship with the number of nodes queried for each subjob, $\#_{nodes,job}$ and the number of concurrent subjobs that PBS admits. Same relationship can be said to exist between $\#_{cores,node}$ and scheduling behavior of PBS. Furthermore, $|\mathcal{J}|$ does not have an effect on concurrency, given that the maximum achievable concurrency is smaller than or equal to the size of the job array. Thus, one can conclude that the practical limit that is set by the cluster availability, is further scaled by the choice of $\#_{nodes,job}$ and $\#_{cores,node}$:

$$s_{conc} = \frac{s_{conc,max}}{\#_{nodes,job} * \#_{cores,node}} \tag{5.5}$$

Keeping in mind the batch execution behavior of PBS, one can define an estimate for total parallel execution time, in terms of sequential execution time of a single node task on a node equipped with c cores, $t_{T(p,c)}$, and overhead of scheduling and resource allocation t_o per a batch of concurrent executions (which was observed to take a consistent amount of time for large enough batch jobs) in the following way:

$$t_{\mathcal{J},par} = \lceil \frac{(|\mathcal{J}|}{s_{conc}} \rceil (t_{T(p,c)} + t_o) \tag{5.6}$$

Keeping in the multiprocessing performance that has been observed in the single node tests, one can assume a linear speedup relationship for the sake of simplicity. With this constraint, and the the definitions provided by equations 5.4, 5.5, and 5.6, the estimated parallel execution time can be rewritten as an equation that is dependent to the cluster configuration parameters:

$$t_{\mathcal{J},par} = \frac{P}{p * \#_{nodes,job}} \frac{1}{s_{conc}} (t_{T(p,c)} + t_o)$$

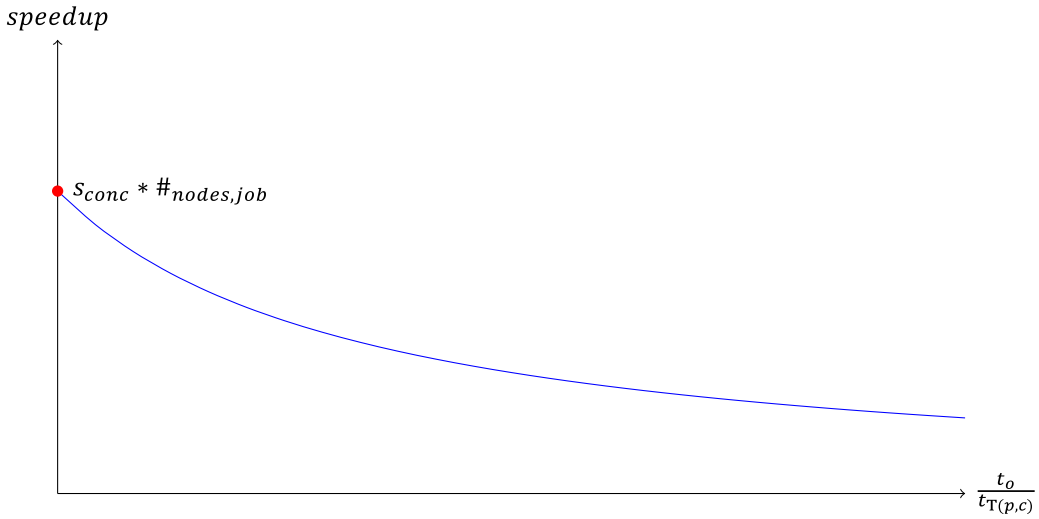


Figure 5.11: Graph for visualising the effect of overhead on speedup, based on the developed performance model for Lab implementation.

On the other hand, The sequential execution time for the modelspace of size P can be defined in terms of a single node task $t_{T(p)}$ consisting of p atomic tasks can be estimated according to the following relation:

$$t_{\mathcal{J},seq} = \frac{P}{p} t_{T(p,c)} \tag{5.7}$$

Finally , using the estimations for $t_{\mathcal{J},par}$ and $t_{\mathcal{J},seq}$, an estimate of speedup from parallel execution can be defined as:

$$\begin{aligned} \text{speedup} &= \frac{t_{\mathcal{J},seq}}{t_{\mathcal{J},par}} \\ &= (s_{conc} * \#_{nodes,job}) \frac{t_{T(p,c)}}{t_{T(p,c)} + t_o} \end{aligned} \tag{5.8}$$

Equation 5.8 indicates that the ratio $\frac{t_o}{t_{T(p,c)}}$, defines the achievable speedup, whose maximum value is determined by $s_{conc,max}$. Higher overhead time with respect to the computation time is detrimental for the speedup. The effect of $\frac{t_o}{t_{T(p,c)}}$ on achievable speedup is visualized in Figure 5.11.

Overhead is mostly coming from PBS scheduling time before every batch of execution, a slice of time where where no computation takes place. This can be minimized by setting $s_{jobarray} \approx s_{conc}$. This approach would also simultaneously maximize $t_{T(p,c)}$ without sacrificing concurrency.

5.2. Fab Implementation with Spark

The experience gained during the Lab implementation paves way to a Fab implementation which utilizes the Hadoop based factory environment of ASML HPC. Single-node computation toolchain, memory usage trends and multi-CPU acceleration concepts of the lab mapped to fab with a few modifications for adopting to the change of the parallelization framework, as it will be explained in depth by upcoming sections. Furthermore, user interface and visualization of Lab implementation are mimicked, albeit with a few small changes.

This section will begin with a description of the parallel programming model that is redefined in accordance with the programming paradigm and constraints of Spark. For a Spark based implementation, the steps for parallel execution follows the dataflow. These steps have been explained, and problem decomposition that agrees with Spark's definition for a task and a job has been given. These definitions naturally lead to the parallel programming model that works well for the specific problem at hand.

The constraints imposed on parallel programming model also effects parallelized toolchain choice, and motivated the move from a pure Spark implementation to a PySpark based implementation. The section will continue with explaining the challenges faced while utilizing Spark ML API. Then, along with a concise definiton of PySpark, the reasons of using this extra layer of tools for implementation will be explained.

Although Fab implementation has a more monolithic approach in development of the client-side software, a functional breakdown is still instrumental in understanding what the client-side software does and how. Implementation of functionality was not always straightforward, and certain solutions had been developed for roadblocks that are observed. These solutions will be laid out during the functional breakdown.

