# Scalable Computing: Practice and Experience

## Scientific International Journal for Parallel and Distributed Computing

ISSN: 1895-1767

Universitatea de Vest din Timişoara

Volume 19(3) September 2018

EDITOR-IN-CHIEF

**Dana Petcu**
Computer Science Department
West University of Timisoara
and Institute e-Austria Timisoara
B-dul Vasile Parvan 4, 300223
Timisoara, Romania
Dana.Petcu@e-uvt.ro

MANAGINIG AND
TEXNICAL EDITOR

**Silviu Panica**
Computer Science Department
West University of Timisoara
and Institute e-Austria Timisoara
B-dul Vasile Parvan 4, 300223
Timisoara, Romania
Silviu.Panica@e-uvt.ro

BOOK REVIEW EDITOR

**Shahram Rahimi**
Department of Computer Science
Southern Illinois University
Mailcode 4511, Carbondale
Illinois 62901-4511
rahimi@cs.siu.edu

SOFTWARE REVIEW EDITOR

**Hong Shen**
School of Computer Science
The University of Adelaide
Adelaide, SA 5005
Australia
hong@cs.adelaide.edu.au

**Domenico Talia**
DEIS
University of Calabria
Via P. Bucci 41c
87036 Rende, Italy
talia@deis.unical.it

EDITORIAL BOARD

**Peter Arbenz,** Swiss Federal Institute of Technology, Zürich, arbenz@inf.ethz.ch
**Dorothy Bollman,** University of Puerto Rico, bollman@cs.uprm.edu
**Luigi Brugnano,** Università di Firenze, brugnano@math.unifi.it
**Giacomo Cabri,** University of Modena and Reggio Emilia, giacomo.cabri@unimore.it
**Bogdan Czejdo,** Fayetteville State University, bczejdo@uncfsu.edu
**Frederic Desprez,** LIP ENS Lyon, frederic.desprez@inria.fr
**Yakov Fet,** Novosibirsk Computing Center, fet@ssd.sscc.ru
**Giancarlo Fortino,** University of Calabria, g.fortino@unical.it
**Andrzej Goscinski,** Deakin University, ang@deakin.edu.au
**Frederic Loulergue,** Northern Arizona University, Frederic.Loulergue@nau.edu
**Thomas Ludwig,** German Climate Computing Center and University of Hamburg, t.ludwig@computer.org
**Svetozar Margenov,** Institute for Parallel Processing and Bulgarian Academy of Science, margenov@parallel.bas.bg
**Viorel Negru,** West University of Timisoara, Viorel.Negru@e-uvt.ro
**Moussa Ouedraogo,** CRP Henri Tudor Luxembourg, moussa.ouedraogo@tudor.lu
**Marcin Paprzycki,** Systems Research Institute of the Polish Academy of Sciences, marcin.paprzycki@ibspan.waw.pl
**Roman Trobec,** Jozef Stefan Institute, roman.trobec@ijs.si
**Marian Vajtersic,** University of Salzburg, marian@cosy.sbg.ac.at
**Lonnie R. Welch,** Ohio University, welch@ohio.edu
**Janusz Zalewski,** Florida Gulf Coast University, zalewski@fgcu.edu

SUBSCRIPTION INFORMATION: please visit http://www.scpe.org

# Scalable Computing: Practice and Experience

Volume 19, Number 3, September 2018

## TABLE OF CONTENTS

© SCPE, Timişoara 2018

DOI 10.12694/scpe.v19i3.1441
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 19, Number 3, pp. iii–iv. http://www.scpe.org © 2018 SCPE

# INTRODUCTION TO THE SPECIAL ISSUE ON INFRASTRUCTURES AND ALGORITHMS FOR SCALABLE COMPUTING

We are happy to present this special issue of the scientific journal Scalable Computing: Practice and Experience. In this special issue on Infrastructures and Algorithms for Scalable Computing (Volume 19, No 3 June 2018), we have selected four papers out of submitted nine, which gone through a peer review according to the journal policy. All papers represent novel results in the fields of distributed algorithms and infrastructures for scalable computing.

The first paper presents present a novel approach for efficient data placement, which improves the performance of workflow execution in distributed datacenters. The greedy heuristic algorithm, which is based on a network flow optimization framework, minimizes the total storage cost, including efforts to move and store the data from different source locations and dependencies. The second paper evaluated the significance of different clustering techniques viz. k-means, Hierarchical Agglomerative Clustering and Markov Clustering in grouping-aware data placement for data-intensive applications with interest locality. The evaluation in Azure reported that Markov Clustering-based data placement strategy improves the local map execution and reduces the execution time compared to Hadoops Default Data Placement Strategy and other evaluated clustering techniques. This is more emphasized for data-intensive applications that have interest locality. The third paper presents an experimental evaluation of the openMP thread-mapping strategies in different hardware environments (Intel Xeon Phi coprocessor and hybrid CPU-MIC platforms). The paper shows the optimal choice of thread affinity, the number of threads and the execution mode that can provide optimal performance of the LU factorization. In the fourth paper, the authors study the amount of memory occupied by sparse matrices split up into same-size blocks. The paper considers and statistically evaluates four popular storage formats and combinations among them. The conclusion is that block-based storage formats may significantly reduce memory footprints of sparse matrices arising from a wide range of application domains.

We use this opportunity to thank all contributors to this Special Issue: all authors who submitted the results of their latest research and all reviewers for their valuable comments and suggestions for improvement. We would like to express our special gratitude for the Editor-in-Chief, Professor Dana Petcu, for her constant support during the whole process of this Special Issue.

Sasko Ristov, University of Innsbruck, Austria

DOI 10.12694/scpe.v19i3.1365
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 19, Number 3, pp. 223–244. http://www.scpe.org © 2018 SCPE

# EXACT AND HEURISTIC DATA WORKFLOW PLACEMENT ALGORITHMS FOR BIG DATA COMPUTING IN CLOUD DATACENTERS

SONIA IKKEN[*], ERIC RENAULT[†], ABDELKAMEL TARI[‡], AND M. TAHAR KECHADI[§]

**Abstract.** Several big data-driven applications are currently carried out in collaboration using distributed infrastructure. These data-driven applications usually deal with experiments at massive scale. Data generated by such experiments are huge and stored at multiple geographic locations for reuse. Workflow systems, composed of jobs using collaborative task-based models, present new dependency and data exchange needs. This gives rise to new issues when selecting distributed data and storage resources so that the execution of applications is on time, and resource usage-cost-efficient. In this paper, we present an efficient data placement approach to improve the performance of workflow processing in distributed datacenters. The proposed approach involves two types of data: splittable and unsplittable intermediate data. Moreover, we place intermediate data by considering not only their source location but also their dependencies. The main objective is to minimise the total storage cost, including the effort for transferring, storing, and moving that data according to the applications needs. We first propose an exact algorithm which takes into account the intra-job dependencies, and we show that the optimal fractional intermediate data placement problem is NP-hard. To solve the problem of unsplittable intermediate data placement, we propose a greedy heuristic algorithm based on a network flow optimization framework. The experimental results show that the performance of our approach is very promising.

**Key words:** Big data placement, Data workflow management, Dataflow model, Distributed datacenters, Storage cost minimization, Scalable storage and computing

**AMS subject classifications.** 15A15, 15A09, 15A23

## 1. Introduction.

This study addresses the problem of intermediate data placement in big data-driven applications. These data are usually stored in multiple cloud datacentres. The main goal is to be able to efficiently process and share them between a set of tasks (jobs or services) according to their needs, dependencies, and their resource requirements. In other words, the problem is how to efficiently store and access the intermediate data taking into account the inter- and intra-job (process) dependencies. These jobs can be any traditional process at the level of the operating system or services at the application level. Because of the popularity of the service delivery model, the cloud consists of a set of services that should be provided to the clients. At a very large scale (cloud scale), a huge number of activities coexist generating or consuming huge amount of intermediate data, which are stored in the form of files, called temporary files, in datacentres. From the point of view of data usage, this follows a workflow depending on the dynamic nature the execution of these services. These dependencies are of very high importance for the correct execution of the services that were provided to the clients. Each workflow has different requirements not only in the dependencies of its intermediate data files but also their sizes and types. Moreover, these services (jobs or set of tasks) can be initiated remotely from different geographic locations, which adds a level of complexity of how these data can be accessed without putting a significant stress on the cloud resources (bottlenecks, long waiting queues, etc.). Therefore, the way that these intermediate data files should be manipulated and managed must take into account their near future usage (frequency of use, life cycle, etc.). Accordingly, their management (storage, movement, processing, etc.) should be derived from the workflow of the jobs and services that are using them, which is called "Data Workflow".

This paper proposes a new approach that takes into account the types of dependencies and accesses to the intermediate data, as these are the key factors for improving big data-driven applications while leveraging cloud resources among the clients in an efficient and scalable manner. We start by formulating the intermediate data placement dependencies derived from multiple workflows running on distributed cloud datacenters. Then we model the whole system as a constrained optimization problem, where constraints represent the dependencies derived from the running workflows. The derived constrained optimization problem that includes storage cost is very complex. Two types of data are considered; the intermediate data that are used by the same job with

[*]Faculty of Exact Sciences, University of Bejaia, 06000 Bejaia, Algeria, and Telecom SudParis, Samovar-UMR 5157 CNRS, University of Paris-Saclay, France (sonia.ikken@telecom-sudparis.eu, sonia.ikken@gmail.com).

[†]Telecom SudParis, Samovar-UMR 5157 CNRS, University of Paris-Saclay, France (eric.renault@telecom-sudparis.eu).

[‡]Faculty of Exact Sciences, University of Bejaia, 06000 Bejaia, Algeria (tarikamel59@gmail.com).

[§]UCD School of Computer Science and Informatics, Dublin, Ireland (tahar.kechadi@ucd.ie).

their intra-dependencies from multiple tasks and intermediate data that are used by different jobs with their inter-job dependencies. Since intra-job dependencies can be split partially and placed on different locations (as in MapReduce), the problem is called minimum cost multiple-sources multicommodity flow problem (MCMF). Intermediate data dependencies that are used by different tasks (of a single job) can be split and kept in the same datacenter and preferably at the location of the task. We formulate the problem with these data as splittable demands which can be solved with an exact algorithm to obtain an optimal fractional solution. However, as most of these problems are NP-hard, it is difficult to obtain an optimal solution using exact methods for the variant that deals with unsplittable intermediate data from inter-job dependencies. Greedy approaches implement simple algorithms but effective for unsplittable flow problem [1, 2, 3, 4, 5], and they scale linearly with the number of instances. Their resolution techniques are adaptable to our intermediate data placement problem that the present paper deals with. Experimental results show that the proposed techniques are very promising for storage cost minimisation.

The rest of the paper is organized as follows: Section 2 summaries the related work. Section 3 introduces the system model and problem definition according to the cloud computing environment and data models. The proposed techniques for the optimal intermediate data dependencies placement are presented in Section 4. Section 5 discusses the performance evaluation and the simulation results. We conclude and give some future directions in Section 6.

**2. Related works.** We have thoroughly investigated recent research works on cloud ressource scheduling in the literature [31, 32, 33]. These works focus primarily on resource sharing and provisioning problems in order to either save energy consumption or reduce its costs by providing efficient application processing. Authors in [31], addressed the power consumption problem and network performance degradation by relying on an optimization model that is based on sliding-scheduled tenant request. The latter allows to manage application execution time as well as their resources for efficient placement and routing. Authors in [32], proposed a genetic algorithmic based heuristic methods to schedule tasks across limited resources, but restricted to use one global cost and time for multiple tasks execution. More recently, authors in [33], addressed the problem of resource scheduling of scientific workflow applications in cloud. They focus on reducing the cost of the communications and the information exchange time across a management framework of multiple-site awareness data administration. Nevertheless, these works did not address the problem of data workflow placement and the dependencies between resources.

Workflow scheduling problems [35, 36] in cloud environments are considered to be very challenging. Many strategies based-heuristic were proposed to solve the tasks scheduling problem without considering the data that are generated by these tasks. Authors in [35], proposed a meta-heuristic approach, called Hybrid GA-PSO (Genetic Algorithm-Particle Swarm Optimization), to solve the workflow tasks scheduling problem. The Hybrid GA-PSO algorithm returns a balanced solution for tasks distribution among different virtual machines in a cloud environment by considering both the total monetary cost and the execution makespan. While the PSO-based algorithm converges quickly to a local optimal solution, this can be far from the global optimal solution. In the same context, authors in [36] proposed a metaheuristic-based algorithm, called Hybrid Bio-inspired Metaheuristic for Multi-objective Optimization (HBMMO), to solve the multiple conflicting objectives optimization problem. The authors considered in their optimization some important requirements of the users or the providers, such as makespan, cost, and load balancing among virtual machines. The proposed HBMMO method optimizes the scheduling of tasks workflow in the cloud environment by considering a non-dominant sorting strategy which is a hybridization of the list-based heuristic algorithm PEFT (Predict Earliest Finish Time) [37] and the discrete version of the metaheuristic algorithm SOS (Symbiotic Organisms Search) [38].

Many researchers [34, 6, 7, 8, 9] have focused on the big data placement optimization problem in distributed system environments. However, most of these studies did not include the dependencies between data workflows. In addition, these solutions did not take into consideration the dependency type constraint for making intermediate data placement decision. Authors in [34], defined an optimization problem based on a greedy heuristic for simultaneous placement of virtual machines and data blocks. A greedy heuristic allows to place on-demand application components by localising network traffic in interconnected datacenters, and therefore, reducing packet transmission delays, increasing network performance, and minimizing the energy consumption of datacenter network infrastructure. Nevertheless, the efficiency of multi-tier application processing through the data communication and correlation is not considered. Authors in [6], proposed a strategy placement for large-volume user's data while minimizing their operational costs of accommodating various social networks. The relation between the user community and dynamic maintenance of the placed user data in an evolving social network are considered, but the use of the data dependency aspects is not explored. In [7], the big data placement problem from a collaborative-aware environment that continuously generates data from different geographical locations has been studied. The authors developed a solution to save the high cost incurring when managing the distributed big data, and they proposed an approximation algorithm by reducing the data placement problem to the minimum cost multicommodity flow problem. Their solution addressed a data placement respecting a fair usage of the cloud services like the quality of service (QoS) requirement of cloud provider while savings computation and bandwidth costs. The solution is closely similar to our context but differs mainly on the conditions and characteristics of data workflow aspects and by no means disclosing intermediate data dependencies. They focused more on maximizing the system throughput in terms of data volume to be placed while saving the computing/storage and communication costs in the distributed datacenters. Sharing intermediate data from computation produced between different workflow MapReduce jobs is studied in [8]. The authors presented a scheduling technique for data-driven jobs sharing opportunities that involves the scan of the input file with the goal of maximizing the likelihood of sharing scans. A similar optimization approach is presented in [9]. A cost model is presented saving processing time and money for MapReduce jobs in order to define an optimization problem that finds an optimal grouping of set of queries and solves it using a dynamic programming approach. These works present a data-driven job scheduling issue which is not exactly the same as the data placement problem. These do not focus on the intermediate data scheduling optimization as well as the incurred storage cost.

Workflow scheduling problems [35, 36] in cloud environments are considered to be very challenging. Many strategies based-heuristic were proposed to solve the tasks scheduling problem without considering the data that are generated by these tasks. Authors in [35], proposed a meta-heuristic approach, called Hybrid GA-PSO (Genetic Algorithm-Particle Swarm Optimization), to solve the workflow tasks scheduling problem. The Hybrid GA-PSO algorithm returns a balanced solution for tasks distribution among different virtual machines in a cloud environment by considering both the total monetary cost and the execution makespan. While the PSO-based algorithm converges quickly to a local optimal solution, this can be far from the global optimal solution. In the same context, authors in [36] proposed a metaheuristic-based algorithm, called Hybrid Bio-inspired Metaheuristic for Multi-objective Optimization (HBMMO), to solve the multiple conflicting objectives optimization problem. The authors considered in their optimization some important requirements of the users or the providers, such as makespan, cost, and load balancing among virtual machines. The proposed HBMMO method optimizes the scheduling of tasks workflow in the cloud environment by considering a non-dominant sorting strategy which is a hybridization of the list-based heuristic algorithm PEFT (Predict Earliest Finish Time) [37] and the discrete version of the metaheuristic algorithm SOS (Symbiotic Organisms Search) [38].

The research works that considered data workflow features are presented in [10, 11, 12, 13, 14]. Nevertheless, the dynamic variation of inter and intra-jobs dependencies from the generated intermediate data was not addressed with the same focus. In [10], an adaptive data-task placement approach is proposed that reflects asynchronous coupling among tasks in order to reduce execution time and data movement overhead. The authors in [11] have dealt with improving the data workflow's execution by clustering the interdependent datasets and distribute them intelligently onto the same datacenters to reduce data transfers. In [14], the authors established a data placement algorithm based on data dependency clustering and recursive partitioning. The aims of the algorithm are to reduce the amount of transmitted data and the time consumption during data-intensive application execution. The pursued strategy is extended with a heuristic to make frequent data movements occuring on high-bandwidth channels of the entire cloud system.

## 3. System model.

**3.1. Cloud storage infrastructure and assumptions.** For the intra- and inter-job data workflow placement problem depicted in Fig. 3.1, the objective is to route and store a set of intermediate data considering their dependencies generated by a collaborative tasks[1] from multiple physical sites while saving their opera-

[1]Tasks are launched and executed from an environment where scientific users collaborate and conduct their research together.

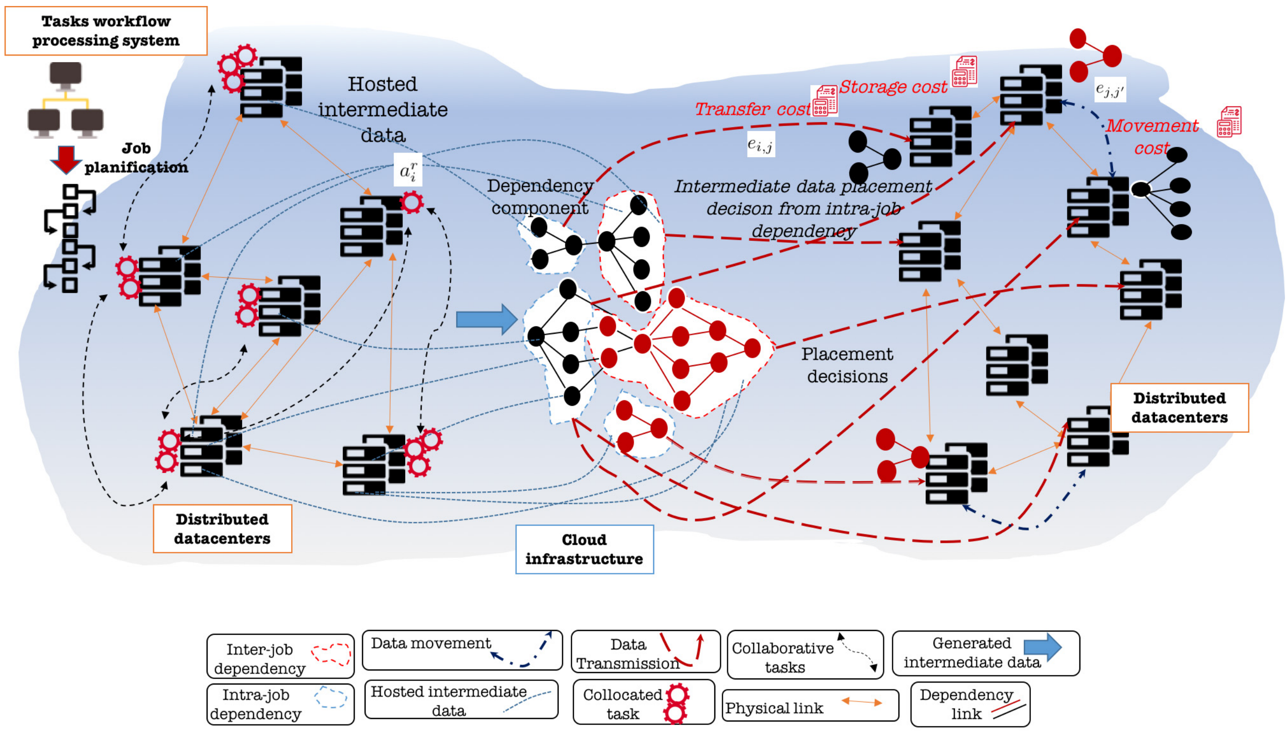


FIG. 3.1. *The system architecture.*

tional costs. Without loss of generality, we assume that the collaborative tasks, which process and generate new intermediate data files, are previously assigned to the cloud infrastructure (model task assignment offered by a cloud infrastructure). Since the intermediate data dependency placements are our most significant concern, we assume that the tasks were assigned to computing nodes following some simple model. The problem of placing the intermediate data files is close to the well-known MCMF problem; an optimization problem described in [15] that involves simultaneously shipping multiple commodities through a single graph, so the total flow obeys the arc capacity constraints by optimizing the cost.

The modeling starts by considering a set of geographically distributed datacenters[2] as a directed graph-based model $G = (DC \cup A, E)$. It forms a cloud infrastructure and constructs a shared computation and storage limited to a set of resources for processing and storing the data workflow. Users, such as enterprises, institutions or researchers, that own and share a cloud infrastructure issued from providers, have an access to the distributed datacenters ($DC$) to process multiple collaborative tasks into multiple processing phases. The distributed datacenters known as storage containers cohabit with collaborative task $A$ through one or multiple jobs $r$ running in parallel [16]. A set of tasks are collocated on multiple source datacenters, and each task $a_i^r \in A$ from job $r$ is assigned to source datacenter $dc_i$. Let $\{e_{i,j}, e_{j,j'}\} \in E$ be the intermediate data transfer and movement (initial and dynamic intermediate data routing respectively) links between source datacenter $dc_i$ and destination datacenter $dc_j$ and between destination datacenters $dc_j$ and another destination datacenter $dc_{j'}$, which are geographically interconnected via the Internet. The placement of the intermediate data dependencies to the set of datacenter destinations $dc_j \in DC$ is considered at the beginning of each phase.

**3.2. Intermediate data dependency model.** Placing intermediate data with the same correlation to a single destination datacenter can significantly decrease the amount of data dependency movements [17]. This leads to consider a vector of all intermediate data files denoted by $\Phi^M$ and $|\Phi^M|$ its size, representing the

[2]The security and communication management aspects in a collaborative processing are supposed to be covered by the cloud SLA policy in a cloud environment.

correlations among them that are generated during the workflow phases divided into equal period of time $t$. These correlations, which reflect the intra- and inter-job dependencies from a set of intermediate data files, are recovered into dependency component $m \in M$. $M$ contains all the different components of dependency that are modeled by a Directed Acyclic Graph (DAG) which takes the advantage of a topology ordering, thus defining relations among nodes [18, 19]. The DAG represents a set of intermediate data files $\phi^m_{a^r_i}(t)$. Let $|\phi^m_{a^r_i}(t)|$ be their respective sizes. These data files have unavoidable complex dependencies that are generated by single task $a^r_i \in A$ from job $r$ at source datacenter $dc_i$. In DAG, set of files $\phi^m_{a^r_i}(t)$, from the inter-job dependencies, are atomic and must be synchronized for their processing. By contrast, for the intra-job dependencies, these files are deduced from a partial correlation with an asynchronous processing [10]. Let $\phi^m(t)$ and $\phi^m_i(t)$ be the intermediate data of a single dependency component $m$ generated at multiple datacenters and the single home datacenter $dc_i$ respectively and, $|\phi^m(t)|$ and $|\phi^m_i(t)|$ be their respective sizes. At the end of each workflow phase, generated intermediate data file $\phi^m_i(t) \in \Phi^m$ must be outsourced and placed through data transfer link $e_{i,j} \in E$ from datacenter $dc_i$ to $dc_j$ for persistent storing or future reuse [20, 21]. It is important to note that the set of dependency components $M$ and the related type are a predetermined value given by scientific user that can be obtained through the data analysis clustering method [22]. We assume that the intermediate data dependencies clustering is given a priori and is beyond the scope of the present work.

**3.3. Capacity and cost model.** To come up with an intermediate data dependency placement from the collaborative task workflow execution in cloud datacenters, we take into consideration the fact that all datacenters and network resources are limited [23, 24]. Thus, let $S_j$ be the storage capacity of datacenter destination $dc_j \in DC$, and $W_{i,j}$, $W_{j,j'}$ be the bandwidth capacities of the data file transfer and movement links $e_{i,j}, e_{j,j'} \in E$ respectively. In order to manage and transfer these files, a data bandwidth denoted $w_\phi$ is assigned for one unit of intermediate data file. During a run-time phase, the available amount of storage capacity in datacenter $dc_j$, when transferring an amount of intermediate data files $\phi^m_i(t)$, is denoted by $s^{avail}_{i,j}(t)$. Let $w^{avail}_{i,j}(t)$, $w^{avail}_{j,j'}(t)$ be the available capacities of a data transfer and movement of links $e_{i,j}, e_{j,j'} \in E$. In addition, transferring and storing the intermediate data dependencies from source datacenter $dc_i$ into destination datacenter $dc_j$ are facing in both storage resource cost and scale. However, they usually consume high costs in a cloud infrastructure due to an inefficient utilization of the resources [25]. In practice, these resource demands are leading to operational cost specifically for data transfers and storage costs (measured per one unit in GB) that embrace the usage-based pricing policy [9]. Moreover, reused intermediate data dependencies that are not locally stored but remotely served on data demands are led to an additional cost, as movement cost, which is deducted from their migration among datacenter destinations [14]. In fact, these operational storage costs are related to the size of the intermediate data files that are transferred, stored and moved among distributed datacenters according to their correlation during each run-time phase. Moreover, each datacenter destination $dc_j \in DC$ is preserved to the geographical area where it is located [26], thus holding a storage cost noted $c_{s_j}$. The proportion of intermediate data dependencies $\phi^m$ of a single dependency component generated from multiple source datacenters and placed separately into different locations $dc_j$ and $dc_{j'}$ are led to a potential dependency movement cost. For clear differentiation from the transfer cost, we assume that the cost of intermediate data movement is proportional to the number of intermediate data dependency files transmitted between datacenter destinations. Therefore, the movement cost is defined as the amount of data moved among two or multiple destination datacenters. Hence, each link $e_{i,j}, e_{j,j'} \in E$ entry faces data bandwidth cost $c_{w_\phi}$.

For the sake of easier reading, Table 3.1 summarizes the notations used in the present work.

**4. Placement algorithms.** From the intermediate data dependency placement issue that reduced to an MCMF problem in $G$, two variants are materialized. In fact, in the case of intra-job dependency type, the routing of intermediate data dependencies can be performed using multiple links. When this assumption is omitted, i.e. when splittable flow routing can be used, variable of the optimization problem becomes continuous and as a consequence the considered problem becomes easier to solve. In contrast, in the case of inter-job dependency type, the routing of intermediate data dependencies in $G$ cannot be fractionated. Thus, the MCMF problem seems to be hard to solve. For this aim, we formulate the intra-job splittable dependency placement based on a Linear Program (LP) approach, and a heuristic approach is proposed in this section as an approximation algorithm for the intra-job unsplittable dependency placement.

Table 3.1
*Symbols for the model*

| **Notation** | **Description** |
|---|---|
| $G$ | The cloud infrastructure (provider) |
| $DC$ | A set of distributed datacenters in cloud infrastructure $G$ |
| $t$ | The run-time window which represents the homogeneous discrete time slot from generated collaborative data-tasks workflow processing |
| $A$ | The set of collaborative tasks in distributed datacenter $DC$ |
| $E$ | The set of links among distributed datacenter $DC$ |
| $i, j, j'$ | Indices used to designate distributed datacenters. $i$ belongs to source datacenter $dc_i$, while $j$ and $j'$ belong to different destination datacenters ($dc_j$ and $dc_{j'}$) |
| $e_{i,j}$ | Data transfer link between source datacenter $dc_i$ and destination datacenter $dc_j$ |
| $r$ | Workflow job in the system |
| $a_i^r$ | The collocated task in source datacenter $dc_i$ |
| $dc_i$ | A source datacenter temporarily storing generated intermediate data from collocated task $a_i^r$ of job $r \in R$ |
| $dc_j$ | A datacenter destination where the intermediate data files are to be placed |
| $M$ | The set of dependency components in the system including correlation among generated intermediate data |
| $\phi^m(t)$ | The intermediate data files of a single dependency component $m$ generated in multiple datacenters at time slot $t$, and $\|\phi^m(t)\|$ its size |
| $\phi_i^m(t)$ | The intermediate data files generated in datacenter $dc_i$ from dependency component $m \in M$ at time slot $t$, and $\|\phi_i^m(t)\|$ its size |
| $\phi_{a_i^r}^m(t)$ | The intermediate data files generated by task $a_{ir}$ of dependency component $m$ at time slot $t$, and $\|\phi_{a_i^r}^m(t)\|$ its size |
| $\Phi^M$ | All generated intermediate data files in the system, and $\|\Phi^M\|$ its size |
| $L_\phi$ | The vector list of intermediate data of all dependency components $m \in M, m = 1, ..., k$ |
| $w_\phi$ | The data bandwidth assigned to one unit of intermediate data file $\phi_{a_i^r}^m(t)$ |
| $W_{i,j}$ | The data bandwidth capacity of movement link $e_{i,j} \in E$ |
| $w_{i,j}^{avail}(t)$ | The available amount of data transfer link $e_{i,j} \in E$ at time slot $t$ |
| $W_{j,j'}$ | A data bandwidth capacity of movement link $e_{j,j'} \in E$ |
| $w_{j,j'}^{avail}(t)$ | The available amount of data transfer link $e_{j,j'} \in E$ at time slot $t$ |
| $s_{i,j}^{avail}(t)$ | The available amount of storage space when transferring an amount of intermediate data files from source datacenter $dc_i$ to destination datacenter $dc_j$ at time slot $t$. |
| $S_j$ | The data storage capacity of destination datacenter $dc_j \in DC$ |
| $x_{i,j}^m(t)$ | A decision variable reflecting the amount of intermediate data flow moving from source datacenter $dc_i$ of dependency component $m$ to destination datacenter $dc_j \in DC$ at time slot $t$. |
| $x_{j,j'}^m(t)$ | A decision variable reflecting the amount of intermediate data dependency component $m$ moving between destination datacenters $dc_j, dc_{j'} \in DC$ at time slot $t$ |
| $c_{s_j}$ | The storage cost of one unit of intermediate data in datacenter destination $dc_j \in DC$ |
| $c_{w_\phi}$ | The data bandwidth cost of one unit of intermediate data |
| $f(\phi_{a_i^r}^m)$ | A dependency component flows in graph $G_p$ |
| $f(\phi^m)$ | All flows from a single dependency component in graph $G_p$ |
| $ShP_\phi$ | The shortest path from $s_{source}$ to $s_{sink}$ in $G_p$ |

**4.1. Exact algorithm.** This section presents an exact analytical algorithm for splittable variant of the intermediate data dependency placement problem from multiple datacenters in cloud infrastructure $G$. The exact algorithm is an LP model with the inclusion of valid conditions expressed in the form of constraints or inequalities. Through the constraints of the problem, the intermediate data placement in a directed graph $G = (DC \cup A, E)$ at time slot $t$ is to route and place intermediate data dependencies $\phi^m(t) \in \Phi^M$ that are considered as continuous commodity flows of dependency component $m$ from multiple source datacenters to one or multiple destination datacenters while saving their transfer, storage and movement costs. A number of decision variables and valid inequalities (as listed for convenience in Table 3.1) are thus defined as follows:

**1) Decision variables:** Let $x^m_{i,j}(t) \in \mathbb{R}$ be the intermediate data of one dependency component $m$ standing for the amount of intermediate data dependency flows transferring from source datacenter $dc_i$ at time slot $t$ to destination datacenter $dc_j$ at time slot $t+1$ on link $e_{i,j} \in G$. In order to take into account the amount of intermediate data dependencies that are moved among different destination datacenters $dc_j$, $dc_{j'}$, we add variable $x^m_{j,j'}(t) \in \mathbb{R}$.

**2) Flow conservation constraint:** One typical constraint or requirement is to ensure that at all time, every flow through directed graph $G$ is physically possible. First, we enforce flow continuity by making sure that the sum of intermediate data dependency flows leaving from source datacenter $dc_i$ at time slot $t-1$ is equal to $\phi^m_i(t)$ which is the sum of flows arriving from the same datacenter $dc_i$ that are considering the same dependency component $m$ at time slot $t$. Formally:

$$\sum_{j \in DC} x^m_{i,j}(t) - \sum_{j \in DC} x^m_{j,i}(t-1) = \phi^m_i(t) \quad \forall m, t, i. \tag{4.1}$$

**3) Capacity constraint of intermediate data flows:** Each intermediate data dependency flow $x^m_{i,j}(t)$ may have its own individual capacity constraint which represents a lower bound on dependency component commodity $m$ through link $e_{i,j}$. This ensures the atomicity of lower bound $\phi^m_{a^r_i}(t)$ on $x^m_{i,j}(t)$ of which all these flows take a same link $e_{i,j}$, hence:

$$0 \leq \phi^m_{a^r_i}(t) \leq x^m_{i,j}(t) \quad \forall i, j, a^r_i, m, t. \tag{4.2}$$

**4) Capacity constraint of data transfer links:** In $G$, each link $e_{i,j}$ may have a capacity constraint like the data bandwidth routing constraint. Equation (4.3) ensures that the routing of aggregate intermediate data dependencies is limited by the available amount of data bandwidth allocated on link $e_{i,j}$ at time slot $t$:

$$\sum_{m \in M} w_\phi \cdot |\phi^m_i(t)| \cdot x^m_{i,j}(t) \leq w^{avail}_{i,j}(t) \quad \forall i, j, t. \tag{4.3}$$

Additionally, link $e_{i,j}$ is bounded by the data bandwidth capacity at all system execution time, hence:

$$\sum_{m \in M} \sum_{t \in T} w_\phi \cdot |\phi^m_i(t)| \cdot x^m_{i,j}(t) \leq W_{i,j} \quad \forall i, j. \tag{4.4}$$

**5) Capacity constraint of data movement links:** In $G$, each link $e_{j,j'}$ may have a capacity constraint like the data bandwidth routing constraint. Equation (4.5) ensures that moving intermediate data dependencies is limited by the available amount of data bandwidth allocated on link $e_{j,j'}$ at time slot $t$:

$$\sum_{m \in M} \sum_{a^r_i \in A} w_\phi \cdot |\phi^m(t) - \phi^m_{a^r_i}(t)| \cdot x^m_{j,j'}(t) \leq w^{avail}_{j,j'}(t) \quad \forall j, j', t. \tag{4.5}$$

Additionally, the link $e_{j,j'}$ is bounded by the data bandwidth capacity at all system execution time, hence:

$$\sum_{m \in M} \sum_{a^r_i \in A} \sum_{t \in T} w_\phi \cdot |\phi^m(t) - \phi^m_{a^r_i}(t)| \cdot x^m_{j,j'}(t) \leq W_{j,j'} \quad \forall j, j'. \tag{4.6}$$

**6) Capacity constraint of dependency component:** A uniqueness constraint is used to ensure that the routed intermediate data dependency flows do not exceed the dependency corresponding component capacity. Formally:

$$\sum_{i\in DC} x^m_{i,j}(t) \le \phi^m_i(t) \quad \forall j, m, t. \tag{4.7}$$

**7) Storage capacity constraint:** Each destination datacenter has a limited amount of storage space available to share across all the intermediate data placement demands. This allows to host only a limited amount of intermediate data dependencies from source datacenter $dc_i$ to destination datacenter $dc_j$. Formally:

$$\sum_{m\in M} |\phi^m_i(t)| \cdot x^m_{i,j}(t) \le s^{avail}_{i,j}(t) \quad \forall i, j, t. \tag{4.8}$$

For any intermediate data placement demands, the data routing must not exceed the total storage capacity at all system execution time. Formally:

$$\sum_{m\in M}\sum_{t\in T} |\phi^m_i(t)| \cdot x^m_{i,j}(t) \le S_j \quad \forall i, j. \tag{4.9}$$

**8) Balancing constraint:** Since the collaborative tasks in the workflow processing generate the intermediate data dependencies in multiple phases, these latter may vary over time in the distributed datacenter environment. In other words, the flow sequence of generated intermediate data dependencies changes as commodity changes. Thus, the flows among the distributed datacenters must be balanced. Hence, source and sink nodes $s_{source}$ and $s_{sink}$ are respectively introduced in graph $G$. Source node $s_{source}$ is connected to every source datacenter $dc_i$, and sink node $s_{sink}$ is connected to every destination datacenter $dc_j$. Source and sink nodes are also subject to a constraint that enforces all the intermediate data dependency flows starting on $s_{source}$ to ending at $s_{sink}$. Formally:

$$\sum_{i\in DC} x^m_{s_{source},i} = \sum_{j\in DC} x^m_{j,s_{sink}} \quad \forall m \in M \tag{4.10}$$

**9) Data transfer cost:** Equation (4.11) denotes the data transfer cost on link $e_{i,j}$ which intermediate data dependency flows are routed.

$$C(w_{i,j}) = \sum_{i\in DC}\sum_{j\in DC}\sum_{m\in M}\sum_{t\in T} |\phi^m_i(t)| \cdot x^m_{i,j}(t) \cdot w_\phi \cdot c_{w_\phi} \tag{4.11}$$

**10) Storage cost:** Equation (4.12) denotes the storage cost of destination datacenter $dc_j$ which intermediate data dependency flows are routed. Formally:

$$C(s_j) = \sum_{i\in DC}\sum_{j\in DC}\sum_{m\in M}\sum_{t\in T} |\phi^m_i(t)| \cdot x^m_{i,j}(t) \cdot c_{s_j} \tag{4.12}$$

**11) Data movement cost:** The proportions of intermediate data $\phi^m$ from one dependency component that are stored separately into different locations $dc_j$ and $dc_{j'}$ are led to potential intermediate data dependency movement cost. With no loss of generality, it is assumed here that the amount of intermediate data that moves from $dc_j$ to $dc_{j'}$ is defined as the set of intermediate data of a single dependency component $m$ that is fractionated from the set of atomic $\phi^m_{a^r_i}(t)$. Formally:

$$C(w_{j,j'}) = \sum_{i\in DC}\sum_{j,j'\in DC}^{j\neq j'}\sum_{m\in M}\sum_{t\in T} |\phi^m(t) - \phi^m_{a^r_i}(t)| \cdot x^m_{j,j'}(t) \cdot w_\phi \cdot c_{w_\phi} \tag{4.13}$$

**12) Objective function:** The objective of the intermediate data placement problem is to find, for a given set of dependency flows $x^m_{i,j}(t)$, a set of destination datacenters that can place them to minimize the aggregate cost of transferring, storing and moving intermediate data dependencies. This can be expressed using the following expression:

$$Minimize \ \ (C(w_{i,j}) + C(s_j) + C(w_{j,j'})) \tag{4.14}$$

Under the formulation listed above, the LP model is polynomial. However, the optimization is carried out with respect to flows $x^m_{i,j}(t)$ that are bounded and constrained as a result of the amount of intermediate data dependencies $\phi^m_{a^r_i}(t)$ generated by a single task $a^r_i$. This converges the exact algorithm into a non-polynomial time regarding to size $|\phi^m_{a^r_i}(t)|$ on very large instances when the splitting of flows $x^m_{i,j}(t)$ becomes marginal. Since a dependency component cannot start before the intermediate data dependencies of their predecessors are materialized, the unsplittable version of the intermediate data placement problem considering all flows for each dependency component from inter-job must be sent along a single link, making the problem NP-hard [15]. Due to the intractability of the problem, a heuristic is presented to address larger scale instances in a reasonable time.

**4.2. Heuristic approach.** The intra-job dependency placement solution is compared to the solution of the exact approach and requires the placement of the amount of intermediate data dependencies into a single destination datacenter. Thus, a naive greedy solution considers an integer commodity of dependency component $m$ from different sources as a single source flow unlike the exact approach that tolerates multiple source of dependency component $m$ independently when solving the problem. Under the unsplittable solution, a commodity is never split along multiple paths during the placement decision. Furthermore, the greedy approach applies a routine procedure in specific graph $G_p$, and assume that the minimum demands are less than or equal to the maximum capacity of the nodes in graph $G_p$ [15]. The latter involving less connection, the local search of the optimum on a specific optimized graph that reduces the search space accelerates the execution time of greedy solutions.

**4.2.1. The greedy optimization framework.** The basic idea behind the proposed framework is to reduce the problem to a minimum cost unsplittable multicommodity flow problem with multiple dependency component sources in specific directed flow network graph $G_p = (DC_p \cup A_p; E_p; u; c)$, and deals with a cost function $c$: $E \to R$ and capacity function u: $E \to R$

The first part of the construction of the network flow graph $G_p$ concerns the assignment of the input flows from multiple sources. For each collocated task $a^r_i \in A$ that generates intermediate data $\phi^m_{a^r_i}(t)$ in the same source datacenter $dc_i$, there is a virtual source datacenter node $dc_i(\phi^m_{a^r_i})_p$ in $DC_p$. For all generated intermediate data $\phi^m(t)$ from multiple collocated tasks belonging to the same dependency component $m \in M$, there is a virtual dependency source datacenter node $dc_i(\phi^m)_p$ representing those intermediate data dependencies for different tasks. For all generated intermediate data dependency components $\phi^m(t)$ hosted in a multiple source datacenter in $G$, there is a virtual dependency component node $dc(\phi^m)_p$ which corresponds to a virtual location of distributed source datacenter $dc_i(\phi^m)_p$ hosting intermediate data of dependency component $\phi^m(t)$. The $dc_i(\phi^m_{a^r_i})_p$, $dc_i(\phi^m)_p$ and $dc(\phi^m)_p$ are added in graph $G_p$.

In network flow graph $G_p$, a virtual source node $s_{source}$ is added and represents the source of all intermediate data dependencies $\sum\limits_{m \in M} \sum\limits_{a^r_i \in A} \phi^m_{a^r_i}(t)$ hosted in the different virtual source datacenter nodes $dc_i(\phi^m_{a^r_i})_p$. Source node $s_{source}$ is connected with a link $(s_{source}, dc_i(\phi^m_{a^r_i})_p)$ in $E_p$ to each $dc_i(\phi^m_{a^r_i})_p$. Also, from this latter to $dc_i(\phi^m)_p$ represented by link $(dc_i(\phi^m_{a^r_i}))_p, dc_i(\phi^m)_p)$, involving cost $c(s_{source}, dc_i(\phi^m_{a^r_i})_p) = 0$, as well as a link capacity demand that is assigned as the set of intermediate data dependencies $\phi^m_{a^r_i}(t)$ generated from each collocated task in the source datacenter at time slot $t$, i.e:

$$u(s_{source}, dc_i(\phi^m_{a^r_i})_p) = u(dc_i(\phi^m_{a^r_i})_p, dc_i(\phi^m)_p) = |\phi^m_{a^r_i}(t)|. \tag{4.15}$$

A link $(dc_i(\phi^m)_p, dc(\phi^m)_p)$ is added to $G_p$ from each virtual dependency source datacenter node $dc_i(\phi^m)_p$ to the corresponding virtual dependency component node $dc(\phi^m)_p$ within the same dependency component

$m \in M$. The corresponding cost is $c(dc_i(\phi^m)_p, dc(\phi^m)_p) = 0$, and the capacity is an amount of dependency component from all source datacenters that temporarily store them, i.e:

$$u(dc_i(\phi^m)_p, dc(\phi^m)_p) = \sum_{a_i^r \in A} |\phi^m_{a_i^r}(t)| = |\phi^m_i(t)|. \tag{4.16}$$

The second part of the optimization framework deals with the identification of potential links for routing intermediate data dependencies to the destination datacenter. For each destination datacenter $dc_j$ in $G$, there is a virtual destination datacenter node $dc_{j_p}$ which hosts all intermediate data dependencies for one or multiple dependency components $\phi^m$. Each virtual destination datacenter node $dc_{j_p}$ is added to $G_p$. The obvious no-bottleneck assumption which was made throughout an unsplittable version of the greedy optimization framework is that a virtual destination datacenter node $dc_{j_p}$ in a network flow graph $G_p$ has enough capacity to satisfy all dependency components $\phi^m$ individually, but not necessarily all commodities. Thus, in graph $G$, destination datacenters that do not have available storage capacity to accommodate each dependency component are excluded from $G_p$. Hence, from each virtual dependency component node $dc(\phi^m)_p$ there is a link $(dc(\phi^m)_p, dc_{j_p})$ to each destination datacenter $dc_{j_p}$ that is added to graph $G_p$. All these links are connected to each virtual destination datacenter node $dc_{j_p}$ that satisfies the placement of an integer dependency component $\phi^m$. A positive cost $c(dc(\phi^m)_p, dc_{j_p})$ is assigned along a link $(dc(\phi^m)_p, dc_{j_p})$ from the virtual dependency component to the destination datacenter node. The corresponding total storage cost represents the sum of the data transfer cost $c_{w_\phi}(dc(\phi^m)_p, dc_{j_p})$ and the storage cost $c_{s_j}(dc(\phi^m)_p, dc_{j_p})$ to host one unit of intermediate data dependency $\phi^m_{a_i^r}$, i.e:

$$c(dc(\phi^m)_p, dc_{j_p}) = c_{w_\phi}(dc(\phi^m)_p, dc_{j_p}) + c_{s_j}(dc(\phi^m)_p, dc_{j_p}). \tag{4.17}$$

In addition to the cost of a virtual link $(dc(\phi^m)_p, dc_{j_p})$, a capacity $u(dc(\phi^m)_p, dc_{j_p})$ is assigned, which is the amount of intermediate data $\phi^m_{a_i^r}(t)$ that can be routed along a virtual link with an available bandwidth capacity upon routing integer dependency component $\phi^m$. The capacity of the bandwidth is shared between each routing unit of a dependency component at time slot $t$. Since, the storage capacity constraint is raised when a link $(dc(\phi^m)_p, dc_{j_p})$ is created in graph $G_p$, the routing of the intermediate data dependency component $\phi^m(t)$ considers only the available amount of a data bandwidth $c_t(dc(\phi^m)_p, dc_{j_p})$ on each corresponding link $(dc(\phi^m)_p, dc_{j_p})$ to different virtual destination datacenters $dc_{j_p}$ at time slot $t$ , i.e:

$$u(dc(\phi^m)_p, dc_{j_p}) = \frac{w^{avail}_{dc(\phi^m)_p, j}(t)}{w_\phi \cdot |\phi^m_{a_i^r}(t)|}. \tag{4.18}$$

A virtual destination node $s_{sink}$ is finally added to a flow graph $G_p$ from each virtual destination datacenter node $dc_{j_p}$. A virtual link $(dc_{j_p}, s_{sink})$ is added between them. A zero cost is assigned to each virtual movement link $(dc_{j_p}, s_{sink})$. A capacity $u(dc_{j_p}, s_{sink})$ for each link $(dc_{j_p}, s_{sink})$ is the available amount of storage space in each one upon storing an integer dependency component $\phi^m$ at time slot $t$, i.e:

$$u(dc_{j_p}, s_{sink}) = s^{avail}_{dc_{j_p}}(t) - |\phi^m(t)| \tag{4.19}$$

Figure 4.1 shows the representation of the generated directed flow graph $G_p = (DC_p \cup A_p; E_p; u; c)$

**4.2.2. Greedy heuristic algorithm.** A greedy heuristic algorithm has been developed for the minimum cost inter-job intermediate data dependency placement problem through the reduction to the minimum cost of unsplittable multicommodity flow with multiple dependency component sources in flow graph $G_p$.

