Future Generation Computer Systems 148 (2023) 623–635

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Profiling and optimization of Python-based social sciences applications on HPC systems by means of task and data parallelism

Lukasz Szustak [a,b,*], Marcin Lawenda [b], Sebastian Arming [c], Gregor Bankhamer [c], Christoph Schweimer [d], Robert Elsässer [c]

[a] Czestochowa University of Technology, Dabrowskiego 69, 42-201 Czestochowa, Poland
[b] Poznan Supercomputing and Networking Center, Jana Pawła II 10, 61-139 Poznan, Poland
[c] Paris Lodron University of Salzburg PLUS, Jakob-Haringer-Straße 2, 5020 Salzburg, Austria
[d] Know-Center GmbH, Inffeldgasse 13/6, 8010 Graz, Austria

ARTICLE INFO

Article history:
Received 28 February 2023
Received in revised form 12 May 2023
Accepted 2 July 2023
Available online 5 July 2023

Keywords:
Social sciences applications
Profiling
Optimization
Task parallelism
Data parallelism
HPC
Co-design
CcNUMA

ABSTRACT

The article presents optimization techniques for two Python-based large-scale social sciences applications: SN (Social Network) Simulator and KPM (Kernel Polynomial Method). These applications use MPI technology to transfer data between computing processes, which in the regular implementation leads to load imbalance and performance degradation. To avoid this effect, we propose a 2-stage optimization. In the first step, the order of tasks is changed, and in the second step, the tasks are divided into smaller ones for easier allocation. In addition, we focus on mitigating performance and memory bottlenecks using modern ccNUMA systems with multiple NUMA domains. As part of the performance analysis, the limitations of communication in data traffic between and within the processor were revealed and resolved through appropriate data allocation. Benchmarking was carried out, examining various environments, including vendors of traditional x86-64 and ARM-based processors for HPC.

1. Introduction

Conducting research on social sciences through simulations has become a common practice in the scientific community. Social science applications are most often based on the Agent Based Modeling (ABM) paradigm [1], which uses graph models to simulate the flow of data between the entities of this analysis.

The overriding motivation for the preparation of this work is to make better use of available resources by better understanding their capabilities. In particular, the analysis is based on HPC resources where we strive for trade-offs and correlations between application and hardware. Our goal is to reduce performance bottlenecks and overcome memory limitations to improve overall performance. We tackle a wide variety of profiling scenarios and deep code analysis to achieve these goals.

In the foundation of the social sciences simulations is reproduction of every entity, whether in the form of a person or another object, e.g. messages in a social network, depending on the type of study. It is common to strive to obtain more and more accurate results by increasing the measurement resolution. This, of course, involves an increase in the amount of input data, the size of memory structures used to store them, the amount of communication between processes, and thus the complexity of processes and the amount of resources necessary to process them.

It should be noted that in this work we do not deal with the correctness and effectiveness of the model of simulation of the flow of messages through the network, but with the optimization of the processing of this simulation through the use of techniques related to the division and allocation of tasks and data using the capabilities of the computing environment. Reducing the calculation time fosters obtaining results faster and reducing the energy consumption needed to perform tasks.

The analysis and optimization in this study are based on two applications: the SN (Social Network) Simulator and the KPM (Kernel Polynomial Method) application. The code analysis and optimization are performed to alleviate performance and memory bottlenecks and improve the overall performance on modern ccNUMA systems with multiple NUMA domains. During the analysis, we reveal some communication constraints for inter- and intra-CPU data traffic.

This activity addresses a variety of modern parallel architectures, including typically a single computing node based on

* Corresponding author at: Czestochowa University of Technology, Dabrowskiego 69, 42-201 Czestochowa, Poland.
E-mail addresses: lszustak@icis.pcz.pl (L. Szustak), lawenda@man.poznan.pl (M. Lawenda), sarming@cs.uni-salzburg.at (S. Arming), gbank@cs.sbg.ac.at (G. Bankhamer), christoph.schweimer@protonmail.com (C. Schweimer), elsa@cs.sbg.ac.at (R. Elsässer).

https://doi.org/10.1016/j.future.2023.07.005

x86-64 and ARM-based architectures. In contrast to the compute oriented code of SN Simulator, the KPM application is memory-intensive and demands a higher memory bandwidth. This application is also parallelized using the MPI standard for Python.

The advantage of the presented approaches is their versatility, which allows them to be used for both analysis and subsequent optimization in many applications with a similar processing model.

2. Related works

Applications simulating the spread of messages in social networks require processing large amounts of data based on complex graph structures. In order to obtain quasi-real-time results, it is usually necessary to implement optimization using available techniques. A common approach is to increase performance by means of system trajectory optimization by specifying the initial conditions to maximize or minimize the final number of active graph nodes. Solutions using efficient algorithms based on statistical physics or heuristics to select nodes for trajectory optimization in cascading processes in graphs are: determining appropriate initial conditions to optimize the final number of active nodes [2], approximation algorithm of selecting the most influential nodes [3], selecting nodes in the network to quickly detect the spread of information [4].

However, the co-design approach seems to be more generic and versatile in use to improve overall application performance. Although it has been less analyzed in such type of simulation so far, therefore it is difficult to provide direct references to other works of this kind. Presented in the paper co-design based on performance analysis and application optimization in two areas: data flow adjustments and effective use of software and hardware solutions offered by modern multiprocessor systems.

In this elaboration, optimization was done using the data parallelism approach, which is the most common parallel decomposition strategy, by which an application's data domain is decomposed into as many data partitions as threads assigned to the computation. In data-parallel model, tasks are assigned to threads, and each task performs similar types of operations on different data. At an abstract programming level, data-parallel programs consist of a loop body executing on different parts of the input data [5].

The presented approach shall be implemented in areas where we deal with large data structures (e.g. matrices, graphs). One of them is the training of deep neural networks in a dynamic stochastic batch size gradient descent approach based on conscious performance technology. This method assesses the processing capacity of each node by dynamically assigning a minibatch to each node, ensuring that the update time of each iteration between nodes is essentially the same, lowering the average node gradient [6].

Another example where data parallelism has been studied is the propagation of beliefs in factor graphs. Operations on potential tables are parallelized by dividing them into small parts, each of them is processed by a separate process. For factor graphs with large potential tables, data parallelism has proven to be an effective approach to speed up belief propagation [7].

Optimization using the hardware and software co-design approach is presented in this work based on the ccNUMA architecture. The NUMA (Non-Uniform Memory Access) architecture is widely used for concurrent access in modern computer systems. It allows programmers to create a logically consistent address space controlled by individual processors controlling a specific pool of operational memory. The peculiarity is that a given processor accesses its own memory much faster than the memory handled by other processors. An improvement in this model is the addition of a hardware-based processor responsible for cache coherency (ccNUMA), which greatly facilitates the programming process of systems based on the NUMA architecture. However, effective use of ccNUMA is still a challenge for developers creating multiprocessor applications and ensuring optimal access to data.

Methodologies for increasing performance in various NUMA configurations for modern processor architectures (AMD Rome and Huawei Kunpeng 920) are presented in [8]. The authors propose a multi-domain implementation for dense matrix factorizations and matrix inversion (DMFI) as well as hybrid task- and loop-level parallelism. The suggested parallelism sets up multi-threaded executions to fix the core-data linkage, taking advantage of locality at the cost of less code modification.

The problem of designing multi-threaded applications, aware of the limitations of the NUMA architecture by reducing data transfer, is also the subject of other studies, specifically in respect of: thread and memory pages migrations [9], thread-to-core allocation [10], reduction of cache coherence traffic [11].

3. Architecture and software overview

In this work, we address a variety of modern parallel architectures, including traditional CPU vendors for HPC based on x86-64 and ARM-based processors. In particular, we explore five dual-socket servers equipped with various processor microarchitectures: MILAN and ROME from AMD, Ice Lake and Cascade Lake from Intel, and Huawei TaiShan ARMv8.2-based product. Table 1 summarizes parameters of these platforms.

Among the parameters that characterize every platform are: base frequency, number of cores, memory specification, and number of NUMA domains. The studied processors include different numbers of cores and are clocked by the base frequency listed in Table 1. The CPU designs of all the platforms feature the out-of-order execution model and support the frequency boost technology [12–14], where the maximum turbo frequency depends on the type and intensity of workload, as well as the number of active cores. The simultaneous multithreading (SMT) is turned off for all the systems. The studied platforms, excluding the Cascade Lake-based server, incorporate eight DDR4 memory channels per socket (CLA-SP offers six memory channels per socket).

All the platforms represent a group of the ccNUMA shared memory architectures [15] combining whole memory regions using 2×8, 2×1, and 2×2 NUMA domains for AMD, Intel, and Huawei processors, respectively. In multi-chip systems, differing distances between cores and NUMA domains cause non-uniform memory access issues and, consequently, can yield CPU load imbalance [16]. Generally, the applications will perform best when the cores access memory on the same NUMA domain. A NUMA-aware memory locality helps parallel codes achieve a more balanced CPU load for memory-intensive applications.

SN Simulator and KPM applications integrate Python packages, including SciPy, NumPy, Pandas, Numba, and MPI4Py. Considering a parallel implementation of tackled applications with MPI for Python library usage, we utilize OpenMPI MPI implementation with the support of MPI-3 Remote Memory Access (RMA). We configure the software environment identically for all the performed tests and all computing platforms. Table 2 summarizes the software configuration used in this work.

