Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master's Thesis

GPPPy: Leveraging HPX and BLAS to accelerate Gaussian Processes in Python

Maksim Helmann

Course of Study:	Simulation Technology
Examiner:	Prof. Dr. Dirk Pflüger
Supervisor:	Alexander Strack, M.Sc.
Commenced:	May 10th, 2024
Completed:	August 14th, 2024

Abstract

Gaussian processes, often referred to as Kriging, are a popular regression technique. They are a widely used alternative to neural networks in various applications, e.g., non-linear system identification. Popular software packages in this domain, such as GPflow and GPyTorch, are based on Python and rely on NumPy or TensorFlow to achieve good performance and portability. Although the problem size continues to grow in the era of big data, the focus of development is primarily on additional features and not on the improvement of parallelization and performance portability.

In this work, we address the aforementioned issues by developing a novel parallel library, GPPPy (Gaussian Processes Parallel in Python). Written in C++, it leverages the asynchronous many-task runtime system HPX, while providing the convenience of a Python API through pybind11. GPPPy includes hyperparameter optimization and the computation of predictions with uncertainty, offering both the marginal variance vector and, if desired, a full posterior covariance matrix computation, hereby making it comparable to existing software packages.

We investigate the scaling performance of GPPPy on a dual-socket EPYC 7742 node and compare it against the pure HPX implementation as well as against high-level reference implementations that utilize GPflow and GPyTorch. Our results demonstrate that GPPPy's performance is directly influenced by the chosen tile size. In addition, we show that there is no runtime overhead when using HPX with pybind11. Compared to GPflow and GPyTorch, our task-based implementation GPPPy is up to 10.5 times faster in our strong scaling benchmarks for prediction with uncertainty computations. Furthermore, GPPPy shows superior parallel efficiency to GPflow and GPyTorch. Additionally, GPPPy, which only computes predictions with uncertainties, outperforms GPyTorch used with LOVE by a factor of up to 2.8 when using 16 or more cores, despite the latter using an algorithm with superior asymptotic complexity.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

Acronyms

GP Gaussian process. 17

GPPPy Gaussian Processes Parallel in Python. 17

HPC High Performance Computing. 17

HPX High-Performance ParalleX. 17

LML Log marginal likelihood. 25

LOVE LanczOs Variance Estimates. 55

MKL Math Kernel Library. 18

MPI Message Passing Interface. 20

NN Neural Networks. 24

oneDNN oneAPI Deep Neural Network Library. 54

OpenMP Open Multi-Processing. 21

RBF Radial Basis Function. 20

SI System identification. 17

SIMD Single Instruction, Multiple Data. 40

SLURM Simple Linux Utility for Resource Management. 59

STL Standard Template Library. 48

SWIG Simplified Wrapper and Interface Generator. 22

1 Introduction

Gaussian processes (GPs), also known as kriging, are a robust and versatile regression technique widely used in various fields [WR95], [RW06]. From geostatistics, where GPs are used for spatial data interpolation and prediction, to control theory and artificial intelligence, GPs provide a flexible framework for modeling complex, non-linear relationships [Mac+98; WR95]. Being fully probabilistic models, they can inherently estimate predictive uncertainty through posterior variances, making them especially valuable in domains where understanding the confidence of predictions is crucial [DFR15; DK17; SS17; WJ18].

Several popular software packages such as GPy [GPy12], GPflow [MvN+17], and GPyTorch [GPB+18], have made GPs accessible to a broad audience. These packages typically rely on Python implementations and aim to achieve satisfactory performance and portability through well-established numerical libraries like NumPy [HMW+20], TensorFlow [MAP+15], and PyTorch [PGM+19]. However, as the era of big data progresses, the size and complexity of the problems addressed using GPs continue to grow. This trend emphasizes the need for more efficient and scalable computational methods. Despite the increasing problem sizes, development efforts in GP software packages often prioritize the addition of new features over performance-critical optimizations. Enhancements such as parallelization and performance portability are frequently outsourced to backend systems, which may not fully leverage the latest advancements in High Performance Computing (HPC). As a result, there is a risk of sub-optimal resource utilization, potentially limiting the scalability and efficiency of GP implementations.

A scientific GP application in control theory where the aforementioned limitation can become a bottleneck is non-linear System identification (SI). Making predictions with a GP for unseen data requires the inversion of a covariance matrix that scales with the size of the training data set, a process that becomes computationally expensive in the context of Big Data[1] [LCOW19; LOSC20].

To address these challenges, the goal of this work is to develop a novel parallel library named Gaussian Processes Parallel in Python (GPPPy). This library is implemented in C++ and leverages the High-Performance ParalleX (HPX) runtime system [KDL+20], a modern C++ library designed for parallelism and concurrency. HPX facilitates task-based parallelism and provides a highly efficient execution model, making it well-suited for implementing computationally intensive algorithms. GPPPy aims to combine the high performance of C++ with the user-friendly interface of Python, ensuring that users can easily integrate the library into their existing workflows. The core components of GP regression within GPPPy will feature futurized, task-based parallelization,

[1] In this context, Big Data mainly refers to data sets with an extremely large sample size N, making them expensive to store and analyze.

which allows for dynamic scheduling and work stealing [BL99]. Additionally, the library will use a Basic Linear Algebra Subprograms (BLAS) backend [BPP+02] to handle numerically intensive operations, further enhancing its performance.

Our main contributions in this work include:

- A novel, fully asynchronous task-based GP library in Python that uses HPX and multiple different tiled algorithms,
- A performance analysis of the tiled algorithms for different tile sizes on multi-core systems,
- A thorough performance comparison between GPPPy, pure HPX, and the reference implementations GPflow and GPyTorch in a node-level strong scaling test on a dual socket AMD EPYC 7742 CPU.

The remainder of this report is organized as follows. Firstly, Chapter 2 reviews existing software frameworks for GPs, HPC, and Python bindings. Secondly, Chapter 3 discusses the fundamental concepts of GPs and the optimization techniques employed in GPPPy, providing a theoretical foundation for the library's implementation. In addition, the application of GPs for SI is outlined in this section. Chapter 4 then describes the numerical tiled algorithms. Subsequently, Chapter 5 discusses the software components used in GPPPy, including HPX for parallelism, the Math Kernel Library (MKL) [Int24] for numerical computations, and pybind11 [JRM17] for interfacing C++ with Python. Chapter 6 deals with the project structure and the implementation details of the reference libraries. Next, Chapter 7 presents the data used for benchmarking, describes the experimental setup in detail, and analyzes the performance results obtained with GPPPy in depth. This chapter also compares the obtained results against the reference implementations and includes various scaling plots. Finally, Chapter 8 summarizes the key findings of this thesis, discusses the implications of results, and outlines potential directions for future research and development.

2 Related work

In this chapter, the related work is reviewed. First, Python libraries that implement GPs are discussed. Next, HPC libraries are examined. Finally, methods for integrating C++ with Python through the use of Python bindings are covered.

2.1 Gaussian processes in Python

In recent years, the modeling of GPs has seen a significant increase in popularity due to their wide range of applications in machine learning and statistical modeling. The growing popularity has been further fueled by the development of several Python libraries that are easily accessible to end users due to their simple front-ends and extensive functionality. Among these, GPy [GPy12], GPflow [MvN+17], and GPyTorch [GPB+18] are the most prominent, particularly due to their comprehensive features and robust performance.

GPy, developed by the Sheffield machine learning group, is an established library offering a straightforward interface for GP modeling. It leverages NumPy for computations, which internally utilize multi-threaded linear algebra routines (BLAS [BPP+02] and LAPACK [ABB+99]) that are implemented in libraries such as MKL or OpenBLAS [WZZY13; XQY12]. GPy is well-suited for smaller-scale projects due to its simplicity and ease of use. However, for more complex and large-scale applications, modern alternatives like GPflow and GPyTorch are more suitable.

GPflow is built on TensorFlow [MAP+15], while GPyTorch is built on PyTorch [PGM+19]. Both leverage the powerful capabilities of their respective deep learning frameworks. Both TensorFlow and PyTorch employ a tape based system for automatic differentiation which is crucial for the optimization of machine learning algorithms such as GPs. Consequently, the tape based system not only facilitates the efficient computation of gradients but also enhances the flexibility and scalability of the models. Moreover, both GPflow and GPyTorch utilize MKL for various matrix and vector computations on Intel CPUs. The integration of MKL significantly accelerates computational performance by optimizing low-level operations. An additional benefit of using MKL together with TensorFlow or PyTorch is the ability to control parallelization. Users can set the number of threads employed by MKL, thereby managing the extent of parallel computation and optimizing the use of available hardware resources. Should other CPUs be used, e.g., AMD CPUs, then Eigen [GJ+10] or OpenBLAS are employed.

2.1.1 GPflow

GPflow is a powerful and flexible open-source library for GP models in Python and is leveraging TensorFlow to provide scalable and efficient solutions for probabilistic machine learning. The library offers a framework for constructing and fitting GP models. Furthermore, GPflow provides a high-level interface that abstracts much of the complexity involved in GP model formulation and optimization. A key feature of GPflow is its comprehensive support for a variety of kernel functions, which are fundamental components of GP models that define the similarity between data points and play a crucial role in the model's performance. GPflow includes commonly used kernels such as the Radial Basis Function (RBF) and periodic kernels. Another significant advantage of GPflow is its scalability. GPs typically have cubic time complexity with respect to the number of data points, which can be prohibitive for large data sets. GPflow addresses this challenge by implementing various approximation methods, such as sparse GPs and variational inference [Tit09], which reduce computational costs while maintaining model accuracy. In addition, GPflow leverages TensorFlow's GPU support to significantly accelerate computations. Utilizing GPUs for matrix operations and other intensive tasks allows GPflow to handle larger data sets and more complex models with increased efficiency.

2.1.2 GPyTorch

GPyTorch is a modern and highly efficient library for GP modeling in PyTorch, which is designed to make large-scale GPs accessible and scalable for a wide range of applications. By leveraging the power and flexibility of PyTorch, GPyTorch provides a user-friendly and extensible framework for researchers and practitioners working with probabilistic models. Similar to GPflow, GPyTorch introduces a unique approach to kernel design and composition. Kernels in GPyTorch are modular and can be easily combined, allowing users to construct complex covariance functions that better capture the underlying structure of their data. The library includes a variety of pre-implemented kernels, such as RBF and periodic kernels. A notable feature of GPyTorch is its deep integration with PyTorch, facilitating seamless incorporation into larger machine learning workflows. This integration enables users to leverage PyTorch's automatic differentiation, GPU acceleration, and dynamic computational graph capabilities, making model training and evaluation efficient and straightforward.

2.2 High Performance Computing

Current state-of-the-art in leveraging parallelization for GP regression has been substantially advanced by libraries like GPflow and GPyTorch. These libraries benefit from the underlying frameworks, TensorFlow and PyTorch, which utilize system-optimized libraries such as MKL, OpenBLAS, or Eigen. TensorFlow and PyTorch use these optimized libraries either by default or when built from source. However, GPflow and GPyTorch often prioritize additional features, such as approximation methods and sparse GPs, at the expense of integrating recent HPC advances. The standard for enhancing computational performance is through message-based execution models, specifically the Message Passing Interface (MPI) standard [GLS99] with implementations such as OpenMPI [GFB+04]. A disadvantage of MPI-based solutions is that they rely on global synchronization barriers. These barriers enforce synchronization at the end of each sub-task, thereby potentially delaying downstream computations that could proceed with partial data [BL+15; GKGK03]. This can significantly impact performance in complex applications such as coupled numerical simulations or SI in control theory [SL19]. Moreover, many MPI implementations focus on specific research questions, often resulting in compatibility issues with other implementations

[GFB+04] which is limiting their portability. Additionally, MPI is better suited for large problems on a small number of machines, as it tends to scale poorly for fixed-size problems when the number of cores increases [Gon14].

Task-based parallel programming models offer a compelling alternative to traditional MPI-based approaches for HPC to better exploit parallel efficiency and concurrency [Gon14; SGCD16]. For shared memory systems, language extensions like Intel Cilk Plus [Rob13] and libraries such as Intel's Threading Building Blocks [WP08] are considered industry standards [TDH+18]. Similarly, Qthreads [WMT08], a runtime system providing a library interface for lightweight thread management, is also utilized for optimizing performance in high-concurrency scenarios. For distributed memory systems, Chapel [CCZ07] and X10 [CGS+05] depict the most notable examples for experimental programming languages. Additionally, HPX [KDL+20] stands out as a C++ library that supports task-based parallelism. It can also run as back-end for Open Multi-Processing (OpenMP) [Ope08] or Chapel. In this work, we will use HPX given its ability to support application scalability and high parallel efficiency, alongside its alignment with modern C++ standards and performance portability. Furthermore, HPX not only outperforms conventional programming models but also facilitates the development of highly efficient and energy-conscious parallel applications [KHA+14]. Additionally, HPX offers advanced scheduling mechanisms, including work stealing [TDH+18]. HPX is an open-source, meritocratic community-driven project. Moreover, HPX guarantees the portability of the performance across NVIDIA, AMD, and Intel CPUs [DBK23].

A recent study [SP23] demonstrates the application of HPX in GP regression, specifically focusing on accelerating the computation of the inverse of a large covariance matrix, which is a common task within the GP framework. Generally, instead of computing this inverse directly, the Cholesky decomposition of the covariance matrix is computed. In their work [SP23], the authors implemented the tiled Cholesky decomposition by using tiled algorithms and a task-based asynchronous execution model that leverages the task dependency graph in HPX. The performed experiments showed that HPX exhibits superior performance and parallel efficiency compared to the MPI-based approach in PETSc [BAA+24; BGMS97] for the given application. Building upon this foundational framework, our work will introduce additional functionalities to develop the GPPPy library.

2.3 Python bindings

Initially, the main goal of HPC has been to improve computing performance, traditionally achieved through the use of compiled languages such as C/C++. While this still remains the main goal, the additional feature "multi-language support" is desired by many users to increase productivity and convenience [LPH23]. Python [PGH11], [Oli07], a dynamic language, fulfills these requirements without compromising the performance of computationally intensive kernels. The strategy is to implement performance-relevant sections in a lower-level language such as C++ and access this code through an extension module within the Python interpreter [KVL23; NTW22]. This approach leverages Python's extensibility with C-like languages, making it a preferred choice for HPC and other performance-sensitive applications [GLMM05]. As a result, the integration of C++ libraries into the Python environment via extension modules, also known as Python bindings, has become increasingly popular.

Several popular Python binding tools are available for integrating C++ libraries into Python, each one with unique features and benefits. Simplified Wrapper and Interface Generator (SWIG) [Bea96] is a powerful and versatile tool known for its ability to generate bindings for multiple languages, including Python, from a single interface file. Although SWIG is highly capable, it can be complex to set up and use compared to other options. BOOST.PYTHON [AG03], part of the Boost C++ Libraries, provides a powerful and flexible way to create Python bindings. However, it requires a deep understanding of both C++ and the Boost framework. pybind11 [JRM17] stands out for its ease of use and lightweight design as a header-only library, allowing developers to write binding code in C++ without learning an additional language. It exposes C++ types in Python and vice versa, making it ideal for creating Python bindings of existing C++ code. pybind11 can be easily installed using pip and it supports modern C++11/14/17/20 standards.

Phylanx [TWS+18], CxxExplorer [DB23], and PyCxxwrap[1] attempt to integrate HPX with Python to leverage the power of high performance parallelism within Python applications. In all these projects, the HPX runtime is managed through the `submit_work()` function, which initializes the runtime by creating a `manage_global_runtime` object. The `submit_work()` function then registers the current thread with HPX, executes the provided function as an HPX thread, waits for its completion, and finally shuts down the HPX runtime. The `manage_global_runtime` object gets deleted at the end of each call. However, this approach has some bottlenecks. First, each call to `submit_work()` reinitializes the HPX runtime if the previous call is completed and cleaned up. This repeated setup and teardown introduces unnecessary overhead. Second, the HPX runtime is not persistent across multiple `submit_work()` calls, leading to unnecessary reinitialization and teardown, which is inefficient for workloads that require frequent interactions with HPX. In our work, we adopt a different approach by directly writing the `hpx::start()` and `hpx::stop()` functions in a .cpp file, which are then exposed to Python via pybind11. This enables users to combine the HPX runtime with Python more seamlessly. In addition, the aforementioned approach provides users with better control over the HPX runtime and the maintenance of a single, consistent HPX runtime. This not only reduces overhead but also helps to avoid inconsistent states.

[1] https://github.com/matrixbot123/pycxxwrap/tree/main

3 Methods

This chapter discusses the fundamental concepts of GPs, including algorithms for computing prediction uncertainty and optimizing GP model hyperparameters. In addition, it outlines the application of GPs for SI.

3.1 Gaussian processes

GPs are a powerful and versatile framework for modeling complex, uncertain phenomena in various domains, ranging from tasks in supervised machine learning, such as classification and regression tasks, to engineering and the physical sciences [Gib98]. The main goal hereby is to find a non-linear function f that returns a function value for given data. To obtain f, the framework is provided with a feature matrix $Z = [\mathbf{z}_1, \mathbf{z}_2, \dots, \mathbf{z}_n]^\top$, where each entry is a feature vector $\mathbf{z}_i \in \mathbb{R}^D$ and observations $\mathbf{y} = [y_1, y_2, \dots, y_n]^\top$ and $y_i \in \mathbb{R}$.

The stated goal can be achieved through direct inference in the function space. A GP can be thought of as a generalization of the Gaussian distribution to infinite dimensions, where any finite collection of random variables follows a joint Gaussian distribution [RW06]. The GP can be completely defined through the mean $m(\mathbf{z})$ and covariance function $k(\mathbf{z}, \mathbf{z}')$ of a real process $f(\mathbf{z})$:

$$m(\mathbf{z}) = \mathbb{E}\left[f(\mathbf{z})\right], \tag{3.1}$$

$$k(\mathbf{z}, \mathbf{z}') = \mathbb{E}\left[\left(f(\mathbf{z} - m(\mathbf{z})\right)\left(f(\mathbf{z}') - m(\mathbf{z}')\right)\right]. \tag{3.2}$$

The resulting GP can be expressed as

$$f(\mathbf{z}) \sim \mathcal{GP}\left(m(\mathbf{z}), k(\mathbf{z}, \mathbf{z}')\right), \tag{3.3}$$

where random variables represent the value of $f(\mathbf{z})$ at any point $\mathbf{z}$. For simplicity, the mean function is assumed to be zero most of the time. However, this is not a requirement and the choice of the mean function can significantly impact the behavior of GP because the mean function provides a prior expectation about the underlying function f before any data is observed [Koc15].

As mentioned earlier, the GP model is a distribution over functions whose shapes and smoothness are defined by the covariance function, also known as the kernel function, $k(\cdot, \cdot)$. The kernel function defines the covariance, or similarity, between pairs of random variables in the input space. Selecting the right kernel function is crucial as it can strongly influence the generalization capability of the model [Duv14]. In our work, we choose the squared exponential covariance function

$$\operatorname{cov}\left(f(\mathbf{z}_i), f(\mathbf{z}_j)\right) = k\left(\mathbf{z}_i, \mathbf{z}_j\right) = \nu \cdot \exp\left[-\frac{1}{2 \cdot l}\sum_{d=1}^{D}\left(z_{i,d} - z_{j,d}\right)^2\right] + \delta_{ij}\sigma^2, \tag{3.4}$$

where D is the length of the feature vectors. If we postpone the introduction of the three parameters l, ν, σ^2, and focus solely on the kernel function, then it is evident that if two points $\mathbf{z}_i$ and $\mathbf{z_j}$ are considered similar by the kernel, then the respective function outputs are expected to be similar too. The mentioned three additional parameters, also called hyperparameters, length-scale l, vertical-scale ν and noise variance σ^2, are crucial for the quality of the GP model. In contrast to the hyperparameters of Neural Networks (NN), e.g., the learning rate or the number of hidden layers, that are tuned using techniques such as grid or random search, the parameters in a GP model are optimized directly within the code as part of the model fitting process using methods like maximum likelihood estimation. δ_{ij} depicts the Kronecker delta which is one if and only if $i = j$ and zero otherwise. The noise variance is stemming from the fact that typically, realistic modeling situations only output a noisy version of the underlying function value, $y_i = f(\mathbf{z}_i) + \epsilon$. For the noise, the assumption can be made that it is additive independent identically distributed Gaussian noise ϵ with variance σ^2.

Once the kernel function is selected the covariance matrix $K \in \mathbb{R}^{N\times N}$ can be computed by evaluating each combination of the given feature vectors with the kernel function. Following this, the joint distribution of the training outputs $\mathbf{y}$ and the function values at the test inputs $f\left(\hat{Z}\right)$ under the prior are [RW06]

$$\begin{bmatrix}\mathbf{y}\\ \hat{\mathbf{y}}\end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} K & K_{Z,\hat{Z}} \\ K_{\hat{Z},Z} & K_{\hat{Z},\hat{Z}}\end{bmatrix}\right), \tag{3.5}$$

where $K_{\hat{Z},Z} \in \mathbb{R}^{M\times N}$ is the cross covariance matrix constructed from feature vectors of observed data $Z \in \mathbb{R}^{N\times D}$ and unseen data $\hat{Z} \in \mathbb{R}^{M\times D}$. $K_{\hat{Z},\hat{Z}} \in \mathbb{R}^{M\times M}$ depicts the prior covariance matrix, computed through the feature matrix $\hat{Z}$ of unseen data.

To make predictions, the conditional distribution of $f\left(\hat{Z}\right)$ given the observations $\mathbf{y}$ is computed. The posterior distribution is given by:

$$f\left(\hat{Z}\right)|Z, \mathbf{y}, \hat{Z} \sim \mathcal{N}\left(\boldsymbol{\mu}_{\hat{Z}}, \Sigma_{\hat{Z}}\right), \tag{3.6}$$

where the mean of the distribution depicts the GP predictions for test inputs, expressed as

$$\hat{\mathbf{y}} = K_{\hat{Z},Z}K^{-1}\mathbf{y}. \tag{3.7}$$

Additionally, prediction uncertainty can be derived from the variance of the joint distribution. This uncertainty is represented by the diagonal elements of the posterior covariance matrix $\Sigma_{\hat{Z}}$, which provide the variances of the predictions at the test points. The posterior covariance matrix can be computed via

$$\Sigma_{\hat{Z}} = K_{\hat{Z},\hat{Z}} - K_{\hat{Z},Z}K^{-1}K_{Z,\hat{Z}}. \tag{3.8}$$

Typically, due to the number of observed data N being way larger than the number of test points M, the main bottleneck of the prediction (Equation (3.7)) and prediction uncertainty (Equation (3.8)) is the calculation of the inverse of the covariance matrix K. This computation generally has a complexity of $O(N^3)$, where N is the number of data points. However, directly inverting the covariance matrix K is not advisable. Instead, a more numerically stable and efficient method is to perform Cholesky decomposition to compute the Cholesky factor L of the factorized covariance matrix $K = LL^\top$. This is possible because the constructed covariance matrix is symmetric and positive definite. The biggest advantage of this approach is that once the Cholesky factor L is computed, it can be reused for the calculation of the predictions and prediction uncertainties. An in-depth discussion of the prediction uncertainty computation is provided in Section 3.2.

