Int. J. Appl. Math. Comput. Sci., 2024, Vol. 34, No. 2, 263–275
DOI: 10.61822/amcs-2024-0019

amcs

ASSIGNMENT OF TASKS TO MACHINES UNDER DATA REPLICATION WITH A TIE TO STEINER SYSTEMS

PAWEŁ WOJCIECHOWSKI [a,b,*], MARTA KASPRZAK [a]

[a] Institute of Computing Science
Poznan University of Technology
Piotrowo 2, 60-965 Poznań, Poland
e-mail: {pwojciechowski,mkasprzak}@cs.put.poznan.pl

[b] Institute of Bioorganic Chemistry
Polish Academy of Sciences
Noskowskiego 12/14, 61-704 Poznań, Poland

In the paper a problem of assignment of tasks to machines is formulated and solved, where a criterion of data replication is used and a large size of data imposes additional constraints. This problem is met in practice when dealing with large genomic files or other types of vast data. The necessity of comparing all pairs of files within a big set of DNA sequencing results, which we collected, maintained, and analyzed within a national genomic project, brought us to the proposed results. This problem resembles that of generating a particular Steiner system, and a mechanism observed there is employed in one of our algorithms. Based on the problem complexity, we propose two heuristic algorithms, which work very well even for instances with tight constraints and a heterogeneous environment defined. In addition, we propose a simplified method, nevertheless capable of finding very good solutions and surpassing the algorithms in some special cases. The methods are validated in tests on a wide set of instances, where values of parameters reflect our real-world application and where their usefulness is proven.

Keywords: task assignment, genomic data processing, integer linear programming, Steiner system, heuristic algorithm.

1. Introduction

Analysis of genomic data, especially when dealing with the whole-genome type, requires significant computing resources and disk space. In the case of the human genome, files resulting from standard genome sequencing (NGS technology with 30x coverage) of one sample take up about 100 GB, while a file with results of analyses performed for that sample to detect genetic variants takes about 250 MB (Wojciechowski *et al.*, 2021). These are the sizes for compressed files. While working in a national genomic project, we encountered the problem of pairwise comparison of a significant number of such large files. In this case, it was a matter of finding potential relatedness between samples (in population studies, as a rule, samples of closely related individuals are removed from further analysis because they could introduce a bias into results). The issue is the allocation of files (samples) to machines so that similarity of each pair of files can be computed with limited disk resources of each available machine. Naturally, the computations should finish as quickly as possible and the data replication should be as small as possible, due to the above mentioned limited disk space and costs of data transfer.

This problem has much broader application. For example, building phylogenetic trees or using clustering algorithms based on genome sequences often requires determining symmetric distances between all pairs of genomes (see, e.g., Bogdanowicz and Giaro, 2013). For this purpose, approximate methods of similarity assessment based on similarity sets of k-mers are most often used (e.g., average nucleotide identity, the measure implemented in fastANI (Jain *et al.*, 2018)), but these are very simplified approaches (Ondov *et al.*, 2016). Another example is the problem of de novo DNA sequence assembly. In general, there are two main approaches

*Corresponding author

to solve the problem, one based on overlap graphs and the other on subgraphs of de Bruijn graphs (Blazewicz *et al*., 2018). The first strategy assumes, in its initial stage, the construction of a graph where vertices correspond to DNA sequences and edges correspond to overlaps between these sequences. In a standard experiment, there are generally millions of such sequences; thus, an exact comparison of all pairs is impossible. However, with new coming technologies, the number of sequences decreases and their length increases significantly (see, e.g., Berlin *et al*., 2015; Nurk *et al*., 2020), and the problem formulated here and associated algorithms could help also in such a kind of genome research. But the issues discussed in the paper have wider applicability, outside the area of bioinformatics, wherever a comparison (in any sense) of all pairs of large files from a given set is necessary.

According to our knowledge, this problem has not been solved in the literature. There is a lot of work related to task assignment in a distributed system (see, e.g., Majdzik, 2022), also mentioning data replication, but with a different formulation as well as uncomparable due to the requirements in our problem. For example, Gopalakrishnan and Caccamo (2006) used a criterion of data replication for task allocation to heterogeneous multiprocessors in a distributed system, but their tasks were periodic, associated with release times and deadlines, and solved multiple times on different machines. Briquet *et al.* (2007) scheduled tasks connected with transfer of large input data files and also considered data replication, but they assumed shared memory and a dynamically changing environment. Their scheduling pattern, called temporal tasks grouping, consists in grouping tasks needing the same input data and will not work for tasks processing different pairs of files. Nukarapu *et al.* (2011) proposed a scheduling problem in data-intensive distributed systems with data kept in many resource sites possibly outside machines. However, reduction in the time of access to files justifies data multiplication as long as there is free disk space in the sites. Povoa and Xavier (2018) worked on a static problem that combined both scheduling and replication problems in data-intensive distributed systems. Their model, as ours, assumes the same processing power and different storage capacity of machines but uses a different criterion function and is much more complex than required for our needs. None of the approaches was based on the assumption that tasks consist in processing all pairs of data packages given at the input.

The organization of the paper is as follows. In Section 2 the new problem is described and formulated in terms of integer linear programming. Section 3 provides additional analysis on constraints from the problem together with an approximate method used further for comparison. The problem complexity is analyzed in Section 4. In Section 5 two heuristic algorithms are proposed with proofs of their correctness and complexity. Results of a computational experiment are presented and discussed in Section 6. We conclude the paper in Section 7.

2. Problem formulation

In this real-world problem the goal is to process all defined tasks optimally according to a specified criterion, on a series of machines, such that the load balancing and file transfer are kept within given limits. The tasks altogether consist in comparing (in some sense) all pairs of large files given at the input, i.e., making computations from the whole upper triangular matrix, besides the diagonal, where rows and columns correspond to the files, and entries correspond to elementary computations. Differences in sizes of particular files are negligible; therefore, the file transfer can be expressed in the number of files. A task assigned to a machine consists in computing some pairs of files for a package of files sent to the machine (see Fig. 1 for an illustration).

The number of machines in the problem is specified and all are assumed, in principle, to be involved in computations. The balance of workloads must be kept within a range defined by a user. Because of substantial sizes of files, their transfer to machines and storage are critical in the problem. The optimization criterion concerns data replication, namely, the total amount of data transferred to machines is minimized. In addition, there are constraints for every machine on the data amount they can receive. It is assumed that, except for disk capacity, the machines are characterized by similar parameters regarding computations.

Data replication grows with the growing number of machines; on the other hand, the average share of a machine in the total amount of transferred data decreases. For example, for a certain number of files, processing all their pairs on one machine needs all files to be uploaded once, on three machines approximately twice, and on five machines approximately thrice, assuming relatively uniform workload. At the same time, one machine out of three gets on average ca. $2/3$ of the whole set of files, while one machine out of five ca. $3/5$ of the files.

With regard to minimization of data transfer, optimal use of a package of files sent to a machine means processing all pairs present there. But in different packages the same pairs of files inevitably will occur; therefore, with regard to load balancing, the upper triangular matrix should be partitioned into disjoint and possibly equal subareas in the number equal to that of machines. Such a subarea does not have to be in one part. If we did not consider data replication, each of m machines would get a subarea of around $1/m$ of the number of entries of the upper triangular matrix, i.e.,

around $n(n-1)/(2m)$ entries, where n is the number of files. Such a subarea, in the worst case (unattainable in practice), if it were composed of entries not involving common files, would require around $n(n-1)/m$ files to be sent to a machine. In the best case, the transfer would include around $\sqrt{n(n-1)/m}$ files and all pairs present there would be processed on that machine. In our problem, an optimal size of the transferred package will be in between these values, much closer to the second one.

