Minimizing overhead in distributed computing: Application for expensive optimization problem

Teppo Tammisto
Techila Technologies Ltd

Juho Kanniainen
Tampere University of Technology

2019-07-05

1 Introduction

Numerical optimization has a central role in many fields of applied mathematics ranging from quantitative finance to control theory. Sometimes the loss function –an objective to be minimized– can be computationally expensive, which calls need for the use of high-performance computing (HPC) to speed up the optimization. This document describes how we can innovatively accelerate the optimization in a case where the evaluation of the loss function can be parallelized into independent sub-tasks. By exploiting *Techila Distributed Computing Engine middleware*, *we propose a solution to speed-up an optimization procedure when the loss-function is expensive and its calculation can be distributed.*

The use of distributed computing for iterative problems, such as for optimization, is not very straightforward. Particularly, existing algorithmic approaches can suffer from iterative information transfers that take place between the computational environment, such as a cloud or private grid, and end-user workstation. Hereafter, we use 'workstation' to refer to a computer where the end-user is logged in in general. Respectively, we use 'cloud' to refer to a computational environment in general.

This communication *between* the cloud and the workstation can be relatively slow, having higher latency compared to internal communication between the computer nodes within the cloud. Additionally, existing algorithmic approaches may need to create a new subtask for each optimization iteration, and consequently, computer nodes in the cloud need to be reactivated and deactivated in each optimization iteration, which can cause significant overhead. The third drawback of these existing algorithmic approaches is that in multi-user environments, resources can be occupied by another user between the optimization iterations, meaning that an optimization task can lose its place in the queue between the iterations.

In our new algorithmic approach, all operations related to the optimization process are performed *within* the cloud. The principle of the design approach is to:

i. offload the optimization routine from the workstation to the cloud,
ii. offload the loss-functions to the cloud,
iii. connect the loss-functions to the optimization routine while
iv. providing feedback from the optimization routines progress to the end-user

This is in contrast with existing approaches where the optimization routine is managed by the individual end-user's workstation, which is physically outside of the cloud, thus causing unnecessary overhead and delay. The goal of our proposed approach is to i) minimize the overhead from data transfer between nodes in the cloud environment and the end-user workstation and ii) minimize the overhead caused by iterative initializations and finalizations of computing nodes in every optimization iteration.

This paper discussed how to use Techila Distributed Computing Engine (TDCE) technology to implement the new algorithmic approach to distribute computationally expensive optimization problems into the cloud, and benchmarks the performance of the proposed design and the current design.

TDCE is a software solution that is developed by Techila Technologies Ltd. The technology is based on a patented Autonomic Computing architecture. It enables scalable interactive HPC runs in a way that adheres to the proposed design approach, providing functionality to:

i) use programming language specific APIs to push workloads to the cloud,
ii) automatically install and configure required software on the cloud,
iii) transfer data directly between computer nodes within the cloud and
iv) display interactive feedback information from the optimization routines to the end-user

2 Architecture of the Techila Environment in the Distributed Procedures

Figure 1 illustrates the architecture of Techila Distributed Computing Engine (TDCE) that consists of three logical main components:

i) *Techila Server* – Manages the computing environment and acts as the communication endpoint for the end-user. Also delivers the subtasks received from the end-user to the Techila Workers.

ii) *Techila Workers* – Computer nodes in the cloud (or another public or private computational environment), which provide computational capacity to calculate the optimization tasks. These nodes can communicate with each other within the cloud to share and transfer computational data.

iii) *Techila SDK* – Software Development Kit that is installed on the end-user's workstation, providing programming language specific APIs that can be used to push computational workloads to the cloud.

