J. Parallel Distrib. Comput. 93–94 (2016) 87–101

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

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

Data-aware task scheduling for all-to-all comparison problems in heterogeneous distributed systems

Yi-Fan Zhang, Yu-Chu Tian *, Colin Fidge, Wayne Kelly

School of Electrical Engineering and Computer Science, Queensland University of Technology (QUT), GPO Box 2434, Brisbane QLD 4001, Australia

HIGHLIGHTS

- Abstraction of all-to-all comparison computing pattern with big data sets.
- Formulation of all-to-all comparisons in distributed systems as a constrained optimization.
- Data-aware task scheduling approach for solving the all-to-all comparison problem.
- Metaheuristic data pre-scheduling and dynamic task scheduling in the approach.

ARTICLE INFO

Article history:
Received 25 October 2015
Received in revised form
14 February 2016
Accepted 14 April 2016
Available online 25 April 2016

Keywords:
Distributed computing
All-to-all comparison
Data distribution
Task scheduling

ABSTRACT

Solving large-scale all-to-all comparison problems using distributed computing is increasingly significant for various applications. Previous efforts to implement distributed all-to-all comparison frameworks have treated the two phases of data distribution and comparison task scheduling separately. This leads to high storage demands as well as poor data locality for the comparison tasks, thus creating a need to redistribute the data at runtime. Furthermore, most previous methods have been developed for homogeneous computing environments, so their overall performance is degraded even further when they are used in heterogeneous distributed systems. To tackle these challenges, this paper presents a data-aware task scheduling approach for solving all-to-all comparison problems in heterogeneous distributed systems. The approach formulates the requirements for data distribution and comparison task scheduling simultaneously as a constrained optimization problem. Then, metaheuristic data pre-scheduling and dynamic task scheduling strategies are developed along with an algorithmic implementation to solve the problem. The approach provides perfect data locality for all comparison tasks, avoiding rearrangement of data at runtime. It achieves load balancing among heterogeneous computing nodes, thus enhancing the overall computation time. It also reduces data storage requirements across the network. The effectiveness of the approach is demonstrated through experimental studies.

1. Introduction

The size of data sets has grown rapidly across a variety of systems and applications [2,15]. Distributed computing using a distributed data storage architecture has been widely applied in data intensive and computationally intensive problems due to its cost effectiveness, high reliability and high scalability [4,14,21]. It decomposes a single large computing problem into multiple smaller ones and then schedules those smaller ones to distributed worker nodes. Its performance largely depends on the data distribution, task decomposition, and task scheduling strategies. A significant performance degradation may result from inappropriate data distribution, poor data locality for computing tasks, and unbalanced computational loads among the distributed worker nodes. An inappropriate data distribution consumes excessive storage space. Poor data locality means that the data required by a particular worker is unavailable locally, thus creating overheads associated with rearranging data between the nodes at runtime. Load imbalances lengthen the overall computation time due to the need to wait for the slowest nodes. Thus, innovative approaches are required to deal with all these issues for distributed computing of large-scale problems with distributed data.

All-to-all comparison is a type of computing problem with a unique pairwise computing pattern [18,19,33]. It involves comparing two different data items from a data set for all possible pairs of data items. It is widely found in various application domains such

* Corresponding author.
E-mail address: y.tian@qut.edu.au (Y.-C. Tian).

http://dx.doi.org/10.1016/j.jpdc.2016.04.008
0743-7315/

as bioinformatics, biometrics and data mining. For example, in data mining, clustering algorithms use all-to-all comparisons to derive a similarity matrix to characterize the similarities between objects. For instance, in a study of music information retrieval, 3090 pieces of music were pairwise compared to determine the similarity between any pair of items [3].

Existing solutions for general distributed computing have been adapted to distributed processing of all-to-all comparison problems. They generally consider data distribution and comparison task scheduling in two independent phases. There are two representative ideas for data distribution: (1) to copy all data to all nodes [1,11,24]; and (2) to distribute data by using the Hadoop computing framework [28,6]. While considering data distribution and task scheduling in two independent phases simplifies the system's design, significant weaknesses are caused by the lack of coordination between the two phases, leading to poor performance of the overall distributed computation. This is in addition to the inherent drawbacks of these two approaches for all-to-all comparison problems, as is discussed below in Section 2.

To solve these problems, here we present a data-aware task scheduling approach for distributed computation of large-scale, all-to-all comparison problems with distributed storage in heterogeneous distributed systems. Specific contributions of this work include:

1. A formalization of distributed all-to-all comparison computations as a constrained optimization problem, which considers data distribution and task scheduling simultaneously;
2. A metaheuristic pre-scheduling strategy, as a solution to the constrained optimization requirement, for task-oriented data distribution with consideration of storage usage, data locality, and load balancing; and
3. Runtime-scheduling strategies, as a refinement to the pre-scheduling strategy, for static and dynamic scheduling of comparison tasks, with consideration of the computing power of the individual computing nodes in heterogeneous distributed systems.

The paper is organized as follows. Section 2 discusses related work and motivations. Section 3 describes all-to-all comparison problems and their challenges. This is followed by a formalization of distributed all-to-all comparisons as a constrained optimization problem in Section 4. Then, metaheuristic pre-scheduling and runtime-scheduling strategies are presented in Sections 5 and 6, respectively. The strategies are further analysed in Section 7. Section 8 presents experimental results. Finally, Section 9 concludes the paper.

2. Related work and motivations

All-to-all comparison problems occur in various application domains. However, the principle of solving all these problems is the same. A few typical methods are reviewed below.

A brute-force solution to all-to-all comparison problems copies all data onto each node in a distributed system [24,23,32]. Moretti et al. [24] designed a computing framework for all-to-all comparison problems on campus grids. To give each comparison task the required data, they proposed a spanning tree method to deliver the data to every node efficiently. Macedo et al. [23] presented an MPI/OpenMP mixed parallel strategy to run the DIALIGN-TX algorithm in heterogeneous multi-core clusters. They focused on comparing different task allocation policies to show an appropriate choice. Xiao et al. [32] optimized the BLAST algorithm in a GPU–CPU mixed heterogeneous computing system. Their implementation was measured to achieve a six-fold speed-up for the BLAST algorithm. For other big data computing problems other than all-to-all comparisons, distributing all data everywhere was also a widely used data strategy [1].

Distributing all data to all nodes has its advantages and disadvantages. When the data sets are distributed everywhere, scheduling any comparison task to any node will achieve perfect data locality, and load balancing becomes straightforward. However, there are also obvious and major drawbacks. (1) The brute-force replication of data results in worst-case storage usage, the longest time consumption for data transmission, and the highest cost of network communications. For example, a typical all-to-all comparison problem presented by Hess et al. [13] needs to process 268 GB of cow rumen metagenome data, and copying all the data to each node pushes the limits of the available storage resources. In an experiment by Das et al. [9], the average time for deploying 10 GB data sets within a cluster of 14 worker nodes and a 10 Mbps network took nearly 150 min. This long time for data transmission has a significant negative effect on the overall performance of the computing problem. (2) Even if all the data can be duplicated efficiently, much of the data stored in the nodes will never be used in actual comparison tasks, wasting the storage resources considerably. (3) These two drawbacks become even more evident and serious for large-scale problems with big data sets. As all-to-all comparison is a type of combinatorial problem, the complexity of processing big data sets increases exponentially with the size of the data.

Hadoop is a widely-used distributed computing framework for big data problems, based on the MapReduce computing pattern. Therefore, recent attempts have been made to implement domain-specific all-to-all comparison applications using Hadoop [28,6]. Using Hadoop, CloudBurst [28] parallelizes a specific read-mapping algorithm optimized for mapping next-generation sequence data to the human genome and other reference genomes. In a test using a homogeneous cluster of 24 processors, CloudBurst has been shown to achieve an up to 30 times speed-up compared to execution on a single core. Similarly, MRGIS [6] is a parallel and distributed computing platform implemented in Hadoop for geoinformatics applications. A Hadoop environment with 32 homogeneous worker nodes has been investigated for testing the efficiency of the MRGIS system [6]. These solutions benefit from Hadoop's advantages such as scalability, redundancy, automatic monitoring, and distributed computing with simple application programming interfaces (APIs). However, they are domain-specific, and thus not suitable for general all-to-all comparison problems with big data sets from different application domains. Therefore, they were not used as benchmark examples in our experiments to evaluate the performance of our solution for general all-to-all comparison problems.

While Hadoop is widely used, it is inefficient for large-scale all-to-all comparison problems for two reasons. (1) Hadoop is based on the MapReduce computing pattern, which is fundamentally different from the pairwise comparison pattern needed in all-to-all comparison problems. In MapReduce problems, each data item can be processed separately and there is no requirement for pairwise data locality. In Hadoop data items are randomly distributed with a fixed number of replications (the default is 3). Hadoop's data distribution strategy does not consider the data locality requirements for comparison tasks. A naïve attempt to use Hadoop's data distribution for all-to-all comparisons causes a need to redistribute the data between the nodes at runtime [33]. In Qiu et al.'s experiments [27], Hadoop was shown to execute inefficiently for all-to-all comparisons due to frequent switches between 'map' and 'communication' activities. (2) Hadoop does not address load balancing directly, which is also a key requirement for improved performance of the overall execution time in all-to-all comparison problems. This becomes most evident in heterogeneous distributed systems, in which

Table 1
Nomenclature.

Notation	Description
ΔE	Energy difference between neighbour and current states
ΔF	Cost difference, $\Delta F = F(S) - F(S')$
Δf	The value of the cost difference
$C(x, y)$	Comparison task between data items x and y
D_i	Data set allocated to node i
$\|D_i\|$	The number of data files in the data set D_i
$F(S)$	Fitness of solution S, $F(S) = \{\|D_1\|, \ldots, \|D_N\|\}$
F_l	The flexibility for the system
f	Objective function
G_i	The set of tasks that can be executed by node i
i, j	Worker node IDs, $i, j \in \{1, 2, \ldots, N\}$
k	Iteration step in simulated annealing
L_i	The set of all worker nodes available to execute task i
$\|L_i\|$	The number of worker nodes in set L_i
M	The number of data files to be processed
N	The number of worker nodes in the distributed system
P	The total number of feasible solutions
P_i	The processing capability of node i
P_r	Acceptance probability function
Q	The number of distinguishable tasks
r	The number of data replications in Hadoop (variable r)
S, S'	A feasible data allocation solution and its alternative
S_t	The Stirling number
U	The set of unscheduled comparison tasks
$\|U\|$	The number of comparison tasks in set U
G_i	The set of comp. tasks that can be executed by node i
T	The set of all comparison tasks
T_i	The set of comparison tasks assigned to node i
$\|T_i\|$	The number of tasks in the task set T_i
T_{data}	Time consumption for data distribution
T_{task}	Time consumption for comparison task execution
T_{total}	Total time spent on data distribution and task execution
t	Annealing temperature

storage and computing resources may differ significantly among the distributed computing nodes. Nevertheless, Hadoop is still the only viable distributed computing platform so far for big data processing without the need to copy all data to every node and is applicable to general all-to-all comparison problems. Therefore, our approach presented in this paper was evaluated in experiments against general Hadoop systems to show how much improvement our approach can achieve over Hadoop.