This problem can be formulated as the following 0-1 integer linear programming (0-1 ILP) problem:

$$\min \sum_{k=1}^{m} \sum_{i=1}^{n} d_i^k$$

subject to

$$\sum_{k=1}^{m} x_{ij}^k = 1, \quad i = 1, \dots, n-1, \quad j = i+1, \dots, n, \tag{1}$$

$$s_i^k = \sum_{j=1}^{i-1} x_{ji}^k + \sum_{j=i+1}^{n} x_{ij}^k, \quad k = 1, \dots, m, \quad i = 1, \dots, n, \tag{2}$$

$$s_i^k - (n-1) d_i^k \le 0, \quad k = 1, \dots, m, \quad i = 1, \dots, n, \tag{3}$$

$$s_i^k - d_i^k \ge 0, \quad k = 1, \dots, m, \quad i = 1, \dots, n, \tag{4}$$

$$\sum_{i=1}^{n} d_i^k \le B_k, \quad k = 1, \dots, m, \tag{5}$$

$$l_k = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} x_{ij}^k, \quad k = 1, \dots, m, \tag{6}$$

$$|l_i - l_j| \le K, \quad i = 1, \dots, m-1, \quad j = i+1, \dots, m, \tag{7}$$

$$d_i^k \in \{0, 1\}, \quad k = 1, \dots, m, \quad i = 1, \dots, n, \tag{8}$$

$$x_{ij}^k \in \{0, 1\}, \quad i = 1, \dots, n-1, \quad j = i+1, \dots, n, \quad k = 1, \dots, m, \tag{9}$$

where the number of machines $m \ge 2$ and the number of files $n \ge 2$; x_{ij}^k are decision variables for all i, j, k; B is a vector of size equal to m containing, for every machine an upper bound on the size of a package of files sent to the machine; K is an admitted difference between sizes of tasks of particular machines expressed in the number of compared pairs of files. Values l_k, s_i^k, and d_i^k are derived from x_{ij}^k and mean, respectively, the number of pairs assigned to machine k, the number of pairs including file i assigned to machine k, and a binary variable stating whether file i is assigned to machine k. The minimized objective function divided by n gives the replication ratio of transferred data. The resulting assignment of tasks to machines is read from the decision variables x_{ij}^k, $i = 1, \dots, n-1$ and $j = i+1, \dots, n$, where 1 means that the pair of files i and j is compared on machine k.

3. Dealing with constraints

If the constraint (5) cannot be satisfied because of insufficient disk storage capacity of a machine or several machines, the solution may be to divide tasks into parts. The division can be realized simply by multiplying m in the above problem. The lengthened vector B is then filled with copies of the initial B. For example, replacing m by $2m$ and doubling B implies division of tasks into halves; the resulting assignment for machine k is read from x_{ij}^k (the first task) and x_{ij}^{k+m} (the second task) for the whole range of i and j. A drawback of such an approach is an increase in the total amount of transferred data. Another method may be to solve the problem with m equal to the number of machines and with selected positions of vector B set to arbitrarily large values, and next, for every machine k with changed B_k, to send files in portions resulting from partitioning the subarea of the matrix assigned to machine k. This approach does not enlarge the data transfer, because files once uploaded to a machine stay there as long they are needed for further computations. For example, a subarea of the matrix covering a set X of rows and a set Y of columns may be divided into disjoint parts with respect to $X = X_1 \cup X_2$; files are then transferred first in the portion $X_1 \cup Y$, and, after finishing computations and removing from the machine files from $X_1 \setminus Y$, the remaining files from $X_2 \setminus Y$ are uploaded. Sometimes the division must be done differently, when sets X and Y share a substantial number of files, because $X_1 \cup Y$ may not give a required reduction. For example, if $X = Y$, at least three parts of the subarea are necessary to reduce the maximal number of files transferred to a machine, $X_1 \cup X_2$, $X_2 \cup X_3$, and $X_1 \cup X_3$. But, actually, the case of $X = Y$ usually implies a lower number of files involved than the case of $X \cap Y = \emptyset$, and such a reduction may be unnecessary.

The approach with lengthening vector B can also be used for selected machines, those more computationally efficient than others, if such exist and we want to use them multiple times. The values B_i corresponding to these machines are then appended to B, and m is increased to the new length of B. It may be accompanied by an adjustment of the constraint on K.

With a very large number of files, in order to improve the efficiency of solving the above problem, we can change the granularity of objects taken at the input. We can treat a group of files as a single file, i.e., place it in one row/column of the matrix. If all rows and columns are associated with groups of a similar cardinality, each entry of the matrix brings similar load, as in the basic problem; however, some adjustment of the value of K may be necessary. We solve then the problem with

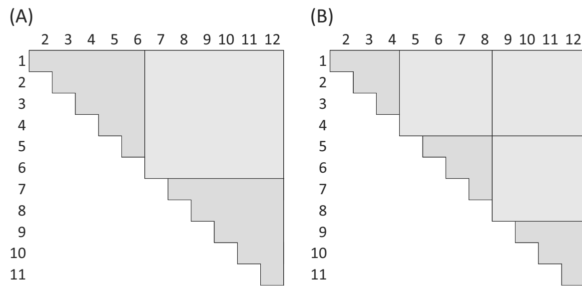

Fig. 1. Tasks from the problem altogether compose the upper triangular matrix, besides the diagonal, where rows and columns correspond to compared files. In a simplified approach the matrix can be divided as shown, where each subarea defines a task for a machine. The method `cell-adjust` divides the matrix into a predefined number of cells and assigns to one machine either one regular cell (in light grey) or one half-filled cell (in dark grey), or one and a half, depending on the granularity. Part (A) depicts the example for $m = 3$ machines, $n = 12$ files, and $p = 2$ groups of files; each machine will get one cell. Part (B) depicts the example for $m = 3$ machines, $n = 12$ files, and $p = 3$ groups of files; each machine will get one regular cell and one half-filled cell. A subarea of the matrix assigned to a machine (i.e., a task) may be composed of separated parts.

values from B reduced to the integer part due to dividing them by the cardinality of the largest group. However, in this scheme we get additional entries to process, the ones from the diagonal of the new matrix, where we need to compare all pairs of single files composing a group. Refer to Fig. 1, where matrices with single files corresponding to rows and columns are presented. If we group the files by six or four, we obtain new matrices with two or three rows/columns, respectively. In these compressed matrices, entries at the diagonal also bring some computation to be done. Let us call the areas of initial matrices that are compressed to entries of the new matrices "cells", "half-filled cells" if the compressed entries lie at the diagonal, and "regular cells" otherwise. The simplest way to compute such a half-filled cell, which does not increase the transferred amount of data, is to assign it to the machine that gets this group of files with a regular task.