For Fab implementation, how multinode performance scales with cluster configuration parameters is investigated and a performance model is extracted from the findings. This performance model is used to analyze experiments that encompass a complete solution for mapping model optimization.

5.2.1. Parallel Programming Model

Parallel programming model of the Fab implementation is based on Spark's dataflow based programming approach. The parallel task can be seen as an example of model parallelism, which is similar to a data parallelism, where the distributed data represents the modeling attributes and execution parameters for computation of a model, see Figure 5.12.

With the Lab implementation, individual model parameters that the node will work on were not explicitly being passed, as the communication system was rather rudimentary. Instead, an enumerated index was passed during a remote shell call to every individual node, where the index can be resolved by the node to identify where the particular model for calcula-

tion lies in the model space. From there on, each node would have taken in the index and execute a combinator function to obtain the parameters. In contrast Spark's programming model fully relies on RDD, a highly abstract and scalable parallel data structure that has been explained in more depth in section 3.4. Having this tool available, it makes sense to compute all combinations that make up the total model space on the driver program and then ”parallelize” these combinations into an RDD, where each partition of RDD will contain all parameter information that a node needs to compute a model.

The Lab implementation was based on placing the dataset on a network drive and then passing the directory to the nodes, where the script being executed would read the data. This approach was causing a disk read and write bottleneck, in addition to the network bottleneck. Although with the sample dataset this bottleneck was not a limiting issue, one should admit that relying on networked disks for distributing the dataset is not the most scalable approach, especially considering that with the increasing number of lithography machines and experiments on each machine, the modeling effort may require to be extended to bigger datasets in the future. An obvious workaround would have been to pass the dataset via MPI broadcasting routines, but an MPI based implementation is taken out of scope of the Lab implementation. Spark offers a similar broadcasting functionality to MPI, built-in. Therefore, it was natural to improve the programming model of the Lab implementation and to free the application from disk dependency for distribution of the dataset.

Once a parameterized model space is constructed, parallelized and once each node receives the broadcasted dataset, the modeling pipeline can execute. Each worker node have access to an identical PySpark environment, where the ingredients to construct the necessary pipeline is available. One can see the dataset running through the pipeline that is constructed using the parameter grid partition that each node holds as a mapping operation from the dataset to the resulting model. After each task (mapping of each RDD partition) is finished, the results can be collected back to the driver node.

5.2.2. Execution Model

Similar with the Lab implementation, Fab implementation is designed to minimize communication between processes that get executed on different nodes. In Spark context, this means relying on narrow transformations to avoid shuffling operations. Remembering that narrow transformations mean any partition of a resulting child RDD of a transformation should only depend on a single partition of the parent RDD, all dimensions of the model space satisfy the narrow transformation principle. However, in order to calculate the average of cross-validation set scores, only a wide transformation can be defined, where many partitions of the parent RDD would be mapping to one partition of the child RDD.

Following the strategy that guided the Lab implementation, one can achieve an execution that is only composed of narrow transformations and shuffling can be avoided. If a task of Spark based execution model is defined as performing cross-validation to the subset of the model space that is defined by p “atomic” tasks, an atomic task $\tau(\gamma_n, \lambda_l)$ of the execution model is performing the modeling with γ_n in complexity using regularization hyperparameter λ_l. There needs to be no shuffling between the nodes in the mapping performed by the machine learning pipeline.

Implementing the execution model with the above description, the job of performing the grid search on the model space is composed of only one stage. The resulting Spark DAG is a very straightforward one, as shown in Figure 5.13.

5.2.3. Auxilliary Functions

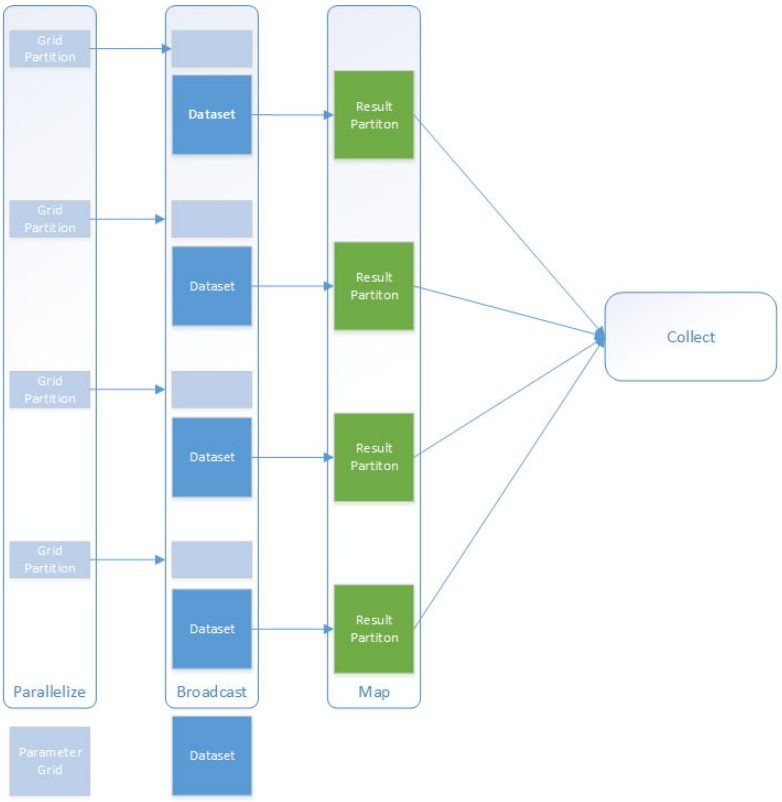


Figure 5.12: Visualization of high level programming model for the Fab implementation.

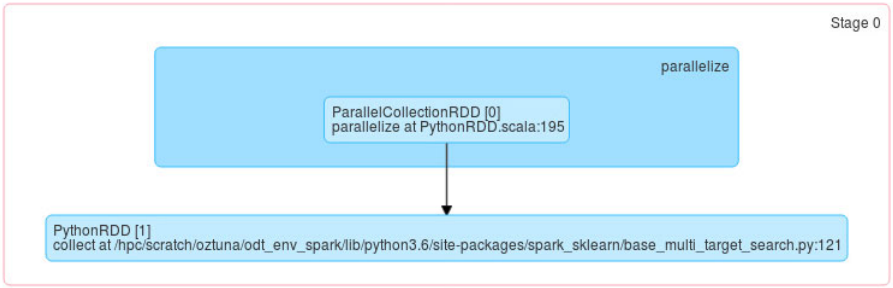


Figure 5.13: Visualization of Spark execution DAG for the Fab implementation.