To address the challenges discussed above in existing general Hadoop systems, a data-aware scheduling approach is presented in this paper for all-to-all comparison problems involving big data in heterogeneous distributed systems. It is a substantial extension of our preliminary work with heterogeneous systems [34]. Given the features of heterogeneous distributed systems, the research presented below differs fundamentally from our previous studies: heterogeneous factors are considered in the problem formulation, which leads to a totally new optimization scheduling problem; and a metaheuristic method is chosen in the scheduling strategy development instead of the former greedy idea. In the task scheduling part, runtime dynamic task scheduling is firstly considered in this paper to achieve system dynamic load balancing. Moreover, more experiments running on a larger cluster were completed to compare with solutions based on different Hadoop settings.

3. Problem statement and challenges

An all-to-all comparison computation pairwise compares data items for a whole data set. An example is shown in Fig. 1 as a graph, where vertices represent data files to be compared and edges represent comparison tasks between two data files. For an all-to-all comparison problem with M data files, the total number of comparison tasks is $M(M-1)/2$. Thus, a graph with M vertices and hence $M(M-1)/2$ edges is denoted as a pair $(M, M(M-1)/2)$. All notations used in this paper are summarized in Table 1.

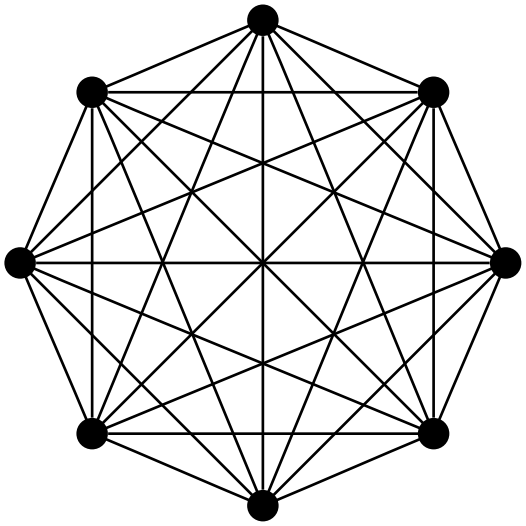

Fig. 1. Graph representation of an all-to-all comparison problem. The vertices and edges of the graph represent data files and pairwise comparison tasks, respectively. A graph with M vertices and thus $M(M-1)/2$ edges is denoted as a pair $(M, M(M-1)/2)$.

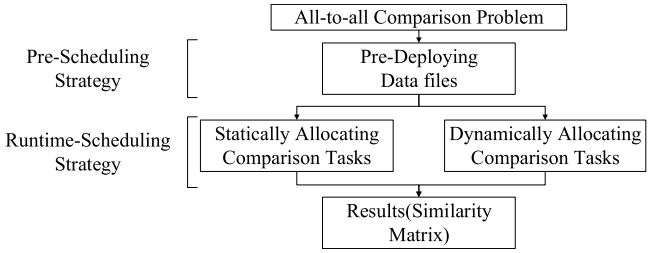

Fig. 2. A general design for data distribution and task scheduling.

3.1. Principles of processing the all-to-all scheduling problem

As noted above, existing all-to-all comparison approaches have considered data deployment and task scheduling in two separate phases. This leads to poor data locality and a consequent need for remote data access to complete comparison tasks. Efficient scheduling of comparison tasks for all-to-all comparison problems with big data sets requires us to avoid runtime data movement and also to consider the available computational resources. From this perspective, two basic principles are essential for scheduling all-to-all comparison tasks with big data sets:

(1) Task-oriented data distribution; and
(2) Comparison task scheduling with data locality.

Following these two principles, a general design of data distribution and task scheduling is depicted in Fig. 2. It consists of two main stages: pre-scheduling, and runtime static and dynamic scheduling. These major stages are followed by a trivial results-gathering step to produce the final matrix of pairwise comparison outcomes. For each of the two main stages, theoretical development and detailed design of our solutions are presented below.

3.2. Challenges of the all-to-all scheduling problem

When each worker node stores only part of the data set, a well-designed data distribution strategy is needed to meet the above two principles, especially when the size of the data set becomes large. However, the Hadoop framework deviates from these two principles when used for all-to-all comparisons. With a fixed number of data replications, it allocates data files randomly to

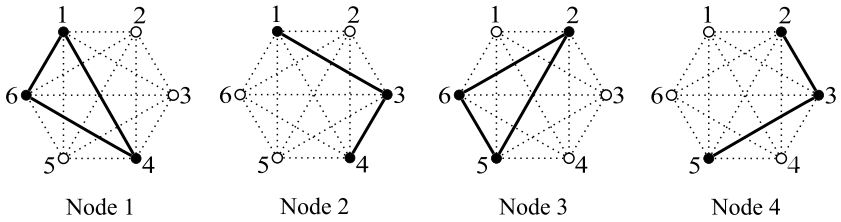

Fig. 3. A data distribution of 6 files on 4 nodes. Solid and dotted lines represent scheduled and unscheduled tasks, respectively. Solid and hollow points indicate scheduled and unscheduled data items, respectively.

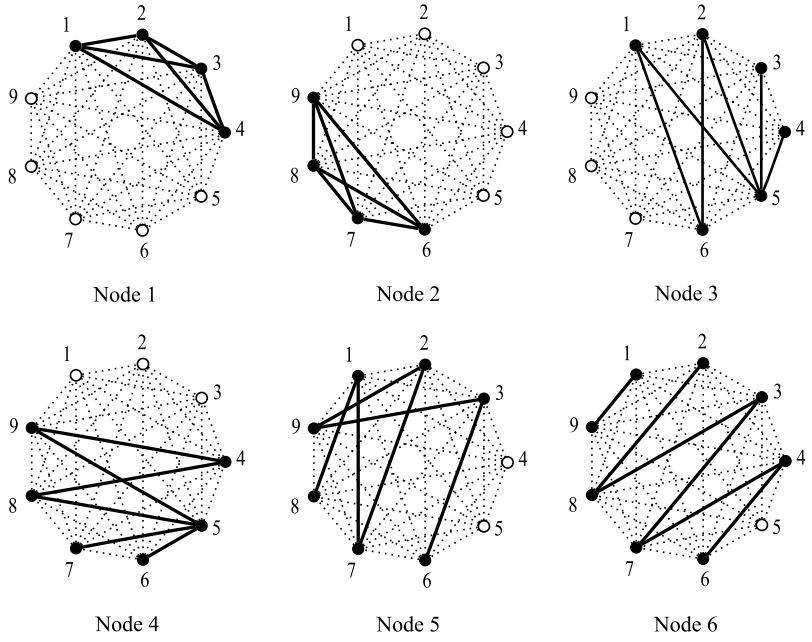

Fig. 4. A possible data distribution of 9 files on 6 nodes.

distributed worker nodes. Experiments reported by Qiu et al. [27] have demonstrated the inefficiency of the Hadoop framework for all-to-all comparison problems: (1) The computation iteratively switches between 'map' and 'communication' activities; and (2) Hadoop's simple scheduling model may not produce optimal scheduling of comparison tasks. These properties result in poor data locality and unbalanced task loads when using a Hadoop-like framework for all-to-all comparisons.

The characteristics of the all-to-all scheduling problem are summarized as the following four challenges.

Challenge 1: Distributing data evenly does not necessarily achieve load balancing. In order to achieve system-wide load balancing, an obvious approach is to evenly distribute data files to worker nodes. However, for all-to-all comparison problems, such a data distribution method does not promise that the data pairs needed for comparisons are available locally on the same worker node. For example, given a comparison problem with 6 data items the total workload can be expressed as a graph with the number of vertices and edges equal to (6, 15). Assuming we have 4 worker nodes, a possible task allocation using an even data distribution strategy is shown in Fig. 3. There are 2 copies of each of the 6 data items and each of the 4 nodes stores 3 data items. However, load balancing is not achieved as the nodes have different numbers of comparison tasks: Node 1 and Node 3 each have 3 tasks, but Nodes 2 and 4 each have only 2 tasks. Even worse, there are 5 comparison tasks that cannot even execute due to the lack of data locality. For instance, no node has copies of both data items 3 and 6.

Challenge 2: Task load balancing may cause a data imbalance. Another way of attempting to achieve load balancing is to schedule similar workloads to each of the worker nodes. While this promises to put a similar number of comparison tasks on each node, the separation of task allocation from data distribution can cause a severe data imbalance, resulting in large amounts of wasted storage, particularly when processing big data sets. Consider a scenario with 9 data items, which means that 36 comparisons are required. Fig. 4 shows how these 36 comparison tasks could be distributed across 6 worker nodes. For load balancing, each worker node is allocated 6 tasks. It can be seen that to achieve the necessary data locality, it is sufficient for Node 1 to store only 4 data items. However, Node 6 has to store 8 data items, double the number of Node 1, to avoid remote data accesses.