Let $S_{dc_j,min}$ be the minimum storage capacity of a destination datacenter $dc_{j_p}$ on a network flow graph $G_p$, and $\phi^m_{max}$ the largest dependency component generated from virtual source datacenter node $dc(\phi^m)_p$. As storage resources are scalable in a flow graph $G_p$ acting as a cloud environment, it is realistic to assume that $|\phi^m_{max}| \leq S_{dc_j,min}$ from the construction of the flow graph $G_p$. Since the splittable exact algorithm is a relaxation of the unsplittable heuristic algorithm, a feasible solution is assumed for the splittable exact algorithm which is fractional feasible flow $f_0$ that satisfies all demands of dependency component $\phi^m$. Since all dependency

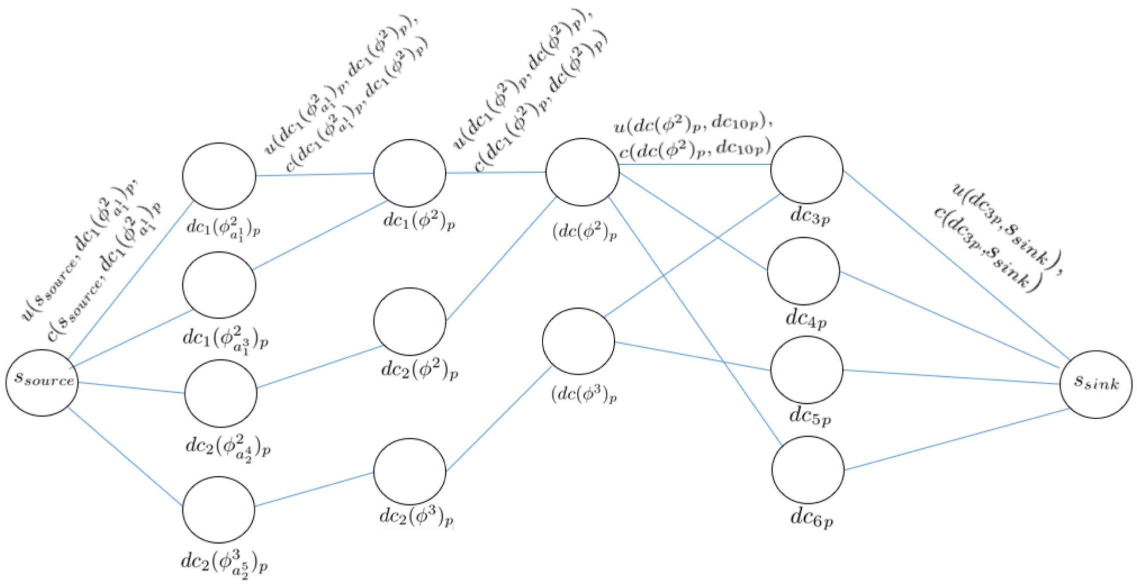

$u(s_{source}, dc_1(\phi^2_{a^1_1})_p$,
$c(s_{source}, dc_1(\phi^2_{a^1_1})_p$
$u(dc_1(\phi^2_{a^1_1})_p, dc_1(\phi^2)_p)$,
$c(dc_1(\phi^2_{a^1_1})_p, dc_1(\phi^2)_p)$
$u(dc_1(\phi^2)_p, dc(\phi^2)_p)$,
$c(dc_1(\phi^2)_p, dc(\phi^2)_p)$
$u(dc(\phi^2)_p, dc_{10p})$,
$c(dc(\phi^2)_p, dc_{10p})$
$u(dc_{3p}, s_{sink})$,
$c(dc_{3p}, s_{sink})$
$s_{source}$
$dc_1(\phi^2_{a^1_1})_p$
$dc_1(\phi^2_{a^3_1})_p$
$dc_2(\phi^2_{a^4_2})_p$
$dc_2(\phi^3_{a^5_2})_p$
$dc_1(\phi^2)_p$
$dc_2(\phi^2)_p$
$dc_2(\phi^3)_p$
$(dc(\phi^2)_p$
$(dc(\phi^3)_p$
$dc_{3p}$
$dc_{4p}$
$dc_{5p}$
$dc_{6p}$
$s_{sink}$

FIG. 4.1. *The generated directed flow graph $G_p = (DC_p \cup A_p; E_p; u; c)$.*

components are known a priori, so is their generation order. Hence, the greedy heuristic algorithm adopts an orderly greedy method and starts with the initial placement and works in steps. At the end of each step, it outputs a set of destination datacenters and transfers intermediate data dependencies to that destination datacenters, considering a minimum transfer and storage cost. As the greedy algorithm gives sequential placement solutions, there is no congestion problem on the different dependency components sharing links. Therefore, the greedy algorithm just takes care of the integer dependency component placement (bandwidth capacity is shared between the flows of a single dependency component at time slot $t$) to their destination. The greedy heuristic algorithm execution on a network flow graph $G_p$ is provided in the following steps:

*Step 1.* Let $f(\phi^m)$ be the dependency component flow for all dependency intermediate data-task flows $\sum\limits_{a^r_i \in A} f(\phi^m_{a^r_i})$ with the minimum total storage cost from $s_{source}$ to $s_{sink}$. Flows $f(\phi^m)$ route dependency component commodities $\phi^m_{a^r_i}$ from different virtual source datacenter nodes connected from source node $s_{source}$ to their destination datacenter nodes $dc_j(\phi^m)_p$, the latter being connected with destination node $s_{sink}$. A set of dependency component commodities $\sum_{m \in M} \phi^m$ are routed to $s_{sink}$ in graph $G_p$ according to the ascending order of their respective size as dependency component demands: $|\phi^1|$, $|\phi^2|$, ..., $|\phi^k|$, with $\phi^1 \geq \phi^2 \geq \phi^3 \geq ... \geq \phi^k$. Let $L_\phi = (\phi^1, \phi^2, ..., \phi^k)$ be the dependency component list.

*Step 2.* Start with the first dependency component by selecting it from list $L_\phi$. The algorithm scans each dependency component value $\phi^m_{a^r_i}(t)$ in $G_p$ to find the possible path which routes the selected dependency component flow $f(\phi^m)$ along each link $(dc(\phi^m)_p, dc_{j_p})$ in $G_p$ that satisfies the flow conservation in $G_p$ i.e from any nodes $dc_p$, $dc(0)_p \in DC_p \setminus \{s_{source}, s_{sink}\}$, there is $\sum_{dc(0)_p \in DC_p} f(dc_p, dc(0)_p) = \sum_{dc(0)_p \in DC_p} f(dc(0)_p, dc_p)$.

*Step 3.* For each solution of dependency component flow $f(\phi^m)$, find the shortest path noted $ShP_\phi$ from $s_{source}$ to $s_{sink}$ in $G_p$ according to the total minimum storage cost, i.e., $c(ShP_\phi)$ $=\min_{(dc(\phi^m)_p, dc_{j_p}) \in E_p} c(dc(\phi^m)_p,$ $dc_{j_p})$. Once the shortest path $ShP_\phi$ is found, set $f(ShP_\phi) = \phi^m$ and delete iteratively its flow value $f(\phi^m_{a^r_i})$. Define residual capacity $u_{res}(s_{source}, s_{sink})$ from $s_{source}$ to $s_{sink}$ in order to decrease the routed flows in graph $G_p$, i.e., $u_{res}(s_{source}, s_{sink}) = u(s_{source}, s_{sink})$ - $f(ShP_\phi)$. Delete the routed dependency component $\phi^m$ from list $L_\phi$ and repeat the sub-procedure of step 2 until all flow values $f(\phi^m_{a^r_i})$ are scanned.

*Step 4.* Repeat the sub-procedure of step 3 until $L_\phi \longleftarrow \varnothing$ and carry the largest flow values iteratively. Then, restore these shortest paths including the optimal cost and denote for each $ShP_\phi$ the pair $< \phi^m, dc_j >$

corresponding to graph $G$.

**4.2.3. Time complexity.** To build a network flow graph $G_p$ for the greedy framework optimization, two steps are needed. The first one consists in assigning each source datacenter $dc_j \in DC$ hosting intermediate data to its dependency component node $m \in M$, and the second one to each destination datacenter $dc_j \in DC$ which is capable of accommodating. The construction of $G_p$ takes $O(M + |DC|)$ for the first step and $O(M^2 + |DC|^2)$ for the second one. Finally, we analyze the time complexity of the greedy solution which considers mostly the shortest path computing step and the sorting of the list of dependency components. The worst complexity of the sorting computation has a fundamental requirement of $O(M^2)$. The shortest path computation step is more complex and requires computing the distance between all intermediate data dependency components and datacenter destinations. This leads to $O(M^2 \times |DC|^2)$ time complexity to consider all combinations or couples. In summary, the average computational complexity of the proposed greedy heuristic algorithm is $O(2M^2 + |DC|^2 \times (M^2 + 1) + M + |DC|)$ in the worst case.

**5. Performance evaluation.** This section gives an overview of the simulation, evaluation conditions and settings of the proposed algorithms. A dedicated simulation program has been developed to conduct the performance assessments of the heuristic and compare it with the exact algorithm, random and uniform strategies, named random heuristic and uniform heuristic respectively. The random heuristic strategy randomly selects a datacenter to host the intermediate data until its capacity is exhausted and then selects another one as in default Hadoop scheduler [30] (random capacities and random costs). The uniform heuristic strategy is based on the uniform storage capacity of the distributed datacenter upon intermediate data dependency placement decision (balanced capacities and variable costs). This data placement strategy excludes the storage requirements as in [27, 28, 29, 11]. Subsequently, the performance evaluation overall intends to present relevant comparisons between the solutions found by the greedy heuristic algorithm with the optimal ones found by the exact algorithm in terms of performance metrics like optimality, scalability and convergence time.

**5.1. Simulation environment.** The heuristic is evaluated through a C++ language implementation. The exact algorithm is implemented with IBM ILOG AMPL and solved optimally using CPLEX. The objective of a numerical evaluation is to quantify the amount of total storage cost saving (objective function) that can be expected when routing intermediate data dependencies through cloud storage infrastructures using the greedy heuristic and exact algorithms. The evaluation also reflects particularly the influence of the number of datacenters, the amount of the routed intermediate data and the dependency parameters on the performance metrics.

The assessment scenarios correspond to a cloud infrastructure consisting of 50 distributed datacenters including source and destination datacenters which are connected to each other randomly. We run the simulation program for 20 random tasks, each one including an amount of a random intermediate data generated per one hour time slot in random adjacency matrix-based DAG, each one having a size ranging from 10 GB to 100 GB [12], including their dependencies that are generated randomly as correlation links in DAG from input to output intermediate data. The latter are assigned randomly to the set of source datacenters in charge of temporarily storing them.

The intra-job dependency is described by a dependency parameter value $\alpha$ generated randomly from range [0, 1] and belonging to each intermediate data-task in the DAG. Value 1 corresponds to a splitting rate of an intermediate data file (a fraction of 1 GB splitting for each file from partial correlation), and the opposite case is represented by value 0. A dependency parameter value $\beta$ is given also that is generated randomly from range [1, 20] which represents the number of clusters randomly grouping intermediate data-tasks. For the inter-job dependency, we set the value of $\alpha$ to 0 from full correlation coupled with dependency parameter value $\beta$ (the case of inter-job dependency is intrinsically related to the intra-job dependency case when the $\alpha$ value of the latter converges to 0 and has the same dependency value $\beta$). On all the carried out experiments, the case when $\alpha = 1$ and $\beta = 20$ are excluded which means that the intermediate data are completely independent. The same dependency parameter value $\beta$ is assigned to both intra- and inter-job dependencies according to each experiment.

The storage space capacity is considered for the datacenters as randomly set from range [10 GB, 1000 GB] [12]. The transfer link capacity of one unit of intermediate data transmission between distributed datacenters

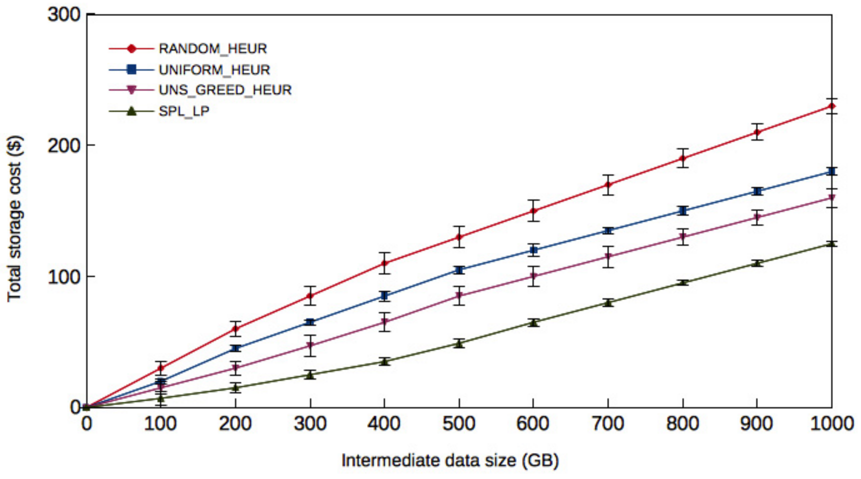


FIG. 5.1. *The total storage cost of algorithms exact, greedy, random and uniform heuristics while varying the intermediate data size when the number of datacenters is set to 50.*

are randomly drawn from range [1, 10] Gbps [7] with a random transfer cost (in $) ranging from 0 to 0.09. Both storage and transaction costs (in $) of one unit of intermediate data dependency are set within [0.02, 0.04] and [0, 0.09] respectively, in relation to the typical charges in Amazon S3 [3].

**5.2. Simulation results.** To present the performance of the proposed algorithms regarding their effectiveness against comparison strategies solutions, we study the optimality of the exact and greedy heuristic algorithms in terms of the total storage cost ratio, and the results of the scalability and the convergence are reported below.

**5.2.1. Impact of the amount of routed intermediate data on algorithms performance.** For the specific needs of the simulation, a variation of the amount of intermediate data must be placed from 100 to 1000 GB with an increment of 100 while the number of datacenters $DC$ is set to 50.

To continue to appropriately analyze the simulation, we reflect the concerns of dependency parameter values on the algorithms performance. In this case, each solution found from the execution algorithms is a mean of the results obtained by varying dependency parameters $\alpha$ and $\beta$ from range [0, 1] and [1, 20] respectively.

Figure 5.1 depicts the curves of total storage cost delivered by the proposed algorithms and the two other strategies. The figure shows that both greedy heuristic and exact algorithms outperform random heuristic and uniform heuristic strategies in terms of cost. The optimal result obtained by the exact solution reaches a cost of $125 when the amount of placed intermediate data achieves 1000 GB, and the greedy heuristic algorithm achieves a nearly optimal storage cost of $160, which is lower than the costs of the random heuristic and uniform heuristic algorithms (43% and 12% respectively). Clearly, the gap between greedy heuristic and uniform heuristic algorithms is very small since the uniform heuristic is independent of the capacity of the cloud infrastructure, so the cost within the datacenters contrast on the placement decision.

Figure 5.2 depicts the curves of the total storage costs of the algorithms by increasing the simulation time. In this instance, the obtained result of the total storage cost is the aggregation of the previously calculated costs during the same simulation (continuous placement). In addition, the simulation test is conducted for 48h in order to validate the need of the greedy heuristic algorithm and to estimate the probability to have good solutions. The lengthening of simulation at time slot 48 while the number of datacenters $DC$ is set to 50 makes the total storage cost of algorithms greedy heuristic, exact, random heuristic and uniform heuristic to $4900, $3300, $7000 and $5400 respectively. Typically, this means that the cost of greedy heuristic is 10% and 42% less than those of uniform heuristic and random heuristic algorithms, while the result of the exact algorithm

[3]https://aws.amazon.com/fr/s3/pricing/

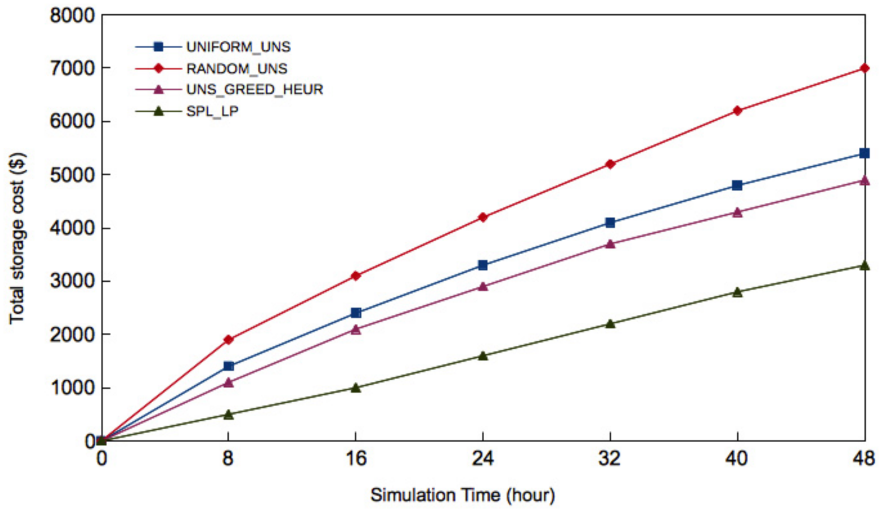


FIG. 5.2. *Total storage cost of algorithm exact, greedy, random and uniform heuristics when the simulation time is extended to 48h, while the number of datacenters is set to 50.*

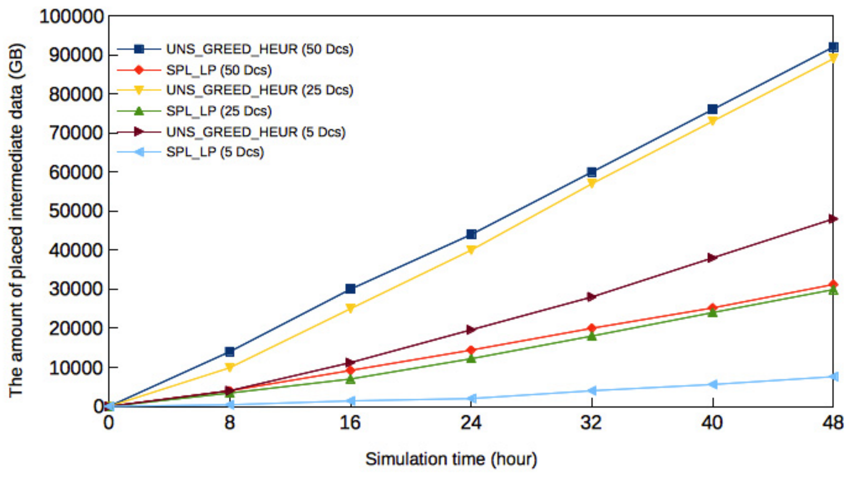


FIG. 5.3. *The amount of intermediate data accumulated per time slot for the proposed algorithms while the number of datacenter ranges from 5 to 50.*

as expected remains the best total storage cost. These results show that the uniform capacity constraint on the intermediate data growth directly affects their placement cost as in the case of the uniform heuristic algorithm. In the case of random heuristic algorithm, some datacenters offering the lowest cost are not involved unintentionally (random selection) or of the causes of inability to host data.

The aim of the performed simulation as depicted in Fig. 5.3 is to quantify the amount of intermediate data placed continuously by varying the number of datacenters from 5 to 50 according to capacities. The accumulated intermediate data placement increases with the increase of the number of datacenters for both algorithms, and begins to be stable when cloud infrastructure handles 25 datacenters. Moreover, the amount of intermediate data accumulated by greedy heuristic algorithm is very important over all variations of datacenter instances, more importantly, from 25 to 50, due to the abundance of the cloud infrastructure capacity, i.e. more intermediate data can be placed in the cloud infrastructure. The decline of intermediate data placement from 25 to 50 is due to the fact that the intermediate data routing is limited by data bandwidth $w_{i,j}^{avail}(t)$ and $w_{j,j'}^{avail}(t)$

since the bandwidth capacity is shared by all dependency components in the exact algorithm. In contrast, in the greedy heuristic algorithm, the data bandwidth capacity is shared by a single dependency component. However, in the exact algorithm, the amount of placed intermediate data is less in comparison with the greedy solution, because it expands the search space for the exact solution since it is based on the simplex method. Thus, the greedy heuristic algorithm responds well to large datacenter instances for which the exact algorithm has more difficulty to find solutions.

**5.2.2. Impact of dependency parameters on the algorithms performance.** This section studies the impact of the dependency parameters $\alpha$ and $\beta$ on the algorithms performance in terms of optimal cost. Since the behavior of the proposed execution algorithms regarding dependency type that are processed are different, a need of a variation of quantitative values is experienced for achieving a useful analysis and allowing optimal cost to the unsplittable placement solutions to be more efficiently identified. On the one hand, interval values are considered to align the types of dependency. On the other hand, values are considered when dependency types diverge. In the following, the assessment scenarios correspond to varying dependency parameters ($\alpha$, $\beta$) pair values. Then, simulation results correspond to pair ranges from ($\alpha$, $\beta$)= (0.1, 18), (0.3, 14), (0.5, 10), (0.7, 6), (0.9, 2). These results are reported on figures below keeping the value of $\alpha$ to 0 and with the same dependency values $\beta$ for the greedy heuristic algorithm while the number of datacenters is set to 50.

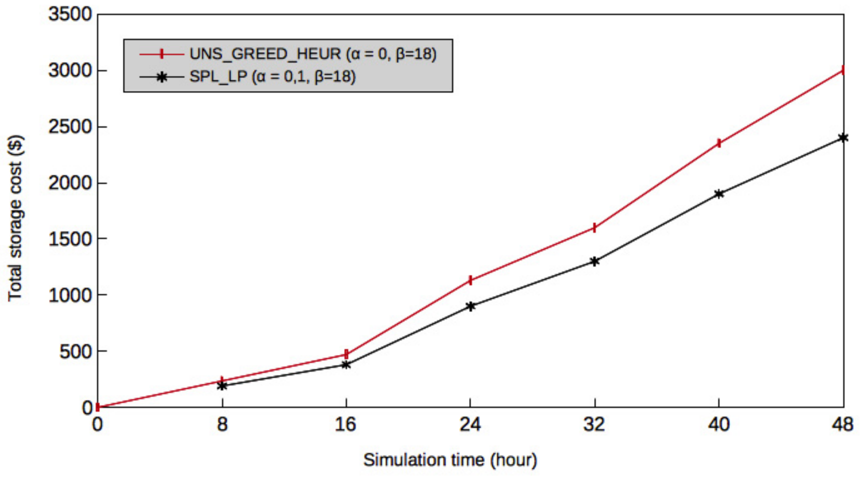


FIG. 5.4. *Greedy heuristic versus exact solutions for the total storage cost when $\alpha$ = 0.1 and $\beta$ = 18.*

Figure 5.4 depicts the best optimal cost achieved by the objective function for exact and heuristic solutions. The greedy heuristic algorithm performs very well close to the optimal one and achieves a cost of \$3000 at time slot 48 that is near to the optimal result of \$2400 for the exact algorithm. This is due to the fact that the behavior of the two algorithms have to deal with the aligned correlation $\alpha = 0.1$ (practically, no amount of intermediate data is splitted with exact, and 0 defaults to the heuristic) with $\beta = 18$. Therefore, this has a direct impact on the reduction of the transfer, storage and movement costs for both algorithms.

Figures 5.5 and 5.6 depict the second best-case results for the total storage cost which does not exceed \$2800, \$3000 for the exact algorithm, and \$4000, \$4500 for the greedy heuristic algorithm at time slot 48. Since, the amount of the dependency movements are marginal to half ($\alpha$ = (0.3, 0.5)) in the exact algorithm, the movement cost is reduced, which reflects the total storage cost. In addition, the heuristic algorithm processes less intermediate data dependencies (10 to 14 clusters). Therefore, it has more chance to find datacenters that have the capacity to allocate those clusters and at the same time offer a better cost.

Fig. 5.7 shows the case when the amount of dependency movements increase more than half ($\alpha = 0.7$) with the growth of the intermediate data dependency volume ($\beta = 6$). This gives a total storage cost of \$3800 and \$5800 respectively for exact and greedy heuristic algorithms. It can be seen that the total storage cost of the

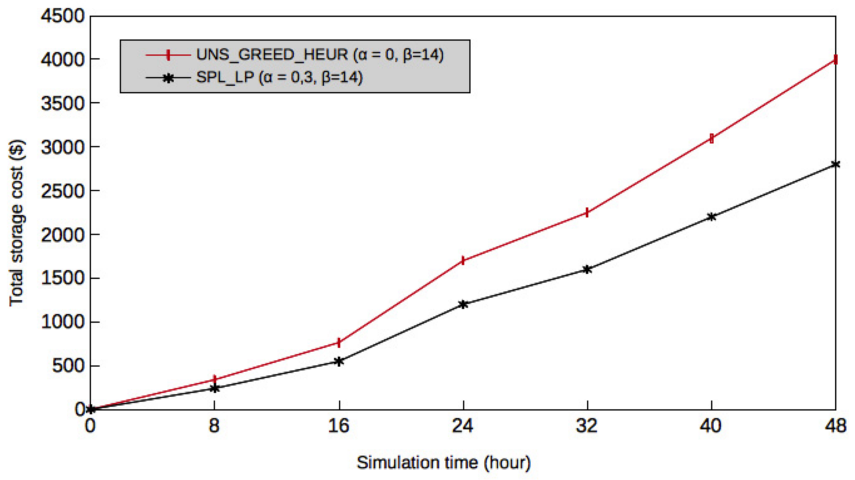


FIG. 5.5. *Greedy heuristic versus exact solutions for the total storage cost when* $\alpha$ *= 0.3 and* $\beta$ *= 14.*

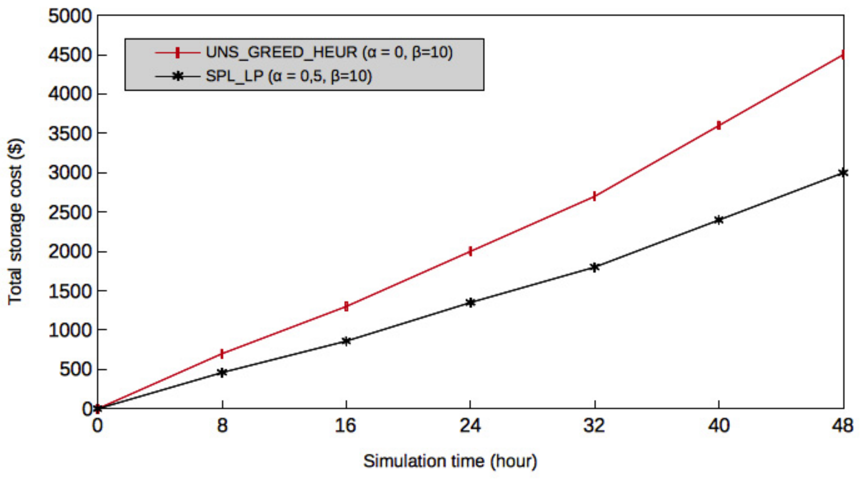


FIG. 5.6. *Greedy heuristic versus exact solutions for the total storage cost when* $\alpha$ *= 0.5 and* $\beta$ *= 10.*

two algorithms dependents on the amount of intermediate data dependencies. This validates our analysis for the results discussed above (Fig. 5.4, 5.5 and 5.6).

Fig. 5.8 shows the worst case when the highest total storage cost is found for both algorithms. The total storage cost reaches \$4200 and \$7200 for exact and greedy heuristic algorithms respectively at time slot 48 since the amount of intermediate data dependencies that transit between destination datacenters are significant ($\alpha =$ 0.9) for the exact algorithm (defined by variable $x^m_{j,j'}(t)$). In contrast, the heuristic processes more dependencies grouped onto two clusters. In addition, the same capacity values were considered for each solution reached with the different values of $\alpha$ and $\beta$. Therefore, it has less opportunity to find datacenters than the capacity to allocate those large clusters, and at the same time it offers a better cost. This influences considerably the search for the optimal result which is a real compromise for both algorithms.

The performance of the greedy heuristic algorithm as compared to the exact fractional optimal solutions in terms of total storage cost are represented as a cost ratio between the cost delivered by the heuristic algorithm $HEUR$ which is a greedy approximation approach for the unsplittable intermediate data dependency placement problem, and the fractional optimal solution $FRAC\_OPT$ provided by the simplex method to the problem of

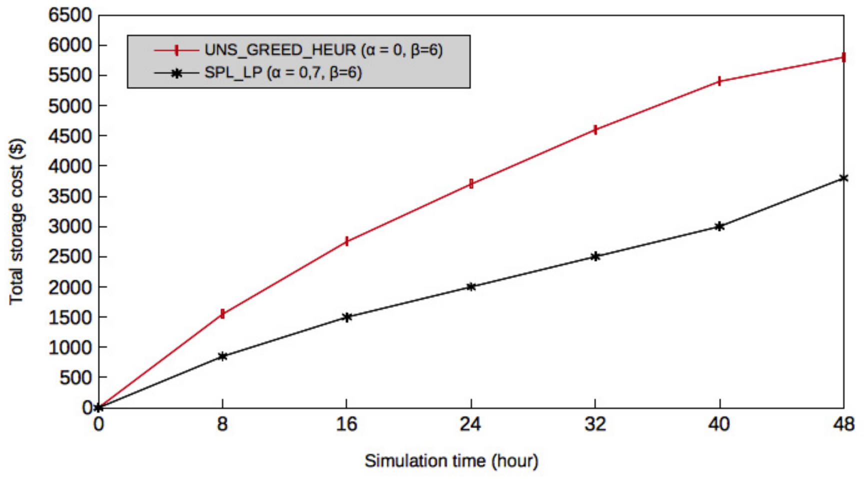


FIG. 5.7. *Greedy heuristic versus exact solutions for the total storage cost when $\alpha = 0.7$ and $\beta = 6$.*

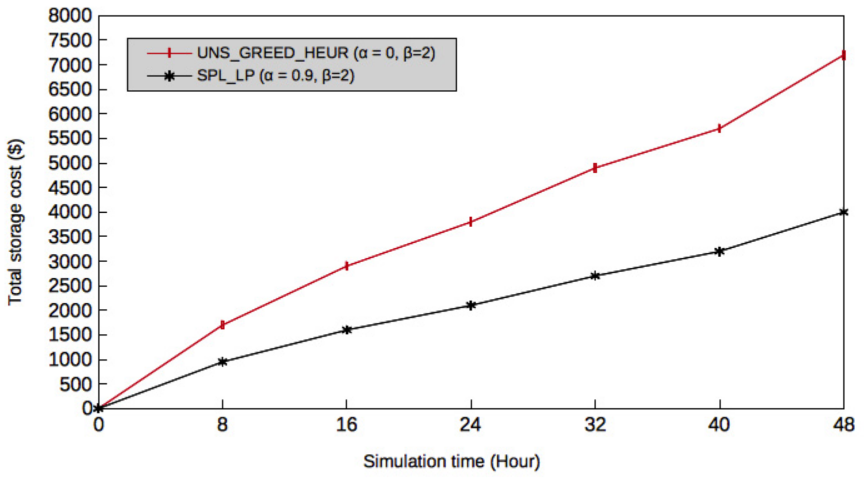


FIG. 5.8. *Greedy heuristic versus exact solutions for the total storage cost when $\alpha = 0.9$ and $\beta = 2$.*

splittable variant of the placement. The cost ratio of the heuristic $HEUR$ is $\epsilon = \frac{HEUR}{FRAC\_OPT}$. The cost ratio of the different curves above (Fig. 5.4 to 5.8 ) is reported in Table 5.2 when the number of datacenters varies from 5 to 50.

One can be see that the cost ratio of the greedy heuristic algorithm is no more than 1.85. Indeed, for simulated instances in the range from 5 to 50 datacenters when dependency parameter pairs $(\alpha, \beta) = \{(0.1, 18); (0.3, 14), (0.5, 10)\}$, the cost ratio of the greedy heuristic algorithm performs closer to the optimal solution and does not exceed 1.25, 1.42 and 1.52 respectively for each pair in fairly adverse conditions.

However, in the range from 5 to 50 datacenters when dependency parameter pairs $(\alpha, \beta) = (0.7, 6)$, the greedy heuristic encounters some difficulties in finding an optimal solution. Thus, the cost ratio of greedy algorithm reaches 1.81. Note that, in the greedy algorithm, the feasibility of the solution is assumed by scaling datacenter capacities. Thus, there is a solution to the problem when $\beta = 2$. The cost ratio of the greedy algorithm in this case reaches 1.85, which diverges considerably from the optimal solution, as a condition for finding any solutions that matches the optimal ones when $\alpha \leq 0.5$ and $\beta \geq 10$. Even as well, if dependency types are well identified, it is more difficult in these cases to find the best cost ratio meeting the dependency

Table 5.1
*Gaps between the greedy heuristic and the exact algorithms in terms of cost ratio.*

| $(\alpha;\beta)$ / $DC$ | (0-0.1, 18) | (0-0.3, 14) | (0-0.5, 10) | (0-0.7, 6) | (0-0.9, 2) |
|---|---|---|---|---|---|
| 5 | 1.255 | 1.410 | 1.510 | 1.819 | 1.858 |
| 10 | 1.253 | 1.422 | 1.509 | 1.820 | 1.859 |
| 15 | 1.249 | 1.413 | 1.511 | 1.809 | 1.857 |
| 20 | 1.248 | 1.401 | 1.512 | 1.809 | 1.856 |
| 25 | 1.245 | 1.402 | 1.509 | 1.808 | 1.856 |
| 30 | 1.239 | 1.402 | 1.513 | 1.810 | 1.851 |
| 35 | 1.241 | 1.411 | 1.520 | 1.810 | 1.851 |
| 40 | 1.241 | 1.411 | 1.520 | 1.819 | 1.850 |
| 45 | 1.250 | 1.420 | 1.519 | 1.819 | 1.849 |
| 50 | 1.249 | 1.419 | 1.519 | 1.819 | 1.850 |

restrictions. Indeed, each proposed algorithm responds differently to the dependency requirements as well.

A special cases are also considered which are not reported on Table 5.2, when dependency parameter pairs are set from a range of $(\alpha, \beta)$ = (0.1, 1), (0.1, 2), (0.9, 19), (0.9, 18). These parameter values are the most extreme and contradictory cases, in the sense that for dependency pairs (0.1, 1) and (0.1, 2), the exact algorithm finds a solution with an adjustment of time (beyond the days) but could not find an optimal solution, and for the latter cases (0.9, 19) and (0.9, 18), this does not reflect the correlation-type of intra-job dependency.

We conclude that the cost ratio of the greedy algorithm depends on the value of the dependency parameters and the amount of intermediate data that increase at each time slot. In the two cases, where the dependency parameters nearly correlate $(\alpha, \beta) = \{(0.1,\ 18);\ (0.3,\ 14),\ (0.5,\ 10)\}$, the cost ratio is more profitable. This means that the two proposed algorithms reacted well to these dependency value requirements. However, the cost ratio of the greedy algorithm that is reported in Table 5.2 increases as dependency parameters deviate $(\alpha, \beta) = \{(0.7,\ 6);\ (0.9,\ 2)\}$.

**5.2.3. Convergence time of the proposed algorithms.** To pursue the extensive experiments, we evaluate the effectiveness of the proposed algorithms and compare them in terms of scalability and convergence time from input parameters. For the comparison, we extend the simulation by varying the number of datacenters from 10 to 100 and by setting the amount of routed intermediate data from 100 GB to 1000 GB. Obviously, the values of the dependency parameters must also vary in order to better understand the behavior of the execution time of the proposed algorithms as regard to the dependencies. Thus, the value of dependency parameters is set as specified in Sec. 5.2.2. Algorithm running times are recorded as follows.

First, the execution time of the greedy algorithm solves the placement problem one to four orders of magnitude faster than the exact solution. However, the exact algorithm solves the NP-hard problem in exponential time for large instances since a part to solve the simplex-based LP method takes much time, particularly for $\alpha$ values between 0.1 and 0.5 because the intermediate data splitting parameters are less tolerated throughout their placement. Furthermore, the values of dependency parameters correlate with the continuous amount of intermediate data bounded by a discrete quantity. Not surprisingly, greedy heuristic is much easier to solve than the exact algorithm.

Indeed, Fig. 5.9 shows the best convergence time for each of the proposed algorithms.

The time needed to find an optimal solution when the amount of intermediate data to be hosted is 100 GB remains very satisfactory for datacenter sizes below 10, with less than 0.075 and 0.7 seconds for greedy heuristic and exact algorithms respectively. For datacenter sizes below 50, the convergence time remains fairly reasonable too, with less than 0.15 and 1.05 seconds for greedy heuristic and exact algorithms respectively. For the latter, it slightly increases when the number of datacenters is beyond 100 (about 5 seconds). In fact, the exact algorithm performance gradually degrades with input network topology and exponentially grows for wide range (not shown in Fig. 5.4). The following figures (Fig. 5.10 and 5.11) show these behaviors.

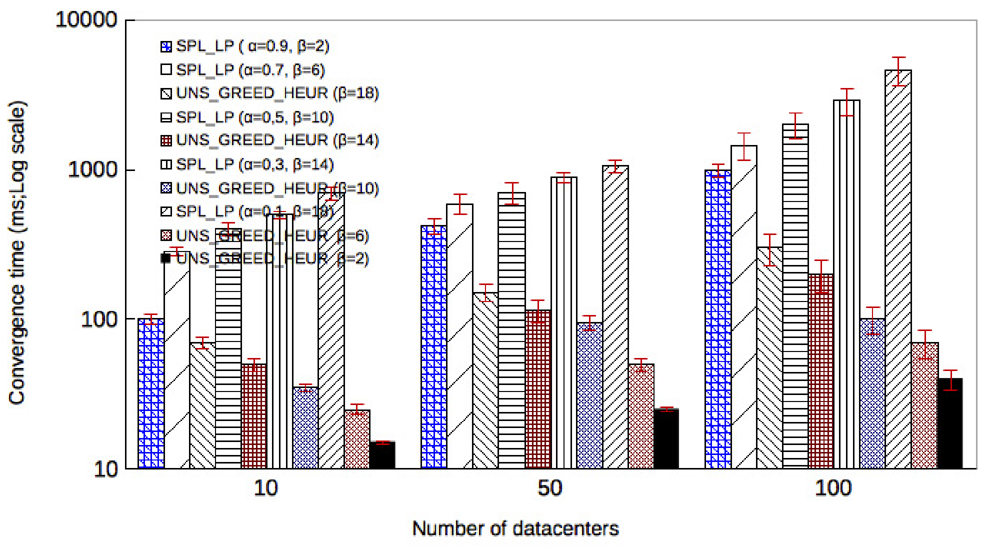


FIG. 5.9. *Time execution comparison between greedy heuristic and exact algorithms for different datacenter size when varying the amount of generated intermediate data ($|\Phi^M|$= 100 GB).*

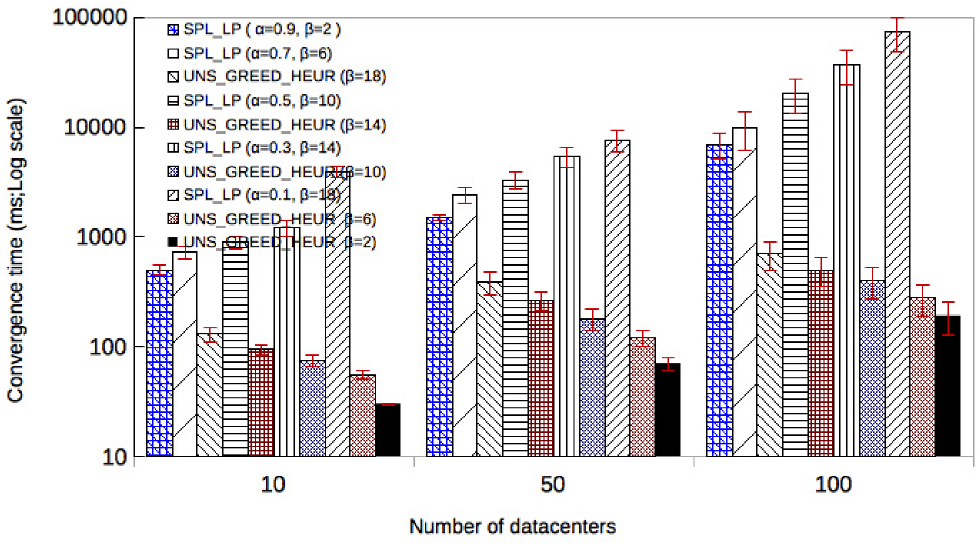


FIG. 5.10. *Time execution comparison between greedy heuristic and exact algorithms for different datacenter size when varying the amount of generated intermediate data ($|\Phi^M|$= 500 GB).*

Figures 5.10 and 5.11 show the worst cases for the exact algorithm. Then, the time needed for convergence grows mainly for an amount of placed intermediate data. These amount varies between 500 GB and 1000 GB for the simulated scenarios from 50 to 100 datacenters while the time running the greedy heuristic remains very fast to find solutions with a convergence time improvement ratio in range [$10^1$,$10^3$] as compared to the exact algorithm. Although dependency parameter values vary, the number of routing $\beta$ from 2 to 18 commodities, the heuristic algorithm scales better already for these large instances and it is more robust in ensuing scenarios and simulations. By contrast, the exact algorithm performance reacts poorly to dependency parameter variations.

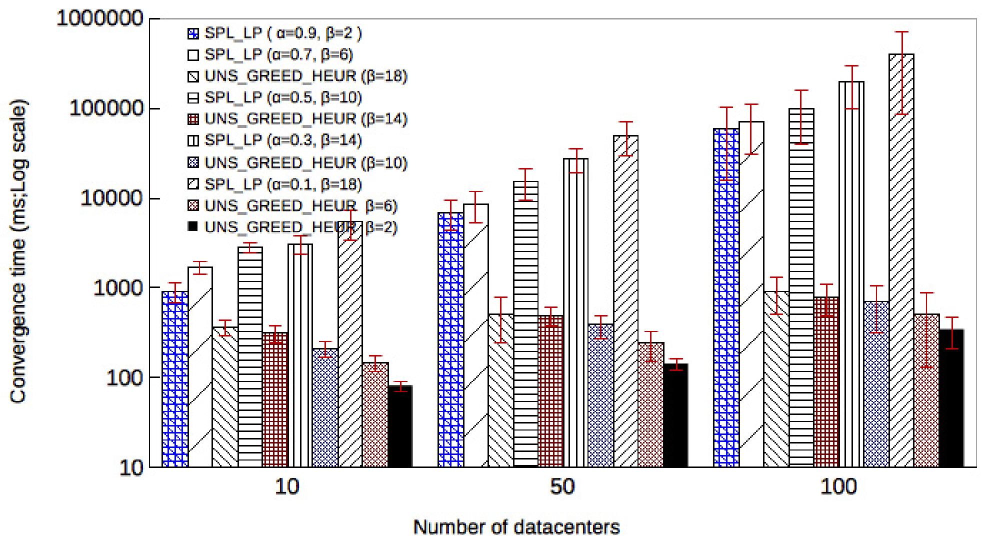


FIG. 5.11. *Time execution comparison between greedy heuristic and exact algorithms for different datacenter size when varying the amount of generated intermediate data ($|\Phi^M|$= 1000 GB).*

Particularly, this corresponds to values of $\alpha$ that vary from 0.1 to 0.5, where the exact algorithm exceeds a running time of one minute as shown in Fig. 5.10.

