Chapter 7: Parallelisation, Challenges and Future Trends

This chapter discusses parallel and distributed features inherent in tensor modelling, their current implementations in C/C++, and how to use them in Python packages. Then the book is summarised with a discussion of challenges and future trends in tensor decomposition applications.

7.1 Tensor Computing Parallelisation

Since machine learning, particularly deep learning, is compute-intensive and memory-intensive because of the ever-increasing size and dimensionality of the available models, datasets, and compute iterations, parallelisation is essential, whether on the software or hardware level. What is known as Moore's law is the observation made by Gordon Moore in 1965 that the number of transistors in a dense integrated circuit (IC) doubles every 18 months, increasing processing speed. This observation held true from the 1950s up until around the 2010s. Adding more transistors and ever decreasing in size ICs reached their physical limits causing heat dissipation that can not be solved using the same miniaturisation approach. New approaches continue feeding the increasing need for faster processing of the many sectors that have become dependent on it. Various technologies competed to provide speed ups, such as multi-CPU, multi-core, 3D CPU transistors, GP-GPU, TPUs, ASICs, microcontrollers, SoC, SiP, and distributed processing over clusters of computing nodes such as instances in the cloud, supercomputers, IoT devices connecting various sensors, BioChips, and many other passive components. Most machine learning and deep learning packages are designed on a massive stack of optimised algorithms that run on the detected hardware from CPUs and GPUs. You might need to set the library to use these technologies at the higher level of the stack. You can learn the logic, syntax and semantics of using these libraries when you are programming your own algorithms.

The serial programming approach solves the problem logic using a sequence of instructions that are executed sequentially, one after another, in one processor. On the other hand, the parallel programming approach divides the solution steps into discrete parts that can be

solved concurrently on several cores and computing nodes, using any of the Parallel Computer Architectures. Speedups are achieved by carefully planning the division of independent partitions of the logical sequence or managing the dependencies without losing the speedups gained by communication latency.

Computer architectures with a single CPU or multiple processors/cores have shared memory and enable the use of multiple threads. When a network connects an arbitrary number of such computing elements, this architecture will have distributed memory and need message passing. A computer with a GPU card will require moving the processing between CPU device memory and GPU device memory. Python packages to implement parallel programs include threading for multiple threads, mpi4py for message passing, and Numba for GPU processing. These are mainly C or C++ packages for faster processing with Python wrappers. Each of these packages has its online documentation and tutorials from which mastering them first-hand from the developers is easy. Numpy is the Python package for working with optimised arrays with vectorisation, which is the process of computing in parallel all the elements of an array, unlike the built-in sequential arrays of Python.

Building a parallel algorithm requires taking care of various implementation details such as shared memory synchronised access using mutexes or similar structures, distributed processing synchronised sending and receiving, and creating barriers and waves of computation. The TEDx course of Linear Algebra For Frontiers is managed by the authors of a package “Flame for Matrix Partitioning” that implements matrices without manual index manipulations to achieve parallelisation (Geijn and Quintana-Ort´, 2008)**.** A detailed Python parallel and distributed programming experience can be gained from (Zaccone, 2015). Also, the work in the PhD thesis for Tensor Partitioning on a cluster of computing nodes provides an example of wavefront processing of N-D arrays applied to the Multiple Sequence Alignment problem(Helal et al., 2008), (Helal et al., 2009)**.** In this thesis, ND-arrays are also expressed dynamically using a data structure accepting N, shape, and data as parameters and creating a linear array in memory. The N-dimensional index is then parameterised to access a specific location in the array or update it. This is very important not to fix the array dimensionality and shape the way Numpy, MatLab, Mathematica and others do, limiting the ability to dynamically create these arrays, particularily in the tensoriation step as shown in the tensorisation notebook.

Google TensorFlow, Facebook PyTorch , Apache MXNet, Microsoft CNTK and the many DL Frameworks are built over a stack of optimised algorithms that run in parallel in various hardware architectures that they can be compiled to during installation. Table 1 summarises the features of some of the available DL frameworks in terms of hardware readiness and the type of computational graph built. The Open Neural Network Exchange (ONNX) has been proposed to provide interoperability between frameworks. It accepts a model in one of the

supported frameworks as input, identifies the common operators, and generates a file for a lower-level compiler optimised for a particular hardware platform.