Next, the marginal likelihood is introduced as it is a crucial component when discussing GPs, particularly in the context of model selection and hyperparameter optimization [Koc19]. However, the Log marginal likelihood (LML) is numerically more stable. In both cases, the goal is to maximize the $\log \mathcal{L}$ of the training data $\mathbf{y}|Z \sim \mathcal{N}(\mathbf{0}, K)$:

$$\log \mathcal{L}\left(l, \nu, \sigma^2\right) = -\frac{1}{2}\log\left(|K|\right) - \frac{1}{2}\mathbf{y}^\top K^{-1}\mathbf{y} - \frac{N}{2}\log\left(2\pi\right). \tag{3.9}$$

Due to the precomputed Cholesky factor L, Equation (3.9) can be computed more efficiently:

$$\log \mathcal{L}\left(l, \nu, \sigma^2\right) = -\frac{1}{2}\sum_i^N \log\left(L_{ii}^2\right) - \frac{1}{2}\boldsymbol{\beta}^\top\boldsymbol{\beta} - \frac{N}{2}\log\left(2\pi\right), \tag{3.10}$$

where $\boldsymbol{\beta} \in \mathbb{R}^N$ is the solution of $\boldsymbol{\beta} = L^{-1}\mathbf{y}$. In practice, we minimize the negative LML

$$\ell\left(l, \nu, \sigma^2\right) = -\log \mathcal{L}\left(l, \nu, \sigma^2\right) = \frac{1}{2}\log\left(|K|\right) + \frac{1}{2}\mathbf{y}^\top K^{-1}\mathbf{y} + \frac{N}{2}\log\left(2\pi\right), \tag{3.11}$$

in order to optimize the hyperparameters. Hyperparameter optimization is discussed in Section 3.3.

Last but not least, we provide a pseudocode in Algorithm 3.1 for the implementation of GP regression [Wan23]. In this work, we build upon previous framework [SP23] that implemented the

Algorithm 3.1 Gaussian process regression (without optimization)

```
Require: $Z$: Training input
Require: $\mathbf{y}$: Targets
Require: $\hat{Z}$: Test input
Require: $k$: Kernel function
Require: $\left[l, \nu, \sigma^2\right]$: Hyperparameters
1: $L = \text{cholesky}(K)$
2: Solve $L\boldsymbol{\beta} = \mathbf{y}$
3: Solve $L^\top\boldsymbol{\alpha} = \boldsymbol{\beta}$
4: $\hat{\mathbf{y}} = K_{\hat{Z},Z}\boldsymbol{\alpha}$                      {Predictive mean (Equation (3.7))}
5: Solve $L\mathbf{v} = K_{Z,\hat{Z}}$
6: $\Sigma_{\hat{Z}} = K_{\hat{Z},\hat{Z}} - \mathbf{v}^\top\mathbf{v}$                 {Predictive variance (Equation (3.8))}
7: $\log \mathcal{L}(l, \nu, \sigma^2) = -\sum_i^N \log L_{ii} - \frac{1}{2}\boldsymbol{\beta}^\top\boldsymbol{\beta} - \frac{N}{2}\log 2\pi$          {LML (Equation (3.10))}
8: return $\hat{\mathbf{y}}$ (Predictions), $\Sigma_{\hat{Z}}$ (Variance), $\log \mathcal{L}(l, \nu, \sigma^2)$ (LML)
```

tiled Cholesky decomposition [DKL+16]. The next two sections will talk about the computation of prediction uncertainty, hyperparameter optimization, and how they are incorporated into the existing fully asynchronous task-based framework.

3.2 Prediction uncertainty

The prediction uncertainty is a key feature in GPs that distinguishes them from many other regression techniques. The obtained uncertainty is quantified by the predictive variance, which provides a measure of the confidence in the model's predictions at any given point. This is particularly useful in fields such as robotics and finance, where understanding the reliability of predictions can guide decision-making under uncertainty. As a side note, the predictive variance is influenced by the distance from observed data points and the underlying noise in the data, typically increasing as we move further from known data points. Furthermore, the prediction uncertainty depends solely on the input values Z and $\hat{Z}$ [Wan23].

The equation to compute the prediction uncertainty (Equation (3.8)) was introduced in the previous section. In the following, we talk about the intermediate steps of the calculation process and two possible ways to compute the uncertainty prediction. Algorithm 3.2 depicts the pseudo implementation steps of the first variant. Per default, line 3 only computes diagonal elements

Algorithm 3.2 Prediction uncertainty (Variant 1)

```
Require: $K^{\mathrm{diag}}_{\hat{Z},\hat{Z}}$: Diagonal of prior covariance matrix
Require: $K_{\hat{Z},Z}$ ($K_{Z,\hat{Z}}$): (Transpose) Cross covariance matrix
Require: $L$: Cholesky factor
1: Solve $LV = K_{Z,\hat{Z}}$                                   {Compute in-place forward substitution}
2: $V^{\mathrm{diag}} = \mathrm{diag}(V^{\top}V)$                 {Compute only diagonal terms of the product}
3: $\Sigma^{\mathrm{diag}}_{\hat{Z}} = K^{\mathrm{diag}}_{\hat{Z},\hat{Z}} - V^{\mathrm{diag}}$   {Predictive uncertainty (Equation (3.8))}
4: return Variance: $\mathrm{Var}(\hat{\mathbf{y}})\left(:= \Sigma^{\mathrm{diag}}_{\hat{Z}}\right)$
```

of the posterior covariance matrix. The computation of the full posterior covariance matrix is outlined in Algorithm 3.4. However, most of the time the full posterior covariance is not required. Hence, it is not computed as the former is much faster. The complexities of the calculation in Algorithm 3.2 are $O(MN^2)$ for a forward substitution (line 1), $O(MN)$ to compute diagonal elements of a matrix-by-matrix product using a dot product (line 2) and $O(M)$ to perform the subtraction of diagonal elements (line 3). Thus, the dominating term is the forward substitution in line 1.

The alternative approach to calculate the prediction uncertainty is based on the observation that $K_{\hat{Z},Z}K^{-1}$ is present in Equation (3.7), that computed predictions, and Equation (3.8), that computes prediction uncertainty. Thus, a different sequence of steps can be performed as outlined in Algorithm 3.3, to only compute $K_{\hat{Z},Z}K^{-1}$ once. Lines $1-3$ need to be computed only once and

Algorithm 3.3 Prediction uncertainty (Variant 2)

```
Require: $K^{\mathrm{diag}}_{\hat{Z},\hat{Z}}$: Diagonal of prior covariance matrix
Require: $K_{\hat{Z},Z}$ ($K_{Z,\hat{Z}}$): (Transpose) Cross covariance matrix
Require: $L$: Cholesky factor
1: $A_{\hat{Z},Z}LL^{\top} = K_{\hat{Z},Z}$
2: Solve $B_{\hat{Z},Z}L^{\top} = K_{\hat{Z},Z}$          {Set $B = AL$; Compute $B$ via forward substitution}
3: Solve $A_{\hat{Z},Z}L = B_{\hat{Z},Z}$                  {Compute $A$ via backward substitution}
4: $C^{\mathrm{diag}} = \mathrm{diag}(A_{\hat{Z},Z}K_{Z,\hat{Z}})$   {Compute only diagonal terms of the product}
5: $\mathrm{Var}(\hat{\mathbf{y}}) = K^{\mathrm{diag}}_{\hat{Z},\hat{Z}} - C^{\mathrm{diag}}$   {Predictive uncertainty (Equation (3.8))}
6: return Variance: $\mathrm{Var}(\hat{\mathbf{y}})$
```

can be reused for predictions. Similar to Algorithm 3.2 only the diagonal values of the posterior covariance matrix are computed. The complexity of Algorithm 3.3 is dominated by the forward and backward substitutions in lines $2-3$ that are $2 \times O(MN^2)$.

Comparing the complexity of both approaches against each other and additionally considering the time complexity of the prediction, we can observe that the first option involves $1 \times O(MN^2)$, $2 \times O(N^2)$, $2 \times O(MN)$, and $1 \times O(M)$. In comparison, the second option accounts $2 \times O(MN^2)$, $2 \times O(MN)$, and $1 \times O(M)$. Since the dominant term $O(MN^2)$ occurs (only) once in Algorithm 3.2, it is computationally less expensive. Therefore, it is more advantageous to use Algorithm 3.2 when computing the prediction uncertainty for big data sets.

Should the full posterior covariance matrix $\Sigma_{\hat{Z}}$ be calculated, then the calculation steps must be changed slightly. Algorithm 3.4 depicts the pseudo implementation of the calculation of the full covariance matrix. For clarity, the computations in lines 2 and 3 are separated, but in practice one

Algorithm 3.4 Full posterior covariance matrix

```
Require: $K_{\hat{Z},\hat{Z}}$: Prior covariance matrix
Require: $K_{\hat{Z},Z}$ ($K_{Z,\hat{Z}}$): (Transpose) Cross covariance matrix
Require: $L$: Cholesky factor
1: Solve $LV = K_{Z,\hat{Z}}$                {Compute in-place forward substitution}
2: $W = V^\top V$                            {Compute matrix-by-matrix product}
3: $\Sigma_{\hat{Z}} = K_{\hat{Z},\hat{Z}} - W$
4: return Posterior covariance matrix: $\Sigma_{\hat{Z}}$
```

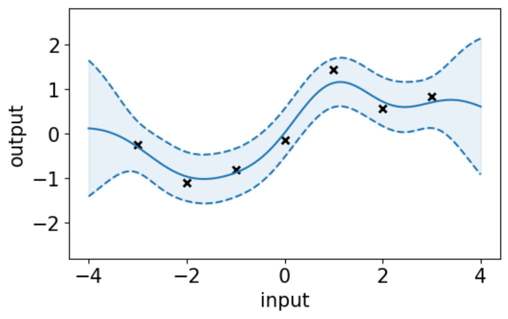
combines these two into a single computation that uses the `gemm()`[1] function, which computes a matrix-matrix product with general matrices. The complexities of the calculation in Algorithm 3.4 are $O(MN^2)$ for a forward substitution (line 1), $O(M^2N)$ to compute matrix-by-matrix product (line 2), and $O(M^2)$ to perform the subtraction of matrices (line 3). Thus, the dominating term is the forward substitution in line 1 because usually $N \geq M$.

3.3 Hyperparameter optimization

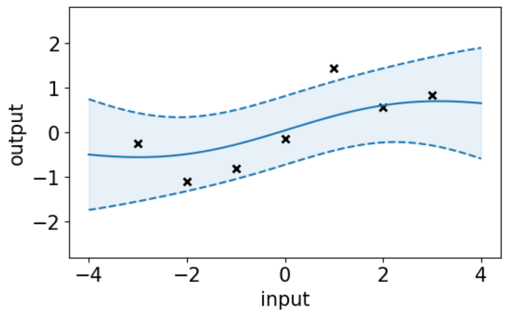
Kernel functions range from well-established options such as RBF to customized designs tailored to specific needs based on model requirements such as smoothness and differentiability [Duv14]. Therefore, the optimization of hyperparameters in kernel-based methods plays a very important role, as the hyperparameters have a direct impact on the quality of the GP model. In the following, a short illustration of how the hyperparameters influence the model and the benefits of their adjustment are discussed. Afterward, algorithms are introduced for optimizing the hyperparameters to improve the quality of the GP model.

As an illustration example, we take the hyperparameters of the squared exponential kernel (Equation (3.4)). Namely the length-scale l, vertical-scale (also known as signal variance) v, and noise variance σ^2. By varying the length-scale we demonstrate how the change can influence the

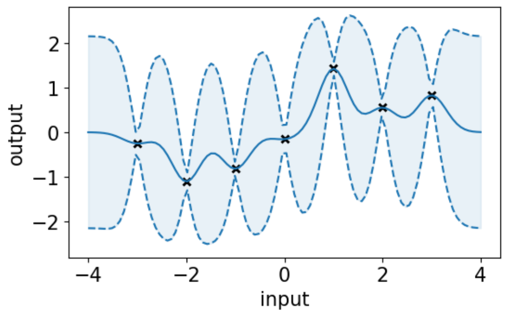
[1] `https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-dpcpp/2024-2/gemm.html#ONEMKL-BLAS-GEMM`



(a) $l = 1.0$



(b) $l = 3$



(c) $l = 0.3$

Figure 3.1: Varying length-scale for the same training data (**x**). (a) Mean function (blue line) and 95% confidence region (light blue area) obtained from a GP model with hyperparameters $(l, \nu, \sigma^2) = (1, 1, 0.1)$. Subfigure (b) and (c) again show the 95% confidence region, but for different hyperparameter values $(3, 1, 0.7)$ and $(0.3, 1.2, 0.005)$ respectively.

performance of the GP model. Figure 3.1 depicts three different scenarios. All of them show the same training data marked by (**x**), but different mean functions (blue line) and confidence regions (light blue area). For Figure 3.1a the mean function and confidence region were generated from a GP model with the squared exponential kernel with $(l, \nu, \sigma^2) = (1, 1, 0.1)$. As emphasized in Section 3.3, the confidence interval gets larger for test values that are further away from the training points (**x**). Recalling Equation (3.8) for predictive variance (uncertainty), it can be verified that when the test input is far from the training data, the elements of the cross covariance matrix are small because the kernel function typically decreases with distance. This makes $K_{\hat{Z},Z}K^{-1}K_{Z,\hat{Z}}$ small, and thus $\Sigma^{\text{diag}}_{\hat{Z}}$ approaches $K^{\text{diag}}_{\hat{Z},\hat{Z}}$, which is the vertical-scale ν. The prior standard deviation $\sqrt{\nu}$ represents our initial belief about the variability of the function. When data is sparse or absent, the prediction uncertainty reflects this initial belief, manifesting as larger error bars.

In contrast, if the length-scale is set to a larger value, e.g., 3 (as shown in Figure 3.1b), and the other two parameters are set after optimization. Then the resulting mean function is smoother. The larger length-scale implies that the correlation between function values at different points decays more slowly, leading to smoother variations over the input space. Despite the smooth mean function, the noise variance of 0.7 indicates that there is still a considerable amount of noise even in the observations.

Figure 3.1c depicts predictions from a process where $l = 0.3$. The vertical-scale and noise variance were optimized as in the previous case, resulting in $\sigma^2 = 0.005$. In this case, the noise parameter is reduced due to the greater flexibility of the GP with a smaller length-scale. This can be observed for the two data points at $x = -2$ and $x = -1$ in all three figures. For Figure 3.1a these are not completely fitted by the mean function, suggesting that the model views these variations as noise rather than sharp changes in the underlying function. On the other hand, Figure 3.1c displays a much more fluctuating mean function closely fitting the data points. The confidence intervals are narrow near the data but grow rapidly as we move away from them, indicating high uncertainty in regions without data.

Therefore, optimizing the GP model hyperparameters is crucial to ensure the model appropriately balances smoothness and flexibility. A balanced GP accurately captures the underlying structure of the data. To optimize the hyperparameters, we minimize the negative marginal likelihood (Equation (3.11)) by taking the negative gradient of $\ell\left(l, \nu, \sigma^2\right)$ with respect to each hyperparameter and then updating them using, e.g., the gradient descent algorithm. Assuming $\boldsymbol{\theta}$ is a vector of hyperparameters then the gradient w.r.t. parameter θ_i is

$$\begin{aligned}
-\frac{\partial}{\partial\theta_i}\ell(\boldsymbol{\theta}) &= -\frac{\partial}{\partial\theta_i}\log\left(p(y|X,\boldsymbol{\theta})\right) && (3.12)\\
&= \frac{\partial\left[\frac{1}{2}\log\left(|K|\right) + \frac{1}{2}\mathbf{y}^\top K^{-1}\mathbf{y} + \frac{N}{2}\log(2\pi)\right]}{\partial\theta_i} && (3.13)\\
&= \frac{1}{2}\frac{\partial\log(|K|)}{\partial\theta_i} + \frac{1}{2}\mathbf{y}^\top\frac{\partial K^{-1}}{\partial\theta_i}\mathbf{y} && (3.14)\\
&= \frac{1}{2}\frac{\partial(\mathrm{tr}(\log(K)))}{\partial\theta_i} - \frac{1}{2}\mathbf{y}^\top\left(K^{-1}\frac{\partial K}{\partial\theta_i}K^{-1}\right)\mathbf{y} && (3.15)\\
&= \frac{1}{2}\left\{\mathrm{tr}\left(K^{-1}\frac{\partial K}{\partial\theta_i}\right) - \mathbf{y}^\top K^{-1}\frac{\partial K}{\partial\theta_i}K^{-1}\mathbf{y}\right\} && (3.16)\\
&= \frac{1}{2}\left\{\mathrm{tr}\left[K^{-1}\frac{\partial K}{\partial\theta_i}\right] - \mathrm{tr}\left[\mathbf{y}^\top K^{-1}\frac{\partial K}{\partial\theta_i}K^{-1}\mathbf{y}\right]\right\} && (3.17)\\
&= \frac{1}{2}\left\{\mathrm{tr}\left[K^{-1}\frac{\partial K}{\partial\theta_i}\right] - \mathrm{tr}\left[K^{-1}\mathbf{y}\mathbf{y}^\top K^{-1}\frac{\partial K}{\partial\theta_i}\right]\right\} && (3.18)\\
&= \frac{1}{2}\mathrm{tr}\left[\left(K^{-1} - K^{-1}\mathbf{y}\mathbf{y}^\top K^{-1}\right)\frac{\partial K}{\partial\theta_i}\right] && (3.19)
\end{aligned}$$

The derivation for Equation (3.13) results in three possible gradient computation approaches:

1. $\frac{1}{2}\mathrm{tr}\left[\left(K^{-1} - K^{-1}\mathbf{y}\mathbf{y}^\top K^{-1}\right)\frac{\partial K}{\partial\theta_i}\right]$
2. $\frac{1}{2}\mathrm{tr}\left[K^{-1}\left(I - \mathbf{y}\mathbf{y}^\top K^{-1}\right)\frac{\partial K}{\partial\theta_i}\right]$
3. $\frac{1}{2}\mathrm{tr}\left(K^{-1}\frac{\partial K}{\partial\theta_i}\right) - \frac{1}{2}\mathbf{y}^\top K^{-1}\frac{\partial K}{\partial\theta_i}K^{-1}\mathbf{y}$

First of all, it should be noted that all three approaches require the inverse of the covariance matrix K^{-1}. While the inverse is not computed directly but through the Cholesky factorization and subsequent forward and backward substitution, the computation is still rather expensive, as it scales with $O(N^3)$. Thus, this term can become a computational bottleneck when the data size gets large. For the implementation, we use the third equation, as it has the lowest summed up complexity. While all of them are quite similar the last expression has terms that scale with $1 \times O(N^2)$ and $1 \times O(N)$ instead of $2 \times O(N^2)$. Additionally, another $O(N^2)$ operation is saved when computing the inverse of K^{-1}, as this term also occurs in the loss function (Equation (3.10)). Since K^{-1} is needed for the computation of the gradient, it is not beneficial to compute $L^{-1}\mathbf{y}$ first and later the backward substitution, but rather to reuse the computed inverse.

The gradient $\frac{\partial K}{\partial \theta_i}$ needs to be computed for each hyperparameter separately. For the three parameters in the squared exponential kernel, we obtain

$$\frac{\partial K}{\partial \tilde{\sigma}^2} = \frac{\partial \left\{ f(\tilde{v}) \cdot \exp\left[-\frac{1}{2 \cdot f(\tilde{l})^2} \sum_{d=1}^{D} \left(z_{i,d} - z_{j,d} \right)^2 \right] + \delta_{ij} f\left(\tilde{\sigma}^2\right) \right\}}{\partial \tilde{\sigma}^2}, \tag{3.20}$$

$$= \delta_{ij} \cdot \frac{\partial f(\tilde{\sigma}^2)}{\partial \tilde{\sigma}^2}, \tag{3.21}$$

$$\frac{\partial K}{\partial \tilde{v}} = \frac{\partial \left\{ f(\tilde{v}) \cdot \exp\left[-\frac{1}{2 \cdot f(\tilde{l})^2} \sum_{d=1}^{D} \left(z_{i,d} - z_{j,d} \right)^2 \right] + \delta_{ij} f(\tilde{\sigma}^2) \right\}}{\partial \tilde{v}}, \tag{3.22}$$

$$= \exp\left[-\frac{1}{2 \cdot f(\tilde{l})^2} \sum_{d=1}^{D} \left(z_{i,d} - z_{j,d} \right)^2 \right] \cdot \frac{\partial f(\tilde{v})}{\partial \tilde{v}}, \tag{3.23}$$

$$\frac{\partial K}{\partial \tilde{l}} = \frac{\partial \left\{ f(\tilde{v}) \cdot \exp\left[-\frac{1}{2 \cdot f(\tilde{l})^2} \sum_{d=1}^{D} \left(z_{i,d} - z_{j,d} \right)^2 \right] + \delta_{ij} f(\tilde{\sigma}^2) \right\}}{\partial \tilde{l}}, \tag{3.24}$$

$$\begin{aligned} &= -\frac{2 \cdot f(\tilde{v})}{f(\tilde{l})} \left(-\frac{1}{2 \cdot f(\tilde{l}^2)} \sum_{d=1}^{D} \left(z_{i,d} - z_{j,d} \right)^2 \right) \\ &\cdot \exp\left[-\frac{1}{2 \cdot f(\tilde{l})^2} \sum_{d=1}^{D} \left(z_{i,d} - z_{j,d} \right)^2 \right] \cdot \frac{\partial f(\tilde{l})}{\partial \tilde{l}}. \end{aligned} \tag{3.25}$$

Where $f(\cdot)$ depicts a transform function, which maps the parameter from the unconstrained space $\mathbb{R}$ to the constrained space $\mathbb{R}_{\geq 0}$. The necessity of moving between spaces arises because most optimizers, e.g., Adam [KB17] operate in an unconstrained space. After the hyperparameters are updated, a reverse transform function is applied to map them back to the constrained space. There exist different functions that fulfill the requirement to map between unconstrained and constrained values. For instance, the libraries, GPflow [MvN+17] and GPyTorch [GPB+18], use the Softplus function

$$f(\tilde{x}) = \text{softplus}(\tilde{x}) = \log(1 + \exp(\tilde{x})) \tag{3.26}$$

where log depicts the natural logarithm and $\tilde{x}$ the unconstrained parameter value. The output of the Softplus function is always positive, as required by all parameters of the squared exponential kernel. Extra care needs to be taken when transforming the noise variance value because it can have values that are close to zero. To ensure the covariance matrix remains numerically stable during Cholesky decomposition, the diagonal needs to be perturbed. Therefore, a shift is added after applying the Softplus function

$$f(\tilde{x}) = \text{softplus}(\tilde{x}) = \log(1 + \exp(\tilde{x})) + \gamma, \tag{3.27}$$

where γ is an offset, here 1e-6. Aside from $f(\cdot)$, the gradients also require the derivative of the Softplus function. The derivative is given by the Sigmoid function

$$f'(\tilde{x}) = \text{sigmoid}(\tilde{x}) = \frac{1}{1 + \exp(-\tilde{x})}, \tag{3.28}$$

which is bounded between 0 and 1. The shift is canceled out as it is a constant value.