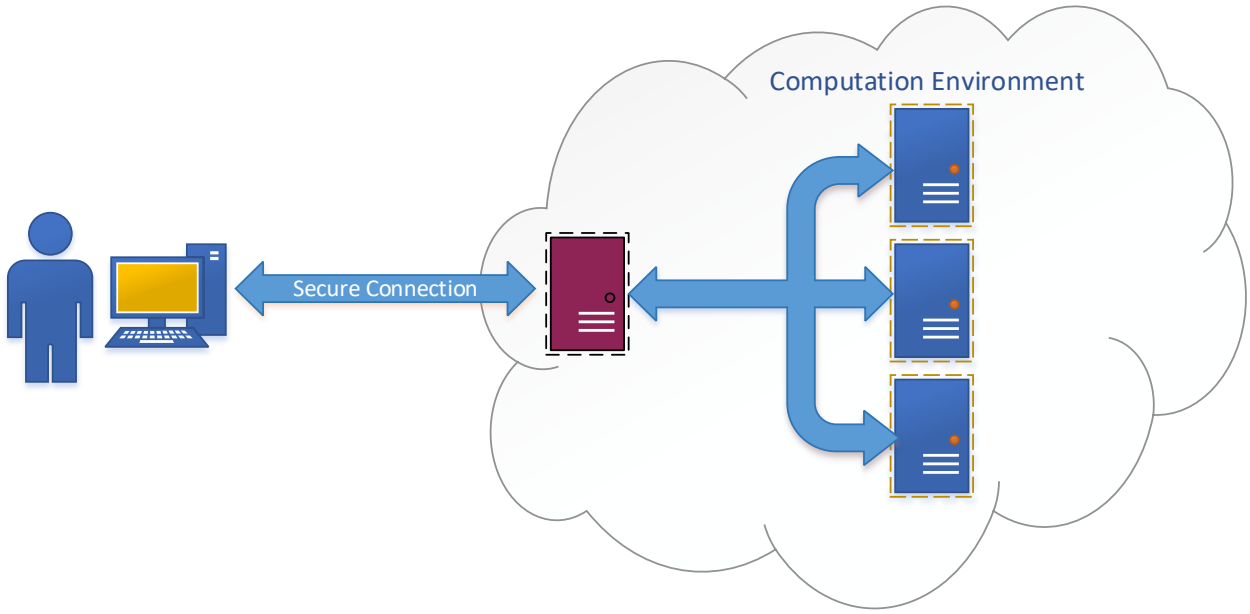

Figure 1: Computation environment with Techila. TDCE offers built-in security between all system components (TLS, PKI).

The system design of TDCE is built around a highly modular service-oriented architecture (SOA) with module lifecycle management. The solution has a Techila Worker-centric solution architecture with autonomic management to support scalability and fault tolerance in loosely coupled distributed operating environments.

3 Optimization Problem with Distributed Computing

In this section, we demonstrate both the existing approaches and our proposed approach to solve an optimization problem using distributed computing. We are focusing on a problem, where the loss-function is computationally expensive, but can be solved using distributed computing. Generally, this framework is appropriate for any loss function that requires numerical methods that can be parallelized, such as Monte Carlo methods or series expansion methods. In finance, such loss-functions can be found, for example, in the calibration of financial option pricing models (see Christoffersen and Jacobs 2004a, 2004b, Kanniainen and Piché 2012, Kanniainen et al. 2014, and Yang and Kanniainen 2016).

We consider an optimization problem with a loss-function of the form of

$$f(\boldsymbol{x}) = (g(\boldsymbol{x}) - \hat{g})^2,$$

where $g(\boldsymbol{x})$ is a function of parameters $\boldsymbol{x} \in \mathbb{R}^{\boldsymbol{k}\times\boldsymbol{1}}$ and $\hat{g}$ represents given observations. $g(\boldsymbol{x})$ and $\hat{g}$ can be understood as model solution and correct (observable) solution, respectively, and the objective of the optimization problem is to find (model) parameters $\boldsymbol{x}$ that minimize the error. Our distribution approach is suitable when $g(\boldsymbol{x})$ is computationally expensive and can be evaluated in a distributed way. As an example, we calibrate an option pricing model against actual option market quotes. Under our stochastic model, the pricing of options requires the use of Monte-Carlo methods and, more importantly, the Monte Carlo paths can be distributed into subtasks to be run independently on separate computer nodes. The proposed approach can be used for other kind of problems, too, as long as

- The computation of a loss-function is an embarrassingly parallel problem (for embarrassingly parallel problems, see, for example, Wilkinson and Allen 1999), were it based on Monte-Carlo or other numerical methods.
- An optimization algorithm is iterative, such as Nelder-Mead algorithm (see Lagarias et al. 1998) or Gradient Methods (for a review of Gradient Methods, see Ruder 2016).

Our example provided in Section 4 shows that the overhead is quite remarkable in terms of both wall-clock and CPU times. Next, we will demonstrate the optimization procedures under the following conditions:

i) Optimization without distribution.

ii) Optimization with distribution so that the optimization is managed by the end-user workstation and nodes are initialized and finalized in each optimization iteration. This is called Approach A. This approach is employed for example in (Kanniainen and Piché 2013, Kanniainen et al. 2014)

iii) Optimization with a distribution approach, where the optimization is managed by one of the nodes in the cloud and all information transfer happens between nodes within the cloud, and the nodes are initialized and finalized only once (at the beginning and end of the whole optimization project). This is called Approach B.

3.1 Optimization without distribution

Figure 2 demonstrates the standard optimization procedure on a single local workstation without distributed computing. The optimization procedure contains of M optimization iterations at the most. In each optimization iteration, loss-function value $f(\boldsymbol{x}) = (g(\boldsymbol{x}) - \hat{g})^2$ is computed using numerical methods, say, obtained with N Monte-Carlo iterations.

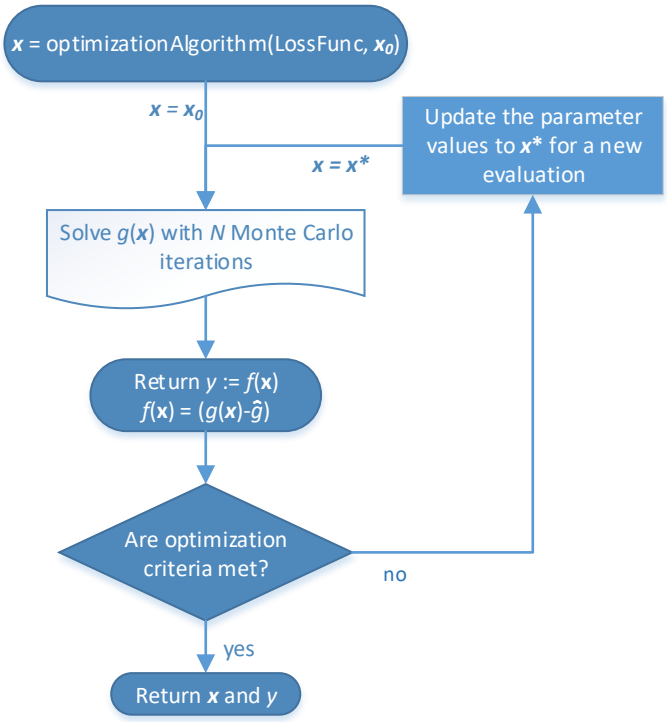

Figure 2. Optimization without distribution.

