A biased random-key genetic algorithm for single-round divisible load scheduling

Julliany S. Brandão[1],[2] ⋆, Thiago F. Noronha[3]*, Mauricio G. C. Resende[4]⋆⋆, and Celso C. Ribeiro[1]*

[1] Universidade Federal Fluminense
Rua Passo da Pátria 156, Niterói, RJ 24210-240, Brazil
{jbrandao,celso}@ic.uff.br
[2] Centro Federal de Educação Tecnológica Celso Suckow da Fonseca
Av. Maracanã, 229, Rio de Janeiro, RJ 20271-110, Brazil
[3] Universidade Federal de Minas Gerais
Avenida Antônio Carlos 6627, Belo Horizonte, MG 24105, Brazil
tfn@dcc.ufmg.br
[4] Mathematical Optimization and Planning, Amazon.com
333 Boren Avenue North, Seattle, WA 98109, USA
resendem@amazon.com

Abstract. A *divisible load* is an amount $W \in \mathbb{R}$ of computational work that can be arbitrarily divided into chunks and distributed among a set P of worker processors to be processed in parallel. Divisible load applications occur in many fields of science and engineering. They can be parallelized in a master-worker fashion, but they pose several scheduling challenges. The Divisible Load Scheduling Problem consists in (a) selecting a subset $A \subseteq P$ of active workers, (b) defining the order in which the chunks will be transmitted to each of them, and (c) deciding the amount of load α_i that will be transmitted to each worker $i \in A$, with $\sum_{i \in A} \alpha_i = W$, so as to minimize the makespan, i.e., the total elapsed time since the master began to send data to the first worker, until the last worker stops its computations. In this work, we propose a biased random-key genetic algorithm for solving the divisible load scheduling problem. Computational results show that the proposed heuristic outperforms the best heuristic in the literature.

Keywords: Divisible load scheduling, Random-key genetic algorithms, metaheuristic, parallel processing, scientific computing

⋆ This work was partially supported by the Brazilian National Council for Scientific and Technological Development (CNPq), the Foundation for Support of Research of the State of Minas Gerais (FAPEMIG), the Foundation for Support of Research of the State of Rio de Janeiro (FAPERJ), and the Coordination for the Improvement of Higher Education Personnel (CAPES), Brazil

⋆⋆ Work of this author was done when he was employed by AT&T Labs Research.

1 Introduction

A *divisible load* is an amount $W \geq 0$ of computational work that can be arbitrarily divided and distributed among different processors to be processed in parallel. The processors are arranged in a star topology and the load is stored in a central *master* processor. The master splits the load into chunks of arbitrary sizes and transmits each of them to other *worker* processors. In the remainder of the text we refer to each worker as a processor, while the term master is used to differentiate the master processor from the worker processors.

The master can only send load to one processor at a time. It is assumed that it does not process the load itself. Any processor can only start processing after it has completely received its respective chunk of the load. The processors are heterogeneous in terms of processing power, communication speed, and setup time for start communicating with the master. Not all available processors should necessarily be used for processing the load. Consequently, despite how the load is split, the choice of (i) which processors are used and (ii) the order in which the chunks are transmitted influences the total processing time of the load.

The Divisible Loading Scheduling Problem (DLSP) was introduced in [13], motivated by an application in intelligent sensor networks. Applications of DLSP arise from a number of scientific problems, such as parallel database searching [12], parallel image processing [27], parallel video encoding [28,38], processing of large distributed files [40], and task scheduling in cloud computing [30], among others.

In this work, we deal with the same DLSP variant treated in [1, 11]. Let $W \in \mathbb{R}$ be the amount of load to be processed, 0 (zero) be the index of the master processor, and $P = \{1, ..., n\}$ be the set of worker processors indices. Each processor $i \in P$ has (i) a setup time $g_i \in \mathbb{R}$ to start the communication with the master, (ii) a communication time $G_i \in \mathbb{R}$ needed to receive each unit of the load from the master and (iii) a processing time $w_i \in \mathbb{R}$ needed to process each unit of the load. Therefore, it takes $g_i + \alpha_i \cdot G_i$ units of time for the master to transmit a load chunk of size $\alpha_i \in \mathbb{R}$ to the processor $i \in P$. Furthermore, it takes an additional $w_i \cdot \alpha_i$ units of time for this worker to process the chunk of load assigned to it.

The scheduling problem consists of (a) selecting a subset $A \subseteq P$ of active processors, (b) defining the order in which the chunks will be transmitted to each active processor and (c) deciding the amount of load α_i that will be transmitted to each processor $i \in A$, with $\sum_{i=1}^{n} \alpha_i = W$, so as to minimize the makespan, i.e., the total elapsed time since the master began to send data to the first processor, until the last processor stops its computations. This problem was proved NP-Hard in [45]. We note that $\alpha_i = 0$, for all $i \in P \setminus A$.

An example of an optimal single-round solution for DLSP is displayed in Figure 1, while an example of a non-optimal solution is displayed in Figure 2. One can see three time bars for each processor in these figures. The first bar represents the amount of time that is necessary to start the communication with the master, while the second corresponds to the amount of time needed to receive the respective chunk of load. Finally, the third bar represents the time spent by

the worker to process this chunk. It can be seen that the master only starts the communication with a processor after finishing the transmission to the previous one.

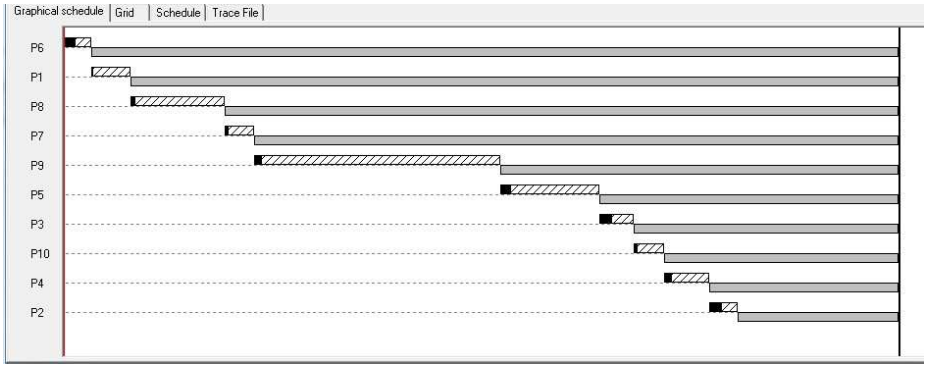

Fig. 1. Example of an optimal single-round scheduling.