The convergence time increases with the number of datacenters as well as slightly less with the amount of intermediate data (see Fig. 5.11). As a matter of fact, the capacity of bandwidth is limited to 10 GB for data transfer, but with more transfer links through the growth of the number of datacenters.

The gap between the algorithms is in range [$10^2$,$10^4$] with an improvement factor in favor of the greedy heuristic.

As expected for the exact algorithm that was built upon the simplex method (which is based on the number of intermediate data to be fractionated and this is done for each iteration as the scale of the datacenter and links between them, meaning that the separation procedure is generally not polynomial) and even with the use of a set of dependency constraint values to limit the convex hull problem to find the optimal solution faster that goes beyond 500 GB for100 datacenters, the convergence time remains widely slow at about 7 minutes.

In conclusion, the execution time of the proposed algorithms depends mostly on the cloud infrastructure topology, and slightly less on the amount of intermediate data dependencies for the exact algorithm. Besides, the change in dependency parameter values influences largely the exact algorithm performance and much less the greedy heuristic. This validates the motivation for the use of a heuristic approach to find solutions faster even if there are bound to be approximated (as reported in Table 5.2).

**6. Conclusion.** In this work, we have studied the problem of intermediate data dependency placement. We presented and evaluated an exact model, as well as a greedy heuristic. Our proposed solutions try to save the total storage cost for an economical and efficient task workflow processing across distributed datacenters. The presented solutions take into consideration both intra- and inter-job dependencies including fractional and atomic demands respectively. The exact algorithm based on the LP model introduces new locality constraints on the optimal placement of intermediate data dependencies. The latter can be fractionated and routed in the same physical datacenter or assigned to different destinations. In addition, the exact model is generic enough to optimize the data placement for task workflow processing in cloud environment thanks to the use of a generic objective function that combines multiple criteria such as data bandwidth and storage capability, as well as data movement optimization with an approved scalability for medium instances. Despite our formulation for the LP model, the number of datacenters and the variation of intermediate data dependency parameters makes it only solvable for medium instances. In order to ensure the placement of inter-job dependency-based intermediate data for larger instances, we developed a heuristic based on a greedy optimization framework, this, solves the problem in very fast time, making an assumption of an optimal fractional solution. The evaluation tests show that the greedy heuristic algorithm performs closer to the exact formulation solution (in the case of converged correlations), and boots higher performance as compare to other state of the art strategies. We evaluated also the convergence time of the proposed algorithms. It is improved by several orders of magnitude for the greedy heuristic algorithm compared to the exact algorithm, while making possible to solve large cloud infrastructures in a reasonable time.

## REFERENCES

[1] P. Kolman, *A note on the greedy algorithm for the unsplittable flow problem*, in Information Processing Letters Elsevier, vol. 88, no 3, (2003), pp. 101–105.

[2] P. Krysta, *Greedy approximation via duality for packing, combinatorial auctions and routing*, in International Symposium on Mathematical Foundations of Computer Science (Springer 2005), pp. 615–627.

[3] Belaidouni, Meriema and Ben-Ameur, Walid, *On the minimum cost multiple-source unsplittable flow problem*, RAIRO-Operations Research , 41(3), (2007), pp. 253-273

[4] A. Chakrabarti, C. Chekuri, A. Gupta, and A. Kumar, *Approximation algorithms for the unsplittable flow problem*, Algorithmica, (2007), vol. 47, no 1, pp. 53-78.

[5] H. Pirsiavash, D. Ramanan, and C. C. Fowlkes, *Globally-optimal greedy algorithms for tracking a variable number of objects*, Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on. IEEE, (2011). pp. 1201-1208.

[6] Q. Xia, W. Liang, and Z. Xu, *The operational cost minimization in distributed clouds via community-aware user data placements of social networks*, Computer Networks (2017), vol. 112, pp. 263-278.

[7] Q. Xia, Z. Xu, W. Liang, and A. Y. Zomaya, *Collaboration-and fairness-aware big data management in distributed clouds*, in IEEE Transactions on Parallel and Distributed Systems (2016), 27(7), pp. 1941-1953.

[8] P. Agrawal, D. Kifer, and C. Olston, *Scheduling shared scans of large data files*, Proceedings of the VLDB Endowment (2008), vol. 1, no 1, p. 958-969.

[9] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas, *MRShare: sharing across multiple queries in MapReduce*, Proceedings of the VLDB Endowment 3.1-2 (2010), pp. 494-505.

[10] Q. Sun, M. Romanus, T. Jin, H. Yu, P. T. Bremer, S. Petruzza, ... and M. Parashar, *In-staging data placement for asynchronous coupling of task-based scientific workflows*, in Extreme Scale Programming Models and Middlewar (ESPM2), International Workshop on (2016), pp. 2-9. IEEE.

[11] Q. Zhao, C. Xiong, X. Zhao, C. Yu, and J. Xiao, *A data placement strategy for data-intensive scientific workflows in cloud*. In Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on (2015), pp. 928-934. IEEE.

[12] M. Ebrahimi, A. Mohan, A. Kashlev, S. Lu, *BDAP: a big data placement strategy for cloud-Based scientific workflows*. In Big Data Computing Service and Applications (BigDataService), IEEE First International Conference on (2015), pp. 105-114. IEEE

[13] R. Vilaa, R. Oliveira, J. Pereira, *A correlation-aware data placement strategy for key-value stores*, in IFIP International Conference on Distributed Applications and Interoperable Systems (2011), pp. 214-227. Springer Berlin Heidelberg.

[14] Q. Zhao, C. Xiong, P. Wang, *Heuristic Data Placement for Data-Intensive Applications in Heterogeneous Cloud*, in Journal of Electrical and Computer Engineering, 2016.

[15] Asano, Y., *Experimental Evaluation of Approximation Algorithms for the Minimum Cost Multiple-source Unsplittable Flow Problem*, In ICALP Satellite Workshops (2000), pp. 111-122

[16] D. Warneke, O. Kao, *Nephele: efficient parallel data processing in the cloud*, in Proceedings of the 2nd workshop on many-task computing on grids and supercomputers. ACM (2009), pp. 8.

[17] X. Liu, A. Datta, *Towards intelligent data placement for scientific workflows in collaborative cloud environment*, in Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on. IEEE (2011), pp. 1052-1061.

[18] Z. Wu, X. Liu, Z. Ni, D. Yuan, and Y. Yang, *A market-oriented hierarchical scheduling strategy in cloud workflow systems*, The Journal of Supercomputing (2013), vol. 63, no 1, pp. 256-293.

[19] X. Liu, Z. Ni, Z. Wu, D. Yuan, J. Chen, and Y. Yang, *A novel general framework for automatic and cost-effective handling of recoverable temporal violations in scientific workflow systems*, in Journal of Systems and Software (2011), vol. 84, no 3, pp. 492-509.

[20] D. Wang, and J. Liu, *Optimizing big data processing performance in the public cloud: opportunities and approaches*, IEEE network (2015), vol. 29, no 5, pp. 31-35.

[21] J. Jin, J. Luo, A. Song, F. Dong, and R. Xiong, *Bar: An efficient data locality driven task scheduling algorithm for cloud computing*, in Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. IEEE Computer Society (2011), pp. 295-304.

[22] L. Kaufman, and P. J. Rousseeuw, *Finding groups in data: an introduction to cluster analysis*, vol. 344, (2009) John Wiley & Sons.

[23] B. Rajasekar, and S. K. Manigandan, *An Efficient Resource Allocation Strategies in Cloud Computing*, in International Journal of Innovative Research in Computer and Communication Engineering (2015), vol. 3, no 2, pp. 1239-1244.

[24] V. P. Anuradha, and D. Sumathi , *A survey on resource allocation strategies in cloud computing*, in Information Communication and Embedded Systems (ICICES), 2014 International Conference on (2014), IEEE, 2014. pp. 1-7.

[25] D. Ardagna, G Casale, M. Ciavotta, J. F. Prez, and W. Wang, *Quality-of-service in cloud computing: modeling techniques and their applications*, Journal of Internet Services and Applications (2014), vol. 5, no 1, pp. 11.

[26] Z. Xu, and W. Liang, *Operational cost minimization of distributed data centers through the provision of fair request rate allocations while meeting different user SLAs*, Computer Networks (2015), vol. 83, pp. 59-75.

[27] S. Agarwala, D. Jadav, and L. A. Bathen, *iCostale: adaptive cost optimization for storage clouds*, in Cloud Computing (CLOUD), 2011 IEEE International Conference on. IEEE (2011). pp. 436-443.

[28] D. Yuan, Y. Yang, X. Liu, G. Zhang, and J. Chen, *A data dependency based strategy for intermediate data storage in scientific cloud workflow systems*, in Concurrency and Computation: Practice and Experience (2012), vol. 24, no 9, pp. 956-976.

[29] X. Ren, P. London, J. Ziani, and A. Wierman, *Joint data purchasing and data placement in a geo-distributed data market*, ACM SIGMETRICS Performance Evaluation Review. ACM (2016). pp. 383-384.

[30] T. da Silva Morais, *Joint data purchasing and data placement in a geo-distributed data marketSurvey on frameworks for distributed computing: Hadoop, Spark and Storm*, in Proceedings of the 10th Doctoral Symposium in Informatics Engineering-DSIE (2015). Vol. 15.

[31] A. Dalvandi, M. Gurusamy, and K. C. Chua, *Application scheduling, placement, and routing for power efficiency in cloud data centers*, in IEEE Transactions on Parallel and Distributed Systems (2017). 28, no. 4 pp. 947-960.

[32] A. Gawanmeh, S. Parvin and A. Alwadi, *A Genetic Algorithmic method for scheduling optimization in cloud computing services*, in Arabian Journal for Science and Engineering (2017), pp. 1-10.

[33] B. A. Rao and L. V. Vahini, *Efficient scheduling of scientific workflows using multiple site awareness big data management in cloud*, in International Journal of Scientific Research in Computer Science, Engineering and Information Technology (2018). Volume 3 — Issue 1 — ISSN : 2456-3307.

[34] M. H. Ferdausa, M. Murshedb, N Rodrigo Calheirosc and R. Buyyac, *An algorithm for network and data-aware placement of multi-tier applications in cloud data centers*, in Journal of Network and Computer Applications (2017), vol. 98, pp. 65-83.

[35] A. M Manasrah and H. Ba Ali, *Workflow Scheduling Using Hybrid GA-PSO Algorithm in Cloud Computing*, in Wireless Communications and Mobile Computing (2018), vol. 2018.

[36] N. Anwar and H. Deng, *A Hybrid Metaheuristic for Multi-Objective Scientific Workflow Scheduling in a Cloud Environment*, in Applied Sciences (2018), vol. 8, no 4, pp. 538.

[37] H. Arabnejad and J. G. Barbosa, *List scheduling algorithm for heterogeneous systems by an optimistic cost table*, in IEEE Transactions on Parallel and Distributed Systems (2014), vol. 25, no 3, pp. 682-694.

[38] M. Abdullahi and M. A. Ngadi, *Symbiotic Organism Search optimization based task scheduling in cloud computing environment*, in Future Generation Computer Systems (2016), vol. 56, p. 640-650.

*Edited by:* Sasko Ristov
*Received:* Feb 14, 2018
*Accepted:* Aug 24, 2018

DOI 10.12694/scpe.v19i3.1375
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 19, Number 3, pp. 245–257. http://www.scpe.org © 2018 SCPE

# SIGNIFICANCE OF HIERARCHICAL AND MARKOV CLUSTERING IN GROUPING-AWARE DATA PLACEMENT FOR DATA INTENSIVE APPLICATIONS WITH INTEREST LOCALITY [*]

SHANMUGASUNDARAM VENGADESWARAN[†] AND SADHU RAMAKRISHNAN BALASUNDARAM[‡]

**Abstract.** During the execution of complex queries, the execution time increases exponentially, resulting in more waiting time for the user, which may sometimes extend to hours or even days in the worst cases. By virtue of their parallel and distributed computing capability, Hadoop and Spark are considered as an ideal solution for such complex query processing. Even though they are considered as an efficient solution for complex query processing, they have their own limitations when the data to be processed exhibits interest locality (i.e.) the data required for any query execution follows grouping behaviour wherein only a part of the BigData is accessed frequently. Since the data placement provided by these frameworks does not consider interest locality, it is possible that the dependent blocks required for execution will be concentrated within fewer computing nodes, resulting in several lacunas such as underutilisation of resources, and increased query execution time. Hence this paper proposes an Optimal Data Placement (ODP) Strategy based on grouping semantics. The significance of different clustering techniques viz. k-means, Hierarchical Agglomerative Clustering (HAC) and Markov Clustering (MCL), in grouping-aware data placement for data intensive applications with interest locality has been examined in this paper. Initially, the user access pattern is identified by dynamically analysing the history log. Then, clustering techniques (k-means, HAC and MCL) are separately applied over the access pattern to obtain independent clusters. These clusters are interpreted and validated to extract the Optimal Data Groupings (ODG). Finally, the proposed strategy reorganises the default data layouts in Hadoop Distributed File System (HDFS) based on ODG to achieve maximum parallel execution per group subjective to Load Balancer and Rack Awareness. Our proposed strategy is tested in 10 node cluster placed in a multi-rack with Hadoop installed in every node deployed in the cloud platform. The proposed strategy reduces the query execution time, significantly improves the data locality and CPU utilisation, and is proved to be more efficient for massive dataset processing in a heterogeneous distributed environment. In addition, MCL shows a marginal improved performance over HAC and k-means for queries exhibiting interest localities.

**Key words:** BigData, Storage and Compute Infrastructure, Interest Locality, Data Placement, Hierarchical Agglomerative Clustering, Markov Clustering, Heterogeneous Hadoop Cluster, Cloud

**AMS subject classifications.** 68-M14, 68-M20, 68-W10, 68-W15

## 1. Introduction.

In the current data era, massive volumes of data are being generated every second in a variety of domains such as geosciences, the social web, finance, e-commerce, healthcare, climate modelling, physics, astronomy, government sectors etc. BigData is the term applied to such large volumes of datasets whose size is beyond the ability of the commonly used software tools to capture, manage, and process within a tolerable elapsed time [1, 2]. By virtue of their parallel and distributed computing capability, Hadoop and Spark [3, 4, 5] are considered ideal solutions to analyse and gain insights from BigData and are well-recognised as de facto BigData processing platforms in the cloud; they have been adopted extensively and are currently used widely in many application domains. Apache Hadoop [1, 6] facilitates the distributed processing of large datasets across clusters of commodity hardware using simple programming models. Here, local storage and computation are achieved through the two major components namely Hadoop Distributed File System *(HDFS)* and MapReduce *(MR)*. The fundamental concept of HDFS [7] and MR [8] is to distribute data among nodes and process them in parallel. HDFS is a distributed file system capable of storing large files across multiple nodes. It follows a master-slave architecture, consisting of one NameNode and multiple DataNodes. When a file is dumped into HDFS, it is broken into fixed-size blocks and stored on multiple DataNodes. The DataNodes periodically report the blocks stored in them to the NameNode, thereby updating the metadata. When a query is executed from a client, it will reach out to the NameNode to retrieve the metadata, and then reach out to the DataNodes to retrieve the data blocks.

The major challenge in processing BigData in HDFS is faced during query execution, since the time taken

[*]The research work reported in this paper is supported by Department of Electronics & Information Technology (DeitY), a division of Ministry of Communications and IT, Government of India.

[†] Department of Computer Applications, National Institute of Technology, Tiruchirappalli 620015, India. (meetvengadesh@gmail.com). Thanks to Microsoft Azure for sponsoring the cloud infrastructure required to carry out the experiments under the Microsoft Azure for Research Award.

[‡] Department of Computer Applications, National Institute of Technology, Tiruchirappalli 620015, India. (blsundar@nitt.edu).

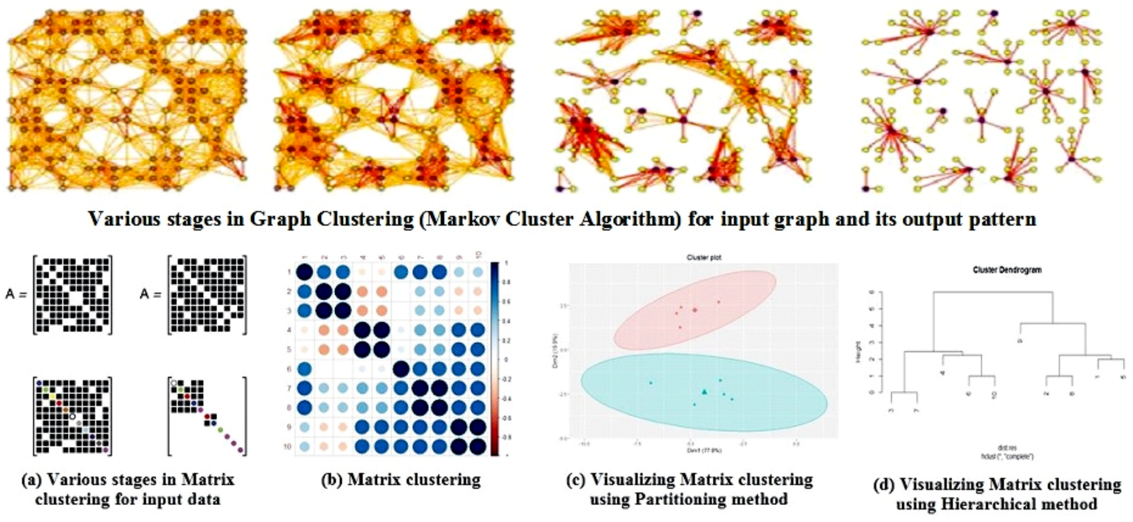

Various stages in Graph Clustering (Markov Cluster Algorithm) for input graph and its output pattern
A =
A =
Cluster plot
Cluster Dendrogram
Height
(a) Various stages in Matrix clustering for input data
(b) Matrix clustering
(c) Visualizing Matrix clustering using Partitioning method
(d) Visualizing Matrix clustering using Hierarchical method

FIG. 1.1. *Various stages of clustered graph by applying clustering algorithm*

to execute a query and return the results increases exponentially as the amount of data increases, leading to a long waiting time for the user [9]. Sometimes, the waiting times could range from minutes, to hours, to days in the worst cases. During query execution, it is commonly observed that most of the data-intensive applications exhibit interest locality [10]. It may be different for different domain analysts based on geographical location, time, person etc. (i.e. domain scientists are only interested in a subset of the whole dataset, and are likely to access one subset more frequently than others. For example, in the bioinformatics domain, X and Y chromosomes are related to the offsprings gender. Both chromosomes are often analysed together in generic research rather than all 24 human chromosomes). Mostly, for query execution, only a part of such BigData sets is utilised. The detailed analysis of various query executions clearly shows a significant similarity in the data required to execute the query during a set of time intervals. These data blocks will then have the highest frequency of being accessed as a group during executions. Data grouping is then formally defined as grouping semantics to represent the possibility of two or more data being accessed as a group. In Hadoops Default Data Placement Strategy *(HDDPS)*, the data blocks are placed randomly across the cluster of nodes without considering the nature of queries likely to be executed in the system. Due to such non- consideration of interest locality, it is possible for the required data blocks to be concentrated within fewer computing nodes, which, in turn, results in an increase in query execution time, query latency etc. In this paper, an Optimal Data Placement *(ODP)* Strategy based on grouping semantics is proposed. The natural behavioural groupings in the dataset are identified by applying clustering algorithms and the data-placement decision is taken based on the observed grouping behaviour. Clustering is the task of grouping a set of objects in such a way that objects in the same group are more similar to each other than to those in other groups [11, 12].

In this paper, we experiment the significance of different clustering techniques viz. k-means [13], Hierarchical Agglomerative Clustering *(HAC)* [14] and Markov Clustering *(MCL)* [15] in grouping-aware data placement for data-intensive applications with interest locality. It has been proved in a heterogeneous distributed environment for the e-commerce dataset [16, 17]. The results show that queries are solved by the domain analyst at the earliest possible time to enable quick decisions, as well as deriving maximum utilisation of resources. Fig.1.1 shows the various stages in MCL for an input dataset. Fig.1.1(a) shows the various stages in HAC for an input dataset. The clustered matrix obtained by applying the HAC is shown in Fig.1.1(b). The visualisation of matrix clustering by k-means and HAC methods is depicted in Fig.1.1(c) and Fig.1.1(d) respectively.

**2. Related Works.** Several works were carried out in data placement for massive datasets in some specific ways to support high-performance data accesses. ODP strategy, which focuses on reducing energy consumption and resource utilisation, was proposed by Ashwin Kumar et al. (2013) [18] and Wu et al. (2017) [19]. They proposed ODP by locating the related data blocks together. However, the major drawback in this area of focus is the increased query execution time. Here, the focus is on reducing the utilisation of resources, but this cannot be considered as a viable solution, since the real objective of processing BigData is achieving timely results.

Some significant works have also been carried out on data placement to achieve a reduced query execution time. Lee et al. (2014)(2014) [20] proposed an ODP by taking into account the computing capacity of a data node so that faster computing nodes are allocated with more data. This reduces the overall query execution time and provides high throughput of data. However there is no mechanism to ensure that the data blocks which are required for execution are proportionately present in those nodes since the grouping semantics of the dataset is not taken into account. Xiong et.al (2015) [21] proposes a heterogeneity aware data placement algorithm which initially groups the Data-Nodes as several virtual storage tiers (VST). The data blocks are placed across the nodes in each VST circuitously according to the hotness of data. This strategy shows an improved MR performance with reduced disk space utilization. However the individual requirements of data blocks are only assessed for measuring the hotness. But the relative dependency among various blocks for the different task executionsis not considered, which may lead to concentration of popular data within a node leading to reduced parallel execution.

ODP to reduce query execution time based on grouping semantics by applying clustering algorithms is also discussed by few researchers. Wang et al. (2014) [10] and Wu, w et al. (2016) [22] proposed an ODP algorithm based on grouping semantics, which reduces the query execution time and improves the data locality. It improves the parallel execution of datasets with interest locality. This ODP strategy use the Bond energy algorithm (BEA) to cluster the dependency matrix, which leads to a higher execution time. This is due to the time complexity in BEA for finding the permutations of all rows. In addition, for further execution of any new task, all iterations of BEA must be repeated.

Liao et al. (2016) [23] focus on optimising resource utilisation using a novel scheduling algorithm. Similarly, Shivaswamy et al.(2017) [24] suggest scheduling the work flow of jobs during concurrent executions for optimal resource utilisation. However, in both cases, the existence of a general behaviour pattern among the tasks executed during a period of time is not considered. Hence, these queries with interest locality require further consideration. Some studies elucidate that some significant clustering techniques [13, 14, 15] are available that can be applied to find the natural groupings in a dataset with reduced computations without compromising the clustering performance. We harness these clustering approaches in large-scale data management to achieve improved performance in terms of reduced execution time, through ODP, especially when data-intensive applications exhibit interest locality.

**3. CORE-Optimal Data Placement Strategy.** An ODP strategy based on grouping semantics is proposed in this paper. The entire workflow diagram is shown in Fig. 3.1. The different steps involved in the proposed strategy are detailed below.

**Step 1:** ***Analysing User History Log*** The meta-information and user history log will be the input for this step. Analysing the characteristics of the cluster from the user history log for various workloads is the key for making an optimal placement decisions. These log files are voluminous and varied (semi-structured). All MapReduce applications executed in the cluster save the task execution details as a log file, which consists of two files *(i) the Job Configuration file* and *(ii) the Job Status file* - for each job executed in the machine.

**Step 2:** ***Tracing Network Topology*** NameNode contains meta-data from which the network topology is constructed to identify the different DataNodes present in the cluster and the data blocks present in each DataNode.

**Step 3:** ***Building Task Frequency Table*** Using these logs as input, the task frequency table is constructed, which contains different tasks, the frequency of each task, and the blocks required for each task.

**Step 4:** ***Constructing Task Execution Graph*** The computations of parallel processing can be solved efficiently, only if the task executions and the blocks required are depicted as a graph. The task execution graph shown in Fig.3.2 is obtained by analysing the task frequency table using the iGraph network analysis tool [25]. The task execution graph is an unordered pair $GTex = (B, T)$, where $B$ represents a set of vertices as blocks and $T$ represents a set of edges as tasks executed. $GTex$ is undirected and may hold parallel edges since some sets of blocks ($B' \subseteq B$) may be required for different task executions $Ti$.

**Step 4a:** ***Clustered Task Execution Graph (CGTex)*** The task execution graph ($GTex$) is then

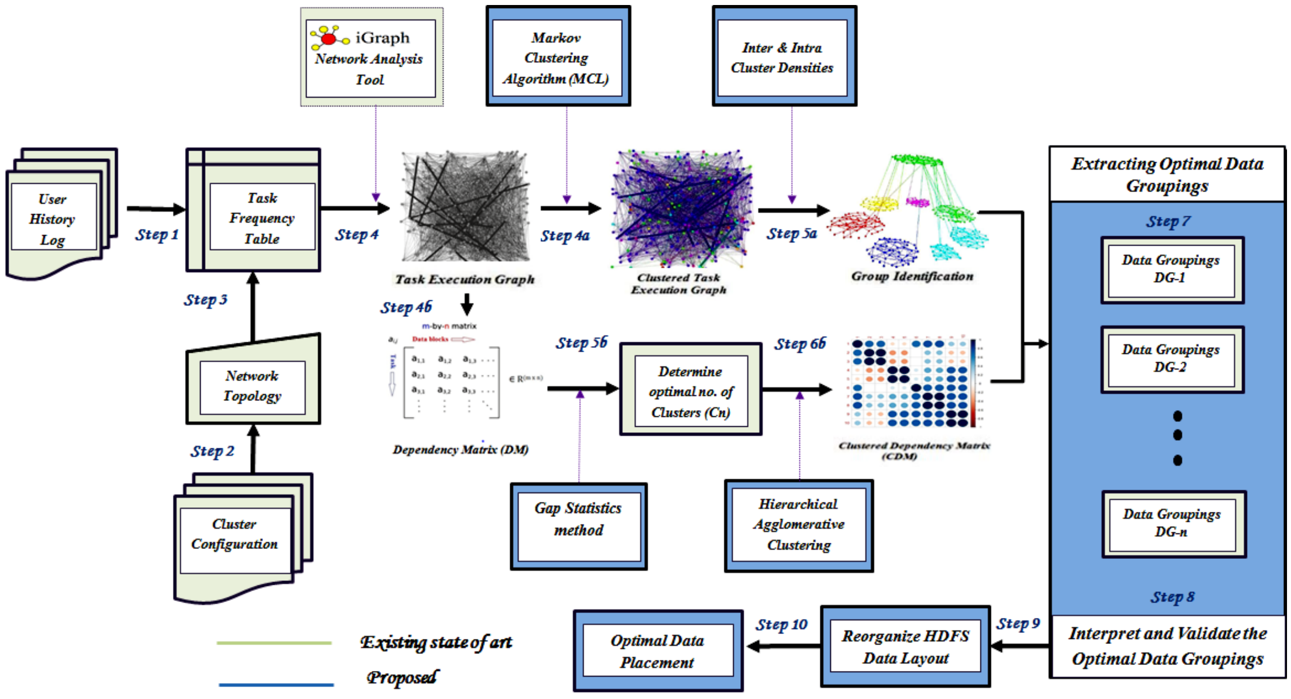


FIG. 3.1. *Workflow diagram for the proposed work*

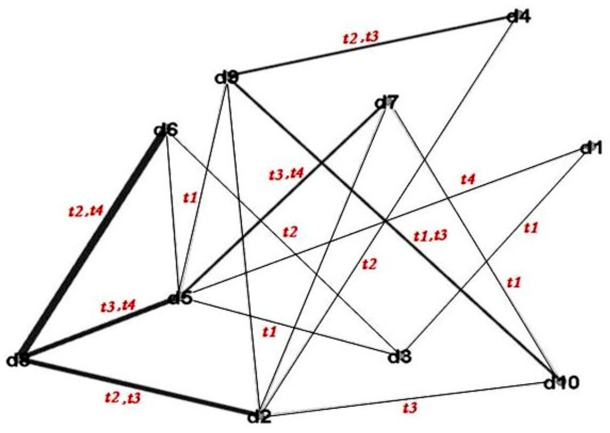


FIG. 3.2. *Task execution graph for sample graph consisting of 10 blocks and 4 tasks*

converted into a clustered task execution graph ($CGTex$) by applying the graph clustering algorithm [15]. The normal representation of the graph may not reveal any natural cluster characteristics. When a uniformly distributed graph is applied with a clustering algorithm, the graph will be arbitrarily grouped into clusters based on the similarity metric. To identify the natural groupings in the graph, MCL algorithm, fast, scalable, and unsupervised algorithm, is applied over the $GTex$ and the various stages of the clustered graph obtained are shown in Fig.3.3.

**Step 5a:** ***Group Identification*** The clusters obtained from the clustered task execution graph ($CGTex$) are separated into various groups. A subset of vertices can be said to form a good cluster if sub-graphs are dense with more connections within the group and only a very few connections exist from the group to the rest of graph. Accordingly, each group in the cluster will have individual characteristics showing high intra-cluster

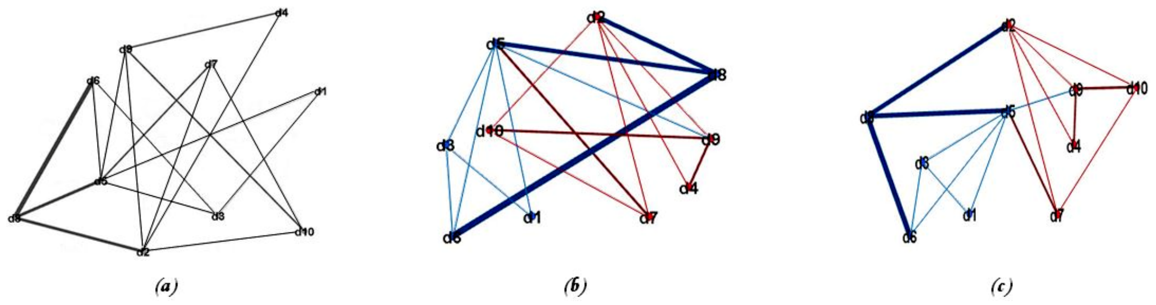


FIG. 3.3. *Various stages of clustered graph by applying MCL algorithm*

and low inter-cluster density (refer eqns 3.1 and 3.2). Based on the grouping behaviour, the associated clusters are grouped together by applying MCL.

$$Intra\ cluster\ density\ \delta_{int}(c) = \frac{|\{\{v,u\}|v \in C, u \in C\}|}{|C|(|C|-1)} \tag{3.1}$$

$$Inter\ cluster\ density\ \delta_{int}(G|C_1,....,C_k) = \frac{1}{k}\sum_{i=1}^{k}\delta_{int}(c_i) \tag{3.2}$$

**Step 4b:** ***Constructing Dependency Matrix (DM)*** From the task execution graph ($GTex$) and the information available from the task frequency table, the dependency matrix ($DM$) is constructed. $DM$ is a symmetric matrix of order $nxn$, where $n$ is the number of blocks present in the cluster. $DM$ exhibits the degree of dependency between various blocks during simultaneous execution of tasks. The diagonal elements of the $DM$ represent the number of tasks for which the corresponding block is required. Any other element in $DM_{ij}$ will show the number of tasks for which one block $b_i$ will be accessed along with the block $b_j$ for execution.

**Step 5b:** ***Determining optimal no. of clusters - Gap Statistics (GS)*** The gap statistic method can be used to calculate the optimal number of clusters for the given dataset. The gap statistic compares the total within intra-cluster variation ($w_k, wk$) for different values of $k$ with their expected values under null reference distribution of the data. The gap statistic for a given $k$ is defined as follows:

$$Gap_n(k) = E_n^*\{log(W_k)\} - log(w_k) \tag{3.3}$$

$$Gapn(k) = En * log(Wk) - log(Wk) \tag{3.4}$$

The standard deviation ($sd_k\ sdk$) of $log(W_k^*)log(Wk_*)$ is also computed in order to define the standard error ($S_k\ sk$) of the simulation as follows.

$$s_k = sd_k * \sqrt{1+\frac{1}{B}} \tag{3.5}$$

$$sk = sdk * 1 + 1/B \tag{3.6}$$

Finally, a more robust approach is to choose the optimal number of clusters $K$ as the smaller $k$, such that:

$$Gap(k) \geq Gap(k+1) - s_{k+1} \tag{3.7}$$

The smallest value of $k$ is chosen so that the gap statistics is within one standard deviation of the gap at $k+1$. Based on this, we can calculate the optimal number of clusters for a given dataset.

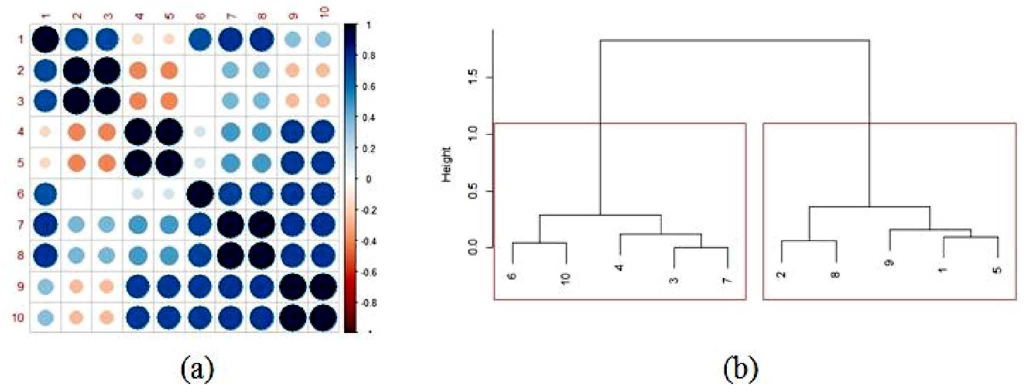


FIG. 3.4. *(a) Clustered correlation matrix and (b) Dendogram for hierarchical clusters*

**Step 6b:** ***Clustered Dependency Matrix (CDM)*** The dependency matrix ($DM$) is then converted into a clustered dependency matrix ($CDM$) by applying the matrix clustering algorithm. In this paper, HAC is used to cluster the matrix into groups. The application of HAC technique is examined for the proposed work and is explained below. In the HAC method, a hierarchy of clusters is formed to identify the natural groupings in the dataset. A bottom-up approach is used in this algorithm. Initially, HAC considers each data block as a single entity, then it combines the blocks with most similar blocks to form a bigger cluster. The iteration is then repeated, with further merging of the clusters with the output obtained earlier. Once all the blocks are merged into the required number of clusters derived from the gap statistics in step 5b, the algorithm ends.

The clusters obtained are merged based on similarity/dissimilarity measures. The Euclidean distance is used to measure the distance between each pair of data blocks $(d_i, d_j)$ from the dataset D $((d_i, d_j) \subseteq D)$.

$$Dist_{eq}(d_i, d_j) = \sqrt{(x_{d_i} - x_{d_j})^2 + (y_{d_i} - y_{d_j})^2} \tag{3.8}$$

To merge two blocks in a cluster, linkage methods can be used to decide the neighbouring pair of blocks to be merged. In this paper, a single linkage method is adopted. It computes all pairwise dissimilarities between the elements in cluster 1 and the elements in cluster 2, and considers the smallest of these dissimilarities as a linkage criterion.

$X_1, X_2, ..., X_k$ = Observations from Cluster1,
$Y_1, Y_2, ..., Y_k$ = Observations from Cluster2,
$d(x, y)$ = Distance between observation vector $X$ with $Y$.

$$SingleLinkage\ :\ d_{12} = min_{i,j}\ d(X_i, Y_j) \tag{3.9}$$

The reason for the use of HAC is due to its flexibility, versatility and, mostly, its lower computational complexity. The HAC algorithm clusters the highly associated data together based on the grouping behaviour and generates data groupings as shown in Fig.3.4. Initially, each data is considered as a cluster. Computing the distances between all pairs of data blocks takes $O(m^2)$ computation. Then, the data is sorted to find the smallest, which takes $O(m^2\ log\ m)$ time. The closest pair are then merged and all the distance pairs are again recomputed with the new cluster, which takes $O(m\ log\ m)$. The iteration process continues $(m-1)$ times until all the data merges to form a single cluster, which takes $(m-1) * O(m\ log\ m)$. Hence, the computational complexity for HAC to find the natural groupings of data blocks in the dataset takes $O(m^2\ log\ m) + (m-1) * O(m\ log\ m) = O(m^2\ log\ m)$.

**Step 7:** ***Extracting ODG*** Then, both the HAC and MCL algorithms with the optimal number of clusters are independently applied over the history log. The resulting output of each method will be a unique set of data groupings. It is confirmed that each grouping obtained is conceptually distinguishable by validating and interpreting each obtained group.

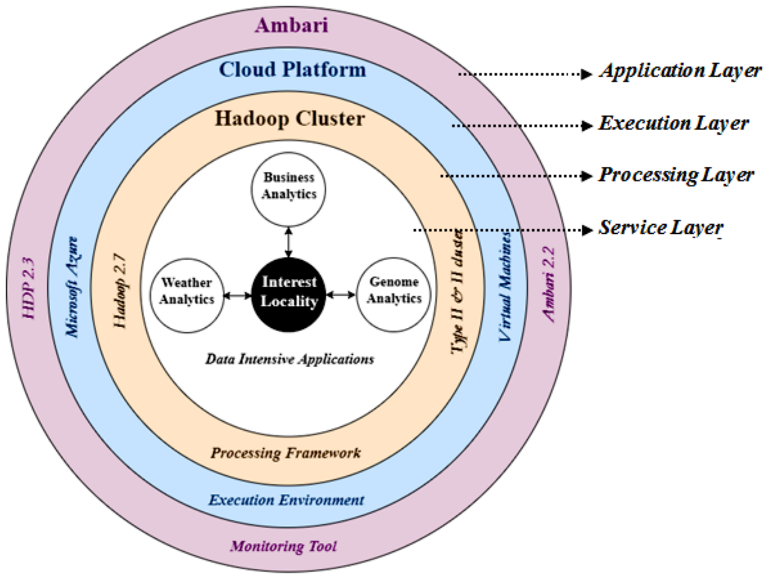


FIG. 4.1. *Schematic diagram for execution framework*

**Step 8:** ***Interpreting and validating ODG*** Then, the extracted data groupings must be interpreted and analysed to find how well the obtained groupings fit the data without reference to external attributes. Then, the data groupings obtained from the two different sets of cluster analysis are compared to determine the optimal data groupings using the silhouette method. We can separately execute and test the validated groupings for local map tasks in distributed settings.

**Step 9:** ***Reorganising HDFS data layout*** The implementation of our proposed strategy will dynamically reorganise the HDFS data layout in order to achieve an optimal data placement for improved execution; this program for proposed work is launched as a utility to be executed manually as and when required. The execution of this utility modifies the machine instruction, which is a triplet $< B_{id}, SN, DN >$, where $B_id$ is the Block ID, $SN$ is the Source Node, and $DN$ is the Destination Node. If $SN$ and $DN$ are different, then the reorganisation has been carried out considering the rack topology and the load balancer.

**Step10:** ***Achieving optimal data layout*** After reorganisation of the default data layouts in HDFS, our proposed work achieved an optimal data layout that ensures maximum parallel execution per group. It does not guarantee 100% local map task execution every time, but it will always produce an improved result over the naive data placement strategy, which is tested with the production cluster (explained in detail in the subsequent section).

**4. Experimental Results and Analysis.** The experiments were tested in a cloud platform, since the cloud is emerging as a preferred paradigm to deploy highly available and scalable systems for the processing of BigData [27]. It is also a reliable, fault-tolerant, flexible, and low-cost environment. Microsoft Azure provides a platform to collect, store, process, analyse, and visualise BigData in the cloud.

In order to carry out the experiments, 10 node heterogeneous clusters, deployed in a multi-rack environment, with every node having Hadoop, were established in the Azure cloud. The cluster was configured with one Master (NameNode) and nine Slaves (DataNodes). In order to have a heterogeneous environment, the DataNodes were chosen with varied configurations. Table 4.1 and Table 4.2 depicts the detailed cluster configuration, file system configuration respectively. The clusters were provisioned, managed, and monitored using Apache Ambari. The schematic diagram for the execution framework is shown in Fig. 4.1.

To evaluate the performance of MR, we experimented with an Amazon product review dataset [28] consisting of product reviews from Amazon, spanning approximately 18 years (1996-2014). This dataset covers reviews of multiple products such as Books, Baby products, Electronics, Kindle store, Movies and TV, Health and

Table 4.1
*Cluster configuration*

| Property | NameNode- 1 | DataNode- 9 | | |
|---|---|---|---|---|
| | NN- 1 | DN- 2 | DN- 3 | DN- 4 |
| **Instance Type** | DS5 v2 | DS4 v2 | DS3 v2 | DS2 v2 |
| **vCPU** | 16 | 8 | 4 | 2 |
| **RAM** | 56 GB | 28 GB | 14 GB | 8 GB |
| **Processor** | Intel Xeon E5-2673@2.4 GHz | | | |
| **OS** | CentOS 7.3 | | | |
| **Hadoop Version** | Hadoop 2.7.2 (Stable Version) | | | |

personal care etc. This dataset (size 19.5GB) is freely available to download from Stanford Network Analysis Project (SNAP). Each of the reviews will contain the following information (ProductID, Title, UserID, Price, Helpfulness, ProfileName, Score, Time, Summary).

Table 4.2
*Data, distributed file system and cluster - configuration parameters*

| **HDFS Status : Healthy** | |
|---|---|
| **Total Size** | 19651541778 B |
| **Total files** | 5 |
| **Average block size** | 66390343 B |
| **Total blocks (validated)** | 296 |
| **Default replication factor** | 1 |
| **Number of DataNodes** | 9 |
| **Number of racks** | 3 |

Table 4.3
*Data relating to interest domain*

| Name | Size | Block Size | No. of Records |
|---|---|---|---|
| **Reviews_Baby.json** | 580.22 MB | 64 MB | 915446 |
| **Reviews_Books.json** | 13.74 GB | 64 MB | 16302134 |
| **Reviews_Electronics.json** | 1.38 GB | 64 MB | 1689188 |
| **Reviews_Kindle_Store.json** | 789.46 MB | 64 MB | 982619 |
| **Reviews_Movies_and_TV.json** | 1.85 GB | 64 MB | 1697533 |

During the execution of interest-based queries, it is observed that there is a severe drag in MR performance. In the business forecasting domain [16, 17] in particular, to predict future product demand/sales of particular products, the reviews in respective categories alone need to be analysed rather than sweeping through the reviews in all categories. The data relating to the interest domain in the Amazon review data is shown in Table.4.3. When this data relating to the interest domain is uploaded in HDFS, the data splits into even-sized data blocks and distributed randomly across the DataNodesThe data are placed without any consideration of the nature of the queries likely to be executed. Due to this, it is possible that dependent blocks required for execution will be concentrated within fewer computing nodes, resulting in several lacunas such as underutilisation of resources and increased query execution time.

To prove the significance of clustering in data placement, several experiments were conducted by executing various interest-based queries (Join and Aggregate) related to business analytics (e-commerce dataset). The tasks were chosen in such a way that they had specific dependent blocks and were executable only within a subset of the whole dataset. Application benchmark performance was also executed for the evaluation, e.g. different tasks related to prediction modelling (regression) for different products was executed for evaluation.

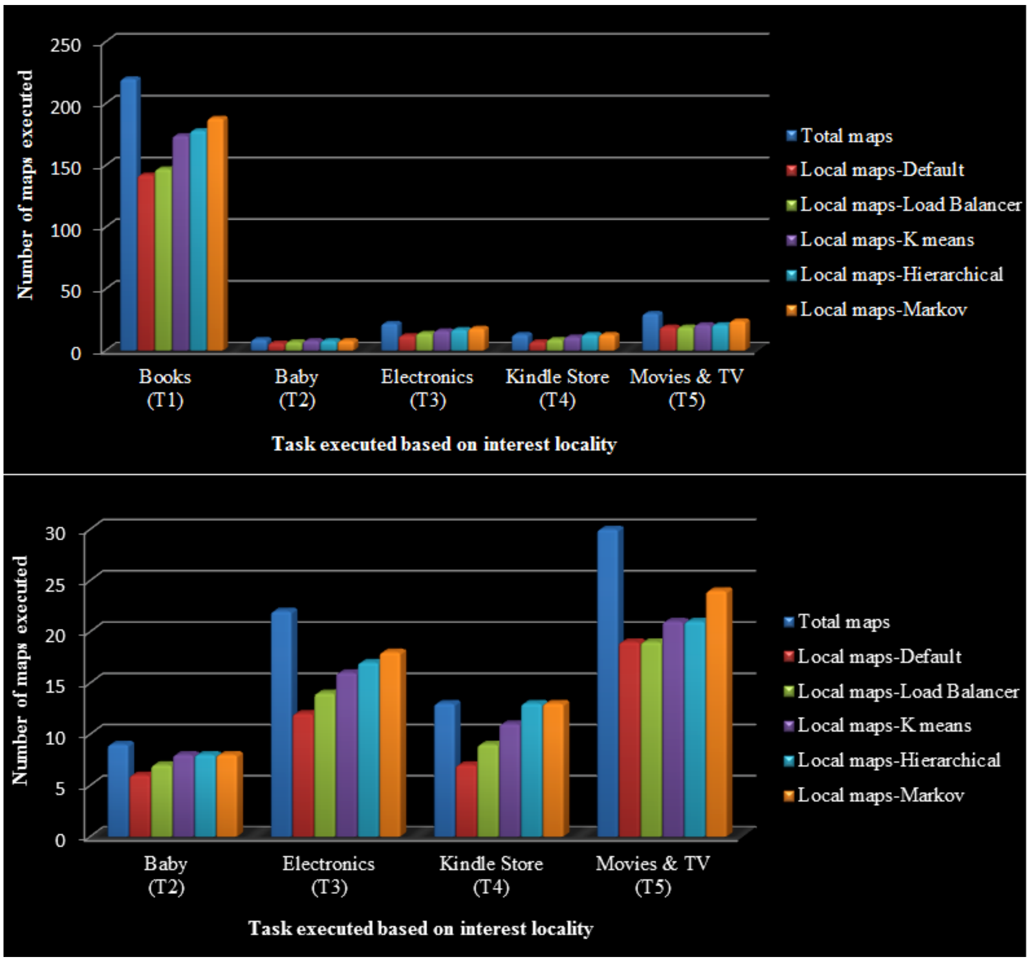


FIG. 4.2. *Graphs showing the performance improvement in local map task execution*

Other join and aggregate queries were also taken into consideration, e.g. finding an electronic product with a higher rating during a specific period, finding a book that has been reviewed more, finding the usefulness of a Kindle product during 2006 to 2007etc.