Particularly, the steps are following:

1. *Evaluation:* For given initial parameter values, $\boldsymbol{x} \leftarrow \boldsymbol{x_0}$, solve $g(\boldsymbol{x})$ (with N Monte-Carlo iterations) and calculate the corresponding loss-function value $f(\boldsymbol{x}) = (g(\boldsymbol{x}) - \hat{g})^2$.
2. *Criteria check:* If optimization criteria are met, the optimization procedure is completed.
3. *Optimization:* If given criteria are not met, update parameter values, $\boldsymbol{x} \leftarrow \boldsymbol{x_i}$, where i refers to ith optimization iteration.
4. *Re*-evaluation: Calculate a new value for the loss function (with Monte-Carlo).
5. Repeat 3-4 until the criteria are met or until M optimization iterations are performed.[1]

Overall, the procedure is just about sequential evaluation of the loss-function that is performed with a single workstation only.

3.2 Optimization with distributed computing: Approach A (optimization managed by end-user's workstation)

Figure 3 demonstrates the optimization procedure managed by end-user's workstation utilizing a distributed computing in the cloud. Here it is the End-user's workstation that runs the iterative optimization algorithm and sends the subtasks to the computer nodes in the cloud between each optimization iteration. Because the optimization algorithm is iterative, the loss-function is evaluated sequentially, and more importantly, each loss-function evaluation is considered as a separate project in terms of distribution. Particularly, at the beginning of each optimization iteration, N Monte Carlo paths are distributed for n nodes. Essentially, in this approach, the *nodes are reactivated at the beginning of each optimization iteration and deactivated at the end of each iteration*. Reactivation and deactivation are called as initialization and finalization. Also, there is information transfer between the end-user's workstation and the nodes in a cloud at the beginning

[1] Different optimization algorithms have different strategies to update the parameters, which is not the focus of this document.

and end of each optimization iteration. Both iterative initializations and finalizations and information transfer between local end-user's workstation and cloud require extra computing capacity and time, which increases the overhead in distributed computing.

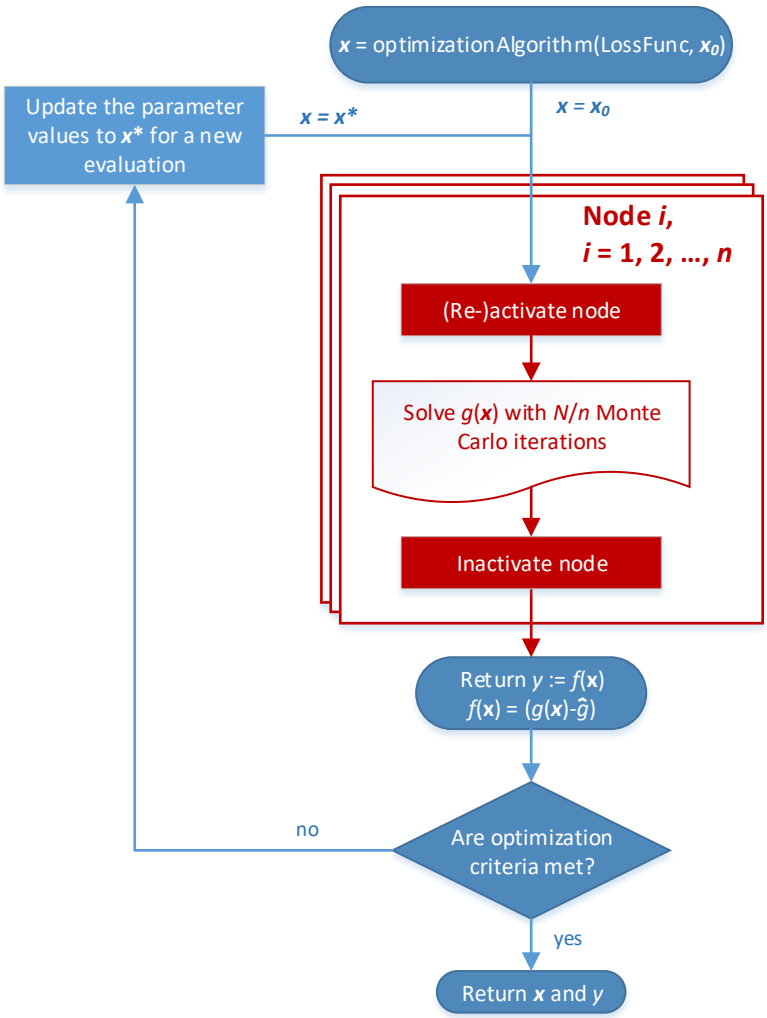

Figure 3. Optimization with Approach A. Activities on a local End-User's workstation are indicated by blue color and activities performed in the cloud by red color.