4. Use case of task-parallel application: SN simulator

4.1. Overview of SN Simulator

The Social Networks (SN) Simulator has been designed to model and simulate the spread of messages in social networks.

Table 1
Specification of computing platforms.

Platform name	Type of CPU	Codename	Cores	Freq.	Memory	NUMA domain
Gigabyte H262-Z63 Gen10 Plus v2	2 × AMD EPYC 7763	MILAN	2 × 64	2.45 GHz	512 GiB DDR4-3200	2 × 8
HPE Proliant DL385 Gen10 Plus	2 × AMD EPYC 7742	ROME	2 × 64	2.25 GHz	512 GiB DDR4-3200	2 × 8
Supermicro SYS-120C-TR	2 × Intel Xeon Platinum 8360Y	Ice Lake (ICL-SP)	2 × 36	2.4 GHz	512 GiB DDR4-3200	2 × 1
Huawei CH121 V5	2 × Intel Xeon Platinum 8268	Cascade Lake (CLA-SP)	2 × 24	2.9 GHz	192 GiB DDR4-2933	2 × 1
Huawei Atlas 800 model 3000	2 × Kunpeng 920-4826	TaiShan (ARMv8.2)	2 × 48	2.6 GHz	512 GiB DDR4-2933	2 × 2

Table 2
Software configuration for SN Simulator and KPM benchmarks.

Software/Package	All platforms
Python	3.9.6
Numba	0.53.1
Numpy	1.21.1
Scipy	1.7.1
Pandas	1.3.2
MPI4Py	3.1.1
MPI Library	OpenMPI 4.1.1

To develop the simulation model, we first analyzed the structure of networks, in which messages w.r.t. certain topics are spread. These networks exhibit a different degree distribution than most known large social networks. Especially, instead of the well-known power law property we observed a so called χ^2-distribution. Furthermore, these networks have a high clustering coefficient and small distances between two randomly chosen nodes [17]. The structure of these networks also influence the spreading and cascading behavior of messages.

Second, we looked into features and properties that influence the spreading of messages in networks. We investigated various studies [18–24] to find the most influential features that play a role whether a message has a higher or lower chance to be resent. In our analysis, we initially identified 25 numerical and binary features, related to the user and the message, that are potentially influential. We gradually eliminated features that are either highly correlated to another feature or only have a minor influence on the chance for a message to be resent with a recursive feature elimination technique, which reduced the number of features to 11. Further analysis showed that one feature (poll in the message) is rarely being used and that two other features (number of characters and day of the week) also only have little influence on the chance for a message to be resent, such that we decided to eliminate these features from the model as well, leading to 8 features, 4 related to the user and 4 related to the message. The user features that have the main influence on the spread of messages are:

- Is the user verified
- Is the user an active user
- Does the user account have a link to another URL
- Are the default profile settings being used

The activity of a user is measured in terms of messages posted per day, where a user is labeled as an active user if the user has posted more messages per day than the mean in the respective data sets. The most important features associated with a message are:

- Is there a hashtag in the message
- Is there a mention in the message
- Does the message contain a URL
- Does the message contain a media element (e.g., photo)

The SN Simulator incorporates these features. When a message is created at a certain node (representing a user in a social network such as Twitter or Facebook), we assign a binary feature vector to this message. Additionally, each node in the simulated network is assigned a binary feature vector, which is selected according to a distribution we observe in real world social networks.

The messages are then propagated throughout the network as follows. Each time a message arrives at a certain node (including the node it was initially created on), the message is resent to the node's neighbors with a certain probability. The probability and also the amount of neighbors that receive the message depends on (i) the feature vector of the message, and (ii) the feature vector of the original author of the message. This probability of forwarding the message is further influenced by the current distance of the message to the initiator. We observed that the probability for a message to be resent at a certain node depends on the distance of this node to the initiator of the message.

The SN Simulator accounts for the structure of the network, in which messages of a certain type are generated, the two feature vectors when generating a message, and it simulates the spread of a large number of messages according to these features as well as the distance of the nodes reached by each of these messages at a certain point in time. To analyze the spreading behavior with several millions of messages of different types, as well as for validation purposes, it is of key importance to design a parallel, highly scalable version of the SN Simulator.

4.2. Data sets and parameters

One way to construct a specific simulation is by providing two input files to the simulator. First, a graph that describes the structure of the social network that should be simulated. In our experiments, we used follower–followee relationship graphs extracted from Twitter. This graph consists of Twitter users that tweeted about a certain topic. An edge (u, v) in this graph indicates that v is a follower of u. Second, we provide a file that consists of a set of tweets these users made about a certain topic. For each tweet, it lists the tweets features, the ID of the originating author, the features of the originating author and the number of times it was retweeted. The SN Simulator extracts information out of this data and uses it to create the propagation model.

In our experiments we consider multiple data sets, each corresponding to one of the following topics: *vegan*, *neos*, *fpoe* and *covid19*. For each topic a list of keywords was used to crawl a follower–followee graph as well as a set of Tweets via the Twitter API. The tweets of each data set were collected during a consecutive 41 day period between August 2019 and March 2020. The *vegan* data set consists of keywords related to veganism. The *fpoe* and *neos* data sets focus on tweets related to the

Table 3
Size of the data sets for the SN Simulator.

	Number of users	Number of tweets
vegan	12 088	6030
neos	8400	4487
fpoe	21 514	19 418
covid19	51 391	375 736

Austrian political parties FPÖ and NEOS, respectively. They were acquired before the Parliamentary elections in Austria at the end of 2019. Finally, the *covid19* data set consists of tweets discussing the social distancing measures in February and March 2020. In Table 3, we state the size of the graphs as well as the number of tweets in each data set. The first column lists the name of the data set, the second column the number of users (or nodes) in the corresponding graph. The final column lists the number of tweets that were collected.

Besides the two input files, there are two important parameters that control the scope of the simulation. There is the parameter `sources` which describes the number of users that are designated to initiate tweets. Next, there is `samples`, which denotes the number of tweets initiated by each such user.

Such a selection of authoring users is performed for each possible combination of message and user feature vectors that was encountered in the list of tweets (i.e., in the second input file). The overall number of initiated tweets is therefore proportionate to "number of feature vector combinations" times `sources` times `samples`.

Throughout the experiments in this section, we employ `samples` = 1000 and set `sources` between 100 and 400. For such parameter values, the number of initiated Tweets is of magnitude 10^7. The resulting simulations are of sufficient size for the purpose of collecting various metrics about the propagation behavior of the simulated Tweets. Additionally, these simulations are small enough to be executed on a single compute node. In Section 6.1 we investigate larger simulations that run on multiple nodes.

The workflow of a typical simulation run performed as part of our experiments may be summarized as follows. First, we decide on a topic, e.g. *vegan*. The follower–followee graph as well as the set of Tweets corresponding to this topic are then provided to the simulator as input. Additionally, values for `samples` and `sources` are specified to control the scope of the simulation. Given these input files and parameters, the SN Simulator then creates a simulation model and runs the simulation. As an output the SN simulator delivers several statistics, for example the average number of times that Tweets with a certain feature were retweeted.

4.3. Parallelization strategy and performance analysis

The SN Simulator code implements a task-based parallelism strategy by employing the MPI standard for Python. In contrast to the traditional execution scheme for the message-passing interface, the application execution scheme uses the master/worker approach offered by the `mpi4py.futures` package. This package provides a high-level interface for asynchronously executing tasks on a pool of worker processes using the MPI functionality for inter-process communication.

During the initial application stage at the master process (rank 0), all required data are loaded and initialized. Afterward, the required data are exposed by the master to the worker processes using a hybrid distributed-shared memory model.

To achieve this goal, we first split the global pool of workers into groups of workers assigned to the shared memory regions of subsequent computing nodes by using the collective MPI method `Split_type` with `MPI.COMM_TYPE_SHARED` parameter. Next, a head process is selected from every group to receive the broadcasted data from the master process. Each head process allocates all received data in the shared memory space of every computing node using the collective MPI routine `MPI_Win_allocate_shared`. As a result, every computing node consists of a single group of MPI processes that share memory regions and enable immediate load/store operations on the data.

In the next application stage, the master process composes and submits the schedules for the pool of tasks to be executed asynchronously and then waits for the partial outcomes. Simultaneously, the worker processes perform a series of tasks submitted by the master in a round-robin manner, execute them one by one and further communicate back the outcomes of every task to the master process. Meanwhile, the master process aggregates the received partial results from the workers to produce the final results of the simulation.

Generally, the execution of subsequent tasks is the critical and most time-consuming part of the code. A task consists of the simulation of `sample`-many tweets that originate from a fixed user. Out of every feature vector, multiple users are sampled, and for each such user a task is submitted. The number of users sampled per feature vector is specified with the `sources` parameter. The computations assigned to every task are rather compute-bound and dominated by the calls to the function `edge_sample` (see Section 4.5).

We observe that in the basic version of the code, all workers process a similar number of tasks. However, the performed analysis reveals a large load imbalance between worker processes. Fig. 1 illustrates an example of performance analysis for the basic version of SN Simulator obtained for the *vegan* data set when using a platform with two Intel Xeon Platinum 8268 CPUs.