If we decide to change the granularity of objects taken at the input, we can adjust their number to our purposes, which may be balancing workloads. To achieve it, one might settle the number of regular cells equal to m. In most cases it is not possible to adjust the number of cells exactly in this manner, but we have also these half-filled cells at the diagonal and it is the margin which guarantees relatively equable workloads with a deviation $\pm$ half of a cell. The matrix has $p(p-1)/2$ regular cells, where p is the number of groups of files (rows/columns after the compression), and p half-filled cells. We need to determine the value of p to keep m within the range $\langle p(p-1)/2; p(p+1)/2\rangle$. The boundaries of this interval are defined by two quadratic functions, which are distant always by a unit on p axis in the Cartesian plane. Thus m, where $m \geq p(p-1)/2$ and $m \leq p(p+1)/2$, for each its natural value, has a corresponding natural value of p inside this interval (at least one value, at most two). For two possible values of p, choosing the smaller one will cause machines to get tasks composed of either one regular cell or one half-filled cell; the greater value of p will cause machines to get either one regular cell or one and a half, but groups of files assigned to cells will be smaller then. For example (Fig. 1), for $m = 3$, one can choose $p = 2$ or 3; with $n = 12$ files (resulting in $n(n-1)/2 = 66$ pairs to compare) and $p = 2$, each machine will get either 36 pairs (one regular cell, groups of 6 files) or 15 pairs (one half-filled cell); for $p = 3$, each machine will get 22 pairs (1.5 of a cell, groups of 4 files). The replication ratio will be 2 for both values of p. For $m = 6$, one can choose $p = 3$ or 4; with 12 files and $p = 3$, each machine will get either 16 pairs to compare (one regular cell, groups of 4 files) or 6 pairs (one half-filled cell); for $p = 4$, each machine will get either 9 pairs (one regular cell, groups of 3 files) or 12 pairs (1.5 of a cell). The replication ratio will be 3 for both cases. Thus, the greater of two possible values of p seems to be a better choice.

This method gives very good approximate results for any number of machines and files. We treat its results as reference values in our comparison in Section 6, where we refer to it as `cell-adjust`.

4. Problem complexity

The problem in its general version is supposedly computationally hard. Its restricted variant, where the difference K between task sizes is zero, resembles the problem of generating one of possible Steiner systems S(2,k,n) with k taking values significantly greater than 2. In the problem, k is the size of a block (being in our problem the package of files sent to a machine, where the machine has to compute all pairs of uploaded files) and n is the number of elements (our set of all files). Even if we know that for given values of n and k the Steiner system can be generated (for many values it cannot be), its construction is not possible in time restricted by a polynomial of n if k is polynomially dependent on n. First of all, the input size is of order $\log n$, and the output is of size polynomially dependent on n. But even when measuring the complexity related to n, an algorithm for a Steiner system construction goes beyond polynomial dependence. Babai and Wilmes (2013) give a proof that a canonical form of Steiner system S(2,k,n) can be generated in a quasi-polynomial time of n. If, in our problem, the number of machines is bounded by a small constant not related polynomially to the

number of files, the decision problem does not belong to class NP, because the size of the output of order n^2 (when considering variables x^k_{ij}) or n (for variables d^k_i) is exponentially greater than the input size of order $\log n$; thus, verification of the correctness of a solution for the problem is exponential in time. An open question is the existence of an exact pseudopolynomial algorithm solving the problem (restricted by a polynomial function of m and n).

5. Algorithms

We propose two heuristic algorithms solving the general assignment problem (the 0-1 ILP problem). A greedy heuristic is presented in Section 5.1 together with its procedure `refineLoad()`. A heuristic based on a block design mechanism is in Section 5.2 together with an additional procedure `assignRange()`. Their performance is analyzed in Section 6.

5.1. Greedy heuristic. A feasible solution for the problem, if it exists, can be generated, e.g., by a greedy heuristic. However, we must keep in mind that it can fail even if a solution exists, especially if we must deal with tight bounds for K and B. The algorithm `greedy-opt`, given further on as a pseudocode, realizes a greedy heuristic approach extended by a refinement of a result. Without loss of generality we can assume $B_k \geq 2$ for every k, as machines not satisfying this can be simply removed from the instance, and that entries in B are sorted in non-increasing order.

The instruction `exit(1)` in the algorithm `greedy-opt` terminates the algorithm without a solution. Symbol $\oplus_m$ means addition modulo m. Lines 6–25 of the algorithm are for initializing assignments for particular machines such that they involve (if possible) different pairs of files. Lines 26–37 realize the greedy strategy, where every next entry of the matrix is assigned in the way that results in the least increase of the objective function. If more than one machine is optimal in this sense, the one is chosen that is safer from the point of view of the problem constraints. The procedure `refineLoad()` realizes a heuristic adjustment of workloads, especially for satisfying the constraint on K.

Algorithm 1. `greedy-opt`

Input: $m \geq 2$, $n \geq 2$, $K \geq 0$, $B_k \geq 2$ for $k = 1..m$
Output: x^k_{ij} for $i = 1..(n-1)$, $j = (i+1)..n$, $k = 1..m$

1: **if** $\sum_{k=1}^{m} B_k < n$ **then**
2:   `exit(1)`
3: **end if**
4: $x^k_{ij} \leftarrow 0$ for $i = 1..(n-1)$, $j = (i+1)..n$, $k = 1..m$
5: $t \leftarrow 0$
6: **for** $i = 1..n$ **do**
7:   **if** $i > 2m$ **then**
8:     **if** $B_{t+1} = \sum_{q=1}^{n} d^{t+1}_q$ **then**
9:       $t \leftarrow 0$
10:     **end if**
11:     $x^{t+1}_{2t+2,i} \leftarrow 1$
12:     $t \leftarrow t \oplus_m 1$
13:   **else if** i even **then**
14:     $x^{t+1}_{i-1,i} \leftarrow 1$
15:     $t \leftarrow t \oplus_m 1$
16:   **end if**
17: **end for**
18: **if** $n < 2m$ **then**
19:   $i \leftarrow n$
20:   **while** $i > n/2 + 1$ **and** $t < m$ **do**
21:     $x^{t+1}_{n-i+1,i} \leftarrow 1$
22:     $i \leftarrow i - 1$
23:     $t \leftarrow t + 1$
24:   **end while**
25: **end if**
26: **while** $\exists\, i, j\colon\ \sum_{q=1}^{m} x^q_{ij} = 0$ **do**
27:   **if** $\exists\, k\colon\ d^k_i = 1$ **and** $d^k_j = 1$ **then**
28:     $t \leftarrow$ such k of smallest l_k
29:   **else if** $\exists\, k\colon\ (d^k_i = 1$ **or** $d^k_j = 1)$ **and** $\sum_{q=1}^{n} d^k_q < B_k$ **then**
30:     $t \leftarrow$ such k of greatest $B_k - \sum_{q=1}^{n} d^k_q$
31:   **else if** $\exists\, k\colon\ \sum_{q=1}^{n} d^k_q < B_k - 1$ **then**
32:     $t \leftarrow$ such k of greatest $B_k - \sum_{q=1}^{n} d^k_q$
33:   **else**
34:     `exit(1)`
35:   **end if**
36:   $x^t_{ij} \leftarrow 1$
37: **end while**
38: `refineLoad()`
39: **if** $\mathtt{max}\,(l_k\colon\ k = 1..m) - \mathtt{min}\,(l_k\colon\ k = 1..m) > K$ **then**
40:   `exit(1)`
41: **end if**
42: **return**

Proposition 1. *The algorithm `greedy-opt` returns a feasible solution or none and has time complexity O(n^4m^2).*

Proof. First, the algorithm terminates for every input. Although it is obvious for the main part of the algorithm, the procedure `refineLoad()` may seem to be at risk of endless cycles of the same moves. In the first loop `while` of this procedure (l. 1–5), each iteration leads to removing a file from a machine; thus, there is always a progress toward improving the value of the objective function. In the second loop `while` (l. 10–37), in each iteration, adding an entry to a machine of a smallest l leads to increasing this value to $l + 1$, simultaneously the machine losing that entry does not fall with its load