Table 1: DL Framework comparison

	Static Graph	**Dynamic Graph**	**Both**
General Purpose HW: CPU/GPU	TensorFlow/ TensorRT / Caffe/ CNTK	PyTorch	MXNet
Embedded: ARM/IoT	TensorFlow Lite	PyTorch Mobile	Gluon
High-Level API	Keras	FastAI	Gluon

The numerical recipes optimising solving equations and various Linear Algebra computations are standardised in the Basic Linear Algebra Subprograms (BLAS) libraries. DL computations have been standardised in similar libraries such as MKL-DNN and cuDNN. Nvidia TensorRT accepts models from all DL frameworks and uses optimisation accelerators on supported hardware platforms. It supports graph optimisation (e.g., layer fusion) and low-bit quantisation with a large collection of highly optimised Nvidia GPU kernels. TensorRT also enables Kernel-Auto Tuning by choosing the best data layer and best parallel algorithms for the target hardware platform. It also has Dynamic Tensor Memory (Memory optimisation) feature that reduces memory footprint and improves memory re-use by allocating memory for each Tensor only for the duration of its usage, and a Multi-Stream Execution feature that scales to multiple input streams by processing them in parallel using the same model and weights. For example, a PyTorch 1.4.0 model is accelerated by quantisation from FP32 to FP16 using TensorRT to Cuda 10.1 Titan V, i7-7800X achieved almost double speedup than the original model. ***Research Project:*** BLAS was introduced in 1969 with vector operations, updated in the 70s for matrix-vector operations and in the 80s for matrix-matrix operations. Tensor-tensor operations BLAS for the fourth order tensors were implemented in (Liu and Wang, 2017), including tensor (Kronecker) product, KhatriRao product, Hadamard product, tensor contraction, t-product or L-product. The need for a similar highly optimised Tensor Decomposition for a variable (n) order tensors mathematical library that is efficient on computational tractability and interoperable between the different DL frameworks and hardware is also required.

Please note that Tensor in TensorFlow, TensorRT and many other DL current frameworks are limited to predefined ordered two- or three- or four-dimensional arrays and are still

working on the traditional vectorised pair-wise models or spatial matrices for convolutions layers. The first dimension is usually the Batch number (learning occurs through epochs of applying input batches to the input layer), the second or last dimension is usually a timestep/or channel, and the data is in vector form in the remaining dimension for Fully connected layers. For 2D convolution, two dimensions are used for the spatial dimensions, such as image width and height. The video dataset has three spatial dimensions; the extra dimension is for the time or frames. Therefore, the data is in vector form, fixed 2D, or fixed 3D ordered dimensions, while the framework's name has Tensor in it, implying that it can grow as a multi-way for any multi-way dataset. Graph Neural Networks (GNN) are currently well developed to accept data as adjacency matrices of graph data structures. The various NN tensorisation projects reviewed in chapter six did not create a framework yet. I will mention Tensor Decomposition models to mean the ability to accept input in N-D arrays as implemented in Numpy and Tensorly with the ability to define the modes in any order based on the problem requirements. ***Research Project:*** Tensorised NN Frameworks can be built with all Tensorised Layer types implemented, tensorised activation functions, and tensorised forward and backwards propagation algorithms such as the SGD with DMRG algorithms, AutoDiff (Paszke et al., 2017) and DDSP (differentiable digital signal processing) (Engel et al., 2020).

The chips/processors that were designed for AI are called XPU. These include GPUs, FPGAs, and Application Specific Integrated Circuits (ASICs), such as neural processing units (NPUs). Matrix operations specialised/dedicated hardware has been built to optimise DL performance, such as Google TPU, Hisilicon NPU, Apple Bonic, tensor cores in NVIDIA Volta/Turing Architecture, Intel Nervana neural network processors (NNP), Tensor Computing Processor BM1684 , Amazon Inferentia with NeuroCores, Hanguang Alibaba Ali-NPU, Knupath Hermosa, Baidu XPU, Qualcomm Cloud AI 100, Cambricon MLU270, Graphcore GC2, AVX512 vector units and tensor core, and FPGA DL ready components.