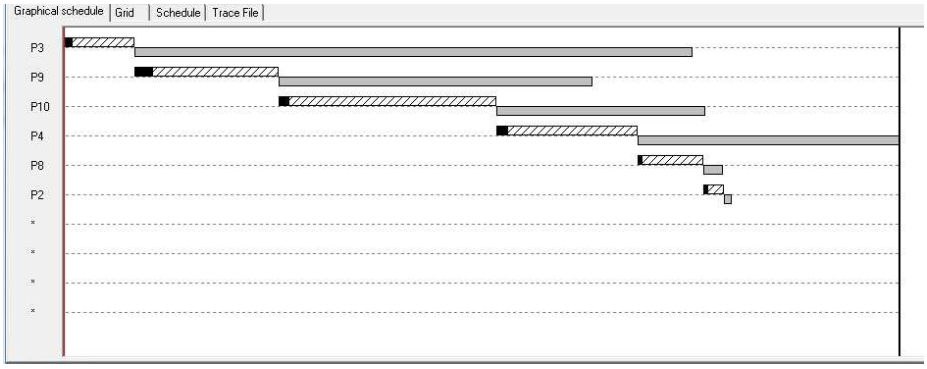

Fig. 2. Example of a non-optimal single-round scheduling.

Since the load can be split arbitrarily, all processors stop at the same time in the optimal solution [5]. In addition, if the order in which the chunks are transmitted to the processors is fixed, then the best solution can be computed in $O(n)$ time, using the AlgRap algorithm developed in [1]. In other words, given a permutation of the processors in P, AlgRap computes the set of active processors and the amount of load that has to be sent to each of them to minimize the makespan. Therefore, DLSP can be reduced to the problem of finding the best permutation of the processors, i.e., the one which induces the solution with the minimum makespan. In order to find this permutation, we propose a biased

random-key genetic algorithm [19, 20, 24], which has been successfully used for solving many permutation based combinatorial optimization problems [18, 20–23, 25, 32, 33].

The remainder of the paper is organized as follows. Related work is reviewed in the next section. The proposed heuristic is described in Section 4. Computational experiments are reported and discussed in Section 5. Concluding remarks are drawn in the last section.

2 Related work

There are many variants of DLSP in the literature. Divisible load scheduling may be performed in a *single round* or in *multiple rounds*. In the single-round case [1, 3–5, 7, 9, 11, 14, 16, 17, 26, 29, 35, 39, 41, 42], each active processor receives and processes a single chunk of the load. In the multi-round case [1, 4, 6, 8, 15–17, 34, 36, 42–44], each active processor receives and processes multiple chunks of load. After the master finishes the transmission of the first round of load to all active processors, it immediately starts the transmission of the next batch of load chunks following the same order, until all the load is distributed among the processors.

The processors may be *homogeneous* or *heterogeneous*. If the processors are homogeneous, the values of g_i, G_i, and w_i are the same for all processors $i \in P$ [4,8,9,26,43,44]. Contrarily, in the heterogeneous case the values of g_i, G_i, and w_i may be different for each processor [1, 3, 5, 6, 11, 14–17, 29, 34–36, 39, 41–44].

The system can be *dedicated* or *non-dedicated*. When the system is dedicated, it is assumed that all resources (processors, memory, network, etc.) are used to process a single computational load [1, 5–7, 9, 11, 14, 16, 17, 34, 39, 41–44]. Non-dedicated systems may be used to simultaneously process different computational loads [6, 7, 35].

There may be a limitation to the maximum chunk size that can be received by each processor. When such a limitation does not exist, the problem is said to be *unconstrained* [1, 4, 5, 8, 9, 11, 14, 34–36, 39, 41]. If the maximum chunk size is limited, the problem is said to be *buffer constrained* [6, 7, 15–17, 29, 42–44].

In this work, we consider the unconstrained single-round DLSP with dedicated and heterogeneous processors [1, 4, 5, 11, 16, 41, 45], that was proved to be *NP*-hard in [45]. Blazewicz and Drozdowski [11] showed that once a permutation of the processors is given, a solution with minimum makespan can be obtained in $O(n \log n)$ time, where $n = |P|$ is the number of processors. Later, Abib and Ribeiro [1] proposed the faster AlgRap algorithm that finds this solution in $O(n)$ time. Beaumont et al. [5] showed that if $g_i = 0$, for all $i \in P$, then DLSP can be polynomially solved by sorting the processors in non-decreasing order of the G_i values. They also showed that if $G_i = G_j$, for all $i, j \in P$, then the problem can be solved by sorting the processors in non-decreasing order of the products $g_i \cdot w_i$.

Non-linear programming formulations for the unconstrained single-round DLSP with dedicated and heterogeneous processors were proposed in [4, 14]. However,

to the best of our knowledge, there are no efficient algorithms in the literature to solve these formulations. The first mixed integer linear programming formulation for DLSP was proposed in [1]. This formulation is described below and used to assess the quality of the heuristic proposed later in this paper.

Let a DLSP instance be defined by $< W, P, g, G, w >$. Formulation (1)-(11) rely on decision variables $x_{ij} \in \{0, 1\}$, with $x_{ij} = 1$ if the processor $i \in P$ is the j^{th} processor to receive its load, and $x_{ij} = 0$ otherwise. Variables $\alpha_{ij} \geq 0$ amount for the size of the load chunk received by the processor $i \in P$ if it is the j^{th} processor to receive its load. We notice that $\alpha_{ij} = 0$ if the processor i is not the j^{th} to receive its load and that $\sum_{j \in \{1,\ldots,|P|\}} \alpha_{ij} = 0$ if processor i is not active. Non-negative auxiliary variables T and t_j, for all $j \in \{1, \ldots, |P|\}$, stand for the makespan and the time the j^{th} processor starts receiving its chunk, respectively:

$$\textit{Minimize} \quad T \tag{1}$$

$$\sum_{i=1}^{n} x_{ij} \leq 1 \qquad \forall j \in \{1, ..., n\} \tag{2}$$

$$\sum_{j=1}^{n} x_{ij} \leq 1 \qquad \forall i \in \{1, ..., n\} \tag{3}$$

$$\sum_{i=1}^{n} x_{ij} \geq \sum_{i=1}^{n} x_{i,j+1} \qquad \forall j \in \{1, ..., n-1\} \tag{4}$$