Procedure 1. `refineLoad()`

1: **while** $\exists\, i,j,q,k:\ x_{ij}^q = 1$ **and** $(s_i^q = 1$ **or** $s_j^q = 1)$ **and** $d_i^k = 1$ **and** $d_j^k = 1$ **and** $q \neq k$ **do**
2: $t \leftarrow$ such k of smallest l_k
3: $x_{ij}^q \leftarrow 0$
4: $x_{ij}^t \leftarrow 1$
5: **end while**
6: $c \leftarrow 1$
7: **if** $\mathtt{max}\,(l_k:\ k = 1..m) - \mathtt{min}\,(l_k:\ k = 1..m) \leq K$ **then**
8: $c \leftarrow 0$
9: **end if**
10: **while** $c = 1$ **do**
11: $t \leftarrow k$ of smallest l_k
12: $u \leftarrow k$ of greatest l_k
13: **if** $\exists\, i,j,q:\ d_i^t = 1$ **and** $d_j^t = 1$ **and** $x_{ij}^q = 1$ **and** $l_q > l_t + 1$ **then**
14: $x_{ij}^q \leftarrow 0$
15: $x_{ij}^t \leftarrow 1$
16: **else if** $\exists\, i,j,q:\ x_{ij}^u = 1$ **and** $l_u > l_q + 1$ **and** $d_i^q = 1$ **and** $d_j^q = 1$ **then**
17: $x_{ij}^u \leftarrow 0$
18: $x_{ij}^q \leftarrow 1$
19: **else if** $\sum_{k=1}^n d_k^t < B_t$ **and** $\exists\, i,j,q:\ (d_i^t = 1$ **or** $d_j^t = 1)$ **and** $x_{ij}^q = 1$ **and** $l_q > l_t + 1$ **then**
20: $x_{ij}^q \leftarrow 0$
21: $x_{ij}^t \leftarrow 1$
22: **else if** $\exists\, i,j,q:\ x_{ij}^u = 1$ **and** $l_u > l_q + 1$ **and** $(d_i^q = 1$ **or** $d_j^q = 1)$ **and** $\sum_{k=1}^n d_k^q < B_q$ **then**
23: $x_{ij}^u \leftarrow 0$
24: $x_{ij}^q \leftarrow 1$
25: **else if** $\sum_{k=1}^n d_k^t < B_t - 1$ **and** $l_u > l_t + 1$ **and** $\exists\, i,j:\ x_{ij}^u = 1$ **then**
26: $x_{ij}^u \leftarrow 0$
27: $x_{ij}^t \leftarrow 1$
28: **else if** $\exists\, i,j,q:\ x_{ij}^u = 1$ **and** $l_u > l_q + 1$ **and** $\sum_{k=1}^n d_k^q < B_q - 1$ **then**
29: $x_{ij}^u \leftarrow 0$
30: $x_{ij}^q \leftarrow 1$
31: **else**
32: $c \leftarrow 0$
33: **end if**
34: **if** $\mathtt{max}\,(l_k:\ k = 1..m) - \mathtt{min}\,(l_k:\ k = 1..m) \leq K$ **then**
35: $c \leftarrow 0$
36: **end if**
37: **end while**
38: **return**

below $l+1$. The receiving machine may, in next iterations, further increase or decrease its load but never comes back to the initial smallest value of l. Similar reasoning is applied to the case of a machine with a greatest l. Hence, a series of iterations in the second loop `while`, although it may include receiving and losing the same entry of the matrix by the same machine, consequently leads to narrowing the range between extreme sizes of workloads.

A solution is feasible if all the constraints (1–9) are satisfied. For the constraint (9) this follows from data structures. The constraints (2)–(4), (6), and (8) define auxiliary variables on the basis of x_{ij}^k. The constraint (5) is kept throughout the algorithm; no new file can be added to a machine when there is no space for it. The constraint (7) can be disobeyed during computations, but the refinement step works as long as it is not fulfilled or exits without a solution. It remains to prove that each entry of the matrix is assigned to exactly one machine. The initial part of the algorithm (l. 6–25) assigns either at most one entry $[i,j]$ per any $j = 1..n$ or two entries $[i_1,j]$, $[i_2,j]$ per a j, where $i_1 = j-1$ and $i_2 < j-1$, i.e., $i_1 \neq i_2$. The rest of the algorithm completes the assignment, every entry being assigned exactly once due to the condition in the main loop `while` of the algorithm (l. 26–37). The refinement step (l. 38) can only replace one machine by another.

The main part of the algorithm (without l. 38) has time complexity O(n^2m), when a current size of a package of files assigned to a machine is kept in a variable. The procedure `refineLoad()` has complexity O(n^4m^2), which follows from the second loop `while` (l. 10–37) run at most O(n^2m) times. Therefore, the whole algorithm is of complexity O(n^4m^2). ■

The time complexity seems to be quite noticeable; however, such a worst case is rarely met. What is more, the elementary operations are very simple and the execution time of the algorithm can be short even for large n, see the experimental results.

5.2. Block design heuristic. The second heuristic algorithm is inspired by the Steiner system construction problem. A Steiner system described as S(2,k,n) is composed of k-element subsets (blocks) of an n-element set such that each pair of elements from the set belongs to exactly one block (see, e.g., Reid and Rosa, 2010). In relation to our problem, a pair of elements is a pair of files, and blocks could correspond to packages of files assigned to machines. A Steiner system would be then an optimal solution to the assignment problem for n files and the number of machines equal to the number of blocks in S(2,k,n), where every machine would get the same number of entries to compute provided that it could receive k files (see Appendix for an illustration). A Steiner system S(2,k,n) can be constructed only for some pairs of values of k and n. For example, for $k = 3$, the value of n must give 1 or 3 as the result of the division mod 6; for $k = 4$, the value of n mod 12 must be 1 or 4; for $k = 5$, the value of n mod 20 must be 1 or 5 (Grannell

Algorithm 2. `block-design`

Input: $m \geq 2, n \geq 13, K \geq 0, B_k \geq 2$ for $k = 1..m$
Output: x_{ij}^k for $i = 1..(n-1), j = (i+1)..n, k = 1..m$

1: **if** $\sum_{k=1}^{m} B_k < n$ **then**
2: `exit(1)`
3: **end if**
4: $x_{ij}^k \leftarrow 0$ for $i = 1..(n-1), j = (i+1)..n, k = 1..m$
5: $b \leftarrow \lfloor (n-1)/12 \rfloor$
6: $q \leftarrow (n-1) \bmod 12$
7: **for** $k = 1..m$ **do**
8: **if** $k \leq 13$ **then**
9: `assignRange(`k`)`
10: **else if** $q > 0$ **then**
11: $x_{12b+q,12b+q+1}^k \leftarrow 1$
12: $q \leftarrow q - 1$
13: **end if**
14: **end for**
15: **while** $\exists\, i, j\colon \sum_{q=1}^{m} x_{ij}^q = 0$ **do**
16: **if** $\exists\, k\colon d_i^k = 1$ **and** $d_j^k = 1$ **then**
17: $t \leftarrow$ such k of smallest l_k
18: **else if** $\exists\, k\colon (d_i^k = 1$ **or** $d_j^k = 1)$ **and** $\sum_{q=1}^{n} d_q^k < B_k$ **then**
19: $t \leftarrow$ such k of greatest $B_k - \sum_{q=1}^{n} d_q^k$
20: **else if** $\exists\, k\colon \sum_{q=1}^{n} d_q^k < B_k - 1$ **then**
21: $t \leftarrow$ such k of greatest $B_k - \sum_{q=1}^{n} d_q^k$
22: **else**
23: `exit(1)`
24: **end if**
25: $x_{ij}^t \leftarrow 1$
26: **end while**
27: `refineLoad()`
28: **if** `max(`$l_k\colon k = 1..m$`)` $-$ `min(`$l_k\colon k = 1..m$`)` $> K$ **then**
29: `exit(1)`
30: **end if**
31: **return**

and Griggs, 1994). For greater values of k, these cases are even sparser. In real applications of our problem, k may reach significant values dependent on n, and, for every n, a solution is expected. What is more, the number of blocks in a non-trivial Steiner system S(2,k,n) reaches the value of n or more, and we cannot expect such a big number of machines. A heuristic approach allows us to omit such limitations.