Challenge 3: Heterogeneous systems are harder to schedule than homogeneous ones. To make full use of the computing resources in heterogeneous distributed systems, the worker nodes should be allocated a task workload proportional to their respective processing power. However, many existing solutions, e.g., Hadoop, are designed with an implicit assumption of a homogeneous environment. For instance, consider a scenario with 6 data items, thereby requiring 15 comparisons, and 3 worker nodes, as shown in Fig. 5. A possible data task scheduling solution for a homogeneous environment, with balanced data distribution, is shown in the upper part of Fig. 5. In this case, each of the 3 worker nodes is allocated 5 comparison tasks to ensure good data locality, thus balancing data distribution and the number of comparison tasks. However, the same solution may cause a significant load imbalance in a heterogeneous system in which the worker nodes have different computing power. For instance, if Node 1 has three times the processing power of either Node 2 or 3, load balancing can only be achieved when Node 1 is assigned 9 tasks and each of the other two nodes is allocated 3 tasks. Unfortunately, with

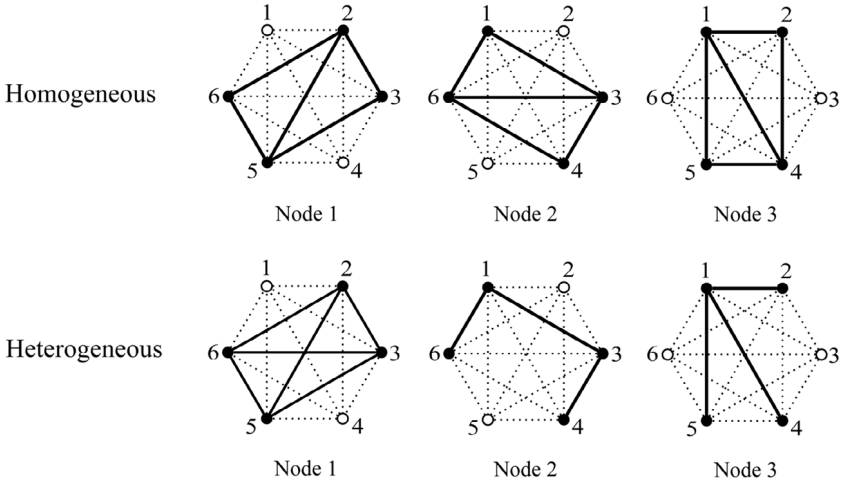

Fig. 5. Load balancing in a homogeneous system (upper part); and a potential load imbalance in a heterogeneous system for the same data distribution (lower part) under the assumption that Nodes 2 and 3 have the same computing power while Node 1 is three times as computationally powerful as either Node 2 or Node 3. In the heterogeneous case, not all tasks have been allocated yet.

the data distribution strategy suitable for a homogeneous system, there is no way to allocate enough tasks to Node 1 to prevent it from finishing its computation long before the other two nodes. An incomplete attempt at allocating the tasks in this scenario is shown in the lower part of Fig. 5. Not all of the 15 comparisons have been scheduled, but adding any further tasks to node 1 will unbalance the data distribution.

Challenge 4: The solution space for task scheduling is very large. The problem of scheduling comparison tasks and distributing related data to worker nodes can be treated as a classic problem in combinatorial mathematics, placing M objects into N boxes under certain constraints. The scheduling problem investigated in this paper has the following characteristics:

1. All comparison tasks are distinct. For all-to-all comparison problems, each of the comparison tasks is different and processes a different data pair.
2. All worker nodes are distinct in a heterogeneous distributed environment, and generally have different computing power and storage capacities.

These characteristics must be considered when designing task scheduling and data distribution strategies.

Consider a scheduling problem that allocates Q distinguishable comparison tasks to N distinguishable worker nodes. Each node is allocated at least one comparison task. From combinatorial mathematics, the total number of feasible solutions $P(Q, N)$ is expressed based on the Stirling number $S_t(Q, N)$ [10] as:

$$P(Q, N) = N! S_t(Q, N). \tag{1}$$

The Stirling number $S_t(Q, N)$ counts the number of ways to partition a set of Q distinguishable elements into N non-empty subsets:

$$S_t(Q, N) = \frac{1}{N!} \sum_{i=0}^{N} (-1)^i \binom{N}{i} (N - i)^Q, \tag{2}$$

where $\binom{N}{i}$ is a binomial coefficient.

Let us consider a special case of $N = 4$. From Eqs. (1) and (2), the number of all possible distribution solutions $P(Q, 4)$ is:

$$P(Q, 4) = 3 * 2^{Q+1} - 4 * 3^Q + 4^Q - 4. \tag{3}$$

$P(Q, 4)$ is illustrated in Fig. 6. It is seen from the figure that the number of possible solutions increases exponentially as the number of tasks increases. For 10 tasks and 4 nodes, $P(10, 4)$ is over

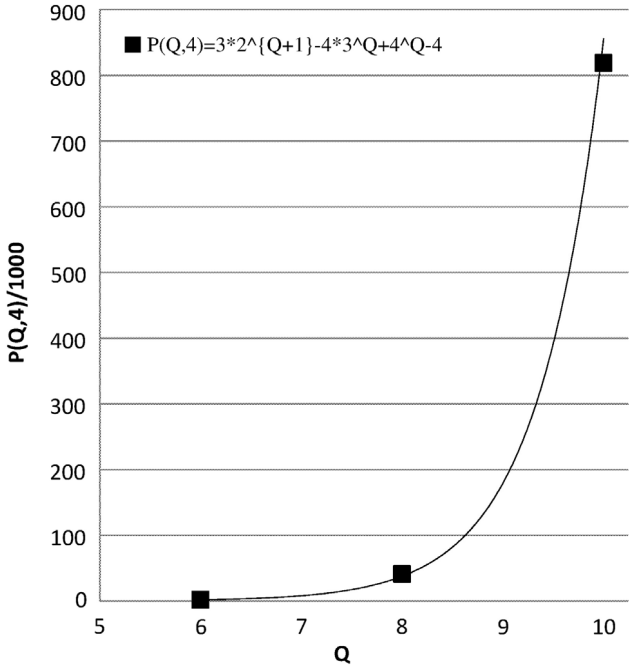

Fig. 6. The growth trend of the solution space for $P(Q, 4)$, for scenarios with 4 worker nodes and Q tasks to be distributed.

800,000. In real-life applications, this means that any non-trivial scenario has hundreds of thousands of possible solutions that would need to be evaluated in order to find an optimal one [33].

This growth trend, for even the trivial case of 4 workers, implies that it is generally impossible to evaluate all possible solutions to find the best answer in a reasonable period of time. Thus, developing heuristic solutions is the only viable approach for all-to-all scheduling problems in heterogeneous environments. This will be further discussed later after the problem is formalized in Section 4.

4. Formulation for pre-scheduling

This section formally describes the requirements for data distribution and comparison task scheduling. Then, it defines the overall objective and constraints from these requirements and formulates the pre-scheduling of data distribution as a constrained

optimization problem for all-to-all comparisons in heterogeneous distributed systems.

4.1. Overall considerations

From the two principles developed in Section 3.1, i.e., task-oriented data distribution and comparison task scheduling with data locality, the following three aspects are considered for task-oriented data distribution.

Time and storage consumption for distributing data. For large-scale all-to-all comparison problems, distributing data sets among all worker nodes should consider not only the storage used on each node within its storage capacity, but also the time consumed in distributing the data.

Execution performance of a single comparison task. Executing a comparison task requires access to, and processing of, the related pair of data files. Remote data access will delay a task's execution. Hence, data pre-scheduling should co-locate pairs of data files to be compared on a given node, implying maximization of 'data locality' for all comparison tasks.

Overall execution time performance of the all-to-all comparison problem. In a heterogeneous system, all worker nodes, each with different computing power, perform their own comparison tasks concurrently. The overall comparison problem is completed only when all worker nodes have completed their respective tasks. Therefore, making full use of the computing power for each of the worker nodes and allocating data and tasks to the worker nodes with a minimum load imbalance are critical to improving the overall performance of the computing problem.

4.2. Reducing time and storage consumption

Though time consumption for data distribution is affected by many factors, it is largely proportional to the total size of the data sets to be distributed for a given system. Let $|D_i|$ denote the number of files to be allocated to worker node i, and N represent the number of worker nodes in the system. As all-to-all comparison problems usually process data files with similar sizes, the time consumption T_{data} for distribution of all data files to N nodes can be approximated as:

$$T_{data} \propto \sum_{i=1}^{N} (|D_i|) . \tag{4}$$

For storage usage, each of the nodes should be assigned data files within its storage capacity. When the storage usage on each node is reduced, the total storage usage in the distributed system is also reduced.

Considering reducing both the data distribution time and data storage usage, a pre-scheduling of data distribution aims to minimize the amount of data stored on any node 1 to N, i.e., the constraint is to:

$$\text{Minimize } \max \{|D_1| , |D_2| , \ldots , |D_N|\} . \tag{5}$$

4.3. Improving performance for individual tasks

For all-to-all comparison problems, each of the comparison tasks running on the corresponding node has to access and process the required data items. The time spent on data processing is problem-specific. It depends on what the specific comparisons are, what algorithms are used and how the algorithm is implemented. The time spent on data access can be minimized to its lowest possible value when all required data items are made available locally.

Let us formulate the requirement for data locality. Let D_i and T_i respectively denote the data and task sets allocated to node i. Also, let $C(x, y)$ represent the comparison task for data items x and y. T denotes the set of all comparison tasks of the all-to-all comparison problem, and N is the number of nodes. Perfect data locality for all comparison tasks is then defined as a situation where each task is allocated to a node containing both data items it needs, i.e.:

$$\begin{aligned} &\forall C(x, y) \in T, \qquad \exists i \in \{1, \ldots , N\}, \\ &\quad x \in D_i \wedge y \in D_i \wedge C(x, y) \in T_i . \end{aligned} \tag{6}$$

4.4. Improving overall computational performance

The best possible overall performance of an all-to-all comparison problem is achieved if all nodes are allocated a workload proportional to their respective processing capabilities. This load balancing requirement means that all worker nodes will complete their respective set of comparison tasks at around the same time.

The requirement for load balancing is formalized as follows. Let $|T_i|$ denote the number of pairwise comparison tasks performed by node i, and P_i be the processing capability of node i. In this paper, the processing capability of node i is determined by both the processing power of node i and the execution time of comparison tasks on the node. The processing power and task execution time have a substantial correlation. Considering the execution time of different comparison tasks is mainly determined by the processing power of related worker nodes and also general all-to-all comparison problems are usually CPU-intensive [29], it is reasonable to use the hardware processing capability (i.e., the number of CPUs in this paper) of each worker node to represent the system and task heterogeneity. For an all-to-all comparison problem with M data items pairwise compared on N nodes, the total number of comparison tasks is $M(M-1)/2$. Load balancing requires the number of comparison tasks allocated to each node to be proportional to its processing power. This is achieved if the tasks are divided as evenly as possible among the nodes, taking into account the relative processing power of each node, i.e., for each node i

$$|T_i| \leq \left\lceil \frac{P_i}{\sum_{i=1}^{N} P_i} \cdot \frac{M(M-1)}{2} \right\rceil \tag{7}$$

where $\lceil \cdot \rceil$ is the ceiling function.

