Cluster Computing (2021) 24:2927–2956
https://doi.org/10.1007/s10586-021-03275-7

A novel simulated annealing-based optimization approach for cluster-based task scheduling

Esra Celik[1] · Deniz Dal[1]

Received: 3 September 2020 / Revised: 17 March 2021 / Accepted: 20 March 2021 / Published online: 27 May 2021

Abstract
Rapidly advancing technology brings a huge volume of data along the way that grows at a staggering pace and cannot be processed with traditional algorithms/hardware. Therefore, storing, processing, and analyzing this data in a timely manner requires distributed data clusters. One of the most critical problems facing these clusters is referred to as task scheduling. In this context, task scheduling is simply the name of the task-cluster node mapping process that will allow the last task to complete its execution as early as possible. Due to the NP-hard nature of the scheduling problem at hand, there is an inevitable need for metaheuristics to solve this problem in such a way that it can produce near-optimal (if possible optimal) solutions at reasonable times. In this study, a simulated annealing-based metaheuristic for cluster-based task scheduling is developed, and serial and parallel (shared memory) versions of the method are implemented in C++. The effectiveness of the proposed approach is demonstrated through twelve famous benchmarks from the Braun dataset. Both the serial and the parallel versions of the approach produce results that are much better than the best latency values ever reported in the literature for all benchmarks within the time constraint of 90 s. For example, the percentage of improvement of the serial version ranges from 0.01% to 0.49%. To decrease the execution time of the developed computer program and improve the quality of the scheduling solutions, different random number generation and perturbation techniques, data structures, early loop termination conditions, exploitation-exploration rates, and compiler effects are also analyzed in detail within the scope of this study.

Keywords Cluster · Heterogeneous computing · Cloud computing · Task scheduling · Metaheuristic · Simulated annealing · Parallel computing · Shared memory · OpenMP

1 Introduction

A computer cluster or simply a cluster is defined as a group of individual computers (nodes) that are connected over a network. A cluster can be classified into two categories based on the homogeneity of the cluster nodes: homogeneous and heterogeneous. A computer cluster is said to be homogeneous when the nodes are from the same vendor and are identical in terms of the configurations of the associated hardware components (e.g. cpu, memory, interconnection technology, etc.) and the software (e.g. operating system, compilers, etc.). On the other hand, a heterogeneous cluster is formed by the nodes of different configurations as opposed to a homogeneous one.

There are several advantages of a cluster, and scalability is one of them. This technical term is used to define the modular growth property of a cluster. In this context, the insertion of a new node into a cluster is a trivial task.

When a cluster is formed from scratch, it often tends towards being homogeneous. However, this nature starts changing after the addition of every new node since the technology changes over time. In the long run, the cluster becomes heterogeneous, and the computational capabilities of the nodes differ considerably [1]. This heterogeneity even increases when grid computing, which is simply defined as a cluster of clusters, is concerned. In other

✉ Deniz Dal
ddal@atauni.edu.tr

Esra Celik
esra.celik@atauni.edu.tr

[1] Computer Engineering Department, Faculty of Engineering, Ataturk University, 25240 Erzurum, Turkey

words, in grid computing, geographically distributed clusters are connected by a network.

In recent years, the rapid increase in computing power and high-speed networks has led to the widespread use of distributed computing environments, such as grid computing, for solving complex problems [2]. One of the critical issues encountered in such heterogeneous systems is the process of mapping tasks, that can be expressed as sub-components of a problem that needs to be solved, to the appropriate nodes in the system. This mapping is not easy at all, as each task has a different execution time on the nodes with different configurations. This problem is also known as the Heterogeneous Computing Scheduling Problem (HCSP). The HCSP is ultimately a task scheduling problem (also known as task mapping, task assignment, and task planning) that aims to minimize the completion time of the last task. It is classified in the NP-hard category, like other scheduling problems in the literature [3].

The development of technology, the global spread of the Internet and the ease of access to the information through the Internet, the widespread use of computers, mobile and IoT devices, economical access to storage areas, and the increase in the use of high-speed networks have brought the concept of big data that we frequently hear. This so-called big data is useless in its raw form [4]. Thus, it needs to be made meaningful depending on the usage. In 2006, a new IT service called Cloud was introduced to address this necessity [5]. A cloud computing infrastructure can help to cope with all the necessary processes from the storage of big data at hand to its analysis [6].

Cloud infrastructures play a vital functionality in transforming big data into information, and one of the critical steps in this transformation is task scheduling [7]. This scheduling performed in the cloud is the task-server mapping process that will allow all tasks to be used to analyze big data to be executed on the cloud servers as early as possible. Therefore, this problem is similar to the HCSP [8].