The join and aggregate queries on e-commerce were written using PIG scripts and executed in a TEZ execution engine. The prediction modelling for business analytics was written using Mahout, a scalable machine learning library, and executed in the MR execution engine. The output metrics were collected using Ambari monitoring tool, deployed in the HDP platform. These applications were executed in real time and the performance was compared with existing data placements such as HDDPS, load balancer, and proposed data placement with different clustering algorithms (k-means, HAC, MCL). The output presented in Table 4.4 shows an interesting result, with improved local map task and reduced execution time. Fig. 4.2 and 4.3 depict the graphical representations.

From Table 4.5, with a maximum of 296 maps required for execution, HDDPS has 186 data local maps (i.e. 62.8%), whereas as MCL has 251 local maps (i.e. 84.7%), showing an improvement of 34.9% ((251-186)/186) of local map executions. Similarly, the execution time was also decreased from 18,879 secs to 13,762 secs, thereby showing an overall improvement of 27.1% ((18879-13762)/18879). In addition, the data placement based on MCL shows an improved performance (5.9% in data locality, 4.3% in execution time) over HAC and an improved performance (9.1% in data locality, 12.5% in execution time) over data placement based on k-means for queries exhibiting interest localities.

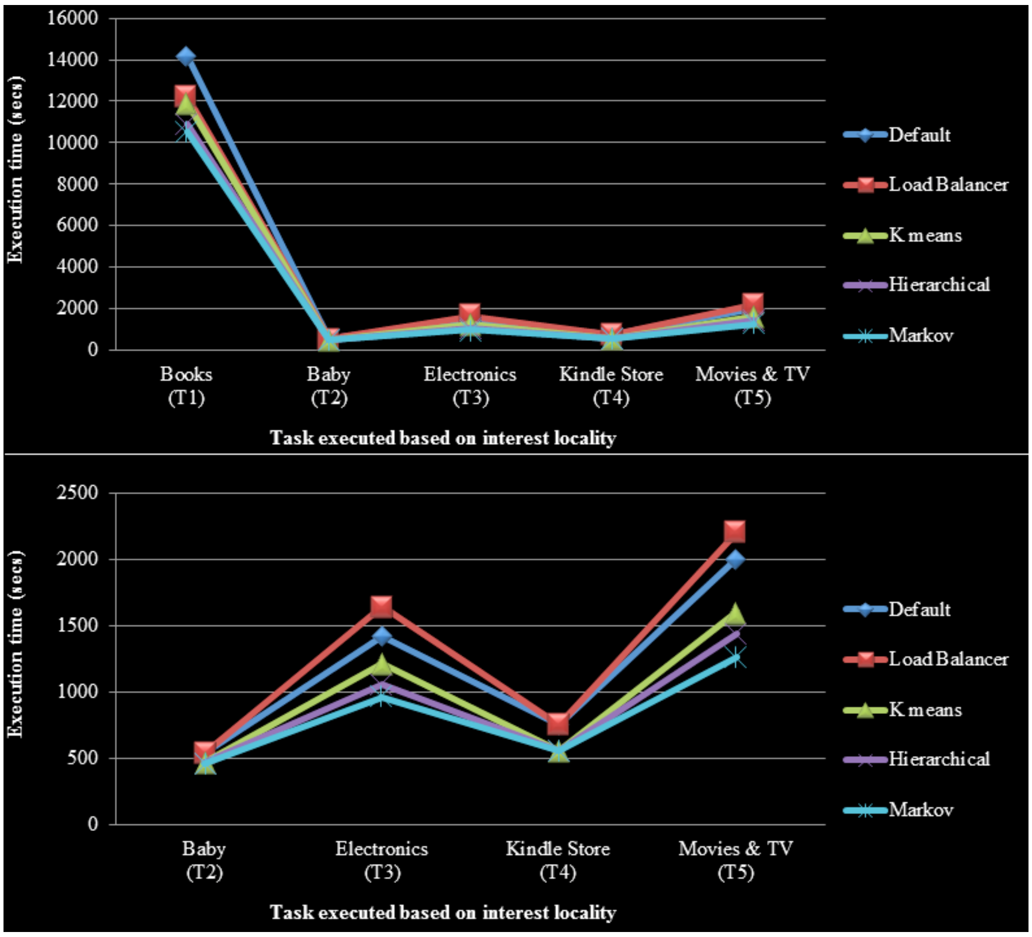


FIG. 4.3. *Graphs showing the performance improvement in execution time*

Data placement based on Markov clustering shows improved performance, especially when data-intensive applications have interest locality. This is because the natural clusters obtained through MCL exhibit higher utilisation of resources with less complexity. The reduced execution time is due to the significant improvement achieved in CPU utilisation through Markov clustering. The CPU utilisation of each node and every node in the cluster is improved. Also, the average CPU utilisation of the cluster increased from 55.2% to 81.3%, showing an improvement of 26.1%, as depicted in Fig.4.4. When tested in the worst case, where any interest locality does not exist, i.e. all data blocks are required to be accessed for execution, the proposed strategy shows the same efficiency as default.

**5. Conclusion and Future Work.** Optimal Data Placement (ODP) Strategy based on grouping semantics is proposed in this paper. The significance of different clustering techniques viz. k-means, Hierarchical Agglomerative Clustering (HAC) and Markov Clustering (MCL) in grouping-aware data placement for data-intensive applications with interest locality has been tested. The experiments were carried out in a 10-node cluster placed in a multi-rack environment deployed in the Azure cloud. The results conclude that the MCL-based data placement strategy improves the local map execution by 34.5% and reduces the execution time by 27.8% compared to Hadoops Default Data Placement Strategy (HDDPS). In addition, it can be inferred that the MCL-based data placement strategy shows an improved performance (5.9% in local map execution, 4.3% in execution time) over HAC (9.1% in local map execution, 12.5% in execution time) and over data placement based on k-means for queries exhibiting interest localities. The results strengthen the proposed work and prove to be more efficient for massive datasets processing in a distributed environment.

Table 4.4
*Performance improvement in local maps and execution time - for various interest domains*

| **Amazon Review Product Dataset - SNAP (20 GB)** | | | | | | |
|---|---|---|---|---|---|---|
| ***Data Placement*** | **Interest Domain** | **Books (T1)** | **Baby (T2)** | **Electronic (T3)** | **Kindle (T4)** | **Movies (T5)** |
| | **Total maps (nos)** | 220 | 9 | 22 | 13 | 30 |
| **Default** | *Local maps (%)* | 64.5 | 66.6 | 54.5 | 53.8 | 63.3 |
| | *Exe. time (secs)* | 14168 | **533** | **1422** | **753** | **2003** |
| **Load Balancer** | *Local maps (%)* | 66.8 | 77.7 | 63.6 | 69.2 | 63.3 |
| | *Exe. time (secs)* | 12224 | 533 | 1640 | 753 | 2202 |
| **K-means** | *Local maps (%)* | 79.0 | 88.8 | 72.2 | 84.6 | 70.0 |
| | *Exe. time (secs)* | 11882 | 472 | 1216 | 557 | 1602 |
| **Hierarchical** | *Local maps (%)* | 80.9 | 88.8 | 77.2 | 100 | 70 |
| | *Exe. time (secs)* | 10877 | 464 | 1057 | 557 | 1440 |
| **Markov** | *Local maps (%)* | 85.4 | 88.8 | 81.8 | 100 | 83.3 |
| | *Exe. time (secs)* | 10520 | **457** | **963** | **557** | **1265** |

Table 4.5
*Overall comparison of proposed strategy with existing data placement policies*

| | **Total maps (nos)** | **Local maps (nos)** | **Local maps (%)** | **Exe. time (secs)** |
|---|---|---|---|---|
| **Default** | | *186* | *62.8* | *18879* |
| **Load Balancer** | | *196* | *66.2* | *17352* |
| **k-means** | *296* | *230* | *77.7* | *15729* |
| **Hierarchical** | | *237* | *80.0* | *14395* |
| **Markov** | | *251* | *84.7* | *13762* |

Even though the results are very optimistic, there is still scope for improvement, since the layout obtained in the proposed work considers only the horizontal relationships among the data. Hence, an Optimal Data Placement (ODP) Algorithm considering inter-relationships (vertical) among the blocks can be proposed in the additional data groupings obtained, which could further improve the performance during the execution of simultaneous map tasks.

**Acknowledgments.** This work is supported under Visvesvaraya PhD scheme funded by Ministry of Electronics and Information Technology, Government of India. We also like to thank Microsoft Azure for having sponsored the cloud Infrastructure required for carrying out the experiments under Microsoft Research Award.

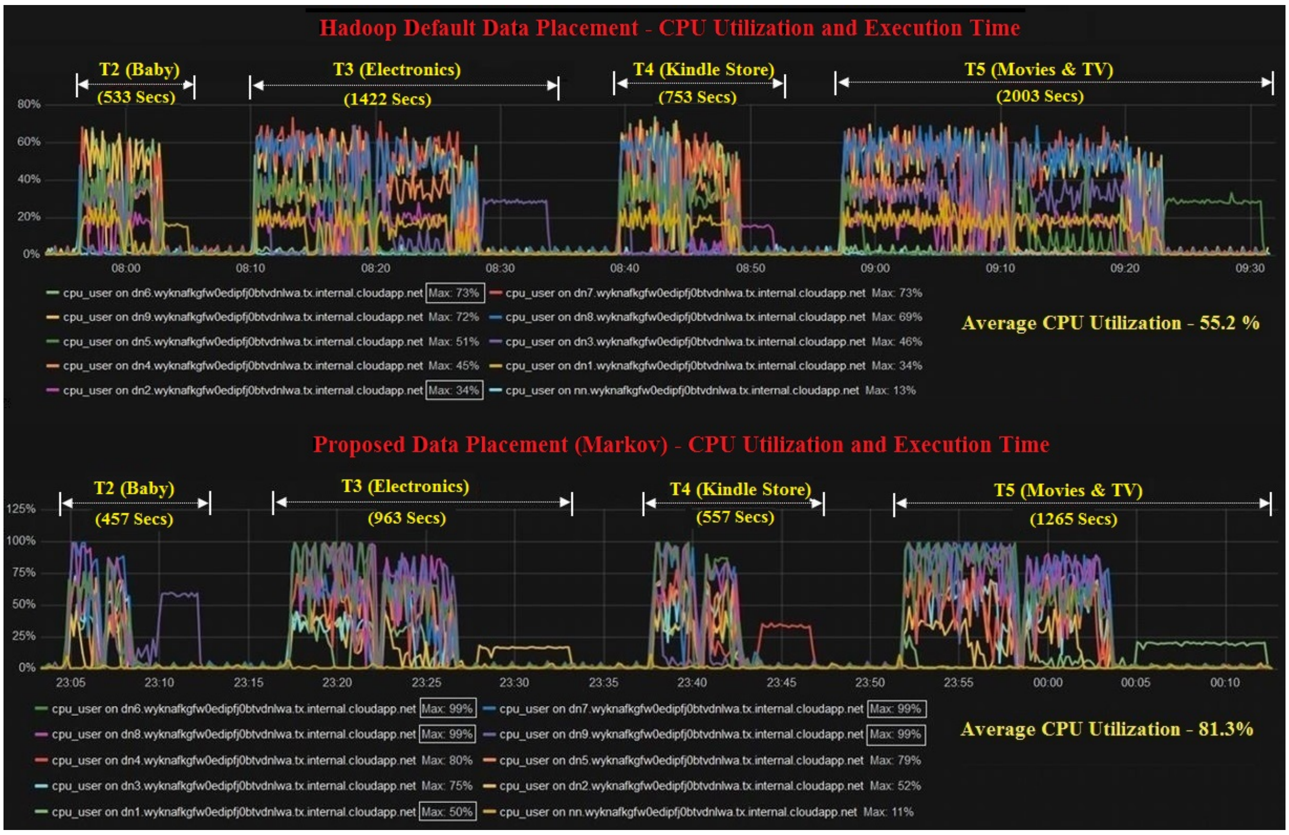


FIG. 4.4. *Graphs showing the performance improvement in CPU utilization of each DataNode*

## REFERENCES

[1] WHITE, T.: Hadoop: The definitive guide. OReilly Media, Inc., 2012

[2] HU, H., WEN, Y., CHUA, T. S., & LI, X.: Toward scalable systems for big data analytics: A technology tutorial, IEEE access, 2014, Vol.2, pp.652-687

[3] SAMMER, E.: Hadoop operations. O'Reilly Media, 2012

[4] ZHANG, Y., REN, J., LIU, J., XU, C., GUO, H., & LIU, Y.: A survey on emerging computing paradigms for big data. Chinese Journal of Electronics, 2017, Vol.26(1), pp.1-12

[5] SHIRAHATA, K, & MATSUOKA, S.: Big Data processing using Apache Spark and Hadoop. Big Data and Software Defined Networks, IET, 2018, Chap.6, pp. 115-138

[6] LITKE, W., & BUDKA, M.: Scaling Beyond One Rack and Sizing of Hadoop Platform. Scalable Computing: Practice and Experience, 2015, Vol.16 (4), pp.423-436

[7] SHVACHKO, K., KUANG, H., RADIA, S., & CHANSLER, R.: The hadoop distributed file system. Proceedings of the 26th Symposium on Mass Storage Systems and Technologies (MSST), IEEE, May 2010, pp.1-10

[8] DEAN, J., & GHEMAWAT, S.: MapReduce: simplified data processing on large clusters. Communications of the ACM, 2008, Vol.51 (1), pp.107-113

[9] CHEN, C. P., & ZHANG, C. Y.: Data-intensive applications, challenges, techniques and technologies: A survey on Big Data. Information Sciences, 2014, Vol.275, pp.314-347

[10] WANG J, XIAO Q, YIN J, & SHANG P.: 'Draw: A new data-grouping-aware data placement scheme for data intensive applications with interest locality', IEEE Transactions on Magnetics, 2013, Vol.49 (6), pp.25142520

[11] AGGARWAL, C. C., & REDDY, C. K.: Data clustering: algorithms and applications. Chapman and Hall/CRC, 2013

[12] FAHAD, A., ALSHATRI, N., TARI, Z., ALAMRI, A., KHALIL, I., ZOMAYA, A.Y., & BOURAS, A.: A survey of clustering algorithms for big data:Taxonomy and empirical analysis. IEEE transactions on emerging topics in computing, 2014, Vol.2 (3), pp.267-279

[13] KHEDR, A. M., & BHATNAGAR, R. K.: New Algorithm for Clustering Distributed Data Using k-Means. Computing & Informatics, 2014, Vol.33 (4), pp.943-964

[14] BOUGUETTAYA, A., YU, Q., LIU, X., ZHOU, X., & SONG, A.: Efficient agglomerative hierarchical clustering. Expert Systems with Applications, 2015, Vol.42 (5), pp.2785-2797

[15] Dongen, S.: Performance criteria for graph clustering and Markov cluster experiments, 2000

[16] Shao, F., & Yao, J.: The Establishment of Data Analysis Model about E-Commerces Behavior Based on Hadoop Platform. Proceedings of the International Conference on Intelligent Transportation, Big Data & Smart City (ICITBS), IEEE, January 2018, pp. 436-439

[17] Suguna, S., Vithya, M., & Eunaicy, J. C.: Big data analysis in e-commerce system using HadoopMapReduce. Proceedings of the International Conference on Inventive Computation Technologies (ICICT), IEEE, August 2016, Vol. 2, pp. 1-6

[18] Kumar, K. A., Deshpande, A., & Khuller, S.: Data placement and replica selection for improving co-location in distributed environments. arXiv preprint arXiv:1302.4168, 2013

[19] Wu, W., Lin, W., Hsu, C. H., & He, L.: Energy-efficient hadoop for big data analytics and computing: A systematic research insights. Future Generation Computer Systems, 2017

[20] Lee, C. W., Hsieh, K. Y., Hsieh, S. Y., & Hsiao, H. C.: A dynamic data placement strategy for hadoop in heterogeneous environments, Big Data Research, 2014, Vol.1, pp.14-22

[21] Xiong, R., Luo, J., & Dong, F.: Optimizing data placement in heterogeneous Hadoop clusters, Cluster Computing, 2015, Vol.18 (4), pp.1465-1480

[22] Wu, J. X., Zhang, C. S., Zhang, B., & Wang, P.: A new data-grouping-aware dynamic data placement method that take into account jobs execute frequency for Hadoop. Microprocessors and Microsystems, 2016, Vol.47, pp.161-169

[23] Liao, J., Zhang, L., Li, T., Wang, J., & Qi, Q.: Efficient and fair scheduler of multiple resources for MapReduce system. IET Software, 2016, Vol.10 (6), pp.182-188

[24] Rashmi, S., & Basu, A.: Resource optimised workflow scheduling in Hadoop using stochastic hill climbing technique. IET Software, 2017, Vol.11 (5), pp.239-244

[25] Csardi, G., & Nepusz, T.: The igraph software package for complex network research. InterJournal, Complex Systems, 2006, Vol.1695 (5), pp.1-9

[26] Tibshirani, R., Walther, G., & Hastie, T.: Estimating the number of clusters in a data set via the gap statistic. Journal of the Royal Statistical Society, 2001, Vol.63 (2), pp.411-423

[27] Manogaran, G., & Lopez, D.: Big Data in cloud data centers. Big Data and Software Defined Networks, IET, 2018, pp.159-182

[28] McAuley, J., Pandey, R., & Leskovec, J.: Inferring networks of substitutable and complementary products, Proceedings of the International conference on Knowledge Discovery and Data Mining, ACM, 2015, pp. 785-794

*Edited by:* Sasko Ristov
*Received:* Mar 1, 2018
*Accepted:* Sep 3, 2018

DOI 10.12694/scpe.v19i3.1373
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 19, Number 3, pp. 259–274. http://www.scpe.org © 2018 SCPE

# AN EXPERIMENTAL EVALUATION OF THE OPENMP THREAD MAPPING FOR SOME FACTORISATIONS ON XEON PHI COPROCESSOR AND ON HYBRID CPU-MIC PLATFORM

BEATA BYLINA AND JAROSLAW BYLINA [*]

**Abstract.** Efficient thread mapping relies upon matching the behaviour of the application with system characteristics. The main aim of this paper is to evaluate the influence of the OpenMP thread mapping on the computation performance of the matrix factorisations on Intel Xeon Phi coprocessor and hybrid CPU-MIC platforms. The authors consider parallel LU factorisations with and without pivoting as well as parallel QR and Cholesky factorizations — all from MKL (Math Kernel Library) library. The results show that the choice of thread affinity, the number of threads and the execution mode have a measurable impact on the performance and the scalability of the factorisations.

**Key words:** thread mapping, LU factorisation, QR factorization, Cholesky factorization, execution mode, Intel Xeon Phi, hybrid platform, performance

**AMS subject classifications.** 15A06, 15A30, 15A23

**1. Introduction.** Modern computing platforms are getting more and more efficient, but it comes with a price — computer architectures are getting more and more complicated. There are a lot of low-level details of machine architecture which have to be considered by HPC programmers and scientists to benefit from the promised performance. Thus, the efficient HPC software development is getting harder despite more and more capable hardware. Therefore, we have to identify and know well the existing software tools and their weak and strong points on hybrid platforms — as the one used in this paper, namely Intel Xeon CPU coupled with Intel Xeon Phi (called here a hybrid CPU-MIC platform). Such hybrid architectures add another layer of the complexity and thus, an effective level of parallelism is difficult to achieve in heterogeneous architectures — especially, when both such different units are to perform computing-intensive parts of the algorithm. This causes more and more difficulties in the optimisation of the code. One of the techniques for optimising the code in order to effectively exploit the potential of the coprocessors and the hybrid CPU-MIC platform is the thread mapping.

However, the hybrid nature of the hardware hinders the efficient use of the thread mapping in practice. There is a similar problem with the proper choice of number of threads and the prospective use of various modes (native and automatic offload). Our goal is to experimentally answer these questions. The objects of our study are some well-known and widely used algorithms, namely the LU (without and with pivoting), QR and Cholesky factorisations. We investigate the practical use of the thread mapping for different modes and the number of threads.

Operating systems on Intel Xeon Phi and on the hybrid CPU-MIC platform run numerous software threads and these threads share a complex hierarchical memory. Since the architecture consists of many processing units, these software threads have to be assigned to appropriate processing units (that is, hardware threads). Such an assignment is called thread mapping [5]. This assignment should be used to efficiently exploit the potential of modern multiprocessors. Efficient parallel numerical algorithms and their implementations on different contemporary parallel machines are crucial for engineering applications and computational science.

Determining the efficiency of the thread mapping depends on the machine and the application. There is not a single thread mapping strategy that suits all the applications. We studied the OpenMP thread mapping strategies for matrix decompositions on multicore architectures in our work [3]. The results showed that the choice of thread affinity has the measurable impact on the executed time of the matrix factorisations. Here, we extend this investigation by an experimental evaluation of the OpenMP thread mapping for the LU (without and with pivoting), QR and Cholesky factorisations from MKL library (Math Kernel Library) [15] on the Intel Xeon Phi coprocessor and on the hybrid CPU-MIC platform. While the determining of the OpenMP thread

[*]Institute of Mathematics, Marie Curie-Sklodowska University, Pl. M. Curie-Sklodowskiej 5, Lublin, 20-031, Poland, beata.bylina@umcs.pl, jaroslaw.bylina@umcs.pl

mapping on Intel Xeon Phi is not very difficult, the same task on a hybrid CPU-MIC platform remains a challenging issue. The contribution of this paper to areas of the scalable algorithms on coprocessors and hybrid platform is an experimental evaluation of the LU factorisations, QR and Cholesky factorizations from MKL library in two modes, namely native on coprocessor and automatic offload on the hybrid CPU-MIC platform. This assessment takes into account the performance for the different settings of the OpenMP thread mapping and for the different number of threads on coprocessor and the different matrix size.

The rest of this paper is organised as follows. Section 2 describes related works regarding the thread mapping and the LU factorisation. Section 3 reviews the matrix decomposition, namely the block LU factorisation with and without pivoting, QR and Cholesky factorisations. Section 4 contains the overview of Intel Xeon Phi and an introduction to the programming model on Intel Xeon Phi and the hybrid CPU-MIC platform. Section 5 presents different thread mapping strategies on Intel Xeon Phi and the hybrid CPU-MIC platform. Section 6 shows the results of numerical experiments carried out on Intel Xeon Phi and on the hybrid CPU-MIC platform for the LU factorisation with and without pivoting, QR and Cholesky factorisations and Section 7 contains some considerations about the impact of various factors on the algorithms' performance. Finally, Section 8 concludes our research and presents the future plans.

## 2. Related work.

**2.1. Thread Affinity.** In the last years, the issue of the thread mapping control in OpenMP on different parallel architectures for different applications has been researched. The authors of [14] investigated the possibilities to improve thread mapping in OpenMP programs for several simple applications (for example, SpMV — sparse matrix-vector multiplication — and Jacobi solver) and presented the ways to apply this knowledge to larger application codes on ccNUMA and multicore architecture. In the work [10], a solution to control thread mapping in OpenMP programs was presented and shown to be compatible with MPI in hybrid use cases. The authors of [12] discussed effective thread mapping strategies through comparing the computing performance and analysing the performance differences between various mapping methods using the $k$-means application program to fully exploit the computing potential of the MIC (Many Integrated Core) coprocessor, as well as the hybrid system consisting of MIC and a traditional multicore CPU. Results of these papers showed that there is no single thread mapping strategy adapted for all the applications.

**2.2. Factorisation.** Recently, several groups have been working on the efficient parallel linear algebra libraries, particularly the Gaussian elimination. The Gaussian elimination on multicore and manycore architectures was studied, among others, in works [9], [2], [6] and [8]. In the work [9] the authors investigated the parallelization of sub-cubic Gaussian elimination. They focused on the parallelization of three subroutines, namely, the matrix multiplication, the triangular equation solver and the LU factorisation with pivoting. In [2], a class of parallel tiled linear algebra algorithms for multicore architectures is presented, the LU factorisation with pivoting, Cholesky among others. The article [6] describes recent developments in parallel implementations of Gaussian elimination for shared memory architecture. Four different approaches to pivot in the LU factorisation are investigated — partial pivoting among others, and all approaches were compared with the implementation of the LU without pivoting. The comparison given in that article gives a good insight into the performance properties of the different LU factorisation algorithms using relatively large shared memory systems. In the work [8] the design and implementation of several fundamental dense linear algebra (DLA) algorithms for multicore with Intel Xeon Phi coprocessors were presented. In particular, algorithms for solving linear systems were considered, namely the LU factorisation with pivoting. The research by Intel [11] shows a great performance of LINPACK benchmark. In this work, we research the LU factorisation, QR and Cholesky factorisations implementation from a vendor library, namely MKL.

**3. Factorisations.** The LU decomposition with pivoting factorises a matrix into matrices, namely a lower triangular matrix $\mathbf{L}$, an upper triangular matrix $\mathbf{U}$ and a permutation matrix $\mathbf{P}$. It has the following form:

$$\mathbf{PA} = \mathbf{LU}$$

For improving computing performance on the contemporary computer architecture, a block version of the LU decomposition is applied. The block LU decomposition is a matrix decomposition of a block matrix into a lower block triangular matrix **L**, an upper block triangular matrix **U** and block permutation matrix **P**. The block version of the LU decomposition is implemented in LAPACK [1]. That implementation is based on BLAS. The parallelism of that block version of the LU factorisation arises from the use of a multithreaded BLAS. The MKL library provides exactly this kind of implementation of BLAS and exactly this kind of parallel version of the block LU decomposition. The block LU algorithm is described in detail in [4]. The LAPACK LU algorithm is described in the following steps:

- A panel of $b$ columns is factorised along with creation of a pivoting pattern (DGETF2 routine).
- Panel factorisation gives elementary transformations which are performed as block operations in the rest of the matrix — some rows are swapped correspondingly to the pivoting pattern (DLASWP) and top $b$ rows are treated with the triangular solver (DTRSM).
- A matrix factorisation is performed (DGEMM) — the square remainder of the matrix is updated with the product of the panel (without top $b$ rows) and the top $b$ rows without the panel items.

The LU decomposition without pivoting factorises a matrix into two matrices, namely a lower triangular matrix **L** and an upper triangular matrix **U**. It has the following form:

$$\mathbf{A} = \mathbf{LU}$$

The implementation of LU without pivoting can be carried out very rarely in practice without risking serious numerical consequences. The LU without pivoting exists if the matrix **A** has a strict dominant diagonal. Giving up the pivoting improves performance — because we get rid of the rows swapping and because the panel operations can be easily parallelized now.

The total number of floating point operations (add, multiply, divide) for the LU factorisations without and with pivoting are the same and equal approximately $\frac{2}{3}n^3$. The number of the floating point comparisons for the LU factorisation with pivoting equals approximately $\frac{1}{2}n^2$ and for the LU factorisation without pivoting equals zero. Flops measurement gives only an approximated performance — because of the differences in kernels and dynamic of the parallelism. Section 6 shows the experiments which give a better comparison.

In this work, we investigate the LAPACK implementation of the LU factorisation from MKL library, namely `dgetrf` (LU with pivoting) and `dgetrfnpi` (LU without pivoting) routines.

The QR factorisation is a decomposition of a form $\mathbf{A} = \mathbf{QR}$, where **R** is a usual upper triangular matrix, and **Q** is an orthogonal matrix (that is, $\mathbf{Q}^T\mathbf{Q} = \mathbf{Q}\mathbf{Q}^T = \mathbf{I}$). It is used to solve least square problems and eigenvalues problems. The number of floating-point operations in the QR factorisation is $\frac{4}{3}n^3 + o(n^2)$ for a given matrix **A** of the size $n \times n$. Here, we use and study the LAPACK implementation of the QR factorisation from MKL library (`dgeqrf`).

The Cholesky factorisation is a decomposition of a form $\mathbf{A} = \mathbf{L}\mathbf{L}^T$, where **L** is a lower triangular matrix — and it is defined only for **A** being Hermitian and positive-definite. The number of floating-point operations in the Cholesky factorisation is $\frac{1}{3}n^3 + o(n^2)$ for a given matrix **A** of the size $n \times n$. Here, we use and study the LAPACK implementation of the Cholesky factorisation from MKL library (`dpotrf`).

**4. Intel Xeon Phi and its programming models.** Intel Xeon Phi coprocessors [13] are multicore coprocessors designed on the basis of Intel MIC (Many Integrated Cores) architecture, where more than 50 redesigned Intel CPU cores are connected. The cores allow running up to 4 hardware threads per each core. The cores ensure hardware support for the FMA (Fused Multiply-Add) instruction and also have their own vector processing unit (VPU). Additionally, the cores are enriched with 64-bit service instructions and a cache memory. In this work, we address the first generation of Intel Xeon Phi devices known as Knight Corner (KNC). KNC is connected to CPU through the PCIe bus. Contrary, the second generation call Knight Landing (KNL) is a separate processor.

MIC provides a general-purpose programming environment similar to that provided for CPUs. It supports the source-code portability between coprocessor and CPU allowing running the same code using CPU or MIC. The Intel company offers a set of programming tools assisting programming process — such as compilers, debuggers, libraries that allow creating parallel applications (e.g. OpenMP, Intel TBB) and different kinds of mathematical libraries (e.g. Intel MKL) similarly to conventional multicore CPUs.

The MKL library on MIC can be used in two ways: native and offload. The native mode does not require changing the multithreaded code, but only adding the `-mmic` option during compilation. In the native mode,

Table 5.1
*Number of Intel Xeon Phi cores used for various affinity settings*

| **number of threads** | **cores used in the affinity setting** `compact` | `balanced` | `scatter` |
|---|---|---|---|
| 60 | 15 (4 thr./core) | 60 (1 thr./core) | 60 (1 thr./core) |
| 120 | 30 (4 thr./core) | 60 (2 thr./core) | 60 (2 thr./core) |
| 180 | 45 (4 thr./core) | 60 (3 thr./core) | 60 (3 thr./core) |
| 240 | 60 (4 thr./core) | 60 (4 thr./core) | 60 (4 thr./core) |

the MKL routines are called from the program which runs directly on the coprocessor, treated as a separate processor.

In the offload mode, the indicated parts are executed on the coprocessor and the rest on CPU and thus, this platform is treated as a hybrid CPU-MIC computing platform. Typically, the CPU controls the code execution and the data transfer between the CPU and MIC. The programmer can indicate by himself which part of the program will be executed on the coprocessor with the use of suitable pragmas or using the automatic offload version of the MKL library (which is studied in this work). Only some computationally intensive level 3 BLAS routines (`GEMM`, `TRSM`) and LAPACK functions (for example `dgetrf` and `dgetrfnpi` routine) can be called in the automatic offload mode.

To obtain the good performance for these routines we need to use square matrices of the huge size. In the automatic offload mode, the runtime system is responsible for workload division between the host (CPU) and coprocessor (MIC). Moreover, it sends data between processing units. The programmer must only make some alternations in the code. Calls to the `mkl_mic_enable()` routines in the code enable switching on the automatic offload mode of the MKL library and switching off this mode is realised by `mkl_mic_disable()`. The programmer may set the percentage of workload between the host and coprocessor by calling `mkl_mic_set_workdivision()` with proper parameters. In our code, we use `MKL_MIC_AUTO_WORKDIVISION` which indicates that division of workload between the host and coprocessor will be determined by the runtime system.

## 5. Thread Mapping.

In this section, we briefly describe the thread mapping on MIC and on a hybrid CPU-MIC platform. The thread mapping (which is included in the Intel runtime library) provides different ways to bind the OpenMP threads to the hardware threads (we have 2 hardware threads per core on CPU and 4 hardware threads per core on MIC). On CPU, there are three types, and on MIC, there are four types of distribution of the OpenMP threads between hardware threads. The first one is `compact` type: threads sequentially bound (one after another) to successive hardware threads. A single core is filled by two OpenMP threads on CPU and four ones on MIC. The second type `scatter`: threads are bound sequentially to the successive cores as evenly as possible across the entire system. `Scatter` is the opposite of `compact`. The third type is `balanced`: threads are bound evenly to the successive hardware threads, which are the neighbouring threads; this type does not exist on CPU. Using the fourth type, `none`, we leave out the order in which threads are bound to the operating system.

In this research, we control the thread affinity using the environment variable `KMP_AFFINITY` on CPU and `PHI_KMP_AFFINITY` on MIC. We studied the OpenMP thread mapping strategies for matrix decompositions on multicore architectures in our work [3]. The results showed that the choice of `scatter` has the measurable impact on the executed time of the matrix factorisations on CPU. Thus, we set `scatter` for CPUs and change only the value of the environment variable `PHI_KMP_AFFINITY`. To avoid threads migration between cores we set the value `granularity=thread` for both the environment variables.

Table 5.1 shows the usage of the system with different affinity settings. We can see that for `compact` affinity, the load balance is only ensured for 240 threads. For `scatter` and `balanced` settings, the load is always the same, although the thread arrangement is different. We can also observe that the `balanced` with 60 threads should be equivalent to `scatter` with 60 threads, and the `balanced` with 240 threads should be equivalent to `compact` with 240 threads. It is because the threads in `balanced` mode are put on cores in sequence (e.g. for 120 threads — first and second on the first core etc.), and in `scatter` mode they are put in a round robin

Table 6.1
*Hardware and software used in the experiments*

| | CPU | MIC |
|---|---|---|
| | 2 × Intel Xeon E5-2670 v.3 (Haswell) | Intel Xeon Phi 7120 (Knights Corner) |
| # cores | 24 (12 per socket) | 61 |
| # threads | 48 (2 per core) | 244 (4 per core) |
| clock | 2.30 GHz | 1.24 GHz |
| level 1 instruction cache | 32 kB per core | 32 kB per core |
| level 1 data cache | 32 kB per core | 32 kB per core |
| level 2 cache | 256 kB per core | 512 kB per core |
| level 3 cache | 30 MB | — |
| SIMD register size | 256 b | 512 b |
| compiler | Intel ICC 16.0.0 | Intel ICC 16.0.0 |
| BLAS/LAPACK libraries | MKL 2016.0.109 | MKL 2016.0.109 |

fashion (first on the first core, second on the second one etc.). Hence, the access to the memory is different. We can see that some combinations can be eliminated at sight (like `compact` with 60 threads), because only a part of the system works. Moreover, we should expect the best results for the full workload, that is for 240 threads. However, the `scatter` mode is not equivalent to `compact` and `balanced`. Thus, we expect it to behave poorer because `scatter` is less cache-friendly and the threads in the tested algorithms prefer access to neighbouring memory areas. On the other hand, the `balanced` mode reduces the data flow between caches of different cores what gives a higher throughput and lower latency. So, the `balanced` mode should give the best results, regardless of the number of threads.

**6. Numerical Experiments.** We tested the performance of two matrix factorisations, namely the block LU factorisation with and without pivoting from the MKL library on Intel Xeon Phi using native mode and on hybrid CPU-MIC platform using automatic offload mode. We compared four implementations:

- an optimised multithreaded implementation of the `dgetrfnpi` routine from the MKL library, which computes the complete LU factorisation of a general matrix without pivoting. In our case, the matrices are square, diagonally dominant and their size is $n \times n$. In the implementation of the `dgetrfnpi` routine, the panel factorisation (factorisation of a block of columns) is used, as well as the level 3 BLAS routines (`DTRSM` and `DGEMM`). We denoted this LU factorisation implementation by *LU without piv.*
- an optimised multithreaded implementation of the `dgetrf` routine from the MKL library, which computes the complete LU factorisation of a general matrix with pivoting. We denoted this LU factorisation implementation by *LU with piv.*
- an optimised multithreaded implementation of the `dgeqrf` routine from the MKL library, which computes the QR factorisation of a general matrix with pivoting. We denoted this QR factorisation implementation by *QR.*
- an optimised multithreaded implementation of the `dpotrf` routine from the MKL library, which computes the Cholesky factorisation of a symetric positive-definite matrix. We denoted this Cholesky factorisation implementation by *Cholesky.*

Table 6.1 shows details of the specification of the hardware and software used in the numerical experiments. All the experiments reported below were performed with the use of the double-precision arithmetic. In the automatic offload mode, we used all available cores on CPU and thus the number of threads was set to 24, and we changed only the number of threads on the coprocessor.

**6.1. LU factorisation without pivoting.** Figure 6.1 presents the performance of the LU factorisation without pivoting in the function of matrix size on Intel Xeon Phi in native mode for the four values of `PHI_KMP_AFFINITY` for a different number of the threads. For the native mode, we achieved the best performance for the `scatter` value of this environment variable for 120 threads or the `compact` value for 240 threads. All the

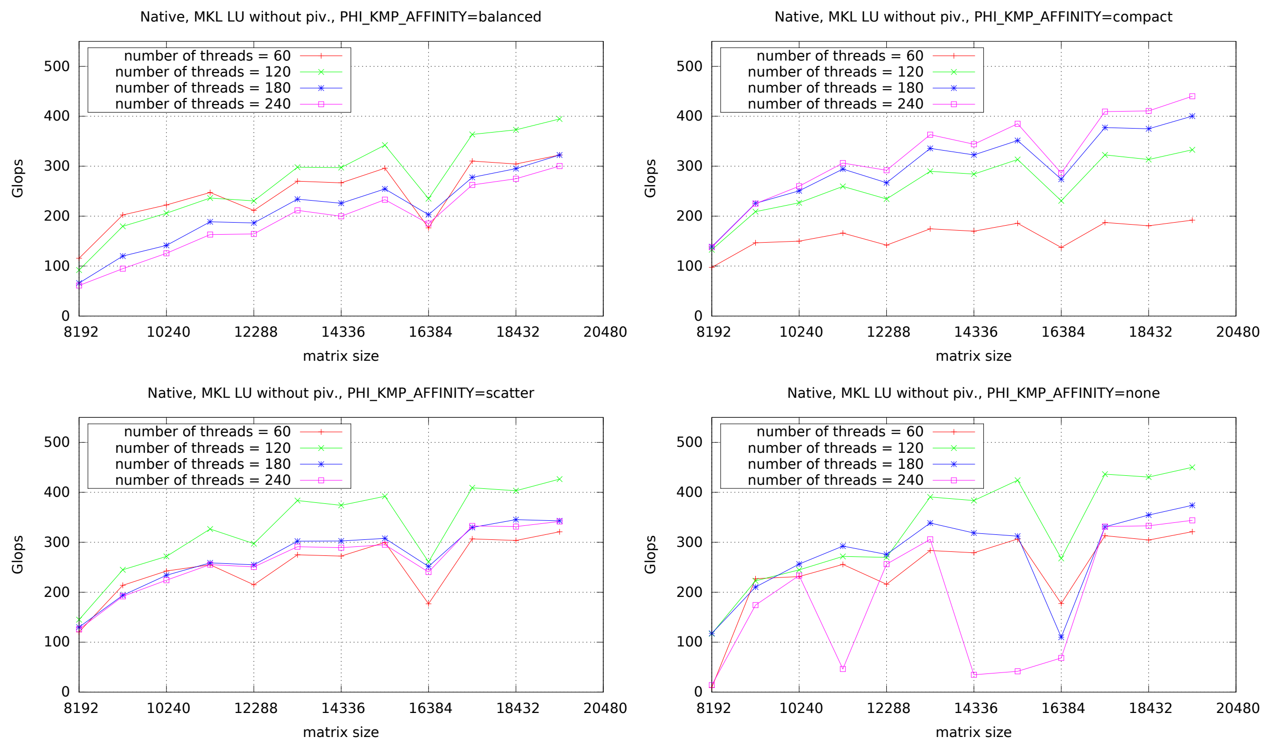


FIG. 6.1. *The performance of the LU factorisation without pivoting (MKL library's implementation) in the native mode on Intel Xeon Phi — for different matrix sizes, number of the threads, and the thread mapping settings.*

results are expected from Sect. 5 — besides `balance` with 240 threads (it should be the same as `compact` with 240 threads). The algorithm scales well with respect to the matrix size — except at the size of 16384. However, when we consider scaling with respect to the number of the threads, it is only the case for the `compact` affinity; other values give poor scalability with respect to the number of the threads. When we choose the `none` value of the affinity settings, the performance is chaotic and the scalability is poor both with respect to the size and to the threads — it is caused by the fact that the thread affinity is controlled by the operating system which makes decisions about it not suitable for computing. The last issue demanding an explanation is a sudden drop in performance at the size of $16384 = 2^{14}$. It seems to be caused by the cache size — for the matrix size of 16384 the blocks fit in cache ideally and there is no room for other data.

Figure 6.2 presents the performance of the LU factorisation without pivoting in the function of matrix size on hybrid CPU-MIC platform (with AO — automatic offload). In Fig. 6.2 (as well as in Figs. 6.3, 6.5 and 6.6), `cpu_aff/mic_aff` denotes the affinity settings both for CPU (the first value) and for coprocessor (the second value). The run-time reports say that the algorithm works exclusively on CPU up to the size of 14336 (the run-time systems believes that including MIC cannot improve the performance for such small data), so there is a poor scalability here. For bigger matrices, there is a performance drop, because the MIC gets some work and it is somewhat slower then sole CPU; however, after that, the performance grows almost up to the earlier level.

Table 6.2 shows the percentage work division between CPU and MIC for the LU decomposition without pivoting (for 24 threads on CPU and 240 threads on MIC). It is hard to determine the best number of threads because the computations — even for big matrices — are performed mainly on CPU. Thus, all the sizes except 14336 perform similarly (with the performance of about 600 Gflops) — the matrix size and the Xeon Phi settings (the number of threads and the affinity) matter little.

Figure 6.3 shows the performance of the LU factorisation in the function of the number of the threads for the matrix size of 19456 on Intel Xeon Phi in native mode and on the hybrid CPU-MIC platform in automatic offload mode for `KMP_AFFINITY=scatter` on CPU and the different values for `PHI_KMP_AFFINITY`. We can see

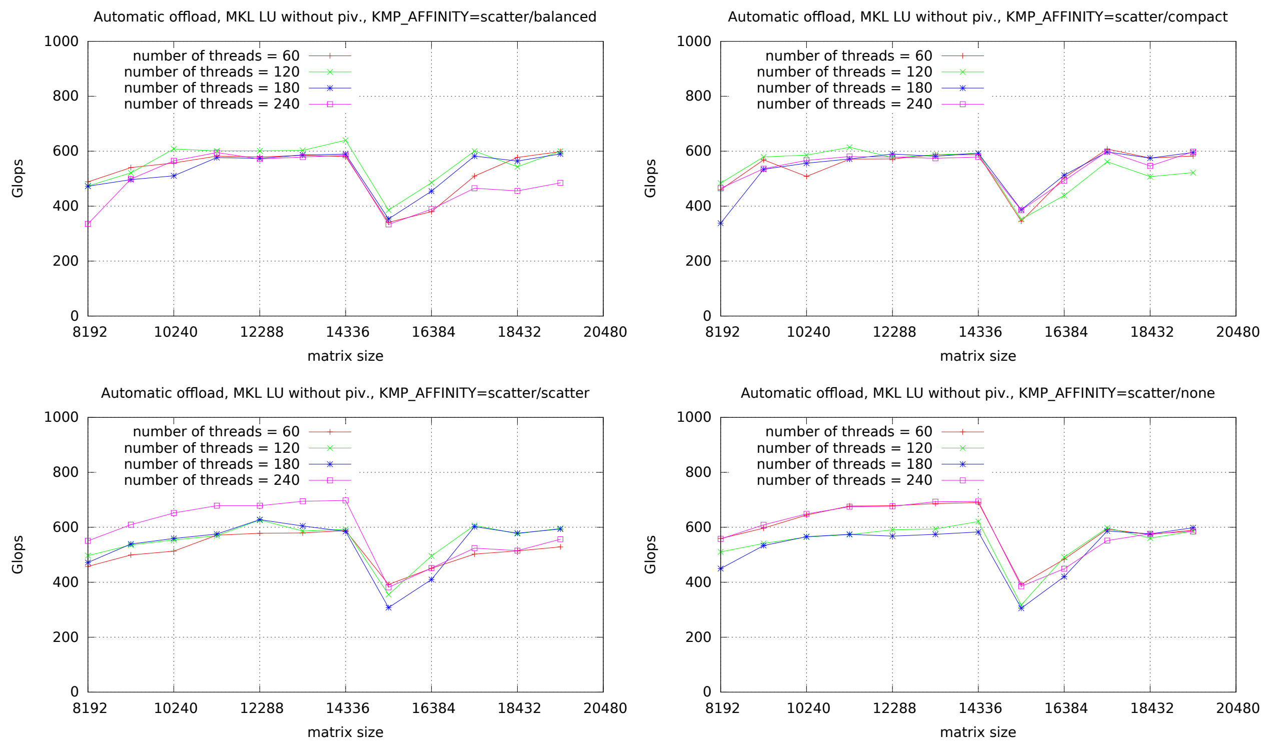


Fig. 6.2. *The performance of the LU factorisation without pivoting (MKL library's implementation) in the automatic offload mode — for different matrix sizes, number of the threads, and the thread mapping settings.*

Table 6.2
*An exemplary percentage work division between CPU and MIC for the LU decomposition for the automatic offload*

| **matrix** | DTRSM | | DGEMM | |
|---|---|---|---|---|
| **size** | CPU | MIC | CPU | MIC |
| 15360 | 91% | 9% | 97% | 3% |
| | 74% | 26% | 84% | 16% |
| 16384 | 89% | 11% | 93% | 7% |
| | 75% | 25% | 83% | 17% |
| 17408 | 97% | 3% | 92% | 8% |
| | 93% | 7% | 83% | 17% |
| 19456 | 93% | 7% | 90% | 10% |
| | 83% | 17% | 82% | 18% |

that the performance is better for the AO mode than the native mode. It is caused by the fact that our CPU is generally faster than the Intel Xeon Phi and even employing both of them (as in AO mode), it is not easy to boost the efficiency.