Spark Cluster Setup Script

For submitting the application, a Spark cluster should be set up in ASML HPC. For this reason, a script that had been written by Frederico Valente was slightly tweaked and integrated with the application submission from the Driver node. The script performs following actions in order:

1. A Hadoop delegation token is created upon asking the user for credentials.
2. A Master node with the specified hardware is requested from PBS. The script waits until the PBS returns a master node.
3. Configuration directories are created in a network drive.
4. A specified number of Worker nodes with specified hardware are requested from PBS. these Worker nodes point to the configuration directories. The script does not wait until PBS returns all the Worker nodes, therefore one can start the Spark application execution and workers can join the cluster as they are administered.
5. Spark Master node URL and WebUI URL are returned, for submission and monitoring of applications, respectively.

Toolchain and Development Environment

By default, Spark comes with a comprehensive machine learning package. This library provides model selection tools for splitting the dataset into train and validation subsets and perform a cross validation routine on a parameter grid. Model selection tools require an estimator, a set of candidate parameters and an evaluator. The estimator can be a single modeling algorithm or a pipeline, where the resulting model is compared to another one using the metric provided by the evaluator. A general execution flow goes as follows:

1. `TrainValidationSplit` splits the input data into seperate training and test datasets.
2. For each training and test pair, `CrossValidator` iterates through a set of ParamMaps by fitting the estimator (ex. `RegressionEvaluator`) using the parameters, get the fitted model and evaluate the model's performance using the evaluator.
3. `CrossValidator` returns the best-performing set of parameters.

A parameter grid can be constructed using `ParamGridBuilder` utility provided by Spark. By default, the candidate parameters from the grid are evaluated in serial. In the latest versions, Spark started providing an option for parallel parameter evaluation. However, this parallel execution is not at cluster level but node level. In other words, multiple tasks are created for parameter evaluation and these tasks are executed in cluster in parallel fashion as long as the resources allow. The main parallelism that Spark ML framework provides is input data parallelism, and this is a design choice that Spark developers have made. The choice is understandable, considering that Spark is meant to be efficient in operating with massive amounts of data. However, the problem that this thesis is trying to address is a specific form of data parallelism, where the input data is rather small, but the parameter grid that defines the model space is massive. For such kind of problems, the latest machine learning libraries from Spark do not provide a high-performance solution.

After hitting the bottleneck described as above, the experiences from Fab implementation started to guide the toolchain choice. As explained before, Python is a language with comprehensive data science libraries such as Scikit-learn providing high level functions needed to tackle many problems including the one that is the subject of this thesis. Combined

with Jupyter development environment, it enables a developer to rapidly develop high performance single node applications that are maintainable and extendable. For Python developers, Apache Spark foundation provides an API called PySpark, based on Py4J, with the intent to facilitate Python programmers to work with Spark. In addition to PySpark, libraries such as NumPy and Scikit-learn are utilized for Fab implementation, in a similar fashion to the Lab implementation.

Visualization and User interface

Due to different definition of cluster and job properties, the Fab implementation has a slightly different user interface. It provides the following options to the user:

- Modeling options:
  - Path to aggregated dataset.
  - Resampling method for dataset partitioning.
  - The ratio of dataset partitioning.
  - Regularization hyperparameter range and resolution.
  - Number of cross-validation sets.
  - Model selection metric and criterion for cross-validation.
- Parallel execution options:
  - Number of worker nodes.
  - Specification of the worker nodes, in terms of number of CPU cores and amount of memory in GB.

A visualization module is implemented for aggregated view of model scores and the estimation of prediction accuracy.

It offers the end user the following options of data and visualization:

- Data options:
  - Statistical information about cross-validation scores of all models.
  - Modeling parameters of winning models for each level of complexity.
  - Cross-validation scores with respect to the choice of regularization hyperparameter.
  - Modeling parameters and prediction estimation of the best model.
- Graphing options:
  - Linear graphs.
  - Histogram (for score data).

5.2.4. Application Profiling

Efforts for profiling the Fab implementation are focused on multi-node profiling, as extensive single node profiling has been performed with the Lab implementation. Moreover, Spark WebUI already provides insight to the single node performance through access to standard streams of the processes executing in the Worker nodes, as well as statistical data on timings of task execution.

The main goal of multi-node profiling is to investigate the effect of parallel execution parameters to the application performance. For all experiments, the problem is defined as performing all atomic subtasks defined by the tuples (γ_n, λ_l), where $|\Lambda|$ is chosen to be 1 and $|\Phi|$ is 50, to have identical problem size used for profiling the Lab implementation. By experimenting on a subset of the model space, the optimum settings for processing of full model space are sought to be found. The tunable parameters of parallel execution are the number of atomic tasks each Spark task is going to process, the number of CPU cores each worker node will have, and the number of CPU cores that will be allocated to each executor process. A controlled experiment setting for these there parameters are identified as:

1. For a fixed number of worker nodes having a fixed number of CPU cores each and where all CPU cores in each worker node assigned fully to one executor process, investigate the effect of varying number of atomic tasks per Spark task on the performance. The model space used performs the grid search on one hyperparameter, with 50 cross-validation executions for each complexity and hyperparameter tuple.

2. For a fixed number of total CPU cores for the Spark cluster and a fixed number of atomic tasks per Spark task, observation of the effect of having different number of CPU cores per worker node. The model space used is identical to the first experiment.

3. For a Spark cluster with fixed settings for every other parameter, assigning a varying number of CPU cores per executor task and observing the effect on performance. The model space used is kept same with previous experiments.

This section will explain the findings in detail.

The first experiment is performed on a cluster with 256 worker nodes with 8 CPU cores each, where the number of atomic tasks per Spark task has been varied through powers of two, namely 8, 32, 1024 and 4096. Naturally, when each Spark task is composed of a bigger subset of the model space, the time taken for each task increases accordingly, as well as task deserialization time. However, as each Spark task gets bigger, the number of identical tasks that has to be performed goes down, which potentially means a reduction in scheduling overhead. The experiments performed are meant to put mentioned hypotheses into test. Also, the size of the collected result on the driver memory (and time it takes to aggregate the result from worker nodes over the network) at the end of the execution flow is also expected to increase considerably with the increasing number of Spark tasks (with each task being composed of less subtasks), due to the memory overhead of increased number of result objects coming from each task.