$$\sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_{ij} = W \tag{5}$$

$$\alpha_{ij} \leq W \cdot x_{ij} \qquad \forall i \in \{1, ..., n\}, \forall j \in \{1, ..., n\} \tag{6}$$

$$t_1 = 0 \tag{7}$$

$$t_j \geq t_{j-1} + \sum_{i=1}^{n} (g_i \cdot x_{i,j-1} + G_i \cdot \alpha_{i,j-1}) \qquad \forall j \in \{2, ..., n\} \tag{8}$$

$$t_j + \sum_{i=1}^{n} (g_i \cdot x_{ij} + (G_i + w_i) \cdot \alpha_{ij}) = T \qquad \forall j \in \{1, ..., n\} \tag{9}$$

$$x_{ij} \in \{0, 1\} \qquad \forall i \in \{1, ..., n\}, \forall j \in \{1, ..., n\} \tag{10}$$

$$\alpha_{ij} \geq 0 \qquad \forall i \in \{1, ..., n\}, \forall j \in \{1, ..., n\}. \tag{11}$$

The objective function (1) consists in minimizing the makespan. Constraints (2) imply that at most one processor can be the j^{th} to receive data. Constraints (3) ensure that each processor can be activated at most once. Constraints (4) guarantee that there must be j previously activated processors when the $(j + 1)^{th}$ is activated. Constraint (5) enforces that the full load is divided over the processors. Constraints (6) establish processor i can only be the j^{th} to receive data if it was chosen to be the j^{th} in the activation order. Constraint (7) is used

to indicate that data transmission starts at time zero. Constraints (8) are used to enforce that the j^{th} processor to be activated will start receiving data after the previous processor in the activation order finishes receiving its data. Constraints (9) imply that the makespan T is equal to the time t_j in which any processor j starts receiving data plus the time $\sum_{i=1}^{n} g_i x_{ij} + (G_i + w_i)\alpha_{ij}$ it needs to receive and process it. These constraints rely on the fact that, in any optimal solution, all processors finish at the same time [10]. Constraints (10) and (11) define the integrality of variables x_{ij} and the non-negativity of variables α_{ij}, respectively. This formulation improves and extends that in [4].

Computational experiments reported in [1] have shown that the CPLEX branch-and-cut algorithm based on this formulation was able to find optimal solutions for 490 out of 720 instances with up to 160 processors. However, for the largest unsolved instances, the integrality gaps were very high, which motivated the development of heuristics for solving DLSP.

Once again, to the best of our knowledge, the HeuRet heuristic proposed by Abib and Ribeiro [1] for the DLSP variant studied in this paper is the most effective in the literature. At each iteration, their algorithm (i) estimates a performance index e_i for each processor $i \in P$ and (ii) builds a solution by taking the processors in P in a non-increasing order of the e_i values. The algorithm sets $e_i = G_i$, for all $i \in P$, in the first iteration and makes use of the exact algorithm AlgRap to compute an initial solution s_0 with makespan T_0. Next, and at each forthcoming iteration k, the estimated performance e_i of each processor $i \in P$ is updated using the values of g_i, w_i, G_i, and T_{k-1} and a new solution with makespan T_k is built by algorithm AlgRap considering the new order defined on the processors by the newly updated performance indices e_i. The procedure stops when $T_{k-1} < T_k$, i.e., when the new solution degenerates the makespan of the previous one due to the use of some processor that is not needed.

The heuristic proposed in Section 4 improves upon HeuRet because, instead of defining a greedy local search based on the performance estimations of the processors, it performs a global search in the space of processor permutations in order to find the permutation that induces the optimal or a near-optimal solution. As both HeuRet and the new heuristic rely on the AlgRap algorithm, we describe it in the next section.

3 Solving DLSP with a fixed activation order

In this section, we describe the linear-time algorithm AlgRap proposed in [1] for the special case in which the processor activation order is fixed beforehand. In this case, the scheduling problem consists exclusively in computing the load to be sent to each processor.

Without loss of generality and for easiness of notation, we assume that processor $i \in P$ is the i-th to be activated. We denote by α_i^* the optimal amount of data to be sent to processor i and we define $f_i = (w_i + G_i)/w_{i-1}$, for $i = 1, \ldots, n$.

3.1 Feasible solutions with a given number of processors

We first suppose that the number ℓ of processors to be used is also known. In this case, Blazewicz and Drozdowski [11] established that the solution to the system

$$\alpha_k \cdot w_k = g_{k+1} + \alpha_{k+1} \cdot (w_{k+1} + G_{k+1}), \quad k = 1, \ldots, \ell - 1 \tag{12}$$

$$\sum_{k=1}^{\ell} \alpha_k = W \tag{13}$$

gives the optimal loads:

$$\alpha_k = \alpha_\ell \prod_{j=k+1}^{\ell} f_j + \sum_{j=k+1}^{\ell} (\frac{g_j}{w_{j-1}} \prod_{i=k+1}^{j-1} f_i), \quad k = 1, \ldots, \ell - 1 \tag{14}$$

$$\alpha_\ell = \frac{W - \sum_{k=1}^{\ell-1} \sum_{j=k+1}^{\ell} (\frac{g_j}{w_{j-1}} \prod_{i=k+1}^{j-1} f_i)}{1 + \sum_{k=1}^{\ell-1} \prod_{j=k+1}^{\ell} f_j}. \tag{15}$$

These values yield a feasible solution if $\alpha_k \geq 0$, for every $k = 1, \ldots, \ell$. Equations (12) imply that the products $\alpha_i \cdot w_i$ are non-increasing for $i = 1, \ldots, \ell$, since all constants G_i, w_i, and g_i are non-negative. Therefore, $\alpha_i \geq 0$ if and only if $\alpha_\ell \geq 0$, for $i = 1, \ldots, \ell$. Finally, we conclude from equation (15) that the above solution is feasible if and only if $V(\ell) = \sum_{k=1}^{\ell-1} \sum_{j=k+1}^{\ell} (\frac{g_j}{w_{j-1}} \prod_{i=k+1}^{j-1} f_i) \leq W$.