Cloud task scheduling is a critical area of research in modern cloud computing infrastructures. A task's execution time is fixed in a traditional computing environment after the CPU and all other necessary resources are allocated. However, in a cloud computing environment, the completion time of a task depends on the node where the calculation is performed. The change in the completion time is due to the computer's location where the data are stored. Since the nodes can be located in different geographic locations, the distance between the compute and data nodes plays a vital role in the actual execution time required to complete any task [9]. In this respect, an umbrella term of cluster-based task scheduling is utilized throughout this paper to indicate the task scheduling problem in heterogeneous cluster environments that include cloud computing infrastructures.

An analysis regarding the search space size of the problem at hand is critical since it helps to determine the appropriate technique that can be employed. Let $\boldsymbol{T}$ be the number of tasks that form the problem and $\boldsymbol{N}$ be the number of nodes that the cloud infrastructure has. Since any task can be assigned to any node during scheduling, the number of different solutions is calculated by $\boldsymbol{N^T}$ that indicates an exponential time complexity. At this point, a couple of tests with different $\boldsymbol{T}$ and $\boldsymbol{N}$ values might be handy. For example, according to [10], the size of the solution space for a problem that contains 512 tasks and 16 nodes is a huge number that consists of 617 digits. When the number of tasks and nodes are doubled in a similar scenario, another enormous number with 1542 digits is required to represent the size of the associated search space. It can be concluded from this discussion that it is impractical to find the optimal schedule with brute force-based algorithms when considering the obtained solution space sizes.

On the other hand, task-server mapping cannot be treated as a linear assignment problem (LAP) either, since each resource is assigned to one and only one target in LAP [11]. (In cluster-based task scheduling, there is no such restriction since a task can be assigned to any node.) Therefore, an algorithm in this category, such as the Hungarian Algorithm, cannot be employed.

In this context, integer linear programming (ILP) can be utilized [12]. Thanks to ILP, it is possible to reach the optimal solution of the problem at hand, according to the given constraints with the help of a mathematical function. However, the time to reach a solution is unacceptably long in this case. Since the maximum time limit for the used dataset ([2]) is 90 s ([13]), ILP is not considered as a feasible technique to employ. Thus, we need the help of heuristic or metaheuristic approaches that can produce optimal or near-optimal solutions for this problem at reasonable times.

In this article, a simulated annealing-based metaheuristic for the cluster-based task scheduling was developed. The serial and parallel versions of the proposed approach were converted into a computer program using the C++ programming language. The OpenMP library was also utilized for the parallel version. The effectiveness of the proposed technique was tested with twelve popular benchmarks created by the Braun ([2]) model and widely used in the literature. It was observed that both the serial and parallel versions of the approach produced much better results within the 90-s time constraint than the best latency values ever reported in the literature for all comparisons. For example, the percentage of improvement of the serial version ranges from 0.01% to 0.49%. To decrease the

execution time of the developed computer program and improve the quality of the scheduling solutions, different random number generation and perturbation techniques, data structures, early loop termination conditions, exploitation-exploration rates, and compiler effects were also analyzed in detail within the scope of this study.

The remaining of this paper is organized as follows. Section 2 gives background information. The cluster-based task scheduling problem is introduced in Sect. 3. Section 4 is a fairly detailed literature review on the subject. The Braun dataset and the test environment are presented in Sect. 5. The serial and parallel implementations of the proposed optimization approach are elaborated in Sects. 6 and 7, respectively. Associated experimental results and findings are also discussed with detailed charts and tables in subsections of Sects. 6 and 7. Finally, Sect. 8 concludes this paper and presents some future work opportunities.

2 Background

In the subsections of this section, some basic information is provided to facilitate the understanding of the cluster-based task scheduling problem and the proposed optimization methodology. In this regard, the cloud computing infrastructure and its subcomponents are firstly introduced since it is a highly-preferred heterogeneous cluster environment choice of researchers in recent years. Then, the simulated annealing and its inner working principles are elaborated as it is the metaheuristic utilized in this study.

2.1 Big data

Data is a collection of unarranged, unorganized, and unmeaningful raw material that needs to be interpreted and processed by a human or a machine to be meaningful in a decision-making process [14]. When this so-called data is diverse, produced by small, medium, and large-scale systems, has a high production rate and cannot be processed with traditional algorithms/hardware; it is referred to as big data. Big data emerges with the increase in the use of data-generating technologies and brings various challenges such as accessibility, real-time analysis, and fault tolerance [15]. While it is possible to list a large number of key elements for the formation of big data, the most prominent one is the worldwide increase in the use of technological devices (especially mobile and IoT devices) and the Internet.