**6.2. LU factorisation with pivoting.** Figure 6.4 presents the performance of the LU factorisation with pivoting in the function of matrix size on Intel Xeon Phi in native mode for the four values of `PHI_KMP_AFFINITY` for a different number of the threads. For the native mode, we achieved the best performance for the `balanced` and `compact` values of this environment variable (both for 240 threads). These were expected from the analysis from Sect. 5. For the `balanced` and `compact` affinities, the algorithm scales very well with respect to both the size of the matrix and the number of threads. The `scatter` affinity gives quite a nice scalability only up to the

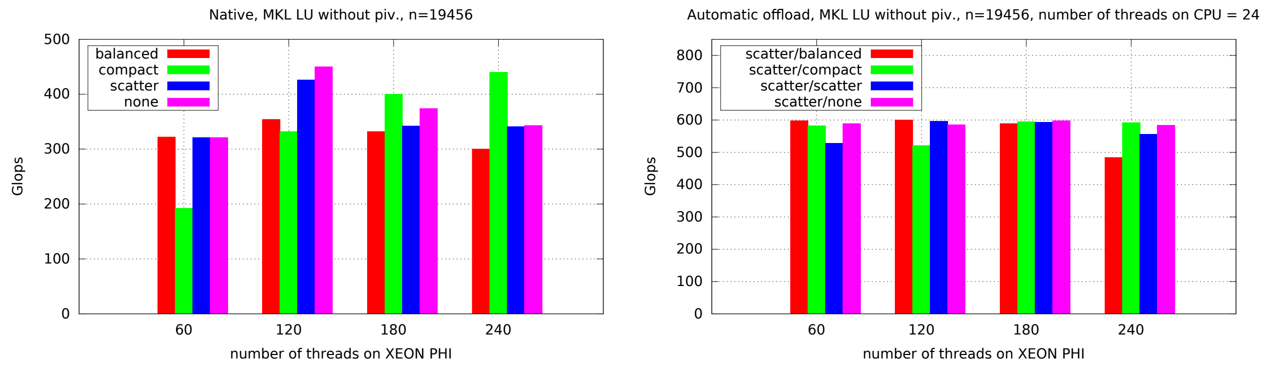


FIG. 6.3. *The performance of the LU factorisation without pivoting (MKL library's implementation) for the matrix size of 19456 on Intel Xeon Phi in the native mode (left) and on the hybrid CPU-MIC platform in the automatic offload mode (right).*

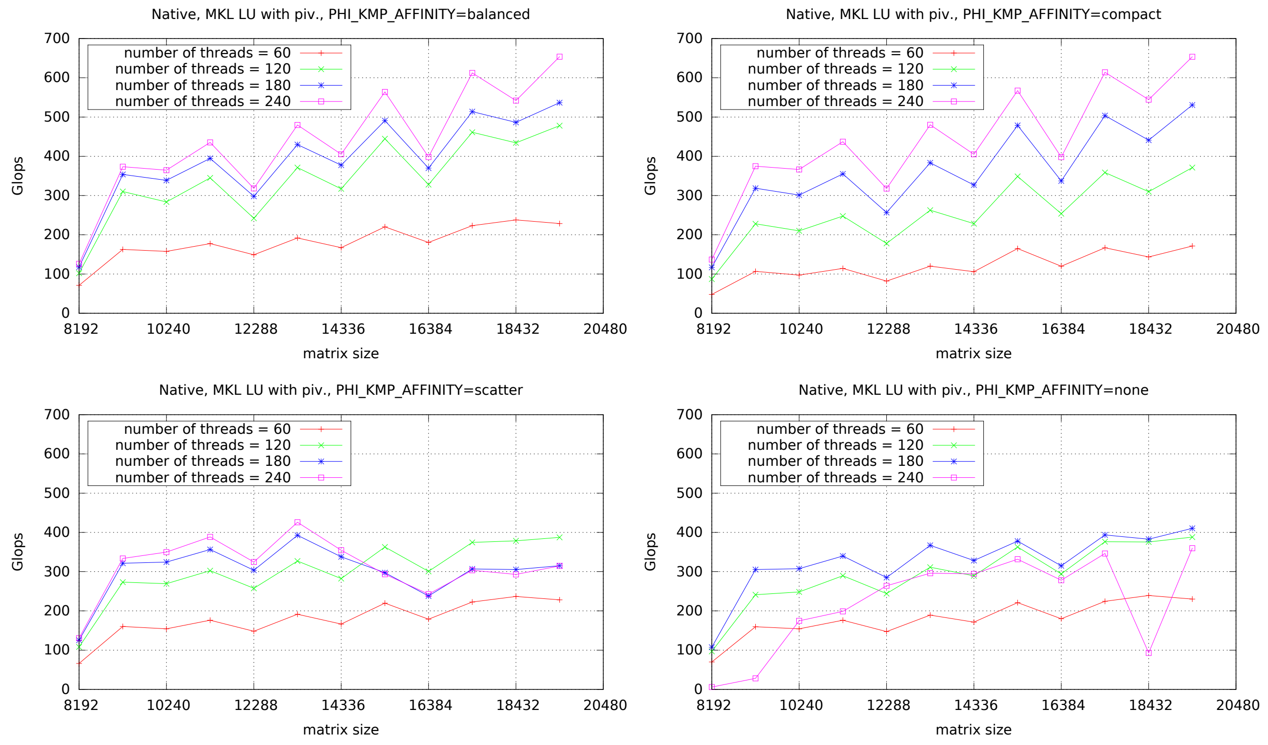


FIG. 6.4. *The performance of the LU factorisation with pivoting (MKL library's implementation) in the native mode on Intel Xeon Phi — for different matrix sizes, number of the threads, and the thread mapping settings.*

size 14336. The `none` affinity scales poor and is chaotic — just like for the version without pivoting, and for the same reason. For all affinity settings, we can see a saw shape of the chart — these spikes and drops are results of the relationship between the cache size and the size of the matrix.

Figure 6.5 presents the performance of the LU factorisation with pivoting in the function of matrix size on the hybrid CPU-MIC platform (with AO) for `KMP_AFFINITY=scatter` and 24 threads on CPU and the different values of `PHI_KMP_AFFINITY` on MIC. The algorithm scales very well with respect to both the size of the matrix and the number of threads. It seems that a lot of work is done on CPU (the report for this routine does not show the percentage work division, although, it shows the time used by both parts of the hybrid system — see

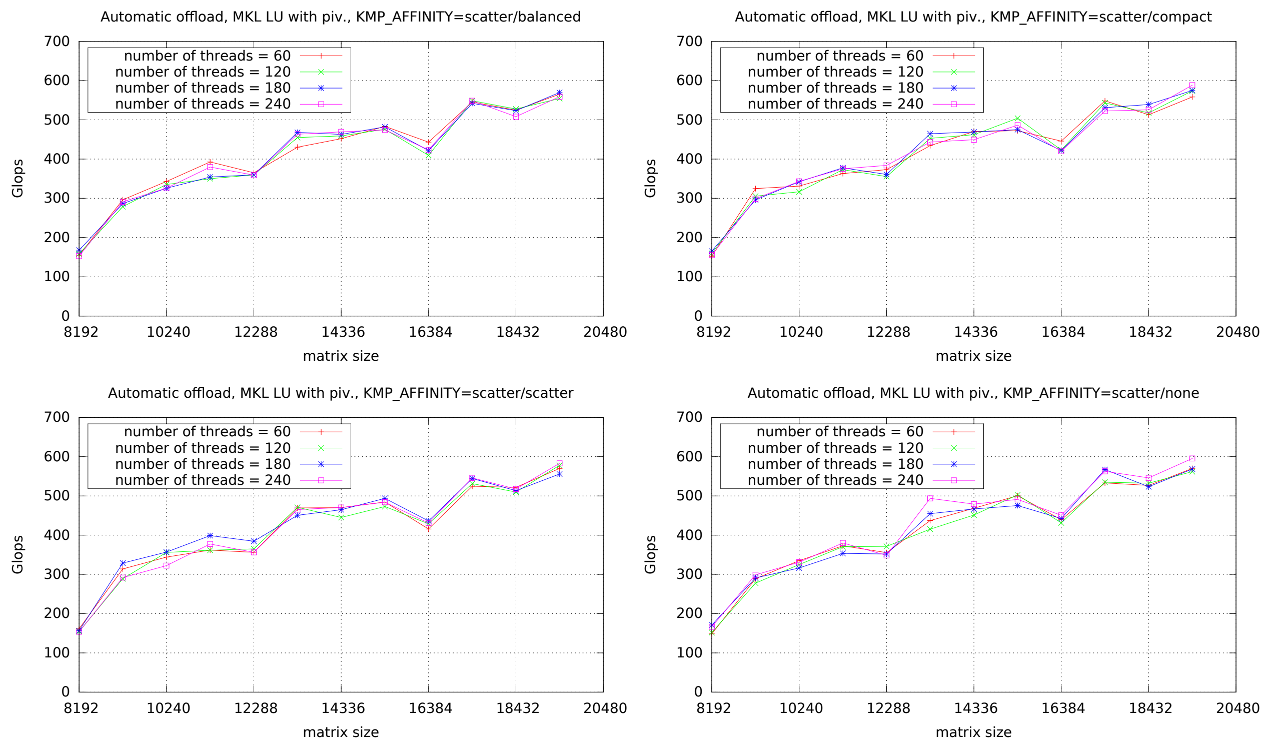


Fig. 6.5. *The performance of the LU factorisation with pivoting (MKL library's implementation) in the automatic offload mode — for different matrix sizes, number of the threads, and the thread mapping settings.*

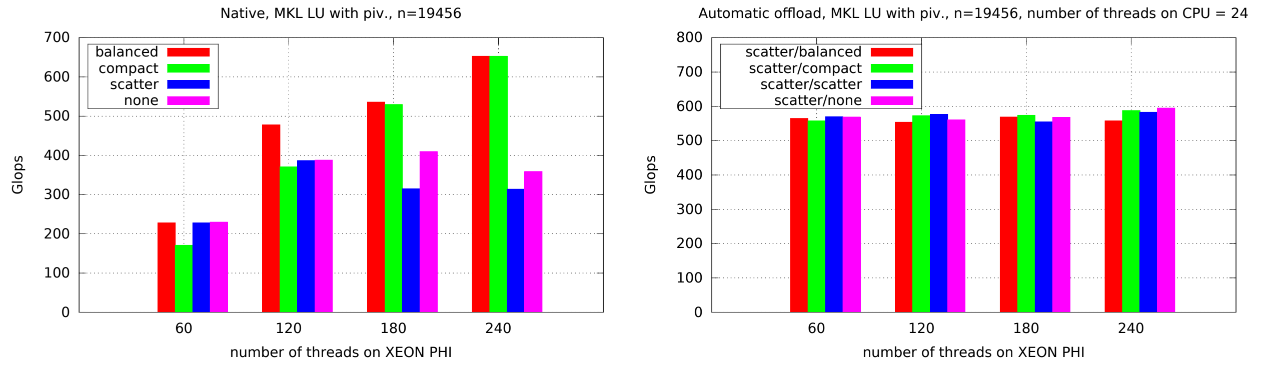


Fig. 6.6. *The performance of the LU factorisation with pivoting (MKL library's implementation) for the matrix size of 19456 on Intel Xeon Phi in the native mode (left) and on hybrid CPU-MIC Platforms in the automatic offload mode (right).*

Fig. 6.7; the time on CPU is much bigger).

Figure 6.6 shows the performance of the LU factorisation with pivoting in the function of the number of the threads for the matrix size of 19456 on Intel Xeon Phi in native mode and on hybrid CPU-MIC platform in automatic offload mode for `KMP_AFFINITY=scatter` on CPU and different values of `PHI_KMP_AFFINITY`. As we can see, the native mode version achieves the better performance than the automatic offload mode one. It seems that the former is very well optimised: it scales well with respect to both the number of threads and the matrix size. The AO version demands some more development, because (potentially) it could achieve even 1000 Gflops — taking into account combined forces of both the processing units.

```
[MKL] [MIC --]   [AO Function] DGEMM
[MKL] [MIC --]   [AO DGEMM Workdivision] 0.91 0.09
[MKL] [MIC 00] [AO DGEMM CPU Time] 0.275540 seconds
[MKL] [MIC 00] [AO DGEMM MIC Time] 0.170474 seconds
[MKL] [MIC 00] [AO DGEMM CPU->MIC Data] 33423360 bytes
[MKL] [MIC 00] [AO DGEMM MIC->CPU Data] 3932160 bytes
[MKL] [MIC --]   [AO Function] DTRSM
[MKL] [MIC --]   [AO DTRSM Workdivision] 0.97 0.03
[MKL] [MIC 00] [AO DTRSM CPU Time] 0.277691 seconds
[MKL] [MIC 00] [AO DTRSM MIC Time] 0.041793 seconds
[MKL] [MIC 00] [AO DTRSM CPU->MIC Data] 125829120 bytes
[MKL] [MIC 00] [AO DTRSM MIC->CPU Data] 7864320 bytes

[...9 analogous calls hidden...]

[MKL] [MIC --]   [AO Function] DGEMM
[MKL] [MIC --]   [AO DGEMM Workdivision] 0.91 0.09
[MKL] [MIC 00] [AO DGEMM CPU Time] 0.066654 seconds
[MKL] [MIC 00] [AO DGEMM MIC Time] 0.044188 seconds
[MKL] [MIC 00] [AO DGEMM CPU->MIC Data] 33423360 bytes
[MKL] [MIC 00] [AO DGEMM MIC->CPU Data] 3932160 bytes
```

```
[MKL] [MIC --]   [AO Function] DGETRF
[MKL] [MIC --]   [AO DGETRF Workdivision] -1.00 -1.00
[MKL] [MIC 00] [AO DGETRF CPU Time] 4.737727 seconds
[MKL] [MIC 00] [AO DGETRF MIC Time] 3.010779 seconds
[MKL] [MIC 00] [AO DGETRF CPU->MIC Data] 2796011520 bytes
[MKL] [MIC 00] [AO DGETRF MIC->CPU Data] 1242562560 bytes
```

FIG. 6.7. *Automatic offload reports generated by MKL's LU factorisation routines for the matrix size 15360 — without pivoting (top; fragments) and with pivoting (bottom; whole). The reports for MKL's QR and Cholesky factorisations are analogous to the one for MKL's LU factorisation with pivoting (bottom).*

**6.3. Comparison of the pivot and non-pivot version.** The peak performance of the AO mode is about 600 Gflops — the same for LU without pivoting and LU with pivoting. It arises from the fact that much more computations are performed on CPU (see Fig. 6.7) and both CPU versions seems equally optimised. However, in the native mode, LU without pivoting performs much worse (about 400 Gflops) than LU with pivoting (about 650 Gflops) — which is very surprising. Moreover, if we expected any differences, they would be in favour of the version without pivoting. However, from Fig. 6.7 we can see that the implementations of both factorisations are substantially different. It seems that they are very various algorithms, the pivot one being an implementation of the original LAPACK algorithm. Also, [16] says that this algorithm uses Intel Threaded Building Blocks and nothing like that is said about the non-pivot routine.

**6.4. QR and Cholesky factorisations.** Figures 6.8 and 6.9 present the performance of the QR and Cholesky factorisations in the function of matrix size on Intel Xeon Phi in native mode for the four values of `PHI_KMP_AFFINITY` for a different number of the threads.

Figures 6.10 nad 6.11 present the performance of the QR and Cholesky factorizationthe function of matrix size on the hybrid CPU-MIC platform (with AO) for `KMP_AFFINITY=scatter` and 24 threads on CPU and the different values of `PHI_KMP_AFFINITY` on MIC.

Figures 6.12 and 6.13 show the performance of the QR and Cholesky factorisations in the function of the number of the threads for the matrix size of 19456 on Intel Xeon Phi in native mode and on hybrid CPU-MIC platform in automatic offload mode for `KMP_AFFINITY=scatter` on CPU and different values of `PHI_KMP_AFFINITY`.

The performance results of the QR and Cholesky factorisations are consistent with the results of the LU factorisation with pivoting.

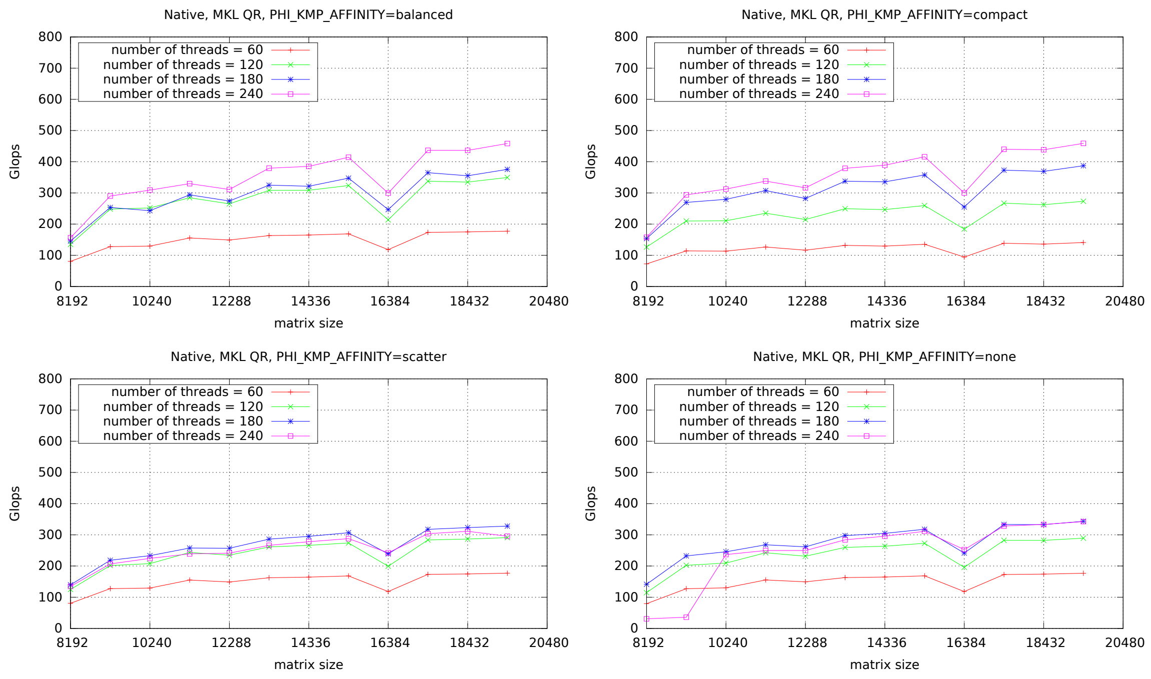


Fig. 6.8. *The performance of the QR factorisation (MKL library's implementation) in the native mode on Intel Xeon Phi — for different matrix sizes, number of the threads, and the thread mapping settings.*

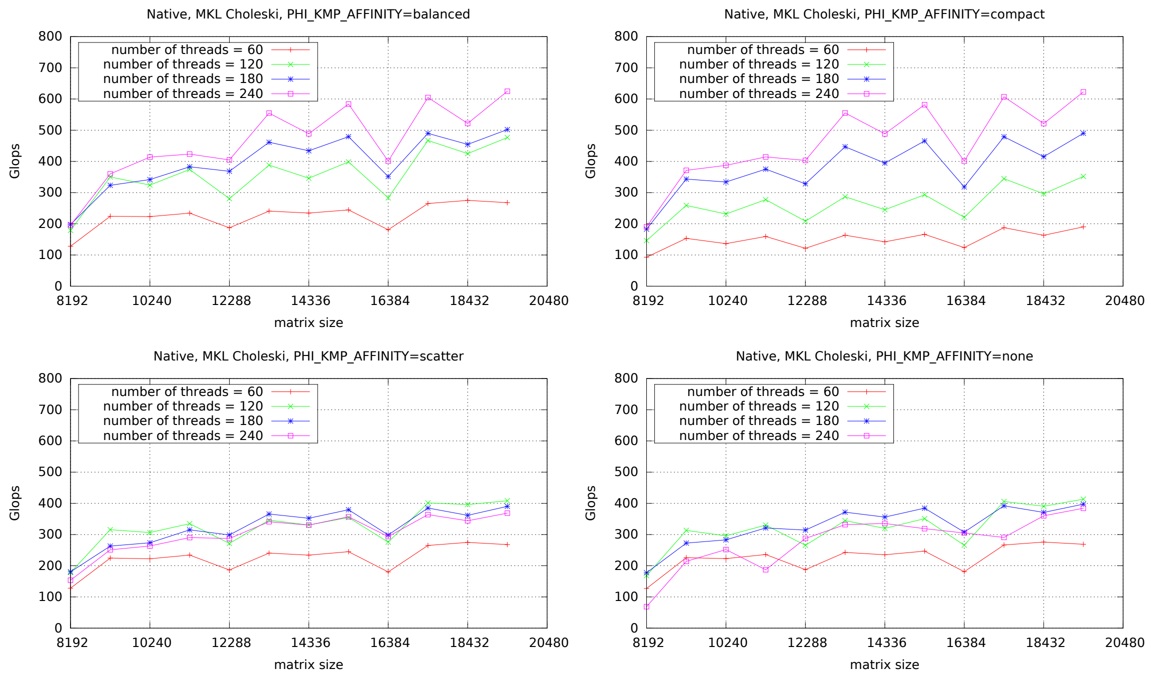


Fig. 6.9. *The performance of the Cholesky factorisation (MKL library's implementation) in the native mode on Intel Xeon Phi — for different matrix sizes, number of the threads, and the thread mapping settings.*

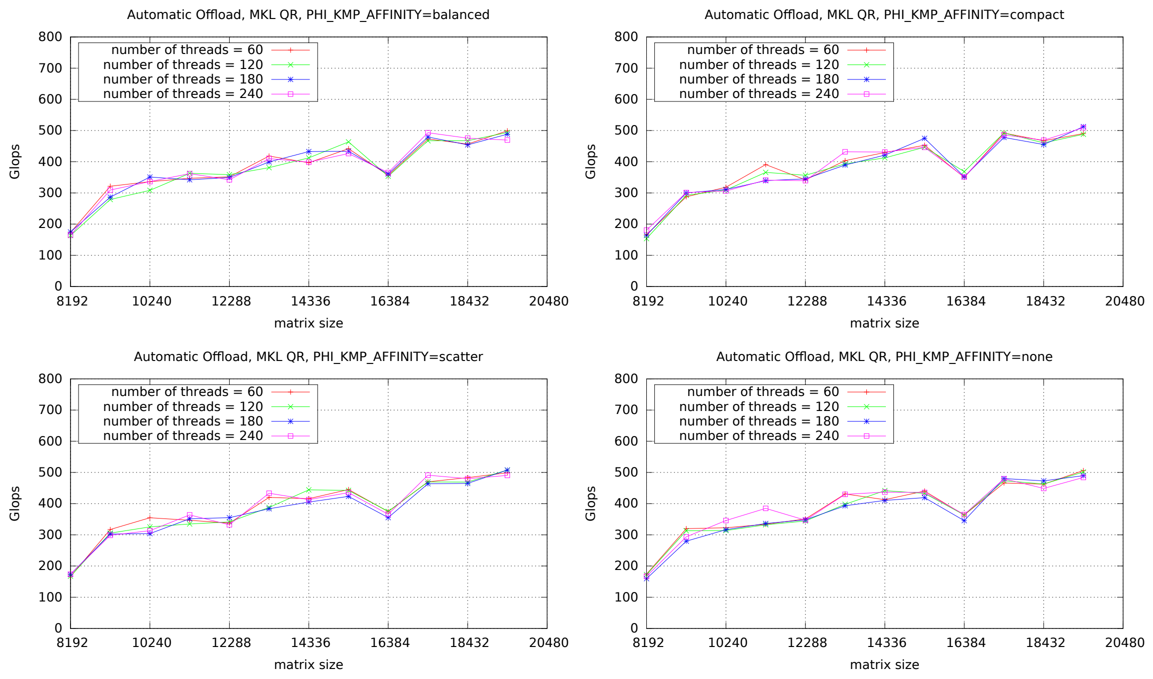


FIG. 6.10. *The performance of the QR factorisation with pivoting (MKL library's implementation) in the automatic offload mode — for different matrix sizes, number of the threads, and the thread mapping settings.*

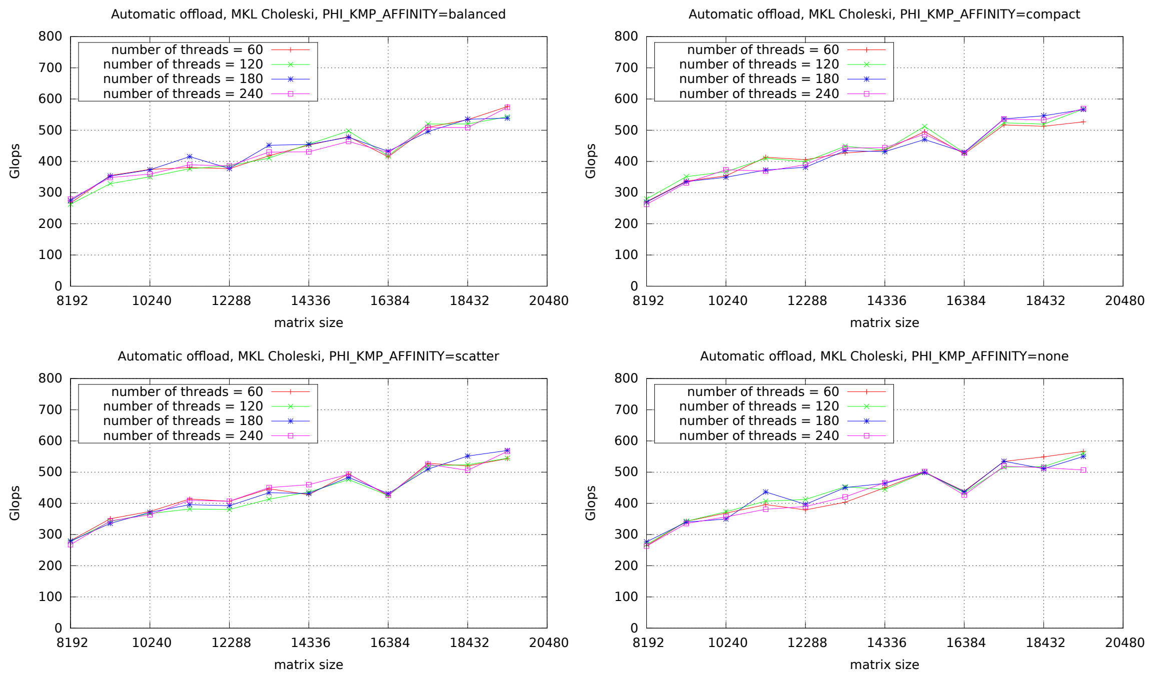


FIG. 6.11. *The performance of the Cholesky factorisation with pivoting (MKL library's implementation) in the automatic offload mode — for different matrix sizes, number of the threads, and the thread mapping settings.*

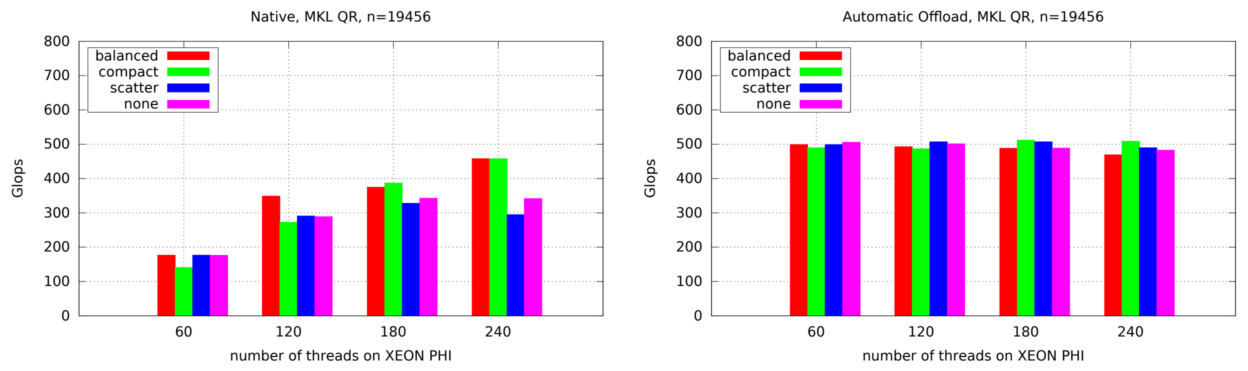


FIG. 6.12. *The performance of the QR factorisation (MKL library's implementation) for the matrix size of 19456 on Intel Xeon Phi in the native mode (left) and on hybrid CPU-MIC Platforms in the automatic offload mode (right).*

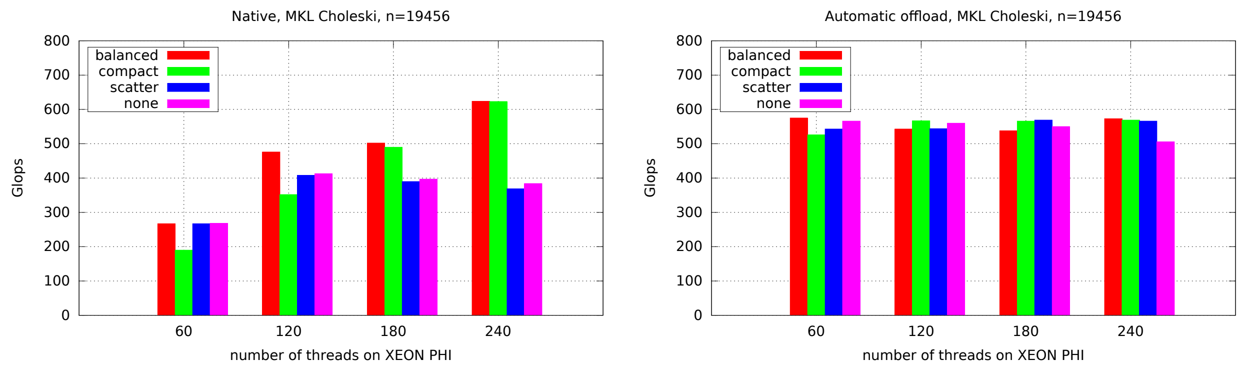


FIG. 6.13. *The performance of the Cholesky factorisation (MKL library's implementation) for the matrix size of 19456 on Intel Xeon Phi in the native mode (left) and on hybrid CPU-MIC Platforms in the automatic offload mode (right).*

## 7. Discussion.

**7.1. Thread Mapping.** It is obvious from the experiments that the proper setting of the thread mapping improves the performance. Moreover, the performance of the three factorisations is sensitive to the thread mapping. The best setting is `balanced` (see also Sect. 5), for all the tested number of threads and modes. The `balanced` thread affinity is the best because it uses well the system computing power and the cache at the same time. The `none` setting is the worst and largely unpredictable (because all the decisions are passed to the operating system and the threads can wonder freely between cores) and it should not be used in serious computational applications. The `scatter` affinity gives the second worst performance. All the performance results of the LU factorisation with pivoting, as well as QR and Cholesky facorisations (especially in native mode) confirm our analysis from Sect. 5.

**7.2. Number of threads.** All the factorisations effectively utilize a large number of cores in the native mode. Thus, it is the best to use all the cores with hyperthreading (that gives 240 threads, that is, 4 threads per core). That way, we use the computing power of the Intel Xeon Phi the most efficiently. On the other hand, the number of the threads has no impact on the performance in the automatic offload mode at all — the AO mode is controlled by the library.

**7.3. Mode.** If we can afford the native mode, we should rather use it — the native mode is better optimised than the automatic offload for the LU and Cholesky factorisations with pivoting. On the contrary, the AO is better than the native mode for the QR factorisation. It shows that this factorisation could be more optimised for the MAC architecture.

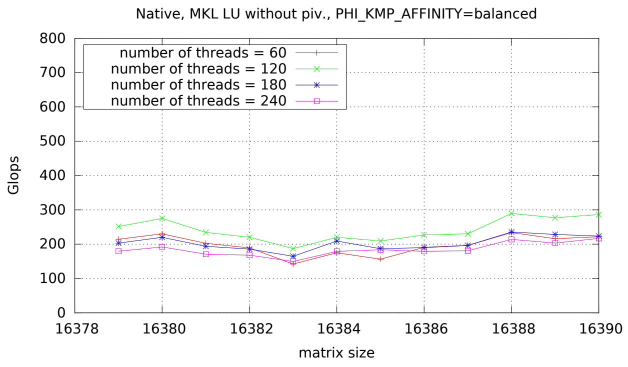


FIG. 7.1. *The performance of the LU factorisation without pivoting (MKL library's implementation) in the native mode on Intel Xeon Phi — for different matrix sizes, number of the threads, and the* `balanced` *thread mapping setting (cache associativity).*

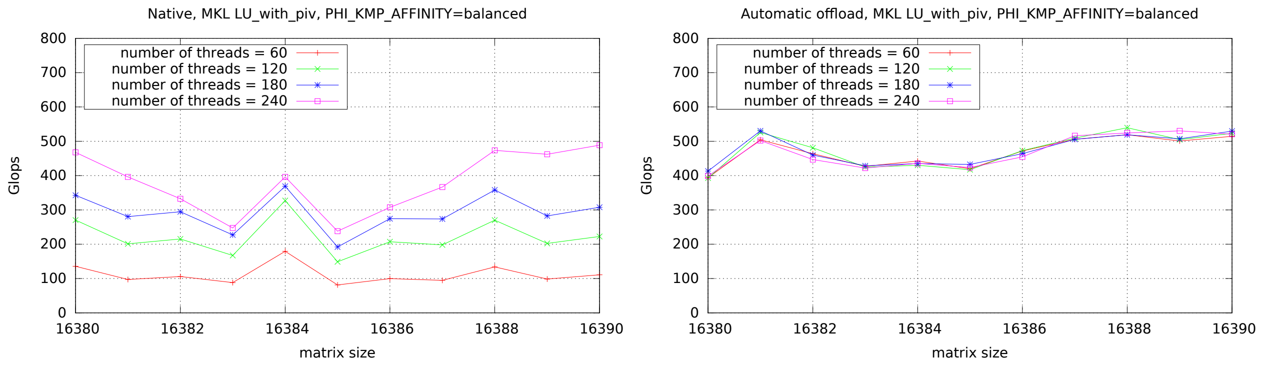


FIG. 7.2. *The performance of the LU factorisation with pivoting (MKL library's implementation) in the native mode (left) and in the automatic offload mode (right) on Intel Xeon Phi — for different matrix sizes, number of the threads, and the* `balanced` *thread mapping setting (cache associativity).*

The LU factorisation with pivoting is the most highly optimised of these three factorizations in native mode.

On the other hand, the LU factorisation without pivoting performs completely differently than three other factorisations — and it is caused by the fact that it works (and was written) completely differently — which is proven by Fig. 6.7.

**7.4. Cache associativity.** To test the influence of the cache associativity on the performance we should investigate the behaviour of the subject algorithms for the matrix of the size around $8192 \times 8192$ — because 8192 double-precision floats occupies 32 kB which is the size of the L1 cache. However, for this size, the algorithms do not utilize all the computing power (they enter the automatic offload mode only for significantly larger matrices). On the other hand, we got some performance drop around the size $16384 \times 16384$ (and 16384 double-precision floats is twice the size of the L1 cache), and that is why we decided to take a look at sizes around this number. In this manner we can investigate the cache associativity.

All the tests for cache associativity were performed only for the `balanced` thread affinity, as it proved the best for tested algorithms.

**7.4.1. LU without pivoting.** Figure 7.1 shows the performance of the LU facorisation without pivoting for the matrix sizes about 16384. We present only the native mode, because — as we see in Figure 6.2 — in the automatic offload mode, there is no efficiency drop around this size. The figure shows that there is a performance drop around 16384 (although the number itself is a weak local maximum). The performance minimum does not have to be precisely at the multiple of cache size, because there are some more auxiliary variables, but it is clearly visible for all the thread mappings.

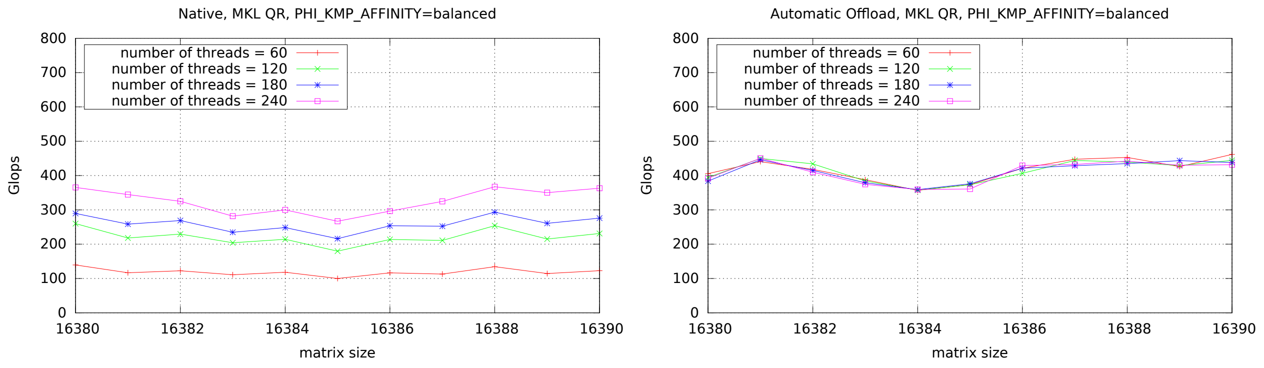


FIG. 7.3. *The performance of the QR factorisation (MKL library's implementation) in the native mode on Intel Xeon Phi (left) and in the automatic offload mode (right) — for different matrix sizes, number of the threads, and the* `balanced` *thread mapping setting (cache associativity).*

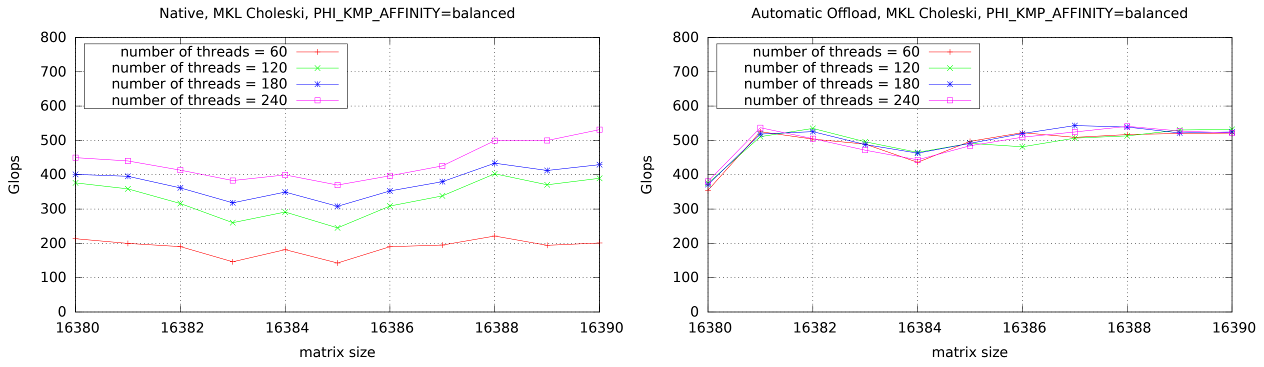


FIG. 7.4. *The performance of the Choleski factorisation (MKL library's implementation) in the native mode on Intel Xeon Phi (left) and in the automatic offload mode (right) — for different matrix sizes, number of the threads, and the* `balanced` *thread mapping setting (cache associativity).*

**7.4.2. LU with pivoting.** Figure 7.2 shows the performance of the LU facorisation with pivoting for the matrix sizes about 16384. However, here we present both the native mode and the automatic offload mode. In the native mode, there is an efficiency drop around 16384, but again, in the 16384 we have a clear although local maximum. On the other hand, in the automatic offload mode we can see almost a flat line around 16384, which is lower than the neighbourhood. Again, all the thread mappings behave similarly.

**7.4.3. QR and Cholesky.** Figures 7.3 and 7.4 show the behaviour of the QR and Cholesky (respectively) factorisations around the size 16384 in both modes (left: native, right: automatic offload). The plots are quite similar to the respective plots for the LU factorisation with pivoting, although the local maximum in 16384 in the native mode is very slight. So, the cache associativity is shown in both modes, independent of the thread mapping.

**8. Conclusion.** The paper reports the effect of thread-mapping for the LU (without and with pivoting), QR and Cholesky factorisations from MKL library on Intel Xeon Phi and the hybrid CPU-MIC platform. Our results showed that there is one thread mapping strategy adapted for all optimised factorisation on Xeon Phi, namely `balanced`. Determining the most efficient OpenMP thread mapping depends highly on the number of thread and it sets the system load. It is surprised that the performance of MKL's `dgetrf` (LU factorisation with pivoting) is much better than MKL's `dgetrfnpi` (LU factorisation without pivoting) on KNC in native mode. This situation indicates that Intel does not optimise `dgetrfnpi` for KNC. However, it should be very easy for them to make optimised `dgetrfnpi`, by just removing the pivoting code from `dgetrf`. In the native mode, the LU with pivoting, QR and Cholesky factorisations are scalable on Intel Xeon Phi but the LU factorisation without pivoting is not. The comparison given here gives good insight into the performance properties of the different factorisation algorithms on Intel Xeon Phi and hybrid CPU-MIC platform. These results can be generalised as the paper gives the performance analysis of some other similar algorithms (namely, the QR and Cholesky factorisations). In future works, the authors plan to research the impact of the thread mapping on the performance and the energy saving for other applications from the domain of the dense linear algebra on shared memory multicore and manycore architectures and to compare it with the results obtained in this work.

## REFERENCES

[1] Anderson E., Bai Z., Bischof C., Blackford S., Demmel J., Dongarra J., Du Croz J., Greenbaum A., Hammarling S., McKenney A., Sorensen D.: LAPACK Users' Guide. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, (1999)

[2] Buttari A., Langou J., Kurzak J., Dongarra J.: A class of parallel tiled linear algebra algorithms for multicore architectures. Parallel Computing, 35(1):38–53, (2009)

[3] Bylina B., Bylina J.: OpenMP Thread Affinity for Matrix Factorization on Multicore Systems, Proceedings of the 2017 Federated Conference on Computer Science and Information Systems, Annals of Computer Science and Information Systems 11, 489–492, (2017)

[4] Demmel J. W.: Applied Numerical Linear Algebra. SIAM, (1997)

[5] Diener M., Cruz E. H. M., Alves M. A. Z., Navaux M. A. Z., Koren I.: Affinity-based thread and data mapping in shared memory systems. ACM Comput. Surv., 49(4):64:1–38, (2016)

[6] Donfack S., Dongarra J., Faverge M., Gates M., Kurzak J., Luszczek P., Yamazaki I.: A Survey of Recent Developments in Parallel Implementations of Gaussian Elimination Concurrency and Computation, Practice and Experience, 27:1292–1309, (2015)

[7] Dongarra J., DuCroz J., Duff I. S., Hammarling S. : A set of level-3 Basic Linear Algebra Subprograms. ACM Trans. Math. Software, **16**: 1–28, (1990)

[8] Dongarra J., Gates M., Haidar A., Jia Y., Kabir K., Luszczek P., Tomov S.: Portable hpc programming on intel many-integrated-core hardware with magma port to xeon phi. In PPAM 2013, Warsaw, Poland, (2013)

[9] Dumas J. G, Gautier T., Pernet C., Roch J. L, Sultan Z.: Recursion based parallelization of exact dense linear algebra routines for Gaussian elimination. Parallel Computing, 57:235–249, (2016)

[10] Eichenberger A. E, Terboven Ch, Wong M., Mey D.: The design of openmp thread affinity. In Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World, IWOMP'12, Berlin, Heidelberg, 15–28 (2012)

[11] Heinecke, et al.: Design and Implementation of the Linpack Benchmark for Single and Multi-node Systems, https://software.intel.com/en-us/articles/design-and-implementation-of-the-linpack-benchmark-for-single-and-multi-node-systems-based, IPDPS 2013 (2013).