The TPU is 15 to 30 times faster than current GPUs and CPUs. A TPU uses a matrix as a primitive instead of a vector or scalar in CPUs and GPUs, which means ND-Arrays still need to be matricised. It includes Matrix Multiplier Unit (MXU), Unified Buffer (UB), and Activation Unit (AU), which is driven with CISC instructions by the host processor. The MXU is power and area optimised and is composed of a systolic array to perform matrix multiplications. The TPU architecture has bottlenecks that are identified in (Wang et al., 2019). Their study proposed a new parameterised deep learning benchmark suite (ParaDNN) to evaluate the performance of six DNN models on Google TPU, Nvidia V100 GPU and Intel Skylake CPU platforms.

The neuromorphic systems are inspired by biological brain science, such as IBM's TrueNorth and Intel's Loihi. TrueNorth employs high connectivity between artificial neurons to simulate

the brain tissues. The pulse-time-dependent synaptic plasticity model (STDP) mechanism is employed by Loihi, simulating the brain's regulation of the synaptic strength by the relative time of pre-synaptic and post-synaptic pulses. These systems are still in their infancy and more suitable for long-term learning than large-scale modelling. For example, (Sharifshazileh et al., 2021) designed a neuromorphic system with a neural recording head-stage with a spiking neural network (SNN) processing core for processing intracranial Electroencephalography-EEG (iEEG) for the detection of High-Frequency Oscillations (HFO) from Brain Tissues. These systems have low power and latency compared to CPUs and GPUs and read the analogue signal directly from the head-stage.

DL compilers such as TVM (Tensor Virtual Machine), Tensor Comprehension (TC), Glow, nGraph, XLA, and FPGA specific DL code generators such as DNN Weaver, AngelEye, ALAMO, FP-DNN, SysArrayAccel, fpgaConvNet , DeepBurning, Haddoc2, and AutoCodeGen have been proposed that compile a model in a given framework to a given hardware architecture. The survey in (Li et al., 2021) provides anatomical design steps for building a DL compiler, such as the Front-end dealing with the input framework and building a NN Intermediate Representation (IR) that is passed on to the back-end that uses mature compiler toolchains targetting specific hardware such as LLVM. The back-end turns the high-level IR into a lower-level IR and uses third-party toolchain optimisation and code generation, including multiple compilation passes, memory allocation, and HW parallelisation, along with other possible features. The survey evaluates these compilers by creating 19 sample NN models (ResNet, DenseNet, VGG series, and lightweight models: MobileNet and MNASNet series) on Torchvison, and the GluonCV, then used ONNX specific relays such as tvm.relay.frontend.from_onnx for TVM, and the other corresponding relays of the other compilers. They evaluate the compilers' features' presence and the generated output model speed of execution on CPUs and GPUs on coarse-grained level (end-to-end) and fine-grained level (per-layer) performance metrics. They identified the successful compilation per Model type, compiler, and architecture and also identified the lack of compatibility of some models with some compilers. They conclude that TVM is among the best performance in several experiments. ***Research Project:*** I assume none of these is suitable for tensorised DL applications, but an extensive study should experiment with their readiness or what it needs to achieve similar compilers for Tensor decomposition and tensorised NN models

Production servers also enable better use of available CPU, GPU and hardware resources by launching multiple instances of one or more ML models as required, using a scheduler and static or dynamic batching of inference requests to reduce response latency and increase throughput. NVIDIA Triton Inference, IBM Watson Studio, MS Azure Machine Learning, Kubeflow and others offer different services to utilise in deploying ML models for inference.

Various performance metrics can be collected to help optimise an ML model configuration with profiling.

Figure 1 illustrates the technologies below Tensor structures from lower levels, highly optimised libraries, to hardware. It also shows the algorithms up to the applications and compilers that can benefit from tensor computation methods. At every level, further investigations are still an active research area with potential innovation.