2.2 Cloud computing

Cloud computing is an infrastructure that enables software applications, data storage, and processing services accessible over the Internet [16]. It is a powerful technology that allows us to perform large-scale and complex computations due to its pay-per-use model. In this way, an upfront investment cost is minimized.

Big data and cloud computing are tightly linked. Making sense of big data is a challenging and time-consuming process. Therefore, a powerful computational infrastructure is required to distribute the data to the nodes of one or more computational clusters, process it, complete the analysis in a reasonable time, and combine the intermediate results from the nodes. Cloud computing makes use of distributed data processing software such as Hadoop for this purpose [17].

Virtualization technology is also utilized in cloud infrastructure. It is a technique that allows a physical server to be used as multiple virtual servers, each running its own independent operating system. Virtualization plays a critical role in reducing hardware costs and ensuring the efficient use of system resources in terms of resource utilization and sharing [18]. Each virtual server on this infrastructure is called a virtual machine. In this scenario, the component that allows virtualization on a physical server is the hypervisor layer. It allows the server to use its core physical resources efficiently among the virtual machines.

2.3 Storage units

In recent years, different storage unit technologies have emerged as traditional external disks cannot meet the high storage needs of big data applications. Direct Attached Storage (DAS), Network Attached Storage (NAS), and Storage Area Network (SAN) can be listed among these technologies. DAS is a storage unit located on a physical server or directly attached to it. NAS is placed directly on the network and can be accessed via Ethernet or wireless network when needed. On the other hand, SAN makes use of interconnected storage devices instead of a single storage unit in cases where the associated NAS is insufficient. There are also cloud storage technologies available today that allow a large amount of data to be stored for free, without the need for installation and maintenance, with only a few software installed on the user's system [9].

2.4 Hadoop

The structure that contains storage units and servers, each of which in turn has several virtual machines, is called a rack. The racks are brought together to form a cluster. Hadoop is an open-source library developed in Java and runs related applications that will store and process big data on a cluster's nodes [19]. It is defined by Apache as "a framework that allows the distribution of large datasets

among computer clusters using a simple programming model" [20]. Hadoop consists of two components: HDFS (Hadoop Distributed File System) and MapReduce. HDFS is a distributed file system that handles large datasets, while MapReduce is responsible for extracting and filtering the data required for a task from the relevant blocks and producing the final output by combining the intermediate results.

2.5 Scheduling

Scheduling or task scheduling is the process of task-cluster node mapping that tries to minimize the completion time of the last task used to analyze big data. Hadoop employs three different scheduling methods: first-in, first-out scheduling, fair scheduling, and capacity scheduling. First-in, first-out scheduling is the default scheduling utilized by Hadoop [21]. Fair scheduling was developed by Facebook to equally allocate resources to each task in the Hadoop cluster [21]. Capacity scheduling is a scheduling algorithm developed by Yahoo to make fair use of resources [22].

2.6 Metaheuristic

A metaheuristic is a high-level and problem-independent framework that can easily be adapted to solve different discrete or continuous optimization problems. Since it contains various mechanisms for efficiently exploring and exploiting the search space without being trapped into a local optimum, it is possible to reach the optimal or a near-optimal solution in a reasonable time through a metaheuristic [23][24].

A metaheuristic is an iterative process, and it often has components that involve randomness to guide and diversify the search. Metaheuristics can be divided into two broad categories according to the number of solutions maintained during the optimization: trajectory-based (single solution-based) and population-based. Trajectory-based metaheuristics, such as simulated annealing and tabu search, start with a single initial solution and try to improve it iteratively by generating neighbor solutions at each step along the way. On the other hand, population-based metaheuristics, such as genetic algorithm and ant colony optimization, utilize a set of solutions rather than a single solution. In this case, a new solution set is generated through the current set, and this process iterates until a stopping criterion is met.

2.7 Simulated annealing

Simulated annealing (SA) is a single solution-based metaheuristic that originates from the Metropolis algorithm developed by Nicholas Metropolis et al. in 1953 [25]. The primary deficiency of the Metropolis algorithm is that the temperature value remains constant, and the number of iterations is uncertain. In 1983, the shortcomings of the Metropolis algorithm were eliminated by Scott Kirkpatrick et al., and they subsequently revealed that simulated annealing was applicable to optimization problems [26].