[12] Ju T., Zhu Z., Wang Y., Li L., Dong X.T.: Thread Mapping and Parallel Optimization for MIC Heterogeneous Parallel Systems. In: Sun X. et al. (eds) Algorithms and Architectures for Parallel Processing. ICA3PP 2014. Lecture Notes in Computer Science, vol 8631. Springer, 300–311, (2014).

[13] Rahman R.: Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers, 1st edn. Apress, Berkely, CA, USA (2013)

[14] erboven Ch., Mey D., Schmidl D., Jin H., Reichstein T.: Data and thread affinity in openmp programs. In Proceedings of the 2008 workshop on Memory access on future processors: a solved problem? (MAW '08). ACM, New York, NY, USA, 377-384, (2008)

[15] Intel Corporation, Intel Math Kernel Library (MKL), (2017)

[16] https://software.intel.com/en-us/articles/intel-mkl-113-release-notes

*Edited by:* Sasko Ristov
*Received:* Mar 1, 2018
*Accepted:* August 24, 2018

DOI 10.12694/scpe.v19i3.1358
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 19, Number 3, pp. 275–291. `http://www.scpe.org` © 2018 SCPE

# ANALYSIS OF MEMORY FOOTPRINTS OF SPARSE MATRICES PARTITIONED INTO UNIFORMLY-SIZED BLOCKS[*]

D. LANGR[†][‡]AND I. ŠIMEČEK[†]

**Abstract.** The presented study analyses memory footprints of 563 representative benchmark sparse matrices with respect to their partitioning into uniformly-sized blocks. Different block sizes and different ways of storing blocks in memory are considered and statistically evaluated. Memory footprints of partitioned matrices are then compared with their lower bounds and CSR, index-compressed CSR, and EBF storage formats. The results show that block-based storage formats may significantly reduce memory footprints of sparse matrices arising from a wide range of application domains. Additionally, measured consistency of results is presented and discussed, benefits of individual formats for storing blocks are evaluated, and an analysis of best-case and worst-case matrices is provided for in-depth understanding of causes of memory savings of block-based formats.

**Key words:** block, memory footprint, partitioning, sparse matrix, storage format

**AMS subject classification.** 65F50

## 1. Introduction.

The way how sparse matrices are stored in a computer memory may have a significant impact on the required memory space, i.e., on the matrix memory footprints. Reduction of matrix memory footprints may positively influence related computations and executions of corresponding programs [14, 23, 20, 21]. One way of reducing memory footprints of sparse matrices is their partitioning into blocks. Much has been written about block processing of sparse matrices, frequently in the context of memory-bounded character of *sparse matrix-vector multiplication* (SpMV) [3, 4, 5, 6, 7, 8, 9, 10, 13, 16, 17, 18, 19, 24, 25, 27, 28, 30, 29, 31, 32, 33, 34, 35]. In this article, we address the problem of minimizing memory footprints of sparse matrices by their partitioning into uniformly-sized blocks. Its solution raises two essential questions:

1. How to choose a suitable block size?
2. How to store resulting nonzero blocks in a computer memory?

These questions form a multi-dimensional optimization problem that needs to be solved prior to the partitioning itself. We refer to both these problems—optimization and partitioning—as (*block*) *preprocessing*.

The above introduced optimization problem raises another question: How to specify the optimization space, i.e., the space of tested configurations? Intuitively, the larger the optimization space is, the lower matrix memory footprint can be found, however, at a price of longer preprocessing runtime. To amortize block processing of a sparse matrix, the optimization space thus need to be chosen wisely in a form of a trade-off: we want it to be small enough to ensure its fast exploration but also large enough to contain the optimal or nearly-optimal configuration generally for any sparse matrix.

We present a study that analyses memory footprints of 563 representative sparse matrices from the University of Florida Sparse Matrix Collection (UFSMC) [11] with respect to their partitioning into uniformly sized blocks. These matrices arose from a large variety of applications of multiple problem types and thus have highly diverse structural and numerical properties. Our goal is to minimize memory footprints of matrices and we consider an optimization space that consists of different block sizes and different ways of storing blocks in memory. Based on the obtained results, we finally provide suggestions for both efficient and effective block preprocessing of sparse matrices in general.

This article is an extended version of our conference paper [21]. It recapitulates main contributions of the paper and adds new material, which mainly covers following subjects:

1. To assess representativeness of benchmark matrices, we measured the consistency of results across randomly selected subsets of these matrices (see Section 3.5).

[*]This work was supported by the Czech Science Foundation under grant no. 16-16772S and by the Czech Ministry of Education, Youth and Sports under Grant No. CZ.02.1.01/0.0/0.0/16_019/0000765.

[†]Department of Computer Systems, Faculty of Information Technology, Czech Technical University in Prague, Thákurova 9, 160 00, Praha, Czech Republic (`daniel.langr@fit.cvut.cz`).

[‡]Výzkumný a zkušební letecký ústav, a.s., Beranových 130, 199 05, Praha, Czech Republic.

2. We investigated influence of individual storage formats to the results and show that there is practically no reason to use the CSR storage format for storing blocks (see Section 3.6).
3. Originally, memory footprints of matrices were compared with their lower bounds and with 32-bit indexed CSR storage format (CSR32), which is the most commonly used format in practice. However, in our block-based approach, we assume index compression. It makes therefore sense to compare memory footprints of matrices also with index-compressed implementation of CSR (see Section 3.8).
4. We show that the lower bound for matrix memory footprint is the amount of memory required to store the values of matrix nonzero elements [25, 21]. However, in practice, we can approach this lower bound only if we make some assumptions about a matrix structure. Without such assumptions, i.e., in generic cases, we can define another lower bound for memory footprints of sparse matrices that stems from compression of data about matrix nonzero structure. This concept is used by so-called *entropy-based format* (EBF) and within this article, we present memory footprints of matrices compared to EBF as well (see Section 3.9).
5. We provide statistical analysis of measured memory footprints with respect to CSR32 as a function of some of their characteristics, namely their application domain, density of nonzero elements, and deviation of number of nonzero elements per rows (see Section 3.7).
6. We provide in-depth analysis of best-case and worse-case matrices, which are matrices most suitable and unsuitable for block processing, respectively. This analysis allows better understanding of our block-based concepts and may clarify some of their aspects (see Section 3.11).

Some parts of the original paper were shortened or omitted.

**2. Methodology.** In Section 1, we referred to a *matrix memory footprint* as to an amount of memory space required to store a given matrix in a computer memory. More precisely, we can define it as a number of bits (or bytes) that is needed to store the values of the nonzero elements of a given matrix together with the information about their structure, i.e., their row and column positions. The ways how sparse matrices are stored in a computer memory are generally called *sparse matrix storage formats*; we call them *formats* only if the context is clear. Matrix memory footprint is thus a function of a given matrix and a used format (memory footprints for the same matrix but distinct formats may differ considerably).

**2.1. Block Storage Schemes.** In case of partitioned sparse matrices, their nonzero blocks represent individual submatrices that can be treated separately. In practice, well-proven formats used for nonzero blocks of sparse matrices are:

- The *coordinate* (COO) format, which stores values of block nonzero elements together with their row and column indices [6, 24, 30].
- The *compressed sparse row* (CSR) format, which stores values and column indices of lexicographically ordered block nonzero elements together with the information about which values / column indices belongs to which block row [24, 27, 28, 30].
- The *bitmap* format, which stores values of block nonzero elements in some prescribed order and encodes their row and column indices in a bit array [7, 18, 24].
- The *dense* format, which stores values of both nonzero and zero block elements in a dense array (row and column indices of nonzero elements are thus effectively determined by positions of their values within this array) [2, 16, 17, 24].

Considering these formats, we have 6 options how to store nonzero blocks of a sparse matrix in memory:

1. store all the blocks in the COO format,
2. store all the blocks in the CSR format,
3. store all the blocks in the bitmap format,
4. store all the blocks in the dense format,
5. store *all the blocks* in the same format such that the format minimizes the memory footprint of a given matrix (we refer to this option as *min-fixed*),
6. store *each block* generally in a different format such that the format minimizes the contribution of this block to the memory footprint of a given matrix (we refer to this option as *adaptive*).

We call these options *block storage schemes*, or shortly *schemes* only. Since the first 4 schemes prescribe a fixed format for all the blocks, we call them *fixed-format schemes*.

For the min-fixed and adaptive schemes, we consider formats for nonzero blocks to be chosen from COO, CSR, bitmap, and dense. In case of the min-fixed scheme, the matrix memory footprint thus contains 2 additional bits for storing the information about the format used for all the nonzero blocks. In case of the adaptive scheme, the matrix memory footprint contains 2 additional bits for each nonzero block to store the information about its format.

**2.2. Block Sizes.** To evaluate memory footprints of a given matrix for different schemes and some particular tested block size, we need information about numbers of nonzero elements of all nonzero blocks [24]. In the end, this information must be obtained for each distinct block size from the optimization space, which represents the most demanding part of the whole optimization process [22]. The block preprocessing runtime is thus approximately proportional to the number of distinct tested block sizes. Consequently, the lower is their count, the higher are the chances that the partitioning will be profitable at all.

Generally, there is $O(m\times n)$ ways how to set a block size for an $m\times n$ matrix, but for fast block preprocessing, we need to choose only few of them.[1] One possible approach is to consider only block sizes

$$2^k \times 2^\ell, \quad \text{where} \quad 1 \le k \le K \quad \text{and} \quad 1 \le \ell \le L, \tag{2.1}$$

which reduces the number of tested block sizes to $K \times L$. The rationale behind such a choice consists, e.g., of much faster block preprocessing, full employment of bits for storing in-block row and column indices, higher utilization of caches, and possible storage of block elements in *Z-Morton order* [26] (for detailed explanation of these aspects, see [21]).

Within the presented study, we consider block sizes (2.1) and set $K = L = 8$. The choice of these upper bounds stemmed from our auxiliary experiments which showed that space-optimal block sizes have mostly less than 64 rows/columns. Taking into account block sizes with up to 256 rows/columns should cover even the remaining corner cases.

Consequently, each $m \times n$ matrix is further treated as a set of block matrices of sizes $\lceil m/2^k \rceil \times \lceil n/2^\ell \rceil$, where $1 \le k, \ell \le 8$.

**2.3. Optimization Space.** In the summary, our optimization space is initially defined by $\mathcal{S}_6 \times \mathcal{B}_{64}$, where $\mathcal{S}_6$ denotes a set of selected block storage schemes:

$$\mathcal{S}_6 = \big\{\text{COO}, \text{CSR}, \text{bitmap}, \text{dense}, \text{min-fixed}, \text{adaptive}\big\} \tag{2.2}$$

and $\mathcal{B}_{64}$ denotes a set of selected block sizes:

$$\mathcal{B}_{64} = \big\{2^k \times 2^\ell : 1 \le k, \ell \le 8\big\}. \tag{2.3}$$

**2.4. Additional Considerations.** Additionally, when measuring matrix memory footprints, we need to decide how to represent information about nonzero blocks and how to represent indices. In the presented study, we assume:

1. nonzero blocks stored in memory in the lexicographical order;
2. explicit storage of block column index for each nonzero block;
3. storage of the number of nonzero blocks for each block row;
4. a minimum possible number of bits, i.e., $\lceil \log_2 n \rceil$ bits, to store an index related to $n$ entities (such an approach is in the literature sometimes referred to as *index compression*).

**2.5. Benchmark Matrices.** Sparse matrices are often divided into two main categories—*high performance computing (HPC) matrices* and *graph matrices*, the latter being binary matrices for unweighted graphs. Efficient processing of graph matrices is generally governed by special rules that are different from those being effective for HPC matrices [1, 8, 36] (e.g., higher matrix memory footprints in some cases lead to higher performance of computations and graph matrices are also typically not suitable for simple block processing mainly due

[1]In addition to multiplication and Cartesian product, we also use the multiplication sign "×" to specify matrix/block sizes. In such cases, $m \times n$ does not denote multiplication, but a matrix/block size of height $m$ and width $n$ (i.e., having $m$ rows ans $n$ columns).

Table 2.1

*Counts of tested matrices falling under particular problem types (referred to as "kinds" in the UFSMC).*

| Problem | Matrices | Problem | Matrices |
|---|---|---|---|
| 2D/3D | 36 | least squares | 7 |
| acoustics | 4 | linear programming | 51 |
| chemical process simulation | 25 | materials | 15 |
| circuit simulationi | 41 | model reduction | 11 |
| computational fluid dynamics | 47 | optimization | 66 |
| computer graphics/vision | 8 | power network | 35 |
| counter-example | 2 | semiconductor device | 16 |
| duplicate model reduction | 5 | statistical/mathematical | 1 |
| economic | 24 | structural | 82 |
| eigenvalue/model reduction | 2 | theoretical/quantum chem. | 42 |
| electromagnetics | 11 | thermal | 11 |
| frequency-domain circuit sim. | 4 | weighted graph | 17 |

to emergence of hypersparse blocks [7, 8]). Within this work, we focused mainly (but not exclusively) on HPC matrices. Particularly, for experiments, we took real matrices from the UFSMC that contained more than $10^5$ nonzero elements and that exhibited a unique structure of nonzero elements.[2] This way, we obtained 563 sparse matrices arising from different application problems (see Table 2.1) and thus having different structural (and numerical) properties; we denote these matrices by $A_1, \ldots, A_{563}$. Of these matrices, 281 were square symmetric and the remaining 282 were either rectangular or square unsymmetric.

For symmetric matrices, we always assume storage only of their single triangular parts in memory, which is a common practice. When referring to the *number of nonzero elements* of a matrix, we thus generally need to distinguish between the number of *all* nonzero elements and the number of elements that are assumed to be *stored* in a computer memory. While measuring memory footprints of sparse matrices, we take into account the latter one.

**2.6. Matrix Memory Footprint.** According to the text above, a matrix memory footprint for a sparse matrix $A_k$ partitioned into uniformly-sized blocks is a function of the following parameters:

1. a sparse matrix itself ($A_k$),
2. a block storage scheme $s \in \mathcal{S}_6$,
3. a block size $h \times w \in \mathcal{B}_{64}$,
4. a number of bits $b$ required to store a value of a single matrix nonzero element.

We denote this function by $\mathrm{MMF}_{\boxplus}(A_k, s, w \times h, b)$. We further assume storing values of matrix nonzero elements in either single or double precision IEEE floating-point format [15], which implies $b = 32$ or $b = 64$, respectively, in case of real matrices. We refer to such a floating-point precision as *precision* only.

We say that a matrix memory footprint for a given matrix $A$ and a given precision determined by $b$ is *optimal* (with respect to our work) if it equals

$$\min\bigl\{\mathrm{MMF}_{\boxplus}(A, s, h \times w, b) : s \in \mathcal{S}_6, h \times w \in \mathcal{B}_{64}\bigr\}. \tag{2.4}$$

We call the corresponding block storage scheme and block size optimal as well.

**2.7. Optimization Subspaces.** Let $\mathcal{S} \subseteq \mathcal{S}_6$ and $\mathcal{B} \subseteq \mathcal{B}_{64}$. $\mathcal{S} \times \mathcal{B}$ thus define a subspace of the optimization space $\mathcal{S}_6 \times \mathcal{B}_{64}$. Let

$$\Delta^b_{\mathcal{S},\mathcal{B}}(k) = \left( \frac{\min\bigl\{\mathrm{MMF}_{\boxplus}(A_k, s, h \times w, b) : s \in \mathcal{S}, h \times w \in \mathcal{B}\bigr\}}{\min\bigl\{\mathrm{MMF}_{\boxplus}(A_k, s, h \times w, b) : s \in \mathcal{S}_6, h \times w \in \mathcal{B}_{64}\bigr\}} - 1 \right) \times 100. \tag{2.5}$$

[2]As for April, 2016.

Table 3.1
*Minimum, average and maximum values of $\mathcal{U}^b_{s,\mathcal{B}_{64}}$ (in percents).*

| | Single precision ($b = 32$) | | | Double precision ($b = 64$) | | |
|---|---|---|---|---|---|---|
| Scheme ($s$) | Minimum | Average | Maximum | Minimum | Average | Maximum |
| COO | 0.00 | 4.78 | 15.27 | 0.00 | 2.52 | 7.67 |
| CSR | 0.73 | 6.84 | 19.13 | 0.41 | 3.74 | 11.05 |
| bitmap | 0.00 | 3.13 | 22.01 | 0.00 | 1.75 | 12.38 |
| dense | 0.00 | 84.61 | 217.04 | 0.00 | 92.40 | 249.02 |
| min-fixed | 0.00 | 1.19 | 5.41 | 0.00 | 0.64 | 2.94 |
| adaptive | 0.00 | 0.10 | 2.24 | 0.00 | 0.05 | 1.30 |

This function expresses of how much percent is the minimal memory footprint of $A_k$ from $\mathcal{S} \times \mathcal{B}$ higher (worse) than its optimal memory footprint. To assess the subspace $\mathcal{S} \times \mathcal{B}$, we define the following parametrized set

$$\mathcal{U}^b_{\mathcal{S},\mathcal{B}} = \left\{ \Delta^b_{\mathcal{S},\mathcal{B}}(k) : 1 \leq k \leq 563 \right\}. \tag{2.6}$$

The minimum, mean (average; $\mu$), and maximum of $\mathcal{U}^b_{\mathcal{S},\mathcal{B}}$ then reflect the best, average, and worst cases, respectively, for $\mathcal{S} \times \mathcal{B}$ across the tested matrices. If $\mathcal{S}$ or $\mathcal{B}$ consists of a single element only, we omit the curly braces in the subscript of $\mathcal{U}$ for the sake of readability; e.g., we write $\mathcal{U}^b_{s,\mathcal{B}_{64}}$ and $\mathcal{U}^b_{\mathcal{S}_6,h\times w}$ instead of $\mathcal{U}^b_{\{s\},\mathcal{B}_{64}}$ and $\mathcal{U}^b_{\mathcal{S}_6,\{h\times w\}}$.

## 3. Results and Discussion.

**3.1. Block Storage Schemes.** First, we assessed block storage schemes. Complete statistics of $\mathcal{U}^b_{s,\mathcal{B}_{64}}$ are presented in Table 3.1 and lead to the following observations:

- No fixed-format scheme minimized matrix memory footprints in comparison with the others. Bitmap was the best in average, however, it was inferior to both COO and CSR in worst cases.
- Dense provided extremely high matrix memory footprints in average and worst cases. Due to the explicit storage of zero elements, this scheme is suitable only for kinds of matrices that contain highly dense blocks; obviously, there were only few such matrices in our tested suite.
- The lowest memory footprints were provided by the min-fixed and adaptive schemes; their numbers are considerably lower in comparison with the fixed-format schemes.

**3.2. Block Sizes.** Similarly as block storage schemes, we assessed block sizes. Figure 3.1 shows for how many tested matrices were individual block sizes optimal in case of double precision measurements; for single precision, the results differed only for 2 matrices. We may observe that some block sizes were especially favourable. The $8 \times 8$ block size was optimal for 257 matrices, which corresponds to 45.6% of their total count. Together with $4 \times 4$ and $16 \times 16$, these 3 block sizes were optimal for 65.2% of tested matrices. However, the numbers from Figure 3.1 reflect only best cases. To find out how much were particular block sizes better than the others in average and for their worst-cases matrices, we present the average and maximum values of $\mathcal{U}^b_{\mathcal{S}_6,h\times w}$ in Tables 3.2 and 3.3 for single and double precision, respectively. According to these results, some blocks sizes—especially $8 \times 8$—provided alone average matrix memory footprints close to their optimal values. However, there was not a single block size that would yield the same outcome for all the tested matrices; the maxima were for all the block sizes relatively high.

**3.3. Subsets of Block Sizes.** Let us remind that one of our goals is a possible reduction of the number of block sizes in the optimization test space. The question thus is whether there is some subset $\mathcal{B} \subset \mathcal{B}_{64}$ that would, at the same time:

1. significantly reduce the number of block sizes ($|\mathcal{B}|$),
2. provide matrix memory footprints close to their optimal values for most of the tested matrices (average of $\mathcal{U}^b_{\mathcal{S}_6,B}$ close to zero),
3. provide low matrix memory footprints for all the tested matrices (low maximum of $\mathcal{U}^b_{\mathcal{S}_6,B}$).

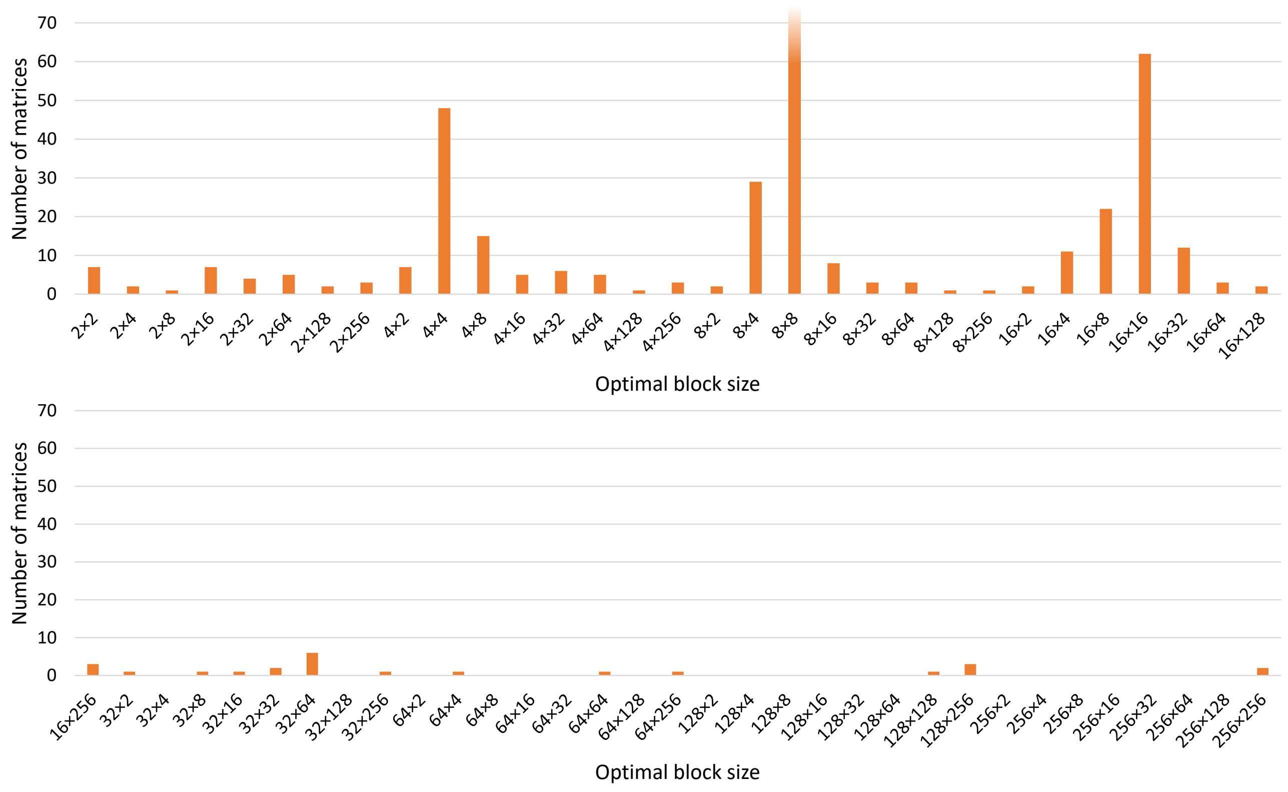


FIG. 3.1. *Numbers of tested matrices for which are block sizes optimal, measured for double precision; block size $8 \times 8$ was optimal for 257 matrices.*

TABLE 3.2
*Average and maximum values of $\mathcal{U}^{32}_{\mathcal{S}_6,h\times w}$ (in percents), sorted by average.*

| Rank | $h \times w$ | Avg. | Max. | Rank | $h \times w$ | Avg. | Max. |
|---|---|---|---|---|---|---|---|
| 1 | 8×8 | 1.23 | 18.36 | 11 | 16×32 | 4.03 | 23.75 |
| 2 | 8×16 | 2.14 | 19.35 | 12 | 32×8 | 4.13 | 23.97 |
| 3 | 16×8 | 2.26 | 21.41 | 13 | 4×32 | 4.36 | 18.71 |
| 4 | 4×8 | 2.32 | 17.31 | 14 | 32×16 | 4.53 | 24.45 |
| 5 | 8×4 | 2.38 | 19.52 | 15 | 32×4 | 4.87 | 23.60 |
| 6 | 16×16 | 2.56 | 21.82 | 16 | 32×32 | 5.20 | 26.50 |
| 7 | 4×4 | 2.92 | 21.94 | ... | ... | ... | ... |
| 8 | 4×16 | 2.99 | 16.51 | 62 | 256×2 | 14.44 | 37.33 |
| 9 | 16×4 | 3.23 | 20.44 | 63 | 256×128 | 14.61 | 38.32 |
| 10 | 8×32 | 3.65 | 21.26 | 64 | 256×256 | 14.65 | 35.42 |

Based on the analysis presented in detail in [21], we propose the following *reduced sets of block sizes*:

$$\mathcal{B}_8 = \{2^k \times 2^k : 1 \leq k \leq 8\}, \tag{3.1}$$

$$\mathcal{B}_{14} = \mathcal{B}_8 \cup \{2^k \times 2^\ell : 2 \leq k, \ell \leq 4\}, \tag{3.2}$$

$$\mathcal{B}_{20} = \mathcal{B}_8 \cup \{2^k \times 2^\ell : 2 \leq k, \ell \leq 5\}. \tag{3.3}$$

A subscript $i$ in $\mathcal{B}_i$ expresses the number of its block sizes, i.e., $|\mathcal{B}_i| = i$.

Table 3.3
*Average and maximum values of* $\mathcal{U}^{64}_{\mathcal{S}_6,h\times w}$ *(in percents), sorted by average.*

| Rank | $h \times w$ | Avg. | Max. | Rank | $h \times w$ | Avg. | Max. |
|---|---|---|---|---|---|---|---|
| 1 | 8×8 | 0.69 | 11.07 | 11 | 16×32 | 2.19 | 12.84 |
| 2 | 8×16 | 1.18 | 11.67 | 12 | 32×8 | 2.26 | 14.45 |
| 3 | 16×8 | 1.25 | 12.91 | 13 | 4×32 | 2.40 | 10.56 |
| 4 | 4×8 | 1.30 | 9.74 | 14 | 32×16 | 2.47 | 14.04 |
| 5 | 8×4 | 1.33 | 10.98 | 15 | 32×4 | 2.68 | 14.23 |
| 6 | 16×16 | 1.40 | 13.16 | 16 | 32×32 | 2.82 | 14.18 |
| 7 | 4×4 | 1.63 | 12.34 | ... | ... | ... | ... |
| 8 | 4×16 | 1.66 | 9.96 | 62 | 256×2 | 7.88 | 21.59 |
| 9 | 16×4 | 1.79 | 12.32 | 63 | 256×128 | 7.92 | 19.56 |
| 10 | 8×32 | 1.99 | 11.97 | 64 | 256×256 | 7.93 | 18.96 |

Table 3.4
*Average and maximum values of* $\mathcal{U}^{b}_{s,\mathcal{B}_j}$ *(in percents) for* $j \in \{64, 20, 14, 8\}$.

(a) Single precision ($b = 32$)

| | $s =$ min-fixed | | $s =$ adaptive | |
|---|---|---|---|---|
| Block sizes | Average | Maximum | Average | Maximum |
| $\mathcal{B}_{64}$ | 1.19 | 5.41 | 0.10 | 2.24 |
| $\mathcal{B}_{20}$ | 1.32 | 6.23 | 0.22 | 4.21 |
| $\mathcal{B}_{14}$ | 1.35 | 6.89 | 0.28 | 6.81 |
| $\mathcal{B}_{8}$ | 1.51 | 10.06 | 0.51 | 11.07 |

(b) Double precision ($b = 64$)

| | $s =$ min-fixed | | $s =$ adaptive | |
|---|---|---|---|---|
| Block sizes | Average | Maximum | Average | Maximum |
| $\mathcal{B}_{64}$ | 0.64 | 2.94 | 0.05 | 1.30 |
| $\mathcal{B}_{20}$ | 0.71 | 3.52 | 0.12 | 2.37 |
| $\mathcal{B}_{14}$ | 0.73 | 3.77 | 0.16 | 3.83 |
| $\mathcal{B}_{8}$ | 0.81 | 5.34 | 0.28 | 5.88 |

**3.4. Reduced Optimization Subspaces.** Table 3.1 revealed that to minimize memory footprints of (all) the tested matrices, we had to use either the min-fixed or the adaptive block storage scheme. To reduce the block preprocessing overhead, we now proposed several reduced sets of block sizes. Let us now assess these options together. We measured the statistics of $\mathcal{U}^{b}_{s,\mathcal{B}_j}$ for all the combinations of $s \in \{\text{min-fixed}, \text{adaptive}\}$ and $j \in \{64, 20, 14, 8\}$; the results are presented in Table 3.4. The average matrix memory footprints were in all cases close to their optimal values.

**3.5. Consistency.** Up to now, we have presented measurements conducted for all 563 tested matrices. To assess their "representativeness", we measured the consistency of memory footprints statistics across randomly selected subsets of these matrices. Such an experiment should reveal to which extent are our measurements sensitive to the set of input matrices.

Let $\mathcal{R}^{(i)}_n$ denote an $i$th set of $n$ randomly selected tested matrices; different $i$ thus allows us to distinguish different random selections. Let $\mathcal{K}^{(i)}_n$ denote a set of matrix indices from $\mathcal{R}^{(i)}_n$, thus $\mathcal{R}^{(i)}_n = \{A_k : k \in \mathcal{K}^{(i)}_n\}$. Let

$$\mathcal{V}^{b,(i)}_{s,\mathcal{B}_j,n} = \left\{ \Delta^{b}_{s,\mathcal{B}_j}(k) : k \in \mathcal{K}^{(i)}_n \right\}. \tag{3.4}$$

TABLE 3.5
*Standard deviations of* $\operatorname{avg} \mathcal{V}^{b,(i)}_{s,\mathcal{B}_j,200}$ *and* $\max \mathcal{V}^{b,(i)}_{s,\mathcal{B}_j,200}$ *(in percents) for* $1 \leq i \leq 50$.

(a) Single precision ($b = 32$)

| | $s =$ min-fixed | | $s =$ adaptive | |
|---|---|---|---|---|
| Block sizes | Average | Maximum | Average | Maximum |
| $\mathcal{B}_{64}$ | 0.06 | 0.33 | 0.02 | 0.24 |
| $\mathcal{B}_{20}$ | 0.07 | 0.37 | 0.03 | 0.64 |
| $\mathcal{B}_{14}$ | 0.08 | 0.42 | 0.04 | 1.41 |
| $\mathcal{B}_{8}$ | 0.09 | 0.86 | 0.07 | 1.40 |

(b) Double precision ($b = 64$)

| | $s =$ min-fixed | | $s =$ adaptive | |
|---|---|---|---|---|
| Block sizes | Average | Maximum | Average | Maximum |
| $\mathcal{B}_{64}$ | 0.03 | 0.19 | 0.01 | 0.14 |
| $\mathcal{B}_{20}$ | 0.04 | 0.23 | 0.02 | 0.29 |
| $\mathcal{B}_{14}$ | 0.04 | 0.22 | 0.02 | 0.82 |
| $\mathcal{B}_{8}$ | 0.05 | 0.50 | 0.04 | 0.61 |

$\mathcal{V}^{b,(i)}_{s,\mathcal{B}_j,n}$ thus expresses of how much percents are memory footprints of matrices from $\mathcal{R}^{(i)}_n$—measured for scheme $s$, a set of block sizes $\mathcal{B}_j$, and a precision given by $b$—higher than their optimal memory footprints. Similarly as before, we were interested in average and maximum values of $\mathcal{V}^{b,(i)}_{s,\mathcal{B}_j,n}$; let them denote by $\operatorname{avg} \mathcal{V}^{b,(i)}_{s,\mathcal{B}_j,n}$ and $\max \mathcal{V}^{b,(i)}_{s,\mathcal{B}_j,n}$, respectively. To assess the consistency introduced above, we measured standard deviations of these metrics for 50 sets of 200 randomly selected tested matrices, i.e., standard deviations of the following sets:

$$\left\{\operatorname{avg} \mathcal{V}^{b,(i)}_{s,\mathcal{B}_j,200} : 1 \leq i \leq 50\right\} \quad \text{and} \quad \left\{\max \mathcal{V}^{b,(i)}_{s,\mathcal{B}_j,200} : 1 \leq i \leq 50\right\}. \tag{3.5}$$

The results obtained for the min-fixed and adaptive schemes, sets of blocks sizes $\mathcal{B}_{64}, \mathcal{B}_{20}, \mathcal{B}_{14}, \mathcal{B}_8$, and both precisions are shown in Table 3.5.

The measured standard deviations are of 1 to 2 orders of magnitude lower than the corresponding numbers from Table 3.4. By normalizing the standard deviations (with respect to Table 3.4), we found out that the standard deviations ranged from 5.16 to 9.28 percents for the min-fixed scheme and from 10.30 to 21.33 percents for the adaptive scheme. Seemingly, the min-fixed scheme provides more consistent relative memory footprints of matrices with respect to their optimal values, while the adaptive scheme is more sensitive to the selection of matrices. Note, however, that the measured standard deviations were according to Table 3.5 in all cases relatively small with the maximum value 1.41; recall that these numbers are relative differences in percents between optimal matrix memory footprints and those measured for particular tested configurations. Especially, the standard deviations for average metrics are practically negligible, which manifests high level of representativeness of the tested matrices.

**3.6. Block Storage Schemes Without CSR.** We have defined the min-fixed and adaptive block storage schemes such that the format used for storing blocks is selected—from COO, CSR, bitmap, and dense—either for all blocks collectively or for each block separately; the corresponding results were presented by Table 3.4. However, we were also interested in how these results would change if we modified the min-fixed and adaptive schemes by excluding individual formats. We carried out such measurements and their results revealed that:

1. without the COO or bitmap format, the memory footprints of matrices grew significantly;
2. without the CSR or dense formats, the memory footprints of matrices grew negligibly;
3. without both the CSR and dense formats, the memory footprints of matrices grew negligibly as well.

The question therefore is whether the CSR and dense formats are at all useful for storing blocks. Based on our knowledge and experience, we would not suggest to exclude the dense format. Though this format is optimal

Table 3.6
*Average and maximum values of $\mathcal{U}^b_{s,\mathcal{B}_k}$ (in percents) for $k \in \{64, 20, 14, 8\}$ with excluded CSR.*

(a) Single precision ($b = 32$)

| | $s$ = min-fixed-w/o-CSR | | $s$ = adaptive-w/o-CSR | |
|---|---|---|---|---|
| Block sizes | Average | Maximum | Average | Maximum |
| $\mathcal{B}_{64}$ | 1.20 | 5.55 | 0.15 | 2.24 |
| $\mathcal{B}_{20}$ | 1.32 | 6.44 | 0.26 | 4.22 |
| $\mathcal{B}_{14}$ | 1.35 | 6.89 | 0.31 | 6.82 |
| $\mathcal{B}_{8}$ | 1.51 | 10.06 | 0.54 | 11.07 |

(b) Double precision ($b = 64$)

| | $s$ = min-fixed-w/o-CSR | | $s$ = adaptive-w/o-CSR | |
|---|---|---|---|---|
| Block sizes | Average | Maximum | Average | Maximum |
| $\mathcal{B}_{64}$ | 0.65 | 3.01 | 0.09 | 1.30 |
| $\mathcal{B}_{20}$ | 0.71 | 3.52 | 0.15 | 2.37 |
| $\mathcal{B}_{14}$ | 0.73 | 3.77 | 0.18 | 3.84 |
| $\mathcal{B}_{8}$ | 0.82 | 5.34 | 0.30 | 5.88 |

in rare cases only, it is likely the most efficient format for matrix computations. For example, multiplication of a block stored in the dense format with a corresponding vector part can be performed by invoking a relevant operation from some dense linear algebra library, such as BLAS [12]. In practice, every HPC system provides at least one optimized implementation of such a library that is highly-tuned for a given hardware architecture (e.g., ATLAS, BLIS, Cray LibSci, IBM ESSL, Intel MKL, OpenBLAS, etc.).

On the contrary, CSR does not provide the same benefits as the dense format, especially when it is implemented together with index compression. Moreover, CSR is the only considered format that prescribes a fixed order of nonzero elements; consequently, it does not allow to store them in an order that might be computationally more efficient, such as the Z-Morton order. One therefore might consider excluding CSR from the min-fixed and adaptive schemes to simplify related algorithms and their implementations. We call such modified schemes *min-fixed-w/o-CSR* and *adaptive-w/o-CSR* and present the results for them in Table 3.6. Obviously, the numbers are either the same or only slightly higher than those measured for the original min-fixed and adaptive schemes; see Table 3.4.

**3.7. Memory Savings Against CSR32.** Likely the most widely-used storage format for sparse matrices in practice is CSR, which is supported by vast majority of software tools and libraries that work with sparse matrices. To distinguish between CSR used for blocks of partitioned matrices and CSR used for whole (not-partitioned) matrices, we call the latter CSR32, since it is typically implemented with 32-bit indices. Researchers frequently demonstrate the superiority of their algorithms and data structures (formats) by comparison with CSR32, which have become de facto an etalon in sparse-matrix research [25].

Comparison of memory footprints of sparse matrices partitioned into blocks and the same matrices stored in CSR32 allows us to assess our block approach. Let $\mathrm{MMF}_{\mathrm{CSR32}}(A, b)$ denote a memory footprint of a matrix $A$ stored in memory in CSR32 with respect to a precision given by $b$. The function

$$\Lambda^b(k) = \left(1 - \frac{\min\{\mathrm{MMF}_{\boxplus}(A_k, s, h \times w, b) : s \in \mathcal{S}_6, h \times w \in \mathcal{B}_{64}\}}{\mathrm{MMF}_{\mathrm{CSR32}}(A_k, b)}\right) \times 100 \tag{3.6}$$

then expresses how much memory in percents we would save if we stored the tested matrix $A_k$ in its optimal block configuration instead of in CSR32. We measured these memory savings for all the tested matrices and processed them statistically; the results are presented by Table 3.7. The obtained numbers speaks strongly in favour of partitioning of sparse matrices in general. Even in worst cases, our block approach reduced the memory footprints of matrices of 25.46% and 17.08% for single and double precision, respectively. In average,

TABLE 3.7
*Statistics of $\Lambda^b(k)$, i.e., memory savings of optimal block configurations against CSR32 in percents, across the tested matrices.*

| Statistics | Single precision | Double precision |
|---|---|---|
| Minimum | 25.46 | 17.08 |
| Average | 42.29 | 28.67 |
| Maximum | 50.21 | 35.86 |

the savings were 42.29% and 28.67%, which significantly reduces the amount of data that needs to be transferred between memory and processors during computations.

Table 3.7 shows the statistics of memory savings across all the tested matrices. However, we also wanted to find out which matrices were especially suitable/unsuitable for partitioning in general. For this reason, we measured the memory saving against CSR32 also as a function the following criteria, which are commonly used to distinguish/quantify different types of sparse matrices:

1. application problem type,
2. relative count of matrix nonzero elements (their density),
3. uniformity of the distribution of matrix nonzero elements across its rows.

The application problem types were introduced by Table 2.1. As for the second criterion, we define the *density* of nonzero elements for an $m \times n$ matrix $A$ with $nnz$ nonzero elements in percents as $\varrho(A) = nnz/(m \times n) \times 100$. Its values thus ranges from 0 for an empty matrix to 100 to a fully dense matrix.

Let $rnnz(i)$ denote a number of nonzero elements of $i$th row of $A$; $rnnz(i)$ thus ranges from 0 for empty rows to $n$ for fully dense rows. To allow a collective evaluation of matrices with different row lengths, we transform $rnnz(i)$ into relative counts in percents as follows: $prnnz(i) = rnnz(i)/n \times 100$. The standard deviation of $prnnz(i)$ for $i = 1, \ldots, m$ then represents an inverse measure of the above introduced third criterion for $A$. Zero standard deviation of $prnnz(i)$ then implies a matrix whose all rows have exactly the same number of nonzero elements.

Recall that in Section 2 we defined two kinds of the numbers of nonzero elements, counting either all of them or just those stored in a computer memory (for unsymmetric matrices, these numbers would be equal). Accordingly, we can quantify the above introduced second and third matrix criteria in two ways; we further show results for both of them.

The measurements for the first criterion and double precision are presented by Table 3.2; the results for single precision are practically the same, just scaled accordingly. We need to be careful when making general conclusions based on these results, since for some problem types, our tested suite of matrices contain only few representatives. However, we may observe that the memory savings against CSR32 were relatively consistent across problem types; there was no problem type that would provide much better or much worse savings than the others, including even the graph matrices.

The measurements for the second and third criteria are presented by the top and bottom parts of Table 3.3, respectively. Again, we show results only for double precision for the same reason as above. Seemingly (and maybe interestingly), there is no obvious correlation between the memory savings of partitioned matrices against CSR32 and the density of nonzero elements of matrices / uniformity of their distribution across matrix rows.

In summary, the obtained results support the potential profitability of partitioning of sparse matrices in general.

**3.8. Memory Savings Against Index-Compressed CSR.** Recall that within our block approach we assumed index compression, i.e., a minimum amount of bits to be used for all indices (Table 2.4). However, for CSR32, we considered all indices to occupy 32 bits each, since this is the most common implementation of CSR in practice. To provide fair comparison with CSR, we therefore also considered storage of matrices in the implementation of CSR with indexed compressed indices; we call such a variant CSRic.

Let $\mathrm{MMF}_{\mathrm{CSRic}}(A, b)$ denote a memory footprint of a matrix $A$ stored in memory in CSRic with respect to

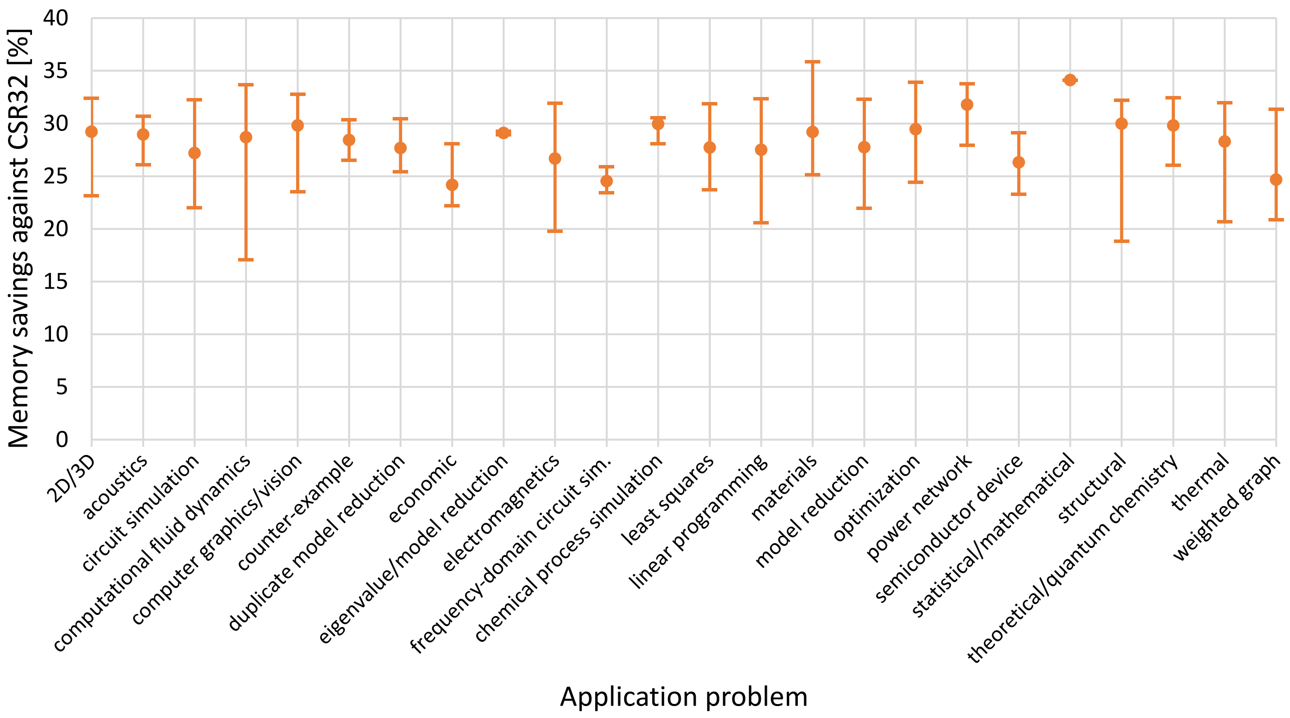


Fig. 3.2. *Statistics of relative memory savings against CSR32 in percents across the tested matrices grouped by individual problem types, measured for double precision. Circles represent average values, the extents from minimal to maximal values are indicated by bars.*

Table 3.8
*Statistics of $\Omega^b(k)$, i.e., memory savings of optimal block configurations against CSRic in percents, across the tested matrices.*

| Statistics | Single precision | Double precision |
|---|---|---|
| Minimum | 1.41 | 0.84 |
| Average | 22.43 | 13.70 |
| Maximum | 35.63 | 22.43 |

a precision given by $b$. The function

$$\Omega^b(k) = \left(1 - \frac{\min\bigl\{\mathrm{MMF}_{\boxplus}(A_k, s, h \times w, b) : s \in \mathcal{S}_6, h \times w \in \mathcal{B}_{64}\bigr\}}{\mathrm{MMF}_{\mathrm{CSRic}}(A_k, b)}\right) \times 100 \tag{3.7}$$

then expresses how much memory in percents we would save if we stored the tested matrix $A_k$ in its optimal block configuration instead of in CSRic. We measured these memory savings for all the tested matrices and processed them statistically; the results are presented by Table 3.8. Memory footprints of matrices stored in CSRic are either the same or more likely lower than memory footprints of matrices stored in CSR32. Therefore, memory savings $\Omega^b(k)$ are lower than $\Lambda^b(k)$. However, even when compared to CSRic, our block approach still reduces memory footprints of matrices in average by 22.43% and 13.70% for single and double precision, respectively, which represents significant memory savings.