Table 5.4 paints a clear picture for the verification of the hypotheses. As the number of atomic tasks per Spark task get smaller, each individual task get less time to finish. However, the whole selected subset takes more time to process, and the result size gets larger, increasing memory requirements for the driver node considerably. When monitored using SparkUI, it was moreover observed that the increase in processing time does not only come from scheduling overhead. Combined with the observation that the time taken to collect the result into the driver node after the execution of tasks are reported as done increases with decreasing number of atomic tasks per Spark task, one can conclude that the majority of difference between execution time is a result of network and processing usage to aggregate a high number of individual Spark tasks. After the number of atomic tasks per Spark task reaches a certain number (i.e. 1024), the parameter starts to have (almost) no effect on performance, but only on the size of the aggregated result.

The second experiment is about the distribution of fixed amount of resources of a cluster among a varying number of worker nodes. The parallel programming model involves two different approaches to exploit the parallelism to the problem. There is parallelism among concurrent execution of Spark tasks, and there is parallelism among concurrent multiprocessing

Table 5.4: Profiling results of Fab implementation on a subset of the model space with varying numbers of atomic tasks per Spark task.

n_{subset}	n_{task}	$t/task$ [s]			$t/problem$ [s]	result size [KB]
		min	median	max		
8	131072	1	1	2	960	9481420436
32	32768	7	8	11	780	2371289369
1024	1024	96	108	144	558	75305557
4096	256	390	420	558	558	19755295

Table 5.5: Profiling results for the Fab implementation, investigating effect of number of CPUs per node to performance, when the total number of CPUs in cluster is kept constant.

$\#_{cores,node}$	$\#_{nodes,cluster}$	$t/task$ [s]			$t/problem$ [s]
		min	median	max	
2	1024	306	456	576	570
4	512	168	234	276	558
8	256	96	108	144	558
12	171	72	78	96	516

execution of subatomic tasks within a Spark task. Both parallel executions have their associated speedup limits which need to be investigated to find the right balance for the distribution of parallel execution. The speedup limit for Spark level concurrency is from scheduling, executor process creation and network communication overhead, whereas the speedup limit of single-node multiprocessing comes from process creation and context switching overhead at OS level.

The performance investigation of the second profiling experiment for the Fab implementation is shown in Table 5.5. Noting that, one can conclude that the difference in is within the boundaries of Spark's accuracy of reporting the execution time. This shows that, given identical total resources for the cluster, concurrency can be achieved using intra-node multiprocessing or inter-node cluster application levels, without a considerable impact on the performance.

The third experiment is similar to the second one in the sense that it is about how to distribute a fixed amount of resources of a cluster among different approaches to parallelism, however it has a more exploratory nature due to a hypothesis that stems from the execution model of Spark. In Spark, a worker node assigns all of its resources to a single executor JVM instance, unless specified otherwise. Using the configuration parameters, one can specify the number of CPUs that gets assigned to each executor, and if a worker node has enough resources to host more than one instance, it will do so. The potential gain in having more than one executor instance on a worker is to avoid the overhead stemming from creation of each JVM instance.

A series of experiments, outlined in Table 5.6, have been performed to investigate whether an optimum tuning for the parameter in question can be found. The results show that varying the number of CPUs per task and thereby potentially spawning multiple executors in each node does not seem to bring a performance benefit in terms of the time it takes for the subset to be processed or median scheduler delay per task.

5.2.5. Performance Model

This section describes the development of a model to estimate the scaling behavior of Fab implementation, emphasizing the observations of single and multi-node profiling efforts as

a tool for understanding parallel execution performance. The model seeks to specify total execution time as a function of cluster properties and problem size.

One can start with a high level, reasonably straightforward performance model equation that is to be broken down to individual contributors using empirical data, to estimate the parallel execution time $t_{|\mathcal{J}|,par}$ for a problem space size of $\mathcal{J}$. An example is:

$$t_{\mathcal{J},par} = E\left[t_{\mathrm{T}(p,c)}\right] \frac{|J|}{\#_{nodes,cluster}} + t_o \tag{5.9}$$

where $E\left[t_{\mathrm{T}(p,c)}\right]$ is an estimation for the time it takes for a single Spark task composed of p individual atomic tasks, being executed at a node where c amount of CPU core resources are assigned to each Spark task. As soon as Spark application starts to execute, total number of tasks to be executed is known. If one assumes homogeneous distribution of tasks, the total time taken for program execution, at the highest level, is a simple multiplication of estimated time per task by the ratio of concurrency found by dividing the total number of Spark tasks to the number of nodes that can concurrently process, $\frac{|J|}{\#_{nodes,cluster}}$. Execution time needs to be added by parallelization overhead t_o to achieve a measure for the total time spent executing the parallel task. Homogeneity assumption is not exact, as the size of the models to be computed by each Spark task can vary by an order of magnitude, provided that each of them are composed of a low number of atomic tasks. As the number of atomic tasks per Spark task increases, the homogeneity among Spark tasks improves accordingly. Still, the high level view of the performance model should give a good estimation, given an accurate estimation $E\left[t_{\mathrm{T}(p,c)}\right]$ can be extracted from the profiling experiments.

There are two ways to obtain an estimation of task execution time, as described above: one based on single node profiling results and the other based on multi-node profiling results. The single node results provide a more detailed information, in the form of a breakdown for each task's execution. Moreover, this detailed information is obtained with higher accuracy, simply because Spark WebUI provides timing information in granularity of 100 milliseconds, whereas a Python script can be profiled more accurately using cProfile, as it was done for single node experiments. However, even each run is conducted multiple times to have a statistical understanding, it is not feasible to reach the statistical accuracy of a multi-node experiment, where the task gets executed potentially thousands of times, by only experimenting on a single node.

Nevertheless, there is extensive information on execution time for Spark tasks composed of varying number of atomic tasks, processed by a varying number of CPU cores on a single node. One can average all these results to come up with a reasonable estimate for $E\left[t_{\mathrm{T}(p,c)}\right]$, as outlined in Equation 5.10:

$$E\left[t_{\mathrm{T}(p,c)}\right] = \chi \frac{p}{\#_{cores,node}} \tag{5.10}$$

where χ is a constant determined to be 0.90 [s] based on the collected experimental data,

Table 5.6: Profiling results for the Fab implementation, invesgating effect of number of cores per task (equivalently, executors per node) to performance, when the total number of CPUs in cluster is kept constant.