The algorithm `block-design` uses a mechanism of assignment of elements to blocks (files to machines) similar to the one observed in the Bose algorithm for construction of Steiner systems S(2,4,n) (Bose, 1939), adapted to our needs. Our heuristic partitions the upper triangular matrix (besides the diagonal) into blocks of similar size, which are assigned to machines provided that the constraints on B are satisfied. Due to separating

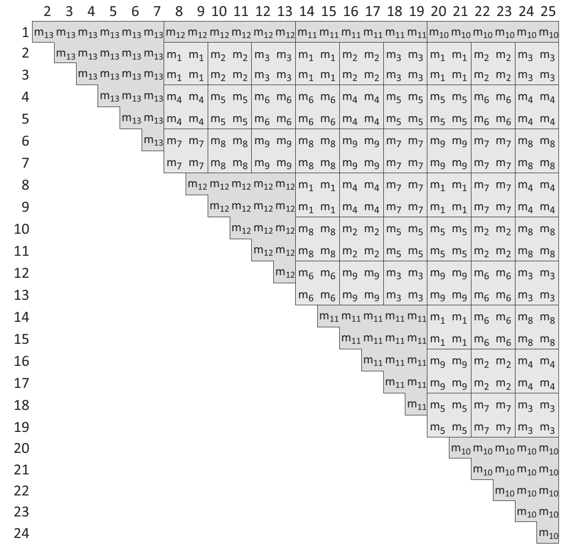

Fig. 2. Example assignment produced by the algorithm `block-design` for $m = 13$ and $n = 25$. Entry m_k at position $[i, j]$ means $x_{ij}^k = 1$, i.e., the pair of files i and j will be compared on machine k (assumed that some defined constraints on K and B are satisfied). The squares distinguished in the matrix are of size equal to $b = \lfloor (n-1)/12 \rfloor$, here 2. If n and m were equal to 13, the assignment would reflect the blocks generated by the Bose algorithm for the Steiner system S(2,4,13) after appropriate mapping of its elements to rows and columns of the matrix (see Appendix).

13 such blocks, the algorithm performs best for a similar number of machines or lower. It is assumed here that the number of files cannot be smaller than 13, and that entries in B are sorted in non-increasing order.

The instruction `exit(1)` in the algorithm `block-design` terminates the algorithm without a solution. The procedure `refineLoad()` realizes a heuristic adjustment of workloads and it is the same procedure as in the previous subsection. The procedure `assignRange()`, given below, assigns entries of the matrix according to the mechanism of S(2,4,13). The instruction `exit(0)` in the procedure `assignRange()` means the return to the main algorithm without an error; see Fig. 2 for an example.

Proposition 2. *The algorithm* `block-design` *returns a feasible solution or none and has time complexity O(n^4m^2).*

Proof. A solution is feasible if all the constraints (1)–(9) are satisfied. The reasoning for the constraints (2)–(9) is the same as in the proof of Proposition 1. It remains to prove that each entry of the matrix is assigned to exactly one machine. The procedure `assignRange(`t`)`, for different values of $t \leq 9$, assigns entries from disjoint subareas, regardless of the offset (p_1, p_2), because each

t gives a different pair of values (w_1, w_2). For the same (p_1, p_2) and two different pairs (w_1, w_2) and $(w_{1'}, w_{2'})$, w_1 differentiates the ranges if $w_2 = w_{2'}$ and vice versa, due to the component $w_1 \oplus_3 aw_2$ in both dimensions ($a = p_1 - 2$ or $a = p_2 - 2$), which can be easily checked by enumeration. For $9 < t \leq 13$, every new subarea considered is disjoint from the previous ones, because it is located outside the area assigned when $t \leq 9$, and for $t > 9$, the ranges related to columns are disjoint for different values of t. In the main algorithm, for remaining machines (if any), single entries outside the previous subareas are assigned and they are all different (l. 10–12). The rest of the algorithm completes the assignment, every entry being assigned exactly once due to the condition in the main loop `while` of the algorithm (l. 15–26). The refinement step (l. 27) can only replace one machine by another.

The procedure `assignRange()` has time complexity O(n^2); altogether with the main part of the algorithm (without l. 27) we have O(n^2m), when a current size of a package of files assigned to a machine is kept in a variable. The procedure `refineLoad()`, identical as in the algorithm `greedy-opt`, has complexity O(n^4m^2); thus, the whole algorithm is of complexity O(n^4m^2). ■

The proposed algorithms have the same worst-case complexity resulting from the final refinement procedure. However, both algorithms in many cases will distribute entries among machines nearly evenly. As a result, at the beginning of the procedure `refineLoad()`, the constraint on K is often satisfied and the most time consuming part is not executed.

6. Results

The computational experiment was performed on a PC workstation with a single CPU, AMD Ryzen 5900 3 GHz (12 cores/24 threads), 32 GB RAM. The program execution was single-threaded. We used in the tests the following values of parameters. The number of files n takes values from the range $\langle 500; 5000 \rangle$ with step 500 and the number of machines m from $\langle 2; 20 \rangle$ with step 2, and additionally 13. Vector B is filled with values dependent on n and m just to give enough space to find a solution but not much else. We used a formula based on the lower bound $\sqrt{n(n-1)/m}$ discussed in Section 2, here with a safe margin added. For identical machines (in terms of capacity), $B_k = \lfloor n/\sqrt{m} + 0.3 \cdot n \rfloor$, $k = 1, \dots, m$, if $m > 2$, and $B_k = n$ otherwise. For a set of non-identical machines (where $m \in \{8, 10, 12\}$), some of the values are modified, $B_1 = B_2 = \lfloor 1.2 \cdot B_3 \rfloor$ and $B_m = B_{m-1} = B_{m-2} = B_{m-3} = \lfloor 0.8 \cdot B_3 \rfloor$. The accepted difference between task sizes K is set to a percentage (depending on the test case) from the average number of entries per machine, i.e., from $n(n-1)/(2m)$.