More specifically, the steps are following:

1. *Evaluation*
    a. *Initialization*: Activate n nodes (for simplicity, each node for one subtask)
    b. Distribute N Monte-Carlo iterations into n subtasks.
    c. For given initial parameter values, $\boldsymbol{x} \leftarrow \boldsymbol{x_0}$, solve $g(\boldsymbol{x})$ with N/n Monte-Carlo iterations in each node.
    d. Aggregate results to obtain the value of the loss function, $f(\boldsymbol{x}) = (g(\boldsymbol{x}) - \hat{g})^2$.
    e. *Finalization*: Inactivate all the nodes.
2. *Criteria check:* If the criteria are met, the optimization procedure is completed. Stop.
3. *Optimization:* If given criteria are not met, update parameter values, $\boldsymbol{x} \leftarrow \boldsymbol{x_i}$, where i refers to ith optimization iteration.
4. *Re*-evaluation: Calculate a new value for the loss function with new parameter values according to step 1.
5. Repeat 3-4 until the criteria are met or until *M* optimization iterations are performed.

As the above steps and Figure 3 illustrate, the distributed implementation has additional sub-steps under the loss-function evaluation step. Particularly, the evaluation of a loss-function value is distributed across n nodes that are first activated. That is, N Monte-Carlo iterations are distributed into n independent subtasks such that one node carries out one subtask.[2] End-user's workstation delivers information about subtasks to the nodes via the server and once the realizations of each subtask are available, the results are aggregated and the evaluated value of the loss-function is sent to the end-user workstation that runs the optimization algorithm. Finally, at the end of the evaluation step, nodes are deactivated. Once the optimization algorithm changes parameter values, the nodes are reactivated and the procedure of the evaluation step described above is applied again. This procedure is repeated as long as optimization criteria are met or optimization is stopped for another reason.

3.3 Optimization with distributed computing: Approach B (optimization managed within cloud)

The distribution scheme in Approach A described in Figure 3 is problematic especially because:

- There is information transfer between nodes and end-user workstation before and after each optimization iteration.
- New jobs are created and nodes are reactivated and deactivated for each optimization iteration.
- In multi-user environments, resources can be occupied by another user between then optimization iterations.

Figure 3 elaborates the idea of a new way to implement optimization procedures with distributed computing with Approach B in Techila environment. The steps are the following:

1. *Initialization*: Activate *n* nodes (for simplicity, one subtask for each node)
    a. The first node, so-called Master Node, is dedicated for
        i. the management of the optimization algorithm,
        ii. the management of the distribution of subtasks to other nodes,
        iii. the computation of the N/n realizations of $g(\boldsymbol{x})$ for one subtask,
        iv. the aggregation of results from other nodes to obtain the value of $f(\boldsymbol{x}) = (g(\boldsymbol{x}) - \hat{g})^2$.
    b. The other $n-1$ nodes, so-called co-nodes, are dedicated for
        i. the computation of the N/n realizations of $g(\boldsymbol{x})$ for each subtask,
        ii. the sending of the results to the master node.
2. *Evaluation*:
    a. **Master node:** Distribute N Monte-Carlo iterations into n subtasks and send $n-1$ subtasks to other nodes.
    b. **Master node:** For given initial parameter values, $\boldsymbol{x} \leftarrow \boldsymbol{x_0}$, solve $g(\boldsymbol{x})$ with N/n Monte-Carlo iterations.
    c. **Co-nodes:**

[2] In fact, one node could carry out several subtasks, but for simplicity, we assume that one there is one subtask for each node.

i. In each co-node, for given initial parameter values, $\boldsymbol{x} \leftarrow \boldsymbol{x_0}$, solve $g(\boldsymbol{x})$ with N/n Monte-Carlo iterations.
ii. Wait until the next task

d. **Master node:** When all results are completed by all the nodes, aggregate the results from other nodes and itself to obtain the value of $f(\boldsymbol{x})$. In Figure 3, this is demonstrated by a line from "return <g>" to "average of <g>".

3. *Criteria check with the* **master node**: If the criteria are met, the optimization procedure is completed. Go to step 7.
4. *Optimization with the* **master node**: If given criteria are not met, update parameter values, $\boldsymbol{x} \leftarrow \boldsymbol{x_i}$, where i refers to ith optimization iteration.
5. *Re*-evaluation *with **all the*** **nodes, managed by the master node**: Calculate the new value of the loss function with new parameter values according to step 2.
6. Repeat 3-5 until the criteria are met or until M optimization iterations are performed.
7. *Finalization*: Deactivate all the nodes.

Also in this this approach, N Monte-Carlo iterations are distributed into n independent subtasks such that one node carries out one subtask. However, the distribution, and in fact, the overall optimization procedure, is managed by the master node *within the cloud*, instead of end-user's workstation. End-user's workstation delivers information about the optimization problem only at the beginning of the project and then directly receives the final results at the end of the project. The optimization procedure has the same principle as before, but now there is no need to reactivate nodes in the course of the optimization procedure.

There are clear advantages for Approach B over Approach A. First, optimization algorithm is managed by the master node within the same computation environment (cloud) with other nodes, which leads to faster information transfer. In Approach A, the optimization algorithm is run on the end-user workstation and therefore there is need for communication between the end-user workstation and computation environment between each optimization iteration. Second, the frequency of communication is drastically lower. Third, in the proposed approach, the initialization and finalization are done once, just at the beginning and at the end of the whole optimization procedure, whereas in the existing approach, initialization and finalization are done in each optimization iteration. Each initialization takes wall-clock time but also uses CPUs. Therefore, Approach B leads not only faster but also cheaper results.