$\#_{cores,task}$	$t/task$ **[s]**			$t/problem$ **[s]**
	min	**median**	**max**	
2	354	522	570	570
4	186	264	282	564
8	90	132	144	570

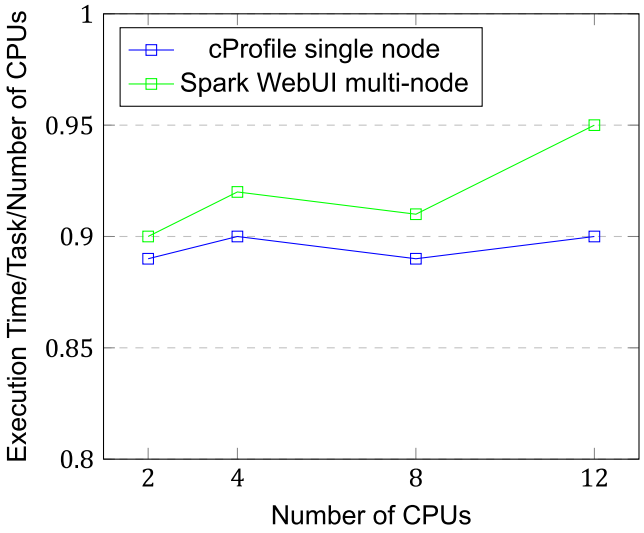
Table 5.7: Table showing average execution time of one atomic task divided by the number of CPU cores used on a single node.

$\#_{cores,node}$	Number of Experiments	Average Execution Time per Number of Cores [s]
1	6	0.89
2	6	0.90
4	6	0.90
8	6	0.89
12	5	0.90
24	4	0.91

that is laid out in Table 5.7.

$E\left[t_{T(p,c)}\right]$ can also be appropriated using the median task time reported by Spark WebUI. This would be an estimation based on the multi-node profiling results. Figure 5.14 shows the data collected from Spark WebUI as well as the results that are outlined in Table 5.7, and more importantly, how two different estimations compare with respect to $\#_{cores,node}$.

Figure 5.14: Estimation of task execution time per number of CPU cores, using different sources of data.



Naturally, there is a second part of Equation 5.9 one needs to address for a complete view of the performance model and that is the parallelization overhead. The only source of information for this part is the multi-node experiments. Therefore the breakdown of parallelization overhead is driven by the information that Spark WebUI provides to the user, namely the median statistics $\tilde{t}_{scheduling}$ and $\tilde{t}_{serialization}$ on scheduling and serialization of Spark tasks, respectively. Moreover, communication overhead can be measured, when it is defined as the time it takes for the driver to collect the node results after all the tasks that compose a Spark job is completed. Thus, parallelization overhead can be appropriated using the information at hand as:

$$t_o = [\tilde{t}_{scheduling} + \tilde{t}_{serialization}]\frac{|J|}{\#_{nodes,cluster}} + t_{communication} \tag{5.11}$$

where communication overhead is characterized with the measured network bandwidth and size of the result that is aggregated at the driver node. Furthermore, one can specify the communication overhead as a function of the total amount of data that needs to be communicated divided by the network bandwidth:

$$t_{communication} = \frac{\mu |J|}{bw_{network}} \tag{5.12}$$

where μ is found to be around 73KB per Spark task where $|\Lambda|$ is chosen to be 1 and $|\Phi|$ is 50, including object overhead. Available network bandwidth can in principle be collected as statistics during a job execution.

Putting all parts of Equation 5.9 together, one can estimate total time that parallel execution of a given problem will take. Performance models of both Lab and Fab implementations will be put to test by execution of both on full model space and observing the scaling behavior.

6

Results and Comparison

After implementations are optimized for performance and profiled, experiments are conducted to answer quantitative research questions about performance scaling and performance model accuracy.

The initial profiling of both implementations offer promising results in terms of speedup, given the model space is kept within the default boundaries defined by the baseline sequiential implementation. Naturally, scaling behavior of the performance increase obtained from both parallel implementations should be investigated and compared with each other. The reason for this is two fold. Investigation of scaling behavior is necessary for verifying the performance models defined during implementation and initial profiling. Moreover, a data driven modeling effort is expected to benefit from larger spaces of candidate models, and one can hope to reach to better predictive models through finer parameter tuning and benefit of larger datasets, as this space increases.

Investigation of scaling behavior has been driven with respect to two main questions:

- How does parallel execution time scale, when orthogonal dimensions of the model space get larger, necessitating the exhaustive search to be performed on an increased number of candidate models.
- What is the effect of an increase in the size of the dataset provided to modeling as input, to parallel execution times?

The following sections lay out the experimental approach and obtained results, as well as insights gained on parallel execution of both implementations.

6.1. Model Space Scaling

First investigation on performance scaling of the implementations have been performed with varying sizes of complete model space along the directions that are parameterized so that the users of application can specify them. Keeping in mind that the problem space is always meant to spawn the whole combinations of model complexity, the scalable dimensions are the size of the hyperparameter set and number of cross-validation partitions.

Experiments have been conducted in both parameterized dimensions, keeping the dataset fixed. The cluster setup for all model space scaling experiments has been defined as a fixed

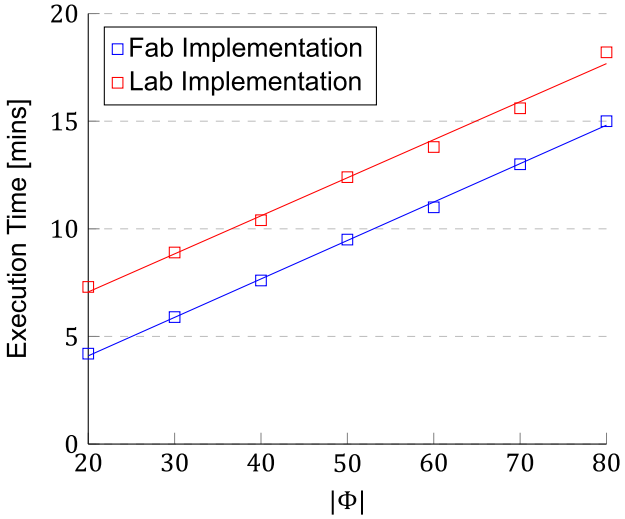


Figure 6.1: Performance scaling of Lab and Fab implementations with respect to the model space size along dimension of cross-validation partitions set.

amount of resources, where $\#_{nodes,job} = 256$, $\#_{cores,node} = 8$ and $\#_{ram,node} = 8$. For the first set of experiments, $|\Lambda|$ is scaled through a range of values, namely $(5, 10, 15, 20)$ where $|\Phi|$ is kept constant at 50. The second set of experiments spans $|\Phi|$ through $(20, 30, 40, 50, 60, 70, 80)$ while keeping $|\Lambda| = 1$. Figure 6.2 and Figure 6.1 show the execution time of runs, along with linear trend lines.