Procedure 2. `assignRange(t)`

1: **if** $t \leq 9$ **then**
2: $w_1 \leftarrow (t-1) \bmod 3$
3: $w_2 \leftarrow \lfloor (t-1)/3 \rfloor$
4: **for** $p_1 = 1..3$ **do**
5: **for** $p_2 = (p_1+1)..4$ **do**
6: **if** $p_1 > 1$ **then**
7: $z_1 \leftarrow 3b(p_1-1) + b(w_1 \oplus_3 w_2(p_1-2)) + 2$
8: **else**
9: $z_1 \leftarrow bw_2 + 2$
10: **end if**
11: $z_2 \leftarrow z_1 + b - 1$
12: $z_3 \leftarrow 3b(p_2-1) + b(w_1 \oplus_3 w_2(p_2-2)) + 2$
13: $z_4 \leftarrow z_3 + b - 1$
14: **for** $i = z_1..z_2$ **do**
15: **for** $j = z_3..z_4$ **do**
16: **if** $\sum_{k=1, k \notin \{i,j\}}^{n} d_k^t > B_t - 2$ **then**
17: `exit(0)`
18: **else**
19: $x_{ij}^t \leftarrow 1$
20: **end if**
21: **end for**
22: **end for**
23: **end for**
24: **end for**
25: **else**
26: $z_1 \leftarrow 3b(13-t) + 2$
27: $z_2 \leftarrow z_1 + 3b - 1$
28: **for** $i = z_1..z_2$ **do**
29: **if** $\sum_{k=2, k \neq i}^{n} d_k^t > B_t - 2$ **then**
30: `exit(0)`
31: **else**
32: $x_{1i}^t \leftarrow 1$
33: **end if**
34: **end for**
35: **for** $i = z_1..(z_2-1)$ **do**
36: **for** $j = (i+1)..z_2$ **do**
37: **if** $\sum_{k=1, k \notin \{i,j\}}^{n} d_k^t > B_t - 2$ **then**
38: `exit(0)`
39: **else**
40: $x_{ij}^t \leftarrow 1$
41: **end if**
42: **end for**
43: **end for**
44: **end if**
45: **return**

Let us begin with results of the approximate method `cell-adjust` from the end of Section 3; they are presented in Tables 1 and 2. The method partitions the matrix without referring to B and K but allows assigning near the same number of files to each machine for most cases (for other cases, when $m = 13$, 14, or 20, the maximal difference between file package sizes

Table 1. Total number of files sent to identical machines according to the method `cell-adjust`.

m	p	n									
		500	1000	1500	2000	2500	3000	3500	4000	4500	5000
2	2	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
4	3	1334	2667	4000	5334	6667	8000	9334	10667	12000	13334
6	4	1500	3000	4500	6000	7500	9000	10500	12000	13500	15000
8	4	2000	4000	6000	8000	10000	12000	14000	16000	18000	20000
10	5	2000	4000	6000	8000	10000	12000	14000	16000	18000	20000
12	5	2400	4800	7200	9600	12000	14400	16800	19200	21600	24000
13	5	2500	5000	7500	10000	12500	15000	17500	20000	22500	25000
14	5	2500	5000	7500	10000	12500	15000	17500	20000	22500	25000
16	6	2668	5336	8000	10668	13336	16000	18668	21336	24000	26668
18	6	3000	6000	9000	12000	15000	18000	21000	24000	27000	30000
20	6	3000	6000	9000	12000	15000	18000	21000	24000	27000	30000

Table 2. Maximal difference between machines' workloads according to the method `cell-adjust`, for identical machines, measured in the number of the matrix entries.

m	n									
	500	1000	1500	2000	2500	3000	3500	4000	4500	5000
2	250	500	750	1000	1250	1500	1750	2000	2250	2500
4	13528	55278	125250	220778	346528	500500	678028	887778	1125750	1385278
6	7750	31125	70125	124750	195000	280875	382375	499500	632250	780625
8	125	250	375	500	625	750	875	1000	1125	1250
10	4950	19900	44850	79800	124750	179700	244650	319600	404550	499500
12	5050	20100	45150	80200	125250	180300	245350	320400	405450	500500
13	5050	20100	45150	80200	125250	180300	245350	320400	405450	500500
14	5050	20100	45150	80200	125250	180300	245350	320400	405450	500500
16	3403	13695	31375	55278	86320	125250	169653	221445	281625	346528
18	249	830	250	999	2080	500	1749	3330	750	2499
20	3652	14525	31375	56277	88400	125250	171402	224775	281625	349027

for machines is up to 20% of n). We can observe almost no difference between machines' workloads when the number of machines is equal to $p^2/2$ for some p, but the assignment is not such even for other cases. The replication ratio of transferred data (being the objective function value from Table 1 divided by n) for `cell-adjust` depends only on the number of machines—it is a property of this method.

These values characterize very good solutions, yet not feasible in some settings. For all the cases, except for $m = 2$, 8, or 18, K should be set to at least 40% of the average machine load to make the solutions feasible. The two proposed algorithms work much better in this sense, even when restrictions in the problem are very strict. The algorithm `greedy-opt` reaches from 94% to 164% of the value of the objective function returned by `cell-adjust` but is able to fit in very tight limits imposed for capacity of machines and balance of workloads. Actually, it returned, in each test case, allocations of files and tasks of almost the same sizes for all machines. The maximal difference between file package sizes for machines within a test case is not greater than 3. The maximal difference between numbers of entries assigned to machines is shown in Table 3, with the average value equal to 760, where K was set to 1% of the average number of entries per machine. The maximal computation time for these tests is 4 s.

More information about the quality of results generated by `greedy-opt` is provided in Fig. 3, and also in Table 5 and Fig. 4, where tests for machines differing in capacity are summarized. In Table 4, the quality of results of the algorithm `block-design` is presented, for sets of identical machines.

The algorithm `block-design` supported by the procedure `refineLoad()` works for any set of problem parameters but needs a lot of time when K is set to small values and m is greater than 13. For this reason, K in these tests was set to 40% of the average number

Table 3. Maximal difference between machines' workloads obtained by the algorithm `greedy-opt`, for identical machines, measured in the number of the matrix entries.

m	n									
	500	1000	1500	2000	2500	3000	3500	4000	4500	5000
2	252	502	752	1002	1252	1502	1752	2002	2252	2502
4	250	872	750	1747	1250	2622	1750	3497	2250	4372
6	178	71	751	713	178	1497	1251	284	2248	1785
8	57	88	802	458	1003	682	2362	2253	1864	1423
10	37	373	444	576	472	567	677	836	932	976
12	103	96	82	217	550	443	497	657	341	788
13	78	210	237	454	375	633	607	862	697	1186
14	83	167	291	304	440	721	310	1186	1405	282
16	71	167	191	233	321	308	760	451	596	713
18	33	157	103	304	358	691	492	567	325	956
20	62	97	51	216	224	344	416	279	346	549

Table 4. Total number of files sent to identical machines according to the algorithm `block-design`.

m	n									
	500	1000	1500	2000	2500	3000	3500	4000	4500	5000
2	999	1999	2999	3999	4999	5999	6999	7999	8999	9999
4	1321	2643	3965	5286	6611	7930	9255	10575	11896	13221
6	1999	3994	5924	8130	9883	12159	14479	16102	18617	20277
8	2206	4458	6670	8864	11187	13727	16028	17882	20455	22747
10	2404	4827	7362	9686	11936	14660	16995	19382	22046	24500
12	2197	4378	6597	8781	10950	13151	15337	17519	19728	21913
13	2000	4000	6000	8000	10000	12000	14000	16000	18000	20000

of entries per machine. For K set to 1% of this value and $m = 13$, this algorithm returned values of the objective function as follows: [2056, 4153, 6186, 8263, 10357, 12416, 14493, 16568, 18621, 20699], for n from 500 to 5000, respectively. However, the computation time is noticeably greater; the time for K as 1% and $m = 13$ was from below 1 s. for $n = 500$ to above one hour for $n = 5000$, for 40% it was not more than 4 s. for any instance. With greater m, time significantly increases. Still, it is not a problem in practical use, when a user has a given number of files and machines and needs one assignment determined at a time. As to the quality of results, depending on m, they can be much better than the ones generated by other methods. Also for $m > 13$ they are very good, for example, for $n = 3000$ and K set as 40%, `block-design` achieved the objective function value equal to 12993, 14908, 16833, and 18612, for $m = 14$, 16, 18, and 20, respectively.