3.2 Optimal number of processors

We now investigate the exact number ℓ^* of processors that should be activated in an optimal solution. The term $V(\ell)$ appearing in the numerator of equation (15) may be recursively defined by

$$V(\ell) = \frac{g_\ell}{w_{\ell-1}} \sum_{k=1}^{\ell-1} \prod_{i=k+1}^{\ell-1} f_i + V(\ell - 1). \tag{16}$$

For any activation order, $V(\ell)$ is a non-decreasing function of the number ℓ of activated processors. A solution is infeasible if $V(\ell) > W$. Let the function $F(\ell) = \sum_{k=1}^{\ell-1} \prod_{i=k+1}^{\ell-1} f_i$ be recursively defined by

$$\begin{aligned} F(1) &= 0, \\ F(2) &= 1, \text{ and} \\ F(\ell) &= 1 + F(\ell - 1) f_{\ell-1}. \end{aligned}$$

Therefore, $V(1) = 0$ and for any $\ell > 1$

$$V(\ell) = \frac{g_\ell}{w_{\ell-1}} F(\ell) + V(\ell - 1) \tag{17}$$

may be computed in time $O(1)$ from $V(\ell-1)$ and $F(\ell)$.

Let us assume that a feasible solution exists for r processors and suppose that one additional processor is made available. If a feasible solution satisfying constraints (12-13) still exists, then some load will be transferred from one of the original r processors to the new processor taking part in the computations. In consequence, the loads assigned to the other processors will be decreased and the makespan will be reduced. Therefore, the optimal solution (i.e., that with minimum makespan) will have as many processors as possible to achieve feasibility.

3.3 Linear-time algorithm

Given a fixed activation order, the algorithm starts by sending all the load exclusively to the first processor. Next, the number ℓ of processors is iteratively increased from 1 to n, until $V(\ell)$ turns out to be greater than W. Then, the optimal number of processors is set as $\ell^* = \ell - 1$.

Once the optimal number ℓ^* of processors has been computed, the load α_{ℓ^*} sent to the last processor is computed from equation (15). The other loads α_i, for $i = 1, \ldots, \ell^* - 1$, are recursively computed from ℓ^* using equation (14). Algorithm 1 implements the above computations in time $O(n)$.

In addition to the number of processors and all their data, this algorithm takes as input a vector π describing the activation order, such that $\pi(i) = j$ indicates that processor j is the i-th to be activated, for $i, j = 1, \ldots, n$. For instance, if $n = 3$ and $\pi =< 2, 3, 1 >$, then processor 2 is the first to be activated, processor 3 is the second, and processor 1 is the third.

4 Biased random-key genetic algorithm

Genetic algorithms with random keys, or random-key genetic algorithms (RKGA), were first introduced by [2] for combinatorial optimization problems for which solutions can be represented as a permutation vector. Solutions are represented as vectors of randomly generated real numbers called keys. A deterministic algorithm, called a decoder, takes as input a solution vector and associates with it a feasible solution of the combinatorial optimization problem, for which an objective value or fitness can be computed. Two parents are selected at random from the entire population to implement the crossover operation in the implementation of a RKGA. Parents are allowed to be selected for mating more than once in a given generation.

A biased random-key genetic algorithm (BRKGA) differs from a RKGA in the way parents are selected for crossover, see Gonçalves and Resende [20] for a review. In a BRKGA, each element is generated combining one element selected at random from the elite solutions in the current population, while the other is a non-elite solution. The selection is said biased because one parent is always an elite individual and has a higher probability of passing its genes to the new generation.

Require: Vector π establishing the activation order
Ensure: Minimum makespan T^*, number ℓ^* of processors, loads α_i for $i = 1, \ldots, n$

$F[1] \leftarrow 0$
$F[2] \leftarrow 1$
for $i = 3$ to n **do**
  $F[i] \leftarrow 1 + F[i-1] \cdot (w_{\pi[i-1]} + G_{\pi[i-1]})/w_{\pi[i-2]}$
end for
$V[1] \leftarrow 0$
for $i = 2$ to n **do**
  $V[i] \leftarrow (g_{\pi[i]}/w_{\pi[i-1]}) \cdot F[i] + V[i-1]$
end for
for $\ell = 1$ to n **do**
  if $V[\ell] \leq W$ **then**
    $\ell^* \leftarrow \ell$
  end if
end for
$numerator \leftarrow W - V[\ell^*]$
$product \leftarrow (w_{\pi[\ell^*]} + G_{\pi[\ell^*]})/w_{\pi[\ell^*-1]}$
$denominator \leftarrow 1$
for $k = \ell^* - 1$ down to 1 **do**
  $denominator \leftarrow denominator + product$
  if $k \neq 1$ **then**
    $product \leftarrow product \cdot (w_{\pi[k]} + G_{\pi[k]})/w_{\pi[k-1]}$
  end if
end for
$\alpha_{\pi[\ell^*]} \leftarrow numerator/denominator$
for $k = \ell^* - 1$ to 1 **do**
  $\alpha_{\pi[k]} \leftarrow (g_{\pi[k+1]} + \alpha_{\pi[k+1]} \cdot (w_{\pi[k+1]} + G_{\pi[k+1]}))/w_{\pi[k]}$
end for
for $k = \ell^* + 1$ to n **do**
  $\alpha_{\pi[k]} \leftarrow 0$
end for
$T^* \leftarrow \alpha_{\pi[1]} \cdot (w_{\pi[1]} + G_{\pi[1]}) + g_{\pi[1]}$

Algorithm 1: Linear-time algorithm AlgRap for a fixed activation order.

The BRKGA-DLS biased random-key genetic algorithm for DLSP evolves a population of chromosomes that consists of vectors of real numbers (keys). Each solution is represented by a $|P|$-vector, in which each component is a real number in the range $[0, 1]$ associated with a processor in P. Each solution represented by a chromosome is decoded by a heuristic that receives the vector of keys and builds a feasible solution for DLSP. The decoding heuristic is based on algorithm AlgRap described in the previous section. Decoding consists of two steps: first, the processors are sorted in a non-decreasing order of their random keys; next, the resulting order is used as the input for AlgRap. The makespan of the solution provided by AlgRap is used as the fitness of the chromosome.