Comparing the scaling along two orthogonal dimensions of the model space, one should note that the hyperparameter set, $|\Lambda| = 1$, consists of only one element for the cross-validation scaling tests. This is admittedly not a realistic approach for a real data science application, as one would like to span a range of hyperparameter values to optimize the regularization effort. However, as the cross-validation partitions are likely to have a high number (as used in the experiments), a combination of this high number and a large hyperparameter set would have lead to long experimentation times, while not contributing further to an analytical observation of the scaling trend seen in Figure 6.1. On the other hand, hyperparameter scaling tests provided a good opportunity to test the performance of implementations in a realistic setting, while enabling to run through an iteration of experiments in a feasible amount of time.

The results show that execution time scales linearly on both dimensions of the model space, as implicitly suggested by the performance models of both implementations. Furthermore, linear scaling of atomic task execution time for single node application is observed from SparkUI reports, which further supports the observation.

The quality of the linear fit to the results have been measured by the R^2 values. For the fab implementation and varying number of elements for the hyperparameter space Λ, the linear trend line has $R^2 = 0.9826$. For different numbers of cross-validation partitions, again measured for fab implementation, the linear trend fits with quality $R^2 = 0.9807$. In the light of these values, one can conclude that the time complexity of the fab implementation is $\mathcal{O}(|\Lambda|)$ and $\mathcal{O}(|\Phi|)$.

Another observation is that, there is a considerable difference between the performances of Lab and Fab implementations, which becomes more pronounced as the model space gets bigger, although the trend for fab implementation remains fairly linear. There is an offset between the scaling behavior of two implementations, as can be seen in Figure 6.1. Looking at the profiling results from the previous chapter, one can conclude that this offset is not due to execution time and scaling of it, but due to the scaling of scheduling overhead that is

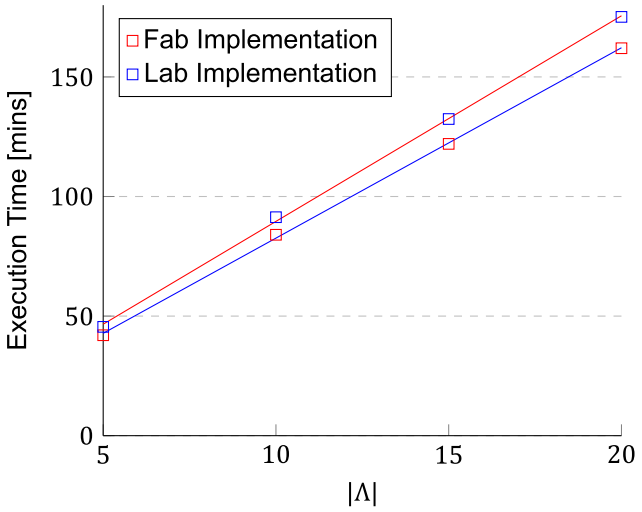


Figure 6.2: Performance scaling of Lab and Fab implementations with respect to the model space size along dimension of hyperparameter set.

more prominent for the Lab implementation compared to the fab implementation. Furthermore, although the concurrency of the Lab implementation was fairly stable during multiple runs, job array size scales with the model space, increasing the scheduling overhead. Thus, scheduling overhead is fairly consistent among repeated experiments, but is increasing as the model space gets bigger.

For verification of the Lab performance model, the results from scaling the number of cross-validation partitions have been used. As the execution model is flattened across this dimension of model space, different values of $|\Phi|$ has no effect on parallel execution aspects such as job array size and number of atomic tasks per node. The only part of the performance model that scales with the dimension is $t_{\mathrm{T}(p,c)}$ and this provides a more controlled set of experimental data for verification. Expected average overhead that is calculated with the data gathered from profiling results are observed to be inaccurate, and the average overhead is recalculated as 1.36 seconds per batch execution. This value is used as the new overhead parameter for estimating the parallel execution time of model space scaling.

Parallel execution and problem size parameters are fed into the equation Equation 5.1.5 and the comparison are laid out on Table 6.1. As it can be seen from the delta between estimated and actual parallel execution times, the performance models can be utilized to provide expectations about the order of magnitude to the user before actually performing an execution.

A similar approach is taken for verification of the Fab performance model. It is important to remind that the performance model for Fab implementation use the previous statistical data collected from SparkUI for estimating atomic task execution time. It is also dependent on a few paralel execution parameters, namely the total number of nodes and number of subtasks. As can be seen from Table 6.2, the estimations provide. An observation to note is that that Fab implementation executes more consistently then Lab implementation, both in terms of execution and overhead times. Overall, Spark provides a more determinate parallel execution environment than PBS for this particular application.

Table 6.1: Comparison of performance model estimates and actual execution times for the Lab implementation, scaling in cross-validation dimension.

$\|\Phi\|$	Estimate [min]	Actual [min]
20	6.3	7.3
30	8.8	8.9
40	10.2	10.4
50	12.2	12.4
60	14.1	13.8
70	16.0	15.6
80	18.0	18.2

Table 6.2: Comparison of performance model estimates and actual execution times for the Fab implementation, scaling in cross-validation dimension.

$\|\Phi\|$	Estimate [min]	Actual [min]
20	4.1	4.2
30	5.9	5.9
40	7.5	7.6
50	9.1	9.5
60	11.1	11.0
70	12.7	13.0
80	14.3	15.0

6.2. Dataset Scaling

Another important aspect for the scalability of the parallel application is how it evolves, keeping the model space fixed, as the dataset evolves to contain more information. Assuming wafer align models that utilize the application will use HOWA3 descriptions, the column dimension of a two-dimensional representation of wafer data is not expected to change. Therefore, dataset is expected to scale only along the row dimension, which is equal to the number of unique wafers used for alignment modeling. As the number of wafers used in modeling increases, the dataset will have more rows.