Looking at Equation (3.23) and Equation (3.25), it is evident that all of them contain the term $(\star)$ (see Equation (3.29)). Thus this term is computed once and then reused for the subsequent computations and for the covariance matrix K. The resulting equations for the gradients and covariance matrix simplify to

$$(\star) := \left[-\frac{1}{2 \cdot f(\tilde{l}^2)} \sum_{d=1}^{D} \left(z_{i,d} - z_{j,d} \right)^2 \right], \tag{3.29}$$

$$K(z_i, z_j) = f(\tilde{v}) \cdot \exp(\star) + \delta_{ij} \cdot f\left(\tilde{\sigma}^2\right), \tag{3.30}$$

$$\frac{\partial K}{\partial \tilde{v}} = \exp(\star) \cdot \frac{\partial f(\tilde{v})}{\partial \tilde{v}}, \tag{3.31}$$

$$\frac{\partial K}{\partial \tilde{l}} = -\frac{2 \cdot f(\tilde{v})}{f(\tilde{l})} (\star) \cdot \exp(\star) \cdot \frac{\partial f(\tilde{l})}{\partial \tilde{l}}, \tag{3.32}$$

$$\frac{\partial K}{\partial \tilde{\sigma}^2} = \delta_{ij} \cdot \frac{\partial f\left(\tilde{\sigma}^2\right)}{\partial \tilde{\sigma}^2}. \tag{3.33}$$

In the next subsection, the algorithm to update the hyperparameters using gradient descent is presented, including step-by-step instructions and the mathematical expressions needed for the implementation.

3.3.1 Gradient Descent

Gradient descent is an optimization algorithm commonly used in machine learning [ADG+16; WYZ21; Zha19] to minimize a loss function (Equation (3.10)). The main idea behind gradient descent is to iteratively adjust the hyperparameters of a model to find the values that minimize the loss function, here the negative log likelihood. In our case, the parameters of the squared exponential kernel and the noise variance are optimized. The gradients are computed using Equation (3.16). Note that for the second part of the equation instead of computing the matrix-by-matrix products directly, we first compute the vector-by-matrix multiplication. This way, the complexity of the second part becomes $O(N^2)$. In the trace term, only the diagonal terms of the product need to be computed. The hyperparameter update in a gradient descent step is performed as follows

$$\theta_i = \theta_i - \eta \cdot \frac{\partial \ell(\boldsymbol{\theta})}{\partial \theta_i}, \tag{3.34}$$

where η is the step size and $\frac{\partial \ell(\boldsymbol{\theta})}{\partial \theta_i}$ the gradient of the loss function with respect to θ_i. One step of the gradient descent algorithm including the computation of the loss, gradient w.r.t. the hyperparameters, and the update step is outlined in Algorithm 3.5.

Algorithm 3.5 Single step of gradient descent algorithm

```
Require: $Z$: Training input
Require: $\mathbf{y}$: Targets
Require: $\eta$: Step size
Require: $k$: Kernel function
Require: $[l, \nu, \sigma^2]$: Hyperparameters
1: Compute lower triangle part of $K$
2: Compute gradients $\frac{\partial K}{\partial \nu}$ and $\frac{\partial K}{\partial l}$
3: Compute Cholesky decomposition of $K$            {Obtain Cholesky factor $L$}
4: Compute $K^{-1}$        {Solve $LL^T K^{-1} = I$ via forward and backward substitution}
5: Compute $\boldsymbol{\beta} = K^{-1}\mathbf{y}$                    {Matrix-by-vector multiplication }
6: Compute negative log likelihood loss
      • Calculate $\frac{1}{2}\sum_i^N \log(L_{ii}^2)$
      • Calculate $\frac{1}{2}\mathbf{y}^\top \boldsymbol{\beta}$
      • Add normalization constant $\frac{N}{2}\log(2\pi)$
7: Compute $\partial \ell(\boldsymbol{\theta})/\partial \theta_i$
      1. Compute trace of $K^{-1}\frac{\partial K}{\partial \theta_i}$                {Part 1 of Equation (3.16)}
      2. Compute $\boldsymbol{\beta}^T \frac{\partial K}{\partial \theta_i}\boldsymbol{\beta}$                    {Part 2 of Equation (3.16)}
8: Update hyperparameter: $\theta_i = \theta_i - \eta \cdot \frac{\partial \ell(\boldsymbol{\theta})}{\partial \theta_i}$
```

3.3.2 Adam

Using simple gradient descent is not advisable. Especially for non-convex functions, the algorithm may get stuck in a local minima. Thus the solution may be suboptimal. To prevent this, we use Adam [KB17], which is one of the most commonly used optimization algorithms today. The algorithm is a first-order gradient-based optimization of a stochastic objective function. Additionally, Adam is straightforward to implement, computationally efficient, and has small memory requirements. We opted for this algorithm because the reference libraries use this algorithm, thus ensuring a fair comparison between the runtimes of the different libraries. There are four hyperparameters for the Adam algorithm. Namely, the step size η, exponential decay rates for the moment estimates β_1 and β_2, and the term ϵ added to the denominator to improve numerical stability. Using Adam changes line 8 in Algorithm 3.5. The update of one hyperparameter θ_i using Adam is depicted in Algorithm 3.6. For Adam hyperparameters, we used the following default values: $\beta_1 = 0.9, \beta_2 = 0.999$, $\eta = 0.1$, and $\epsilon = 10^{-8}$.

3.4 System identification

GPs serve as a popular alternative to NN in control theory [PDC+14], specifically for identifying and modeling non-linear systems [BS15]. The primary objective in using GPs for SI is to accurately predict the behavior of a system when it is exposed to new, previously unseen control inputs. This prediction is based on analyzing a provided set of input and output time sequences, which

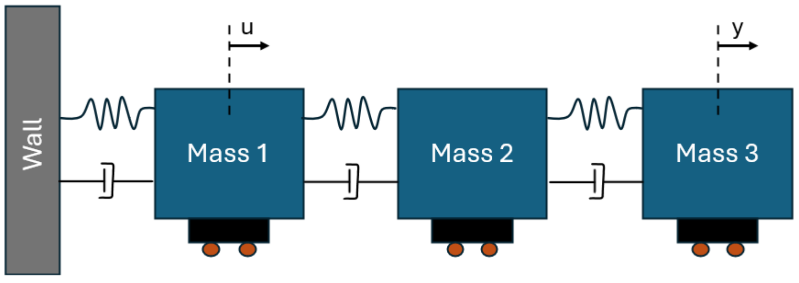
Algorithm 3.6 Adam [KB17] for a single optimization step, as replacement of simple gradient descent in step 8 of Algorithm 3.5. Before the optimization loop is entered, the 1st and 2nd moment, and the time step need to be initialized to 0. β_1^t and β_2^t denote β_1 and β_2 to the power of t.

```
Require: $\frac{\partial \ell(\theta)}{\partial \theta_i}$: Gradient of the loss function (Equation (3.16))
Require: $\eta$: Step size
Require: $t$: Time step
Require: $\theta_i \in \{l, \nu, \sigma^2\}$: Hyperparameter
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $m_{t-1}$: 1st moment
Require: $w_{t-1}$: 2nd moment
1: $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$    {Update biased first moment estimate}
2: $w_t \leftarrow \beta_2 \cdot w_{t-1} + (1 - \beta_2) \cdot g_t^2$    {Update biased second raw moment estimate}
3: $\eta_t = \eta \cdot \sqrt{1 - \beta_2^t}/(1 - \beta_1^t)$    {Compute step size at time step $t$}
4: $\theta_t \leftarrow \theta_{t-1} - \eta_t \cdot m_t/(\sqrt{w_t} + \epsilon)$    {Update parameter}
```

encapsulate the system's past behavior. To achieve this, feature vectors are constructed with so-called regressors. These regressors play a crucial role as they include lagged system states, which are essentially previous states of the system recorded at various time intervals. By incorporating these historical states, GPs can effectively capture the dynamic characteristics of the system [Koc05]. For demonstration purposes, a non-linear finite impulse response model [Sär19] is used. This model only requires the input time sequences as regressors. Consequently, the feature vectors take the form $\mathbf{z}_i = (u_{i-D}, ..., u_i)$, where u_i represents the control inputs and D is the number of regressors.

To illustrate this approach, consider a non-linear system of coupled mass-spring dampers, as depicted in Figure 3.2. Non-linearity is introduced into the system through non-linear force profiles in the

Figure 3.2: Non-linear mass spring damper system as introduced in [RWM20].



springs between the masses. The control input u is the force applied to the initial mass, while the observations y consist of the position of the last mass. For our experiments, we used the data from [RWM20], which consists of $N = 10^5$ training and $M = 5000$ test samples. To enhance the stability of the Cholesky decomposition, the input data is scaled to a consistent range (normalization), and the output data is transformed to have zero mean and unit variance (standardization).

GPs offer several advantages in this context. Firstly, they provide a probabilistic framework, meaning they not only predict the system's output but also give a measure of uncertainty about the prediction. This is particularly useful in control theory, where understanding the confidence in predictions can help in making more informed decisions. Additionally, GPs are non-parametric models [Mur13], which means they do not assume a specific functional form for the relationship between inputs and outputs. This flexibility allows them to model complex, non-linear systems more precisely than some traditional parametric models [IMR24]. By leveraging lagged system states as regressors within feature vectors, GPs can be effectively employed for SI in control theory, enabling the prediction of system behavior for new control inputs and handling the non-linear dynamics of the system.

4 Parallel algorithms

In our task-based implementation, parallel algorithms need to be deployed to fully exploit the benefits of parallelism and concurrency. Parallel algorithms are designed to take advantage of multiple cores in a computing environment to solve problems more efficiently. These algorithms work on a tiled version of the present problem. This means that before computations, e.g., the Cholesky factorization, are performed, the required components, e.g., the covariance matrix, need to be divided into tiles. In general, the computational effort for all involved computations of the GP model for N training and M test samples mainly consists of the following operations, which may occur more than once

- Matrix assembly, e.g., for the covariance, in $O(N^2)$
- Compute Cholesky factor L in $O(N^3)$
- Matrix-by-vector multiplication $\hat{\mathbf{y}} = K_{\hat{Z},Z}\boldsymbol{\alpha}$ in $O(MN)$
- Forward and backward substitution for matrices $LX = K_{Z,\hat{Z}}$ in $O(MN^2)$
- Matrix-by-matrix multiplication $W = V^\top V$ in $O(M^2N)$
- Computation of diagonal elements of a matrix-by-matrix multiplication $\mathrm{diag}(K^{-1}\frac{\partial K}{\partial \theta_i})$ in $O(N^2)$

The four major functions, such as assembly, Cholesky decomposition, optimization, and prediction with uncertainty, of the GP model employ all of the mentioned operations. Nevertheless, each of them has its own nuances regarding implementation and parallelism. Assembly in parallel is quite simple because there are no data dependencies since only read operations are performed on the feature vectors. The only dependency is the intermediate step where the distance (Equation (3.29) on page 31) is computed and then reused by the covariance matrix and cross covariance matrix. Still, many of the computations can be computed efficiently in parallel. Since the covariance matrix K is symmetric, only the lower triangular tiles are assembled. For the transposed cross covariance matrix, the tiles of the cross covariance matrix are reused to avoid duplicate computations. A Cholesky decomposition using the tiled Cholesky algorithm was implemented and rigorously discussed in [SP23]. In addition, the previous work also implemented the forward and backward substitution for the matrix-by-vector multiplication and the prediction without uncertainty.

In the following two sections, additional tiled algorithms for the computation of prediction uncertainty and optimization are discussed. A tile in the r-th row and c-th column of some arbitrary matrix K is denoted as K_{rc}.

4.1 Tiled triangular solve for matrices

The computation of K^{-1}, needed for the gradient computation in Equation (3.16) on page 29, using the computed Cholesky factor L, can be done by solving $LL^TK^{-1} = I$. $I \in \mathbb{R}^{N\times N}$ depicts the identity matrix. First, a forward substitution $LX = I$ with the intermediate matrix $X \in \mathbb{R}^{N\times N}$ is performed. Then X is used as the right-hand side for the backward substitution $L^TK^{-1} = X$. The forward substitution of $LX = I$ is basically an independent triangular vector solve for each column of I, the same logic holds for the backward substitution. Nevertheless, it is more efficient, also memory-wise, to use a triangular matrix solve. The tiled version of the forward and backward substitution algorithms require two BLAS operations [BLKD07; DDHD90]:

- Triangular matrix solve (TRSM):

$$L_{cc} \cdot X_{cr} = I_{cr}, \tag{4.1}$$

- General matrix-matrix multiplication (GEMM):

$$I_{mr} = I_{mr} - L_{mc} \cdot I_{cr}. \tag{4.2}$$

Algorithm 4.1 depicts the pseudo-code for the forward substitution. For the backward substitution, the operations need to be executed on different tiles, as depicted in Algorithm 4.2. Both algorithms work in-place to save storage. The right-hand side, here the identity matrix I, which becomes the inverse of K after executing the two algorithms, consists of T_N^2 tiles. The BLAS operations TRSM and GEMM each have a complexity of $O(\texttt{tile\_dim}^3)$, where `tile_dim` denotes the dimension of a tile T_N.

Algorithm 4.1 In-place tiled forward substitution for matrices

```
Require: $I$: Right hand side
Require: $L$: Cholesky factor
1: for $r = 0$ to $T_N - 1$ do
2:     for $c = 0$ to $T_N - 1$ do
3:         $I_{cr} = \text{TRSM}(L_{cc}, I_{cr})$
4:         for $m = c + 1$ to $T_N - 1$ do
5:             $I_{mr} = \text{GEMM}(L_{mc}, I_{cr}, I_{mr})$
6:         end for
7:     end for
8: end for
```

Algorithm 4.2 In-place tiled backward substitution for matrices

```
Require: $I$: Right hand side
Require: $L^T$: Transposed Cholesky factor
1: for $r = 0$ to $T_N - 1$ do
2:     for $c = T_N - 1$ to $0$ do
3:         $I_{cr} = \text{TRSM}(L_{cc}^\top, I_{cr})$
4:         for $m = c - 1$ to $0$ do
5:             $I_{mr} = \text{GEMM}(L_{cm}^\top, I_{cr}, I_{mr})$
6:         end for
7:     end for
8: end for
```

4.2 Tiled vector-vector dot product for diagonal elements

Terms, like $\text{trace}\left(K^{-1}\frac{\partial K}{\partial \theta_i}\right)$ (Algorithm 3.5 on page 32, line 7.1) and $V^\top V$ (Algorithm 3.4 on page 27, line 2), require only the computation of diagonal elements and not the complete matrix-by-matrix product. Thus, only the dot product of the respective rows and columns needs to be computed. For the dot product, we use the BLAS dot function for vectors [LHKK79]:

- Vector-vector dot product (DOT) for the j-th column of V_{nm}:

$$\mathbf{d}_m[j] = \sum_{i=1}^{N} V_{nm}[i,j] \cdot V_{nm}[i,j]. \tag{4.3}$$

Algorithm 4.3 depicts the pseudo-code for the in-place tiled vector-vector dot product. To save memory and avoid computing the transpose of V, only the matrix V is used as input. This does not affect the calculation as instead of calculating the dot product between the row of $V^\top$ and the column of V, we can directly calculate the dot product of each column of V with itself. Matrix V consists of $T_N \times T_M$ tiles and vector d of T_M tiles. For the case $T_M \neq T_N$, Algorithm 4.3 still works because we first perform the computations for the first column of the matrix V, i.e., for $m = 1$ we consider the tiles $V_{11}, V_{21}, \ldots, V_{n1}$, and store the results in d_1. Then we do the same for $m = 2$ to $m = T_M$.

Since DOT takes the flattened matrix tile V_{nm} as input, which is stored in a row-major format, an additional stride parameter must be passed to cblas_ddot that is called by DOT. In this case, the function call for the j-th column of V_{nm} looks as follows after setting the input parameters: cblas_ddot$(N, \&V_{nm}[j], M, \&V_{nm}[j], M)$. Where N is the number of elements in the column, $\&V_{nm}[j]$ is the pointer to the start of the j-th column in the flattened matrix, and M is the distance between the elements of the column in the flattened matrix. This function is executed for each column j of the flattened matrix tile V_{nm}.

Algorithm 4.3 Tiled vector-vector dot product for diagonal of matrix-matrix product

```
Require: $V$: E.g., Algorithm 3.2 on page 26, step 1
Require: $\mathbf{d}$: Empty vector for diagonal elements
1: for $m = 0$ to $T_M - 1$ do
2:     for $n = 0$ to $T_N - 1$ do
3:         $\mathbf{d}_m = \text{DOT}(V_{nm}, \mathbf{d}_m)$
4:     end for
5: end for
```

5 Software framework

In this chapter, we present the three main software tools that we use. These are the C++ standard library for concurrency and parallelism HPX [KDL+20], MKL [Int24], and pybind11 [JRM17]. HPX supports application scalability and high parallel efficiency, while pybind11 facilitates seamless integration between C++ and Python. Optimized algebraic routines, enhancing the computational capabilities of our framework, are offered by MKL.

5.1 HPX

HPX [KDL+20], a C++ standard library developed by the STE||AR Group and initially released in 2008, addresses concurrency and parallelism in software development, with a predominant focus on scientific computing. The HPX API adheres to the C++11/14/17/20 ISO standards and follows the programming guidelines of the Boost C++ libraries. HPX ensures portability across NVIDIA, AMD, and Intel CPUs [DBK23]. In addition, HPX facilitates the execution of local and remote operations, enabling it to run on distributed systems [KHA+14]. Communication between distant nodes is managed via active messages [Wal82], referred to as “parcels” in HPX [KHA+14], making communication independent of the communication platform, e.g., MPI. This flexibility allows HPX to utilize other portable communication platforms like LCI [YKS23]. Using parcels to encapsulate remote calls and to improve the overlap between communication and computation leads to a reduction in communication costs [DBK24]. Additionally, HPX implements different schedulers, such as work stealing [BL99], to reduce overheads, like synchronization. In work stealing, tasks are distributed among multiple worker threads, each maintaining its own queue of tasks. As each worker processes tasks from its queue, a worker that exhausts its tasks becomes idle. To achieve load balancing, the idle worker then steals tasks from the queues of other workers that still have pending tasks [TDH+18].

A standout feature of HPX is its ability to enable the writing of fully asynchronous code, capable of utilizing hundreds of millions of lightweight threads. This is achieved through HPX futures, which allow developers to write fully asynchronous code without global synchronization barriers, unlike message-based execution models like MPI. HPX also simplifies managing concurrency through its dataflow and future-based synchronization mechanisms, helping developers to handle complex parallel and distributed computing tasks. For runtime measurement, the APEX library [HPC+15] can be installed alongside HPX to time and count annotated function calls.

In our implementation, we mainly utilize the following three HPX components: `hpx::shared_future()`, `hpx::async()`, and `hpx::dataflow()`. The `hpx::shared_future()` is a synchronization primitive in HPX, similar to `std::shared_future()` in the C++ Standard Library. Unlike `hpx::future()`, which can only be moved or accessed by a single thread and whose data becomes inaccessible once the future has returned [BKK+19], `hpx::shared_future()` allows multiple threads to access the result concurrently. The HPX runtime manages the lifetime of the shared future and ensures that the return value remains available for each dependent task. To retrieve the result of the asynchronous operation, the `get()` method can be called multiple times by different threads. If the future is not ready yet, the `get()` method will block the execution of the calling thread until the result is available.

Building on the capabilities of `hpx::shared_future()`, the `hpx::async()` function is a core component of HPX that facilitates asynchronous task execution. It enables functions to be executed in parallel, enabling developers to exploit concurrency. The function `hpx::async()` creates a task that encapsulates the function and its arguments. This task is then submitted to an executor, which then queues it for execution, schedules it, and assigns it to a thread from its pool. After execution, the function returns a future holding the result of the function call.

Complementing `hpx::shared_future()` and `hpx::async()`, the `hpx::dataflow()` construct in HPX provides a powerful abstraction for expressing dependencies between asynchronous operations. It chains asynchronous tasks together, ensuring they are executed in the correct order based on their dependencies. Similar to `hpx::async()`, `hpx::dataflow()` returns a future holding the result of the function call. In addition, `hpx::dataflow()` does not start the thread until all futures, passed as arguments to the function, are ready. This can delay the operation. In cases where no exceptions are thrown during the calculation of the futures that are passed to the function, `hpx::unwrapping` can be called [KDL+20]. This function calls `.get()` to replace each `hpx::future` object in the argument list with its future result type before passing it to the function run by `hpx::dataflow` [DBK24].

5.2 MKL

MKL [Int24] is a highly optimized library offering a comprehensive set of linear algebra routines for maximum performance on modern Intel architecture. Being part of Intel oneAPI, MKL is designed for scientific computing, engineering, financial analytics, and machine learning applications. MKL's standout feature is its extensive collection of highly optimized routines, e.g., for linear algebra and vector math, which leverage multi-threading, vectorization, and cache optimization. This allows for significant performance gains without manual code optimization. Furthermore, MKL provides fast and accurate routines for solving, e.g., linear equations and matrix factorizations. In addition, MKL enhances usability through integration with popular development environments and tools. It supports C, C++, Fortran, and Python, and is compatible with major compilers, allowing easy integration into existing codebases. MKL achieves parallelization through the use of OpenMP, which allows calculations to be distributed across multiple CPU cores. By default, MKL is threaded, e.g., it automatically parallelizes tasks unless otherwise specified. This threading behavior is controlled by environment variables such as `OMP_NUM_THREADS`, which controls the number of OpenMP threads for all programs, and `MKL_NUM_THREADS`, which specifically sets the number of threads for MKL routines. Additionally, MKL utilizes Single Instruction, Multiple Data (SIMD), found in modern CPUs, so that a single operation can be executed simultaneously on multiple data points [DYB23].

PyTorch offers various linear algebra routines, such as the Cholesky decomposition, which can be called via the corresponding high-level Python function, e.g., `torch.linalg.cholesky()`. These functions then call the corresponding C++ functions via Python bindings, which can be found in the Aten library. Aten is a PyTorch tensor library, "on top of which almost all other Python and C++

interfaces in PyTorch are built".[1] The called C++ functions perform all necessary input validations and then call optimized routines from libraries like MKL for the numerical calculations.[2] The results from these MKL calls are then passed back to the C++ layer and finally back to the Python environment. Similarly, when using TensorFlow with MKL linked, high-level Python functions like `tf.linalg.matmul` serve as a user interface for linear algebra operations. These functions call into a generated bindings module `gen_linalg_ops.py`, which manages input types and shapes. The bindings then invoke corresponding C++ implementations found in TensorFlow's core, where the actual logic for the operations is defined. If MKL support is enabled, the C++ implementation calls optimized MKL functions to perform the computation efficiently.[3] The results of these MKL calls are returned to the C++ layer, which then passes them back through the bindings module to the Python environment. Similar to MKL, parallelism and threading can be controlled in PyTorch and TensorFlow through the use of OpenMP. Therefore, the environment variables `OMP_NUM_THREADS` and `MKL_NUM_THREADS` can be set in both libraries as well. In our work, we use sequential MKL and parallelize the calculations with HPX.