As shown in Fig. 1a, the overall performance is limited by the worker process that features the highest computation time of tasks. This worker accomplishes assigned tasks about 3 times slower than the worker process with the shortest execution time despite both these workers proceed a comparable number of tasks (Fig. 1b). Generally, the basic parallelization strategy leads to workload imbalance between worker processes.

To explain this behavior, we have to look at the cost of all tasks individually. Fig. 1c traces a detailed cost analysis of every task measured on each worker process separately. In the illustrated example, we reveal the significant difference in the costs of tasks. The performed analysis helps us indicate two main groups of tasks. The first group features a large number of tasks with negligible costs, while the second group includes a relatively small number tasks with high costs. We reveal that in a typical execution scheme, the second group of tasks is submitted by the master process at the end of the tasks queue and, consequently, is executed during the final part of the simulation.

4.4. Optimization method for tasks and workload distribution

To alleviate or even avoid load imbalancing and improve the overall performance, we deliver the 2-stage method for code optimizations. In the first stage, we propose to modify the order of the task queue and move the costliest tasks at the beginning of application execution.

The deep analysis of the task execution scheme enables us to identify that the simulation of messages which are predicted to be more likely to be retweeted frequently should demand more computation time. As a result, we distinguish the best candidates for the costliest tasks and force the master process to submit them at the beginning of the tasks queue.

To achieve this goal, we propose to sort the feature vectors heuristically according to their expected runtime. For each such

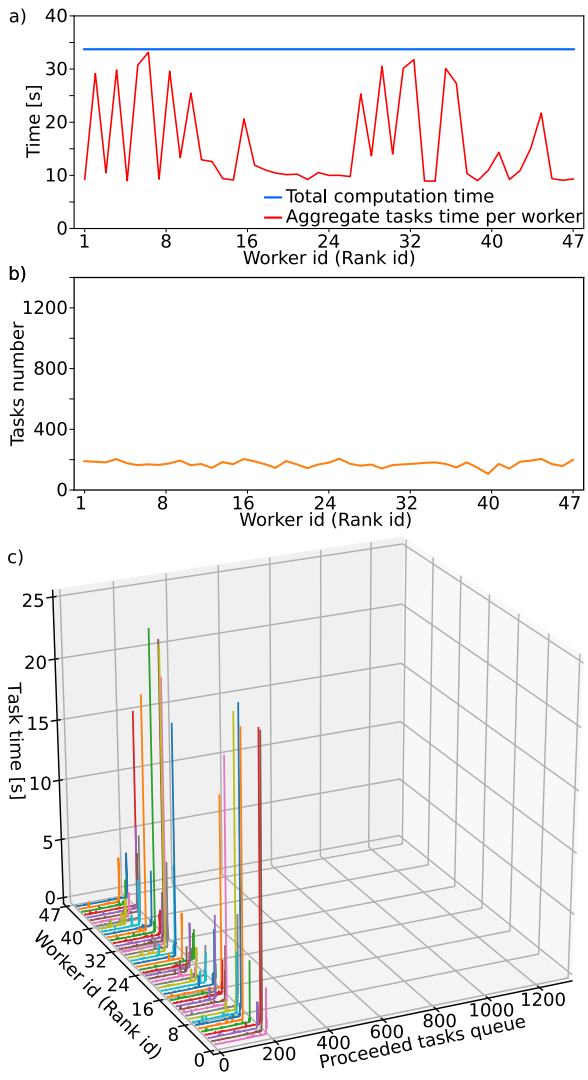


Fig. 1. Performance analysis for the basic code of SN Simulator obtained for the *vegan* data set and two Intel Xeon Platinum 8268 CPUs: (a) aggregated execution time of all tasks per each worker process, (b) tasks distribution between worker processes, and (c) execution time of every task measured on the different worker processes.

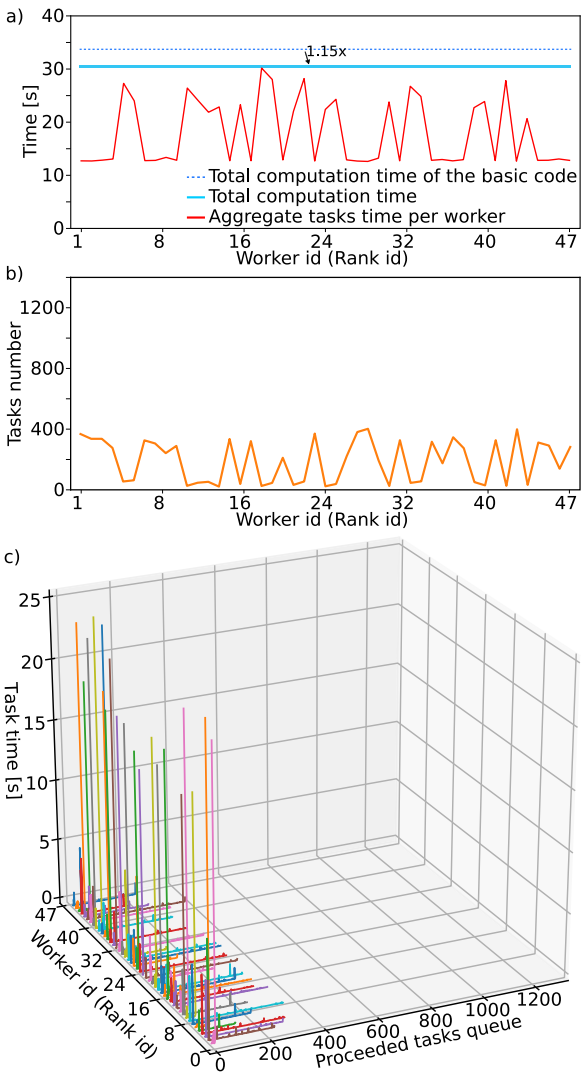


Fig. 2. Performance analysis for SN Simulator with modified order of tasks execution obtained for the *vegan* data set and two Intel Xeon Platinum 8268 CPUs: (a) aggregated execution time of all tasks per each worker process, (b) tasks distribution between worker processes, and (c) execution time of every task measured on the different worker processes.

feature vector, the expected number of retweets is an input of the simulator. This information is usually derived from real world data and the simulation attempts to match this number. Therefore, it serves as a good heuristic to predict the runtime of a simulated tweet.

Fig. 2 demonstrates an example of the impact of the proposed modification on load balancing, workload distribution, and overall performance obtained for the *vegan* data set and two Intel Xeon Platinum 8268 CPUs. Applying new order of task execution enables performing all the costliest tasks at the beginning of execution (Fig. 2c) and slightly reduces load imbalancing (Fig. 2a) despite the uneven distribution of tasks (Fig. 2b). In this case, the proposed method improves the overall performance by about 1.15x compared to the basic code.

In the second stage of the proposed method, we focus on alleviating the load imbalancing performance disadvantage caused by long-running tasks. While analyzing the execution tasks scheme, we reveal that even a relatively small number of heavy tasks can strongly limit the overall performance. The key to understanding this constraint is focusing on the task construction process.

In the basic version of the code, the SN Simulator simulates the propagation of retweets in a Twitter follower graph. Generally, two parameters determine the number of tweets that are simulated. The first parameter (`sources`) determines the number of tweet authors from which tweets originate, while the second one - `samples` - denotes the number of tweets initiated by each of these authors. The samples of many simulated tweets initiated by a fixed author are considered an atomic task. The same worker process executed many simulations on a single core in all of these samples.

As some authors are prone to generate tweets that are likely to be retweeted multiple times, this caused some of these tasks to take way longer to be processed than others. Consequently, it is challenging to balance the workload among workers in a comparable fashion. However, reducing the cost of the task can overcome this constraint and, consequently, lead to better CPU utilization.

To tackle this problem, we deliver the task-splitting approach that assumes reducing the number of tweets simulated inside every task. As a result, this approach replaces heavy tasks with a

more considerable amount of smaller tasks to reach a more efficient task distribution. To achieve this aim, we introduce the third parameter of the SN Simulator application called `sample_split` that helps us partition each task into a set of `sample_split`-many smaller tasks.

However, the more considerable amount of tasks generate extra worker-master communication overhead caused by collecting more partial outcomes from the larger number of smaller tasks. As this constraint can lead to performance degradation, we face searching trade-offs and synergy between the sizes of tasks, communication overhead, and the overall performance.

Experimentally, we validate and examine the proposed approach, exploring different values of `sample_split` parameter for various application data sets, parameters configurations, and computing platforms. Fig. 3 illustrates an example of the correlation between different task sizes and overall performance obtained for the *vegan* data set and two Intel Xeon Platinum 8268 CPUs. To achieve the desired accuracy of this experiment, we perform at least nine repetitions and present the minimum, median, and maximum computation times for every configuration.

As shown in Fig. 3a, calibrating the proposed task-splitting approach and setting parameter `sample_split` to 5 brings significant performance improvements, accelerating computation of about 1.42x. The performed analysis outlines noticeable performance improvement for relatively small values for `sources` parameter (see Fig. 3a and b). In contrast, the performance gain is negligible for larger values of the `sources` parameter (Fig. 3c).