4.5. Optimization for data pre-scheduling

With the three requirements described above, the data pre-scheduling problem can be formalized below. For the overall system, we aim to achieve the objective in Eq. (5) while meeting the constraints in Eqs. (6) and (7). Minimizing the objective in Eq. (5) implies saving storage space and data distribution time, while meeting the constraints in Eqs. (6) and (7) means improving the computing performance of both individual comparison tasks and also the whole set of tasks.

Therefore, data pre-scheduling is formalized as the following constrained optimization problem.

$$\begin{aligned} &\text{Minimize } \max \{|D_1| , |D_2| , \ldots , |D_N|\} \\ &\text{such that Eqs. (6) and (7) are also met.} \end{aligned} \tag{8}$$

Although Eq. (8) only refers to the size of the data sets D_i, it is important that the solution also takes the different computational abilities of each worker node i into account. This is done by

requiring that Eq. (7) be satisfied simultaneously. Eq. (7) uses the relative processing powers P_i of each node to assign a different number of comparison tasks T_i, and hence a differently-sized data set D_i, to each worker node i.

Similar optimization problems have been tackled previously in other domains. Examples include block design [31] and graph coving [30]. In these optimization problems, Balanced Incomplete Block Design is an NP-hard problem [8] and the graph coving problem is NP-complete [30].

Similar to these known NP problems, the scheduling problem formalized here in Eq. (8) is a typical combinational optimization problem. It faces the challenge of a large number of combinations of data items and related comparison tasks. Finding the best solution from this large solution space is difficult, particularly for heterogeneous distributed systems. In the next section, a metaheuristic strategy is developed to address this challenge.

5. Metaheuristic data pre-scheduling

This section presents a metaheuristic data pre-scheduling based on an extended simulated annealing (SA) process. To improve the SA process's performance, specific methods are developed for generation and selection of solutions. The pre-scheduling strategy is then implemented as an algorithm.

5.1. Features of the optimization problem

The pre-scheduling optimization problem formulated in Eq. (8) has the following characteristics:

1. It is not difficult to find a feasible solution that meets both constraints but does not necessarily optimize the objective in Eq. (8). For a given all-to-all comparison problem as shown in Fig. 1, the total number of comparison tasks (edges) is finite and fixed. Any distribution of these tasks and related data files to nodes by following these two constraints gives a feasible solution for pre-scheduling.
2. The quality of a feasible solution can be evaluated quantitatively, by calculating and comparing the objectives of two candidate feasible solutions.
3. New solutions can be generated by changing the existing solutions. There are many ways to do so, but a simple strategy is to swap comparison tasks between two worker nodes.

From these features, solving the pre-scheduling optimization problem can begin with a randomly-generated initial solution. Then, we keep generating new feasible solutions and checking their improvement on the objective, retaining improved solutions and discarding others. This is achieved below through metaheuristics and simulated annealing.

5.2. Simulated annealing

Simulated annealing [17] has been widely used in solving combinatorial optimization problems. The SA process imitates the behaviour of physical systems which melt a substance and lower its temperature slowly until it reaches the freezing point.

Let S denote the solution space, i.e., the set of all possible solutions. The objective function is denoted by f, which is defined on members of S. SA aims to find a solution $S_i \in S$ that minimizes f over S. SA makes use of an iterative improvement procedure determined by neighbourhood generation. Firstly, an initial state is generated. Then, SA generates neighbourhood solutions at each temperature, which is gradually lowered, until a stopping criterion becomes true. The SA procedure is controlled by a group of parameters, which are called a cooling schedule.

Table 2
Cooling parameter settings (k represents the iteration step).

Item	Setting
Temperature decreasing function	$t_{k+1} = 0.99t_k$
Starting temperature	1.0
Ending temperature	10^{-5}
Inner loop iteration threshold	100

While the features of our pre-scheduling optimization problem justify the suitability of SA for solving the problem, the following two issues need to be addressed.

Determine the Annealing and Acceptance Probability modules. For an SA module, a suitable set of parameters of the cooling schedule is important for the efficiency and accuracy of the algorithm. The parameters to be set include the starting temperature, ending temperature, temperature reduction function and termination criterion. In addition, an acceptance probability function must also be chosen for SA to accept an undesirable intermediate state which may lead to a better global solution. In a local optimization algorithm, a new state is accepted when it optimizes the cost function. However, SA can accept an undesirable state based on the value of the chosen acceptance probability function.

Determine the neighbourhood selection method and fitness equation. As a local searching technique, at each temperature, SA generates a new neighbourhood solution randomly from the current solution. For each solution, a fitness equation is used to determine the quality of the solution. Both the neighbourhood selection method and fitness function need to be designed to be specific for our pre-scheduling problem.

5.3. Annealing design for all-to-all pre-scheduling

Here we address the two general simulated annealing requirements identified in Section 5.2 for the specific case of all-to-all comparison problems.

5.3.1. Designing the annealing module

To balance the accuracy of the final solution and the performance of the SA process, here we choose geometric cooling and determine the SA algorithm's parameters specifically for our pre-scheduling problem. Geometric cooling is one of the most widely used SA schedules [7]. At each temperature, a certain number of iterations are carried out to search for more solutions. Table 2 shows a choice of parameter settings, which are used later in algorithm design.

5.3.2. Acceptance probability function

The second SA module to be designed is the acceptance probability function. Let $P_r(\cdot)$ denote the acceptance probability function, ΔE represent the energy difference between the neighbouring and current states, and t be the temperature. We choose the Boltzmann function as the acceptance probability function [26,16]:

$$P_r(\Delta E) = \exp(-\Delta E/t). \tag{9}$$

The chance of accepting an energy-increasing move, i.e., a solution with poorer quality, decreases with the temperature. This enables SA to escape shallow local minima early on and explore deeper minima more fully.

5.3.3. Initial solution

A feasible solution to the pre-scheduling problem in Eq. (8) can be derived by following the constraint in Eq. (7) to allocate

comparison tasks and also by following the constraint in Eq. (6) to distribute data. We use the same notations explained before: M for the number of data files to be processed, N for the number of worker nodes, D_i for the data files stored on node i, and T_i for the task set allocated to node i. In addition, let U denote the set of tasks that have not yet been scheduled. An initial solution can be randomly generated as follows:

1. For each node $i \in \{1, \ldots, N\}$, randomly pick comparison tasks from U and assign them to T_i until $|T_i|$ meets Eq. (7) or $|U| = 0$. A scheduling of all comparison tasks is thus obtained: $T = \{T_1, \ldots, T_N\}$.
2. For each task set $T_i \in T$, schedule all required data files to node i. This forms data set D_i.

Once this is done, the set of task and data allocations to the nodes, $S = \{(T_1, D_1), (T_2, D_2), \ldots, (T_N, D_N)\}$, is a feasible solution, which meets both constraints in Eq. (8).

Such a randomly-generated initial solution S is not optimal in general. However, the SA process improves it step by step until a solution is obtained with acceptable quality.

5.3.4. Neighbourhood selection method

The solution neighbourhood defines the way to get from the current solution to another. Our basic idea for generating a new neighbouring solution is to move a comparison task from one node to another or to swap comparison tasks between two nodes. The generated neighbourhoods should cover the whole solution space. Therefore, the new neighbourhood solution S' is generated from current solution S by using the following neighbourhood selection method, which ensures enough randomness of the SA to reach an arbitrary point in the neighbouring solution space:

1. Randomly pick two different worker nodes i and j from all worker nodes $1, \ldots, N$;
2. Randomly swap two comparison tasks between nodes i and j, then update sets T_i and T_j; and
3. Re-schedule all required data files to nodes i and j based on sets T_i and T_j.

5.3.5. Fitness equation

A fitness equation determines the quality of candidate solutions. Our solutions are expressed as $S = \{(T_1, D_1), (T_2, D_2), \ldots, (T_N, D_N)\}$ (Section 5.3.3). From the solution's structure, the fitness equation for S can be defined as the set of numbers of data files allocated to each of the nodes. Let $F(S)$ denote the fitness of solution S:

$$F(S) = \{|D_1|, |D_2|, \ldots, |D_N|\}. \tag{10}$$

The cost difference between two alternative solutions S and S' is calculated as follows. Firstly, the data set sizes in both $F(S)$ and $F(S')$ are sorted in descending order. This means that a worker node that stores more data files is listed before a worker node that stores fewer data files. Therefore, the worker node with more data files has higher priority to be processed. Then, the vector cost difference ΔF is defined as:

$$\begin{aligned}\Delta F &= F(S) - F(S') \\ &= \left\{\left(|D_1| - \left|D'_1\right|\right), \ldots, \left(|D_N| - \left|D'_N\right|\right)\right\}. \end{aligned} \tag{11}$$

Considering that the target in Eq. (8) is a minimum–maximum optimization problem, it always prefers to reduce the data sizes from the worker nodes with more data files for an improved solution. Finally, the value of the scalar cost difference Δf is defined as:

$$\Delta f = \begin{cases} \text{the 1st non-zero size in } \Delta F, & \text{if any} \\ 0, & \text{otherwise.} \end{cases} \tag{12}$$

In this way, solutions with smaller differences from Eq. (11) are always accepted. The elements of ΔF shown in Eq. (11) are not sorted in descending order. This is because the evaluation of the quality difference between the two solutions $F(S)$ and $F(S')$ is determined by the first non-zero element, which represents the most significant difference between the two solutions. It is not determined by the maximum element in Eq. (11). Also, in comparison with other methods that only compare the maximum values in $F(s)$ and $F(S')$, our evaluation method makes full use of the information of all elements in $F(S)$ and $F(S')$. Thus, the resulting SA algorithm has a higher efficiency in moving from one solution to better ones.

5.4. Data pre-scheduling algorithm

Algorithm 1 summarizes all the features presented in Sections 5.3.3–5.3.5. Our solution uses SA as the underlying optimization technique with consideration of the specific requirements of all-to-all comparison problems. In the algorithm, the temperature decreases from its initial value (Line 3) to the stopping value (Line 4) by following the temperature decreasing function (Line 16). At each temperature, the algorithm generates a number of new solutions (Lines 5 and 7). For each new solution, the algorithm calculates its fitness and the change in the fitness level (Line 9), and then decides whether to accept or discard this new solution (Lines 11–13). At the end of the algorithm, the final solution is returned (Line 18).

Algorithm 1 Data Pre-Scheduling Algorithm

Initial:

1: Randomly generate an initial solution S using the method in Section 5.3.3;
2: Set parameters based on Table 2;
3: Set the current temperature t to be the starting temperature.