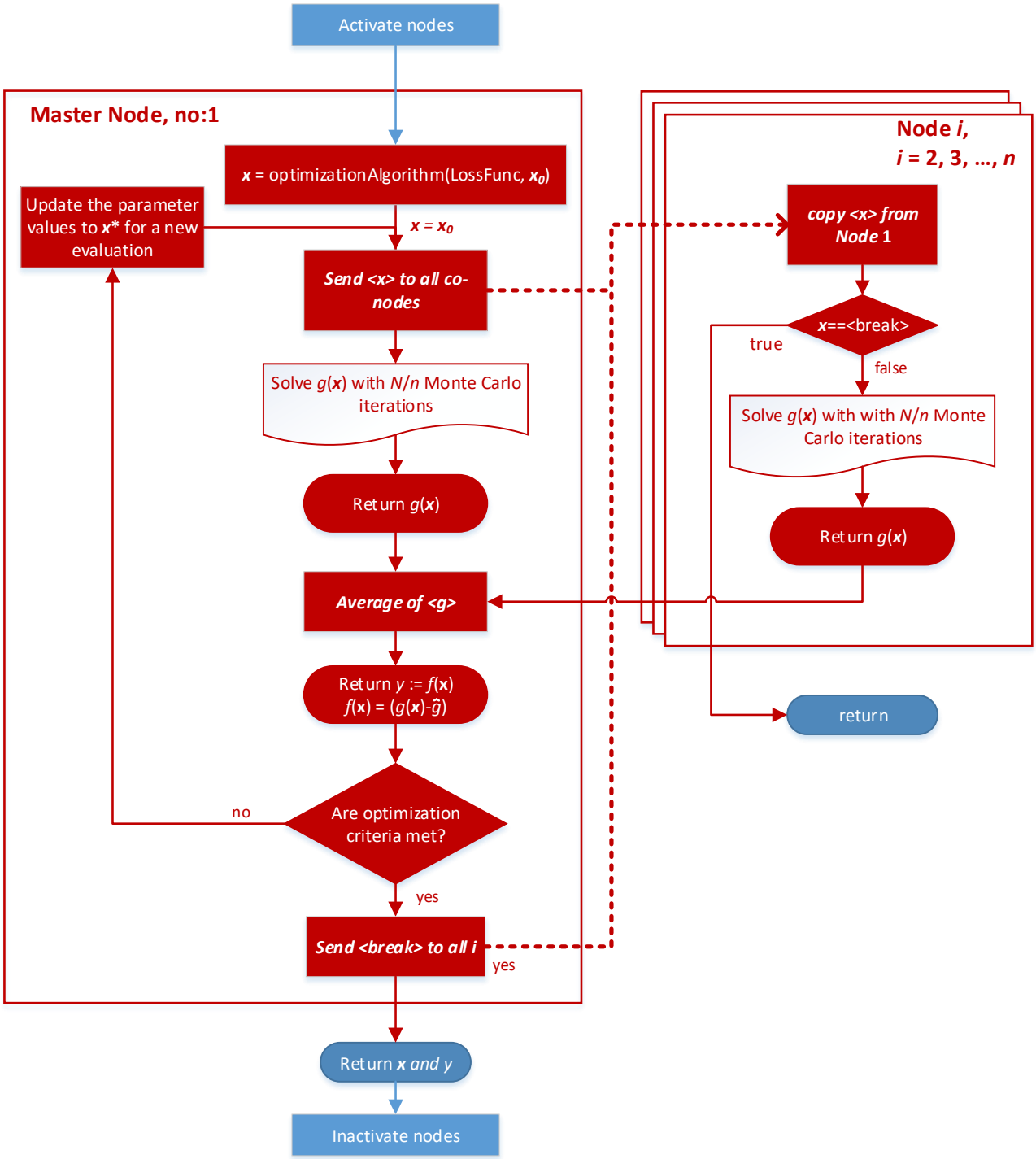

Figure 4: Optimization with Approach B using Techila middleware. Dashed lines represent information transfer between the master node and the co-nodes. Activities on a local End-User's workstation are indicated by blue color and activities performed in the cloud by red color.

4 Case: Calibration of option pricing model

To demonstrate the computational performance of the suggested distribution procedure with Approach B against Approach A, we calibrate a stochastic volatility option pricing model by minimizing error between model's implied volatilities and implied volatilities observed at option markets.

4.1 Data

The data is an empirical snapshot of S&P 500 index options on the 20th of January, 2006, at 12:00pm. Particularly, in the calibration exercise, we use an implied volatility surface that is fitted for observed implied volatilities out-of-the-money put and call option mid-prices. The range of moneyness (the strike price divided by the current index value) is 0.5 and 1.5 (consequently, the range of the log-moneyness is -0.7 and 0.4). The maturity time is from 0.25 to 2 years. The resulting implied volatility surface is demonstrated in Figure 4.