The proposed task-splitting approach opens a way to tackle the load imbalancing disadvantage. Fig. 4 demonstrates an example of the impact of the proposed approach on performance, distribution of tasks and workers workloads. In the illustrated example, the new version of the SN Simulator code features relatively uniform workload distributions over available worker processes despite the uneven distribution of tasks. The task-splitting approach results in reduced tasks costs by requirements of processing a more significant number of tasks in comparison to the basic version of the code (see Fig. 1c). As a result, the cost of every task is reduced by a factor of 5, while the total amount of tasks increases five times.

4.5. Numba-based performance optimization

To elevate the performance of SN Simulator beyond the limits of the standard Python interpreter, we employ the Numba [25] python library to optimize hotspots. We use the just-in-time (JIT) compiler provided by Numba to translate the computing kernels that – in a typical simulation run – are called tens of millions of times and consume more than 95% of computing time. The applied performance analysis of the SN Simulator code shows that the function called `edge_sample` lies at the core of our simulation.

The selected function is the primary hotspot in the simulations of the spread of messages, as it is called each time a simulated message arrives at a user in the simulated network. The function is used to determine which followers of this user will adopt (i.e., retweet) this message. For each of these retweeting users, the function `edge_sample` needs to be called again. This scheme continues until any user no longer retweets the message, and the simulation of the current message stops.

The most noticeable parts inside `edge_sample` function are (i) the generation of random values that follow a binomial distribution and (ii) the random sampling from a list without replacement. Both of these parts are provided by the NumPy python library and can be successfully eligible for performance improvements by Numba.

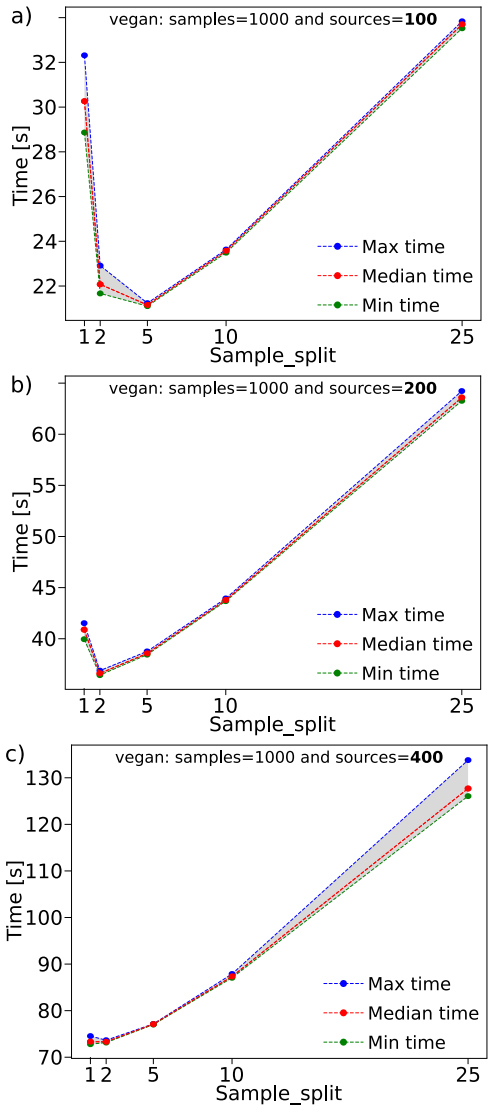


Fig. 3. Impact of different values of `sample_split` parameter on performance obtained for *vegan* data set with different parameters configurations, including (a) `sources` = 100 (b) `sources` = 200 and (c) `sources` = 400, when using platform with two Intel Xeon Platinum 8268 CPUs.

Every indicated part of `edge_sample` function is marked as Numba functions and translated into machine codes during application execution (just-in-time compilation). As a result, these parts are compiled once and used numerous times in a typical simulation run. Additionally, to improve the efficiency of our code, we successfully overlap the time-consuming compilation process of selected hotspots with reading input data. As a result, all workers hide the compilation process during data loading from files tackled by the master.

4.6. Experimental results

We perform a set of tests to explore the efficiency of the proposed methods for different application scenarios that mimic the flow of messages/tweets on twitter about a certain topic at a certain time, including four data sets *neos*, *vegan*, *fpoe*, and *covid19*. We benchmark the proposed approach using five servers (see Tables 1 and 2) with Milan, Rome, Ice Lake, Cascade Lake, and TaiShan V110 (ARMv8.2). To ensure the reliability of benchmark results, every type of test is repeated at least five times, and the

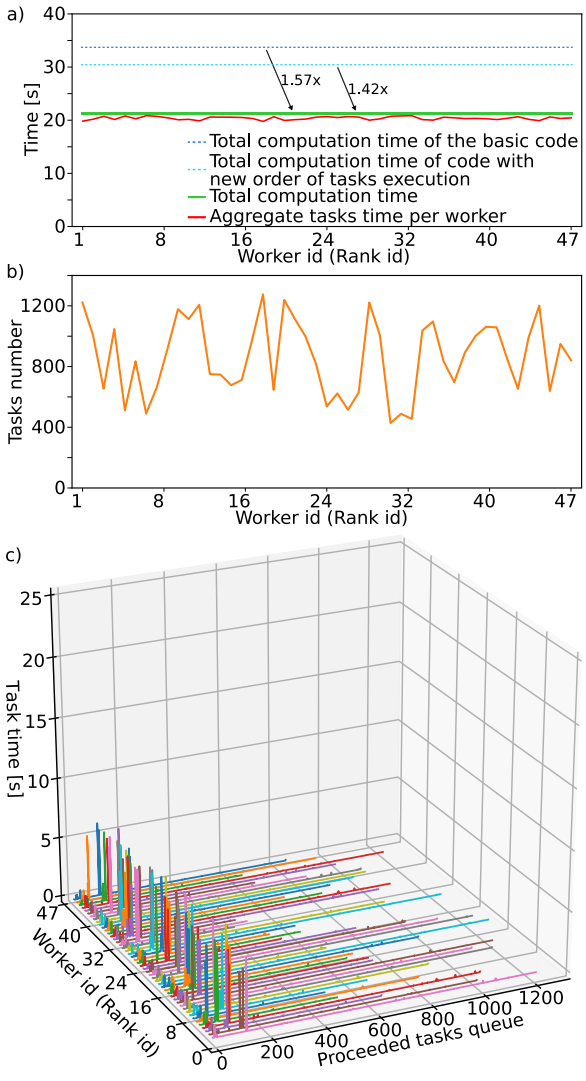


Fig. 4. Performance analysis for SN Simulator with proposed task-splitting approach obtained for `sample_split` = 5 when exploring the *vegan* data set and two Intel Xeon Platinum 8268 CPUs: (a) aggregated execution time of all tasks per each worker process, (b) tasks distribution between worker processes, and (c) execution time of every task measured on the different worker processes.

median value of measurements are used to get statistically sound results, with the relative standard deviation (RSD) less than 2%.

In the first stage of benchmarks, we evaluate the impact of the applied new tasks execution order on performance. Fig. 5 reports an example of performance gains in comparison to the basic version of the code considering application configuration with `samples` = 1000 and `sources` = 100. The applied tests indicate a noticeable performance improvement for the platform with ARM-based processors, where the new execution order of tasks leads to about 2.1x faster execution. The applied code transformation yields relatively smaller performance improvements (up to about 1.8x faster) for other platforms.

The next proposed modification expands the SN Simulator code with the task-splitting approach that helps us reduce the communication constraints for inter- and intra-CPU data traffic. Fig. 6 outlines the final performance gains achieved for the code configuration with `samples` = 1000, `sources` = 100, and the optimal values for the `sample_split` parameter (see Fig. 6b). The proposed modification accelerates the SN Simulator application up to 1.8 times compared to the code version with modified execution order of tasks.

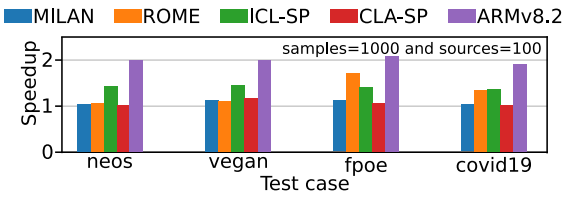


Fig. 5. Impact of proposed new execution order of tasks on the SN Simulator performance.

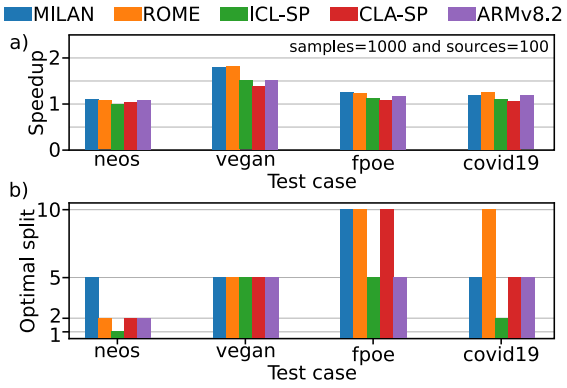


Fig. 6. Impact of task-splitting approach on the performance: (a) total performance gain, and (b) the optimal values for the `sample_split` parameter.