Pre-Scheduling:

4: **while** The current temperature t is higher than the ending temperature **do**
5: **while** The iteration step is below the inner loop iteration threshold **do**
6: (The loop threshold is defined in Table 2)
7: Generate a new solution S' from S: $S' \leftarrow new(S)$
8: (The method new is described in Section 5.3.4);
9: Calculate the change of Fitness, Δf, from Equation (12)
10: (Fitness methods for F, ΔF and Δf are given in Section 5.3.5);
11: **if** $\exp(-\Delta f/t) > random[0, 1)$ **then**
12: Accept the new Solution: $S \leftarrow S'$
13: **end if**
14: Increment the iteration step by 1;
15: **end while**
16: Lower the current temperature t based on the function in Table 2;
17: **end while**
18: Return final solution S.

6. Runtime task scheduling design

As per Fig. 2, two scheduling mechanisms can be used to schedule comparison tasks: static and dynamic. In static task scheduling, the tasks are distributed to worker nodes based on fixed information about the computing power of the nodes. Once allocated to a specific node, a task always executes on that node. Static scheduling simplifies the system design and minimizes the runtime communication costs.

In practice, however, the properties of computing nodes can change dynamically, especially in situations where the worker nodes are shared with other system users. In comparison with static scheduling, dynamic task scheduling makes decisions about task assignments at runtime, allowing the computation to adapt to changes in the computing environment, such as the processing power on a particular node being pre-empted by other system users.

This section defines both static and dynamic scheduling of large-scale all-to-all comparison problems.

6.1. Initial task schedulability

Given the pre-scheduling approach developed in Section 5 and the basic principles presented in Section 3.1, the runtime task scheduling problem can be defined as the need to find a worker node with all required data items for each comparison task.

Using the notations previously defined, let $C(x, y)$ represent the comparison task between data items x and y and D_i is the data set stored on worker node i. Then L_c is the set of all worker nodes available to execute comparison $C(x, y)$:

$$L_c = \{i \mid x \in D_i \wedge y \in D_i\}. \tag{13}$$

From Eq. (6), for each comparison task $C(x, y)$, node set L_c is guaranteed to be non-empty after distributing all data items using our pre-scheduling strategy, thereby ensuring the schedulability of all comparison tasks.

6.2. Static scheduling strategy and algorithm

Static scheduling is inherently considered in our pre-scheduling approach described in Section 5. Directly accepting the task assignment suggested by this pre-scheduling strategy, a static scheduling algorithm is formally described in Algorithm 2. In the algorithm, each worker node in the system keeps executing the comparison tasks that have been allocated to it when it has available computing resources.

Algorithm 2 Static Task Scheduling

Initial:

1: Task set T_i composed of all comparison tasks allocated to worker node i;

Static Scheduling:

2: **for** Each worker node i **do**
3: **while** There are unscheduled tasks in task set T_i and
4: enough available computing resources on node i **do**
5: Pick an unscheduled comparison task from T_i;
6: Assign this task to node i;
7: Mark this comparison task as scheduled;
8: Update the available computing resources on this node i;
9: **end while**
10: **end for**

In Algorithm 2, scheduling is conducted for each of the worker nodes (Line 2). While there are still unallocated tasks in the task set (Line 3) and the node has sufficient computing resources to accommodate more tasks (Line 4), we keep allocating tasks from the unallocated task set one after another to the node (Lines 5–8). Since the tasks have been pre-allocated to the nodes, the only scheduling decision made here is to choose the order in which to perform the comparisons on each node.

This static scheduling algorithm works well provided we have good data locality and (static) load balancing. It requires accurate information about the nodes, data files and comparison tasks. It also requires that the distributed system does not have major changes in its environment and thus does not exhibit much uncertainty. If any of these conditions is not met, runtime dynamic scheduling may be required to compensate, as explained below.

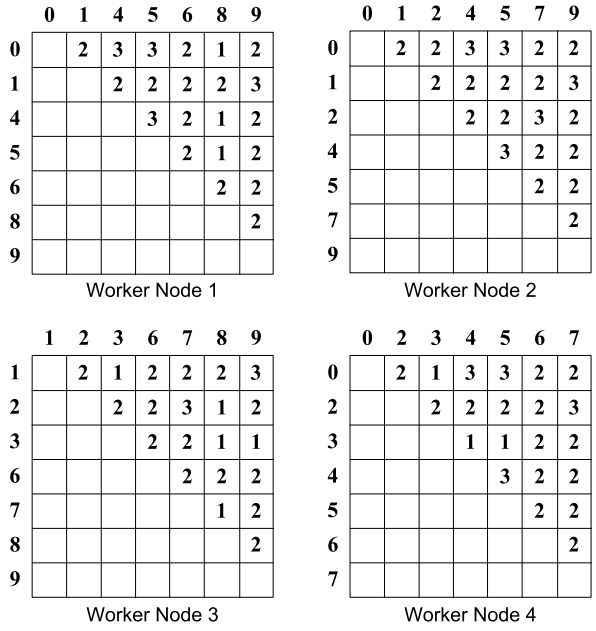

Fig. 7. All possible comparison tasks for each of the 4 worker nodes. The row and column numbers identify the data files stored on each node. The digit numbers in the cells represent how many worker nodes can perform the specific comparison between two such files.

6.3. Runtime dynamic scheduling

Our design for runtime dynamic task scheduling also follows the data pre-scheduling approach from Section 5. To give an insight into the dynamic scheduling strategy, consider the following simple example. Assume an all-to-all comparison problem with 10 data items (numbered from 0 to 9) is to be pairwise compared on 4 worker nodes. Fig. 7 shows a feasible data distribution solution. Nodes 1–4 are allocated data items (0, 1, 4, 5, 6, 8, 9), (0, 1, 2, 4, 5, 7, 9), (1, 2, 3, 6, 7, 8, 9) and (0, 2, 3, 4, 5, 6, 7), respectively.

Comparison tasks for each worker node are also shown in Fig. 7, in which the digit number in each cell represents how many worker nodes can perform that particular comparison task. For example, comparison task (0, 4) between data items 0 and 4 can be executed on Node 1, 2 or 4, so in each of the matrices corresponding to these nodes, the total 3 appears in the cell for this comparison. With such a range of options, dynamic task scheduling needs to decide to which node each specific comparison task is allocated.

6.3.1. Flexibility for dynamic scheduling

As shown in Fig. 7, the locations of all data items are fixed by our pre-scheduling, so there is only a limited number of worker nodes that can execute each comparison task. Let $|L(c)|$ denote the number of worker nodes that can execute comparison task c with data locality. Set U represents all comparison tasks that have not yet been scheduled in the system. We quantify the 'flexibility' of the schedule as:

$$F_\ell = \sum_{c \in U} |L(c)|\,. \tag{14}$$

The highest flexibility occurs when any comparison task can be executed on any worker node. In this extreme situation, the system is able to easily resolve any load imbalance at runtime. In contrast, if each of the comparison tasks can only execute on one worker node, the system has no flexibility to make any runtime adjustment for load balancing.

For all-to-all comparison problems, the initial system flexibility is determined by our data pre-scheduling before the computation begins. Thus, runtime dynamic task scheduling focuses on allocating comparison tasks that have not yet started to execute. The number of these tasks decreases as the computation progresses.

Let set G_i represent those comparison tasks that can be executed by worker node i and set U be all comparison tasks that have not yet been scheduled or have not started to execute in the system. When a comparison task c is scheduled to worker node i, the resulting change in the system's flexibility is denoted by ΔF_ℓ. We have:

$$\Delta F_\ell = |L(c)|, \quad \text{where } c \in U,\ c \in G_i. \tag{15}$$

How to manipulate ΔF_ℓ for dynamic task scheduling determines the scheduling priorities. Two typical examples are to maximize ΔF_ℓ and or to minimize ΔF_ℓ.

Maximizing Flexibility First (MaxFF). As shown in the upper part of Fig. 8, the comparison tasks with the most suitable worker nodes are scheduled in this strategy (maximum ΔF_ℓ). At the beginning, all worker nodes can get enough workload. However, as the computation progresses, the system's imbalance cannot be resolved. Though some worker nodes have idle computing resources (e.g., Node 2), the remaining comparison tasks cannot be scheduled to them because they cannot execute on those nodes.

Minimizing Flexibility First (MinFF). The lower part of Fig. 8 shows that the system can start with allocating and executing the comparison tasks with the least number of suitable worker nodes (minimum ΔF_ℓ). As the computation proceeds, the remaining comparison tasks have the flexibility to be assigned to more suitable worker nodes. This enables the system to have enough flexibility to deal with load imbalances at runtime.

6.3.2. Dynamic scheduling design

Dynamic scheduling aims to minimize the change in the system flexibility after scheduling each task while avoiding any remote data accesses. This is expressed as:

$$\text{Minimize } \Delta F_\ell. \tag{16}$$

The objective in Eq. (16) can be achieved by choosing a task c using the following rule:

$$|L(c)| = \min\{|L(i)| \mid i \in U \wedge i \in G_i\}. \tag{17}$$

Accordingly, our dynamic scheduling strategy is designed in Algorithm 3, which prefers those comparison tasks which can be executed on the smallest number of worker nodes.

Algorithm 3 Dynamic Task Scheduling

Initial:

1: Set U composed of all the unscheduled comparison tasks;
2: Set G_i composed of all comparison tasks that can be executed by each node i;
3: Compute $|L(c)|$ for each comparison task c in set U.

Dynamic Scheduling:

4: **while** There are unscheduled tasks in set U **do**
5: **if** Node i has enough available computing resources **then**
6: Choose a comparison task c from U, satisfying Equation (17);
7: Assign this task c to node i;
8: Mark this comparison task c as scheduled and update set U;
9: Update the available computing resources on this node i;
10: **end if**
11: **end while**

Table 3
Pre-scheduling of Scenario 1: Seven data files and five worker nodes, in which worker nodes A and B are twice as powerful as the other three.