5.3 pybind11

pybind11 [JRM17] is a lightweight and versatile library that enables seamless integration between C++ and Python, allowing developers to expose C++ code to Python and vice versa. It is particularly useful for high performance projects, leveraging C++'s strengths while maintaining Python's ease of use and flexibility. Being fully compatible with modern C++ standards, pybind11 supports C++11/14/17/20, allowing the use of advanced C++ features in bindings.

The design of pybind11 aims to minimize overhead, while preserving the performance of C++ code when called from Python, making it ideal for numerical simulations and real-time data processing. It offers a straightforward syntax for binding C++ code to Python, allowing developers to wrap C++ classes and functions with minimal boilerplate. In addition, pybind11 provides automatic conversion between C++ and Python data types, which simplifies data exchange between the two languages.

However, pybind11 requires explicit instantiation of the templates to be exposed to Python, as Python does not support C++ templates. Therefore, each specific version of an instantiated function template must be defined separately with concrete types.[4] For example, `template <type T> T add(T a, T b);` must be bound explicitly to the types that will be used in Python, such as int or double. If the wrong C++ type is provided when calling templated functions from Python, pybind11 will throw an error, preventing runtime issues. The library also supports function overloading via `py::overload_cast`, which only requires the specification of parameter types.[5] Last but not least, pybind11 integrates easily with existing build systems and works seamlessly with popular tools like CMake and setuptools, enabling smooth integration into existing C++ and Python projects.

[1] https://pytorch.org/cppdocs/#aten

[2] https://github.com/pytorch/pytorch/blob/main/aten/src/ATen/native/mkl

[3] https://github.com/tensorflow/tensorflow/tree/master/tensorflow/core/kernels/mkl

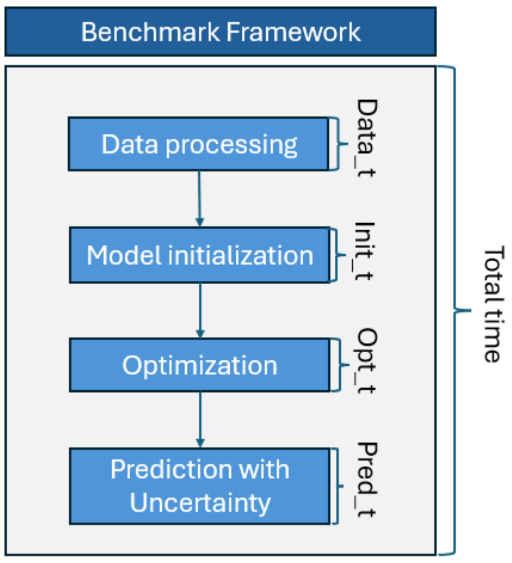
[4] https://pybind11.readthedocs.io/en/stable/advanced/functions.html#binding-functions-with-template-parameters

[5] https://pybind11.readthedocs.io/en/stable/classes.html#overloaded-methods

6 Concepts and implementation

This chapter consists of two parts. First, an introduction to the GPPPy project structure and implementation[1] is provided. We will discuss the integration of the implemented C++ code into Python using pybind11 (Section 5.3 on page 41), with special attention given to how the HPX runtime can be seamlessly started from within the Python runtime. Second, we present the implementation of the benchmark framework as depicted in Figure 6.1 for each of the three libraries, namely, GPflow, GPyTorch, and GPPPy. The main components of the framework include data loading, model initialization, optimization, and prediction with uncertainty. For the C++ and Python versions of the GPPPy library, the implementation of the framework code can be found in Appendix A.3 on page 84. The aim of this benchmark framework is to ensure that all three libraries provide the same functionality. This consistency and comparability between the three libraries is crucial for the later experiments in Chapter 7.

Figure 6.1: Benchmark framework



[1] https://github.com/MHelmann/GPPPy_hpx/tree/grad_option_3/hpx_py11

6.1 GPPPy project

As mentioned in Chapter 5, a pybind11 project can be set up with either CMake or setuptools. We decided to use CMake because it allows us to build a C++ project that works independently of pybind11. The Python library can be then generated by wrapping the written C++ code. An additional benefit is that parts of `CMakeLists.txt` can be reused for the setup of the pybind11 project. The complete structure of our pybind11 project, called Gaussian Processes Parallel in Python (GPPPy), is depicted in Figure 6.2.

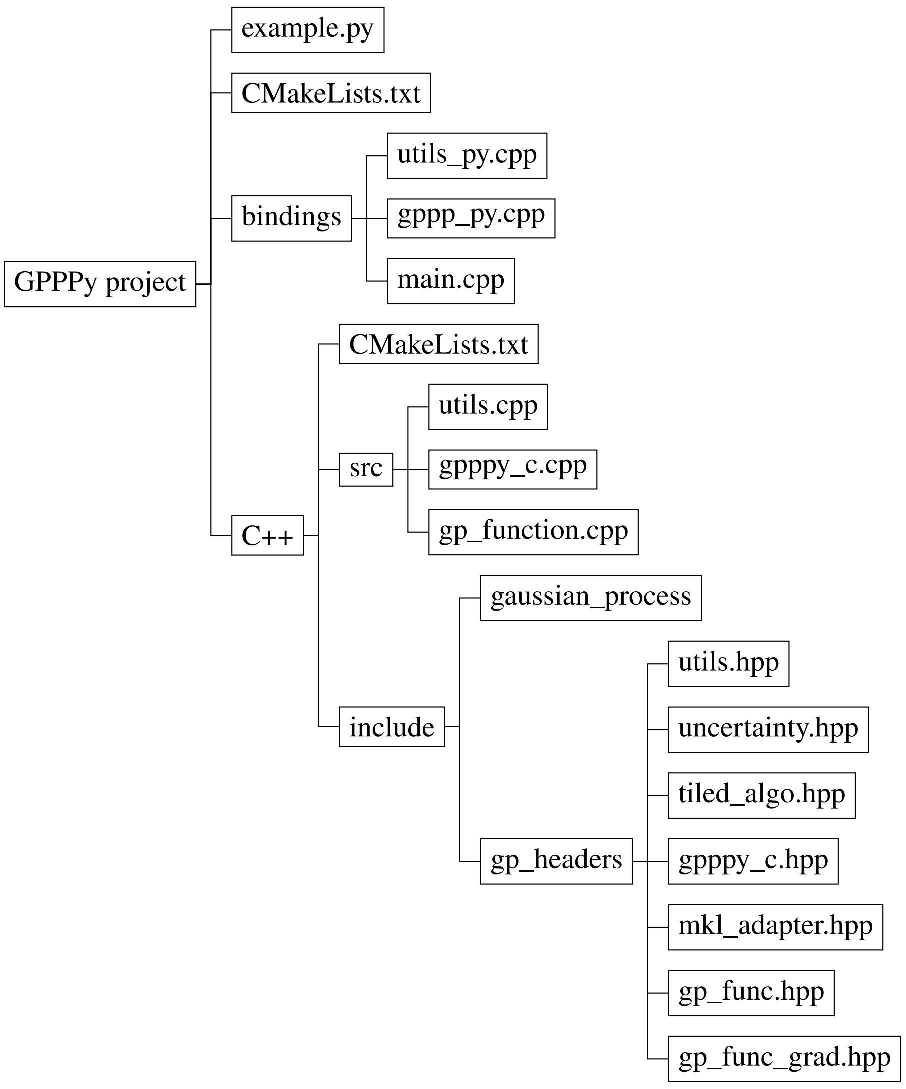


Figure 6.2: GPPPy project structure

The GPPPy project is organized into several key components. At the top level, the project includes directories for C++ and pybind11 code, along with configuration and example files such as `CMakeLists.txt` and `example.py`.

6.1.1 Source code

The C++ directory in Figure 6.2 contains two main subdirectories: `include` and `src`. The `include` subdirectory contains the file `gaussian_process` and the subdirectory `gp_headers`. The former is a guard file to prevent multiple inclusions of the same header file in the compilation process. While the latter houses various header files, like `gp_func_grad.hpp`, `gp_func.hpp`, `mkl_adapter.hpp`, `gpppy_c.hpp`, `tiled_algo.hpp`, and `uncertainty.hpp`. The `src` subdirectory includes implementation files such as `gp_function.cpp`, `gpppy_c.cpp`, and `utils.cpp`. Additionally, a `CMakeLists.txt` file is present in the C++ directory for build configuration.

The program is designed in an object-orientated manner. `gpppy_c.hpp` contains the class definition of the GP object, which has some attributes, such as kernel parameters, number of training tiles, size of a training tile, training input and output, and a vector of bools for trainable parameters of the kernel. Furthermore, it possesses the predict with uncertainty function, optimize function, and compute loss function. Those functions are wrappers for functions that use the HPX syntax to run in the HPX runtime. Before providing an example of such a function and how it is executed, we first describe how the HPX runtime can be started and stopped without using the standard approach of explicitly implementing `hpx_main()`. For the explicit initialization of the HPX runtime, the `hpx_init.hpp` header is typically included. This approach allows the runtime to be started via an explicit main-thread function such as `hpx_main()`, which is defined at the global scope. Within this function, all HPX API functions like futures, async, or parallel algorithms can be utilized. In contrast, our approach allows the combination of HPX with other runtimes, which aligns with our final goal of being able to start and stop the HPX runtime from the Python environment. To initialize the HPX runtime, the function `start_hpx_runtime()` needs to be called before any HPX related computations are executed. Once all HPX functions have finished their execution the function `stop_hpx_runtime()` is called. Internally, this executes `hpx::finalize()` to terminate the runtime. This function, called from one locality, notifies all connected localities to finish execution and returns only after all have exited, signaling that the runtime should stop. The finalize function must be called from the HPX runtime, meaning it needs to be scheduled using `hpx::post()` when called from a non-HPX thread. Following `hpx::finalize()`, the `hpx::stop()` function should be the last HPX function called on every locality, blocking and waiting for the locality to finish before returning to the caller. Note, this function is used only if the runtime was started with `hpx::start()`.

Sometimes, only a small part of the computations are executed in the HPX runtime. In such scenarios, it is beneficial to pause the HPX runtime until the next relevant computation, and then resume it. HPX offers two functions for this: `hpx::suspend()` and `hpx::resume()`. These functions can only be used when running HPX on a single locality. `hpx::suspend()` is a blocking call that waits for all currently scheduled tasks to complete before putting the thread pool OS threads to sleep. "This function can only be called when the runtime is running, or already suspended in which case this function will do nothing".[2] `hpx::resume()` reactivates the sleeping threads, making them ready to process new tasks. "This function can only be called when the runtime suspended, or already running in which case this function will do nothing".[3]

[2] https://hpx-docs.stellar-group.org/latest/html/libs/full/init_runtime/api/hpx_suspend.html#_CPPv4N3hpx7suspendER10error_code

[3] https://hpx-docs.stellar-group.org/latest/html/libs/full/init_runtime/api/hpx_suspend.html#_CPPv4N3hpx6resumeER10error_code

Listing 6.1 Functions to start, resume, suspend, and stop HPX runtime

```cpp
// Initialize HPX, don't run hpx_main
void start_hpx_runtime(int argc, char **argv)
{
    hpx::start(nullptr, argc, argv);
}

// Resume HPX runtime
void resume_hpx_runtime()
{
    hpx::resume();
}

// Wait for all tasks to finish, and suspend the HPX runtime
void suspend_hpx_runtime()
{
    hpx::suspend();
}

// hpx::finalize has to be called from the HPX runtime before hpx::stop
void stop_hpx_runtime()
{
    hpx::post([]()
    { hpx::finalize(); });
    hpx::stop();
}
```

Listing 6.2 Optimize function

```cpp
std::vector<double> GP::optimize(const gpppy_hyper::Hyperparameters &hyperparams)
{
    std::vector<double> losses;
    hpx::threads::run_as_hpx_thread([this, &losses, &hyperparams]()
    {
        losses = optimize_hpx(_training_input, _training_output, _n_tile_size,
        _n_tiles, lengthscale, vertical_lengthscale, noise_variance,
        n_regressors, hyperparams, trainable_params).get();
    });
    return losses;
}
```

Once the runtime is started, functions such as `predict()` or `optimize()` can be executed. Listing 6.2 depicts an example implementation of such a function, here the `optimize()` function. In short, the goal of `optimize()` is to nudge the parameters in a direction that minimizes the negative log likelihood (Equation (3.11) on page 25). The primary focus here is on the function body, in particular on `hpx::threads::run_as_hpx_thread()`. By utilizing this function call, the `optimize_hpx()` function gets scheduled on the HPX runtime. Note that `run_as_hpx_thread()` blocks until completion. `optimize_hpx()` executes the complete Algorithm 3.5 on page 32 and Algorithm 3.6 on page 33, which are implemented with the HPX syntax.

The reason why `run_as_hpx_thread()` needs to be called instead of directly calling `hpx::async()` is the following: After calling `hpx::start()`, the main thread is not automatically an HPX thread. Thus, `hpx::run_as_hpx_thread()` ensures that the function runs as an HPX thread, maintaining the necessary runtime context and blocking until the function completes. This ensures proper synchronization and execution order. When working with the HPX library, it's crucial to follow specific patterns for initializing, executing tasks, and finalizing the HPX runtime. Adhering to these patterns ensures the HPX runtime system functions correctly and efficiently. For detailed options and examples, please refer to the HPX documentation.[4]

The file `gp_func.hpp` contains declarations for functions using HPX syntax, such as `optimize_hpx()` and `predict_hpx()`, which are called from `gpppy_c.cpp`. These functions implement the complete logic described in Chapter 3, including the assembly of various tiled covariance matrices, Cholesky decomposition, triangular system solutions, and loss computation. Depending on the intermediate steps of these operations, they are either outsourced to corresponding files like `gp_func_grad.hpp` or executed using the tiled algorithms in `tiled_algo.hpp`. The file `gp_func_grad.hpp` declares functions related to hyperparameter optimization as discussed in Section 3.3 on page 27. `mkl_adapter.hpp` file provides an interface for integrating MKL functions into the project. In addition, `tiled_algo.hpp` contains declarations for tiled algorithms that enable efficient computation for large matrices. For the computation of prediction uncertainty in GPs, `uncertainty.hpp` defines the relevant functions. Finally, `utils.hpp` declares various utility functions used throughout the project, such as `compute_train_tile_size()`, `compute_test_tiles()`, `load_data()`, `start_hpx_runtime()`, and `stop_hpx_runtime()`.

To build the `gaussian_process` project as a standalone C++ executable, use CMake version 3.16 or higher and set the C++ standard to 17. Additionally, link the program against HPX version 1.9.1 and MKL version 2024.1 packages. Include the directories for HPX and MKL headers, and specify the source files located in the `src` directory. Define the header files to be installed from the `include/gp_headers` directory. Then, create a shared library named `gaussian_process` using the specified source files and link it with the HPX and MKL libraries. Finally, install the built library and header files. Listing A.3 in Appendix A.3 on page 84 provides a complete working example in C++, which is also used for the experiments.

6.1.2 Binding

The `bindings` directory in Figure 6.2 contains source files such as `main.cpp`, `gppp_py.cpp`, and `utils_py.cpp`, which enable the integration of C++ code with Python using pybind11. The purpose of the `main.cpp` file is to create a Python module named `gaussian_process` using the pybind11 library as depicted in Listing 6.3. This module serves as an interface for Python to interact with the C++ code. The `PYBIND11_MODULE` macro (line 11) defines the module and initializes it with functions and classes defined in the C++ code. Specifically, the functions `init_gpppy()` and `init_utils()` are called, which register the components of the GP library into the Python module. This functionality is facilitated through the header file `pybind11.h` which is the core header of pybind11. It includes the primary functionality needed to create bindings, e.g., the definition of modules, classes, and functions as well as the management of the interaction between Python and C++.

[4] https://hpx-docs.stellar-group.org/latest/html/manual/starting_the_hpx_runtime.html

Listing 6.3 Python binding for GPPPy using pybind11

```cpp
// Include the pybind11 header
#include <pybind11/pybind11.h> // PYBIND11_MODULE

// Create a namespace alias for pybind11
namespace py = pybind11;
// Forward declarations of binding functions
void init_gpppy(py::module &);
void init_utils(py::module &);

// Define the module named "gaussian_process" and its initialization function
PYBIND11_MODULE(gaussian_process, m)
{
    // Set the module documentation string
    m.doc() = "Gaussian Process library";
    // Initialize the submodules
    init_gpppy(m);
    init_utils(m);
}
```

The `gpppy_py.cpp` file contains the bindings for the GP C++ class and hyperparameters struct. Listing 6.4 provides an excerpt that highlights the main binding code. The first step is to import the necessary headers from the C++ codebase that contain the functionality such as classes and functions, which should be exposed to Python later. Second, pybind11 relevant headers are imported that include utilities for automatically converting between C++ Standard Template Library (STL) types like `std::vector<T>`, and their corresponding Python types. That way the user can easily pass complex data structures between C++ and Python without writing custom conversion code. Next, the function `init_gpppy()` is created to initialize the Python module. This function gets forward declared in `main.cpp`. Within this function, the GP class from the C++ library is exposed to Python by defining its constructor and methods using pybind11's `.def()` and `.def_readwrite()` functions. `py::init()` is a convenience function that takes the types of a constructor's parameters as template arguments and wraps the corresponding constructor. The arguments for the constructor are specified using `py::arg()`. Default parameters can be set as well, e.g., see lines 15 – 23 in Listing 6.4. The `.def_readwrite()` function is used to expose public member variables of a C++ class to Python. Moreover, these variables can be accessed and modified directly from the Python side, providing a way to read and write the values of these variables. All in all, this allows Python users to instantiate the GP class, access and modify its attributes like parameters of the kernel function, retrieve data, and call methods such as `optimize()`.

The `utils_py.cpp` file provides functions such as `print()`, `start_hpx_wrapper()`, and `stop_hpx()`. Listing 6.5 depicts an excerpt of this file with the focus on `print()`, `start_hpx_wrapper()`, and `stop_hpx()`. The GPPPy `print()` function is designed to print the whole or a part of a `std::vector<double>` with user-defined start and end indices and a separator. In contrast, the Python `print()` function is a general function that can print various objects, e.g., doubles or lists with an optional separator and end parameter. Future work can therefore focus on generalizing the GPPPy `print()` function or exploring ways to use the original Python function as a wrapper.

Listing 6.4 GP class binding of GPPPy using pybind11

```
#include "../cpp_code/include/gp_headers/gpppy_c.hpp"
#include "../cpp_code/include/gp_headers/gp_functions.hpp"
#include <pybind11/stl.h>
#include <pybind11/pybind11.h>

namespace py = pybind11;

void init_gpppy(py::module &m)
{
    ...

    py::class_<gpppy::GP>(m, "GP")
    .def(py::init<std::vector<double>, std::vector<double>, int, int, double,
     double, double, int, std::vector<bool>>(),
    py::arg("input_data"),
    py::arg("output_data"),
    py::arg("n_tiles"),
    py::arg("n_tile_size"),
    py::arg("lengthscale") = 1.0,
    py::arg("v_lengthscale") = 1.0,
    py::arg("noise_var") = 0.1,
    py::arg("n_reg") = 100,
    py::arg("trainable") = std::vector<bool>{true, true, true})
    .def_readwrite("lengthscale", &gpppy::GP::lengthscale)
    .def("get_input_data", &gpppy::GP::get_training_input)
    .def("optimize", &gpppy::GP::optimize, py::arg("hyperparams"))
    ...
}
```

In Section 6.1.1, the mechanism behind the start and stop of the HPX runtime was introduced in a pure C++ environment. Now, we will present how this can be realized from within a Python environment. Technically, the steps needed to start the HPX runtime are similar. The only addition is the wrapper function. This function takes a vector of strings representing command-line arguments and an integer indicating the number of cores. It appends the argument specifying the number of threads to the argument vector, converts the vector of strings to an array of C-style strings, and then starts the HPX runtime with these arguments. This enables the HPX system to be configured and launched with the desired number of OS threads. To stop the HPX runtime, we define a binding in line 32 (Listing 6.5) that exposes the `stop_hpx_runtime()` function from the `utils` namespace to Python. This allows Python code to call the `stop_hpx()` function, which in turn will execute the `stop_hpx_runtime()` function in the underlying C++ code, enabling the termination of the HPX runtime from Python. For the functions `resume_hpx_runtime()` and `suspend_hpx_runtime()`, similar bindings as for `stop_hpx_runtime()` are implemented. These three functions do not require additional wrappers as they do not have input parameters.

To generate an executable that can be imported into Python, the `gaussian_process` project with Python language bindings needs to be built. The build requires CMake version 3.16 or higher, C++17, and pybind11 version 2.10.3. If not found, then the pybind11 package will be fetched and built from source. Furthermore, the project depends on HPX and MKL packages, which must also be available. Include the directories for HPX, MKL, and the project headers from `cpp_code/include/gp_headers`. Also, include the directories for Python-related headers in `python_code`. Then, create a Python

Listing 6.5 Utils binding for GPPPy using pybind11

```cpp
#include "../cpp_code/include/gp_headers/utils_c.hpp"

void start_hpx_wrapper(std::vector<std::string> args, std::size_t n_cores)
{
    // Add the --hpx:threads argument to the args vector
    args.push_back("--hpx:threads=" + std::to_string(n_cores));

    // Convert std::vector<std::string> to char* array
    std::vector<char *> argv;
    for (auto &arg : args)
    argv.push_back(&arg[0]);
    argv.push_back(nullptr);
    int argc = args.size();
    utils::start_hpx_runtime(argc, argv.data());
}

void init_utils(py::module &m)
{
    ... // Other functions such as compute_train_tile_size(), etc.
    m.def("print", &utils::print,
    py::arg("vec"),
    py::arg("start") = 0,
    py::arg("end") = -1,
    py::arg("separator") = " ", "Print elements of a vector with optional
                                  start, end, and separator parameters");

    // Using the wrapper function
    m.def("start_hpx", &start_hpx_wrapper, py::arg("args"), py::arg("n_cores"));
    m.def("resume_hpx", &utils::resume_hpx_runtime);
    m.def("suspend_hpx", &utils::suspend_hpx_runtime);
    m.def("stop_hpx", &utils::stop_hpx_runtime);
}
```

module named `gaussian_process` using pybind11, linking it with HPX and MKL libraries. Finally, install the built Python module. Listing A.4 in Appendix A.3 on page 84 provides a complete working example of GPPPy, which is also used for the experiments.