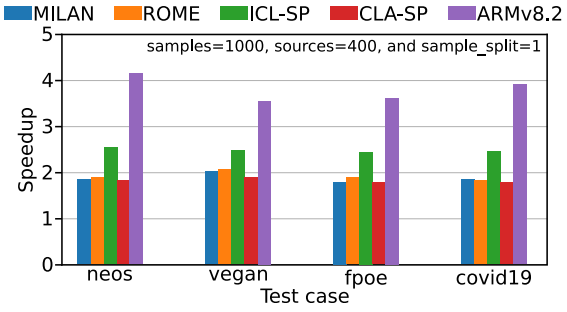


Fig. 7. Total performance gain obtained for SN Simulator with enabled NUMBA-based optimization.

Our experiments are completed by exploring the Numba-based modification. Fig. 7 shows an example of the impact of the Numba compilation process on total performance gains for SN Simulator considering different application data sets with configuration `samples` = 1000, `sources` = 100, and carefully selected `sample_split` = 1. We observe that applying Numba-based modification for SN Simulator significantly reduces the total execution time for all performed tests.

The proposed approach leads to sustained speed up due to more efficient utilization of computing resources. The platform with ARM-based processors features the highest acceleration of computations (up to 4.17 times) when enabling NUMBA. The applied tests also indicate a noticeable performance improvement for the platform with two Intel Xeon Platinum 8360Y, where the Numba-based modification leads to about 2.5 faster execution. The performed modification reduces the execution time by about 2x for both platforms with AMD EPYC CPUs.

5. Use case of data-parallel application: KPM

5.1. Overview of KPM application

In this section, we focus on the KPM (Kernel Polynomial Method) approach to computing the approximate spectrum of eigenvalues of a given graph. The output of the developed KPM code is later used to validate synthetically generated graphs, which may be used as input for the SN Simulator. The KPM application approximates the eigenvalue spectrum of a given graph in form of a histogram of eigenvalues. This metric contains various information about the graph structure, such as the number of connected components or how tightly the input graph is clustered. These statistics can be used to estimate whether a given artificially generated graph is similar to a graph that was extracted from a real world social network.

The KPM application takes as input the normalized Laplacian matrix of a given graph, which can be computed from the graph's adjacency matrix. It then approximates the eigenvalue spectrum of this Laplacian matrix by computing a histogram of all eigenvalues. This computation follows the *polynomial expansion filtering* approach introduced in [26]. This approach efficiently approximates the number of eigenvalues that lie in a fixed interval. A repeated application of this approach allows to compute the histogram bin-by-bin.

We only give a brief overview of the employed approach as it is quite involved and relies on multiple techniques from linear algebra. In essence, to approximate the number of eigenvalues in a single histogram bin, multiple random sample vectors out of $\{-1, 1\}^n$ are generated. The number of such samples can be specified with the `samples` parameter. Each such sample vector is then involved in a series of linear operations. Most notably, it undergoes multiplications with the input matrix. The number of such multiplications corresponds directly to the value specified via the `degree` parameter. An increase in the values of `samples` and `degree` therefore comes with additional computational effort, however, also leads to an improved accuracy of the estimation. To control the resolution of the computed histogram, we use the `intervals` parameter. This parameter simply specifies the number of histogram bins that should be computed.

The primary input for the KPM application are sparse matrices. This is motivated by the desire to compute the eigenvalue spectrum of graphs of social networks. Usually, such adjacency matrices are sparse. For example, consider the friendship relationship graph of the Pokec social network (provided by SNAP [27]) that was used in our experiments. This graph consists of roughly 1 million nodes but has an average degree of less than 20. The application therefore relies on SciPy's compressed sparse row matrix data type `csr_matrix` together with its provided linear operations. Note, for our experiments we used a preprocessed version of this Pokec graph in which all nodes with degree less than 6 were removed. This graph still consists of 990,908 nodes. Fig. 8 presents the approximated eigenvalue spectrum of this graph.

In summary, a typical execution of the KPM application is conducted as follows. First, we provide a graph as input. In our experiments we always use the aforementioned graph of the Pokec social network. Additionally, we set values for `samples`, `degree` and `intervals` to control the accuracy of the histogram. The application then runs and computes a histogram of eigenvalues. The output lists the approximate number of eigenvalues for each histogram bin.

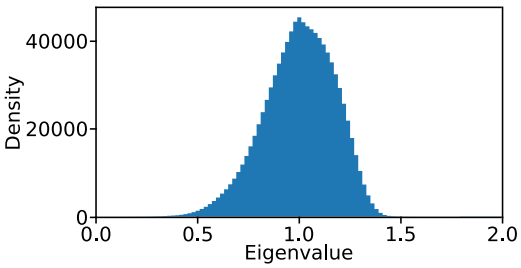


Fig. 8. Approximated eigenvalue histogram of the Pokec friendship relationship graph. Computed with the KPM application and parameters `intervals` = 101, `samples` = 200 and `degree` = 300.

5.2. Parallelization strategy and performance analysis

The KPM application also utilizes the MPI standard for Python to enable data parallelization on clusters of multicore ccNUMA nodes. According to the applied execution strategy, the KPM code combines distributed and shared memory models.

At the initial application stage, a single MPI process loads all required input and read-only data from the file and spreads it to the other MPI processes. The required data are explicitly broadcasted to every computing node and then shared within every node to the MPI processes by the MPI3-RMA interface. As a result, every computing node consists of a single group of MPI processes that share memory regions and enable immediate load/store operations on the data.

In the next application stage, data parallelism is applied to distribute the workload across all MPI processes. The parallelization strategy assumes equal distributions of intervals and samples across all MPI processes. Considering a wide range of computing platforms with different numbers of cores, we also propose slightly calibrating the `intervals` and `samples` values of the input parameters if the divisibility of workload is not fulfilled between the number of used MPI processes. As a result, the overall workload equals `intervals` · `samples`. Execution of a single sample vector consists of `degree`-many matrix–vector multiplications with the input matrix. The operations performed in a given sample do not depend on other samples and, therefore, can be processed by MPI processes independently. In the last application stage, using the MPI reduce operation, a single MPI process aggregates the partial eigenvalue count estimations received from the other MPI processes to construct the final histogram.

The performed analysis reveals that even though the first stage executes mainly sequentially, it does not limit the overall performance. The measured cost, in this case, is negligible for all performed tests and application scenarios. Moreover, the costs of (i) management of workload distribution as well as (ii) aggregation of the partial outcomes are negligible as well. In contrast, we observe that the critical and the most time-consuming part of the code mainly corresponds to the sparse matrix–vector products performed by well optimized SciPy library. The selected computing kernel of the code assists in estimating the eigenvalue counts within subsequent samples for every interval by each MPI process independently.

However, although the proposed parallelization strategy enables equal distributions of intervals and samples across computing resources, the performed analysis reveals the load imbalancing between MPI processes. Fig. 9 characterizes an example of the execution time measurements of the KPM code using the platform with two AMD EPYC 7742 CPUs. For this experiment we set `intervals` = 32, `samples` = 96 and `degree` = 100. This approximates the eigenvalue spectrum of the Pokec graph

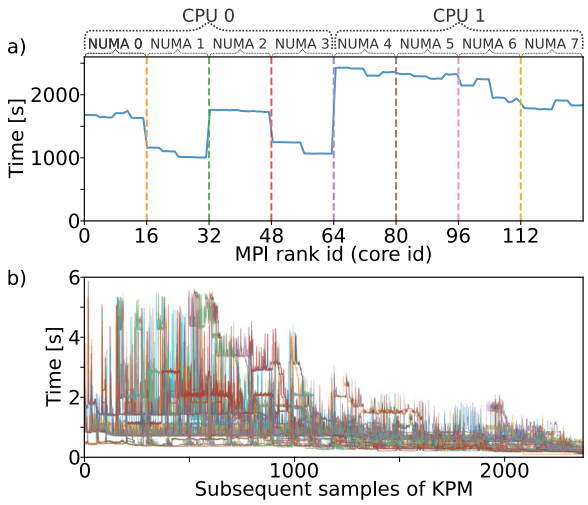


Fig. 9. Performance analysis for a basic version of the KPM code obtained on the system with AMD EPYC 7742 CPUs: (a) pure computation time measured for every MPI process, (b) partial computation time of subsequent samples measured independently for every MPI process.

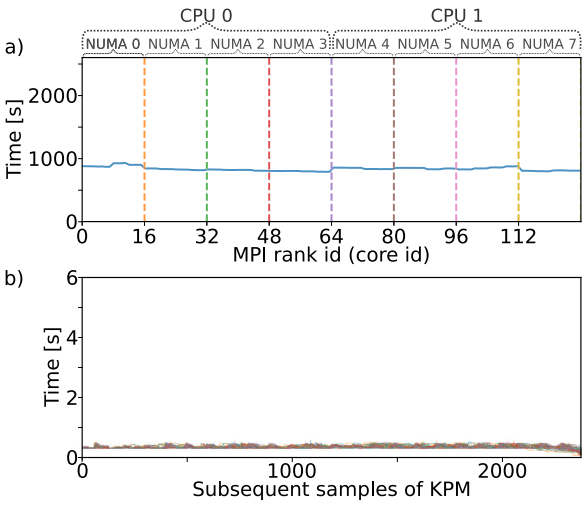


Fig. 10. Performance analysis for a ccNUMA-aware version of the KPM code obtained on the system with AMD EPYC 7742 CPUs: (a) pure computation time measured for every MPI process, (b) partial computation time of subsequent samples measured independently for every MPI process.