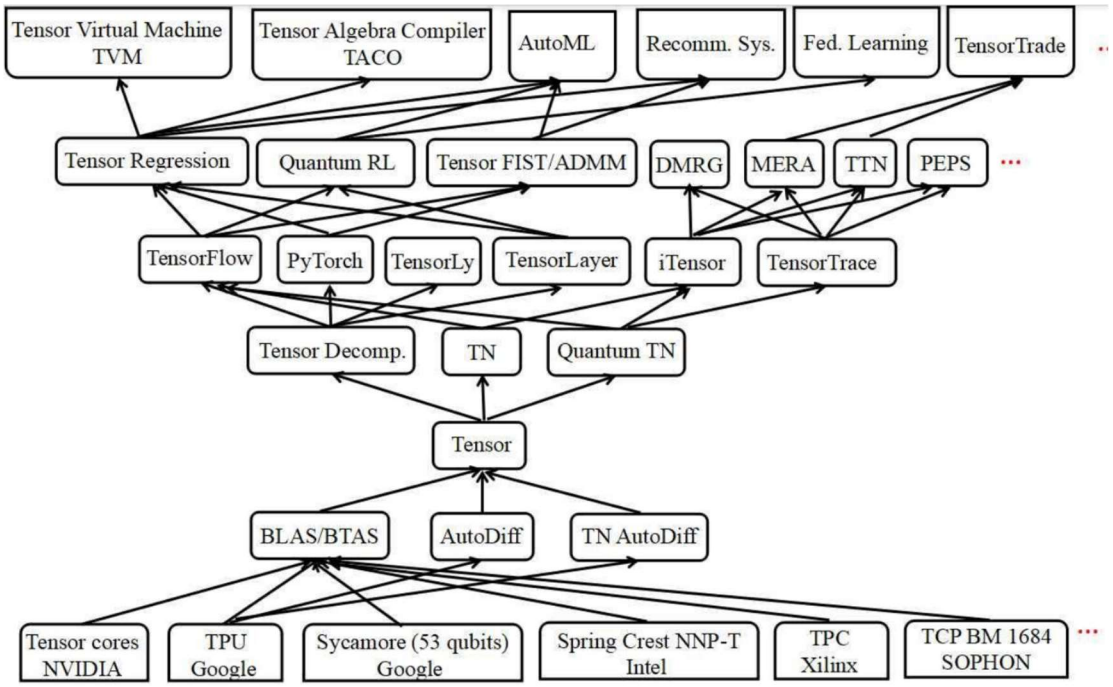

Figure 1: Tensor and Tensor Network Hourglass architecture described by (Liu et al., 2021), showing the hardware at the bottom, the applications on top, and how tensors and tensors networks are in the middle layer central to all future developments and calling for standardisation.

7.2 Challenges and Future Trends

This section summarises state-of-the-art with discussions of existing challenges and expected future trends. As discussed in chapter six, tensorizing NN takes place at the various building blocks of building the network. The number of blocks at which the tensorisation occurs, the type of tensorisation, and the collective performance evaluation is still an active area of research. We have seen examples of the tensorisation at one NN layer, multiple layers, or expressing the whole network using Tensor Networks models. There are tensorisation approaches at the forward phase of neural models (i.e. computing of the neural activation), at the backward phase (i.e. learning), and both. Also, the activation

function choice using recursive tree inputs has been attempted. ***Research Project:*** The type of tensor decomposition used, such as CP, Tucker, HT, TT, or others, needs to be further studied to evaluate the enhancement of the trade-off between compression and performance and the interpretability of different models and the different applications. Preferably, building benchmarks with clear metrics can enhance comparing the different future proposals. Multimodal DL models as well will benefit from tensorisation. Currently, large-scale trained models are merged on deployment, such as in Nvidia Deepstream https://developer.nvidia.com/deepstream-sdk. Merging the representation during training will reduce the model size, training time and deployment cost and achieve better representation and regularisation due to multi-way structure capturing. Investigating through the theory, the empirical results, and the deployment technologies can be pursued.

Chapter six briefly reviewed some work on multi-relational data analysis on single networks, on collections of structured tree samples, Graph NN and DGN, showing that the interpretation of the representation achieved better performance of the various tensorised models than the untensorized models. ***Research Projects:*** Generalising these to handle datasets of graph samples with unconstrained topology also needs to be evaluated. For example, hypergraphs connect a node to a subset of the graph rather than to other nodes only, creating hierarchical models. Systematic evaluation of the expressiveness achieved after tensorisation due to the enhanced representation would clarify the performance evaluation of the newly proposed models (Errica et al., 2020). Also, time-evolving graphs/temporal-spatial learning graphs or online learning rather than batch learning using tensorised models on graphs would be an interesting direction to enhance and evaluate. Tensorising graphs as well would be an ideal research direction for compression and expressiveness and accuracy trade-off evaluations.