The quality of results from the above tests is visualized in Fig. 3 for selected numbers of machines. To increase readability, absolute values of the objective function are converted to the replication ratio (the lower, the better). For different numbers of machines we can observe different ranks of the methods. As mentioned above, `greedy-opt` does not show the highest quality in general, but it beats the two other approaches when machines are few. For $m = 4$, the replication ratio is ca. 2.50 for `greedy-opt`, ca. 2.64 for `block-design`, and 2.67 for `cell-adjust`. When $m = 8$, we have the case where m fits $p^2/2$ for some p, i.e., the ideal case for `cell-adjust`. It is understandable, then, that it outperforms other methods, with the replication ratio being 4.00 vs. the average values 4.49 for `block-design` and 4.59 for `greedy-opt`. For $m = 12$, `block-design` is the best with the replication ratio ca. 4.39, `cell-adjust` is next with 4.80, and `greedy-opt` reaches 6.24 on average.

For non-identical machines, results of the algorithms are compared with the lower bound for the objective function value for identical machines, discussed in Section 2, which is $m\sqrt{n(n-1)/m}$. It must be stressed that only pure Steiner systems can get close to this value when $m > 1$, and it is not possible for real-world instances (of identical machines). The comparison is visualized in Fig. 4, where the quality is expressed in terms of the replication ratio.

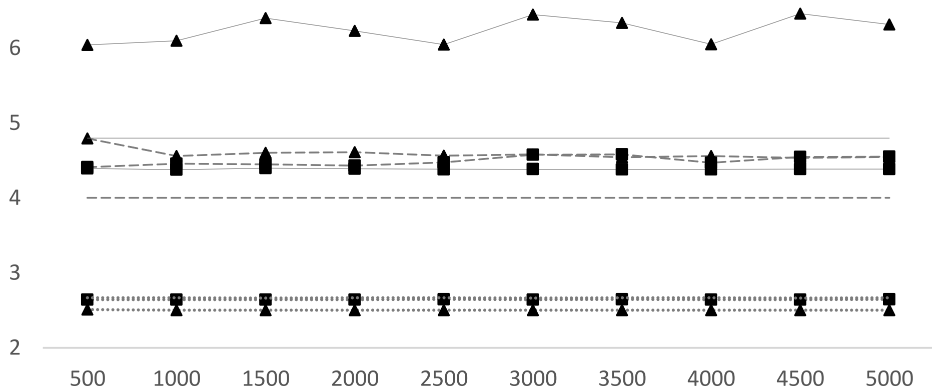

Fig. 3. Replication ratio obtained by the algorithms (y-axis), being the total number of files sent to machines divided by n (x-axis), for identical machines. The results of `greedy-opt` (the triangle marker) and `block-design` (the square marker) are juxtaposed with those of `cell-adjust` (without markers). The dotted line shows the results for $m = 4$, the dashed line for $m = 8$, the solid line for $m = 12$.

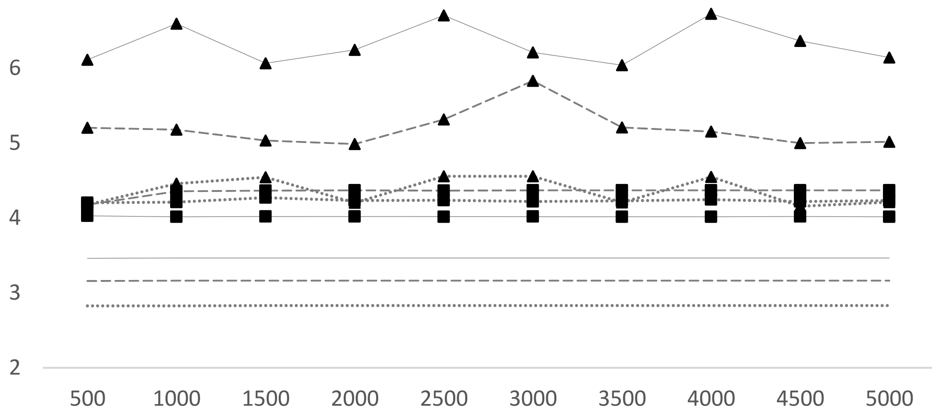

Fig. 4. Replication ratio obtained by the algorithms (y-axis), being the total number of files sent to machines divided by n (x-axis), for non-identical machines. The results of `greedy-opt` (the triangle marker) and `block-design` (the square marker) are juxtaposed with the lower bound for identical machines (without markers). The dotted line shows results for $m = 8$, the dashed line for $m = 10$, the solid line for $m = 12$.

For sets of non-identical machines, K was set to 50% of the average number of entries per machine. As we see in Fig. 4, `block-design` achieved very good results for all the test cases. The results of `greedy-opt` for $m = 8$ are equally good and worse for $m = 10$ or 12. Table 5 presents additional information.

These results clearly show the advantage of the algorithm `block-design` over `greedy-opt`. However, in other circumstances the rank may change. What is important, both algorithms adjust to tight constraints and can quickly produce very good initial solutions even without the refinement step.

7. Conclusions

The problem formulated and solved within this work arose from real needs that emerged during realization of a national genomic project. However, it has a wider application, wherever a comparison (in any sense) of all pairs of large files from a given set is necessary. We proposed three heuristic approaches, each of them having different advantages. The method `cell-adjust` gives very good approximate solutions, also in the sense of even assignment of files to machines, but workloads in most cases are not very well balanced. The algorithm `greedy-opt` returns solutions of even better quality for a small number of machines, but its main advantages are

Table 5. Results for 10 non-identical machines obtained by the algorithms: f stands for the objective function value, $d_{\min}$ and $d_{\max}$ are extreme values for d^k while $l_{\min}$ and $l_{\max}$ for l_k, $k = 1, \ldots, m$, t stands for computation time in seconds. The values for `greedy-opt` are placed at the top of each table cell while for `block-design` at the bottom.

	n = 500	1000	1500	2000	2500	3000	3500	4000	4500	5000
f	2601	5178	7547	9973	13283	17492	18221	20614	22483	25078
	2088	4357	6547	8733	10913	13098	15282	17466	19653	21834
$d_{\min}$	209	418	612	794	1081	1457	1476	1666	1813	2024
	157	337	506	676	845	1014	1184	1353	1522	1691
$d_{\max}$	334	666	984	1297	1699	2199	2341	2654	2924	3259
	282	583	877	1170	1462	1755	2047	2339	2632	2924
$l_{\min}$	11244	45162	95463	189905	294431	421065	555613	720845	839763	1040762
	9846	40381	90374	160979	250169	360314	491564	640167	808991	1001053
$l_{\max}$	17481	70136	151675	289855	450617	645989	861775	1120744	1345900	1665637
	16083	65356	146586	260929	406356	585239	797726	1040067	1315127	1625928
t	0.1	0.9	156	1040	173	1179	121	247	10098	18997
	0.1	0.8	3	9	23	43	83	147	213	342

computation time close to zero in many cases and quite even distribution of files and tasks. It needs more time when such distribution is not the goal, when machines differ in capacity, and the procedure `refineLoad()` has too big participation in computations. The distribution for the algorithm `block-design` fits within defined ranges, but its main advantage is high quality of generated solutions, close to optimum for appropriately defined parameters. Its computation time is negligible comparing to the time spent on genomic (or other) computations within a big project. The approaches were compared on a wide set of instances, where the values of parameters were set to the ones coming from our real-world application. The tests demonstrated usefulness of these approaches in practice as well as for the algorithms, also for cases with a set of machines defined as a diverse computational environment with tight constraints.