(described in Section 5.1) by computing a histogram with 32 bins. The values for `samples` and `degree` are chosen large enough to avoid inaccuracies on the tail ends of the approximated histogram. At the same time, they remain small enough to be suited towards experiments on a single compute node. Fig. 9a illustrates the pure computation time (without MPI communication costs) measured for every MPI process separately. Additionally, Fig. 9b presents partial computation time measured independently for subsequent samples processed by every MPI process.

As shown in Fig. 9a, the MPI processes pinned to the cores on the second CPU spend significantly more time on computation than the MPI processes that utilize cores located on the first CPU. Furthermore, the costs of subsequently processed samples differ from each other (Fig. 9b), although the computational complexity is comparable for all samples

5.3. Adaptation of KPM to ccNUMA systems

During the performance analysis, we reveal some implicit communication constraints that strongly limit the overall performance. To explain and solve this issue, we have to look at the input-data allocation scheme. To achieve this goal, during the application execution, we use the `numastat tool` to track memory statistics on a per-NUMA-node basis, including allocation usage, hits, and misses.

Our experiment shows that according to the first-touch policy, the data allocation takes place only in the memory region closest to a physical core on which the MPI process is executed. Considering the first stage of KPM, where a single MPI process (sub-master) allocates input data within every node, the application exploits a single NUMA memory region per node only. In consequence, the performance of data traffic is limited for the efficiency of a single NUMA region, while the other NUMA regions are not utilized for loading input data. In particular, this communication constraint is strongly noticeable for modern systems with AMD EPYC CPUs (see Fig. 9) that offer up to 16 NUMA regions.

A remedy for this issue is the ccNUMA-aware approach for allocating input data across available NUMA domains. To achieve this goal, we propose to create more groups per given node, with the master rank process broadcasting all input data across these groups. As a result, a single MPI process per every group called group-master (i) receives the required input data, (ii) saves them in its NUMA domain memory, and finally (iii) shares them with other MPI processes from a given group.

In the proposed data-allocation strategy, the critical point is mapping an appropriately chosen number of groups of MPI processes on the number of available NUMA domains and physical cores in a given system. However, the richness of the MPI library makes this issue relatively easy to overcome by using the MPI-3 RMA interface of MPI for Python.

In particular, MPI delivers the communicator split operation that in shared-memory platforms allows the creation of groups of MPI processes such that ranks in each group can share data memory efficiently. Further, the collective call of the `MPI.Win.Allocate_shared` procedure helps us to allocate memory region for input data that is shared among all processes in every group. The proposed approach also requires controlling the binding policy of MPI processes on physical cores, considering mapping successive MPI processes close to each other. We achieve the desired binding policy by using the MPI run-time parameters `--map-by core` and `--bind-to core` offered in the `OpenMPI` implementation of MPI4Py.

5.4. Performance evaluation

The proposed adaptation of the KPM code to the ccNUMA systems helps us reduce or even avoid the communication constraints for inter- and intra-CPU data traffic. Fig. 10 illustrates the performance analysis of the new version of the KPM code calibrated with 16 groups of the MPI processes and mapped on the 16 NUMA domains offered by the system with two AMD EPYC 7742 CPUs.

In contrast to the basic version of KPM, as shown in Fig. 10a, the proposed ccNUMA-aware data allocation strategy provides as balanced as possible workload of computing resources. The costs of subsequently processed samples feature almost equal partial execution time (Fig. 10b) for all MPI processes.

We benchmark the proposed approach using five servers with Milan, Rome, Ice Lake, Cascade Lake, and TaiShan V110 (ARMv8.2) processors. Each type of test is repeated at least five times, and the median values are used to get statistically sound results, with the relative standard deviation (RSD) less than 1%.

Table 4
Impact of ccNUMA-aware data allocation strategy on performance gain for different computing platforms.

Platform	Speedup	Number of groups
2x AMD EPYC 7763	1.7	16
2x AMD EPYC 7742	2.71	16
2x Intel Xeon Platinum 8360Y	1.1	2
2x Intel Xeon Platinum 8268	1.09	2
2x Huawei Kunpeng 920	1.2	4

Table 4 outlines the impact of delivered ccNUMA-aware data allocation strategy on the achieved performance gain. To maximally alleviate communication constraints of inter- and intra-CPU data traffic, we select the best configuration of the number of groups of MPI processes for every platform separately.

As expected, the optimal number of groups equals the available NUMA domains in a given platform. For the platforms with AMD CPUs, the new version of the KPM code requires to use 16 groups of MPI processes to get the best performance. It accelerates the computation of about 1.7x and 2.71x against the basic code for platforms with MILAN and ROME processors, respectively. The performance gains obtained for other platforms with a smaller number of NUMA domains are considerably lower. The configuration with 2 groups of MPI processes gives the best acceleration of computations (up to 1.1x) for both platforms with Intel CPUs. The platform with ARM-based processors features 4 NUMA domains and, consequently, requires calibrating the new version of KPM with 4 groups to accelerate the application by about 1.2 times maximally.

6. Exploration of large-scale computing systems

6.1. Evaluation of SN simulator on the intel-based cluster

In the first stage of benchmarks, we evaluate the performance impact of using the proposed task-reordering and task-splitting optimization methods, as well as the Numba-based modification under different application scenarios on the Altair supercomputer located at *PSNC*.[1] The Altair cluster consists of 1320 compute nodes, with two Intel Xeon Platinum 8268 CPUs of 24 cores each, at 2.9 GHz, with 192 GB RAM each, interconnected with InfiniBand EDR.

Fig. 11 depicts the performance comparison between the basic and optimized versions of SN Simulator obtained for the *vegan* data set on Altair using 1 up to 32 computing nodes (48 up to 1536 cores). In this test, we consider the SN Simulator configuration of `samples` $=$ 20 000 and `sources` $=$ 100, resulting in roughly $2 \cdot 10^8$ simulated tweets in each run. A message simulation of this size could be used to derive insights about the propagation of information in large real world social networks in which millions of messages are sent each day. Following the proposed task-splitting approach (see Section 4.4), we set the `sample_split` parameter to 5 as an optimal value for the *vegan* data set, obtained in the experiment shown in Fig. 3a.

As shown in Fig. 11a, the optimized version of the SN Simulator significantly reduces the execution time when employing 1 to 32 nodes, in comparison to the basic one. The performance gains become more evident as the number of cores increases (Fig. 11b), accelerating the application from 3.19 to 6.13 times when employing 1 and 16 nodes, respectively. At the same time, the optimized version of SN Simulator achieves a performance gain of about 5.6 times against the basic code when employing 32 nodes.

[1] PSNC is the Poznan Supercomputing and Networking Center, in Poland.

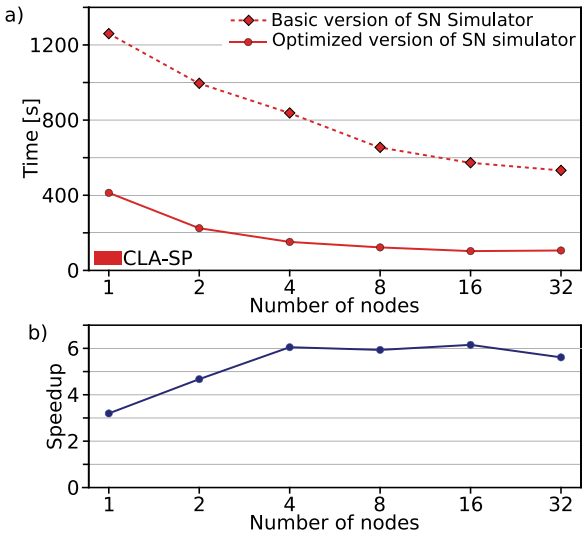


Fig. 11. Performance comparison between the basic and optimized versions of SN Simulator supplied with the *vegan* data set on the Altair cluster with different number of nodes: (a) execution time comparison, (b) total performance gain.

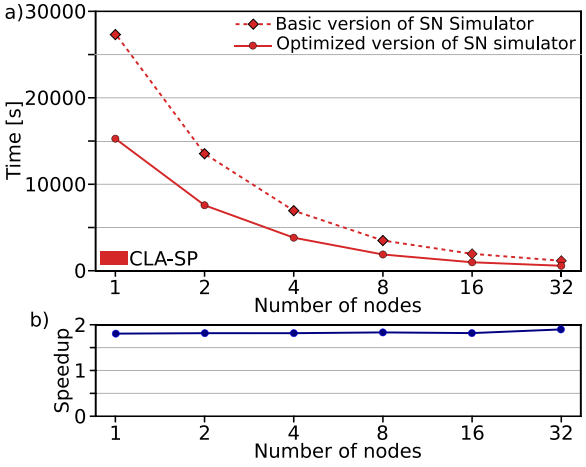


Fig. 12. Performance comparison between the basic and optimized versions of SN Simulator supplied with the *covid19* data set on the Altair cluster with different number of nodes: (a) execution time comparison, (b) total performance gain.