Node	Distributed data files	Allocated comparison tasks
A	0, 1, 2, 3	(0, 1) (0, 2) (0, 3) (1, 2) (1, 3) (2, 3)
B	2, 4, 5, 6	(2, 4) (2, 5) (2, 6) (4, 5) (4, 6) (5, 6)
C	0, 3, 4, 6	(0, 4) (0, 6) (3, 6)
D	0, 1, 3, 5	(0, 5) (1, 5) (3, 5)
E	1, 3, 4, 6	(1, 4) (1, 6) (3, 4)

7. Analysis of our scheduling strategies

After developing the data pre-scheduling strategy and runtime task scheduling strategy, a complete solution for solving all-to-all comparison problems in heterogeneous distributed systems has been provided as we planned in Fig. 2. This section further analyses our solution by considering the challenges mentioned in Section 3. Consider a heterogeneous distributed system with five worker nodes, which are labelled from A to E. The following two scenarios are investigated for demonstration of our approach:

(1) Scenario 1: A data set of seven data files, numbered from 0 to 6, and where worker nodes A and B are twice as powerful as the other three; and
(2) Scenario 2: A data set of nine data files, numbered from 0 to 8, and where worker nodes A and B are three times as powerful as the other three.

Scenario 2 increases the level of heterogeneity from Scenario 1 in the distributed system.

7.1. Scenario 1: seven data files, and worker nodes A and B are twice as powerful as the other three

By applying Algorithm 1 to this scenario, a pre-scheduling solution is obtained as shown in Table 3. Table 3 shows that with task-oriented data distribution, our pre-scheduling algorithm not only suggests data allocations to the worker nodes but also proposes task assignments corresponding to the data allocations.

In this example, each worker node is allocated 4 data files, which indicates a good data balance in the heterogeneous system. Moreover, worker nodes A and B are each assigned 6 tasks, which is double the number of tasks given to nodes C, D and E. This is in accordance with the assumed computing power of the workers, and thus suggests a good load balance in the heterogeneous environment.

Hence, after the data files are distributed as shown in Table 3, scheduling of the all-to-all comparison tasks can also be completed by using our scheduling strategies shown in Algorithm 2 or 3. For the static task scheduling, each worker node will execute the comparison tasks exactly as shown in Table 3. While for dynamic task scheduling, the comparison tasks assigned to certain worker nodes are dynamically determined according to the runtime status of the system.

7.2. Scenario 2: nine data files, and worker nodes A and B are three times as powerful as the other three

With an increased level of heterogeneity from Scenario 1 in the distributed system, Scenario 2 considers nine data files distributed to five worker nodes, where worker nodes A and B are three times as powerful as the other three. Applying our Algorithm 1 gives the results shown in Table 4. Table 4 shows that both worker nodes A and B are assigned 12 tasks, which is three times more than the number of tasks given to nodes C, D and E. Moreover, worker nodes A and B are allocated five data files each, while each of the other three worker nodes is allocated four data files.

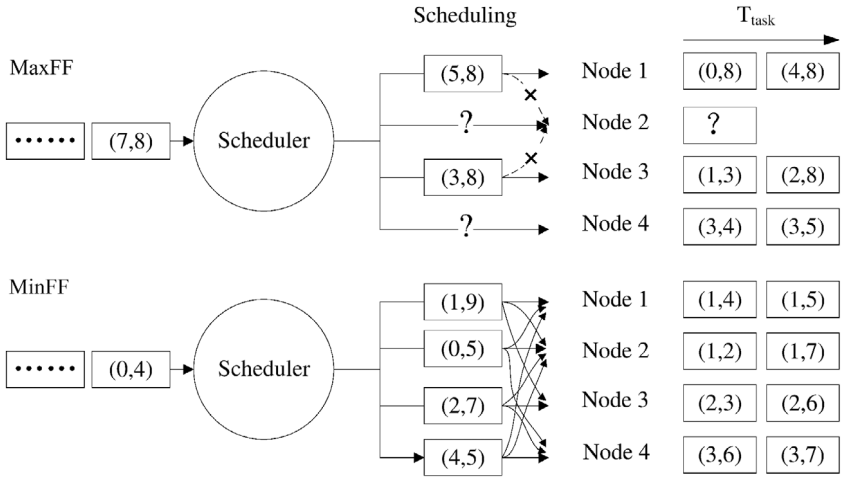

Fig. 8. MaxFF and MinFF dynamic scheduling strategies.

Table 4
Pre-scheduling of Scenario 2: Nine data files and five worker nodes, in which worker nodes *A* and *B* are three times as powerful as the other three.

Node	Distributed data files	Allocated comparison tasks
A	0, 1, 2, 3, 5, 7	(0, 1) (0, 2) (0, 3) (0, 7) (1, 2) (1, 3) (1, 5) (1, 7) (2, 5) (2, 7) (3, 7) (5, 7)
B	0, 3, 4, 5, 6, 8	(0, 4) (0, 5) (0, 6) (0, 8) (3, 5) (3, 6) (3, 8) (4, 5) (4, 6) (5, 6) (5, 8) (6, 8)
C	1, 2, 4, 6, 8	(1, 4) (1, 6) (2, 4) (2, 8)
D	2, 3, 4, 7, 8	(2, 3) (3, 4) (4, 7) (4, 8)
E	1, 2, 6, 7, 8	(1, 8) (2, 6) (6, 7) (7, 8)

7.3. Summary of the two scenarios

By using our solution, it can be seen from the two scenarios investigated above that both Challenges 1 and 2 from Section 3.2 are well addressed because of the consideration of data balancing and static load balancing in the data pre-scheduling strategy. Moreover, both our static and dynamic task scheduling strategies support heterogeneous distributed environments, so Challenge 3 is also addressed. By developing the pre-scheduling strategy based on the extended SA process, an arbitrary search for a suitable solution is avoided, so Challenge 4 is also addressed.

8. Experiments

This section evaluates the performance of our data-aware task scheduling approach for large-scale all-to-all comparison problems in heterogeneous systems. The performance evaluation is conducted from four main perspectives: storage saving, data locality, execution time and scalability.

Storage saving. Storage savings were measured in our experiments by using a group of simulations with different numbers of storage nodes in heterogeneous environments. The results from our approach were also compared with those from Hadoop's data distribution strategy.

Data locality. By meeting the constraint in Eq. (6), all data files are distributed to the worker nodes to allow local accessibility of data pairs for comparisons. The data locality performance of our pre-scheduling approach was compared with that of the widely used Hadoop data distribution strategy.

Execution time performance. Execution time performance was measured for distributing data and pairwise comparing data items, respectively. The time spent on comparison tasks was evaluated for individual tasks, individual worker nodes, and all worker nodes as a whole. Considering both time for data distribution and time for task execution, the total execution time of the all-to-all comparison problem could be evaluated. The results were compared with those from the Hadoop framework.

Scalability. Scalability is one of the main issues that have to be addressed in large-scale distributed computing. Experiments at different scales were carried out to evaluate the scalability of our data-aware task scheduling approach. Changes in the problem's scale included the numbers of input files and worker nodes.

8.1. Experimental settings

In this paper, the execution performance and scalability are measured by running real experiments. The following settings were used for the experiments.

Distributed computing system. A heterogeneous Linux cluster was built with 9 servers, which all run 64-bit Red Hat Enterprise Linux and use Intel(R) Xeon E5-2609. One node acted as the master, and the other eight were workers. Among the eight worker nodes, four had two sockets and 64 GB RAM memory, and the other four had a single socket and 32 GB RAM memory.

Prototype system and Hadoop system. Our prototype system is developed based on Mesos (version 0.9.0) and HDFS (version 1.2.1). By using Mesos's scheduling APIs and HDFS's data distribution plug-in, the data pre-scheduling strategy and runtime task scheduling strategy are both implemented. Beside this, Hadoop (version 1.2.1) with unchanged HDFS data strategy is used as the benchmark system. More specifically, the default block size is 64 MB and the default scheduler is FIFO [22].

Domain applications. As a typical all-to-all comparison problem, the bioinformatics CVTree problem [12] was chosen for our experiments. Three versions of CVTree applications were developed: CVTree with our prototype system, CVTree with Hadoop and CVTree with single machine. Though the three applications use different programming APIs, the comparison algorithms and data processing operations are the same.

Experimental data. A set of dsDNA files from the National Center for Biotechnology Information (NCBI) [25] was chosen as the input data for the CVTree problem. The size of each data file was around 150 MB, and over 20 GB of data in total were used in the experiments.

8.2. Storage saving and data locality

In our simulations, an all-to-all comparison problem with 256 data files was investigated. The distributed system was composed of multiple worker nodes with the number of nodes varying from 8 up to 64. These nodes were grouped into two halves: all nodes

Table 5
Data storage and data locality for $M = 256$ files and N varying from 8 to 64. Half of the nodes had twice the computing power of the other half. The number of data replications in Hadoop was set to be the default value 3. Comparison is also made with our previous work on heterogeneous systems [34].

N	$\max\{\|D_1\|, \ldots, \|D_N\|\}$			Storage saving (%)			Data locality (%)	
	This paper	[34]	Hadoop(3)	This paper	[34]	Hadoop(3)	This paper & [34]	Hadoop(3)
8	162	207	96	37	19	63	100	48
16	128	163	48	50	36	81	100	28
32	94	113	24	63	56	91	100	14
64	60	79	12	77	69	95	100	7

Table 6
T-test at a 5% significance level for the results of our approach for the experiments shown in Table 5 (mean values from 10 runs for all cases).

N	$\max\{\|D_1\|, \ldots, \|D_N\|\}$	Pass	C_i	Variance
8	162	Yes	[159.2392 162.9608]	6.7667
16	128	Yes	[127.9879 129.6121]	1.2889
32	94	Yes	[93.87054 95.52946]	1.3444
64	60	Yes	[59.76032 61.43968]	1.3778

within the same group have the same computing power, while the nodes in one group were twice more computationally powerful as those in the other group. For Hadoop's framework, each of the data blocks has r replications, which are stored on different worker nodes for reliability. The number, r, of data replications is set to be the default value 3 unless otherwise specified.

With the increase in the number of worker nodes, Table 5 shows the storage savings as a percentage for the data pre-scheduling from this paper, our previous work on heterogeneous systems [34] and Hadoop's data distribution strategy. It can be seen from Table 5 that all three strategies save much storage space (compared to copying all data to all nodes), implying a lower time cost on data distribution. This is important especially for big data problems with a large number of worker nodes. For instance, for a distributed system with 64 worker nodes, the storage saving reaches as high as over three quarters (77%) for our data pre-scheduling strategy. It is even as high as 95% for Hadoop.

However, Hadoop is designed without consideration of data locality for comparison tasks. Therefore, a large number of comparison tasks have to be executed with remote data access. This causes runtime data re-arrangement and significant computing performance degradation. For example, Table 5 shows that only 7% of comparison tasks have data locality in a system with 64 nodes, implying that 93% of the comparison tasks rely on remote data access to execute. In comparison, for our data pre-scheduling strategy, which meets the constraint in Eq. (6), all comparison tasks have perfect data locality for any number of worker nodes.