We use the parametric uniform crossover scheme proposed in [37] to combine two parent solutions and produce an offspring. In this scheme, the offspring inherits each of its keys from the best fit of the two parents with probability 0.60 and from the least fit parent with probability 0.40. This genetic algorithm does not make use of the standard mutation operator, where parts of the chromosomes are changed with small probability. Instead, the concept of mutants is used: a fixed number of mutant solutions are introduced in the population in each generation, randomly generated in the same way as in the initial population. Mutants play the same role of the mutation operator in traditional genetic algorithms, diversifying the search and helping the procedure to escape from locally optimal solutions.

The keys associated to each processor are randomly generated in the initial population. At each generation, the population is partitioned into two sets: TOP and $REST$. Consequently, the size of the population is $|TOP| + |REST|$. Subset TOP contains the best solutions in the population. Subset $REST$ is formed by two disjoint subsets: MID and BOT, with subset BOT being formed by the worst elements on the current population. As illustrated in Figure 3, the chromosomes in TOP are simply copied to the population of the next generation. The elements in BOT are replaced by newly created mutants that are placed in the new set BOT. The remaining elements of the new population are obtained by crossover with one parent randomly chosen from TOP and the other from $REST$. This distinguishes a biased random-key genetic algorithm from the random-key genetic algorithm of Bean [2], where both parents are selected at random from the entire population. Since a parent solution can be chosen for crossover more than once in a given generation, elite solutions have a higher probability of passing their random keys to the next generation. In this way, $|MID| = |REST - BOT|$ offspring solutions are created. In our implementation, the population size was set to $|TOP| + |MID| + |BOT| = 5 \times |P|$, with the sizes of sets TOP, MID, and BOT set to $0.15 \times 5 \times |P|$, $0.7 \times 5 \times |P|$, and $0.15 \times 5 \times |P|$, respectively, as suggested by Noronha et al. [32].

5 Computational experiments

In this section, we report computational experiments to assess the performance of the biased random-key genetic algorithm BRKGA-DLS. This algorithm was

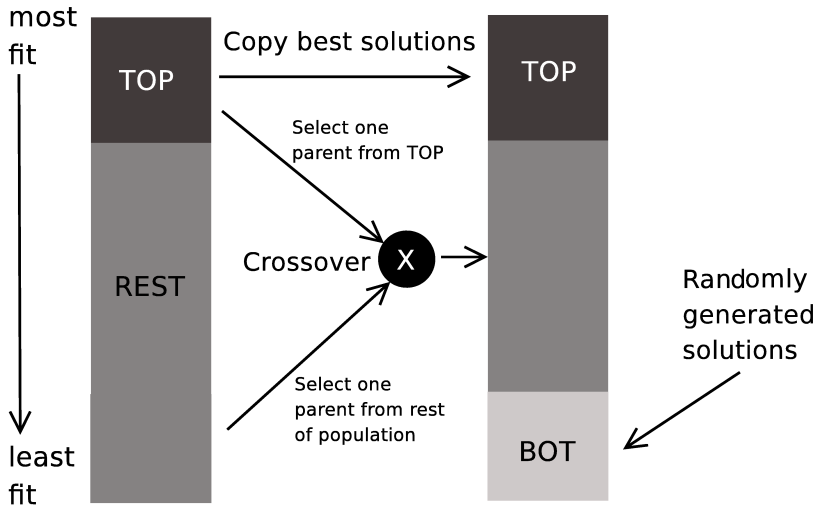

Fig. 3. Population evolution between consecutive generations of a BRKGA.

implemented in C++ and compiled with GNU C++ version 4.6.3. The experiments were performed on a Quad-Core AMD Opteron(tm) Processor 2350, with 16 GB of RAM memory.

The set of instances used in the three first experiments was the same proposed in [1]. There are 120 grid configurations with $n = 10, 20, 40, 80, 160$ worker processors and eight combinations of the parameter values g_i, G_i and w_i, $i = 1, \ldots, n$, each of them ranging either in the interval [1, 100], denoted as *low*, or in the interval [1000, 100,000], denoted as *high*. Three instances were generated for each combination of n, g_i, G_i, and w_i. Six different values of the load $W = 100, 200, 400, 800, 1600, 3200$ are considered for each grid configuration, corresponding to 18 instances for each combination of parameters g_i, G_i and w_i, amounting to a total of 720 test instances. Each heuristic was run 10 times for each instance, with different seeds for the random number generator of [31].

The first experiment consisted in evaluating if BRKGA-DLS efficiently identifies the relationships between keys and good solutions and converges faster to near-optimal solutions. We compare its performance with that of a multi-start procedure that uses the same decoding heuristic as BRKGA-DLS. Each iteration of the multi-start procedure, called MS-DLS, applies the same decoding heuristic of BRKGA-DLS, but using randomly generated values for the keys. In this experiment, BRKGA-DLS was run for 1000 generations and MS-DLS for $1000 \times q$ iterations, where $q = 5 \times |P|$ is the population size of BRKGA-DLS. The results are reported in Figure 4. The first bar gives the average over the 720 instances of the percent relative reduction between the worst solution value returned by BRKGA-DLS with respect to that obtained by MS-DLS. The next two bars give the same information for the average and best solution values. It

can be seen that the average solution values found by BRKGA-DLS were 4.95% better than those provided by MS-DLS. Also, the worst (resp. best) solution values found by BRKGA-DLS were 5.98% (resp. 3.78) smaller than the respective worst (resp. best) solution values obtained with MS-DLS. These results indicate that BRKGA-DLS identifies the relationships between keys and good solutions, making the evolutionary process converge to better solutions faster than MS-DLS.

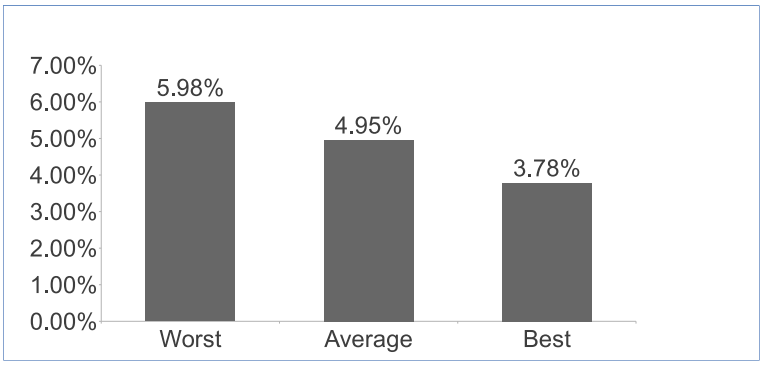

Fig. 4. Average percent relative reduction over the 720 instances of the best, average and worse solution values found by BRKGA-DLS with respect to those obtained by MS-DLS.