As identified in the parallelisation section earlier in this chapter and from the Python code online examples mentioned in the previous chapters, it is clear that many software libraries in various programming languages, not only Python, implement various tensor methods. Merging all efforts to create a framework using similar conventions to scikit-learn and other DL models, such as TensorFlow or Keras, with some embedded systems-ready packages will promote the wide adoption of these methods and enhance their performance. A current list of implementation are as follows:

- scikit-tensor integrates some consolidated tensor decomposition (Tucker, CP and the like) into the scikit-learn universe https://github.com/mnick/scikit-tensor .
- TensorD provides a Python tensor library built on Tensorflow with basic tensor operations and decompositions with support for parallel computation (e.g. GPU).

Their code and examples are published at https://github.com/Large-Scale-Tensor-Decomposition/tensorD (Hao et al., 2018).

- TensorLy is a Python library implementing a wide range of methods for tensor learning, allowing to leverage of different computation back-ends, including NumPy, MXNet, PyTorch, TensorFlow, and CuPy. Tensorly implements CP, Tucker and TT decompositions. Their documentation is found at http://tensorly.org/stable/user_guide/tensor_decomposition.html . They also implemented Tensor Regression, and their documentation is at http://tensorly.org/stable/user_guide/tensor_regression.html. (Kossaifi et al., 2019)
- HOTTBOX is a recent standalone Python toolbox for tensor decompositions, statistical analysis, visualisation, feature extraction, regression and non-linear classification of multi-dimensional data. Their code is published with examples at https://hottbox.github.io/stable/index.html (Kisil et al., 2021).
- "scikit_tt" (https://github.com/PGelss/scikit_tt/)
- ttrecipes by (Cichocki et al., 2016, p. 1) and published at https://github.com/rballester/ttrecipes.
- "tednet" implements various neural network layer types compressed using different tensor decompositions, such as compressing an RNN layer using TR decomposition (TR_RNN). They support ResNet Layers, LSTM Layers, CNN, and Linear Layers, among others (Pan et al., 2022), https://github.com/tnbar/tednet.
- "tensortools" implements time-shifted CP decomposition (Williams et al., 2018). Their code is published at https://github.com/neurostatslab/tensortools.
- "T3F" is built on top of Tensorflow, providing Tensor Train decomposition for neural networks with Riemannian optimisation (Novikov et al., 2020). Their code is published at https://t3f.readthedocs.io/.
- Other essential websites keeping track of tensors decomposition algorithms developments include https://www.tensors.net/, http://tensornetwork.org/software/, and https://qibinzhao.github.io/.

Research Project: Continue expanding the application domain of tensorisation approaches to problems such as graph analysis such as graph clustering, and signal processing for telecommunications and biomedical sciences, such as medical imaging and signals of various modes such as MRI, EEG, ECG, hyperspectral imaging, chemical shift brain imaging and others, scientific computing problems such as excitation-emission spectroscopy, chromatography, and many more.

In chapter four, Tensortrace tool hard-codes the graphical user input into the Python code. Another possible Master's or PhD thesis project (based on the level of investigation) is to

write the code for dynamically creating these tensor shapes with their indices and networks, then the contraction dynamically for datasets. This could be an extension to the gamemaker studio application or in Python only with or without graphical interfaces. It could be to load a dataset file, identify the tensorisation requirements based on some user input from the examples in previous chapters, repeat for all required datasets, build the tensor network, and contract it into one tensor. Tensorisation so far has been tailored to particular applications in this book. Parametrising the tensorisation requirements from the dataset shape and analysis requirements to provide some automation and representation learning is an interesting project. Then test the dynamic tensor, indices, networks and contraction to some real datasets and to some quantum many-body Physical experiments such as the dataset and the benchmark in http://quantum-machine.org/datasets/.

Research Project: A constant review of the physics and math communities' advances and how to benefit from them in developing ML and DL applications is required. Other scientific computation communities are active contributors to the advancement of ML and DL approaches, such as the chemometrics and psychometric communities and others. All these communities produce their own software that is usually specified to a given problem. Computer science professionals/researchers are trained in generalising and creating frameworks using software engineering approaches that maintain backward compatibility while continuously evolving to new or enhanced features or components. A joint effort can contribute to the understanding and wide adoption of methods from one research community to another.