Moreover, compared to our previous work on heterogeneous systems [34], the approach in this paper achieves better results. For instance, $(79 - 60)/79 = 24\%$ more storage savings can be made in a system with 64 nodes, which means much less storage usage and predictable performance improvements.

Furthermore, quantitative t-test results using the R language show that for each of the four cases, the null hypothesis that the results come from a normal distribution with the mean value shown in Table 6 cannot be rejected at the significance level of 5%. The parameter C_i in Table 6 indicates the region into which 95% of the results will fall. It further demonstrates that 95% of the results in each case fall into a small region around the mean value, implying that consistent results can be obtained from our approach.

One might argue that storing more data copies in the distributed system using the Hadoop framework would solve the problem of the lack of data locality. However, the experimental results in Table 7 do not support this argument. Rather, they show the inefficiency of arbitrarily increasing and distributing data copies to the worker nodes. For Hadoop's data strategy, the number of data replications was manually tuned to achieve a similar maximum number of data files to our approach at each node. For a system with 64 worker nodes, even with a complicated tuning, Hadoop achieved only as low as 20% data locality. Even then the effort of manually tuning Hadoop is prohibitive and the flexibility of using the Hadoop framework is lost.

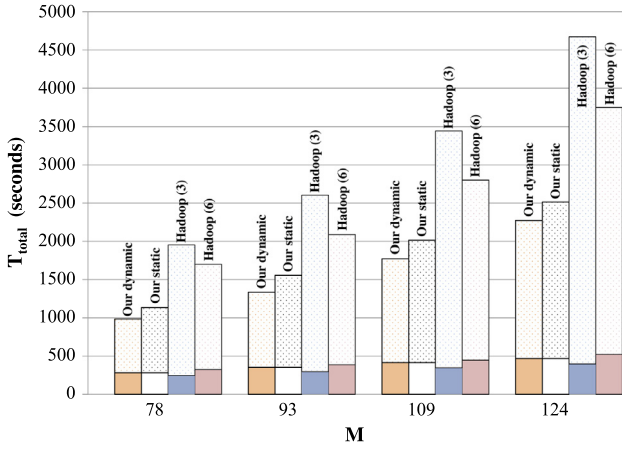

Fig. 9. Total execution time T_{total} performance for all four solutions. $T_{total} = T_{data} + T_{task}$. The lower and upper parts of the bars represent T_{data} and T_{task}, respectively.

8.3. Execution time performance

To demonstrate our data distribution and task scheduling strategies, four different solutions were designed and quantitatively compared through the CVTree problem:

1. The Hadoop(3) solution using Hadoop's data distribution (with three data replications) and task scheduling;
2. The Hadoop(6) solution with six data replications;
3. Our solution using our data pre-scheduling approach incorporating with static task scheduling; and
4. Our solution using our data pre-scheduling approach incorporating with dynamic task scheduling.

For the above solutions, the number of data items to be distributed to each worker node in the distributed system is listed in Table 8. In Table 8, by choosing a suitable value of replications in Hadoop (Hadoop(r)), we make our data pre-scheduling solution use storage space just between two different Hadoop solutions. This can help us check whether arbitrarily increasing the number of data duplications without considering data co-location is helpful in increasing overall execution performance, though the simulation results in Table 7 have already shown evidence in this regard (the poor data locality in the last column).

Fig. 9 compares the total execution time performance T_{total} from the four solutions to the CVTree problem. In Fig. 9, each bar chart is composed of two parts: the lower part represents the time T_{data} spent on data distribution, while the upper part represents the time T_{task} spent on comparison task execution. The height of the bar represents the total execution time T_{total} of the CVTree problem for the specific solution: $T_{total} = T_{data} + T_{task}$.

Table 7
Storage and data locality of our heterogeneous approach and Hadoop (variable r) for $M = 256$ files and N varying from 8 to 64. Half of the nodes had twice the computing power of the other half. The number of data replications r for Hadoop was tuned manually for each case to achieve a similar maximum number of files to our approach on each node.

N	Hadoop(r) data replications	$\max\{\|D_1\|, \ldots, \|D_N\|\}$		Storage saving (%)		Data locality (%)	
		This paper	Hadoop(r)	This paper	Hadoop(r)	This paper	Hadoop(r)
8	6	162	192	37	25	100	52
16	9	128	144	50	44	100	38
32	12	94	96	63	63	100	26
64	15	60	60	77	77	100	20

Table 8
Experimental scenarios for the CVTree problem with $N = 8$ and different M values.

M	$\max\{\|D_1\|, \ldots, \|D_8\|\}$		
	This paper	Hadoop(3)	Hadoop(6)
78	53	30	59
93	68	35	70
109	79	41	82
124	90	47	93

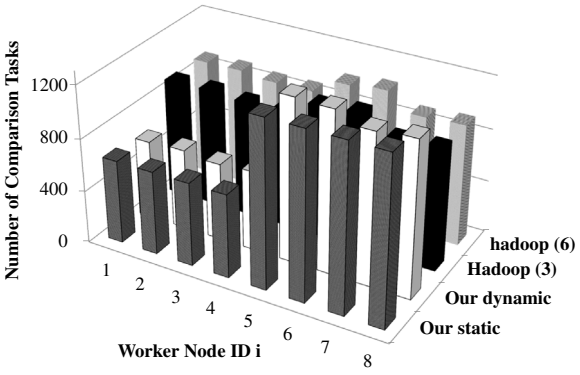

Fig. 10. The number of comparison tasks completed on each node. Half of the worker nodes (Nodes 5–8) are twice as computationally powerful as the other half (Nodes 1–4).

It is worth mentioning that for both our approach and Hadoop's, neither T_{data} nor T_{task} includes the times spent on data distribution scheduling and computation task scheduling, which can be executed independently. In particular, in comparison with the times for data distribution and task computation, the times for data distribution scheduling, computation task scheduling, and result collection are typically negligible for all-to-all comparison problems of big data sets. (They are also not measured in existing literature on Hadoop-based systems.)

Our observations from Fig. 9 are summarized as follows. The first observation from Fig. 9 is that both our static and dynamic solutions outperform the two Hadoop solutions significantly in terms of overall performance T_{total}. The second observation is that increasing the number of data replications from 3 to 6 in Hadoop improves T_{total} but at a cost of increased storage demands. However, the improved performance is still far behind our static and dynamic solutions. This is mainly due to our solutions' perfect data locality, which Hadoop's solutions do not have. The third observation from Fig. 9 is that our dynamic solution behaves with better T_{total} performance than the static one, as expected.

Fig. 10 shows the number of comparison tasks allocated to each worker node from all four solutions. As half of the eight nodes (Nodes 5–8) are twice as computationally powerful as the other half (Nodes 1–4), both our static and dynamic solutions allocate roughly twice as many comparisons to the four high-performance nodes as to the other four nodes. The static schedule allocates comparison tasks purely based on prior knowledge of the system resources, while the dynamic solution uses real-time information of the system's state. In comparison with our solutions, both

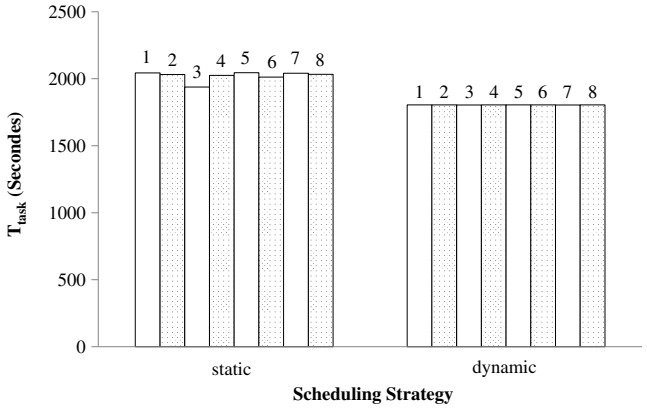

Fig. 11. Task performance T_{task} of our static and dynamic solutions.

the Hadoop(3) and Hadoop(6) solutions do not differentiate high-performance and low-performance nodes directly in task scheduling. As a result, the numbers of tasks allocated tasks by Hadoop is quite different from those from our solutions. The Hadoop system is originally designed for homogeneous systems. Thus, the number of comparison tasks scheduled to each worker node is roughly the same. This leads to poor load balancing and the need for remote data access. This is clearly shown in Fig. 10.

Comparisons of the task execution performance between our static and dynamic solutions are given in Fig. 11. It can be seen from this figure that for either the static or dynamic solution, all nodes complete their respective comparison tasks at a similar time with small deviations, implying good load balancing. But the deviations in task execution time are smaller in the dynamic solution than in the static solution. This means that the dynamic solution balances the load better than the static one. It can also be seen from Fig. 11 that the dynamic solution completes all comparison tasks around 10% faster than the static solution.

The dynamic solution completes all comparison tasks faster than the static one because it accounts for dynamic changes in the environment of the distributed system including changes in I/O speeds, fluctuations of the execution times of the comparison tasks, and other runtime effects. Such changes are not compensated for at all in the static schedule, which is computed prior to the execution of all comparison tasks. However, they are tolerated in dynamic scheduling, which is computed at runtime for each of the comparison tasks by making use of the real-time information of the distributed system.

8.4. Scalability

Scalability characterizes the ability of a system, network, or process to handle a growing amount of work in a capable manner or its ability to be enlarged to accommodate that growth [5]. It was evaluated in our experiments by using the speed-up metric.

The dotted line in Fig. 12 shows the ideal 1:1 linear speed-up which could be achieved if communication overheads, load imbalances and extra computation effort were not considered [20].

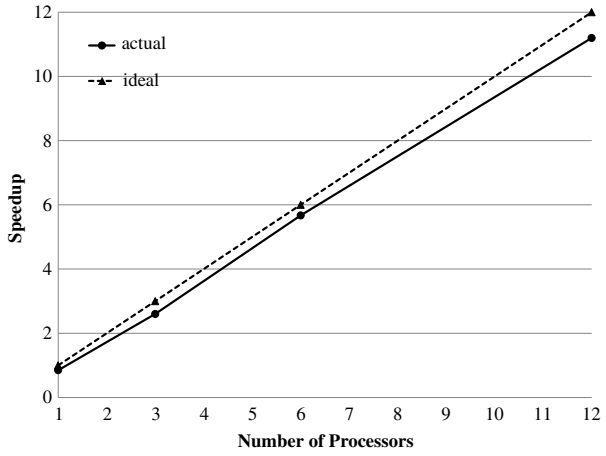

Fig. 12. Speed-up achieved by our dynamic scheduling algorithm for the CVTree problem compared to the theoretical ideal.