6.2 Reference implementations

In the following, the reference implementations for GPflow and GPyTorch, that are used for the experiments, are presented. First, the data loading procedure is discussed that is similar in both cases. Second, the respective implementations and some additional remarks are provided for the two libraries.

6.2.1 Data loading

Data loading differs between GPPPy and the reference libraries, GPflow and GPyTorch. While GPPPy only loads the data into a vector and computes the feature matrix needed for, e.g., the covariance matrix, on the fly, GPflow and GPyTorch need the whole feature matrix Z as input for the model initialization step. The data loading procedure and feature matrix generation for the reference libraries, presented in Appendix A.2 on page 84, involves two functions. The first function `generate_feature_matrix()` creates a feature matrix by padding the original input array with zeros on the left for (`n_regressors`-1) positions and then rolls the array to create a window of size `n_regressors`. The second function `load_data()` reads training and testing input and output data from specified file paths, truncates them to the desired sizes, and generates feature matrices for both the training and testing input data sets using the `generate_feature_matrix()` function. Finally, it reshapes the output data into column vectors and returns the feature matrices and target values for both training and testing data sets. The current implementation loads both training and test data simultaneously, which means that if new test data needs to be loaded, the training data will also be reloaded. This approach is not very efficient if executed more than once. However, in our experiments, we load the data only once, so this implementation suffices. The procedure can be easily modified to load the training and test data separately.

6.2.2 GPflow framework

This section presents the remaining components required for the GPflow reference implementation, as they differ in their implementation between the libraries.

Model initialization Listing 6.6 depicts model initialization in GPflow.[5] The `init_GPflow_model` function sets up a GP regression model using GPflow, with customizable parameters for the kernel vertical-scale (`k_v`), kernel length-scale (`k_lscale`), and noise variance (`noise_var`). This function takes as inputs the feature matrix X and the target vector Y (line 1) and constructs a GP regression model with a squared exponential covariance function (lines 15 – 19). The initialized model is then ready for training or making predictions. During model initialization, no computations like the inversion of K are performed.

Optimization Hyperparameter optimization is important so that the GP model accurately captures the underlying structure of the data (Section 3.3 on page 27). The Python code in Listing 6.7 defines a function `optimize_model()` designed to perform hyperparameter optimization of a GPflow model. The function takes in a GPflow model and the number of optimization steps as arguments. It utilizes the Adam optimizer, configured with specific hyperparameters such as learning rate, beta values, and epsilon (lines 12 – 13). The optimization process is encapsulated in a TensorFlow function, which ensures efficient execution by compiling the operations into a static computation graph. Within each optimization step, a gradient tape is used to compute the loss of the model, which is then differentiated to obtain gradients with respect to the model's trainable variables (lines 17 – 22). These gradients are subsequently applied to update the model parameters (line 24). The

[5] https://gpflow.github.io/GPflow/2.9.0/notebooks/getting_started/basic_usage.html

Listing 6.6 Model initialization in GPflow

```python
def init_GPflow_model(X, Y, k_var=1.0, k_lscale=1.0,  noise_var=0.1):
    """
    Initialize a Gaussian process regression model using GPflow.

    Args:
    X (numpy.ndarray): The input data matrix.
    Y (numpy.ndarray): The target data vector.
    k_v (float): Vertical-scale parameter of the kernel. Defaults to 1.0.
    k_lscale (float): Length-scale parameter of the kernel. Defaults to 1.0.
    noise_var (float): Noise variance parameter. Defaults to 0.1.

    Returns:
    gpflow.models.GPR: Initialize Gaussian process regression model.
    """
    model = gpflow.models.GPR((X, Y),
        kernel=gpflow.kernels.SquaredExponential(
        variance=k_v, lengthscales=k_lscale, ),
        noise_variance=noise_var,
    )

    return model
```

optimization step is iteratively executed for the specified number of iterations to minimize the model's loss, thus improving the model's performance. Future work can extend this by adding the epsilon criterion to allow early stopping.

Prediction with Uncertainty After the parameters are optimized, the model can be used to make predictions with uncertainties. For GPflow this can be done using `predict_with_var` in Listing 6.8. The function accepts two arguments: a trained GPflow model and a set of test input data in the form of a NumPy array (line 1). Within the function, the model's `predict_f` method is called with the test input data, yielding the mean (`f_pred`) and variance (`f_var`) of the latent function values for the test data (line 13). These predictions, encapsulating both the expected values and the associated uncertainties, are then returned as NumPy arrays. If the full posterior covariance matrix is to be calculated, then the parameter `full_cov` in `model.predict_f()` must be set to `True`.

Parallelization To achieve optimal performance and efficiency for TensorFlow on CPUs, it is essential to configure the number of threads used for parallelization. TensorFlow provides the `set_intra_op_parallelism_threads(config["N_CORES"])` function to set the number of intra-operation threads for parallelism, which determines the number of threads employed for operations that can be parallelized within a single operation, such as matrix multiplication. Additionally, the `set_inter_op_parallelism_threads(config["N_CORES"])` function configures the number of inter-operation parallelism threads, dictating how many threads are used to parallelize multiple operations across the computation graph. By specifying the `config["N_CORES"]` variable, we can optimize computational efficiency by tailoring the number of cores used to the available hardware.

Listing 6.7 Hyperparameter optimization in GPflow

```python
def optimize_model(model, opt_iter):
    """
    Optimize the parameters of the given GPflow model.

    Args:
    model (gpflow.models.GPModel): The GPflow model whose parameters needs to be optimized.
    opt_iter (int): Number of optimization steps

    Returns:
    None
    """
    opt = tf.keras.optimizers.Adam(learning_rate=0.1, beta_1=0.9,
                                   beta_2=0.999, epsilon=1e-08)

    @tf.function
    def optimization_step():
        with tf.GradientTape() as tape:
            # Compute the loss inside the tape context
            loss = model.training_loss()
        # Compute the gradients of the loss with respect to the model's
        # trainable variables
        gradients = tape.gradient(loss, model.trainable_variables)
        # Apply the gradients to update the model's trainable variables
        opt.apply_gradients(zip(gradients, model.trainable_variables))

    # Run the optimization step
    for i in range(opt_iter):
        optimization_step()

    return None
```

Listing 6.8 Prediction with uncertainty in GPflow

```python
def predict_with_var(model, X_test):
    """
    Predict latent function values for the given test data.

    Args:
    model (gpflow.models.GPModel): The trained GPflow model.
    X_test (numpy.ndarray): The test input data.

    Returns:
    f_pred (numpy.ndarray): Mean of latent function values for test data.
    f_var (numpy.ndarray): Variance of latent function values for test data.
    """
    f_pred, f_var = model.predict_f(X_test)

    return f_pred, f_var
```

Similarly, the `MKL_NUM_THREADS` environment variable sets the number of threads for MKL to use for parallel processing. OpenMP is used to enhance parallel computation tasks, with the `OMP_NUM_THREADS` environment variable specifying the number of threads used for OpenMP computations.[6] Additionally, CPU affinity settings, managed by `OMP_PLACES=threads` and `OMP_PROC_BIND=close`, determine how workloads are distributed across CPU cores. Furthermore, TensorFlow uses Intel oneAPI Deep Neural Network Library (oneDNN) on the CPU, leveraging OpenMP settings as environment variables to enhance performance for processes such as backpropagation. For versions above 2.9, oneDNN is the default library.

6.2.3 GPyTorch framework

In the following, similar to Section 6.2.2, the remaining components of the GPyTorch reference implementation are presented below.

Model initialization In GPyTorch, model initialization differs from GPflow. The code in Listing 6.9 illustrates the initialization and definition of the GP model for regression using GPyTorch.[7] The class `ExactGPModel` inherits from `gpytorch.models.ExactGP` and is designed to handle exact inference in GP regression tasks (line 1). The constructor `__init__()` initializes the model with training inputs X, training targets Y, and a specified likelihood function (lines 10 – 11). Within the constructor the mean function is set to a constant mean, `gpytorch.means.ConstantMean()`, and the covariance function is defined using a scaled RBF kernel, aka a squared exponential kernel, `gpytorch.kernels.ScaleKernel(gpytorch.kernels.RBFKernel())` (lines 12 – 13). The hyperparameters of the covariance module are initialized to default values, with the base kernel's length-scale set to 1.0 and the output scale also set to 1.0 (lines 15 – 16). These hyperparameters can be subsequently optimized during the training process to better fit the data. The `forward()` method implements the forward pass through the model, computing the mean and covariance of the input data X (lines 29 – 30). It returns a multivariate normal distribution `gpytorch.distributions.MultivariateNormal()` parameterized by the mean and covariance functions, representing the GP posterior over the outputs corresponding to the inputs X.

Optimization The training procedure for the GP regression model using the GPyTorch library in Python is encompassed in the `train()` function. Inputs to the function are the GP model, likelihood, training input data X, training target data Y, and an optional parameter `training_iter` specifying the number of training iterations (line 1). The function begins by setting the model and likelihood to training mode (lines 15 – 16). Like in the other libraries, the Adam optimizer is used to update the model parameters, including those of the Gaussian Likelihood (line 19). The loss function used is the LML, which is negated since the goal is to minimize the loss (line 22). In each iteration, the gradients from the previous step are first set to zero, after which the model generates output predictions from the input data X, and then the LML is calculated based on the model output and the target data Y (lines 26 – 30). Next, the computed loss is backpropagated to compute gradients for all parameters, which are then updated by the optimizer (lines 32 – 34).

[6] https://www.intel.com/content/www/us/en/developer/articles/guide/guide-to-tensorflow-runtime-optimizations-for-cpu.html

[7] https://docs.gpytorch.ai/en/v1.6.0/examples/01_Exact_GPs/Simple_GP_Regression.html

Listing 6.9 Model initialization in GPyTorch

```python
class ExactGPModel(gpytorch.models.ExactGP):
    """
    This class defines the exact Gaussian Process model for regression.

    Args:
    X (torch.Tensor): The training input data.
    Y (torch.Tensor): The training target data.
    likelihood: The likelihood function for the model.
    """
    def __init__(self, X, Y, likelihood):
        super(ExactGPModel, self).__init__(X, Y, likelihood)
        self.mean_mod = gpytorch.means.ConstantMean()
        self.covar_mod = gpytorch.kernels.ScaleKernel(gpytorch.kernels.RBFKernel())

        self.covar_mod.base_kernel.lengthscale = 1.0
        self.covar_mod.outputscale = 1.0

    def forward(self, X):
        """
        Forward pass through the model.

        Args:
        X (torch.Tensor): Input data.

        Returns:
        gpytorch.distributions.MultivariateNormal: Multivariate normal
        distribution over the output.
        """
        mean_x = self.mean_module(X)
        covar_x = self.covar_module(X)
        return gpytorch.distributions.MultivariateNormal(mean_x, covar_x)
```

Prediction with Uncertainty Predictions with uncertainty can be made by calling the function `predict_with_var()`. The function accepts a trained GP model, a Gaussian likelihood, and test input data as its parameters (line 1). By setting both the `model` and `likelihood` to evaluation mode, the latent function of `model` computes the mean (`f_mean`) and variance (`f_var`) at the test points (lines 14 – 20). These predictions are generated within a "no-gradient" context in which the gradient computations are deactivated. This reduces the memory consumption for these computations.

One limitation of GPyTorch is that while the variance can be accessed directly, the implementation first computes the full posterior covariance matrix and then extracts the diagonal elements. The full posterior covariance matrix is also stored. One possible reason for this design is the LanczOs Variance Estimates (LOVE) algorithm [PGWW18], which calculates the posterior covariance matrix much faster than the classical method. However, LOVE does not compute the posterior covariance matrix using the closed form (Equation (3.8) on page 24) but through rapid approximation of the predictive covariance matrix, with the computed variances accurate to within four decimals. As a result, efficient calculation of only the diagonal of the posterior covariance matrix was probably not a priority. This also implies that if LOVE is not used, uncertainty can only be obtained by fully computing the posterior covariance matrix.

Listing 6.10 Hyperparameter optimization in GPyTorch

```python
def train(model, likelihood, X_train, Y_train, training_iter=10):
    """
    Optimize the hyperparameters of the Gaussian process regression model.

    Args:
    model (gpytorch.models.ExactGP): The Gaussian process regression model.
    likelihood (gpytorch.likelihoods.GaussianLikelihood): The likelihood function.
    X (torch.Tensor): The training input data.
    Y (torch.Tensor): The training target data.
    training_iter (int, optional): Number of training iterations.

    Returns:
    None
    """
    model.train()
    likelihood.train()

    # Use adam optimizer; model.paremeters() include GaussianLikelihood parameters
    optimizer = torch.optim.Adam(model.parameters(), lr=0.1)

    # "Loss" for GPs - the log marginal likelihood
    lml = gpytorch.mlls.ExactMarginalLogLikelihood(likelihood, model)

    for i in range(training_iter):
        # Clear gradients from previous iteration
        optimizer.zero_grad()
        # Output from model
        output = model(X)
        # Calc loss and backprop gradients
        loss = -(lml(output, Y))
        # Compute dloss/dtheta_i for every parameter theta_i which has requires_grad=True
        loss.backward()
        # Update the value of theta_i using the gradient theta_i.grad
        optimizer.step()

    return None
```

Parallelization The number of threads in PyTorch needs to be set according to available hardware to facilitate a thorough performance and efficiency comparison on CPUs. This can be done using the `set_num_threads(config["N_CORES"])` function in PyTorch, which specifies the number of threads that PyTorch will utilize for parallel operations. By setting the `config["N_CORES"]` variable, we can control the number of threads employed by PyTorch to perform operations that benefit from parallelization, such as matrix multiplications. In addition to setting the number of threads for general parallel operations, PyTorch offers the `set_num_interop_threads(config["N_CORES"])` function. This function configures the number of threads used for inter-operation parallelism, managing how many threads are allocated for the parallel execution of multiple operations.

Listing 6.11 Prediction with uncertainty in GPyTorch

```python
def predict_with_var(model, likelihood, X_test):
    """
    Predict the mean and variance of latent function values.

    Args:
    model (gpytorch.models.ExactGP): The trained Gaussian process regression model.
    likelihood (gpytorch.likelihoods.GaussianLikelihood): The likelihood function.
    X_test (torch.Tensor): The test input data.

    Returns:
    - f_mean (torch.Tensor): Mean of latent function values.
    - f_var (torch.Tensor): Variance of latent function values.
    """
    model.eval()
    likelihood.eval()

    with torch.no_grad():
        f_pred = model(X_test)
        f_mean = f_pred.mean
        f_var = f_pred.variance

    return f_mean, f_var
```

OpenMP enhances performance for parallel computation tasks. The `OMP_NUM_THREADS` variable sets the number of threads for OpenMP computations.[8] CPU affinity settings, controlled by `OMP_PLACES=threads` and `OMP_PROC_BIND=close`, dictate workload distribution across cores, affecting communication overhead and cache efficiency. Similarly, the `MKL_NUM_THREADS` environment variable[9] sets the number of threads for MKL to use for parallel processing.

However, it is important to note that according to the PyTorch documentation `set_num_threads` always takes precedence over environment variables for intra-op parallelism settings.[9]. Among the environment variables, the variable `MKL_NUM_THREADS` takes precedence over `OMP_NUM_THREADS`.[9]

[8] https://pytorch.org/tutorials/recipes/recipes/tuning_guide.html#intel-openmp-runtime-library-libiomp
[9] https://pytorch.org/docs/stable/notes/cpu_threading_torchscript_inference.html

7 Results

In this chapter, we evaluate the performance and efficiency of GPPPy compared to the reference implementations on CPUs, using the benchmark framework introduced in Chapter 6. Section 7.1 describes the experimental environment. Next, Section 7.2 presents the evaluation of the results. The latter first describes the measured components. Then, the influence of different tile sizes on the performance for different numbers of cores is discussed. This is followed by the strong scaling results for GPPPy, HPX, GPflow, and GPyTorch. For a clearer distinction between the C++ and Python versions of GPPPy, the C++ version of GPPPy is referred to as HPX. After that we address parallel efficiency. Finally, GPPPy is compared against GPyTorch used with LOVE.

7.1 Experimental environment

Table 7.1 depicts the system information, providing a detailed overview of the hardware and software configurations used in our experiments. This includes the specifications of the CPU, RAM, and operating system as well as all relevant software tools, such as GCC.

The GPPPy, GPyTorch, and GPflow implementation are done using C++ and Python. All the programming languages, libraries, and frameworks used in this thesis are open source and can be used for free. The details of these components are listed below:

- **Programming Language**: Python (V 3.8.10)
- **Libraries**: NumPy (V 1.24.3), GPyTorch (V 1.12), GPflow (V 2.9.2)
- **Deep Learning Framework**: PyTorch (V 2.3.1), TensorFlow (V 2.10.1)

CPU	2× AMD EPYC 7742
Cores	2×64
Base clock rate	2.25 GHz
L3 cache size	512 MB
RAM	2 TB DDR4
Operating system	Ubuntu 20.04.6 LTS
Compiler	GCC 9.4.0, optimization level 3
Project build tool	CMake 3.18.2
Runtime system	HPX 1.9.1
Math Kernel Library	oneMKL 2024.1
Resource management system	Simple Linux Utility for Resource Management (SLURM) [YJG03]

Table 7.1: System information

7.2 Results evaluation

This section covers the evaluation findings. Figure 6.1 on page 43 depicts the framework that we use to evaluate our implementation against the reference libraries. Four different components are measured for the evaluation. First comes the data processing step, described in Section 6.2.1 on page 51 for GPflow and GPyTorch. Solely for GPPPy the data is loaded from a file. Second, the model initialization is measured. Then, the time taken to perform hyperparameter optimization is measured. After that the time to compute predictions with uncertainty is measured. Since GPyTorch computes only the full posterior covariance matrix (Algorithm 3.4 on page 27), we will proceed in the same way for all libraries to ensure a meaningful comparison. Finally, the total runtime for each framework is provided. However, in a real scenario, the total time is not particularly significant, as optimization may require several iterations and is therefore performed offline. Predictions, on the other hand, need to be accessed quickly and are faster to compute, which is why they are calculated online. In addition, predictions are usually computed multiple times for different test data sets.

We first investigate the performance of HPX for a different number of cores and number of tiles. Then, strong scaling experiments for optimization, prediction with uncertainty, and total runtime are presented and the corresponding findings are discussed. Afterward, we examine the parallel efficiency of the strong scaling benchmark for prediction with uncertainty. In the last section, we talk about the comparison of GPPPy, which only computes the prediction with uncertainty, and GPyTorch used with LOVE. All runtimes in this section are averaged over 10 runs. Error bars indicate the 95% confidence intervals.

7.2.1 Tile scaling

In the following, tile scaling experiments for optimization and prediction with uncertainty of the task-based implementation (HPX) for different numbers of cores are presented. In both experiments, the problem size is $N = 20000$ training samples and $M = 5000$ test samples, with all computations performed in double precision for the following system specifications as depicted in Table 7.1.

Tile scaling for optimization Figure 7.1 depicts the optimization runtime on a logarithmic scale (y-axis) of the task-based implementation (HPX) for different tile sizes (x-axis) with varying core counts: 16, 32, 64, and 128. For tile sizes $T \leq 4$, all core configurations exhibit similar runtime behaviors, with no significant advantage for higher core counts. For $T \geq 8$, the runtime for 16 cores starts to increase until $T \leq 500$. The lowest runtime is achieved for 64 cores and 25 tiles per dimension. For 128 cores, a similar result is achieved; however, the runtime increases abruptly for $T > 25$. The runtime for 32 cores behaves similarly to that of 64 cores, although it is slightly higher for $T = [25, 50]$. Then for $T \leq 100$ the runtime for 32 cores is similar to the one for 64 cores.

The deterioration of runtime for 16 cores for $T \geq 25$ is due to the scheduling effort exceeding the computational advantages. Additionally, task queuing delay becomes a factor, as more tiles than cores result in some tasks having to wait until a core becomes available. This queuing delay adds to the total optimization runtime. The lowest runtime for 64 cores and 25 tiles per dimension indicates an optimal balance between computation and scheduling. The abrupt increase in runtime for 128 cores after $T = 25$ can be attributed to scheduling overhead.

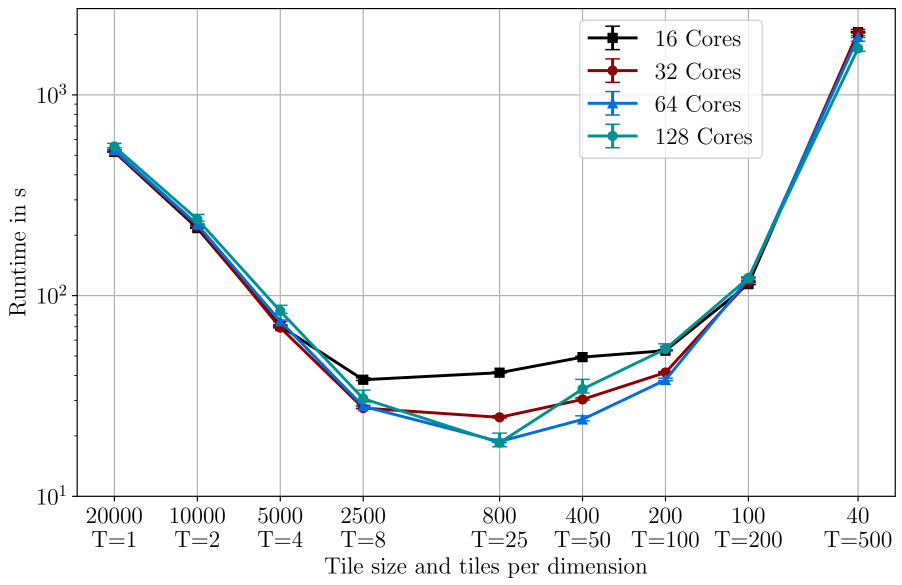


Figure 7.1: Tile scaling comparison for optimization of task-based implementation (HPX) for a different number of cores. Workload size is $N = 20000$ training and $M = 5000$ test samples. All computations are performed in double precision. Hardware is a dual socket AMD EPYC 7742 with 128 cores.

Tile scaling for prediction with uncertainty Figure 7.2 depicts the prediction with uncertainty runtime on a logarithmic scale (y-axis) of the task-based implementation (HPX) for different tile sizes (x-axis) with varying core counts: 16, 32, 64, and 128. For tile sizes $T \leq 8$, all core configurations exhibit similar runtime behaviors, with no significant advantage for higher core counts. For $T = 25$, the runtime for 16 cores no longer decreases. From here on, it starts to increase until $T \leq 500$. The lowest runtime is achieved for 64 cores and 25 tiles per dimension. For 128 cores, the runtime is slightly higher; however, the runtime increases abruptly for $T > 25$. The runtime for 32 cores behaves similarly to that of 64 cores, although it is slightly higher for $T = [25, 50, 100]$.