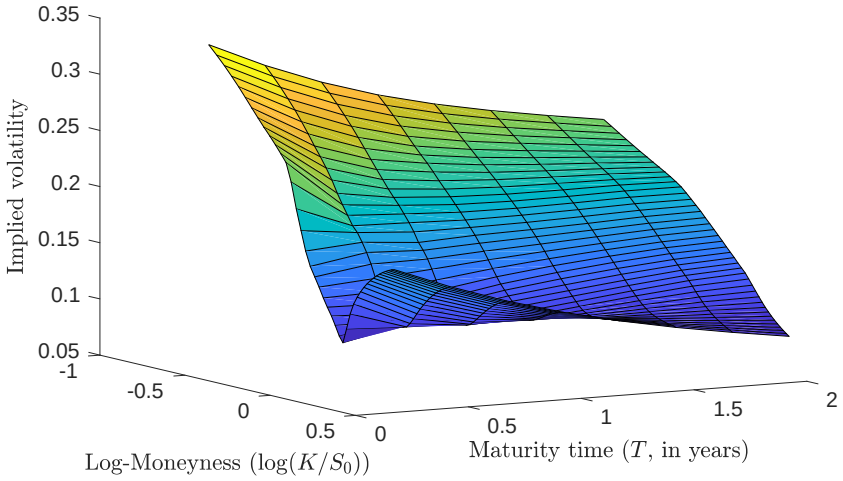

Figure 4: Empirical implied volatility surface for S&P 500 index options, which is used for option pricing model calibration. The time of the observation of mid-prices of out-of-the-money call and put options is the 20th of January, 2006, 12:00pm.

4.2 Option Pricing Model

We calibrate a model that can be seen as a non-affine generalization of Heston model (see Heston 1993, Yang and Kanniainen 2017). Let $\{S_t; t \geq 0\}$ denote the price of an underlying asset (say, S&P 500) and $\{V_t; t \geq 0\}$ denote the instantaneous squared volatility of the returns. Under the risk neutral measure, we assume that the return and volatility dynamics are described by the following stochastic differential equations:

$$d\log(S_t) = \left(r - \frac{1}{2}V_t\right)dt + \sqrt{V_t}dW_{Y,t}, \quad (1.a)$$

$$dV_t = \kappa(\theta - V_t)dt + \xi V_t^{\gamma}\left(\rho dW_{Y,t} + \sqrt{1-\rho^2}dW_{V,t}\right), \quad (1.b)$$

where W_Y and W_V are mutually independent Brownian motions and $S_0 = s_0 > 0$ and $V_0 = v_0 > 0$. In addition, in the volatility process $\kappa > 0$ is the speed of the mean-revision, $\theta > 0$ is long-run variance, $\xi > 0$ is the volatility of volatility, $\rho \in [-1,1]$ the correlation between returns and volatility, and $\gamma > 0$ the coefficient to make the model affine ($\gamma = 0.5$) or non-affine (otherwise). Finally, r is the constant risk-free interest rate.

The extant literature provides strong empirical evidence that non-affine models affine ($\gamma \neq 0.5$) outperform affine ones ($\gamma = 0.5$) (Christoffersen et al., 2010; Kaeck and Alexander, 2012; Kanniainen et al., 2014; Yang and Kanniainen, 2016). On the other hand, non-affine models do not generally have numerically efficient solutions (e.g. semi-closed form FFT) and practically Monte-Carlo methods must be used.

4.3 Calibration Settings

The optimization problem is to find model parameters $\Theta = \{\kappa, \theta, \xi, \rho, \gamma\}$ and spot volatility v_0 by minimizing the squared error between implied volatility yielded by the model and observed from markets:

$$\text{IV-RMSE}(\Theta, v_0) = 100 \times \sqrt{\frac{1}{m}\sum_i \left[\text{IV}_i(\Theta, v_0) - \widehat{\text{IV}}_i\right]^2},$$

where $\text{IV}_i(\Theta, v_0)$ is the "model implied volatility" of ith option given by the model (Equations 1.a and 1.b) with parameters Θ and spot volatility v_0. Correspondingly, $\widehat{\text{IV}}_i$ is the "market implied volatility" for ith option observed in the implied volatility surface constructed using the market data. Moreover, m is the number of implied volatility observations used for the calibration.

In this paper, the optimization is based on MATLAB's `fminsearch` function, provided in the optimization toolbox, to minimize IVRMSE. `fminsearch` uses a derivative-free Nelder-Mead simplex algorithm (Lagarias et al., 1998). Because the method itself is unconstrained, we crafted constrains in such a way that the loss function (Eq. 2) gets very high values (1e10) if the parameters or spot variance are out of the given boundaries.

4.4 Monte-Carlo Settings

In Monte-Carlo pricing, we use three variance reduction methods: (i) Black-Scholes price serves as a control variate (see e.g. Glasserman, 2013, Ch. 4.1); (ii) antithetic variates (see e.g. Glasserman, 2013, Ch. 4.2) are used for both return and volatility processes; and (iii) we implement Empirical Martingale Simulation method, introduced by Duan and Simonato (1998). The Monte-Carlo simulation is implemented efficiently so that one can compute a cross-section of options with different strike prices by one run, and therefore, the range of the moneyness in the volatility surface does not essentially affect the computation time. The most important factor is maturity time; if fact, the paths are simulated according to the longest maturity time, and therefore, if a single option contract with a long maturity time can increase the computational time considerably.