Simulated annealing takes its inspiration from the science of metallurgy. Metals heated to higher temperatures are randomly transformed into a liquid and form a perfect crystal structure when cooled slowly. Inspired by this scientific observation, simulated annealing aims to bring an initial solution representing the problem at hand to the desired global solution by starting from high temperatures and slowly decreasing it in each step.

The pseudocode of simulated annealing that can be used in minimization problems is given in Fig. 1. As can be seen from this figure, SA is an iterative method that sets out with an initial solution (current) representing the problem at hand. Then, it generates a certain number of neighbor solutions at each temperature. A neighbor solution is also called the next solution, and it is obtained by making changes (either minor or major) to the current solution.

In each iteration of the for loop, the cost-decreasing next solutions are accepted unconditionally, while the cost-increasing next solutions are accepted probabilistically. In

Algorithm 1: Pseudocode of Simulated Annealing for Minimization Problems

Input: $Initial\ Temperature(T), Final\ Temperature(T_{final}), Cooling\ Rate(\alpha), Inner\ Loop\ Constant(ILC)$
Output: $Best\ Solution(S_{best}), Best\ Cost(C_{best})$

$S \leftarrow GenerateInitialSolution()$
$S_{best} \leftarrow S$
$C_{best} \leftarrow f(S)$
while $(T > T_{final})$
  for $i \leftarrow 1$ *to* ILC ***by*** 1
    $S' \leftarrow GenerateNextSolution(S)$
    $\Delta C \leftarrow f(S') - f(S)$
    if $(\Delta C < 0)$
      $S \leftarrow S'$
      if $(f(S) < C_{best})$
        $S_{best} \leftarrow S$
        $C_{best} \leftarrow f(S)$
      End if
    else
      $r_{01} \leftarrow GenerateARandomRealNumber(0,1)$
      if$(r_{01} < e^{\frac{-\Delta C}{T}})$
        $S \leftarrow S'$
      End if
    End if
  End for
  $T \leftarrow \alpha \cdot T$
End while
return S_{best} *and* C_{best}

Fig. 1 Pseudocode of Simulated Annealing

this way, SA avoids getting stuck at the local optimum. The factors affecting the probabilistic acceptance are the cost difference and temperature. Therefore, such acceptances are more common at high temperatures. As the temperature decreases, only the cost-reducing solutions are accepted. In other words, SA starts acting as a hill-climbing algorithm.

There are several advantages of SA over other metaheuristics. Firstly, it is relatively easy to code. Any exploitation or exploration technique can be easily adapted by different kinds of optimization problems due to its flexibility. It can converge to the global optimum if enough randomness is used in combination with very slow cooling, thanks to its ability to avoid being trapped in the local optima [27].

2.8 Exploitation, exploration, and strategic oscillation

Exploitation refers to the intensification of the search to a limited region of the search space in the hope that it will improve the current solution. In other words, the search is carried out in the immediate neighborhoods of the present solution through minor changes, and it substantially corresponds to a local search. On the other hand, exploration, also known as diversification, aims to direct the search to different regions of the search space that have not yet been discovered [24]. Unlike exploitation, major changes are made to the current solution during exploration, and thus, the search takes place on a global dimension. Another benefit of exploration is to prevent the search from getting stuck at the local optimum.

Determining how often or when exploitation and exploration will be utilized during optimization is crucial and called strategic oscillation (SO) [28]. Thanks to SO, the balance between exploitation and exploration is strategically maintained, and thus, finding global optimums while avoiding local optimums becomes possible. In cases where this balance is not taken into consideration, for example, in an optimization algorithm where the exploitation rate is high, the search exhibits a greedy behavior. Similarly, an algorithm behaves randomly if the exploration rate is kept high. In both cases, it becomes difficult for the algorithm to produce optimal or near-optimal results.

3 Cluster-based task scheduling problem

Large distributed clusters are needed to store, process, and analyze big data in a reasonable time, as mentioned earlier. Such clusters are frequently encountered in the infrastructures that provide cloud computing services. With Hadoop or other similar software, big data is first partitioned into the blocks, and each block is stored in a cluster node. Big data analytics tools are then employed to discover the hidden patterns, correlations, and any other useful information in the distributed data. A big data analytics tool consists of several tasks that process each block of data. As discussed before, mapping the tasks to the nodes can be performed in different ways due to the heterogeneous structure of the associated cluster. In other words, the runtime of a task depends on the node it is mapped. The cluster-based task scheduling problem (CTSP) is the process of mapping the tasks to the nodes in such a way that the completion time of the last task is minimized. It is an optimization problem that is categorized as NP-hard and seeks the global minimum.