The deterioration of runtime for 16 cores for $T \geq 25$ is due to the scheduling effort exceeding the computational advantages. Additionally, task queuing delay becomes a factor, as more tiles than cores result in some tasks having to wait until a core becomes available. This queuing delay adds to the total prediction with uncertainty runtime. The lowest runtime for 64 cores and 25 tiles per dimension indicates an optimal balance between computation and scheduling. The abrupt increase in runtime for 128 cores after $T = 25$ can be attributed to scheduling overhead.

Overall, using 25 tiles per dimension provides the best runtime performance for optimization and prediction with uncertainty for almost all cores. Therefore, the number of tiles per dimension is set to 25 for all subsequent experiments.

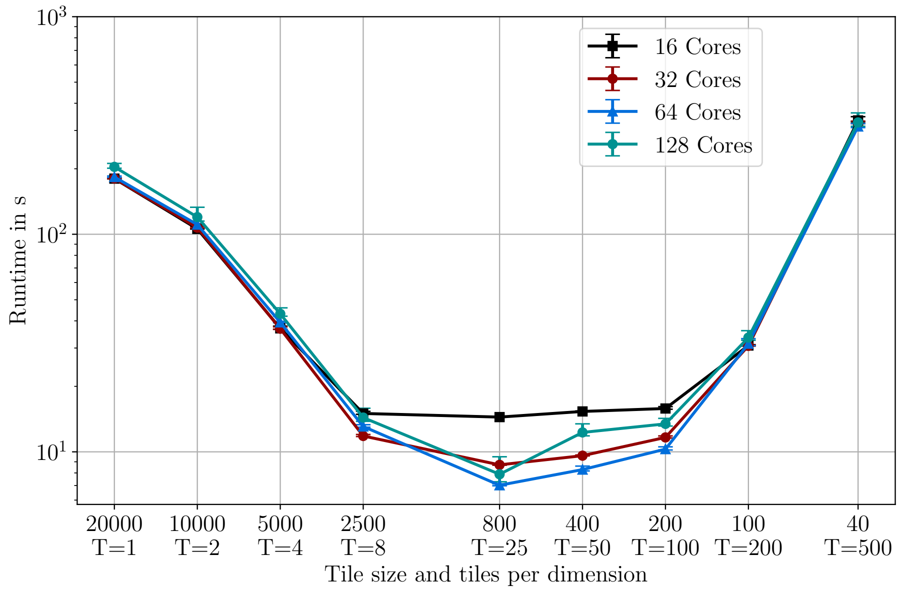


Figure 7.2: Tile scaling comparison for prediction with uncertainty of the task-based implementation (HPX) for a different number of cores. Workload size is $N = 20000$ training and $M = 5000$ test samples. All computations are performed in double precision. Hardware is a dual socket AMD EPYC 7742 with 128 cores.

7.2.2 Strong scaling comparison

Optimization runtime Figure 7.3 depicts a strong scaling benchmark for the optimization step of the GPPPy, HPX, GPflow, and GPyTorch implementations. The y-axis shows the runtime on a logarithmic scale and x-axis the different number of cores. The problem size is $N = 10000$ training samples and $M = 5000$ test samples, with all computations performed in double precision for the following system specifications as depicted in Table 7.1. GPflow has the highest runtime for all cores. Next, both GPPPy and HPX demonstrate a similar strong scaling trend, with a consistent reduction in runtime up to 64 cores. After this point, their runtimes increase for 128 cores. Furthermore, we observe no runtime overhead between HPX and GPPPy, whereby the latter uses pybind11 to expose HPX to Python (Section 6.1.2 on page 47). The dashed gray line represents the ideal scaling for GPPPy and HPX. The best optimization runtime across all cores is achieved by GPyTorch. This implementation shows a speedup of factor 10.5 on 128 cores for this problem size. While GPPPy optimization runtime is slower than that of GPyTorch, it achieves a significant speedup of factor 19.5 on 64 cores.

Firstly, none of the implementations achieve ideal linear scaling, which is expected due to factors such as communication overhead. Secondly, we expected the optimization performance of GPflow to be comparable to that of GPyTorch. However, the difference between the two is clearly visible. Unfortunately, we were not able to determine the exact reason for GPflow's inferior performance. Upon closer inspection of the GPflow code, we observed that GPflow relies solely on TensorFlow functions that internally build the computation graph for backpropagation. In contrast, GPyTorch

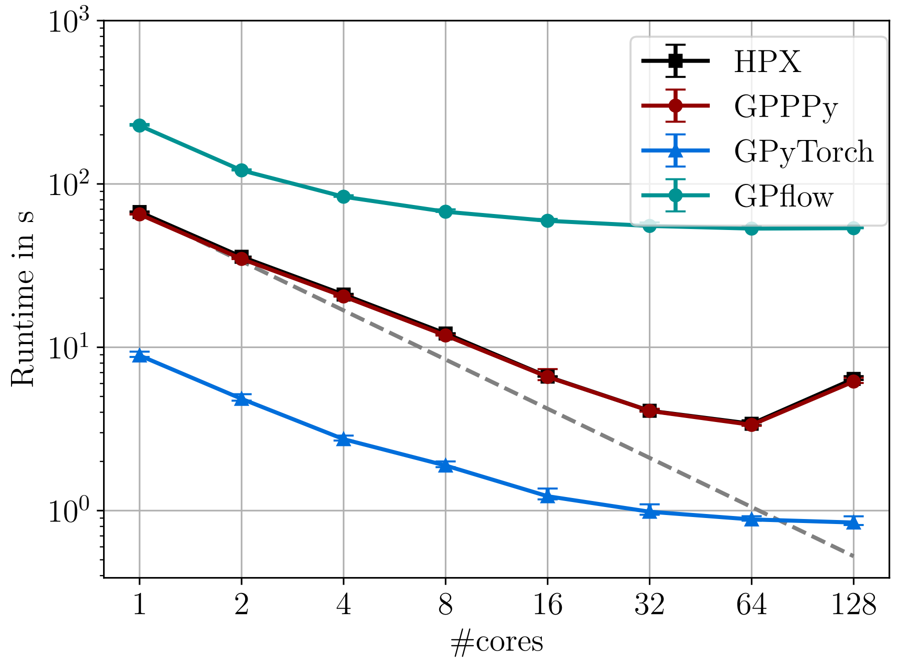


Figure 7.3: Strong scaling optimization runtime comparison between the GPPPy, HPX, GPflow, and GPyTorch implementations on a dual socket AMD EPYC 7742 with 128 cores. For GPPPy and HPX the tile size was set to 25×25 for a workload size of $N = 10000$ training and $M = 5000$ test samples. All implementations use double precision.

uses wrapper functions[1] for PyTorch operations, which include the corresponding forward and backward definitions utilized during the forward pass of the model and backpropagation in the optimization step. This additional customization in GPyTorch might contribute to its superior performance and scalability. The lower performance of GPPPy and HPX is related to the calculation of the gradients, which uses the closed-form solution (Equation (3.16) on page 29). This contrasts GPyTorch's approach where PyTorch accumulates gradients through backpropagation.[2]

The performance scaling of GPPPy and HPX on up to 128 cores is constrained due to the communication overhead, which is related to the hardware design of the two AMD EPYC 7742 CPUs with 64 cores each. Each CPU consists of eight compute dies, with eight cores each. Two adjacent dies, forming a group of 16 cores, share a single NUMA (Non-Uniform Memory Access) domain that manages memory access for these cores. This results in four NUMA domains per CPU. Communication within the 16 cores of a single NUMA domain is efficient due to the proximity of the memory, facilitating good scaling. However, if the communication goes beyond 16 cores, i.e., between two NUMA domains, latency is introduced due to, e.g., additional data routing, thus reducing efficiency. The overhead becomes even more apparent when communication between the two CPUs is required, as this is done over PCIe, using AMD Infinity Fabric. This communication usually has a higher latency compared to intra-CPU communication, which further degrades

[1] https://github.com/cornellius-gp/linear_operator

[2] https://pytorch.org/tutorials/beginner/introyt/autogradyt_tutorial.html#advanced-topic-more-autograd-detail-and-the-high-level-api

performance. Although the system technically has eight NUMA domains, only a single artificial domain combining all eight is currently used, resulting in suboptimal communication. To improve performance, explicit communication between NUMA domains can be implemented, exploiting the inherent locality within the architecture.

The absence of runtime overhead can be attributed to several factors. First, zero-copy interoperability between C++ and Python is supported by pybind11. This allows direct access and manipulation of the same underlying data structures and thus significantly reduces the overhead typically associated with data transfer and conversion. Second, since pybind11 is a lightweight and highly efficient library, it generates minimal overhead for calls between C++ and Python.

Prediction with uncertainty runtime Figure 7.4 depicts a strong scaling benchmark for the prediction with uncertainty step of the GPPPy, HPX, GPflow, and GPyTorch implementations. The y-axis shows the runtime on a logarithmic scale and x-axis the different number of cores. The problem size is $N = 10000$ training samples and $M = 5000$ test samples, with all computations performed in double precision for the following system specifications as depicted in Table 7.1. The results show that there is again no runtime overhead between HPX and GPPPy, whereby the latter uses pybind11 to expose HPX to Python (Section 6.1.2 on page 47). Furthermore, GPPPy and HPX demonstrate significant scaling efficiency with decreasing runtime as the number of cores increases, closely following the ideal scaling line represented by the dashed grey line. Both GPPPy and HPX start with runtimes of around 54 seconds on one core. As the number of cores increases, they reach approximately 2.7 seconds for 32 and 64 cores, achieving a speedup of factor 20.2. However, the runtime increases again to about 4 seconds for 128 cores. In contrast, GPflow exhibits less efficient scaling. Beyond 4 cores, no significant runtime decrease is observable. GPflow begins with a runtime of about 60 seconds at one core and reduces to around 43 seconds at 128 cores, resulting in a speedup factor of 1.4. The highest runtime is observed for GPyTorch at one core, averaging 510 seconds. The runtime decreases continuously and overtakes GPflow when the number of cores is greater than 16. For 128 cores, GPyTorch reaches about 18 seconds with a speedup factor of 28.3. Furthermore, we observe that GPPPy and HPX achieve a speedup of up to 10.5 times over GPyTorch and GPflow.

The poor scaling of GPflow can be attributed to its implementation[3] and TensorFlow. The implementation first solves $LV = K_{Z,\hat{Z}}$ (line 1 in Algorithm 3.2 on page 26). Subsequently, the matrix-by-matrix product $V^\top V$ is computed using `tf.linalg.matmul()`, which is then subtracted from the prior covariance matrix $K_{\hat{Z},\hat{Z}}$, resulting in the full covariance matrix. The computation of the predictions employs V and performs the backward substitution `tf.linalg.triangular_solve(`$L^\top$`, `V`, lower=False)`, where the result is subsequently multiplied with the training output vector $\mathbf{y}$. The bottleneck of this implementation arises from the additional computation of the backward substitution for $L^\top X = V$, which scales with $O(MN^2)$. In contrast, a more efficient approach involves first computing the forward and backward substitution for L and $\mathbf{y}$ that scale with $O(N^2)$. Followed by the cross covariance matrix-by-vector product that scales with $O(MN)$, thereby bypassing the additional $O(MN^2)$ term. Consequently, GPflow's approach results in a higher runtime for large data sets. For 8 cores or less, the runtime plateaus because not all TensorFlow

[3] https://github.com/GPflow/GPflow/blob/master/gpflow/conditionals/util.py#L128

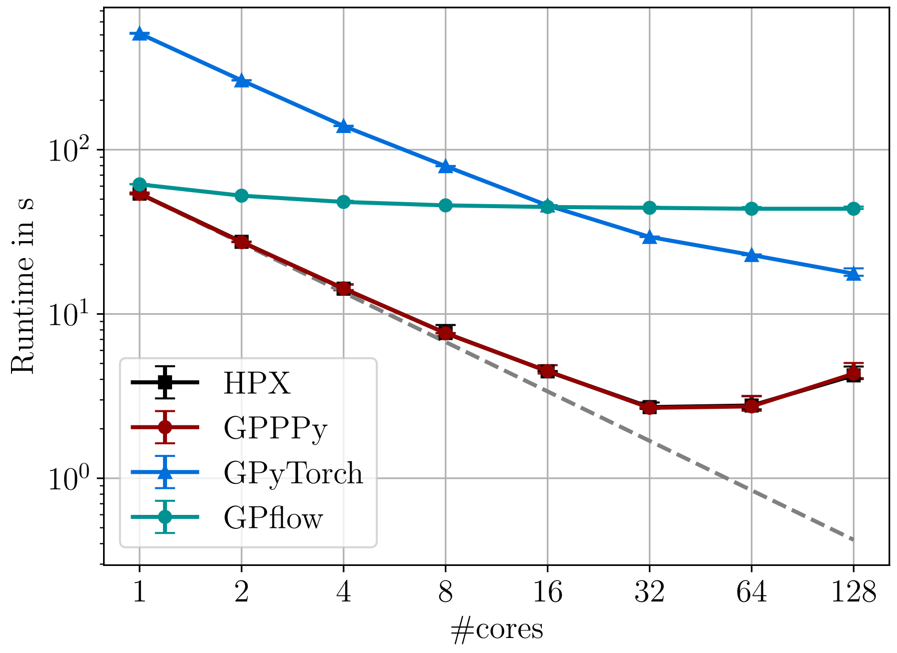


Figure 7.4: Strong scaling prediction with uncertainty runtime comparison between the GPPPy, HPX, GPflow, and GPyTorch implementations on a dual socket AMD EPYC 7742 with 128 cores. For GPPPy and HPX the tile size was set to 25×25 for a workload size of $N = 10000$ training and $M = 5000$ test samples. All implementations use double precision.

functions support parallel computation, despite TensorFlow's parallel processing capabilities.[4] For example, `tf.linalg.cholesky()` and `tf.linalg.triangular_solve()` are not parallelized, whereas `tf.linalg.matmul()` is. This observation is supported by the runtime curve in Figure 7.4, where the runtime decreases until the step with `tf.linalg.matmul()` is executed very efficiently so that the sequential Cholesky decomposition and the triangular solution dominate the runtime.

While GPyTorch shows better scaling than GPflow, its runtime is higher until the number of cores equals 16. GPyTorch achieves better scaling because functions such as `torch.cholesky()`, `torch.linalg.solve_triangular()`, and `torch.addmm()` can be parallelized. Higher runtime can be traced back to GPyTorch's computation of the prediction uncertainty[5] and predictions.[6] First, it solves $KV = K_{Z,\hat{Z}}$, depending on the setting either through Cholesky decomposition followed by subsequent forward and backward substitution or via a direct solve. Then, `torch.addmm(`$K_{\hat{Z},\hat{Z}}, K_{\hat{Z},Z}, V$`)` is called to compute the full covariance matrix. Compared to GPPPy, HPX, and GPflow, this implementation does not reuse the result of the forward substitution (line 1 in Algorithm 3.4 on page 27) and therefore includes an additional term (backward substitution) that scales with $O(MN^2)$. Moreover,

[4] https://www.intel.com/content/www/us/en/developer/articles/guide/guide-to-tensorflow-runtime-optimizations-for-cpu.html

[5] https://github.com/cornellius-gp/gpytorch/blob/develop/gpytorch/models/exact_prediction_strategies.py#L382

[6] https://github.com/cornellius-gp/gpytorch/blob/develop/gpytorch/models/exact_prediction_strategies.py#L325

none of the computed terms are reused for the predictions because GPyTorch does not employ `torch.addmv()` for prediction computation, which requires converting large matrices to dense forms. While this approach may be more memory efficient, especially for large-scale problems, it also means that the intermediate results cannot be reused in the same way as they could be if `torch.addmv()` was employed. This can lead to additional computational overhead due to the repeated matrix-vector multiplications, impacting the overall runtime efficiency despite the better scaling characteristics of GPyTorch. HPX and GPPPy both follow the ideal scaling line, indicating efficient parallelization and load balancing.

The reason for the constrained performance scaling of GPPPy and HPX on up to 128 cores and the reason for the absence of a runtime overhead between GPPPy and HPX, are both identical to those given in the experiment for the optimization (Section 7.2.2).

Total runtime Figure 7.5 depicts a strong scaling benchmark for the total runtime of the GPPPy, HPX, GPflow, and GPyTorch implementation. The y-axis shows the runtime on a logarithmic scale and x-axis the different number of cores. The problem size is $N = 10000$ training samples and $M = 5000$ test samples, with all computations performed in double precision for the following system specifications as depicted in Table 7.1. The total runtime is made up of all the steps depicted in Figure 6.1 on page 43. GPPPy and HPX have lower total runtimes across all cores compared to GPflow and GPyTorch. In addition, we observe no runtime overhead between HPX and GPPPy, whereby the latter uses pybind11 to expose HPX to Python (Section 6.1.2 on page 47). Both implementations scale well up to 64 cores with a speedup factor of 18.6, after that their runtime increases for 128 cores. In contrast, GPflow and GPyTorch show relatively higher runtimes. GPflow's runtime decreases only marginally with increasing cores, suggesting limited parallelization benefits. These observations mirror what we observed for the optimization and predictions with uncertainty runtime benchmarks, as these two are the dominant parts. In addition, we run a simulation for HPX with one core, where all of the settings remain the same, except that the number of tiles is set to one. The observed runtime is higher than the simulation for one core with 25 tiles.

Whether the simulation for one core with 25 tiles is faster than with one tile depends on the data size. With one tile, the entire covariance matrix K is calculated, which significantly impacts the assembly time. For a data size of $N = 10000$, this takes twice as long as the simulation with 25 tiles, where only the lower part of K is assembled. However, the runtimes for Cholesky decomposition and forward and backward substitution increase for 25 tiles due to added overhead. This difference is even more pronounced with 100 tiles. Thus, if the data size becomes significantly larger, to the point where the time for the Cholesky decomposition and related processes outweighs the assembly time, the simulation with one tile (sequential) becomes more efficient than with multiple tiles. This is because the overhead for simulations with multiple tiles on one core eventually increases with a very large amount of data.

The reason for the constrained performance scaling of GPPPy and HPX on up to 128 cores and the reason for the absence of a runtime overhead between GPPPy and HPX, are both identical to those given in the experiment for the optimization (Section 7.2.2).

For higher data size we expect the total runtime of GPPPy and HPX to be higher than the one of GPyTorch because of the less efficient optimization. This will most likely revert to the current behavior once backpropagation as in PyTorch is implemented.

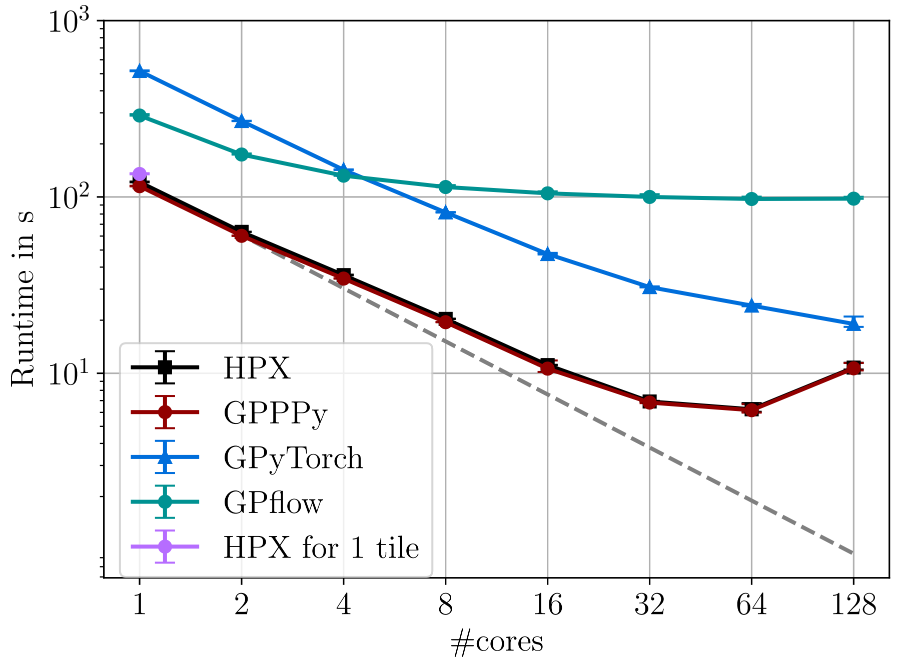


Figure 7.5: Strong scaling total runtime comparison between the GPPPy, HPX, GPflow, and GPyTorch implementations on a dual socket AMD EPYC 7742 with 128 cores. For GPPPy and HPX the tile size was set to 25×25 for a workload size of $N = 10000$ training and $M = 5000$ test samples. All implementations use double precision.

7.2.3 Parallel efficiency

Figure 7.6 depicts the parallel efficiency of the strong scaling benchmark in Figure 7.4 for the GPPPy, HPX, GPflow, and GPyTorch implementation. The y-axis shows the parallel efficiency in % and x-axis the different number of cores. The dashed gray line depict the ideal parallel efficiency. The parallel efficiency of GPPPy and HPX implementations is higher than that of GPyTorch and GPflow. In addition, it is nearly optimal for up to two cores. HPX and GPPPy maintain a high efficiency of over 80% up to 8 cores, while GPflow's efficiency declines sharply after one core, dropping below 20% at 8 cores. For GPPPy, HPX, and GPyTorch the efficiency drops significantly from 8 to 128 cores. Furthermore, only after 64 cores does GPyTorch have better parallel efficiency, before that it maintains relatively high efficiency but is still lower than HPX and GPPPy up to 64 cores. The drop in GPflow's efficiency can be attributed to the limited parallelization capabilities. The decrease in parallel efficiency of GPPPy and HPX can be attributed to the constrained performance scaling on up to 128 cores due to the communication overhead, which is related to the hardware design of the two AMD EPYC 7742 CPUs with 64 cores each. The complete reasoning is provided in the experiment for the optimization (Section 7.2.2).

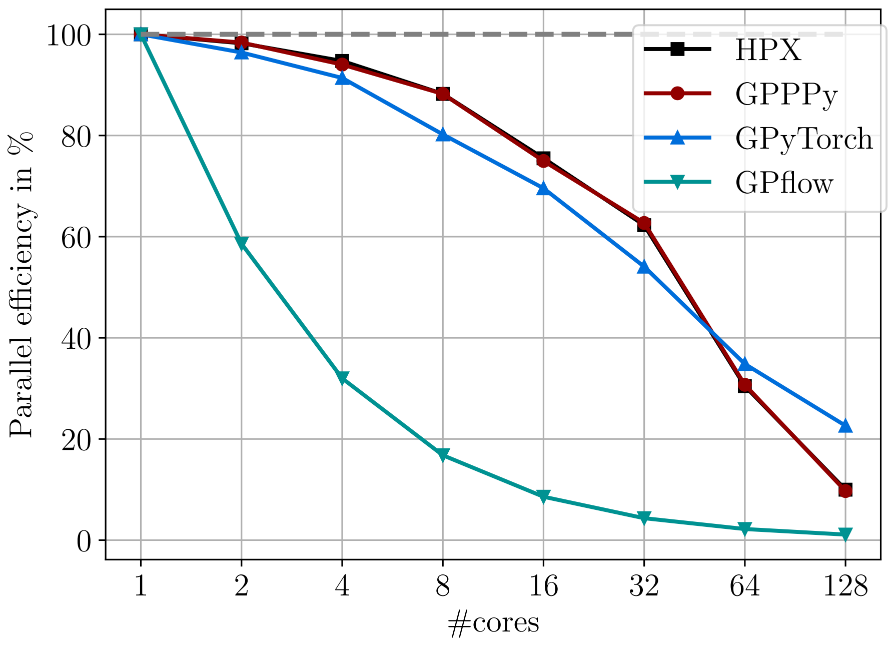


Figure 7.6: Parallel efficiency of the strong scaling benchmark in Figure 7.4 for the prediction with uncertainty step of GPPPy, HPX, GPflow, and GPyTorch in double precision.