A pronounced research direction comes from chapter five discussion of representation theory in relation to abstract algebra and group theoretic frameworks. Many group theory implementations, such as GAP, are not ready for higher-scale calls and wrappers from other programming languages to build further machine learning and deep learning models on top of them.

References

Cichocki, A., Lee, N., Oseledets, I.V., Phan, A.-H., Zhao, Q., Mandic, D., 2016. Low-Rank Tensor Networks for Dimensionality Reduction and Large-Scale Optimisation Problems: Perspectives and Challenges PART 1. FNT in Machine Learning 9, 249–429. https://doi.org/10.1561/2200000059

Engel, J., Hantrakul, L., Gu, C., Roberts, A., 2020. DDSP: Differentiable Digital Signal Processing. Presented at the International Conference on Learning Representations, arXiv.

Errica, F., Bacciu, D., Micheli, A., 2020. Theoretically Expressive and Edge-aware Graph Learning.

Geijn, R.A. van de, Quintana-Ort´, E.S., 2008. The Science of Programming Matrix Computations. www.lulu.com.

Hao, L., Liang, S., Ye, J., Xu, Z., 2018. TensorD: A tensor decomposition library in TensorFlow. Neurocomputing 318, 196–200. https://doi.org/10.1016/j.neucom.2018.08.055

Helal, M., El-Gindy, H., Mullin, L., Gaeta, B., 2008. Parallelising Optimal Multiple Sequence Alignment by Dynamic Programming. IEEE, pp. 669–674. https://doi.org/10.1109/ISPA.2008.93

Helal, M., Mullin, L., Potter, J., Sintchenko, V., 2009. Search Space Reduction Technique for Distributed Multiple Sequence Alignment. IEEE, pp. 219–226. https://doi.org/10.1109/NPC.2009.43

Kisil, I., Calvi, G.G., Dees, BS, Mandic, DP, 2021. HOTTBOX: Higher Order Tensor ToolBOX.

Kossaifi, J., Panagakis, Y., Anandkumar, A., Pantic, M., 2019. TensorLy: Tensor Learning in Python. Journal of Machine Learning Research 20, 1–6.

Li, M., Liu, Y., Liu, X., Sun, Q., You, X., Yang, H., Luan, Z., Gan, L., Yang, G., Qian, D., 2021. The Deep Learning Compiler: A Comprehensive Survey. IEEE Trans. Parallel Distrib. Syst. 32, 708–727. https://doi.org/10.1109/TPDS.2020.3030548

Liu, X.-Y., Wang, X., 2017. Fourth-order Tensors with Multi-dimensional Discrete Transforms.

Liu, X.-Y., Zhao, Q., Walid, A., 2021. Tensor and Tensor Networks for Machine Learning: An Hourglass Architecture. Presented at the International Workshop on Tensor Network Representations in Machine Learning, Japan, p. 7.

Novikov, A., Izmailov, P., Khrulkov, V., Figurnov, M., Oseledets, I., 2020. Tensor Train Decomposition on TensorFlow (T3F). Journal of Machine Learning Research 21, 1–7.

Pan, Y., Wang, M., Xu, Z., 2022. TedNet: A Pytorch Toolkit for Tensor Decomposition Networks. Neurocomputing 469, 234–238. https://doi.org/10.1016/j.neucom.2021.10.064

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., Lerer, A., 2017. Automatic differentiation in PyTorch. Presented at the 31st Conference on Neural Information Processing Systems, CA, USA, p. 4.

Sharifshazileh, M., Burelo, K., Sarnthein, J., Indiveri, G., 2021. An electronic neuromorphic system for real-time detection of high frequency oscillations (HFO) in intracranial EEG. Nat Commun 12, 3095. https://doi.org/10.1038/s41467-021-23342-2

Williams, A.H., Kim, T.H., Wang, F., Vyas, S., Ryu, S.I., Shenoy, K.V., Schnitzer, M., Kolda, T.G., Ganguli, S., 2018. Unsupervised Discovery of Demixed, Low-Dimensional Neural Dynamics across Multiple Timescales through Tensor Component Analysis. Neuron 98, 1099-1115.e8. https://doi.org/10.1016/j.neuron.2018.05.015

Zaccone, G., 2015. Python parallel programming cookbook: master efficient parallel programming to build powerful applications using Python, 1. publ. ed, Quick answers to common problems. Packt Publ, Birmingham Mumbai.