The prices of at-the-money options can be accurately solved with Monte-Carlo methods with a reasonable low number of Monte-Carlo iterations, say with 10,000 iterations. However, the pricing of deep-out-of-the money calls (and deep-in-the-money puts) can require a huge number of iterations, even if all the three variance methods are exploited. This is demonstrated in Figure 5, which is based on 10,000 iterations; for short-term and low-moneyness (high K/S_0) options, implied volatilities are not always available. This is quite intuitive, because a call option is priced at zero if no any of the Monte-Carlo paths of the underlying asset hits the level of the strike price. To price these options correctly, in our experience, one may need even Monte-Carlo 1 million paths, which is computationally quite expensive. On the other hand, deep-out-of-the-money calls (deep-in-the-money puts) can provide very important information for model calibration, and therefore they should not be excluded from the model calibration data set. Consequently, we calibrate the model (Eq 1.a, 1.b) using 1 million iterations, which, at the same time, makes the exercise as a good case study as we have an optimization problem with a computationally expensive loss-function.

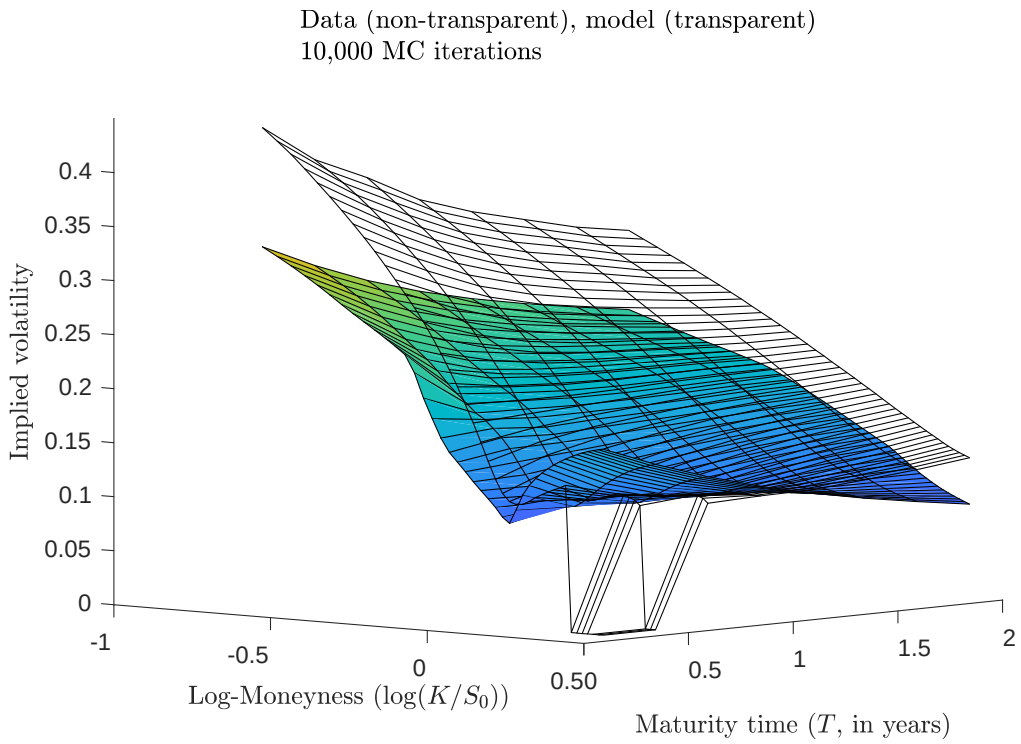

Figure 5: Empirical implied volatility surface and model-based surface with 10,000 Monte-Carlo iterations. Market implied volatilities are represented with the colored (non-transparent) surface and model-based implied volatilities with the transparent surface. The figure shows that 10,000 Monte-Carlo paths is not necessarily enough for the pricing of deep-out-of-the-money calls (deep-in-the-money puts).

4.5 Infrastructure and distribution settings

We use 10 nodes (instances) on Azure Cloud, which are type of D2v3. Nodes have 8 GiB of memory and 2 vCPUs. Dv3-series sizes are based on the 2.3 GHz Intel XEON ® E5-2673 v4 (Broadwell) processor and can achieve 3.5GHz with Intel Turbo Boost Technology 2.0.[3] We use Windows server 2012 R2 and Matlab R2016b.

We divide the generation of 1,000,000 Monte Carlo paths into 10 subtasks. Therefore, each node generates 100,000 paths that are then summed up to get the final price. The seed number of a random number generator used in a Monte Carlo simulation sub-task is equal to the number of node (1..10). This strategy yields identical option pricing results if the model parameter values are not changed, an important property in model calibration. Also, this allows us to achieve identical results, regardless of whether the calibration is done with Approaches A or B. When the model is calibrated without distribution, then we split the problem into 10 subtasks and use the same seed numbers as with distributed experiment, but with a single node only. In this way, we can ensure that the number of optimization iterations is the same independently whether the problem is distributed or not.

4.6 Results

We computed results under three experiments that were implemented according to Sections 3.1, 3.2, and 3.3 using the infrastructure described in 4.5. In all of the experiments, the random number generators were seeded using predetermined values, meaning the results are identical. Table 1 shows the initial and final parameter values and the value of the loss-function before and after optimization.

[3] https://docs.microsoft.com/en-us/azure/virtual-machines/windows/sizes-general#dv3-series

Table 1. Initial and final parameter values and the value of the loss-function before and after optimization.