In practice, of course, no solution can achieve this. For the CVTree problem investigated in our experiments, the actual speed-up achieved by our data pre-scheduling and dynamic task scheduling is also depicted in Fig. 12. It increases linearly with the increase in the number of processors available for the computation of the problem, indicating good scalability of our approach in distributed computing of all-to-all comparison problems.

It is also worth mentioning that despite the inevitable overheads due to network communications, extra memory demands and disk accesses, our heterogeneous scheduling approach achieved about 93% of the ideal 1:1 linear speed-up performance. This was measured at 12 cores as $11.2/12 = 93\%$ from the results shown in Fig. 12.

9. Conclusion

A data-aware task scheduling approach has been presented for distributed computing of large-scale all-to-all comparison problems with big data sets in heterogeneous environments. It has been developed from a formalization of the requirements for storage saving, data locality and load balancing as a constrained optimization problem. To solve this optimization problem, metaheuristic scheduling has been used for task-oriented pre-scheduling of data distribution. Then, static and runtime dynamic scheduling strategies have been developed for allocation of comparison tasks to worker nodes. Experiments have shown that our data-aware task scheduling approach achieves advantages of storage savings, perfect data locality for comparison tasks, improved total execution time performance, and good scalability.

References

[1] J.H. Abawajy, M.M. Deris, Data replication approach with consistency guarantee for data grid, IEEE Trans. Comput. 63 (12) (2014) 2975–2987.
[2] S. Aluru, Y. Simmhan, Editorial: Scalable systems for big data management and analytics, J. Parallel Distrib. Comput. 79–80 (5) (2015) 1–2.
[3] R. Arora, M.R. Gupta, A. Kapila, M. Fazel, Similarity-based clustering by left-stochastic matrix factorization, J. Mach. Learn. Res. 14 (1) (2013) 1715–1746.
[4] M.D. Assunçao, R.N. Calheiros, S. Bianchi, M.A. Netto, R. Buyya, Big data computing and clouds: Trends and future directions, J. Parallel Distrib. Comput. 79–80 (2015) 3–15.
[5] A.B. Bondi, Characteristics of scalability and their impact on performance, in: Proceedings of the 2nd International Workshop on Software and Performance, 2000, pp. 195–203.
[6] Q. Chen, L. Wang, Z. Shang, MRGIS: A MapReduce-enabled high performance workflow system for GIS, in: IEEE Fourth International Conference on eScience (eScience '08), Indianapolis, IN, 2008, pp. 646–651.
[7] H. Cohn, M. Fielding, Simulated annealing: Searching for an optimal temperature schedule, SIAM J. Optim. 9 (3) (1999) 779–802.
[8] D. Corneil, R. Mathon, Algorithmic techniques for the generation and analysis of strongly regular graphs and other combinatorial configurations, Ann. Discrete Math. 2 (1978) 1–32. http://dx.doi.org/10.1016/S0167-5060(08)70319-4.
[9] A. Das, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, C. Yu, Transparent and flexible network management for big data processing in the cloud, in: Presented as part of the 5th USENIX Workshop on Hot Topics in Cloud Computing, USENIX, San Jose, CA, 2013, pp. 1–6.
[10] H.W. Gould, The q-Stirling numbers of first and second kinds, Duke Math. J. 28 (2) (1961) 281–289.
[11] S. Gunturu, X. Li, L. Yang, Load scheduling strategies for parallel DNA sequencing applications, in: Proceedings of the 11th IEEE International conference on High Performance Computing and Communications, IEEE, Seoul, 2009, pp. 124–131.
[12] B. Hao, J. Qi, B. Wang, Prokaryotic phylogeny based on complete genomes without sequence alignment, Mod. Phys. Lett. 2 (4) (2003) 14–15.
[13] M. Hess, A. Sczyrba, R. Egan, T.-W. Kim, Metagenomic discovery of biomass-degrading genes and genomes from cow rumen, Science 331 (6016) (2011) 463–467.
[14] C. Jayalath, J. Stephen, P. Eugster, From the cloud to the atmosphere: running MapReduce across data centers, IEEE Trans. Comput. 63 (1) (2014) 74–87.
[15] K. Kambatla, G. Kollias, V. Kumar, A. Grama, Trends in big data analytics, J. Parallel Distrib. Comput. 74 (7) (2014) 2561–2573.
[16] M. Keikha, Improved simulated annealing using momentum terms, in: 2011 Second International Conference on Intelligent Systems, Modelling and Simulation (ISMS), IEEE, Kuala Lumpur, 2011, pp. 44–48.
[17] S. Kirkpatrick, C.D. Gelatt, M.P. Vecchi, Optimization by simulated annealing, Science 220 (4598) (1983) 671–680.
[18] A.P.D. Krishnajith, W. Kelly, R. Hayward, Y.-C. Tian, Managing memory and reducing I/O cost for correlation matrix calculation in bioinformatics, in: Proc of the IEEE Symp on Computational Intelligence in Bioinformatics and Computational Biology, IEEE, Singapore, 2013, pp. 36–43.
[19] A.P.D. Krishnajith, W. Kelly, Y.-C. Tian, Optimizing I/O cost and managing memory for composition vector method based correlation matrix calculation in bioinformatics, Curr. Bioinform. 9 (3) (2014) 234–245.
[20] K. Li, Y. Pan, H. Shen, S. Zhang, A study of average-case speedup and scalability of parallel computations on static networks, Math. Comput. Modelling 29 (9) (1999) 83–94.
[21] K. Li, X. Tang, B. Veeravalli, K. Li, Scheduling precedence constrained stochastic tasks on heterogeneous cluster systems, IEEE Trans. Comput. 64 (1) (2015) 191–204.
[22] L. Liu, Y. Zhou, M. Liu, G. Xu, X. Chen, D. Fan, Q. Wang, Preemptive hadoop jobs scheduling under a deadline, in: Eighth International Conference on Semantics, Knowledge and Grids, IEEE, Beijing, 2012, pp. 72–79.
[23] E. Macedo, A. Melo, G. Pfitscher, A. Boukerche, Hybrid MPI/OpenMP strategy for biological multiple sequence alignment with DIALIGN-TX in heterogeneous multicore clusters, in: Proceedings of the IEEE International Paraeel and Distributed Workshops and PhD Forum (IPDPSW), IEEE, Shanghai, 2011, pp. 418–425.
[24] C. Moretti, H. Bui, K. Hollingsworth, B. Rich, P. Flynn, D. Thain, All-pairs: An abstraction for data-intensive computing on campus grids, IEEE Trans. Parallel Distrib. Syst. 21 (2010) 33–46. http://dx.doi.org/10.1109/TPDS.2009.49.
[25] NCBI, National Center for Biotechnology Information, 1988. http://www.ncbi.nlm.nih.gov/ (accessed: 3 July 2015).
[26] J. Pepper, B. Golden, E. Wasil, Solving the traveling sales man problem with annealing-based heuristics: A computational study, IEEE Trans. Syst. Man Cybern. A 32 (1) (2002) 72–77.
[27] X. Qiu, J. Ekanayake, S. Beason, T. Gunarathne, G. Fox, Cloud technologies for bioinformatics applications, in: Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers, MTAGS '09, Vol. 6, ACM, Portland, Oregon, USA, 2009, pp. 1–10.
[28] M.C. Schatz, Cloudburst: highly sensitive read mapping with MapReduce, Bioinformatics 25 (11) (2009) 1363–1369.
[29] H. Stockinger, M. Pagni, L. Cerutti, L. Falquet, Grid approach to embarrassingly parallel CPU-intensive bioinformatics problems, in: Second IEEE International Conference on e-Science and Grid Computing, IEEE, Amsterdam, The Netherlands, 2006, p. 58.
[30] S. Thite, On covering a graph optimally with induced subgraphs, 2008. URL: http://arxiv.org/pdf/cs/0604013.pdf.
[31] W.J. van der Linden, B.P. Veldkamp, J.E. Carlson, Optimizing balanced incomplete block designs for educational assessments, Appl. Psychol. Meas. 28 (5) (2004) 317–331.
[32] S. Xiao, H. Lin, W. Feng, Accelerating protein sequence search in a heterogeneous computing system, in: IEEE International Parallel and Distributed Processing Symposium, IEEE, Anchorage, AK, 2011, pp. 1212–1222.
[33] Y.-F. Zhang, Y.-C. Tian, C. Fidge, W. Kelly, A distributed computing framework for all-to-all comparison problems, in: Proceedings IECON 2014—40th Annual Conference of the IEEE Industrial Electronics Society, IEEE, Dallas, TX, USA, 2014, pp. 2499–2505.
[34] Y.-F. Zhang, Y.-C. Tian, W. Kelly, C. Fidge, Distributed computing of all-to-all comparison problems in heterogeneous systems, in: Proceedings IECON 2015—41th Annual Conference of the IEEE Industrial Electronics Society, IEEE, Yokohama, Japan, 2015, pp. 2053–2058.

Yi-Fan Zhang is a research assistant at the School of Electrical Engineering and Computer Science, Queensland University of Technology, Brisbane, Australia. His research interests include distributed computing, big data processing, and reliability of safety-critical software systems. Zhang has a Ph.D. in information technology from Queensland University of Technology, Brisbane, Australia.

Yu-Chu Tian is a professor at the School of Electrical Engineering and Computer Science, Queensland University of Technology, Brisbane, Australia. His research interests include big data processing, distributed computing, cloud computing, computer networks, and control theory and engineering. Tian has a Ph.D. in computer and software engineering from the University of Sydney, Australia. Contact him at y.tian@qut.edu.au.

Colin Fidge is a professor at the School of Electrical Engineering and Computer Science, Queensland University of Technology, Brisbane, Australia. His research interests include information security evaluation of computer software, security engineering of embedded Internet-enabled devices, and risk-aware business process modelling and analysis. Fidge has a Ph.D. in Computer Science from Australian National University, Canberra, Australia.

Wayne Kelly is a senior lecturer at the School of Electrical Engineering and Computer Science, Queensland University of Technology, Brisbane, Australia. His research interests include parallel computing, cluster computing, grid computing, high-performance computing, peer-to-peer (P2P) computing and Web services. Kelly has a Ph.D. in Computer Science from the University of Maryland System, Maryland, USA.