3.1 Formulation of the optimization problem

Let $\boldsymbol{HC}$ be a heterogeneous computational system, $\boldsymbol{T} = \{t_1, t_2, \ldots, t_Y\}$ be a set of tasks containing $\boldsymbol{Y}$ tasks, and $\boldsymbol{N} = \{n_1, n_2, \ldots, n_X\}$ be a set of nodes containing $\boldsymbol{X}$ nodes in $\boldsymbol{HC}$. Let us assume that the nodes in set $\boldsymbol{N}$ execute the tasks in set $\boldsymbol{T}$. The difference between the time when the first task in $\boldsymbol{T}$ starts running and the time when the last task completes its execution is called makespan or latency [7]. The CTSP aims to minimize the makespan by trying to find the best task-node mapping.

In the light of the notation introduced above, the problem can be represented by the following mathematical formulas:

$$\boldsymbol{ETC} : \{1, \ldots, \boldsymbol{Y}\}\{1, \ldots, \boldsymbol{X}\} \rightarrow \boldsymbol{R}^{+} \tag{1}$$

$$\boldsymbol{f} : \boldsymbol{T}^{\boldsymbol{Y}} \rightarrow \boldsymbol{N}^{\boldsymbol{X}} \tag{2}$$

$$\textit{Minimize}\left\{ \textit{max}\left\{ \sum_{j=1}^{X} \sum_{i=1}^{Y} \left\{ \begin{array}{c} \boldsymbol{ETC}(i,j), f(i) \textit{ is equal to } j \\ 0, f(i) \textit{ is not equal to } j \end{array} \right\} \right\} \right. \tag{3}$$

The function, named $\boldsymbol{ETC}$ (Expected Time to Compute) and given by (1), returns the execution time of the task t_i at the node n_j when called as $\boldsymbol{ETC(i, j)}$. The function represented by (2) is used for the task-node mapping. Finally, (3) denotes the objective function of the optimization problem at hand.

3.2 A small example

In this subsection, a small example consisting of five tasks ($\boldsymbol{T} = \{t_1, t_2, t_3, t_4, t_5\}$) and four nodes ($\boldsymbol{N} = \{n_1, n_2, n_3, n_4\}$) is introduced to demonstrate how different task-node mappings affect the final schedule. Table 1, which is also known as the ETC matrix, lists the execution times of the tasks in the nodes. The values in this table are taken from the benchmark named ***u_i_hilo.0*** of

Table 1 Execution times of tasks in nodes

T \ N	n_1	n_2	n_3	n_4
t_1	25607.80	7109.95	18425.05	5886.77
t_2	885.65	296.81	205.98	865.13
t_3	50.37	312.40	221.20	229.18
t_4	6030.89	1247.53	1618.50	5767.75
t_5	1286.03	1682.91	1834.15	378.96

the Braun dataset. The rows and columns of this table represent the tasks and the task execution times in the nodes, respectively. For example, at the intersection of the second row (t_2) and the third column (n_3), the value of 205.98 resides, and it is referred to as the execution time of the task t_2 in the node n_3.

There exist 1024 (4^5) different scheduling solutions to map the five tasks in this table to any of the four servers. To explain how task scheduling is performed in the cluster, three different mappings have been created below, and the latency of each corresponding schedule has been calculated accordingly.

Mapping 1: In the first mapping, the tasks are randomly assigned to the nodes, as shown in Fig. 2.

The execution time values in Table 1 are used to calculate the cost for each node, as shown in Table 2. The latency of the schedule is determined by the node having the maximum cost. For example, the latency of the 1st schedule is the cost of the node n_4, which is 6751.90. The Gantt chart of the 1st schedule is given in Fig. 3.

Mapping 2: As shown in Fig. 4, the second mapping employs a heuristic that assigns each task to the node where its execution time is minimal.

The cost of each node and the latency of the schedule belonging to the 2nd mapping are given in Table 3. Figure 5. depicts the associated Gantt chart. The latency of the 2nd schedule is obtained as 6265.73, which is lower than the cost of the 1st schedule. However, as will be explained in a short while, the heuristic does not find the global minimum. In other words, it gets stuck at the local minimum due to its greedy nature.