Parameter	Initial value	Final value
κ	3	2.3579
θ	0.04	0.0209
ξ	1	1.1839
ρ	-0.8	-0.5945
γ	0.5	0.7609
V_0	0.0324	0.0255
IV-RMSE	5.3509	1.0308

In the calibration of the model, there were 507 optimization iterations until criteria were met. The calibration times of three experiments are reported in Table 2. First, we observe that the wall-clock time can be clearly shortened by distributed computing: Approach A reduces wall-clock time by 80% and Approach B by 86% in comparison to the experiment without distribution. More importantly, in the terms of wall-clock time, *approach B is 33% more efficient than Approach A*, a considerably high value. In other words, by minimizing the overhead caused by i) information transfer between the end-user workstation and cloud and ii) iterative initializations and finalizations of computing nodes, we can save one third of wall-clock time.

Table 2. Results on calibration times under three experiments: i) no distribution, ii) optimization with distributed computing, so that the optimization is managed by the end-user workstation and nodes are initialized and finalized between in every optimization iteration Approach A, and iii) optimization with Approach B, where optimization is managed by one of the nodes in the cloud, all the information transfer happens between nodes within a cloud, and the nodes are initialized and finalized only once.

	Wall-clock time	CPU time
No distribution	1d04h18m	1d04h12m
Approach A	5h44m	1d16h43m
Approach B	3h51m	1d04h49m

CPU time is, by construction, the lowest for the experiment without no distribution. However, Approach B increased CPU time only by 2.2% whereas Approach B increased it by 44%! In other words, if the overhead is defined as a difference between CPU with and without distributed computing, the overhead is 2.2% with Approach B and 44% with the old one. Consequently, the overhead of approach A is 20 times higher compared to Approach B. Given that charge for the cloud service is based on CPU time, the computing costs were almost the same with distribution Approach B and the experiment without distribution, whereas the implementation of the distribution scheme with Approach A increased the costs substantially.

5 Discussion

In this paper, we introduced a new approach for the distribution of computationally expensive iterative optimization problems. In this approach, the optimization algorithm is managed by one of the nodes in the cloud and hence the communication between the end-user's workstation and the cloud is minimized. The calibration of an option pricing model served as a test-case. Based on our experiment, in the terms of wall-clock time, the approach where the optimization is managed in the cloud is 33% more efficient compared that the optimization is managed by the end-user's workstation. Moreover, the overhead of the later approach is 20 times higher compared to the approach with managing optimization in the cloud. Therefore, our approach does not only make the computations faster but also cheaper as cloud provider typically charge based on CPU hours.

Given that the cloud infrastructure includes extremely fast interconnects between the cloud nodes, it is safe to assume that the proposed approach (Approach B) can also readily be applied in use cases where the size of data is large and required data transfer is non-trivial. Finally, a word about computational environments. In principle, the proposed approach could be employed on various environments, such as public clouds or private grids. However, the proposed approach requires a stabile computing environment so that all the subtasks can be run simultaneously so that all the nodes keep running. Therefore, we consider that robust cloud environments are the most suitable platforms for this distribution approach.

6 References

Christoffersen, P. and K. Jacobs (2004), "Which GARCH model for option valuation?" *Management Science* 50, 1204-1221.

Christoffersen, P. and K. Jacobs (2004), "The importance of the loss function in option valuation", *Journal of Financial Economics* 72, 291-318.

Christoffersen, P., K. Jacobs, and K. Mimouni (2010), "Volatility dynamics for the S&P500: evidence from realized volatility daily returns, and option prices", *Review of Financial Studies* 23, 3141–3189.

Duan, J.C., Simonato, J.G. (1998), "Empirical martingale simulation for asset prices", *Management Science* 44, 1218–1233.

Glasserman, P., 2013. Monte Carlo methods in financial engineering. Volume 53. Springer Science & Business Media.

Heston, S. L. (1993). "A closed-form solution for options with stochastic volatility with applications to bond and currency options", *Review of financial studies* 6, 327–343.

Kaeck, A., C. Alexander (2012), "Volatility dynamics for the S&P 500: Further evidence from non-affine, multi-factor jump diffusions", *Journal of Banking & Finance* 36, 3110–3121.

Kanniainen, J., Binghuan L., and H. Yang (2014), "Estimating and using GARCH models with VIX data for option valuation", *Journal of Banking & Finance* 43, 200-211.

Kanniainen, J., and R. Piché (2013), "Stock price dynamics and option valuations under volatility feedback effect", *Physica A: Statistical Mechanics and its Applications* 392, 722-740.

Lagarias, J.C., Reeds, J.A., Wright, M.H., Wright, P.E. (1998), "Convergence properties of the nelder–mead simplex method in low dimensions", *SIAM Journal on optimization* 9, 112–147.

Ruder, S. (2016), "An overview of gradient descent optimization algorithms", *arXiv preprint arXiv:1609.04747*.

Wilkinson, B., and M. Allen. *Parallel programming: techniques and applications using networked workstations and parallel computers*. Prentice-Hall, 1999.

Yang, H. and J. Kanniainen (2017), "Jump and Volatility Dynamics for the S&P 500: Evidence for Infinite-Activity Jumps with Non-Affine Volatility Dynamics from Stock and Option Markets", *Review of Finance* 21, 811-844. https://doi.org/10.1093/rof/rfw001