**3.9. Memory Savings Against EBF.** If the structure of a sparse matrix is completely known (such as in case of tridiagonal matrices), the amount of memory required to store the information about its structure is effectively zero. Many sparse matrix storage formats are based on assumptions that matrices (more or less) match some particular structure of nonzero elements. For instance, block-based formats work best for matrices, that has nonzero elements clustered in dense blocks.

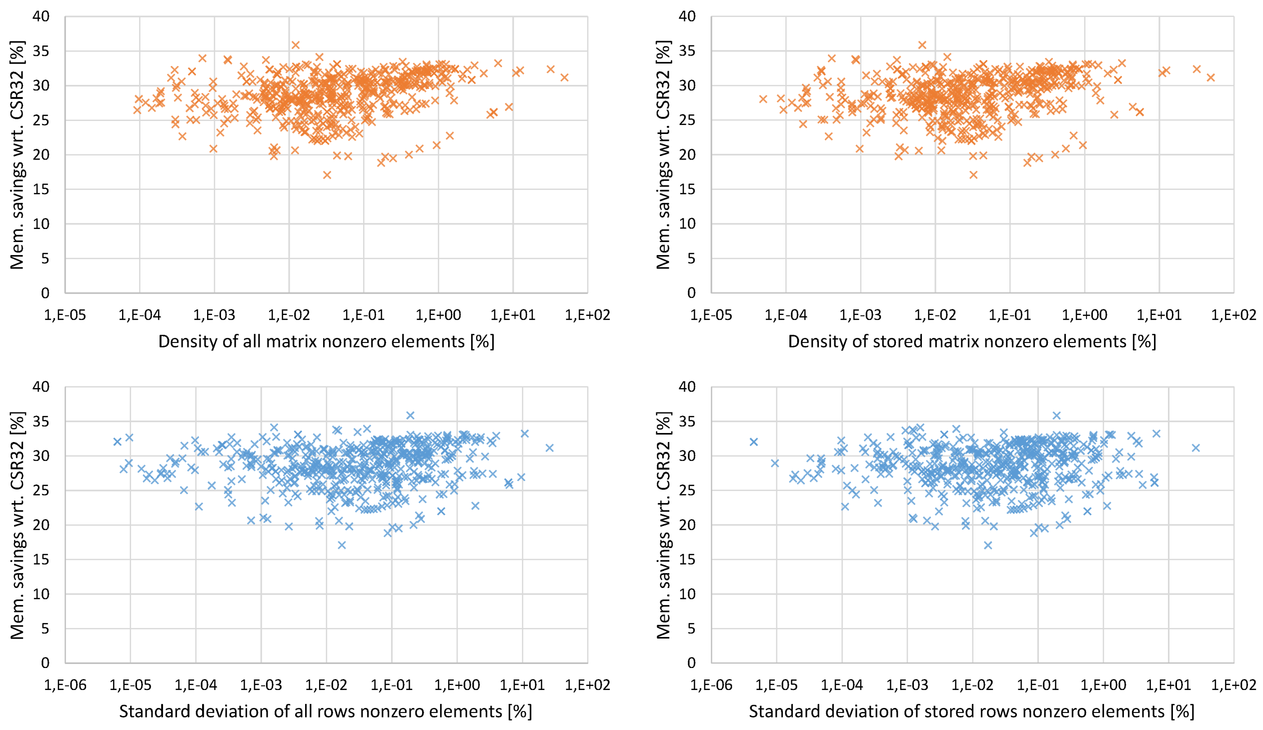


Fig. 3.3. *Relative memory savings against CSR32 in percents as a function of $\varrho$ (*above*) and the standard deviation of prnnz (*below*) measured for the tested matrices and double precision considering both all/stored nonzero elements.*

On the contrary, if we do not assume any particular structure of nonzero elements, we can find an amount of memory required to store the information about matrix nonzero structure as follows: $nnz$ nonzero elements can be placed to $m \times n$ positions in $C(mn, nnz)$ different ways, where

$$C(mn, nnz) = \frac{(mn)!}{(mn - nnz)! \cdot nnz!}. \tag{3.8}$$

To distinguish between them, we thus need $\log_2 C(mn, nnz)$ bits. Such an approach is employed by so-called *Entropy based format* (EBF) [30], which is rather a theoretical concept and establishes a lower bound for generic memory footprints of sparse matrices in cases where no assumptions about their structures are made.

Let $\mathrm{MMF}_{\mathrm{EBF}}(A, b)$ denote a memory footprint of a matrix $A$ stored in memory in EBF with respect to a precision given by $b$. The function

$$\Theta^b(k) = \left(1 - \frac{\min\bigl\{\mathrm{MMF}_{\boxplus}(A_k, s, h \times w, b) : s \in \mathcal{S}_6, h \times w \in \mathcal{B}_{64}\bigr\}}{\mathrm{MMF}_{\mathrm{EBF}}(A_k, b)}\right) \times 100 \tag{3.9}$$

then expresses how much memory in percents we would save if we stored the tested matrix $A_k$ in its optimal block configuration instead of in EBF. We measured these memory savings for all the tested matrices and processed them statistically; the results are presented by Table 3.9. For worst-case matrices, EBF required less memory. However, our block approach still reduced memory footprints of matrices in average by 12.63% and 7.36% for single and double precision, respectively. These results indicate that the nonzero structure of the majority of matrices from our highly diverse benchmark suite have some kind of a block character.

**3.10. Memory Footprints Compared with Lower Bounds.** Another object of our concern within this study was of how much are the memory footprints of the tested matrices higher than their potential minima, i.e., their lower bounds. We further do not consider compression of the values of matrix nonzero elements, since

TABLE 3.9
*Statistics of $\Theta^b(k)$, i.e., memory savings of optimal block configurations against EBF in percents, across the tested matrices.*

| Statistics | Single precision | Double precision |
|---|---|---|
| Minimum | −9.43 | −5.75 |
| Average | 12.63 | 7.36 |
| Maximum | 28.25 | 17.16 |

it is generally worth applying only for special kinds of matrices where nonzero elements contain few unique numbers. To store *nnz* nonzero elements of a matrix $A$ in memory with respect to a precision given by $b$, we thus need $nnz \times b$ bits to store their values and some additional space to store the information about their structure. The lower bound for the latter for any particular structure of nonzero elements is 1 bit, since it is sufficient for distinguishing whether or not a matrix has that particular structure. For instance, we can use this bit to indicate whether a matrix is tridiagonal. If it is, the bit would be set and we can store the values of nonzero elements in a dense array; their row and column indices can then be derived from the positions of values in this array. Such an approach can be generally applied for any particular structure of matrix nonzero elements.

In practice, we would need to store in memory also some additional information about a matrix, such as its dimensions or its number of nonzero elements. However, for large matrices such as those from our tested suite, this additional data require a negligible amount of memory, therefore we define a lower bound for a matrix memory footprint simply as $\mathrm{MMF_{lb}}(A, b) = nnz \times b$.

Let

$$\Gamma^b_{\boxplus}(k) = \left( \frac{\min\{\mathrm{MMF}_{\boxplus}(A_k, s, h \times w, b) : s \in \mathcal{S}_6, h \times w \in \mathcal{B}_{64}\}}{\mathrm{MMF_{lb}}(A_k, b)} - 1 \right) \times 100. \tag{3.10}$$

$\Gamma^b_{\boxplus}(k)$ thus expresses of how much percents is the memory footprint of $A_k$ stored in an optimal block way higher than its lower bound. For comparison purposes, we define corresponding metrics also for CSR32, CSRic, and EBF denoted by $\Gamma^b_{\mathrm{CSR32}}(k)$, $\Gamma^b_{\mathrm{CSRic}}(k)$, and $\Gamma^b_{\mathrm{EBF}}(k)$, respectively:

$$\Gamma^b_{\mathrm{CSR32}}(k) = \left( \frac{\mathrm{MMF_{CSR32}}(A_k, b)}{\mathrm{MMF_{lb}}(A_k, b)} - 1 \right) \times 100, \tag{3.11}$$

$$\Gamma^b_{\mathrm{CSRic}}(k) = \left( \frac{\mathrm{MMF_{CSRic}}(A_k, b)}{\mathrm{MMF_{lb}}(A_k, b)} - 1 \right) \times 100, \tag{3.12}$$

$$\Gamma^b_{\mathrm{EBF}}(k) = \left( \frac{\mathrm{MMF_{EBF}}(A_k, b)}{\mathrm{MMF_{lb}}(A_k, b)} - 1 \right) \times 100. \tag{3.13}$$

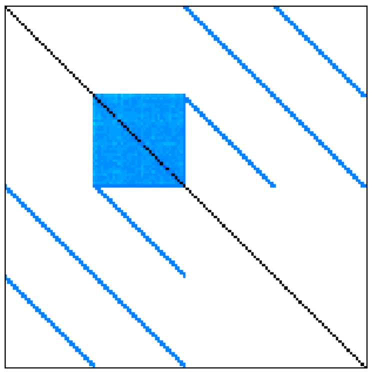
The measured statistics of $\Gamma^b_{\boxplus}(k)$, $\Gamma^b_{\mathrm{CSR32}}(k)$, $\Gamma^b_{\mathrm{CSRic}}(k)$, and $\Gamma^b_{\mathrm{EBF}}(k)$ for the tested matrices are shown in Table 3.10. Memory footprints of partitioned sparse matrices were obviously much closer to the lower bounds than memory footprints of matrices stored in CSR32 and CSRic. Namely, they were 5 times closer in average and 2 times in worst cases than CSR32. In best and average cases, they were even significantly closer to the lower bounds than EBF. In best cases, partitioned matrices almost reached their lower-bound memory footprints. For instance, in double precision, 7, 26, and 120 matrices out of 563 provided memory footprints up to 1, 2, and 5 percents above their lower bounds, respectively.

**3.11. Best-Case and Worst-Case Matrices.** We define a *best-case matrix* and a *worst-case matrix* to be a matrix $A_k$ with the lowest and highest value of $\Gamma^b_{\boxplus}(k)$, respectively.

**3.11.1. Best-Case Matrix.** In our benchmark suite, the best-case matrix was $A_{180}$ called exdata_1 in the UFSMC. It is a square symmetric matrix with 6001 rows/columns, 2269500 nonzero elements in total, and 1137750 nonzero elements in a single triangular part. Its density $\rho(A_{180}) = 6.30\%$ considering all the elements. A visual representation of nonzero pattern of this matrix is shown in Figure 3.4.a.

TABLE 3.10
*Statistics of $\Gamma^b_{\boxplus}(k)$, $\Gamma^b_{\mathrm{CSR32}}(k)$, $\Gamma^b_{\mathrm{CSRic}}(k)$, and $\Gamma^b_{\mathrm{EBF}}(k)$ (in percents) for the tested matrices.*

| | Single precision | | | | Double precision | | | |
|---|---|---|---|---|---|---|---|---|
| Statistics | Blk.-opt. | CSR32 | CSRic | EBF | Blk.-opt. | CSR32 | CSRic | EBF |
| Minimum | 0.63 | 100.02 | 34.52 | 6.37 | 0.31 | 50.01 | 17.26 | 3.18 |
| Average | 21.85 | 111.03 | 57.22 | 39.58 | 10.93 | 55.51 | 28.61 | 19.79 |
| Maximum | 71.31 | 152.39 | 93.19 | 67.15 | 35.66 | 76.19 | 46.60 | 33.58 |


(a) exdata_1

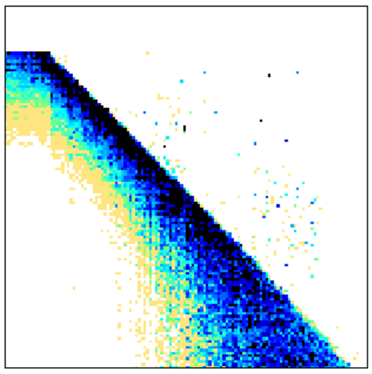

(b) patents_main

FIG. 3.4. *Visualization of nonzero patterns of the best case (*left*) and worst case (*right*) matrices obtained from the UFSMC.*

Relative memory footprints for this matrix are shown in Table 3.11. In case of blocking, the memory footprint of this matrix is of only 0.63% higher than its lower bound. In absolute numbers, the lower bound for single precision is 36408000 bits, while the memory footprint for optimal block configuration is 36637064 bits.

The optimal block size for this matrix is $16 \times 16$, which results in 5592 nonzero blocks, out of which 4278 blocks are fully dense (have all 256 elements nonzero). Vast majority of the matrix nonzero elements (namely 96.26%) are thus stored in dense blocks, which makes the block approach for storage of this matrix such superior in comparison with other formats. Figure 3.4.a shows that this matrix contains one large dense block where most of its nonzero elements are located.

The optimal block scheme for this matrix is adaptive. The second lowest scheme is bitmap, which provides memory footprint 37890016 bits, i.e., 1.03% higher than the optimum.

**3.11.2. Worst-Case Matrix.** The worst-case matrix in our benchmark suite was $A_{385}$ called patents_main in the UFSMC. It is a square unsymmetric matrix with 240547 rows/columns and 560943 nonzero elements. Its density $\rho(A_{385}) = 9.69e-4\%$, thus this matrix is of 4 orders of magnitude more sparse than the best-case matrix exdata_1. A visual representation of the matrix is shown in Figure 3.4.b.

Relative memory footprint for this matrix are shown in Table 3.11. In case of blocking, it is significantly higher than the lower bound. In absolute numbers, the lower bound for single precision is 17950176 bits, while the memory footprint for optimal block configuration is 30750736 bits.

The optimal block size for this matrix is $256 \times 256$, which results in 146772 nonzero blocks. Out of them, 36003 blocks have only a single nonzero element, 111957 blocks have less than 6 nonzero elements, and no block has more than 22 nonzero elements. Nonzero elements are thus spread all over the matrix and not clustered in dense blocks, which hinders low memory footprints for our block approach. Note, however, that in spite of this is the optimal block memory footprint for this matrix only half of that of CSR32 and lower than that of CSRic.

The optimal block scheme for this matrix is COO. The second lowest scheme is adaptive, which provides

Table 3.11

*Relative memory footprints of the best case and worst case matrices in percents with respect to their lower bounds, measured for single precision.*

| Format | exdata_1 ($k = 180$) | patents_main ($k = 385$) |
|---|---|---|
| $\Gamma^{32}_{\boxplus}(k)$ | 0.63 | 71.31 |
| $\Gamma^{32}_{\mathrm{CSR32}}(k)$ | 100.53 | 142.88 |
| $\Gamma^{32}_{\mathrm{CSRic}}(k)$ | 40.97 | 83.05 |
| $\Gamma^{32}_{\mathrm{EBF}}(k)$ | 16.81 | 56.55 |

memory footprint 31044280, i.e., 9.5e-3% higher than the optimum.

**4. Conclusions.** Within this study, we analyzed memory footprints of 563 representative sparse matrices with respect to their partitioning into uniformly sized blocks. We considered different block sizes and different ways of storing blocks in a computer memory. The obtained results led us to the following conclusions:

1. Partitioning of sparse matrices substantially reduces memory footprints of sparse matrices when compared to the most-commonly used storage format CSR32. The average observed memory savings in case of single and double precision were 42.3 and 28.7 percents of memory space, respectively. The corresponding worst-case savings were 25.5 and 17.1 percents.
2. The corresponding memory savings with respect to index-compressed implementation of CSR, i.e., CSRic, were in case of single and double precision 22.4 and 13.7 percents in average, respectively. The same metric with respect to EBF were 12.6 and 7.4 percents, respectively.
3. Partitioning of sparse matrices provides memory footprints much closer to their lower bounds than CSR32. In average, the measured memory footprints for optimal block configurations were of only 21.9 and 10.9 percents higher than the lower bounds, while the corresponding memory footprints for CSR32 were higher of 111.0 and 55.5 percents. Moreover, the memory footprints of matrices most suitable for block processing approach the lower bounds; the amount of memory required for storing information about the structure of nonzero elements of such matrices is relatively negligible.
4. Partitioning of sparse matrices generally provides memory footprints closer to their lower bound than CSRic and even than EBF. Many sparse matrices in real world contain such form of a structure of nonzero elements that is suitable for block processing.
5. For minimization of memory footprints of partitioned sparse matrices, we cannot consider only a single format for storing blocks. Instead, we need to choose a format according to the structure of matrix nonzero elements either for all its blocks collectively (min-fixed scheme) or for each block separately (adaptive scheme). The latter approach mostly yields lower memory footprints.
6. For minimization of memory footprints of partitioned sparse matrices, we cannot consider only a single block size. However, we can substantially reduce the set of block sizes in the optimization space and still obtain memory footprints close to their optima. In average, the measured memory footprints for the proposed reduced sets of block sizes $\mathcal{B}_{20}$, $\mathcal{B}_{14}$, and $\mathcal{B}_8$ and the min-fixed/adaptive schemes were at most of only 1.51 percents higher than the optimal values. Even considering square blocks only is thus generally sufficient for minimization of memory footprints of sparse matrices. However, there exist matrices for which the corresponding metrics are significantly higher and are inversely proportional to the number of tested block sizes. One should thus be aware of whether or not his/her matrices fall into this category and if yes, he/she might consider using larger sets of block sizes.
7. The obtained results seem to be consistent across a wide range of real-world matrices arising from multiple applications problems.
8. There is seemingly no advantage for storing blocks in CSR; without considering this format for blocks, the memory footprints of matrices grow only slightly or not at all. The COO and bitmap formats themselves minimize memory footprints of partitioned sparse matrices, while the dense format is likely the most efficient for related computations.

9. We measured memory savings of partitioned sparse matrices against CSR32 as a function of the following criteria, which are frequently used in the literature: the application problem type, the density of matrix nonzero elements, and the standard deviation of the number of nonzero elements across matrix rows. To our best, we did not find any correlation between the memory savings and these criteria; the block approach thus seems reduce memory footprints of sparse matrices in general.

Our findings are encouraging since they show that memory footprints of partitioned sparse matrices can be substantially reduced even when a relatively small block preprocessing optimization space is considered. Whether or not will such a reduction pay off in practice depends on the objective one wants to achieve. A big challenge is to improve the performance of memory-bounded sparse matrix operations due to the reduction of memory footprints of matrices. Within our future work, we plan to face this problem at least partially—we will focus on the development of scalable efficient block preprocessing and SpMV algorithms for the min-fixed and adaptive block storage schemes, and we will evaluate them experimentally on mainstream HPC architectures.

## REFERENCES

[1] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan, *Fast sparse matrix-vector multiplication on GPUs for graph applications*, in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '14, Piscataway, NJ, USA, 2014, IEEE Press, pp. 781–792.

[2] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, PA, 2nd ed., 1994.

[3] M. Belgin, G. Back, and C. J. Ribbens, *Pattern-based sparse matrix representation for memory-efficient SMVM kernels*, in Proceedings of the 23rd International Conference on Supercomputing, ICS '09, New York, NY, USA, 2009, ACM, pp. 100–109.

[4] ———, *A library for pattern-based sparse matrix vector multiply*, International Journal of Parallel Programming, 39 (2011), pp. 62–87.

[5] G. E. Blelloch, M. A. Heroux, and M. Zagha, *Segmented operations for sparse matrix computation on vector multiprocessors*, Tech. Rep. CMU-CS-93-173, School of Computer Science, Carnegie Mellon University, 1993.

[6] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, *Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks*, in Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures, SPAA '09, New York, NY, USA, 2009, ACM, pp. 233–244.

[7] A. Buluç, S. Williams, L. Oliker, and J. Demmel, *Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication*, in Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium, IPDPS '11, IEEE Computer Society, 2011, pp. 721–733.

[8] D. Buono, F. Petrini, F. Checconi, X. Liu, X. Que, C. Long, and T.-C. Tuan, *Optimizing sparse matrix-vector multiplication for large-scale data analytics*, in Proceedings of the 2016 International Conference on Supercomputing, ICS '16, New York, NY, USA, 2016, ACM, pp. 37:1–37:12.

[9] J.-H. Byun, R. Lin, K. A. Yelick, and J. Demmel, *Autotuning sparse matrix-vector multiplication for multicore*, Tech. Rep. UCB/EECS-2012-215, EECS Department, University of California, Berkeley, 2012.

[10] J. W. Choi, A. Singh, and R. W. Vuduc, *Model-driven autotuning of sparse matrix-vector multiply on GPUs*, in Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '10, New York, NY, USA, 2010, ACM, pp. 115–126.

[11] T. A. Davis and Y. F. Hu, *The University of Florida Sparse Matrix Collection*, ACM Transactions on Mathematical Software, 38 (2011), pp. 1:1–1:25.

[12] J. Dongarra, *Preface: Basic linear algebra subprograms technical (BLAST) forum standard*, International Journal of High Performance Computing Applications, 16 (2002), p. 1.

[13] R. Eberhardt and M. Hoemmen, *Optimization of block sparse matrix-vector multiplication on shared-memory parallel architectures*, in Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2016, pp. 663–672.

[14] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris, *Performance evaluation of the sparse matrix-vector multiplication on modern architectures*, The Journal of Supercomputing, 50 (2009), pp. 36–77.

[15] *IEEE Standard for Floating-Point Arithmetic*, IEEE Std 754-2008, (2008), pp. 1–58.

[16] E.-J. Im and K. Yelick, *Optimizing sparse matrix computations for register reuse in SPARSITY*, in Proceedings of the International Conference on Computational Science (ICCS 2001), Part I, vol. 2073 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2001, pp. 127–136.

[17] E.-J. Im, K. Yelick, and R. Vuduc, *Sparsity: Optimization framework for sparse matrix kernels*, International Journal of High Performance Computing Applications, 18 (2004), pp. 135–158.

[18] R. Kannan, *Efficient sparse matrix multiple-vector multiplication using a bitmapped format*, in 20th Annual International Conference on High Performance Computing, 2013, pp. 286–294.

[19] V. Karakasis, G. Goumas, and N. Koziris, *A comparative study of blocking storage methods for sparse matrices on multicore architectures*, in Proceedings of the 2009 International Conference on Computational Science and Engineering (CSE '09).,

vol. 1, Aug 2009, pp. 247–256.

[20] D. Langr, *Algorithms and Data Structures for Very Large Sparse Matrices*, PhD thesis, Czech Technical University in Prague, 2014.

[21] D. Langr and I. Šimeček, *On memory footprints of partitioned sparse matrices*, in Proceedings of the Federated Conference on Computer Science and Information Systems (FedCSIS 2017), vol. 11 of Annals of Computer Science and Information Systems, Polish Information Processing Society, 2017, pp. 513–512.

[22] D. Langr, I. Šimeček, and T. Dytrych, *Block iterators for sparse matrices*, in Proceedings of the Federated Conference on Computer Science and Information Systems (FedCSIS 2016), IEEE Xplore Digital Library, 2016, pp. 695–704.

[23] D. Langr, I. Šimeček, and P. Tvrdík, *Storing sparse matrices in the adaptive-blocking hierarchical storage format*, in Proceedings of the Federated Conference on Computer Science and Information Systems (FedCSIS 2013), IEEE Xplore Digital Library, 2013, pp. 479–486.

[24] D. Langr, I. Šimeček, P. Tvrdík, T. Dytrych, and J. P. Draayer, *Adaptive-blocking hierarchical storage format for sparse matrices*, in Proceedings of the Federated Conference on Computer Science and Information Systems (FedCSIS 2012), IEEE Xplore Digital Library, 2012, pp. 545–551.

[25] D. Langr and P. Tvrdík, *Evaluation criteria for sparse matrix storage formats*, IEEE Transactions on Parallel and Distributed Systems, 27 (2016), pp. 428–440.

[26] Morton, *A computer oriented geodetic data base and a new technique in file sequencing*, Tech. Rep. Ottawa, Canada, IBM Ltd., 1966.

[27] R. Nishtala, R. W. Vuduc, J. W. Demmel, and K. A. Yelick, *Performance modeling and analysis of cache blocking in sparse matrix vector multiply*, Tech. Rep. UCB/CSD-04-1335, Computer Science Division (EECS), University of California, 2004.

[28] R. Nishtala, R. W. Vuduc, J. W. Demmel, and K. A. Yelick, *When cache blocking of sparse matrix vector multiply works and why*, Applicable Algebra in Engineering, Communication and Computing, 18 (2007), pp. 297–311.

[29] I. Šimeček and D. Langr, *Space and execution efficient formats for modern processor architectures*, in Proceedings of the 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2015), IEEE Computer Society, 2015, pp. 98–105.

[30] I. Šimeček, D. Langr, and P. Tvrdík, *Space-efficient sparse matrix storage formats for massively parallel systems*, in Proceedings of the 14th IEEE International Conference of High Performance Computing and Communications (HPCC 2012), IEEE Computer Society, 2012, pp. 54–60.

[31] F. S. Smailbegovic, G. N. Gaydadjiev, and S. Vassiliadis, *Sparse Matrix Storage Format*, in Proceedings of the 16th Annual Workshop on Circuits, Systems and Signal Processing, ProRisc 2005, 2005, pp. 445–448.

[32] P. Stathis, S. Vassiliadis, and S. Cotofana, *A hierarchical sparse matrix storage format for vector processors*, in Proceedings of the 17th International Symposium on Parallel and Distributed Processing, IPDPS '03, Washington, DC, USA, 2003, IEEE Computer Society, p. 61.

[33] Y. Tao, Y. Deng, S. Mu, Z. Zhang, M. Zhu, L. Xiao, and L. Ruan, *GPU accelerated sparse matrix-vector multiplication and sparse matrix-transpose vector multiplication*, Concurrency and Computation: Practice and Experience, (2014).

[34] P. Tvrdík and I. Šimeček, *A new diagonal blocking format and model of cache behavior for sparse matrices*, in Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics (PPAM 2005), vol. 3911 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2006, pp. 164–171.

[35] S. Williams, A. Waterman, and D. Patterson, *Roofline: An insightful visual performance model for multicore architectures*, Commun. ACM, 52 (2009), pp. 65–76.

[36] X. Yang, S. Parthasarathy, and P. Sadayappan, *Fast sparse matrix-vector multiplication on GPUs: Implications for graph mining*, Proc. VLDB Endow., 4 (2011), pp. 231–242.

*Edited by:* Sasko Ristov
*Received:* Feb 8, 2018
*Accepted:* Sep 5, 2018

DOI 10.12694/scpe.v19i3.1397
SCALABLE COMPUTING: PRACTICE AND EXPERIENCE ISSN 1895-1767
Volume 19, Number 3, pp. 293–300. http://www.scpe.org © 2018 SCPE

# REDUCING THE INTERPROCESSORS MIGRATIONS OF THE EKG ALGORITHM

EL MOSTAFA DAOUDI [*], ABDELMAJID DARGHAM [*], AICHA KERFALI[*] AND MOHAMMED KHATIRI [*†]

**Abstract.** In this work, we consider the scheduling problem of a set of periodic implicit-deadline and synchronous tasks, on a real-time multiprocessor composed of $m$ identical processors. It is known that the cost of migrations and preemptions has significant influence on global system performances. The EKG algorithm can generate a great number of migrant tasks, but it has the advantage that each migrant task migrates between two processors only. Later, the EDHS algorithm has been proposed in order to minimize the number of migrant tasks of EKG. Although EDHS minimizes the number of migration compared to EKG, its drawback is the generation of additional preemptions caused by the migrations on several processors. In this paper we propose a new tasks allocation algorithm that aims to combine the advantages of EKG (migrations between two processors only) and those of EDHS (reduction of number of migrations).

**Key words:** Real time scheduling, EKG, semi-partitioning real time scheduling, inter processor migrations.

**AMS subject classifications.** 68W15, 68W10

## 1. Introduction.

There are two basic approaches for scheduling real-time tasks on multiprocessor / multi-core platforms: global and partitioned scheduling. In partitioned scheduling, tasks are organized in groups, and each task group is assigned to a specific processor. After their allocation to processors, the tasks are not allowed to migrate from one processor to another. When selected for execution, a task can only be dispatched to its assigned processor. On each processor the tasks are scheduled using standard known uniprocessor algorithms e.g. RM (Rate-Monotonic) or EDF (Earliest-Deadline-First) [1]. The main disadvantage of the partitioning approach is that the tasks allocation problem is analogous to the bin packing problem which is known to be NP-Hard [2]. So a task cannot be assigned to any of the processors even if the total available capacity of the whole system is still large. When the individual task utilization is high, this waste could be significant, and in the worst-case only half of the system resource can be used. The alternative to partitioned scheduling is global scheduling in which there is a single queue for tasks that are ready to run. At each time, the $m$ highest priority tasks are dispatched to any available processor according to a global priority scheme. The tasks can migrate from one processor to another which makes it possible to achieve a better use of the platform. Partitioned scheduling has gaps due to the absence of task migration from one processor to another. It is shown that a non schedulable system under partitioned policy can be scheduled if given the opportunity to unassigned tasks to run on multiple processors in global scheduling assuming that the cost of preemptions and migrations is neglected. This assumption is not realistic since this cost has an influence on global system performance. Several works have been proposed in the literature to reduce the number of preemptions and migrations [3, 4, 5].

To overcome the problem of the partitioned approach and increase the utilization rate of the system, recent works [6, 7, 8, 9, 10] have introduced the semi-partitioning scheduling in which most of tasks are assigned to particular processors as the partitioned scheduling, but the remaining tasks (unasigned tasks) are allowed to migrate between processors. In other words, each remaining task is splitted into a set of sub-tasks and each one of them is affected to a processor. This approach allows migration but reduces the number of migrant tasks compared to the global approach.

The Semi-partitioning algorithm EKG [11] cuts the set of processors into groups each one is composed of $k$ processors and limits migration within the same group. In addition, a task can migrate between two processors only. Note that EKG allows to schedule optimally a set of periodic implicit tasks on m processors when $k = m$ (EKG with one group). Since EKG allocates migrant and non-migrant tasks simultaneously, this can generate a great number of migrant tasks.

Kato et al. [12] have proposed the EDHS algorithm which improves the EKG algorithm. It proceeds into two separate steps to allocate the tasks: during the first one, the tasks are assigned according to a given partitioning algorithm in order to minimize the number of migrant tasks generated by the EKG algorithm. The second one consists in allocating migrant tasks on multiple processors according to a second algorithm.

Our contribution aims to combine the advantages of the EKG (migration between two processors only) and those of EDHS (reduction of migrant tasks). We proceed also into two steps to allocate the tasks: the first step is similar to the EDHS one, so we generate the same number of migrations as EDHS algorithm. In order to ensure the schedulability as

[*]Faculty of Sciences, LaRi Laboratory, University of Mohammed First, Oujda, Morocco, e.daoudi@ump.ac.ma, daoudie@yahoo.com, abdelmajid.dargham@gmail.com, kerfali_a@yahoo.fr

[†]Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France, khatiri.med@gmail.com

EKG algorithm, our proposed algorithm avoid that a task migrates between more than two processors during the second step of the algorithm. In order to achieve this goal, our key idea consists in reassigning the first allocated task of processors involved by migrations. In this case our algorithm achieves the optimality like EKG for $k = m$.

The remainder of this paper is organized as follows: in Section 2 we present EKG and EDHS algorithms. Section 3 is devoted to present our proposed algorithm. In Section 4 we present experimental simulations and finally we give the conclusion.

**2. Presentation of the EKG algorithm.** The system is composed of n periodic, implicit-deadline and synchronous tasks noted $\tau_1$, $\tau_2$, ..., $\tau_n$ and $m$ identical processors $P_1$,$P_2$,...,$P_m$. All the tasks cannot be executed in parallel and are independent. Each processor can't execute more than one task at any time. Each task $\tau_i$ has a period $T_i$ (that is also the implicit deadline) and an execution time $C_i$. The ratio $C_i/T_i = U(\tau_i)$ defines the utilization rate of the task $\tau_i$.

The EKG algorithm [11] cut the set of processors into groups each one is composed of $k$ processors and limits migration within the same group. In addition, a task can migrate between two processors only. It allows scheduling optimally a set of periodic tasks with implicit deadline on $m$ processors, when setting the parameter $k$ equal to the number of processors ($k = m$).

Basic principle: Unlike partitioned algorithms, EKG allows tasks to run on two different processors (at different times, without parallelism). The algorithm is divided into two stages:

- Tasks allocation (offline): each task is assigned to one or two processors. The algorithm treats heavy and light tasks differently. A task $\tau_i$ is heavy if $C_i/T_i > SEP$, otherwise it is light where $SEP$ is calculated as follows:

$$SEP = \begin{cases} 1, & if \;\; k = m \\ k/(k+1), & if \;\; k < m \end{cases}$$

  First, the algorithm assigns one heavy tasks to one processor where one processor is dedicated for one heavy (one per task). Then the lighter tasks are assigned to the remaining processors where several light tasks may be assigned to the same processor. To obtain a processor load of 1, some tasks can be split to run on two different processors (migrant task) belonging the same group. If a task is attempted to be assigned to the last processor in a group and it fails, then it is not split, but it is simply assigned to the first processor in a new group. This ensures that tasks in a group do not interact with tasks in another group.
- Tasks scheduling on processors (online): For each group, cutting the time into EKG intervals. An EKG interval is defined by two successive wake-up dates of tasks in the same group. It is similar to slots in the DP-Fair terminology [13], but limited to the tasks of the same group. Similar to DP-Fair, the work of migrant tasks should run for a time proportional to their utilization rate and duration of an interval $[t_0,t_1]$ where $t_0$ denotes the time when a task arrives, and $t_1$ denotes the time when any task in that group arrives next. On an interval $[t_0,t_1]$, if a task $\tau_i$ migrate between processors $P_j$ and $P_{j+1}$, it will be splitted into subtasks $\tau_{11}$ and $\tau_{12}$ as shown on Figure 2.1. At $t_0$ it runs on $P_j$ for $U(\tau_{i1}) * (t_1 - t_0)$ time units and ends its execution at $timea$. Towards the end of the interval at $timeb$, the execution of the task restarts on $P_{j+1}$ for a time duration of $U(\tau_{i2}) * (t_1 - t_0)$ units and ends its execution at $t_1$. The non migrant tasks are scheduled according to EDF on $]timea, timeb[$. After assignment of tasks, at runtime, our algorithm uses the same technique as the EKG algorithm to execute them on each processor.

Reducing the number of preemption by mirroring: The mirror technique called (Mirroring) can be easily implemented by inverting simply $\tau_{i1}$ and $\tau_{i2}$. This halves the number of preemptions. Figure 2.2 shows an execution with this technique. Note that it can be reused for other scheduling policies; it is the case for example DP-WRAP [11].

**3. Presentation of EDHS Algorithm.** EKG assigns migrant and non-migrant tasks simultaneously. This assignment produces several migrant tasks. To minimize the number of migrant tasks Kato et al. [12] have proposed the EDHS algorithm which proceeds into two separate steps: during the first one, the tasks are assigned according to a given partitioning algorithm in order to minimize the number of migrant tasks. The second step consists in allocating, on multiple processors, migrant tasks (tasks that have not been allocated during the first step), according to a second algorithm, as shown on Figure 3.1 .

At runtime, the non-migrant tasks run according to EDF but migrant tasks run with high priority without overlap in time. When migrant task has exhausted its running time on a processor, it continues its execution immediately on the next

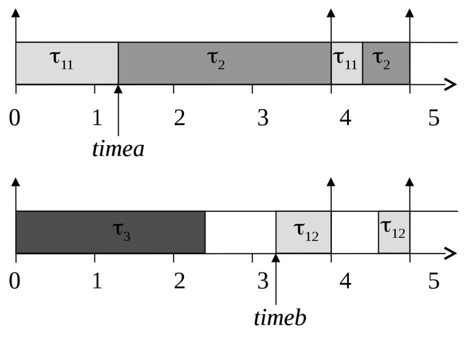


FIG. 2.1. *Migration of the task $\tau_{11}$ between processors $P_j$ and $P_{j+1}$*

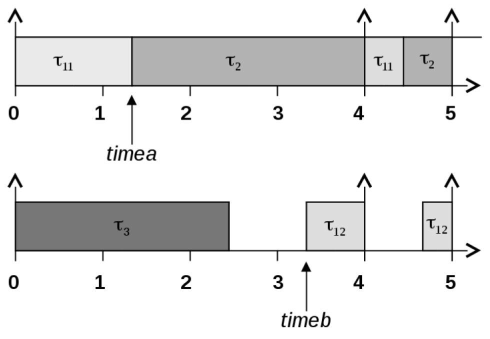


FIG. 2.2. *The execution with mirroring technique*

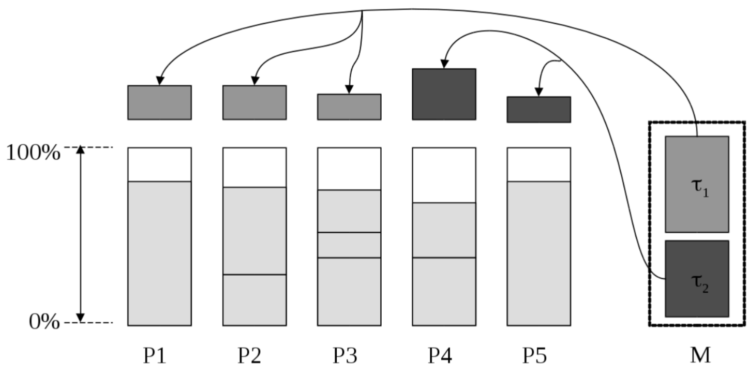


FIG. 3.1. *Allocation of migrant tasks with EDHS algorithm*

processor and preempts the current task as shown on Figure 3.2. Although EDHS minimizes the number of migration compared to EKG, its drawback is the generation of additional preemptions caused by the high priority of migrant tasks on several processors.

**4. The proposed processor allocation heuristic.** The system $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ is composed of a periodic, implicit deadline and synchronous tasks. We assume that $\sum U(\tau_i) \leq m$ and $U(\tau_1) \geq U(\tau_2) \geq \ldots \geq U(\tau_n)$. The following notations are used in the remaining of the paper :

- $M$ : denotes the set of the not allocated tasks. Initially $M = \tau$.
- $\tau[j]$: denotes the set of allocated tasks to the $j^{th}$ processor of the list of processors denoted by $P_j$, for $1 \leq j \leq m$.

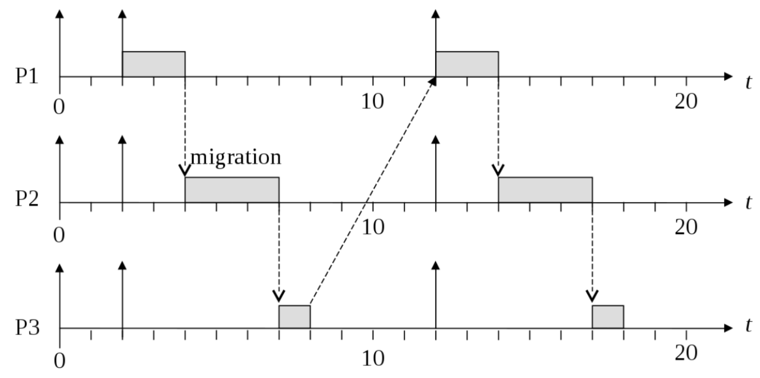


FIG. 3.2. *Execution of migrant tasks with EDHS algorithm*

Initially, $\tau[j] = \emptyset$, for all $j$.

- $U[j]$: denotes the sum of the utilization rates of all tasks allocated to processor $P_j$.
- A processor $P_j$ is full if $U[j] = 1$.
- $cap[j] = 1 - U[j]$: denotes the remaining capacity of processor $P_j$

In order to reduce the number of migrant tasks generated by the EKG algorithm, we proceed into two phases for allocating tasks to processors:

**4.1. First phase.** During the first phase, the allocation of tasks is done by applying one of the most known heuristics based on bin packing problem [14] as EDHS algorithm, namely First-Fit Decreasing, Best-Fit Decreasing and Worst-Fit Decreasing. These algorithms allocate the tasks by sorting them according to their utilization rates in the decreasing order. After this phase, a set of tasks remain still not allocated (the set of migrant tasks). Algorithm 1 describes the First-Fit Decreasing heuristic.

**Algorithm 1** First-Fit Decreasing heuristic

```
for each $\tau_i$ in M do
  j ← 1
  affecter ← 1
  while ( U[j] + U( $\tau_i$ ) > 1) and affecter = 1 do
    j ← j+1
    if $j > m$ then
      affecter ← 0
    end if
  end while
  if affecter=1 then
    $\tau[j] \leftarrow \tau[j] \bigcup \{\tau_i ;\}$
    U[j] ← U[j] + U($\tau_i$);
    M ← $M \backslash \{\tau_i\}$ ;
  end if
end for
```

**4.1.1. Examples.**

- **Example 1:** In the following example, we consider the tasks system $\tau = (\tau_1, \tau_2, \tau_3, \tau_4, \tau_5, \tau_6)$ with U($\tau_1$)=0.7, U($\tau_2$) =0.6, U($\tau_3$) =0.6, U($\tau_4$) =0.4, U($\tau_5$) =0.4 and U($\tau_6$) =0.3. Figure 4.1 shows that the allocation of tasks using EKG algorithm gives rise to two migrant tasks $\tau_2$ ($\tau_{21}$ and $\tau_{22}$) and $\tau_4$($\tau_{41}$ and $\tau_{42}$), but with the First-Fit Decreasing heuristic, there is no migrant task.

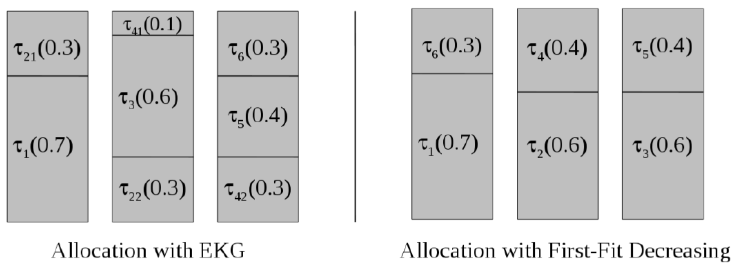


FIG. 4.1. *Allocation with EKG and First Fit Decreasing heuristic on three processors*

- **Example 2:** In the following example we will show that even if we apply the heuristics based on the bin packing problem, we cannot avoid the migration of the tasks. We consider the system $\tau = (\tau_1, \tau_2, \tau_3, \tau_4, \tau_5, \tau_6, \tau_7)$ with U($\tau_1$)=0.9; U($\tau_2$)=0.8; U($\tau_3$)=0.5; U($\tau_4$)=0.3; U($\tau_5$)=0.3; U($\tau_6$)=0.15 and U($\tau_7$)=0.04). In Figure 4.2, it is clear that the First-Fit Decreasing heuristic could not affect the task $\tau_5$, so it must migrate on processors $P_1, P_2$ and $P_3$.

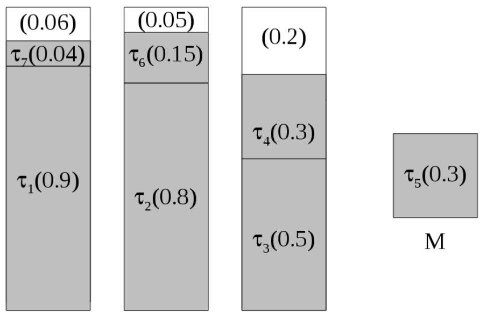


FIG. 4.2. *Allocation with the First Fit Decreasing heuristic on three processors*

**4.1.2. Experimentations.** In Figure 4.3 [5] we compare the number of migrations obtained with the EKG and the proposed algorithms by using the heuristics First-Fit, Best-Fit, Worst-Fit. For simulations, we have considered 10000 task systems and we have calculated the average of the number of migrations in a given interval, for each heuristic. The tasks are randomly generated with the respect of the schedulability condition that is $\sum U(\tau_i) \leq m$. Experimental results show that the heuristics First-Fit, Best-Fit, Worst-Fit reduce significantly the number of migrations. The reduction can reach 60% with the Best Fit and the First Fit heuristics.

**4.2. Second phase.** The second phase consists in allocating the set of remaining tasks (set of migrant tasks). Note that, by construction, the sum of utilization rates of migrant tasks is lower or equal to the sum of remaining processor capacities. Assume that, processors are sorted by decreasing order according to their remaining capacities, $cap[1] \geq cap[2] \geq \ldots \geq cap[m]$ and task $\tau_k$ can migrate on processors $P_1, P_2, \ldots, P_h$ which means that $cap[1] + cap[2] + \cdots + cap[h] \geq U(\tau_k)$ and $cap[1] + cap[2] + \cdots + cap[h-1] < U(\tau_k)$. the width of a time interval is denoted L.