Fig. 2 First mapping

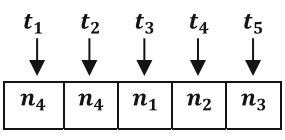


Table 2 Cost calculation for mapping 1

N	T	Cost calculation	Cost
n_1	t_3	$C_{n_1} = ETC(t_3, n_1)$	50.37
n_2	t_4	$C_{n_2} = ETC(t_4, n_2)$	1247.53
n_3	t_5	$C_{n_3} = ETC(t_5, n_3)$	1834.15
n_4	t_1, t_2	$C_{n_4} = ETC(t_1, n_4) + ETC(t_2, n_4)$	6751.90

C: Cost Function

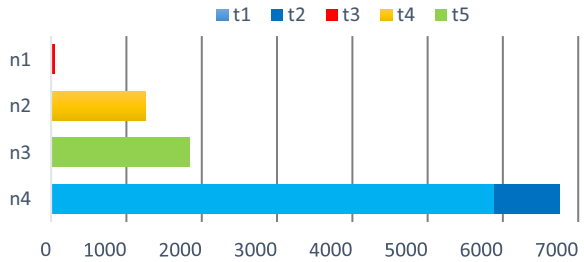


Fig. 3 Gantt chart of Schedule 1

Fig. 4 Second mapping

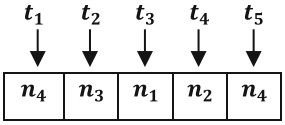


Table 3 Cost calculation for mapping 2

N	T	Cost Calculation	Cost
n_1	t_3	$C_{n_1} = ETC(t_3, n_1)$	50.37
n_2	t_4	$C_{n_2} = ETC(t_4, n_2)$	1247.53
n_3	t_2	$C_{n_3} = ETC(t_2, n_3)$	205.98
n_4	t_1, t_5	$C_{n_4} = ETC(t_1, n_4) + ETC(t_5, n_5)$	6265.73

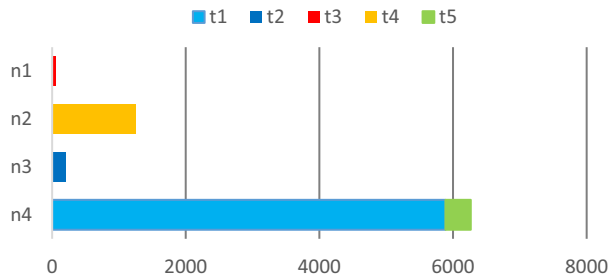


Fig. 5 Gantt chart of Schedule 2

Fig. 6 Third mapping

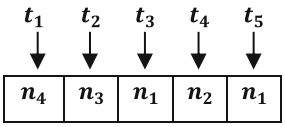


Table 4 Cost calculation for mapping 3

N	T	Cost Calculation	Cost
n_1	t_3, t_5	$C_{n_1} = ETC(t_3, n_1) + ETC(t_5, n_1)$	1336.40
n_2	t_4	$C_{n_2} = ETC(t_4, n_2)$	1247.53
n_3	t_2	$C_{n_3} = ETC(t_2, n_3)$	205.98
n_4	t_1	$C_{n_4} = ETC(t_1, n_4)$	5886.77

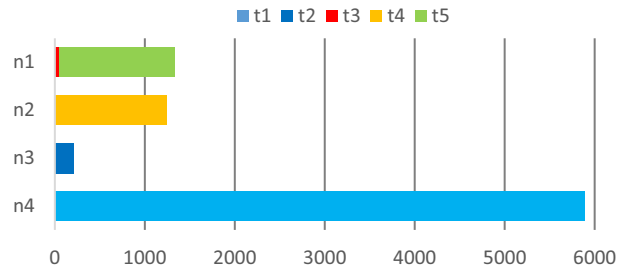


Fig. 7 Gantt chart of Schedule 3

Mapping 3: As shown in Fig. 6, the third mapping optimally assigns the tasks to the nodes.

The cost of each node in the optimal solution and the latency of the 3rd schedule ($\boldsymbol{C_{n_4} = 5886.77}$) are given in Table 4. Figure 7 shows the related Gantt chart.

4 Literature review

In the first two subsections of this section, the methods addressing the task scheduling problem in the literature, such as heuristic and metaheuristic-based approaches and linear programming-based studies, will be discussed. The third subsection will go over different scheduling techniques in heterogeneous multi-cloud environments.

4.1 Heuristic and metaheuristic-based task scheduling approaches

There are different heuristic and metaheuristic methods developed for task scheduling in the literature.

Min–Min is a heuristic that takes the completion times of tasks in nodes into account and assigns each task to the node where its execution time is minimal [29]. This heuristic was utilized in the second mapping of the small example from the previous section.