In the second experiment, we compare BRKGA-DLS with HeuRet and MS-DLS. We first evaluated how many optimal solutions have been obtained by each heuristic over the 720 test instances. The default CPLEX branch-and-cut algorithm based on the formulation of [1] was also run for the 720 instances. Version 12.6 of CPLEX was used and the maximum CPU time was set to 24 hours. CPLEX was able to prove the optimality for 497 out of the 720 test instances. The first line of Table 1 shows that BRKGA-DLS found optimal solutions for 413 instances (i.e. 83.1%) out of the 497 instances for which the optimal solutions are known, while HeuRet found 320 optimal solutions and MS-DLS found only 177 of them. The second line of Table 1 gives the number of instances for which each heuristic found the best known solution value. The third line of this table shows the number of runs for which BRKGA-DLS and MS-DLS found the best known solution values (we recall that HeuRet is a deterministic algorithm, while the others are randomized). Finally, the last line of this table gives, for each of the three heuristics, a score that represents the sum over all instances of the number of methods that found strictly better solutions than the specific heuristic being considered. The lower a score is, the best the corresponding heuristic is. It can be seen that BRKGA-DLS outperformed both HeuRet and MS-DLS heuristics with respect to the number of optimal and best solutions found, as well as with respect to the score value. In particular, BRKGA-DLS obtained better scores

than HeuRet for all but one instance and found the best known solution values for 645 out of the 720 test instances, while HeuRet found the best solution values for only 313 instances.

Table 1. Summary of the numerical results obtained with BRKGA-DLS, HeuRet and MS-DLS for 720 test instances.

	MS-DLS	HeuRet	BRKGA-DLS
Optimal values (over 497 instances)	177	320	413
Best values (over 720 instances)	189	313	645
Best values (over 7200 runs)	2166	-	6191
Score value	803	112	1

In the third experiment, we assess the computation times of BRKGA-DLS and show how they grow with the number of processors. The results are shown in Table 2. The three first columns show the ranges of values of w_i, g_i, and G_i, respectively, for each group of instances. The five next columns display the results for the instances with n equal to 10, 20, 40, 80, and 160. Since there are three instances for each of the six values of W, each table cell gives the average CPU time of BRKGA-DLS over 18 instances with the same values of w_i, g_i, G_i, and n. One can see that the average CPU time for the instances with 10, 20, 40, 80, and 160 was, respectively, 0.07, 0.31, 1.31, 5.63, 25.18 seconds. These results show that the average computation time of BRKGA-DLS increases linearly with the number of processors by a factor of approximately four every time the number of processors is doubled. The maximum average computation time observed for BRKGA-DLS was 25.18 seconds for the instances with 160 processors.

Table 2. Running times in seconds for BRKGA-DLS.

w_i	g_i	G_i	10	20	40	80	160
low	low	low	0.07	0.30	1.28	5.50	24.65
low	low	high	0.06	0.29	1.24	5.40	24.40
low	high	low	0.06	0.29	1.23	5.41	24.34
low	high	high	0.06	0.29	1.23	5.38	24.43
high	low	low	0.08	0.34	1.48	6.48	28.87
high	low	high	0.08	0.35	1.35	5.70	25.23
high	high	low	0.08	0.33	1.36	5.64	24.84
high	high	high	0.07	0.30	1.29	5.51	24.69
Average			0.07	0.31	1.31	5.63	25.18

The fourth and last experiment provides a more detailed comparison between HeuRet and BRKGA-DLS, based on 20 new, larger and more realistic instances with $|P| = 320$ and $W = 10.000$. The values of G_i and g_i have been

randomly generated in the ranges $[1, 100]$and $[100, 100.000]$, respectively. However, differently from [1], the values of w_i have been randomly generated in the interval $[200, 500]$. These values are more realistic, since the processing rate of a real computer is always larger than its communication rate. In this experiment, BRKGA-DLS was made to stop after $|P|$ generations without improvement in the best solution found. The results are reported in Table 3. The first column shows the instance name. The second and third columns display, respectively, the makespan and the CPU time (in seconds) obtained by HeuRet. The next two columns provide the average makespan over ten runs of BRKGA, the corresponding coefficient of variation, defined as the ratio of the standard deviation to the average. The average computation time in seconds of BRKGA over ten runs is given in the sixth column. The last column shows the percent relative reduction between the average solution found by BRKGA-DLS with respect to that found by HeuRet. It can be seen that the average makespan obtained by BRKGA-DLS is always smaller than that given by HeuRet. In addition, the coefficient of variation of BRKGA-DLS is very small, indicating that it is a robust heuristic. The percent relative reduction of BRKGA-DLS with respect to HeuRet amounted to 3.19% for instance dls.320.10 and to 2.38% on average. As in real-life applications the load size may be very large, and the total communication and processing times may take many hours, an average reduction of 2.38% may be very significant. Although the running times of BRKGA-DLS are larger than those of HeuRet, their average values never exceeded the time taken by HeuRet by more than 30 seconds. Since practical applications of parallel processing take long elapsed times, the trade-off between the reduction in the elapsed time and this small additional running time needed by BRKGA accounts very favorably to BRKGA-DLS.

6 Conclusions

We considered the unconstrained single-round divisible load scheduling problem with dedicated and heterogeneous processors. A new heuristic biased random-key genetic algorithm has been proposed for the problem.

The BRKGA-DLS heuristic improves upon the best heuristic in the literature in terms of solution quality, since it performs a global search in the space of processor permutations in order to find the best activation sequence of the processors.

Computational experiments on 720 test instances with up to 160 processors have shown that BRKGA-DLS found optimal solutions for 413 instances (out of the 497 instances where the optimal solution is known), while the HeuRet heuristic found optimal solutions for only 320 of them. Moreover, BRKGA-DLS obtained better scores than HeuRet for all but one instance and found solutions as good as the best known solution for 645 out of the 720 test instances. To summarize, BRKGA-DLS outperformed the previously existing HeuRet heuristic with respect to all measures considered in Table 1.