7.2.4 Additional finding

Figure 7.7 depicts a strong scaling benchmark for the prediction with uncertainty step of the GPPPy, which only computes predictions with uncertainties, and GPyTorch used with LOVE. The y-axis shows the runtime on a logarithmic scale and x-axis the different number of cores. The problem size is $N = 20000$ training samples and $M = 5000$ test samples, with all computations performed in double precision for the following system specifications as depicted in Table 7.1. The results show that GPPPy demonstrates significant performance scaling as the number of cores increases, closely following the ideal scaling line represented by the dashed grey line. GPPPy starts with runtimes of around 198 seconds on one core. As the number of cores increases, it reaches approximately 8.5 seconds for 64 cores, achieving a speedup of factor 22.5. However, the runtime increases to about 13 seconds for 128 cores. GPyTorch used with LOVE has a lower runtime than GPPPy when the number of cores is less than 16. After that GPPPy has a lower runtime. Overall GPyTorch used with LOVE achieves a speedup of factor 1.4. Furthermore, we observe that GPPPy achieves a speedup of up to 2.8 times over GPyTorch used with LOVE, which has a better asymptotic complexity, when using 16 or more cores.

The reason for the constrained performance scaling of GPPPy on up to 128 cores is identical to the one given in the experiment for the optimization (Section 7.2.2). The runtime of GPyTorch used with LOVE is influenced by the parameter `max_root_decomposition_size`. This parameter controls the size of the low-rank decomposition used for variance estimations.[7] Smaller values result in faster computations but with lower accuracy of variance estimates. Although we were not able to

[7] https://docs.gpytorch.ai/en/latest/settings.html#gpytorch.settings.fast_pred_var

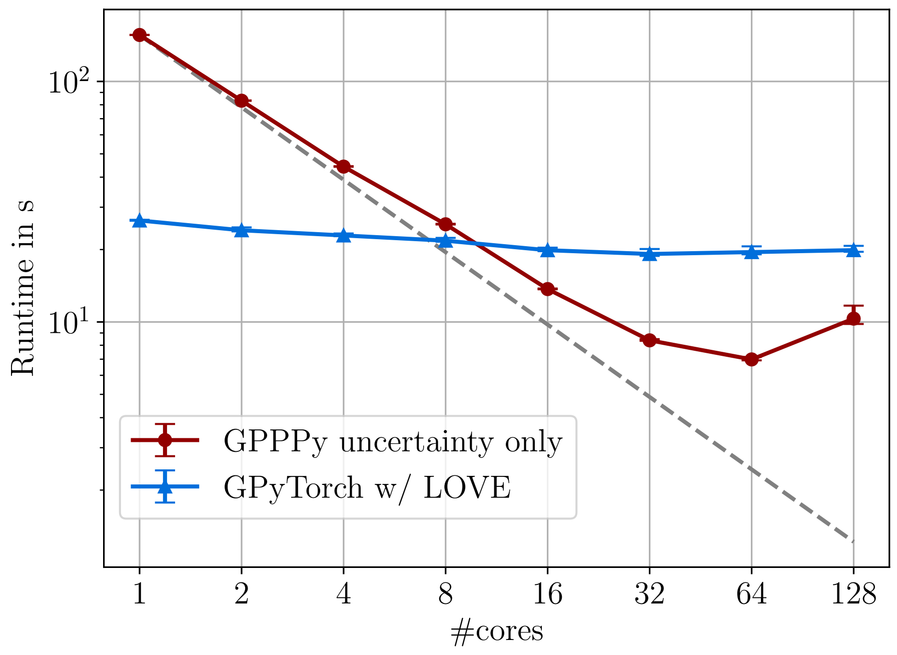


Figure 7.7: Strong scaling prediction with uncertainty runtime comparison between GPPPy and GPyTorch used with LOVE implementations on a dual socket AMD EPYC 7742 with 128 cores. For GPPPy the tile size was set to 25×25 for a workload size of $N = 20000$ training and $M = 5000$ test samples. All implementations use double precision.

identify the exact component of the LOVE method, responsible for the constrained performance scaling, we analyzed the functions, e.g., `torch.matmul()`, which are used to compute the result of the Equation (3.8) on page 24 by using the precomputed approximation. We were able to verify that all these functions can be parallelized, as evidenced by the observed decrease in runtime.

8 Conclusion and outlook

In this work, we introduce a novel GP library for Python named GPPPy. We extend an existing task-based asynchronous GP prediction model by incorporating prediction uncertainty computation and hyperparameter optimization. Additionally, we present a new approach for integrating HPX into Python with pybind11. This allows easy control of the start and termination of the HPX runtime from within the Python environment. To further simplify the combination of HPX with Python, we expose the HPX suspend and resume functions, which either put the OS threads of the thread pool to sleep or wake them up to accept new work. This enables the user to optimize resource management and seamlessly improve the performance of Python applications. Our benchmarks show that there is no runtime overhead when using HPX with pybind11. This ensures that the high-level flexibility offered by using Python does not come at the expense of performance, making the integration highly suitable for practical use.

For our hardware, we determine the optimum number of tiles per dimension on CPUs to be 25. However, it remains uncertain whether this result can be generalized to larger problem sizes. We benchmark our tiled HPX implementation and GPPPy against two reference implementations based on GPflow and GPyTorch that use OpenMP for CPU parallelization. Although the optimization performance of GPPPy is not as good as that of GPyTorch, our library demonstrates significant advantages in terms of prediction performance. In particular, on 64 cores our approach achieves up to 10.5 times faster prediction compared to existing packages. Additionally, GPPPy achieves better parallel efficiency on single-node systems for predictions. Thus, GPPPy can compete with established software tools relying on OpenMP in this non-distributed GP application.

Our in-depth investigation reveals that existing solutions like GPflow do not scale effectively due to the absence of parallelization in TensorFlow operations such as Cholesky decomposition and triangular solve, despite the ability to utilize parallelization for other operations such as the matrix product of tensors. Furthermore, we show that the GPyTorch implementation of the closed-form calculation of predictions with uncertainties includes steps that scale worse than those in GPPPy. This can be explained by the fact that intermediate results are not properly reused for subsequent calculations. In addition, we demonstrate that GPPPy, which only computes predictions with uncertainties, outperforms GPyTorch used with LOVE by a factor of up to 2.8 when using 16 or more cores, despite the latter using an algorithm with superior asymptotic complexity. In conclusion, this highlights the potential for substantial performance improvements in online prediction through our approach, especially in environments with high computational resources.

As the next step, we plan to implement a backpropagation algorithm, similar to those in PyTorch and TensorFlow, to investigate its scaling when used within our task-based asynchronous implementation. Currently, only double precision is supported, we plan to extend our framework to work with single precision as well. We also plan to enable GPU support via HPX's CUDA executors and cuBLAS. Additionally, the implementation can be adjusted to support distributed computing environments, allowing GPPPy to scale across multiple nodes and clusters. Furthermore, the GPPPy implementation will be extended to include more kernel functions, e.g., Matérn kernels, additional optimization algorithms, and support for sparse GPs. Finally, we plan to implement automatic tile size selection. One possible approach is to run simulations for different tile sizes and hardware configurations and then store the results in a file. When new simulations need to be performed, the program only needs to look up the appropriate tile size.

Bibliography

[ABB+99] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen. *LAPACK Users' Guide*. Third. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999. ISBN: 0-89871-447-8 (paperback) (cit. on p. 19).

[ABD+90] E. Angerson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, D. Sorensen. "LAPACK: A portable linear algebra library for high-performance computers". In: *Supercomputing '90:Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*. 1990, pp. 2–11. DOI: 10.1109/SUPERC.1990.129995.

[ADG+16] M. Andrychowicz, M. Denil, S. Gomez, M. W. Hoffman, D. Pfau, T. Schaul, B. Shillingford, N. de Freitas. *Learning to learn by gradient descent by gradient descent*. 2016. arXiv: 1606.04474 [cs.NE]. URL: https://arxiv.org/abs/1606.04474 (cit. on p. 31).

[AG03] D. Abrahams, R. W. Grosse-Kunstleve. "Building hybrid systems with Boost.Python". In: *The C Users Journal archive* 21 (2003). URL: https://api.semanticscholar.org/CorpusID:60451682 (cit. on p. 22).

[BAA+24] S. Balay, S. Abhyankar, M. F. Adams, S. Benson, J. Brown, P. Brune, K. Buschelman, E. Constantinescu, L. Dalcin, A. Dener, V. Eijkhout, J. Faibussowitsch, W. D. Gropp, V. Hapla, T. Isaac, P. Jolivet, D. K. and Dinesh Kaushik, M. G. Knepley, F. Kong, S. Kruger, D. A. May, L. C. McInnes, R. T. Mills, L. Mitchell, T. Munson, J. E. Roman, K. Rupp, P. Sanan, J. Sarich, B. F. Smith, S. Zampini, H. Zhang, H. Zhang, J. Zhang. *PETSc/TAO Users Manual*. Tech. rep. ANL-21/39 - Revision 3.21. Argonne National Laboratory, 2024. DOI: 10.2172/2205494 (cit. on p. 21).

[Bea96] D. M. Beazley. "SWIG: an easy to use tool for integrating scripting languages with C and C++". In: *Proceedings of the 4th Conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4*. TCLTK'96. Monterey, California: USENIX Association, 1996, p. 15 (cit. on p. 22).

[BGMS97] S. Balay, W. D. Gropp, L. C. McInnes, B. F. Smith. "Efficient Management of Parallelism in Object Oriented Numerical Software Libraries". In: *Modern Software Tools in Scientific Computing*. Ed. by E. Arge, A. M. Bruaset, H. P. Langtangen. Birkhäuser Press, 1997, pp. 163–202 (cit. on p. 21).

[BKK+19] M. Bremer, K. Kazhyken, H. Kaiser, C. Michoski, C. Dawson. "Performance Comparison of HPX Versus Traditional Parallelization Strategies for the Discontinuous Galerkin Method". In: *J. Sci. Comput.* 80.2 (2019), pp. 878–902. ISSN: 0885-7474. DOI: 10.1007/s10915-019-00960-z. URL: https://doi.org/10.1007/s10915-019-00960-z (cit. on p. 39).

[BL+15] R. M. Badia Sala, J. J. Labarta Mancho, et al. “A dependency-aware parallel programming model”. In: (2015) (cit. on p. 20).

[BL99] R. D. Blumofe, C. E. Leiserson. “Scheduling multithreaded computations by work stealing”. In: *J. ACM* 46.5 (1999), pp. 720–748. ISSN: 0004-5411. DOI: 10.1145/324133.324234. URL: https://doi.org/10.1145/324133.324234 (cit. on pp. 18, 39).

[BLKD07] A. Buttari, J. Langou, J. Kurzak, J. J. Dongarra. “A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures”. In: *CoRR* abs/0709.1272 (2007). arXiv: 0709.1272. URL: http://arxiv.org/abs/0709.1272 (cit. on p. 36).

[BPP+02] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, et al. “An updated set of basic linear algebra subprograms (BLAS)”. In: *ACM Transactions on Mathematical Software* 28.2 (2002), pp. 135–151 (cit. on pp. 18, 19).

[BS15] F. Berkenkamp, A. P. Schoellig. “Safe and robust learning control with Gaussian processes”. In: *2015 European Control Conference (ECC)*. 2015, pp. 2496–2501. DOI: 10.1109/ECC.2015.7330913 (cit. on p. 32).

[CCZ07] B. Chamberlain, D. Callahan, H. Zima. “Parallel Programmability and the Chapel Language”. In: *Int. J. High Perform. Comput. Appl.* 21.3 (2007), pp. 291–312. ISSN: 1094-3420. DOI: 10.1177/1094342007078442. URL: https://doi.org/10.1177/1094342007078442 (cit. on p. 21).

[CGS+05] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, V. Sarkar. “X10: an object-oriented approach to non-uniform cluster computing”. In: *SIGPLAN Not.* 40.10 (2005), pp. 519–538. ISSN: 0362-1340. DOI: 10.1145/1103845.1094852. URL: https://doi.org/10.1145/1103845.1094852 (cit. on p. 21).

[DB23] P. Diehl, S. R. Brandt. “Interactive C++ code development using C++Explorer and GitHub classroom for educational purposes”. In: *Concurrency and Computation: Practice and Experience* 35.18 (2023), e6893. DOI: https://doi.org/10.1002/cpe.6893. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.6893. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.6893 (cit. on p. 22).

[DBK23] P. Diehl, S. R. Brandt, H. Kaiser. “Shared Memory Parallelism in Modern C++ and HPX”. In: *Asynchronous Many-Task Systems and Applications*. Springer Nature Switzerland, 2023, pp. 27–38. ISBN: 9783031323164. DOI: 10.1007/978-3-031-32316-4_3. URL: http://dx.doi.org/10.1007/978-3-031-32316-4_3 (cit. on pp. 21, 39).

[DBK24] P. Diehl, S. R. Brandt, H. Kaiser. *Parallel C++: Efficient and Scalable High-Performance Parallel Programming Using HPX*. Springer Cham, 2024. DOI: https://doi.org/10.1007/978-3-031-54369-2 (cit. on pp. 39, 40).

[DDHD90] J. J. Dongarra, J. Du Croz, S. Hammarling, I. S. Duff. “A set of level 3 basic linear algebra subprograms”. In: *ACM Trans. Math. Softw.* 16.1 (1990), pp. 1–17. ISSN: 0098-3500. DOI: 10.1145/77626.79170. URL: https://doi.org/10.1145/77626.79170 (cit. on p. 36).

[DFR15] M. P. Deisenroth, D. Fox, C. E. Rasmussen. “Gaussian Processes for Data-Efficient Learning in Robotics and Control”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 37.2 (2015), pp. 408–423. ISSN: 2160-9292. DOI: 10.1109/tpami.2013.218. URL: http://dx.doi.org/10.1109/TPAMI.2013.218 (cit. on p. 17).

[DK17] F. Doshi-Velez, B. Kim. “A Roadmap for a Rigorous Science of Interpretability”. In: *ArXiv* abs/1702.08608 (2017). URL: https://api.semanticscholar.org/CorpusID:17244588 (cit. on p. 17).

[DKL+16] J. Dorris, J. Kurzak, P. Luszczek, A. YarKhan, J. J. Dongarra. “Task-Based Cholesky Decomposition on Knights Corner Using OpenMP”. In: *ISC Workshops*. 2016. URL: https://api.semanticscholar.org/CorpusID:5546947 (cit. on p. 25).

[Duv14] D. Duvenaud. “Automatic model construction with Gaussian processes”. PhD thesis. Apollo - University of Cambridge Repository, 2014. DOI: 10.17863/CAM.14087. URL: https://www.repository.cam.ac.uk/handle/1810/247281 (cit. on pp. 23, 27).

[DYB23] S. Deshmukh, R. Yokota, G. Bosilca. *Cache Optimization and Performance Modeling of Batched, Small, and Rectangular Matrix Multiplication on Intel, AMD, and Fujitsu Processors*. 2023. arXiv: 2311.07602 [cs.PF]. URL: https://arxiv.org/abs/2311.07602 (cit. on p. 40).

[GFB+04] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, T. S. Woodall. “Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation”. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by D. Kranzlmüller, P. Kacsuk, J. Dongarra. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 97–104. ISBN: 978-3-540-30218-6 (cit. on pp. 20, 21).

[Gib98] M. N. Gibbs. “Bayesian Gaussian processes for regression and classification”. PhD thesis. Citeseer, 1998 (cit. on p. 23).

[GJ+10] G. Guennebaud, B. Jacob, et al. *Eigen v3*. http://eigen.tuxfamily.org. 2010 (cit. on p. 19).

[GKGK03] A. Grama, G. Karypis, A. Gupta, V. Kumar. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Addison-Wesley, 2003. ISBN: 0201648652. URL: http://books.google.com/books?hl=en%5C&lr=%5C&id=B3jR2EhdZaMC%5C&oi=fnd%5C&pg=PR17%5C&dq=Purdue+University+Grama%5C&ots=uCCIvLGaHJ%5C&sig=B72Ct5UjD1UI60-JkXLuYbiJr1A (cit. on p. 20).

[GLMM05] J. Generowicz, W. Lavrijsen, M. Marino, P. Mato. “Reflection-Based Python-C++ Bindings”. In: *Lawrence Berkeley National Laboratory* (2005). DOI: 10.5170/CERN-2005-002.441. URL: https://api.semanticscholar.org/CorpusID:73701888 (cit. on p. 21).

[GLS99] W. Gropp, E. Lusk, A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-passing Interface*. Scientific and engineering computation Bd. 1. MIT Press, 1999. ISBN: 9780262571326. URL: https://books.google.de/books?id=xpBZ0RyRb-oC (cit. on p. 20).

[Gon14] P. Gonnet. *Efficient and Scalable Algorithms for Smoothed Particle Hydrodynamics on Hybrid Shared/Distributed-Memory Architectures*. 2014. arXiv: 1404.2303 [cs.DC]. URL: https://arxiv.org/abs/1404.2303 (cit. on p. 21).

[GPB+18] J. R. Gardner, G. Pleiss, D. Bindel, K. Q. Weinberger, A. G. Wilson. “Torch: Blackbox Matrix-Matrix Gaussian Process Inference with GPU Acceleration”. In: *Advances in Neural Information Processing Systems*. 2018 (cit. on pp. 17, 19, 30).

[GPy12] GPy. *GPy: A Gaussian process framework in python*. http://github.com/SheffieldML/GPy. 2012 (cit. on pp. 17, 19).

[HMW+20] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, T. E. Oliphant. “Array programming with NumPy”. In: *Nature* 585.7825 (2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2 (cit. on p. 17).

[HPC+15] M. Huck, A. Porterfield, N. Chaimov, H. Kaiser, A. Malony, T. Sterling, R. Fowler. “An Autonomic Performance Environment for Exascale”. In: *Supercomput. Front. Innov.: Int. J.* 2.3 (2015), pp. 49–66. ISSN: 2409-6008 (cit. on p. 39).

[IMR24] F. Imam, P. Musilek, M. Z. Reformat. “Parametric and Nonparametric Machine Learning Techniques for Increasing Power System Reliability: A Review”. In: *Information* 15.1 (2024). ISSN: 2078-2489. DOI: 10.3390/info15010037. URL: https://www.mdpi.com/2078-2489/15/1/37 (cit. on p. 34).

[Int24] Intel Corporation. *Intel Math Kernel Library*. Version 2024.1.0, retrieved from https://registrationcenter-download.intel.com/akdlm/IRC_NAS/2f3a5785-1c41-4f65-a2f9-ddf9e0db3ea0/l_onemkl_p_2024.1.0.695_offline.sh. Intel Corporation. 2024 (cit. on pp. 18, 39, 40).

[JRM17] W. Jakob, J. Rhinelander, D. Moldovan. *pybind11 – Seamless operability between C++11 and Python*. https://github.com/pybind/pybind11. 2017 (cit. on pp. 18, 22, 39, 41).

[KB17] D. P. Kingma, J. Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG] (cit. on pp. 30, 32, 33).

[KDL+20] H. Kaiser, P. Diehl, A. S. Lemoine, B. A. Lelbach, P. Amini, A. Berge, J. Biddiscombe, S. R. Brandt, N. Gupta, T. Heller, K. Huck, Z. Khatami, A. Kheirkhahan, A. Reverdell, S. Shirzad, M. Simberg, B. Wagle, W. Wei, T. Zhang. “HPX - The C++ Standard Library for Parallelism and Concurrency”. In: *Journal of Open Source Software* 5.53 (2020), p. 2352. DOI: 10.21105/joss.02352. URL: https://doi.org/10.21105/joss.02352 (cit. on pp. 17, 21, 39, 40).

[KHA+14] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, D. Fey. “HPX: A Task Based Programming Model in a Global Address Space”. In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. PGAS ’14. Eugene, OR, USA: Association for Computing Machinery, 2014. ISBN: 9781450332477. DOI: 10.1145/2676870.2676883. URL: https://doi.org/10.1145/2676870.2676883 (cit. on pp. 21, 39).

[Koc05] R. Kocijan Jušand Murray-Smith. “Nonlinear Predictive Control with a Gaussian Process Model”. In: *Switching and Learning in Feedback Systems: European Summer School on Multi-Agent Control, Maynooth, Ireland, September 8-10, 2003, Revised Lectures and Selected Papers*. Ed. by R. Murray-Smith, R. Shorten. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 185–200. ISBN: 978-3-540-30560-6. DOI: 10.1007/978-3-540-30560-6_8. URL: https://doi.org/10.1007/978-3-540-30560-6_8 (cit. on p. 33).

[Koc15] J. Kocijan. “Modelling and Control of Dynamic Systems Using Gaussian Process Models”. In: 2015. URL: https://api.semanticscholar.org/CorpusID:63921279 (cit. on p. 23).

[Koc19] J. Kocijan. *Modelling and Control of Dynamic Systems Using Gaussian Process Models*. Advances in Industrial Control. Springer International Publishing, 2019. ISBN: 9783319793276. URL: https://books.google.de/books?id=98TrvwEACAAJ (cit. on p. 25).

[KVL23] B. Kundu, V. T. Vassilev, W. Lavrijsen. “Efficient and Accurate Automatic Python Bindings with cppyy & Cling”. In: *ArXiv* abs/2304.02712 (2023). DOI: https://doi.org/10.48550/arXiv.2304.02712. URL: https://api.semanticscholar.org/CorpusID:257985473 (cit. on p. 21).

[LCOW19] H. Liu, J. Cai, Y.-S. Ong, Y. Wang. “Understanding and comparing scalable Gaussian process regression for big data”. In: *Knowledge-Based Systems* 164 (2019), pp. 324–335. ISSN: 0950-7051. DOI: https://doi.org/10.1016/j.knosys.2018.11.002. URL: https://www.sciencedirect.com/science/article/pii/S0950705118305380 (cit. on p. 17).

[LHKK79] C. L. Lawson, R. J. Hanson, D. R. Kincaid, F. T. Krogh. “Basic Linear Algebra Subprograms for Fortran Usage”. In: *ACM Trans. Math. Softw.* 5.3 (1979), pp. 308–323. ISSN: 0098-3500. DOI: 10.1145/355841.355847. URL: https://doi.org/10.1145/355841.355847 (cit. on p. 36).