The heuristic method, called Sufferage, uses the sufferage parameter, which refers to the difference between the completion times of a task in any two nodes [29]. On each iteration, the sufferage value for each task is calculated, and the task with the highest sufferage value is assigned to the fastest node.

Minimum Completion Time (MCT), which is another heuristic method, takes a set of randomly sorted tasks into account [30]. Each task is then assigned to the node with the minimum execution time for that particular task.

Cellular Memetic Algorithm (cMA) is a population-based heuristic that helps to find high-quality solutions in a short time [31]. In cMA, individuals are placed in the population dispersedly, and evolutionary processes are applied to neighbors to increase the quality of the solutions.

Tabu Search (TS) is a metaheuristic that wisely explores the search space to avoid getting stuck at the local optimum [32]. In TS, the first solution is generated by the Min–Min heuristic. TS also employs intensification and diversification techniques to create better solutions than the current solution and diversify the solutions.

The Memetic Algorithm (MA) is another population-based method [33]. It is similar to the genetic and evolutionary algorithms, but unlike them, it does not utilize intermediate populations. In this method, the recombination and mutation processes are applied to the entire population to create diversity. Therefore, the size of the population increases with each iteration. In the selection phase, unlimited growth is prevented by reducing the current population to the size of the initial population. Operating the MA using a selection based on the TS method is called MA + TS in the literature [33].

The Genetic Algorithm (GA) is a population-based method used to search in a vast solution space [34]. In GA, the Min–Min heuristic is employed to initialize the population. Evolutionary processes are then applied to the chromosomes, and the algorithm is terminated if a predetermined execution time or a certain number of iterations is reached.

Parallel Genetic Algorithm (PGA) is a method created by parallelizing the GA to improve search efficiency [34]. The population is divided into as many subpopulations as the number of Physical Processing Elements (PPEs) to utilize a parallel execution. Each subpopulation contains a large number of chromosomes, and each PPE executes the original GA algorithm in the population segment allocated to itself. The Struggle Genetic Algorithm (SGA) is a hash-based genetic algorithm that aims to reduce the computation time [35]. The SGA operates similarly to the GA, but only a fraction of the current population is replaced with new individuals to create a new generation. It attempts to replace the worst individual in the population and accepts a new individual only if it has a better fitness value than the one to be replaced. The goal here is to maintain the optimization speed while obtaining a better individual. The implementation of the SGA is simple, but its performance depends on the diversity of the population.

The Non-Dominated Sorting Genetic Algorithm (NSGA-II) is a multi-purpose type of genetic algorithm designed to solve task scheduling problems [36]. It uses a fast non-dominated order and crowding distance to ensure the diversity of the individuals within the population, unlike the GA. The crowding distance denotes how close an individual is to its neighbors. The individuals with high and low distance values are assigned higher and lower fitness values, respectively, based on their location in the order. Individuals with low fitness values of the population are preferred at the time of selection. Thus, NSGA-II determines the best solution in the population by making effective comparisons.

The Cross Heterogeneous Cataclysmic (CHC) is a specialized version of the traditional GA [34]. This method allows the reproduction of only the parents separated from each other by a large number of bits to keep the best individuals in the population. Besides, recombination and reinitialization methods are utilized to ensure diversity in the population. The recombination process is performed by the HUX (Half Uniform Crossover) approach, which involves the random exchange of exactly half of the bits that differ between two chromosomes. In the reinitialization process, the move or swap operator is applied to the best chromosome in the population.

The Parallel Cross Heterogeneous Cataclysmic (p-CHC) is another technique that is utilized to increase the efficiency of CHC, obtain high-quality solutions, and solve difficult optimization problems [34]. The current population is divided into subpopulations to parallelize the CHC. The CHC algorithm is executed concurrently in each subpopulation. Each individual only interacts with the individuals in the subpopulation in which it resides. A unidirectional ring is employed to allow the exchange of individuals between populations. The usage of subpopulations in the p-CHC algorithm causes the solution to lose its diversity quickly. In other words, it leads to stagnation in the search.

pμ-CHC (Parallel Micro Cross Heterogeneous Cataclysmic) is developed to overcome the diversity loss by combining the p-CHC with the μ-GA (Micro GA) algorithm [13]. The pμ-CHC algorithm, unlike the original p-CHC, creates an external population to store elite solutions during the search. A micro-population composed of eight individuals is used in each subpopulation, and its size is set to 8. On the other hand, there are three individuals in the elite population. The worst individual is removed while inserting a new one into the elite set. The pμ-CHC algorithm reports the best solutions ever found in the literature for the Braun ([2]) dataset by utilizing a computer cluster composed of powerful servers.