Table 3. BRKGA vs. HeuRet on the largest instances with 320 processors.

	HeuRet		BRKGA			
Instance	makespan	time (s)	makespan	CV (%)	time (s)	reduction (%)
dls.320.01	312813.64	0.01	306613.22	0.04	27.24	1.98
dls.320.02	321764.07	0.01	313847.15	0.07	16.72	2.46
dls.320.03	402264.85	0.01	392059.46	0.11	23.72	2.54
dls.320.04	348474.15	0.01	341436.89	0.02	28.16	2.02
dls.320.05	342086.46	0.01	334946.37	0.03	21.67	2.09
dls.320.06	311824.17	0.01	305601.28	0.02	21.93	2.00
dls.320.07	325732.30	0.01	316467.42	0.02	23.19	2.84
dls.320.08	323171.95	0.01	315065.11	0.03	26.23	2.51
dls.320.09	312326.77	0.01	305948.81	0.02	25.03	2.04
dls.320.10	296984.47	0.01	287521.34	0.12	24.04	3.19
dls.320.11	290559.21	0.01	284822.15	0.04	20.56	1.97
dls.320.12	343076.56	0.01	333085.72	0.05	19.53	2.91
dls.320.13	287276.21	0.01	281525.27	0.04	22.77	2.00
dls.320.14	311054.47	0.01	303796.42	0.06	26.81	2.33
dls.320.15	362369.67	0.01	352642.18	0.04	20.41	2.68
dls.320.16	287083.60	0.01	281082.29	0.09	25.38	2.09
dls.320.17	339666.43	0.01	329893.61	0.04	23.53	2.88
dls.320.18	368795.14	0.01	361281.06	0.07	22.08	2.04
dls.320.19	347671.73	0.01	338075.70	0.03	27.59	2.76
dls.320.20	372427.24	0.01	364013.40	0.06	18.28	2.26
Average	330371.15	0.01	322486.24	0.05	23.24	2.38

For the new set of larger and more realistic instances with 320 processors, BRKGA-DLS found solution values 2.38% better than HeuRet on average. In addition, the processing times of BRKGA-DLS are relatively small and never exceeded 30 seconds. Therefore, parallel processing applications dealing with large amounts of data and taking long elapsed times can benefit from BRKGA-DLS, since the additional running time needed by BRKGA-DLS may result in a significant reduction in the makespan.

We are currently working on the extension of this approach to the harder case of multi-round (or multi-installment) scheduling. In this case, the load is distributed to the active processors in several consecutive bursts, reducing the waste in each processor and making better use of the resources to reduce the overall makespan, as illustrated in Figure 5.

References

1. Abib, E.R., Ribeiro, C.C.: New heuristics and integer programming formulations for scheduling divisible load tasks. In: Proceedings of the IEEE Symposium on Computational Intelligence in Scheduling, pp. 54–61. Nashville (2009)
2. Bean, J.C.: Genetic algorithms and random keys for sequencing and optimization. ORSA Journal on Computing **2**, 154–160 (1994)

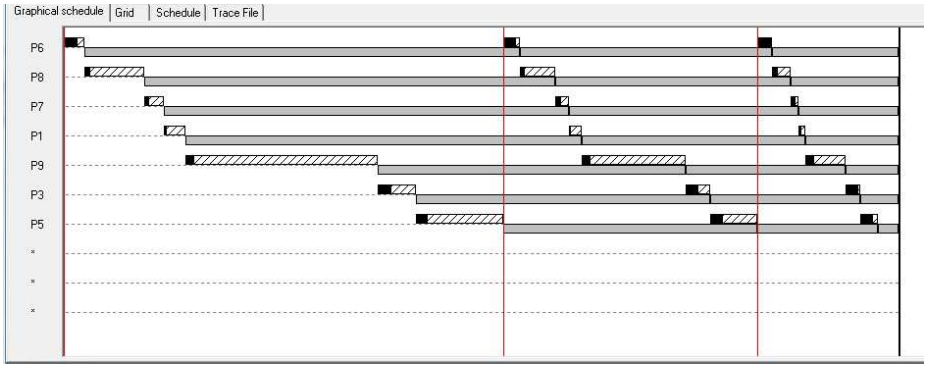

Fig. 5. Example of a multi-round scheduling.

3. Beaumont, O., Bonichon, N., Eyraud-Dubois, L.: Scheduling divisible workloads on heterogeneous platforms under bounded multi-port model. In: International Symposium on Parallel and Distributed Processing, pp. 1–7. IEEE, Miami (2008)
4. Beaumont, O., Casanova, H., Legrand, A., Robert, Y., Yang, Y.: Scheduling divisible loads on star and tree networks: results and open problems. IEEE Transactions on Parallel and Distributed Systems **16**, 207–218 (2005)
5. Beaumont, O., Legrand, A., Robert, Y.: Optimal algorithms for scheduling divisible workloads on heterogeneous systems. In: 12th Heterogeneous Computing Workshop, pp. 98–111. IEEE Computer Society Press, Nice (2003)
6. Berlinska, J., Drozdowski, M.: Heuristics for multi-round divisible loads scheduling with limited memory. Parallel Computing **36**, 199–211 (2010)
7. Berlińska, J., Drozdowski, M., Lawenda, M.: Experimental study of scheduling with memory constraints using hybrid methods. Journal of Computational and Applied Mathematics **232**, 638–654 (2009)
8. Bharadwaj, V., Ghose, D., Mani, V.: Multi-installment load distribution in tree networks with delays. IEEE Transactions on Aerospace and Electronic Systems **31**, 555–567 (1995)
9. Bharadwaj, V., Ghose, D., Mani, V., Robertazzi, T.G.: Scheduling divisible loads in parallel and distributed systems. Wiley - IEEE Computer Society Press (1996)
10. Blazewicz, J., Drozdowski, M.: Scheduling divisible jobs on hypercubes. Parallel Computing **21**, 1945–1956 (1995)
11. Blazewicz, J., Drozdowski, M.: Distributed processing of divisible jobs with communication startup costs. Discrete Applied Mathematics **76**, 21–41 (1997)
12. Błażewicz, J., Drozdowski, M., Markiewicz, M.: Divisible task scheduling–concept and verification. Parallel Computing **25**, 87–98 (1999)
13. Cheng, Y.C., Robertazzi, T.G.: Distributed computation with communication delay. IEEE Transactions on Aerospace and Electronic Systems **24**, 700–712 (1988)
14. Drozdowski, M.: Selected Problems of Scheduling Tasks in Multiprocessor Computer Systems. 321. Politechnika Poznanska (1997)
15. Drozdowski, M., Lawenda, M.: Multi-installment divisible load processing in heterogeneous systems with limited memory. Parallel Processing and Applied Mathematics **3911**, 847–854 (2006)
16. Drozdowski, M., Wolniewicz, P.: Divisible load scheduling in systems with limited memory. Cluster Computing **6**, 19–29 (2003)