Throughout the development and profiling phase, the a dataset provided by the overlay team has been used, where the number of wafers represented (or in HOWA3 descriptions, number of rows in the dataset) are 90. To investigate the impact of both smaller datasets than this particular sample, and bigger ones, experiments are conducted where $\#_{rows,dataset}$ are $(30, 60, 90, 120, 150)$. The experiments showed that the impact of number of rows to performance within this range is very small and not easily detectable by the accuracy of the Spark WebUI reports. Single node experiments have been conducted and the magnitude of the impact of an increasing number of rows to single node performance is also seen to be very small.

As the datasets of sizes that are likely to be used in production grade overlay modeling was not indicative of any scaling behavior, a different approach was taken for further experimentation. The input dataset size (artificially) has been increased to row numbers of $(300, 3000, 30000)$ to see whether exponential growth in dataset would lead to a performance difference. Indeed at these admittedly unrealistic sizes, as seen in Table 6.3, scaling behavior exposes itself. This can be further understood by inspecting the graph visualization presented in Figure 6.3.

One should also note that dataset scaling behavior extending towards very large datasets is closely coupled to the complexity of linear algebra routines and regression methods chosen at a single node level. Furthermore, any selected linear algebra implementation and regression method is not expected to have any effect on cluster related performance criteria such as

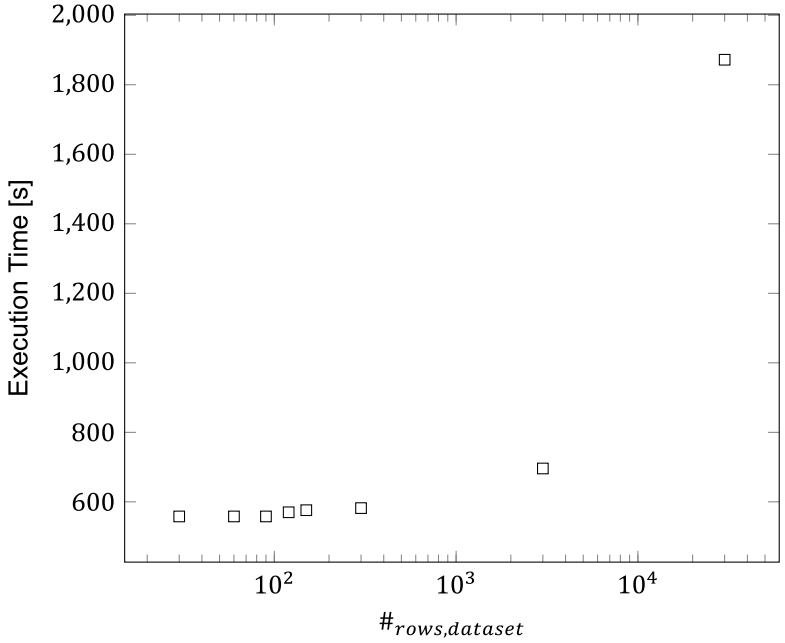


Figure 6.3: Scaling results for the Fab implementation, investigating effect of number of rows in the input dataset to execution time, when the cluster size is kept constant.

Table 6.3: Scaling results for the Fab implementation, investigating effect of number of rows in the input dataset to execution time, when the cluster size is kept constant.

$\#_{rows,datasaet}$	$t/task$ **[s]**			$t/problem$ **[s]**
	min	**median**	**max**	
30	95	106	144	558
60	96	107	144	558
90	96	108	144	558
120	98	112	144	570
150	98	112	144	576
300	99	115	147	582
3000	102	131	153	720
30000	319	360	476	1860

scheduling and communication overhead due to the execution model of Fab implementation. Combined with the experimental observations laid out in this section, one can conclude that dataset sizes that are likely to be used for overlay modeling purposes in forseable future have no effect on performance in terms of execution time of the whole program.

7

Conclusions

This thesis project has the aim of designing and comparing different parallel implementations for exhaustive grid search along the whole model space for finding the optimum mapping from HOWA3 description of a wafer align model to the overlay model. The search algorithm leads to an effectively intractable problem as long as sequential implementation is concerned, but a parallel implementation using the technologies provided by ASML HPC pave the way to tackle the challenge.

Upon investigation of a sequential baseline implementation, bottlenecks of the execution have been identified. A number of parallel concepts have been developed using different frameworks that are exposed to the ASML HPC developer community by the platform maintainers. Among these concepts, the most promising ones with respect to a defined set of criteria have been chosen to carry on with the implementation effort. During the concept investigation and later, it has been established that, albeit embarrassingly parallel execution of individual model calculations, an internode communication pattern is neccessary if full parallelism of the problem is to be exploited. With the motivation that the level of inherent parallelism of the problem being orders of magnitude higher than the number of cores available in ASML cluster, a communication-free execution model has been devised by flattening the model space along the dimension of cross-validation; the part of the algorithm that necessitates internode communication. By confining model calculations that belong to the same cross-validation effort to a single node and then accelerating the algorithm using multiprocessing, communication overhead has been minimized to only be present during the broadcast and result collection phase of the parallel execution pipeline.

It has been show that a PBSPro based implementation can scale on HPC with a parallel efficiency of 66%, under the realistic condition that the user would not have a chance to profile and optimize for concurrency behavior of PBSPro right before runtime. Majority for the loss of efficiency stems from the overhead experienced while issuing batches of jobs to the scheduler. The relatively opaque execution behavior of the scheduler led to expanding the scope further towards a Spark based implementation, as trial runs suggested that Spark engine has a potential to improve efficiency as well as allow for better control over the parallel execution of the program.

The second implementation has an increased efficiency of 82%, paving a way for speedup of almost 1700x for a cluster with 2048 cores. The resource utilization for both implementations has been observed to be higher than 90% in average. Furthermore, it has been shown experimentally that performance scales linearly over the model space dimensions. As the dataset scales to the sizes that are likely, execution time for the whole problem does not change significantly. For datasets that are orders of magnitude larger than size of a traditional dataset

sizes, the impact becomes more prominent. This is mostly related to the computational complexity associated with the linear algebra routines of single node program.

The baseline sequential implementation is expected to take, by extrapolation, 2590 hours to finish without multicore optimizations on an 8 core machine with 64GB of RAM. Refactoring the sequential implementation to use multiple cores through multiprocessing can drop this down to 330 hours for the same machine, and to 115 hours when the number of cores are increased to 24. For a model space constructed by 10 hyperparameters and 50 cross validation partitions, the parallel implementation takes 1.6 hours to find the most optimum model on a cluster of 256 machines composed of 8 cores and 8 GB of RAM.