As possible future work on this subject, we could mention developing procedures better suited for some special cases. For certain settings, where the procedure `refineLoad()` works too long, its complexity starts to be noticeable. Supported by another procedure, `refineLoad()` could be dedicated strictly to refining the resulting assignment.

References

Babai, L. and Wilmes, J. (2013). Quasipolynomial-time canonical form for Steiner designs, *Proceedings of the 45th Annual ACM Symposium on Theory of Computing (STOC'13), Palo Alto, USA*, pp. 261–270.

Berlin, K., Koren, S., Chin, C., Drake, J., Landolin, J. and Phillippy, A. (2015). Assembling large genomes with single-molecule sequencing and locality-sensitive hashing, *Nature Biotechnology* **33**(6): 623–630.

Blazewicz, J., Kasprzak, M., Kierzynka, M., Frohmberg, W., Swiercz, A., Wojciechowski, P. and Zurkowski, P. (2018). Graph algorithms for DNA sequencing—Origins, current models and the future, *European Journal of Operational Research* **264**(3): 799–812.

Bogdanowicz, D. and Giaro, K. (2013). On a matching distance between rooted phylogenetic trees, *International Journal of Applied Mathematics and Computer Science* **23**(3): 669–684, DOI: 10.2478/amcs-2013-0050.

Bose, R. (1939). On the construction of balanced incomplete block designs, *Annals of Eugenics* **9**: 353–399.

Briquet, C., Dalem, X., Jodogne, S. and de Marneffe, P.-A. (2007). Scheduling data-intensive bags of tasks in P2P grids with bittorrent-enabled data distribution, *Proceedings of the 2nd Workshop on Use of P2P, GRID and Agents for the Development of Content Networks (UPGRADE'07), Monterey, USA*, pp. 39–48.

Gopalakrishnan, S. and Caccamo, M. (2006). Task partitioning with replication upon heterogeneous multiprocessor systems, *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06), San Jose, USA*, pp. 199–207.

Grannell, M. and Griggs, T. (1994). An introduction to Steiner systems, *Mathematical Spectrum* **26**(3): 74–80.

Jain, C., Rodriguez-R, L., Phillippy, A., Konstantinidis, K. and Aluru, S. (2018). High throughput ANI analysis of 90K prokaryotic genomes reveals clear species boundaries, *Nature Communications* **9**: 5114.

Majdzik, P. (2022). A feasible schedule for parallel assembly tasks in flexible manufacturing systems, *International*

Journal of Applied Mathematics and Computer Science **32**(1): 51–63, DOI: 10.34768/amcs-2022-0005.

Nukarapu, D., Tang, B., Wang, L. and Lu, S. (2011). Data replication in data intensive scientific applications with performance guarantee, *IEEE Transactions on Parallel and Distributed Systems* **22**(8): 1299–1306.

Nurk, S., Walenz, B., Rhie, A., Vollger, M., Logsdon, G., Grothe, R., Miga, K., Eichler, E., Phillippy, A. and Koren, S. (2020). HiCanu: Accurate assembly of segmental duplications, satellites, and allelic variants from high-fidelity long reads, *Genome Research* **30**(9): 1291–1305.

Ondov, B., Treangen, T., Melsted, P., Mallonee, A., Bergman, N., Koren, S. and Phillippy, A. (2016). Mash: Fast genome and metagenome distance estimation using MinHash, *Genome Biology* **17**: 132.

Povoa, M. and Xavier, E. (2018). Approximation algorithms and heuristics for task scheduling in data-intensive distributed systems, *International Transactions in Operational Research* **25**(5): 1417–1441.

Reid, C. and Rosa, A. (2010). Steiner systems S(2, 4, v)—A survey, *The Electronic Journal of Combinatorics* **#DS18**: 1–34.

Wojciechowski, P., Krause, K., Lukasiak, P. and Blazewicz, J. (2021). The correctness of large scale analysis of genomic data, *Foundations of Computing and Decision Sciences* **46**(4): 423–436.

Paweł Wojciechowski received his PhD degree in computer science from the Poznan University of Technology in 2010. He is currently an assistant professor at the Institute of Computing Science there. His research focuses on bioinformatics, genetics, and parallel computing (including GP-GPU).

Marta Kasprzak is a professor at the Institute of Computing Science, Poznan University of Technology. She focuses her scientific research on bioinformatics, especially on the theoretical analysis of problems: combinatorial modeling of real-world problems, determining their computational complexity, designing algorithms. She has published 73 articles and monographs, 43 of them indexed in the Web of Science. She is a laureate of, among others, the Prime Minister Award for her doctoral dissertation and the Award of the Division IV of Engineering Sciences of the Polish Academy of Sciences in the field of computer science for her habilitation thesis.

Appendix

Figure A1 explains how blocks from the Steiner system S(2,4,13) are mapped to the upper triangular matrix (besides the diagonal), where rows and columns correspond to the elements of the system.

The Bose construction of S(2,4,13)

(0,1,3,9)	*block A*
(0,2,8,12)	*block B*
(0,4,5,7)	*block C*
(0,6,10,11)	*block D*
(1,2,4,10)	*block E*
(1,5,6,8)	*block F*
(1,7,11,12)	*block G*
(2,3,5,11)	*block H*
(2,6,7,9)	*block I*
(3,4,6,12)	*block J*
(3,7,8,10)	*block K*
(4,8,9,11)	*block L*
(5,9,10,12)	*block M*

The blocks of S(2,4,13) in the matrix

	1	2	3	4	5	6	7	8	9	10	11	12
0	A	B	A	C	C	D	C	B	A	D	D	B
1		E	A	E	F	F	G	F	A	E	G	G
2			H	E	H	I	I	B	I	E	H	B
3				J	H	J	K	K	A	K	H	J
4					C	J	C	L	L	E	L	J
5						F	C	F	M	M	H	M
6							I	F	I	D	D	J
7								K	I	K	G	G
8									L	K	L	B
9										M	L	M
10											D	M
11												G

The rearranged matrix

	1	3	9	2	8	12	4	5	7	10	6	11
0	A	A	A	B	B	B	C	C	C	D	D	D
1		A	A	E	F	G	E	F	G	E	F	G
3			A	H	K	J	J	H	K	K	J	H
9				I	L	M	L	M	I	M	I	L
2					B	B	E	H	I	E	I	H
8						B	L	F	K	K	F	L
12							J	M	G	M	J	G
4								C	C	E	J	L
5									C	M	F	H
7										K	I	G
10											D	D
6												D

Fig. A1. Blocks generated by the Bose algorithm for the Steiner system S(2,4,13) (top), the same blocks visualized as parts of the upper triangular matrix (center), and the matrix with rearranged rows and columns (bottom).

Received: 21 July 2023
Revised: 29 November 2023
Accepted: 11 January 2024