17. Drozdowski, M., Wolniewicz, P.: Optimum divisible load scheduling on heterogeneous stars with limited memory. European Journal of Operational Research **172**, 545–559 (2006)
18. Duarte, A., Mart, R., Resende, M., Silva, R.: Improved heuristics for the regenerator location problem. International Transactions in Operational Research **21**, 541–558 (2014)
19. Ericsson, M., Resende, M.G.C., Pardalos, P.M.: A genetic algorithm for the weight setting problem in OSPF routing. Journal of Combinatorial Optimization **6**, 299–333 (2002)
20. Gonçalves, J.F., Resende, M.G.: Biased random-key genetic algorithms for combinatorial optimization. Journal of Heuristics **17**, 487–525 (2011)
21. Gonçalves, J.F., Resende, M.G.: A biased random key genetic algorithm for 2D and 3D bin packing problems. International Journal of Production Economics **145**, 500–510 (2013)
22. Gonçalves, J.F., Resende, M.G.: An extended Akers graphical method with a biased random-key genetic algorithm for job-shop scheduling. International Transactions in Operational Research **21**, 215–246 (2014)
23. Gonçalves, J.F., Resende, M.G., Toso, R.F.: Biased and unbiased random-key genetic algorithms: An experimental analysis. Tech. rep., AT&T Labs Research, Florham Park (2012)
24. Gonçalves, J.F., Resende, M.G.C.: An evolutionary algorithm for manufacturing cell formation. Computers and Industrial Engineering **47**, 247–273 (2004)
25. Gonçalves, J.F., Resende, M.G.C., Costa, M.D.: A biased random-key genetic algorithm for the minimization of open stacks problem. International Transactions in Operational Research (2014). DOI 10.1111/itor.12109
26. Kim, H.J.: A novel optimal load distribution algorithm for divisible loads. Cluster Computing- The Journal of Networks Software Tools and Applications **6**, 41–46 (2003)
27. Lee, C.k., Hamdi, M.: Parallel image processing applications on a network of workstations. Parallel Computing **21**, 137–160 (1995)
28. Li, P., Veeravalli, B., Kassim, A.A.: Design and implementation of parallel video encoding strategies using divisible load analysis. IEEE Transactions on Circuits and Systems for Video Technology **15**, 1098–1112 (2005)
29. Li, X., Bharadwaj, V., Ko, C.: Divisible load scheduling on single-level tree networks with buffer constraints. IEEE Transactions on Aerospace and Electronic Systems **36**, 1298–1308 (2000)
30. Lin, W., Liang, C., Wang, J.Z., Buyya, R.: Bandwidth-aware divisible task scheduling for cloud computing. Software: Practice and Experience **44**, 163–174 (2014)
31. Matsumoto, M., Nishimura, T.: Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Transactions on Modeling and Computer Simulation **8**, 3–30 (1998)
32. Noronha, T.F., Resende, M.G.C., Ribeiro, C.C.: A biased random-key genetic algorithm for routing and wavelength assignment. Journal of Global Optimization **50**, 503–518 (2011)
33. Resende, M.G.: Biased random-key genetic algorithms with applications in telecommunications. TOP **20**, 130–153 (2012)
34. Shokripour, A., Othman, M., Ibrahim, H.: A method for scheduling last installment in a heterogeneous multi-installment system. In: Proceedings of the 3rd IEEE International Conference on Computer Science and Information Technology, pp. 714–718. Chengdu (2010)

35. Shokripour, A., Othman, M., Ibrahim, H., Subramaniam, S.: A new method for job scheduling in a non-dedicated heterogeneous system. Procedia Computer Science **3**, 271–275 (2011)
36. Shokripour, A., Othman, M., Ibrahim, H., Subramaniam, S.: New method for scheduling heterogeneous multi-installment systems. Future Generation Computer Systems **28**, 1205–1216 (2012)
37. Spears, W., deJong, K.: On the virtues of parameterized uniform crossover. In: R. Belew, L. Booker (eds.) Proceedings of the Fourth International Conference on Genetic Algorithms, pp. 230–236. Morgan Kaufman, San Mateo (1991)
38. Turgay, A., Yakup, P.: Optimal scheduling algorithms for communication constrained parallel processing. Lecture Notes in Computer Science **2400**, 197–206 (2002)
39. Wang, M., Wang, X., Meng, K., Wang, Y.: New model and genetic algorithm for divisible load scheduling in heterogeneous distributed systems. International Journal of Pattern Recognition and Artificial Intelligence **27** (2013)
40. Wang, R., Krishnamurthy, A., Martin, R., Anderson, T., Culler, D.: Modeling communication pipeline latency. ACM Sigmetrics Performance Evaluation Review **26**, 22–32 (1998)
41. Wang, X., Wang, Y., Meng, K.: Optimization algorithm for divisible load scheduling on heterogeneous star networks. Journal of Software **9**, 1757–1766 (2014)
42. Wolniewicz, P.: Divisible job scheduling in systems with limited memory. Ph.D. thesis, Poznan University of Technology, Poznań (2003)
43. Yang, Y., Casanova, H.: RUMR: Robust scheduling for divisible workloads. In: Proceedings of the 12th IEEE Symposium on High Performance and Distributed Computing, pp. 114–125. Seattle (2003)
44. Yang, Y., Casanova, H.: UMR: A multi-round algorithm for scheduling divisible workloads. In: Proceedings of the 17th International Parallel and Distributed Processing Symposium, pp. 24–32. Nice (2003)
45. Yang, Y., Casanova, H., Drozdowski, M., Lawenda, M., Legrand, A., et al.: On the complexity of multi-round divisible load scheduling. Tech. Rep. 6096, INRIA Rhône-Alpes (2007)