7.1. Future Work

Although Hadoop Ecosystem (including Spark) is chosen as the current industrialization platform for cluster applications within ASML, there is an initiative in industry towards containerizing distributed applications using recently populer technologies such as Docker and Kubernetes. This would allow one to easily deploy a wider choice of toolchains that are fine tuned for the task at hand. Looking at the Python frameworks that are used for the single node implementation and libraries underlying these frameworks, one may argue that Dask has a potential to be a seamless fit, compared to rather convoluted execution path of PySpark, involving Python to Java VM conversions via Py4J. Naturally, for Dask to be supported in the industrialized platform, a migration from pure Hadoop stack. Such an opportunity lies later in the future, rather than sooner.

Cluster based implementations prove the worth of distributed computing, for performing exhaustive search over a very large candidate model space. If this approach is to be industrialized, it can provide value and insight to the accuracy of lithography machines, therefore be utilized frequently. Moreover, given the choices in modeling algorithms and a variety of statistical parameters to be used for accuracy prediction, iterative approaches mean that a formidable portion of ASML HPC may need to be allocated for overlay modeling purposes. Given already increasing utilization rates of the cluster, there is a potential issue in waiting.

As it is shown that a parallel solution to this problem has the potential to scale in speedup with increasing amount of processing cores, and given that low memory and communication requirements have been consistently observed, the problem looks like a perfect candidate for a GPU based implementation. By scaling out to a GPU, one can avoid allocating cluster resources constantly to overlay modeling efforts. The challenge is that, as experienced during a few weeks of trial for a GPU implementation, current and popular high level machine learning libraries that provide the needed functionality are not easily extensible to GPU execution. One should take on the challenge of a lower level programming effort, or use open source packages such as H2O4GPU as a starting point to build a framework that would provide the connection between Scikit-learn and CUDA. Nevertheless, this thesis has the potential to provide a guiding path to such an implementation.

Bibliography

[1] H. Levinson, *Principles of Lithography*, ser. SPIE Press Monograph. Society of Photo Optical, 2005.

[2] C. Mack, *Fundamental Principles of Optical Lithography: The Science of Microfabrication*. Wiley, 2011.

[3] M. Lapedus. (2015) Getting over overlay. [Online]. Available: https://semiengineering.com/getting-over-overlay/

[4] F. Harrell, *Regression Modeling Strategies: With Applications to Linear Models, Logistic and Ordinal Regression, and Survival Analysis*. Springer International Publishing, 2015.

[5] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning: with Applications in R*. Springer New York, 2013.

[6] S. Theodoridis and K. Koutroumbas, *Pattern Recognition*. Elsevier Science, 2008.

[7] A. Beck and Y. C. Eldar, "Regularization in regression with bounded noise: A chebyshev center approach," *SIAM Journal on Matrix Analysis and Applications*, vol. 29, no. 2, pp. 606–625, 2007.

[8] R. Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Proceedings of the 14th international joint conference on Artificial intelligence-Volume 2*. Morgan Kaufmann Publishers Inc., 1995, pp. 1137–1143.

[9] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2001.

[10] J. Sloan, *High Performance Linux Clusters with OSCAR, Rocks, OpenMosix, and MPI: A Comprehensive Getting-Started Guide*, ser. Nutshell Handbooks. O'Reilly Media, 2004.

[11] Altair, "Pbs professional 13.0 user's guide," Altair, Tech. Rep., 2015. [Online]. Available: https://www.pbsworks.com/pdfs/PBSUserGuide13.0.pdf

[12] T. Sterling, T. Sterling, G. Bell, W. Gropp, and E. Lusk, *Beowulf Cluster Computing with Linux*, ser. Scientific and Computational Engineering Series. Books24x7, 2002.

[13] Altair, "Pbs professional 14.2 big book," Altair, Tech. Rep., 2017. [Online]. Available: https://www.pbsworks.com/pdfs/PBS14.2.1_BigBook.pdf

[14] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the mpi message passing interface standard," *Parallel computing*, vol. 22, no. 6, pp. 789–828, 1996.

[15] S. Sur, M. J. Koop, and D. K. Panda, "High-performance and scalable mpi over infiniband with reduced memory usage: an in-depth performance analysis," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM, 2006, p. 105.

[16] P. J. Lu, H. Oki, C. A. Frey, G. E. Chamitoff, L. Chiao, E. M. Fincke, C. M. Foale, S. H. Magnus, W. S. McArthur, D. M. Tani *et al.*, "Orders-of-magnitude performance increases in gpu-accelerated correlation of images from the international space station," *journal of real-time image processing*, vol. 5, no. 3, pp. 179–193, 2010.

[17] W. Gao, N. T. T. Huyen, H. S. Loi, and Q. Kemao, “Real-time 2d parallel windowed fourier transform for fringe pattern analysis using graphics processing unit,” *Optics express*, vol. 17, no. 25, pp. 23 147–23 152, 2009.

[18] J. A. Van Meel, A. Arnold, D. Frenkel, S. Portegies Zwart, and R. G. Belleman, “Harvesting graphics power for md simulations,” *Molecular Simulation*, vol. 34, no. 3, pp. 259–266, 2008.

[19] nVIDIA, “Cuda c programming guide,” nVIDIA, Tech. Rep., 2018. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[20] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, “Shark: Sql and rich analytics at scale,” in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data*. ACM, 2013, pp. 13–24.

[21] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster computing with working sets.” *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[22] (2018, Nov) A tale of three apache spark apis: Rdds vs dataframes and datasets. [Online]. Available: https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html

[23] M. Beuckelmann, “Boosting numpy: Why blas matters,” Tech. Rep., 2017. [Online]. Available: http://markus-beuckelmann.de/blog/boosting-numpy-blas.html

[24] scikit-learn developers, “Why do i sometime get a crash/freeze with n jobs > 1 under osx or linux?” Tech. Rep., https://scikit-learn.org/stable/faq.html?why-do-i-sometime-get-a-crash-freeze-with-n-jobs-1-under-osx-or-linux. [Online]. Available: http://markus-beuckelmann.de/blog/boosting-numpy-blas.html

[25] T. Moreau, “Loky project description,” PyPI, Tech. Rep., 2019. [Online]. Available: https://pypi.org/project/loky/

[26] P. S. Foundation, “The python profilers,” Python Software Foundation, Tech. Rep., 2018. [Online]. Available: https://docs.python.org/2/library/profile.html