Note that according to the first phase of the heuristic, the first task of each processor $P_j$, for $2 \leq j \leq h$, noted $\rho_j$, verifies $U(\rho_j) \geq U(\tau_k)$. The basic idea is to increase recursively the remaining processor capacities as follows:

- Subdividing the task $\rho_2$ into two subtasks $\rho_{21}$ and $\rho_{22}$, such that U($\rho_{22}$) = cap[1] and U( $\rho_{21}$) = U($P_2$[1])-cap[1]
- Assigning $\rho_{22}$ to $P_1$. In this case $P_1$ becomes full and the capacity of $P_2$ is increased with U( $\rho_{22}$) (cap[2]= cap[2]+ U( $\rho_{22}$).) Thus, task $\rho_{21}$ becomes a migrant task on processors $P_1$ and $P_2$. At runtime:
  - Processor $P_2$ starts its execution by task $P_2$[1] during U( $\rho_{21}$)*L.
  - Processor $P_1$ ends its execution by task $P_2$[1] during U( $\rho_{22}$)*L.
- Recursively, the same process is repeated between processors $P_{j-1}$ and $P_j$, for $2 < j < h$, where the task $\rho_j$ is subdivided into two subtasks $\rho_{j1}$ et $\rho_{j2}$, such that U( $\rho_{j2}$) = cap[j-1]. In this case cap[j]= cap[j]+ U( $\rho_{j2}$). At runtime:

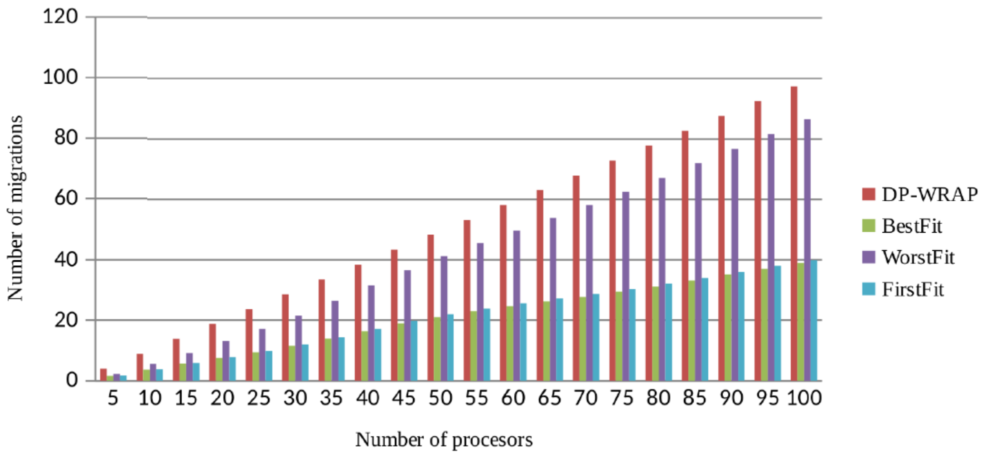


FIG. 4.3. *Number of migration generated by each heuristic*

 - Processor $P_j$ starts its execution by task $\rho_j$ during U($\rho_{j1}$)*L.
 - Processor $P_{j-1}$ ends its execution by task $\rho_j$ during U($\rho_{j2}$)*L.

- After this process, $cap[h-1] < U(\tau_k)$ and $cap[h] + cap[h-1] \geq U(\tau_k)$, then task $\tau_k$ migrates only between processors $P_{h-1}$ and $P_h$ in the following manner: $\tau_k$ is subdivided into two subtasks $\tau_{k1}$ and $\tau_{k2}$, such that U($\tau_{k1}$) = cap[h-1] and U($\tau_{k2}$) = U($\tau_k$) - U($\tau_{k1}$). $P_{h-1}$ starts its execution by task $\tau_{k1}$ during U($\tau_{k1}$)*L and $P_h$ ends its execution by task $\tau_{k2}$ during U($\tau_{k2}$)*L.

With this reallocation, the number of migrations is still the same and each migrant task, migrates between two processors only. In this case our proposed algorithm generates lower migrant tasks than EKG.

**Algorithm 2** allocation of migrant tasks

```
for each $\tau_k$ in M do
  sort in decreasing order the list of processors according to their remaining capacities
  calculate h such as $cap[1]+\ldots+cap[h] \geq U(\tau_k)$ and $cap[1]+\ldots+cap[h-1] < U(\tau_k)$.
  j ← 1
  while $j < h-1$ do
    Subdivide $\rho_{j+1}$ into tow subtasks $\rho_{(j+1)1}$ and $\rho_{(j+1)2}$ such that $U(\rho_{(j+1)2}) = cap[j]$ and $U(\rho_{(j+1)1}) = U(\rho_{j+1}) - cap[j]$
    Assign $\rho_{(j+1)2}$ to $P_j$ then $cap[j+1] = cap[j+1] + cap[j]$ and $P_j$ becomes full.
    Processor $P_{j+1}$ starts its execution by executing task $\rho_{j+1}$ during $U(\rho_{(j+1)1}) * L$
    Processor $P_j$ ends its execution by executing task $\rho_{j+1}$ during $U(\rho_{(j+1)2}) * L$.
    j ← j+1
  end while
  /* $\tau_k$ migrates only between $P_{h-1}$ and $P_h$. */
  Subdivide $\tau_k$ into two subtasks $\tau_{k1}$ and $\tau_{k2}$, such that $U(\tau k1) = cap[h-1]$ and $U(\tau_{k2}) = U(\tau_k) - U(\tau_{k1})$.
  Assign $\tau_{k1}$ to $P_{h-1}$ and $\tau_{k2}$ to $P_h$
  Processor $P_{h-1}$ starts its execution by task $\tau_{k1}$ during $U(\tau_{k1}) * L$
  Processor $P_h$ ends its execution by task $\tau_{k2}$ during $U(\tau_{k2}) * L$.
end for
```

In Figure 4.4 we show the steps of the algorithm in order to allocate a migrant task $\tau_k$ to two processors only instead to allocate it to four processors.

**5. Conclusion.** In this work we have proposed a new semi-partitioned algorithm that reduces the number of migrations like EDHS and limits migrations between two processors only like EKG. The proposed algorithm is designed into

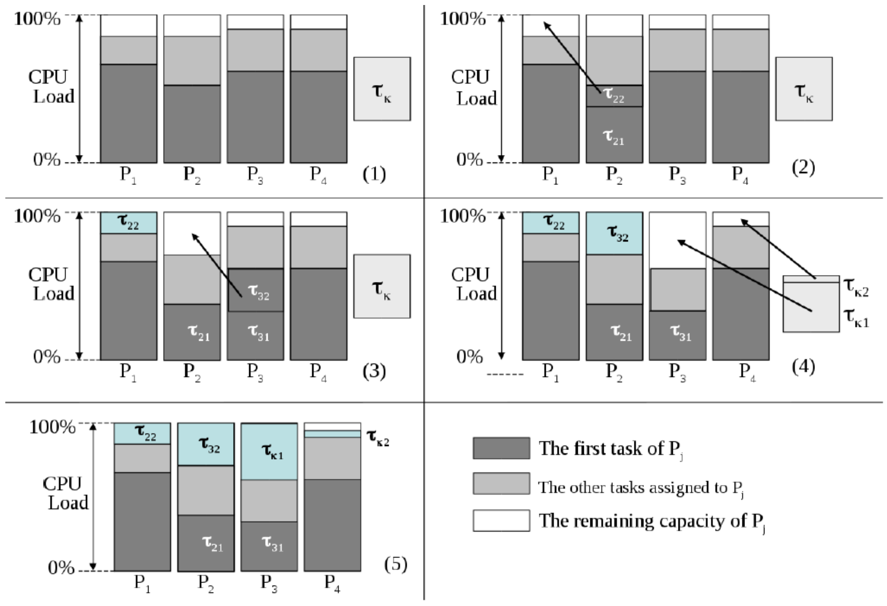

100%
CPU Load
0%
$P_1$ $P_2$ $P_3$ $P_4$
$\tau_K$
$\tau_{22}$ $\tau_{21}$ $\tau_{32}$ $\tau_{31}$
$\tau_{K1}$ $\tau_{K2}$
(1) (2) (3) (4) (5)
The first task of $P_j$
The other tasks assigned to $P_j$
The remaining capacity of $P_j$

FIG. 4.4. *Allocation of the migrant task $\tau_k$ with our proposed algorithm for h=4*

two steps. During the first step we used the well-known bin packing heuristics [14] (the First Fit Decreasing, the Best Fit Decreasing and the Worst fit Decreasing). This step is similar to the first step of the EDHS and it consists in reducing the number of migrant tasks compared to EKG. During the second step of the algorithm, we proposed a new technique that allocates the migrant tasks. Our key idea consists in increasing the number of migrant tasks, each one migrates on two processors, while keeping the same number of migrations: instead to migrate a task between h processors (h-1 migrations), we migrate (h-1) tasks each one between two processors. This reallocation has the advantage that we remain in the same condition of optimality of the EKG for m=k. Experimental simulations show that the number of migrations, compared to EKG, is significantly reduced. This reduction can reach 60%.

## REFERENCES

[1] C. LIU, J. LAYLAND, Scheduling algorithms for multiprogramming in a hard real-time environment. J. ACM, 1(20):46-61, 1973.

[2] M. GAREY, D. JOHNSON, Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman, New York, NY. (1979).

[3] D. AOUN, A.M. DEPLANCHE, Y. TRINQUET, Pfair scheduling improvement to reduce interprocessor migrations. in 16th International Conference on Real-Time and Network Systems (RTNS 2008),

[4] F. NDOYE, Ordonnancement temps reel premptif multiprocesseur avec prise en compte du cot du systme d'exploitation. PhD Thesis, INRIA Paris-Rocquencourt, 2014.

[5] E.M.DAOUDI, A.DARGHAM, A.KERFALI, M.KHATIRI, Reducing the inter processor migrations of the DP-WRAP scheduling. In 12th ACS/IEEE International Conference on Computer Systems and Applications AICCSA. 2015

[6] J.H. ANDERSON, J.P. ERICKSON, U.C. DEVI, B.N. CASSES, Optimal semi-partitioned schedulingin soft real-time systems. In Proceedings of the 20th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications. (2014).

[7] J.A. SANTOS, G. LIMA, K. BLETSAS, S. KATO, Multiprocessor real-time scheduling with a few migrating tasks. in Real-Time Systems Symposium (RTSS), 2013 IEEE 34th; pages 170-181

[8] C. MAIA, P.M. YOMSI, L. NOGUEIRA, L.M. PINHO, Semi-Partitioned Scheduling of Fork-Join Tasks Using Work-Stealing. In Embedded and Ubiquitous Computing (EUC), 2015 IEEE 13th International Conference.

[9] C. MAIA, P.M. YOMSI, L. NOGUEIRA, L.M. PINHO, Real-time semi-partitioned scheduling of fork-join tasks using work-stealing, in EURASIP Journal on Embedded Systems, Springer International Publishing. pp 1-14.

[10] D. CASINI, A. BIONDI, G. BUTTAZZO , Semi-Partitioned Scheduling of Dynamic Real-Time Workload: A Practical Approach Based on Analysis-Driven Load Balancing. In 29th Euromicro Conference on Real-Time Systems (ECRTS 2017). Editor: Marko Bertogna; Article No. 13; pp. 13:1-13:23

[11] B. ANDERSSON, E. TOVAR, Multiprocessor Scheduling with Few Preemptions. 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2006. P. 322 -334.

[12] S. KATO, N. YAMASAKI, Semi-Partitioning Technique for Multiprocessor Real-Time Scheduling. In the 29th IEEE Real-Time Systems Symposium, Work-in-Progress Session (RTSS'08 WiP), 2008

[13] G. LEVIN ET AL., DP-FAIR: A Simple Model for Understanding Optimal Multiprocessor Scheduling. 22nd Euromicro Conference on Real-Time Systems (ECRTS), 2010, pp. 3 -13.

[14] E.G. COFFMAN JR. , M.R. GAREY, D.S. JOHNSON, Approximation algorithms for bin packing: a survey". Boston, MA, USA : PWS Publishing Co., p. 46-93, (1997).

*Edited by:* Dana Petcu
*Received:* Mar 29, 2018
*Accepted:* Aug 24, 2018

DOI 10.12694/scpe.v19i3.1402
SCALABLE COMPUTING: PRACTICE AND EXPERIENCE ISSN 1895-1767
Volume 19, Number 3, pp. 301–307. http://www.scpe.org © 2018 SCPE

# EVALUATING THE SCALABILITY OF BIG DATA FRAMEWORKS

DAVID SANCHEZ[*][¶], OSWALDO SOLARTE[†][¶], VICTOR BUCHELI[‡][¶] AND HUGO ORDONEZ[§]

**Abstract.** The aim of this paper is to present a method based on the isoefficiency model for assessing the scalability in big data environments. The programs word count and sort were implemented and compared in Hadoop and Spark. The results confirm that isoefficiency presented a linear growth as the size of the data sets was increased. They were checked experimentally to ensure that the evaluated frameworks are scalable and a sublinear function was obtained. This paper discusses how scalability in big data is governed by a constant of scalability ($\beta$).

**Key words:** Scalability, Isoefficiency, Big Data, Hadoop, Spark, MapReduce.

**AMS subject classifications.** 68M14

**1. Introduction.** The amount of data created globally is increasing sharply. Every day 2.5 Exabytes of data are generated [23]. Big data refers to data sets that cannot be managed using traditional database systems and techniques. This data may come from diverse sources, such as; emails, videos, images, logs, online transactions, search queries, health records or social networking interactions [17, 23, 6]. Moreover, the Internet of Things (IoT) technology has promoted the creation of several systems that generate enormous quantities of data such as smartphones and sensors. Big data has received increased attention in recent years from researchers and industrial community. This term has five defining characteristics: volume, variety, velocity, veracity and value [6]. Volume refers to the magnitude of data sets (multiple terabytes and petabytes) [6]. Variety refers to the structural heterogeneity in a data set (structured, semi-structured, and unstructured data). Text, images, audio, and video are examples of unstructured data, which do not have the structure required to be analyzed by traditional database systems. Velocity refers to the rate at which data is generated and the speed at which it should be analyzed. As mentioned above, the proliferation of digital devices, such as, smartphones and sensors has led to an unprecedented rate of data creation and is driving a growing need for real-time analytics. Veracity is linked to the unreliability inherent in some sources of data. For example, user feelings in social media are uncertain since they pertain to human judgment. Value is the importance of information extracted from data. The value obtained from big data sets can contribute to improving productivity and competitiveness and create enormous benefits for consumers [2].

The characteristics of big data increase the complexity of storing, processing and analyzing information and pose challenges for obtaining the real value of big data. Several technologies have been proposed for addressing this complexity, such as MapReduce, Hadoop and Spark. MapReduce is a paradigm that parallelizes and executes a program on different machines [4]. Hadoop and Spark are frameworks for non-relational storage and distributed processing. These technologies enable processing semi-structured and unstructured data. Large data sets can be stored and processed in a distributed way [3].

In this scenario, some desirable characteristics of big data environments are scalability, high performance, parallel processing, high fault tolerance and low cost. The scalability is associated with the efficient management of resources and measures the ability of a system to react and adapt to changes in the volume of data without degrading its performance [1]. For example, for opinion mining in twitter, it may be necessary to add new resources continuously, as the number of tweets increases. The models and technologies developed for big data environments have so far been shown to be scalable. Nevertheless, some questions are still only partially answered. How should scalability in big data environments be measured? Is it possible to build a mathematical model of scalability of the big data environment?

In this work, Isoefficiency is introduced as a standard measure of scalability. A model for the evaluation of the scalability of big data environments is also presented. This model was validated using Hadoop and Spark frameworks, for this purpose. Wordcount and Sort programs were developed following the MapReduce paradigm. The experimental evaluation was carried out using data sets whose sizes range from 1Mb to 30Gb (10 data sets). The results show that the Isoefficiency model for each framework is linear and for each program, the growth behavior is sublinear. Thus, the

[*](jose.d.sanchez@correounivalle.edu.co)
[†](oswaldo.solarte@correounivalle.edu.co)
[‡](victor.bucheli@correounivalle.edu.co)
[§]Universidad San Buenaventura, Cali, Colombia,(haordonez@usbcali.edu.co)
[¶]Universidad del Valle, Cali, Colombia

isoefficiency increases as the size of the data set increases, but the isoefficiency growth is low while the file size growth is accelerated. The results show that the tested frameworks are scalable and follow the sublinear behavior of the Isoefficiency function, $Y(s) = \beta X(s)$ where $\beta \approx [0.47 - 0.85] < 1$. Furthermore, scalability is governed by a constant of scalability $\beta$.

The paper proceeds as follows. The next section describes related works. Section 3 discusses the experimental calculation of Isoefficiency, as well as the proposed model. Section 4 discusses the results of the evaluation and section 5 provides a conclusion.

**2. Related work.** Related works are organized into two groups: Isoefficiency based measures and comparison of big data frameworks. Regarding Isoefficiency based measures, several studies have been carried out in different distributed and parallel systems [24, 21, 27]. Moreover, other measures based on Isoefficiency have been proposed for the analysis of the scalability. Firstly, E-differentiation is a measure of scalability in parallel systems, which analyzes the change in the efficiency of a system based on two variables; workload and processors [15]. This measure makes it possible to analyze scalability based on the workload and not only on the number of processors, as is the case in Isoefficiency. Secondly, the Isoefficiency maps allow us to understand the performance of parallel systems, and the measurement is based on techniques such as temperature maps and pressure maps. Isoefficiency maps include several parameters of the parallel performance in two-dimensional planes, which allows considering additional parameters, such as, the communication between parallel processes [5].

Regarding the comparison of frameworks, the infrastructure, workload and information type are usually analyzed [4]. In some approaches, a single cluster configuration is implemented, and the size of the data set varies (this latter approach is used on the present study). This group of studies aims to measure the performance of the framework for large data sets based on the runtime [25, 4, 14].

On the other hand, some studies perform the comparison between structured and unstructured data sets [14], while other authors compare the performance between Spark and Hadoop, using different algorithms [11, 13]. In [25] a comparison of 5 frameworks was carried out. To this end, different SQL queries were executed, and some measures such as response time and fault tolerance were analyzed during the query. Conversely, [9] evaluates the scalability of Hadoop and Spark frameworks based on execution time with relatively small data sets (less than 1.5Gb).

The work shown in [19] presents an Isoefficiency study in the context of big Data. In this work the Isoefficiency is theoretically analyzed. Some variables, such as, calculation costs, synchronization and communication are studied in MapReduce applications, modeled as bulk synchronous parallel tasks. The experimental evaluation shows that the speedup follows a linear trend (100TB data set on 10,000 machines). Finally, other works are devoted to evaluating pattern recognition algorithms based on the MapReduce paradigm [16, 20, 12].

None of the studies reviewed above develop a mathematical model that describes the scalability of big data Systems, nor do they describe the behavior of the relationship between Isoefficiency and workload. Neither of the works is focused on identifying a constant of Isoefficiency that describes the behavior of the scalability in a System. Finally, the model proposed here can be applied to other experimental evaluations (frameworks, programs and data sizes) due to its three layer architecture (processing, storage, and evaluation), seen in Fig. 2.1.

**3. Proposed method for measuring the scalability of big data environments.** Algorithms are described based on the MapReduce paradigm. Isoefficiency is then calculated according to the size of the data set. Lastly, a model is developed, and the $\beta$ constant is computed.

**3.1. Modeling of the programs Word Count and Sort.** The MapReduce model consists of two functions, Map and Reduce. *Map* receives a series of key/value pairs and generates a new intermediate pair key/value that is sent to the *Reduce* function. *Reduce* function combines all the values according to the key [4]. To describe an algorithm using MapReduce, it is necessary to define the Map and Reduce functions. However, there are few models, guidelines or methods that can be found to translate algorithms to the MapReduce paradigm. Therefore, some examples of the programs *Word Count* and *Sort* are presented below.

The Word Count program evaluates the number of appearances of each word in a data set. The *map* function receives a set of key/value pairs, where the key is the name of the document to be processed, and the value is the content of the document. For each word found in the content, a key/value pair is returned, where the key is the word, and the value is 1. The *Reduce* function receives this list and iterates on each element of the list of ones, adding each element. Finally, a key/value pair is returned with the word and the total number of appearances for each word.

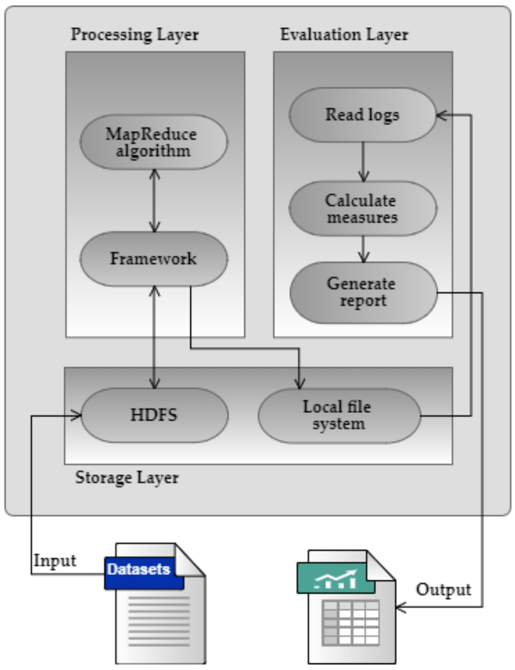


FIG. 2.1. *Architecture for the Proposed Method of the Measuring the Scalability in Big Data Environments*

The Sort program computes a sorted list in alphabetical order of the words from data sets. The *map* function receives a list of key/value pairs, where the key is the name of the document to be processed, and the value is the content of the document. For each word in the content, a key/value pair is returned, where the key is the word, and the value is 1. The *Reduce* function receives a list of key/value pairs and returns the received key, the value is not taken into account in the sort program.

**3.2. Measuring the Isoefficiency.** The measurement is carried out according to the size of the data set and the consumed resources. Figure 2.1 shows the architecture of the approach that consists of 3 layers; processing, storage, and evaluation. The input is a set of flat files stored in HDFS(Hadoop Distributed File System). The processing layer includes the MapReduce based algorithm for evaluation. The storage layer consists of the distributed file system of the framework to be analyzed (Hadoop and Spark), and the local file system. The distributed file system stores the input data, and the data returned by the algorithms in the processing layer. The local file system stores the logs generated by the frameworks during the processing. The logs' files contain data on the processing time, read data, returned data, and used resources among others. Finally, in the evaluation layer, the logs' files are used to calculate all measurements defined for the evaluation. A report is also created to display the results of each algorithm for the data set's different sizes.

The Evaluation was carried out using the following three steps: experimental design, configuration of the framework's infrastructure, and definition of the metrics.

**3.2.1. Experimental design.** Two algorithms (Word Count and Sort) are translated to the MapReduce model to run in Hadoop and Spark frameworks on a cluster with four virtual machines. The efficiency and speedup are used to calculate the Isoefficiency.

**3.2.2. Configuration of the framework infrastructure.** Hadoop is a framework for storing and processing large data sets. Hadoop is based on a master multislave architecture. The main component of Hadoop is the Hadoop distributed file system (HDFS). HDFS stores large data sets in distributed nodes, offering data redundancy and high-performance access. Another component is YARN that manages the resources and the MapReduce tasks. Hadoop also has a library to implement applications following the MapReduce model [18, 22, 26].

For its part, Spark is a fast and general-purpose framework for processing large amounts of data. The main feature of Spark is its memory processing. Spark is a flexible framework that can be integrated with various data storage tools,

TABLE 3.1
*Hardware Description Nodes*

| Hardware | |
|---|---|
| Processors number | 8 |
| Processors model | Intel Xeon E5540 2.53 Ghz |
| Cache size | 8 Mb |
| RAM | 8 Gb |
| Hard drive size | 200 Gb |
| Software | |
| Operating system | Ubuntu 16.04.1 LTS |
| Big data frameworks | Spark 2.0.2, Hadoop 2.7.3 |
| Others | Java 8 |

such as, Hadoop, Cassandra, HBase, among others. Additionally, Spark can be integrated with resource managers, such as, YARN and Apache Mesos [28, 29, 30].

For the experiments, a cluster with four virtual machines connected to the same network was deployed. The cluster consists of a master node and three slave nodes. Each node has the hardware and software specifications shown in Table 3.1.

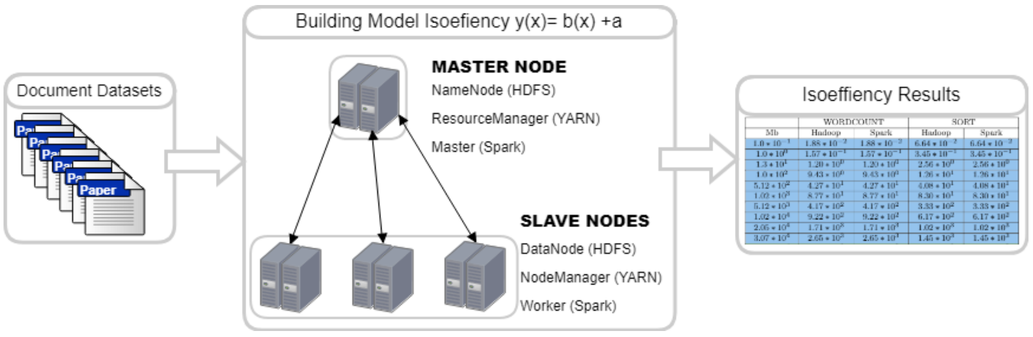
Figure 3.1 depicts the cluster architecture. The data set includes scientific papers from the ISI Web of Knowledge. The data set contains metadata, including title, abstract, authors, etc. However, the analysis was done on the complete paper. The data set is composed of flat files with a total size of 15 Gb. Different samples were taken with different sizes: 100Kb, 1Mb, 13Mb, 100Mb, 512Mb, 1Gb, 5Gb, 10Gb, 20Gb, 30Gb. These data sets were stored in HDFS, and also in a Google Drive repository, available at:
https://drive.google.com/drive/folders/0B0Zl7SmxErLNMm4tZVdyRG8ya1E?usp=sha

**3.2.3. Definition of the Metrics.** Execution time $(T)$ measures the time required to run a program in parallel. $T$ is calculated as $T = TF - TI$, $TF$ is the time when the program ends, and $TI$ the time when the program starts [7].

Speedup $(S)$ measures the performance gain obtained by running a program in parallel, compared to the sequential execution. This metric is defined as the ratio between the time used to solve a problem in a single processor, and the time necessary to solve the same problem in a parallel system with p identical processors. Thus, $TS$ is the runtime of the program executed sequentially, and $T$ the runtime of the program executed in parallel [7]. Speedup is defined by $S = TS/T$.

Efficiency is defined as the ratio between Speedup $(S)$ and the number of processors $P$ [7], mathematically it is defined as $E = S/P$. Furthermore, efficiency is the fraction of the time that each processor is used. In an ideal scenario, the speedup $(S)$ is equal to the number of processors and the efficiency is equal to 1, but generally, the speedup is less than the number of processors and efficiency is between 0 and 1.

Isoefficiency $(W)$ is a measure of the scalability in parallel systems. $W$ measures the relationship between the workload and the number of processors. Here the Isoefficiency is defined by Eq. 3.1 as the rate at which the number of



| | WORDCOUNT | | SORT | |
|---|---|---|---|---|
| Mb | Hadoop | Spark | Hadoop | Spark |
| $1.0 * 10^{-1}$ | $1.88 * 10^{-2}$ | $1.88 * 10^{-2}$ | $6.64 * 10^{-2}$ | $6.64 * 10^{-2}$ |
| $1.0 * 10^{0}$ | $1.57 * 10^{-1}$ | $1.57 * 10^{-1}$ | $3.45 * 10^{-1}$ | $3.45 * 10^{-1}$ |
| $1.3 * 10^{1}$ | $1.20 * 10^{0}$ | $1.20 * 10^{0}$ | $2.56 * 10^{0}$ | $2.56 * 10^{0}$ |
| $1.0 * 10^{2}$ | $9.43 * 10^{0}$ | $9.43 * 10^{0}$ | $1.26 * 10^{1}$ | $1.26 * 10^{1}$ |
| $5.12 * 10^{2}$ | $4.27 * 10^{1}$ | $4.27 * 10^{1}$ | $4.08 * 10^{1}$ | $4.08 * 10^{1}$ |
| $1.02 * 10^{3}$ | $8.77 * 10^{1}$ | $8.77 * 10^{1}$ | $8.30 * 10^{1}$ | $8.30 * 10^{1}$ |
| $5.12 * 10^{3}$ | $4.17 * 10^{2}$ | $4.17 * 10^{2}$ | $3.33 * 10^{2}$ | $3.33 * 10^{2}$ |
| $1.02 * 10^{4}$ | $9.22 * 10^{2}$ | $9.22 * 10^{2}$ | $6.17 * 10^{2}$ | $6.17 * 10^{2}$ |
| $2.05 * 10^{4}$ | $1.71 * 10^{3}$ | $1.71 * 10^{3}$ | $1.02 * 10^{3}$ | $1.02 * 10^{3}$ |
| $3.07 * 10^{4}$ | $2.65 * 10^{3}$ | $2.65 * 10^{3}$ | $1.45 * 10^{3}$ | $1.45 * 10^{3}$ |

FIG. 3.1. *Hardware Architecture for the experiments*

processors must increase in order to keep the efficiency fixed as the workload increases [8, 10].

$$W = K * T_0(w,P) \tag{3.1}$$

where $K = E/1-E$, that is the scalability constant, and $T_0 \approx T * (P-s)$ which is calculated experimentally.

**3.3. The Isoefficiency model.** The Isoefficiency of two programs executed in two different frameworks is calculated. This Isoefficiency is evaluated for different data set sizes$(sd)$. Equation 3.2 represents the fit equation and is computed through the method of least squares.

$$Y(sd) = \beta X(sd) \tag{3.2}$$

where $Y(sd)$ is the Isoefficiency, $X(sd)$ is the data size evaluated in each experiment. $\beta$ is the system scalability coefficient, which is estimated by linear regression. The model is represented by a linear equation, where the least squares adjust model parameters $\beta$ to get a best fit. Therefore, the model constructs a model with a structure that fits the $y_i$ for successive x values, using the parameters $\beta$; For the property RSquared, the computation of the total sum of squares is mean adjusted, other properties related to the sum of squared errors are included such as Standard error, T-statistic and P-value. The model was implemented in the Mathematica software package.

To study the relationship between the program input and the Isoefficiency, a model based on Eq. (3.2) was built. In each case, $\beta$ represents the proportion in which the Isoefficiency grows according to the input. This linear function represents the scalability of the system. In most cases $\beta$ grows sub-linearly depending on the data size $(sd)$. The positive slope $\beta$ less than 1 means sub-linear behavior, that is, the value on the x-axis is higher than the value on the y-axis. Thus, the sublinear growth in the Isoefficiency indicates that the system is scalable since the increase of the resources is less than the increase of the workload.

**4. Results.** Table 4.1 shows the experimental results. Blue displays the experimental data, and green the calculated Isoefficiency of Hadoop and Spark frameworks (for Word Count and Sort programs). It can be seen that for a particular size of data set, the Isoefficiency values are similar for both frameworks. The behavior is the same when the size of the data set is less than 1 Gb. On the other hand, when the size of the data sets is greater than 5 Gb, the behavior is different. The Word Count scalability is less than Sort scalability due to the higher consumption of resources.

Table 4.2 shows the statistical model obtained from the experimental evaluation of the two programs. The table presents the isoefficiency function $(m(x)+b)$, the function slope (m estimate), the standard error and the p-value, and the R2.

The results show that, the standard error values and the t-statistics are similar for both programs. A minimum square error of statistical significance 99% is also achieved. It can be inferred that the results provide reliability in the adjustment of the model. In this sense, the results are confirmed because the positive slope less than 1 indicates a sub-linear behavior. This means, that the distance between each point on the x-axis is higher than the distance between each point on the y-axis. The sub-linear growth of the Isoefficiency indicates that the system is scalable, since the Isoefficiency increases less than the workload (size of the data set) [8]. Thus, it is confirmed that the frameworks are scalable (for the word count and sort programs) since a slight increase in resources is required to keep the efficiency.

**5. Conclusions and Future works.** An experimental evaluation of the Isoefficiency was carried out to analyze the scalability of two big data frameworks: Hadoop and Spark. The function of Isoefficiency has been obtained according to the size of the data sets for two MapReduce programs (Wordcount and Sort). The function regulates the Isoefficiency as a linear function. It was shown that the Isoefficiency of the two frameworks was the same when executing the programs based on the MapReduce paradigm. From the Isoefficiency function, a sublinear growth of the Isoefficiency associated with an increase of the data size was evidenced. That is to say, as the data size increases the Isoefficiency of the frameworks increase in smaller proportion. The experiments show that Hadoop and Spark are an excellent choice to store and process large data sets, since the results obtained from Isoefficiency are similar. A future project might implement a MapReduce program for indexing scientific unstructured data using Hadoop, Spark, and Elasticsearch.

TABLE 4.1
*Results of Scalability Experimintation and Proposed Model evaluation.*

| | WORDCOUNT | | SORT | |
|---|---|---|---|---|
| Mb | Hadoop | Spark | Hadoop | Spark |
| $1.0*10^{-1}$ | $1.88*10^{-2}$ | $1.88*10^{-2}$ | $6.64*10^{-2}$ | $6.64*10^{-2}$ |
| $1.0*10^{0}$ | $1.57*10^{-1}$ | $1.57*10^{-1}$ | $3.45*10^{-1}$ | $3.45*10^{-1}$ |
| $1.3*10^{1}$ | $1.20*10^{0}$ | $1.20*10^{0}$ | $2.56*10^{0}$ | $2.56*10^{0}$ |
| $1.0*10^{2}$ | $9.43*10^{0}$ | $9.43*10^{0}$ | $1.26*10^{1}$ | $1.26*10^{1}$ |
| $5.12*10^{2}$ | $4.27*10^{1}$ | $4.27*10^{1}$ | $4.08*10^{1}$ | $4.08*10^{1}$ |
| $1.02*10^{3}$ | $8.77*10^{1}$ | $8.77*10^{1}$ | $8.30*10^{1}$ | $8.30*10^{1}$ |
| $5.12*10^{3}$ | $4.17*10^{2}$ | $4.17*10^{2}$ | $3.33*10^{2}$ | $3.33*10^{2}$ |
| $1.02*10^{4}$ | $9.22*10^{2}$ | $9.22*10^{2}$ | $6.17*10^{2}$ | $6.17*10^{2}$ |
| $2.05*10^{4}$ | $1.71*10^{3}$ | $1.71*10^{3}$ | $1.02*10^{3}$ | $1.02*10^{3}$ |
| $3.07*10^{4}$ | $2.65*10^{3}$ | $2.65*10^{3}$ | $1.45*10^{3}$ | $1.45*10^{3}$ |
| $5.12*10^{4}$ | $4.38*10^{3}$ | $4.38*10^{3}$ | $2.48*10^{3}$ | $2.48*10^{3}$ |
| $6.14*10^{4}$ | $5.25*10^{3}$ | $5.25*10^{3}$ | $2.97*10^{3}$ | $2.97*10^{3}$ |
| $7.17*10^{4}$ | $6.13*10^{3}$ | $6.13*10^{3}$ | $3.46*10^{3}$ | $3.46*10^{3}$ |
| $8.19*10^{4}$ | $7.0*10^{3}$ | $7.0*10^{3}$ | $3.95*10^{3}$ | $3.95*10^{3}$ |
| $9.22*10^{4}$ | $7.88*10^{3}$ | $7.88*10^{3}$ | $4.44*10^{3}$ | $4.44*10^{3}$ |
| $1.02*10^{5}$ | $8.75*10^{3}$ | $8.75*10^{3}$ | $4.93*10^{3}$ | $4.93*10^{3}$ |
| $1.05*10^{6}$ | $8.97*10^{4}$ | $8.97*10^{4}$ | $5.03*10^{4}$ | $5.03*10^{4}$ |
| $1.07*10^{9}$ | $9.18*10^{7}$ | $9.18*10^{7}$ | $5.14*10^{7}$ | $5.14*10^{7}$ |
| $1.10*10^{12}$ | $9.40*10^{10}$ | $9.40*10^{10}$ | $5.27*10^{10}$ | $5.27*10^{10}$ |
| $1.13*10^{15}$ | $9.63*10^{13}$ | $9.63*10^{13}$ | $5.39*10^{13}$ | $5.39*10^{13}$ |
| $1.15*10^{18}$ | $9.86*10^{16}$ | $9.86*10^{16}$ | $5.52*10^{16}$ | $5.52*10^{16}$ |

TABLE 4.2
*Statistical Model for Scalability*

| Program/ framework | Equation $Y(t)=\beta X(t)$ | $\beta$ estimated | Standard error | T-statistic | P-value | $R^2$ |
|---|---|---|---|---|---|---|
| Wordcount Hadoop | 0.0855x - 0.2874 | 0.0855 | 0.0008 | 110.858 | $4.8987*10^{-14}$ | 0.99 |
| Wordcount Spark | 0.0855x - 0.2874 | 0.0855 | 0.0008 | 110.858 | $4.8987*10^{-14}$ | 0.99 |
| Sort Hadoop | 0.0479x + 29.4126 | 0.0479 | 0.0015 | 32.2172 | $9.3867*10^{-10}$ | 0.99 |
| Sort Spark | 0.0479x + 29.4126 | 0.0479 | 0.0015 | 32.2172 | $9.3867*10^{-10}$ | 0.99 |

## REFERENCES

[1] A. BONDI, *Characteristics of Scalability and Their Impact on Performance*, Proceedings of the 2nd international workshop on Software and performance, (2000), pp. 195–203.
[2] M. CHEN, S. MAO, AND Y. LIU, *Big data: A survey*, Mobile Networks and Applications, 19 (2014), pp. 171–209.
[3] C. COSTA AND M. Y. SANTOS, *Big Data: State-of-the-art concepts, techniques, technologies, modeling approaches and research challenges*, IAENG International Journal of Computer Science, (2017).
[4] J. DEAN AND S. GHEMAWAT, *MapReduce: Simplified Data Processing on Large Clusters*, in Proc. of the OSDI - Symp. on Operating Systems Design and Implementation, USENIX, 2004, pp. 137–149.
[5] M. DROZDOWSKI AND L. WIELEBSKI, *Isoefficiency maps for divisible computations*, IEEE Transactions on Parallel and Distributed Systems, 21 (2010), pp. 872–880.
[6] A. GANDOMI AND M. HAIDER, *Beyond the hype: Big data concepts, methods, and analytics*, International Journal of Information Management, (2015).
[7] A. GRAMA, A. GUPTA, G. KARYPIS, AND V. KUMAR, *Introduction to Parallel Computing, Second Edition*, Addison Wesley, second ed., 2003.

[8] A. Y. Grama, A. Gupta, and V. Kumar, *Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures*, IEEE Parallel and Distributed Technology, 1 (1993), pp. 12–21.

[9] L. Gu and H. Li, *Memory or time: Performance evaluation for iterative operation on hadoop and spark*, Proceedings - 2013 IEEE International Conference on High Performance Computing and Communications, HPCC 2013 and 2013 IEEE International Conference on Embedded and Ubiquitous Computing, EUC 2013, (2014), pp. 721–727.

[10] V. Kumar, V. N. Rao, and K. Ramesh, *Parallel Depth First Search on the Ring Architecture*, (1988).

[11] B. Kupisz and O. Unold, *Collaborative filtering recommendation algorithm based on Hadoop and Spark*, 2015 IEEE International Conference on Industrial Technology (ICIT), (2015), pp. 1510–1514.

[12] J. Li, D. Li, and Y. Zhang, *Efficient Distributed Data Clustering on Spark*, 2015 IEEE International Conference on Cluster Computing, (2015), pp. 504–505.

[13] X. Lin, P. Wang, and B. Wu, *Log analysis in cloud computing environment with Hadoop and Spark*, 2013 5th IEEE International Conference on Broadband Network & Multimedia Technology, (2013), pp. 273–276.

[14] O. O. Malley and A. C. Murthy, *Winning a 60 Second Dash with a Yellow Elephant Hadoop implementation*, Proceedings of sort benchmark, 1810 (2009), pp. 1–9.

[15] Min Wang, Weiqun Ding, and Hong Lin, *E-differentiation for analyzing scalability of parallel algorithms on parallel architectures*, Proceedings of ICICS, 1997 International Conference on Information, Communications and Signal Processing. Theme: Trends in Information Systems Engineering and Wireless Multimedia Communications (Cat. No.97TH8237), 3 (1997), pp. 1457–1461.

[16] S. Muppidi, *Document Clustering with Map Reduce using Hadoop Framework*, International Journal on Recent and Innovation Trends in Computing and Communication, (2015).

[17] S. Sagiroglu and D. Sinanc, *Big data: A review*, International Conference on Collaboration Technologies and Systems (CTS), (2013), pp. 42–47.

[18] R. D. Schneider, *Hadoop for Dummies Special Edition*, John Wiley & Sons Canada, Ltd., special ed., 2012.

[19] H. Senger, V. Gil-Costa, L. Arantes, C. A. Marcondes, M. Marín, L. M. Sato, and F. A. Da Silva, *BSP cost and scalability analysis for MapReduce operations*, in Concurrency Computation, 2016.

[20] N. Shah and S. Mahajan, *Distributed Document Clustering Using K-Means*, International Journal of Advanced Research in Computer Science and Software Engineering, 4 (2014), pp. 2277–128.

[21] S. Shudler, A. Calotoiu, T. Hoefler, and F. Wolf, *Isoefficiency in Practice: Configuring and Understanding the Performance of Task-based Applications*, Principles and Practice of Parallel Programming (PPoPP), (2017).

[22] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, *The Hadoop distributed file system*, 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST2010, (2010).

[23] S. Singh and N. Singh, *Big Data analytics*, 2012 International Conference on Communication, Information & Computing Technology (ICCICT), (2012), pp. 1–4.

[24] V. Singh, V. Kumar, G. Agha, and C. Tomlinson, *Efficient algorithms for parallel sorting on mesh multicomputers*, International Journal of Parallel Programming, 20 (1991), pp. 95–131.

[25] P. Wendell, *Big Data Benchmark*, 2014.

[26] T. White, *Hadoop: The definitive guide*, vol. 54, O'Reilly Media, Inc., fourth edi ed., 2015.

[27] T.-R. Yang and H.-X. Lin, *Isoefficiency Analysis of CGLS Algorithm for Parallel Least Squares Problems*, (1997), pp. 452–461.

[28] M. Zaharia, M. Chowdhury, T. Das, and A. Dave, *Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing*, Nsdi, (2012), pp. 2–2.

[29] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, *Spark : Cluster Computing with Working Sets*, HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing, (2010), p. 10.

[30] M. Zaharia, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, I. Stoica, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, and S. Venkataraman, *Apache Spark: a unified engine for big data processing*, Communications of the ACM, 59 (2016), pp. 56–65.

*Edited by:* Dana Petcu
*Received:* Jun 2, 2018
*Accepted:* Sep 1, 2018

## AIMS AND SCOPE

The area of scalable computing has matured and reached a point where new issues and trends require a professional forum. SCPE will provide this avenue by publishing original refereed papers that address the present as well as the future of parallel and distributed computing. The journal will focus on algorithm development, implementation and execution on real-world parallel architectures, and application of parallel and distributed computing to the solution of real-life problems. Of particular interest are:

**Expressiveness:**
- high level languages,
- object oriented techniques,
- compiler technology for parallel computing,
- implementation techniques and their efficiency.

**System engineering:**
- programming environments,
- debugging tools,
- software libraries.

**Performance:**
- performance measurement: metrics, evaluation, visualization,
- performance improvement: resource allocation and scheduling, I/O, network throughput.

**Applications:**
- database,
- control systems,
- embedded systems,
- fault tolerance,
- industrial and business,
- real-time,
- scientific computing,
- visualization.

**Future:**
- limitations of current approaches,
- engineering trends and their consequences,
- novel parallel architectures.

Taking into account the extremely rapid pace of changes in the field SCPE is committed to fast turnaround of papers and a short publication time of accepted papers.

## INSTRUCTIONS FOR CONTRIBUTORS

Proposals of Special Issues should be submitted to the editor-in-chief.

The language of the journal is English. SCPE publishes three categories of papers: overview papers, research papers and short communications. Electronic submissions are preferred. Overview papers and short communications should be submitted to the editor-in-chief. Research papers should be submitted to the editor whose research interests match the subject of the paper most closely. The list of editors' research interests can be found at the journal WWW site (`http://www.scpe.org`). Each paper appropriate to the journal will be refereed by a minimum of two referees.

There is no a priori limit on the length of overview papers. Research papers should be limited to approximately 20 pages, while short communications should not exceed 5 pages. A 50–100 word abstract should be included.

Upon acceptance the authors will be asked to transfer copyright of the article to the publisher. The authors will be required to prepare the text in $\LaTeX 2_\varepsilon$ using the journal document class file (based on the SIAM's `siamltex.clo` document class, available at the journal WWW site). Figures must be prepared in encapsulated PostScript and appropriately incorporated into the text. The bibliography should be formatted using the SIAM convention. Detailed instructions for the Authors are available on the SCPE WWW site at `http://www.scpe.org`.

Contributions are accepted for review on the understanding that the same work has not been published and that it is not being considered for publication elsewhere. Technical reports can be submitted. Substantially revised versions of papers published in not easily accessible conference proceedings can also be submitted. The editor-in-chief should be notified at the time of submission and the author is responsible for obtaining the necessary copyright releases for all copyrighted material.