As expected, since the *vegan* data set features an essential number of costly tasks (see Section 4.4), the combination of proposed task-reordering and task-splitting approaches achieve noticeable partial performance gains of about 1.6x to 3.3x on 1 up 32 nodes, in comparison to the basic version. Extending the SN Simulator code of the Numba-based optimizations boosts the performance additionally by about 1.9 times.

In the next experiment, we consider our largest data set *covid19* with the configuration of parameters `samples` $=$ 5000 and `sources` $=$ 2000, which initiates simulation for more than $2 \cdot 10^9$ tweets. In Fig. 12, we plot the performance comparison between the basic and optimized versions of SN Simulator with the underlined configuration of *covid19* data set and the optimal value for the parameter `sample_split` $=$ 2, by exploring 1 to 32 nodes of Altair cluster.

For all tested numbers of nodes, the optimized version returns comparable performance improvements, executing the SN Simulator code 1.8–1.94 times faster than the basic version (Fig. 12b). Compared to the previous experiment, the overall workload consists of a larger amount of tasks that are smaller on average.

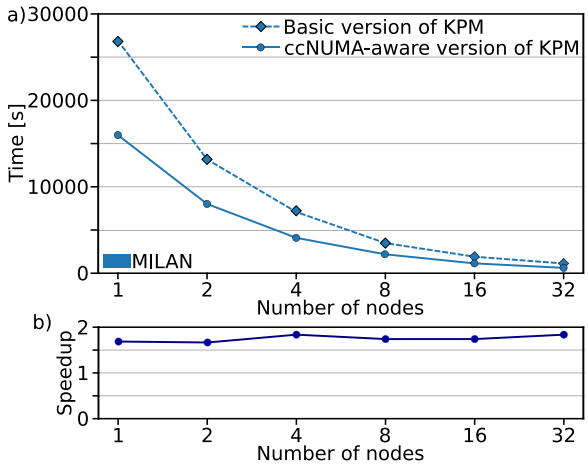


Fig. 13. Performance comparison between the basic and ccNUMA-aware versions of the KPM application obtained for the eigenvalue histogram of the Pokec graph on the cluster with 2x AMD EPYC 7763 per node: (a) execution time comparison, (b) total performance gain.

Additionally, the *covid19* data set seems less prone to producing disproportionately large tasks. Therefore, the proposed optimizations for workload distribution bring negligible partial performance gain, improving the performance by a factor of 1.03 to 1.11. In contrast, applying Numba-based optimization allows further performance improvements, reducing the execution time by about 1.73–1.78 times.

6.2. Evaluation of KPM on the cluster with AMD MILAN CPUs

In the next stage of benchmarks, we examine the impact of the proposed ccNUMA-aware approach for the KPM code on performance by exploring the AMD-based cluster located at the HPC Centre of Excellence, Munich, Germany. This cluster consists of 32 nodes, with two AMD EPYC 7763 CPUs of 64 cores each, and interconnected with InfiniBand HDR. We investigate the KPM application scenario that computes an approximated eigenvalue histogram of the Pokec input graph with 128 bins.

Fig. 13 illustrates the performance comparison between the basic and ccNUMA-aware versions of the KPM application obtained for configuration with `intervals` = 128, `samples` = 128, and `degree` = 300 on 1 up to 32 nodes of the AMD-based cluster. Compared to the previous experiments on a single node, these larger parameter values yield a more accurate approximation of the eigenvalue spectrum at increased computational cost. The resulting histogram has a large enough amount of bins and high enough accuracy to be used in the graph validation process we described in Section 5.1. The performed tests indicate that the proposed optimizations boosts the performance of KPM code, for all tested numbers of nodes (Fig. 13a).

The ccNUMA-aware code modification manages to accelerate the application execution by about 1.64–1.81 times faster (Fig. 13b). Considering the performance results presented in previous experiments (see Table 4), we indicated comparable performance gains with an average improvement factor of about 1.7 for the system with 64-core AMD MILAN processors. At the same time, we observe excellent scalability for the KPM code, where both basic and optimized versions achieve near linear speedup up to 32 available nodes.

7. Conclusions

In this paper we explored optimization techniques for the social sciences simulations: SN Simulator and KPM. Both applications are based on simulating the flow of messages in a synthetic graph representing the social media environment.

In the case of the first application (SN), a large imbalance was noticed due to the suboptimal distribution of tasks to the processors. Changing the order of tasks, their division into smaller chunks and the use of the NUMBA compiler contributed to preventing load imbalance between MPI processes and improve the overall performance for all performed tests (over 4 times for the platform with ARM-based processors).

In the case of KPM, performance analysis revealed communication constraints that severely limit overall performance. In order to overcome communication obstacles, an improved method of data distribution to NUMA domains was proposed, thus reducing intra-processor data traffic and allowing more efficient use of available cores. This helped to improve the acceleration of selected processors (AMD EPYC 7742) by more than 2.7 times.

The implemented strategies were tested on the basis of 1-node test environments using modern processors from manufacturers such as: Intel, AMD and Huawei. In addition, to assess the impact of changes in application performance in a large-scale computing system, tests were carried out on the HPC clusters. It was noted that the optimized version of SN Simulator achieves a performance gain of about 5.6 times against the basic code when employing 32 nodes. The ccNUMA-aware code modification in KPM manages to accelerate the application execution by about 1.8 times faster compared to code without optimization.

CRediT authorship contribution statement

Lukasz Szustak: Term, Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Resources, Writing – original draft, Writing – review & editing, Visualization, Supervision. **Marcin Lawenda:** Term, Conceptualization, Methodology, Formal analysis, Resources, Writing – original draft, Writing – review & editing, Supervision, Project administration, Funding acquisition. **Sebastian Arming:** Methodology, Software, Investigation. **Gregor Bankhamer:** Methodology, Software, Writing – original draft. **Christoph Schweimer:** Conceptualization, Writing – original draft, Writing – review & editing. **Robert Elsässer:** Conceptualization, Writing – review & editing, Supervision, Project administration.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request

Acknowledgments

This work has been supported by the HiDALGO project and has been partly funded by the European Commission's ICT activity of the H2020 Programme under grant agreement number: 824115. This paper expresses the opinions of the authors and not necessarily those of the European Commission. The European Commission is not liable for any use that may be made of the information contained in this paper.

Calculations on the HPC system were performed at Poznan Supercomputing and Networking Center, with the help of the Ministry of Science and Higher Education, Poland 5052/H2020/2019/2.

The authors are grateful to Supermicro and AMD companies for granting access to HPC platforms.

The authors thank Christine Gfrerer, Florian Lugstein and Bernhard Geiger for collecting, managing and preparing the input data sets for the SN Simulator as well as for fruitful discussions in earlier stages of the project. We also would like to thank Jan Velimsky for collecting parts of the data and for many valuable discussions.

Availability

The source code of the SN Simulator [28] and KPM application [29] is freely available on github. Both applications are licensed under the GNU GPL 3.0 [30].

The follower–followee relationship graphs [31] and the sets of Tweets [32] used as an input for the SN Simulator can be found on the CKAN storage of the HiDALGO H2020 project. Various versions of the Pokec graph used as an input for the KPM application can also be found on the CKAN [33]. For the experiments in this paper, we used the graph denoted by `pokec_trimmed`. The remaining files in the directory consist of smaller subgraphs that were extracted from this graph via breadth-first search.

References

[1] E. Chattoe-Brown, Is agent-based modelling the future of prediction? Int. J. Soc. Res. Methodol. 26 (2) (2023) 143–155, http://dx.doi.org/10.1080/13645579.2022.2137923, arXiv:https://doi.org/10.1080/13645579.2022.2137923.

[2] F. Altarelli, A. Braunstein, L. Dall'Asta, R. Zecchina, Optimizing spread dynamics on graphs by message passing, J. Stat. Mech. Theory Exp. 2013 (09) (2013) P09011, http://dx.doi.org/10.1088/1742-5468/2013/09/P09011.

[3] D. Kempe, J. Kleinberg, É. Tardos, Maximizing the spread of influence through a social network, in: Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '03, Association for Computing Machinery, New York, NY, USA, 2003, pp. 137–146, http://dx.doi.org/10.1145/956750.956769.

[4] J. Leskovec, A. Krause, C. Guestrin, C. Faloutsos, J. VanBriesen, N. Glance, Cost-effective outbreak detection in networks, in: Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '07, Association for Computing Machinery, New York, NY, USA, 2007, pp. 420–429, http://dx.doi.org/10.1145/1281192.1281239.

[5] Data parallelism, https://en.wikipedia.org/wiki/Data_parallelism.

[6] K. Abdulaziz Alnowibet, I. Khan, K.M. Sallam, A. Wagdy Mohamed, An efficient algorithm for data parallelism based on stochastic optimization, Alexandria Eng. J. 61 (12) (2022) 12005–12017, http://dx.doi.org/10.1016/j.aej.2022.05.052, URL https://www.sciencedirect.com/science/article/pii/S1110016822003696.

[7] N. Ma, Y. Xia, V.K. Prasanna, Data parallelism for belief propagation in factor graphs, in: 2011 23rd International Symposium on Computer Architecture and High Performance Computing, 2011, pp. 56–63, http://dx.doi.org/10.1109/SBAC-PAD.2011.34.