[LOSC20] H. Liu, Y.-S. Ong, X. Shen, J. Cai. “When Gaussian Process Meets Big Data: A Review of Scalable GPs”. In: *IEEE Transactions on Neural Networks and Learning Systems* 31.11 (2020), pp. 4405–4423. DOI: 10.1109/TNNLS.2019.2957109 (cit. on p. 17).

[LPH23] K. Liegeois, M. Perego, T. Hartland. “PyAlbany: A Python interface to the C++ multiphysics solver Albany”. In: *Journal of Computational and Applied Mathematics* 425 (2023), p. 115037. ISSN: 0377-0427. DOI: https://doi.org/10.1016/j.cam.2022.115037. URL: https://www.sciencedirect.com/science/article/pii/S0377042722006355 (cit. on p. 21).

[Mac+98] D. J. MacKay et al. “Introduction to Gaussian processes”. In: *NATO ASI series F computer and systems sciences* 168 (1998), pp. 133–166 (cit. on p. 17).

[MAP+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/ (cit. on pp. 17, 19).

[Mur13] K. P. Murphy. *Machine learning : a probabilistic perspective*. Cambridge, Mass. [u.a.]: MIT Press, 2013. URL: https://www.amazon.com/Machine-Learning-Probabilistic-Perspective-Computation/dp/0262018020/ref=sr_1_2?ie=UTF8%5C&qid=1336857747%5C&sr=8-2 (cit. on p. 33).

[MvN+17] A. G. d. G. Matthews, M. van der Wilk, T. Nickson, K. Fujii, A. Boukouvalas, P. León-Villagrá, Z. Ghahramani, J. Hensman. "GPflow: A Gaussian process library using TensorFlow". In: *Journal of Machine Learning Research* 18.40 (2017), pp. 1–6. URL: http://jmlr.org/papers/v18/16-537.html (cit. on pp. 17, 19, 30).

[NTW22] D. Nance, S. Tomov, K. Wong. "A Python Library for Matrix Algebra on GPU and Multicore Architectures". In: *2022 IEEE 19th International Conference on Mobile Ad Hoc and Smart Systems (MASS)*. 2022, pp. 770–775. DOI: 10.1109/MASS56207.2022.00121 (cit. on p. 21).

[Oli07] T. E. Oliphant. "Python for Scientific Computing". In: *Computing in Science & Engineering* 9.3 (2007), pp. 10–20. DOI: 10.1109/MCSE.2007.58 (cit. on p. 21).

[Ope08] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 3.0*. 2008. URL: http://www.openmp.org/mp-documents/spec30.pdf (cit. on p. 21).

[PDC+14] G. Pillonetto, F. Dinuzzo, T. Chen, G. De Nicolao, L. Ljung. "Kernel methods in system identification, machine learning and function estimation: A survey". In: *Automatica* 50.3 (2014), pp. 657–682. ISSN: 0005-1098. DOI: https://doi.org/10.1016/j.automatica.2014.01.001. URL: https://www.sciencedirect.com/science/article/pii/S000510981400020X (cit. on p. 32).

[PGH11] F. Pérez, B. E. Granger, J. D. Hunter. "Python: An Ecosystem for Scientific Computing". In: *Computing in Science & Engineering* 13.2 (2011), pp. 13–21. DOI: 10.1109/MCSE.2010.119 (cit. on p. 21).

[PGM+19] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019. arXiv: 1912.01703 [cs.LG]. URL: https://arxiv.org/abs/1912.01703 (cit. on pp. 17, 19).

[PGWW18] G. Pleiss, J. R. Gardner, K. Q. Weinberger, A. G. Wilson. *Constant-Time Predictive Distributions for Gaussian Processes*. 2018. arXiv: 1803.06058 [cs.LG]. URL: https://arxiv.org/abs/1803.06058 (cit. on p. 55).

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[Rob13] A. D. Robison. "Composable Parallel Patterns with Intel Cilk Plus". In: *Computing in Science & Engineering* 15.2 (2013), pp. 66–71. DOI: 10.1109/MCSE.2013.21 (cit. on p. 21).

[RW06] C. E. Rasmussen, C. K. I. Williams. *Gaussian processes for machine learning*. Adaptive computation and machine learning. MIT Press, 2006, pp. I–XVIII, 1–248. ISBN: 026218253X (cit. on pp. 17, 23, 24).

[RWM20] M. Revay, R. Wang, I. R. Manchester. *A Convex Parameterization of Robust Recurrent Neural Networks*. 2020. arXiv: 2004.05290 [cs.LG]. URL: https://arxiv.org/abs/2004.05290 (cit. on p. 33).

[Sär19] S. Särkkä. *The Use of Gaussian Processes in System Identification*. 2019. arXiv: 1907.06066 [stat.ML]. URL: https://arxiv.org/abs/1907.06066 (cit. on p. 33).

[SGCD16] M. Schaller, P. Gonnet, A. B. G. Chalk, P. W. Draper. "SWIFT: Using Task-Based Parallelism, Fully Asynchronous Communication, and Graph Partition-Based Domain Decomposition for Strong Scaling on more than 100,000 Cores". In: *Proceedings of the Platform for Advanced Scientific Computing Conference*. PASC '16. ACM, 2016. DOI: 10.1145/2929908.2929916. URL: http://dx.doi.org/10.1145/2929908.2929916 (cit. on p. 21).

[SL19] J. Schoukens, L. Ljung. "Nonlinear System Identification: A User-Oriented Roadmap". In: *CoRR* abs/1902.00683 (2019). arXiv: 1902.00683. URL: http://arxiv.org/abs/1902.00683 (cit. on p. 20).

[SP23] A. Strack, D. Pflüger. "Scalability of Gaussian Processes Using Asynchronous Tasks: A Comparison Between HPX and PETSc". In: *Workshop on Asynchronous Many-Task Systems and Applications*. Springer. 2023, pp. 52–64 (cit. on pp. 21, 25, 35).

[SS17] P. F. Schulam, S. Saria. "What-If Reasoning with Counterfactual Gaussian Processes". In: *ArXiv* abs/1703.10651 (2017). URL: https://api.semanticscholar.org/CorpusID:16737818 (cit. on p. 17).

[TDH+18] P. Thoman, K. Dichev, T. Heller, R. Iakymchuk, X. Aguilar, K. Hasanov, P. Gschwandtner, P. Lemarinier, S. Markidis, H. Jordan, T. Fahringer, K. Katrinis, E. Laure, D. S. Nikolopoulos. "A taxonomy of task-based parallel programming technologies for high-performance computing". In: *J. Supercomput.* 74.4 (2018), pp. 1422–1434. ISSN: 0920-8542. DOI: 10.1007/s11227-018-2238-4. URL: https://doi.org/10.1007/s11227-018-2238-4 (cit. on pp. 21, 39).

[Tit09] M. K. Titsias. "Variational Learning of Inducing Variables in Sparse Gaussian Processes". In: *International Conference on Artificial Intelligence and Statistics*. 2009. URL: https://api.semanticscholar.org/CorpusID:7811257 (cit. on p. 20).

[TWS+18] R. Tohid, B. Wagle, S. Shirzad, P. Diehl, A. Serio, A. Kheirkhahan, P. Amini, K. Williams, K. Isaacs, K. Huck, S. Brandt, H. Kaiser. "Asynchronous Execution of Python Code on Task-Based Runtime Systems". In: *2018 IEEE/ACM 4th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*. IEEE, 2018. DOI: 10.1109/espm2.2018.00009. URL: http://dx.doi.org/10.1109/ESPM2.2018.00009 (cit. on p. 22).

[Wal82] D. W. Wall. "Messages as active agents". In: *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '82. Albuquerque, New Mexico: Association for Computing Machinery, 1982, pp. 34–39. ISBN: 0897910656. DOI: 10.1145/582153.582157. URL: https://doi.org/10.1145/582153.582157 (cit. on p. 39).

[Wan23] J. Wang. “An Intuitive Tutorial to Gaussian Process Regression”. In: *Computing in Science & Engineering* 25.4 (2023), pp. 4–11. DOI: 10.1109/MCSE.2023.3342149 (cit. on pp. 25, 26).

[WJ18] Z. Wang, S. Jegelka. *Max-value Entropy Search for Efficient Bayesian Optimization*. 2018. arXiv: 1703.01968 [stat.ML]. URL: https://arxiv.org/abs/1703.01968 (cit. on p. 17).

[WMT08] K. B. Wheeler, R. C. Murphy, D. Thain. “Qthreads: An API for programming with millions of lightweight threads”. In: *2008 IEEE International Symposium on Parallel and Distributed Processing*. 2008, pp. 1–8. DOI: 10.1109/IPDPS.2008.4536359 (cit. on p. 21).

[WP08] T. Willhalm, N. Popovici. “Putting intel® threading building blocks to work”. In: *Proceedings of the 1st International Workshop on Multicore Software Engineering*. IWMSE ’08. Leipzig, Germany: Association for Computing Machinery, 2008, pp. 3–4. ISBN: 9781605580319. DOI: 10.1145/1370082.1370085. URL: https://doi.org/10.1145/1370082.1370085 (cit. on p. 21).

[WR95] C. Williams, C. Rasmussen. “Gaussian processes for regression”. In: *Advances in neural information processing systems* 8 (1995) (cit. on p. 17).

[WYZ21] X. Wang, L. Yan, Q. Zhang. “Research on the Application of Gradient Descent Algorithm in Machine Learning”. In: *2021 International Conference on Computer Network, Electronic and Automation (ICCNEA)*. 2021, pp. 11–15. DOI: 10.1109/ICCNEA53019.2021.00014 (cit. on p. 31).

[WZZY13] Q. Wang, X. Zhang, Y. Zhang, Q. Yi. “AUGEM: automatically generate high performance dense linear algebra kernels on x86 CPUs”. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC ’13. Denver, Colorado: Association for Computing Machinery, 2013. ISBN: 9781450323789. DOI: 10.1145/2503210.2503219. URL: https://doi.org/10.1145/2503210.2503219 (cit. on p. 19).

[XQY12] Z. Xianyi, W. Qian, Z. Yunquan. “Model-driven Level 3 BLAS Performance Optimization on Loongson 3A Processor”. In: *2012 IEEE 18th International Conference on Parallel and Distributed Systems*. 2012, pp. 684–691. DOI: 10.1109/ICPADS.2012.97 (cit. on p. 19).

[YJG03] A. B. Yoo, M. A. Jette, M. Grondona. “SLURM: Simple Linux Utility for Resource Management”. In: ed. by D. G. Feitelson, L. Rudolph, U. Schwiegelshohn. Vol. 2862. Lecture Notes in Computer Science. Springer, 2003, pp. 44–60. DOI: 10.1007/10968987_3. URL: https://doi.org/10.1007/10968987%5C_3 (cit. on p. 59).

[YKS23] J. Yan, H. Kaiser, M. Snir. “Design and Analysis of the Network Software Stack of an Asynchronous Many-task System – The LCI parcelport of HPX”. In: *Proceedings of the SC ’23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. SC-W ’23. Denver, CO, USA: Association for Computing Machinery, 2023, pp. 1151–1161. ISBN: 9798400707858. DOI: 10.1145/3624062.3624598. URL: https://doi.org/10.1145/3624062.3624598 (cit. on p. 39).

[YLR+05] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, P. Stodghill. “Is Search Really Necessary to Generate High-Performance BLAS?” In: *Proceedings of the IEEE* 93.2 (2005), pp. 358–386. DOI: 10.1109/JPROC.2004.840444.

[Zha19] J. Zhang. *Gradient Descent based Optimization Algorithms for Deep Learning Models Training*. 2019. arXiv: 1903.03614 [cs.LG]. URL: https://arxiv.org/abs/1903.03614 (cit. on p. 31).

All links were last accessed on August 14, 2024.

A Supplementary materials

A.1 Schur complement of a Cholesky factorization

The computation of prediction uncertainty can alternatively be accomplished using the Schur complement of the Cholesky factorization, rather than following the approach depicted in Section 3.2. First step is to construct the covariance matrix K, prior covariance matrix $K_{\hat{Z},\hat{Z}}$, and trans- and cross-covariance matrices $K_{Z,\hat{Z}}$ and $K_{\hat{Z},Z}$. Then, these matrices are utilized to construct the matrix A

$$A = \begin{bmatrix} K & K_{Z,\hat{Z}} \\ K_{\hat{Z},Z} & K_{\hat{Z},\hat{Z}} \end{bmatrix}. \tag{A.1}$$

For the subsequent computations to be valid, A must be a symmetric positive definite matrix. Upon verification of A's positive definiteness, its Cholesky decomposition can be performed, yielding

$$A = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} L_{11}^\top & L_{21}^\top \\ 0 & L_{22}^\top \end{bmatrix} = \begin{bmatrix} L_{11}L_{11}^\top & L_{11}L_{21}^\top \\ L_{21}L_{11}^\top & L_{21}L_{21}^\top + L_{22}L_{22}^\top \end{bmatrix} \tag{A.2}$$

We aim to demonstrate that $S = L_{22}L_{22}^\top$, where S represents the Schur complement of K, given by $S = K_{\hat{Z},\hat{Z}} - K_{\hat{Z},Z}K^{-1}K_{Z,\hat{Z}}$. This equivalence is crucial as it underpins the computation of prediction variance, ensuring the same result is obtained through this alternative method. Since we know that $K = L_{11}L_{11}^\top$ is invertible because K is symmetric positive definite, it follows that

$$K^{-1} = (L_{11}L_{11}^\top)^{-1} = (L_{11}^\top)^{-1}L_{11}^{-1}. \tag{A.3}$$

Using the result from A.3, we can show that

$$\begin{aligned} S &= K_{\hat{Z},\hat{Z}} - K_{\hat{Z},Z}K^{-1}K_{Z,\hat{Z}} && \text{(A.4)} \\ &= L_{21}L_{21}^\top + L_{22}L_{22}^\top - L_{21}\underbrace{L_{11}^\top(L_{11}^\top)^{-1}L_{11}^{-1}L_{11}}_{=I}L_{21}^\top && \text{(A.5)} \\ &= L_{22}L_{22}^\top && \text{(A.6)} \end{aligned}$$

Now, we observe that the computation of the prediction uncertainty is reduced to a simple matrix-by-matrix multiplication, where we only need to compute the diagonal terms of the product. For the predictions, we can continue to use L_{11} so that no additional steps are necessary compared to the usual calculation of the predictions.

However, there are some bottlenecks associated with this approach. The first is that the Cholesky decomposition scales with $O(N^3)$. This means that if our test input is large, the Cholesky decomposition becomes computationally more expensive. Second, if we need to make predictions for a different set of test data, we would need to recalculate the Cholesky decomposition for A. However, this is not efficient, so in this case we can simply fall back to the approach described in Section 3.2 on page 25 and reuse L_{11}. Investigating this is left to future work.

A.2 Data loading procedure for GPyTorch and GPflow

Listing A.1 Generate feature matrix

```python
def generate_feature_matrix(x_original, n_regressors):
"""
Generate feature matrix by padding the original input array with zeros
from the left for (n_regressors - 1) positions, and rolling over the input
array where the window size equals n_regressors.
"""
X = []
x_padded = np.pad(x_original, pad_width=(n_regressors - 1, 0),  mode="constant",
                  constant_values=(0))

for _ in range(len(x_original)):
X.append(x_padded[:n_regressors])
x_padded = np.roll(x_padded, -1)
return np.array(X)
```

Listing A.2 Data loading procedure for GPyTorch and GPflow

```python
def load_data(train_in_path, train_out_path, test_in_path, size_train: int,
size_test: int,  n_regressors: int,):
"""
Load data and generate feature matrix for Gaussian process regression.
"""
x_train_in = np.loadtxt(train_in_path, dtype="d")[:size_train]
x_test_in = np.loadtxt(test_in_path, dtype="d")[:size_test]

X_train = generate_feature_matrix(x_train_in, n_regressors).astype("d")
X_test = generate_feature_matrix(x_test_in, n_regressors).astype("d")
Y_train = np.loadtxt(train_out_path, dtype="d")[:size_train, None]
Y_test = np.loadtxt(test_out_path, dtype="d")[:size_test, None]

return X_train, Y_train, X_test, Y_test
```

A.3 Example implementation for GPPPy

Listing A.3 provides a complete working example in C++, which also represents the framework for the experiments. The provided C++ code sets up and runs a GP for regression analysis. The configuration section (lines 8 – 13) initializes the number of training and testing data samples, optimization iterations, number of cores for parallel processing, number of tiles, and regressors that depict the number of lagged variables in the feature vector. It also defines file paths for training and testing data (lines 16 – 18). Command-line arguments are modified to specify the number of threads for the HPX runtime, which is used for parallel execution (lines 21 – 28). Once the initialization is done, tile sizes for training and prediction are computed (lines 31 – 332), the hyperparameters are set (lines 36), and the training and testing data is loaded (lines 38 – 40). The next step is to initialize a GP model with training data and previously specified parameters (lines 45 – 48). Then, HPX runtime is started to manage parallel tasks (line 50), after which the GP model's hyperparameters are optimized (line 52). Afterward, predictions with uncertainty are made for the test data using the optimized parameters (lines 54 – 55). To compute the full posterior covariance matrix, the function `predict_with_full_cov()` should be called. Finally, the HPX runtime is stopped before exiting the program (line 57).

Similar to the C++ example of GPPPy, Listing A.4 provides an example of the Python version that is also used for the experiments. The provided Python code sets up and performs a GP regression using the GPPPy library. The main function loads the configuration file and then calls `gpppy_run()`. The configuration section initializes the necessary parameters such as the number of training and testing data samples (`N_TRAIN`, `N_TEST`), the number of optimization iterations (`OPT_ITER`), the number of CPU cores for parallel processing (`N_CORES`), and the number of training tiles (`N_TILES`) (lines 14 – 18). Once the initialization is complete, the tile sizes for training and prediction are computed (lines 20 – 21). Next, file paths for the training and testing data are defined and loaded from the specified file paths (lines 24 – 26). Afterward, the hyperparameters for the optimization using Adam are set (line 29 – 30). A GP model is then initialized with the training data and the previously specified parameters (lines 33 – 34). To manage parallel tasks such as training the model and making predictions, the HPX runtime needs to be started (line 37). The GP model's hyperparameters are optimized (line 40). Predictions with uncertainty are made for the test data using the trained GP model (lines 43 – 44). To calculate the full posterior covariance matrix, the function `predict_with_full_cov()` should be called. Finally, the HPX runtime is stopped before exiting the function (line 47).

Listing A.3 Example for GPPPy in C++

```cpp
#include <gaussian_process>

int main(int argc, char *argv[])
{
    ////////////////////
    /////// configuration
    int n_train = 10000; // Number of training samples
    int n_test = 5000; // Number of testing samples
    const int OPT_ITER = 1; // Number of optimization iterations
    const std::size_t N_CORES = 4; // Number of cores for parallel processing
    const int n_tiles = 25; // Number of tiles
    const int n_reg = 100; // Number of regressors for each lagged variable

    // File paths for training and testing data
    std::string train_path = "../data/training/training_input.txt";
    std::string out_path = "../data/training/training_output.txt";
    std::string test_path = "../data/test/test_input.txt";

    // Create new argc and argv to include the --hpx:threads argument
    std::vector<std::string> args(argv, argv + argc);
    args.push_back("--hpx:threads=" + std::to_string(N_CORES));
    // Convert the arguments to char* array
    std::vector<char *> cstr_args;
    for (auto &arg : args){cstr_args.push_back(const_cast<char*>(arg.c_str()));}
    // Update argc and argv with the new arguments
    int new_argc = static_cast<int>(cstr_args.size());
    char **new_argv = cstr_args.data();

    // Compute tile sizes and number of prediction tiles
    int tile_size = utils::compute_train_tile_size(n_train, n_tiles);
    auto result = utils::compute_test_tiles(n_test, n_tiles, tile_size);

    ////////////////////
    ///// hyperparams
    gpppy_hyper::Hyperparameters hpar = {0.1, 0.9, 0.999, 1e-8, OPT_ITER};
    ////// data loading
    gpppy::GP_data training_input(train_path, n_train);
    gpppy::GP_data training_output(out_path, n_train);
    gpppy::GP_data test_input(test_path, n_test);

    ////////////////////
    ///// GP
    // Specify which hyperparameters are trainable
    std::vector<bool> trainable = {true, true, true};
    // Initialize the GP model with training data and parameters
    gpppy::GP gp(training_input.data, training_output.data, n_tiles,
    tile_size, 1.0, 1.0, 0.1, n_reg, trainable);
    // Initialize HPX runtime with the new arguments, don't run hpx_main
    utils::start_hpx_runtime(new_argc, new_argv);
    // Optimize the hyperparameters
    std::vector<double> losses = gp.optimize(hpar);
    // Make predictions with uncertainty for the test data
    std::vector<std::vector<double>> sum = gp.predict_with_uncertainty(
    test_input.data, result.first, result.second);
    // Stop the HPX runtime
    utils::stop_hpx_runtime();

    // Exit the program
    return 0;
}
```

Listing A.4 Example for GPPPy in Python

```python
import gaussian_process as gpppy
from config import get_config
...

def gpppy_run(config):
    """
    Run Gaussian Process regression using the GPPPy library.

    Parameters:
    config (dict): Configuration dictionary containing various parameters.
    """

    # Number of training data points.
    n_train = config["N_TRAIN"]
    # Number of CPU cores to use.
    cores = config["N_CORES"]
    # Compute the training tile size based on the number of training points and tiles.
    n_tile_size = gpppy.compute_train_tile_size(n_train, config["N_TILES"])
    # Compute the number and size of test tiles based on the total number of test points and tiles.
    m_tiles, m_tile_size = gpppy.compute_test_tiles(config["N_TEST"],
    config["N_TILES"], n_tile_size)

    # Load training and test data from specified files.
    train_in = gpppy.GP_data(config["train_in_file"], n_train)
    train_out = gpppy.GP_data(config["train_out_file"], n_train)
    test_in = gpppy.GP_data(config["test_in_file"], config["N_TEST"])

    # Initialize hyperparameters for the GP with given learning rate and optimization iterations.
    hpar = gpppy.Hyperparameters(learning_rate=0.1, opt_iter=config["OPT_ITER"],
    m_T=[0, 0, 0], v_T=[0, 0, 0])

    # Create the Gaussian Process object with training data and configuration.
    gp = gpppy.GP(train_in.data, train_out.data, config["N_TILES"], n_tile_size,
    trainable=[True, True, True])

    # Initialize the HPX runtime environment but do not start it yet.
    gpppy.start_hpx(sys.argv, cores)

    # Perform optimization on the Gaussian Process to adjust hyperparameters.
    losses = gp.optimize(hpar)

    # Make predictions with uncertainty using the test data.
    predictions, variance = gp.predict_with_uncertainty(test_in.data,
    m_tiles, m_tile_size)

    # Stop the HPX runtime environment.
    gpppy.stop_hpx()

if __name__ == "__main__":
    # Load configuration from a file or other source.
    config = get_config()
    # Run the gpppy process with the specified configuration.
    gpppy_run(config)
```

Declaration