[8] S. Catalán, F.D. Igual, J.R. Herrero, R. Rodríguez-Sánchez, E.S. Quintana-Ortí, Programming parallel dense matrix factorizations and inversion for new-generation NUMA architectures, J. Parallel Distrib. Comput. 175 (2023) 51–65, http://dx.doi.org/10.1016/j.jpdc.2023.01.004, URL https://www.sciencedirect.com/science/article/pii/S0743731523000047.

[9] R. Laso, O.G. Lorenzo, J.C. Cabaleiro, T.F. Pena, J.Á. Lorenzo, F.F. Rivera, CIMAR, NIMAR, and LMMA: Novel algorithms for thread and memory migrations in user space on NUMA systems using hardware counters, Future Gener. Comput. Syst. 129 (2022) 18–32, http://dx.doi.org/10.1016/j.future.2021.11.008, URL https://www.sciencedirect.com/science/article/pii/S0167739X21004374.

[10] J. Schwarzrock, H.M.G. de A. Rocha, A.C.S. Beck, A.F. Lorenzon, Effective exploration of thread throttling and thread/page mapping on NUMA systems, in: 2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2020, pp. 239–246, http://dx.doi.org/10.1109/HPCC-SmartCity-DSS50907.2020.00030.

[11] P. Caheny, L. Alvarez, S. Derradji, M. Valero, M. Moretó, M. Casas, Reducing cache coherence traffic with a NUMA-aware runtime approach, IEEE Trans. Parallel Distrib. Syst. 29 (5) (2018) 1174–1187, http://dx.doi.org/10.1109/TPDS.2017.2787123.

[12] Intel 64 and IA-32 Architectures Optimization Reference Manual, https://software.intel.com.

[13] AMD EPYC 7003 Series Processors, 2023, https://www.amd.com/en/processors/epyc-7003-series.

[14] Huawei Atlas 800 Inference Server, https://e.huawei.com/en/products/computing/ascend/atlas-800-inference-3000.

[15] L. Szustak, R. Wyrzykowski, O. T., V. Mele, Correlation of Performance Optimizations and Energy Consumption for Stencil-Based Application on Intel Xeon Scalable Processors, IEEE Trans. Parallel Distrib. Syst. 31 (11) (2020) 2582–2593.

[16] L. Szustak, et al., Architectural Adaptation and Performance-Energy Optimization for CFD Application on AMD EPYC Rome, IEEE Trans. Parallel Distrib. Syst. 32 (12) (2021) 2852–2866.

[17] C. Schweimer, C. Gfrerer, F. Lugstein, D. Pape, J.A. Velimsky, R. Elsässer, B.C. Geiger, Generating simple directed social network graphs for information spreading, in: WWW '22: The ACM Web Conference 2022, Virtual Event, Lyon, France, April 25 - 29, 2022, ACM, 2022, pp. 1475–1485, http://dx.doi.org/10.1145/3485447.3512194.

[18] Z. Xu, Q. Yang, Analyzing user retweet behavior on Twitter, in: 2012 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining, 2012, pp. 46–50, http://dx.doi.org/10.1109/ASONAM.2012.18.

[19] S. Petrovic, M. Osborne, V. Lavrenko, RT to win! predicting message propagation in Twitter, in: Proc. 15th International AAAI Conference on Web and Social Media, 5, 2011, pp. 586–589, 1.

[20] B. Suh, L. Hong, P. Pirolli, E.H. Chi, Want to be retweeted? Large scale analytics on factors impacting retweet in Twitter network, in: 2010 IEEE Second International Conference on Social Computing, 2010, pp. 177–184, http://dx.doi.org/10.1109/SocialCom.2010.33.

[21] L. Hong, O. Dan, B.D. Davison, Predicting popular messages in Twitter, in: Proceedings of the 20th International Conference Companion on World Wide Web, WWW '11, Association for Computing Machinery, New York, NY, USA, 2011, pp. 57–58, http://dx.doi.org/10.1145/1963192.1963222.

[22] X. Tang, Y. Quan, Q. Miao, R. Hou, K. Deng, Information propagation with retweet probability on online social network, in: H. Wang, H. Qi, W. Che, Z. Qiu, L. Kong, Z. Han, J. Lin, Z. Lu (Eds.), Intelligent Computation in Big Data Era, Springer Berlin Heidelberg, Berlin, Heidelberg, 2015, pp. 326–333.

[23] K. Lee, J. Mahmud, J. Chen, M. Zhou, J. Nichols, Who will retweet this? Automatically identifying and engaging strangers on Twitter to spread information, in: Proceedings of the 19th International Conference on Intelligent User Interfaces, IUI '14, Association for Computing Machinery, New York, NY, USA, 2014, pp. 247–256, http://dx.doi.org/10.1145/2557500.2557502.

[24] H. Zhao, G. Liu, C. Shi, B. Wu, A retweet number prediction model based on followers' retweet intention and influence, in: 2014 IEEE International Conference on Data Mining Workshop, 2014, pp. 952–959, http://dx.doi.org/10.1109/ICDMW.2014.152.

[25] Numba documentation (Ver. 0.53), https://numba.readthedocs.io/.

[26] E. Di Napoli, E. Polizzi, Y. Saad, Efficient estimation of eigenvalue counts in an interval, Numer. Linear Algebra Appl. 23 (4) (2016) 674–692, http://dx.doi.org/10.1002/nla.2048, URL https://onlinelibrary.wiley.com/doi/abs/10.1002/nla.2048, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/nla.2048.

[27] J. Leskovec, A. Krevl, SNAP Datasets: Stanford large network dataset collection, 2014, https://snap.stanford.edu/data/soc-pokec.html.

[28] SN Simulator Version 4.0, 2021, https://github.com/sarming/kpm/releases/tag/v0.4.

[29] KPM Application Version 4.0, 2021, https://github.com/sarming/propagation/releases/tag/v0.4.

[30] GNU General Public License v3.0, 2007, http://www.gnu.org/licenses/gpl.html.

[31] Anonymized follower-followee Twitter graphs, 2021, https://ckan.hidalgo-project.eu/dataset/metis-outer-graphs.

[32] Anonymized Tweets including features, 2021, https://ckan.hidalgo-project.eu/dataset/simulation-features-version-2.

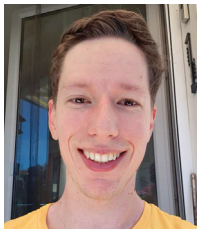
[33] Pokec graph in metis format, 2019, https://ckan.hidalgo-project.eu/dataset/pokec-relationship-graphs.

Lukasz Szustak received a D.Sc. (2019) and a Ph.D. (2012) in Computer Science (Parallel and Distributed Computation) granted by the Czestochowa University of Technology, Poland. His main research interests include parallel computing and mapping algorithms onto parallel architectures. His current work is focused on the development of methods for performance portability, scheduling, and load balancing, including the adaptation of scientific applications to modern HPC architectures.

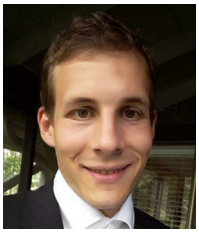
Marcin Lawenda (M) graduated from the Poznań University of Technology and received his M.Sc. in Computer Science (Parallel and Distributed Computation) in 2000. In 2006 he received Ph.D. degree at the same university. He has been working for Poznań Supercomputing and Networking Center for more than 20 years. ML is the project coordinator of HiDALGO2 and the leader of a national and European projects oriented on grid technology and instrumentation (e.g. HiDALGO, AMUNATCOLL, CoeGSS, e-IRGSP5, SGIGrid, RINGrid, DORII, Powiew, PRACE). His research interests include parallel and distributed environments, scheduling and Grid technologies especially in area of applied sciences. He is also author

and co-author of reports and papers (100+) in conference proceedings and journals. He has been a member of the Polish Information Processing Society since 2000.

Sebastian Arming received his Master's degree in Computational Intelligence from TU Wien, Austria, in 2015 and is currently a Ph.D. student at the University of Salzburg, Austria under supervision of Ana Sokolova. His research interests include complexity theory, formal methods and coalgebra. He currently focusses on the verification of probabilistic systems.



Gregor Bankhamer received his Master's degree in Computer Science from the University of Salzburg, Austria, in 2018. Currently he is a Ph.D. student at the same university under supervision of Robert Elsässer. His main research interest lies in the analysis of distributed algorithms and systems. His current work is focused on algorithms that involve randomization. Research problems he recently considered include resilient routing and plurality consensus.



Christoph Schweimer is a freelancing Data Scientist, who previously worked at the Know-Center GmbH in Graz, Austria. He received his Master's Degree in Mathematics from the Paris Lodron university in Salzburg, Austria in 2017. Christoph has been involved in publishing research in the areas of Machine Learning and Artificial Intelligence.

Robert Elsässer is a Professor at the University of Salzburg, Austria. He received his M.Sc. (1998) and Ph.D. (2002) from the University of Paderborn, Germany. From 2003 to 2011, Robert Elsässer worked as a Junior Professor at the University of Paderborn. During 2005/06 he was a visiting scientist at the University of California at San Diego, USA, and in 2009/10 a visiting professor at the University of Freiburg, Germany. His research interests focus on parallel and distributed algorithms, as well as on the design and analysis of large networks.