4.2 Linear programming-based task scheduling methods

There are linear programming-based studies in the literature that have been developed to perform task scheduling, such as Column Pricing Downhill (CPD) and Column Pricing with Restarts (CPR) [1]. The CPD and CPR employ mathematical models and heuristic methods that enable scheduling in heterogeneous environments where independent tasks are executed. Both approaches assign each task to the node where its execution time is minimal. They then transfer tasks from the node with the highest amount of load to the node with the least amount of load and rank the nodes according to their current loads at each iteration. They subsequently calculate the number of tasks that each node runs by using a mathematical model. After calculating the number of tasks for all machines, the problem is solved through a linear programming solver, and a new solution is produced. The program is terminated in two different ways. The first termination happens when the cost of the schedule reaches the lower bound of a math solver, known as the Gurobi 6.0.3 Integer Programming (IP) solver, which is developed in Java. Terminating the program in this way is called the CPD approach. On the other hand, the approach is called the CPR when the termination employs a restart.

4.3 Task scheduling approaches in heterogeneous multi-cloud environments

Different approaches have been developed to perform task scheduling in heterogeneous multi-cloud environments. For example, the Cloud List Scheduling (CLS) is a static scheduling algorithm [8]. The CLS algorithm assigns priority values to tasks by considering the earliest and latest completion times of the tasks. It then creates a task list based on these priorities. Later, the tasks are allocated to the nodes in the order of this list. Another static scheduling algorithm is the Cloud Min–Min Scheduling (CMMS) [8]. The CMMS assigns tasks to the nodes, just like the Min–Min heuristic explained earlier. On the other hand, unlike Min–Min, the CMMS takes the task dependencies into account.

The Minimum Completion Cloud (MCC) algorithm uses the queue data structure and places tasks into the queue with a first-come first-served approach [37]. It then distributes the tasks involved in the queue to the cloud nodes. It calculates the execution times of the tasks assigned to each cloud node to determine the task-cloud pair that gives the minimum completion time and removes the task from the queue. The algorithm is terminated when the queue is empty.

The MEdian MAX (MEMAX) is a two-stage scheduling algorithm [37]. In the first stage of this algorithm, the average execution times of all ready tasks on the cloud nodes are calculated. In the second stage, the task with the maximum average value is selected and assigned to the node with the lowest execution time.

The Cloud Task Partitioning Scheduling (CTPS) is an online algorithm developed for heterogeneous multi-cloud environments [38]. This algorithm also consists of two phases: preprocessing and processing. In the first phase, the maximum preparation time of a task in all cloud nodes is calculated. In the second phase, the completion time of the same task in the cloud nodes is determined. This particular task is then assigned to the node that provides the minimum completion time. The process is repeated for each task.

The Smoothing Based Task Scheduling (SBTS) is a scheduling algorithm consisting of two stages: smoothing and scheduling [39]. During the smoothing stage, all tasks are divided into groups based on their execution times. In the scheduling stage, groups are sent to the cloud nodes, one after the other in descending order. This scheduling aims to minimize the maximum amount of time spent running tasks on the cloud nodes.

The Two Stages Tasks Transfer (TSTT) approach is the result of the need for a rapid task execution time in distributed computing systems [40]. The goal of this approach is to reduce the execution time and increase the usage of the resources. The TSTT algorithm consists of two phases. In the first phase, the nodes are sorted in increasing order based on their load, and the tasks are assigned to the nodes with the earliest completion time. In the second phase, the nodes are checked, and a reshuffle is carried out between the most loaded node and another one. Finally, [8] is reported to be the first study on scheduling in a unified multi-cloud system.

4.4 Discussion

To the best of our knowledge, this study is the first one in the literature that utilizes the simulated annealing (SA) along with the Braun dataset for the task scheduling problem in the cluster-based systems.

The study that has reported the best latency values for each of the benchmarks in the Braun dataset so far is the pμ-CHC ([13]). On the other hand, the serial version of the proposed SA metaheuristic produced better results than the pμ-CHC as far as the best latency values are concerned. This quality increase in scheduling solutions is one of the points that makes this study unique.

When the average latency values are in question, better results were also obtained for six benchmarks with serial SA than all algorithms. For the remaining six benchmarks, the serial SA metaheuristic was slightly behind the pμ-CHC algorithm. On the other hand, the parallel version of the SA metaheuristic, utilizing a shared-memory architecture, surpassed the pμ-CHC in terms of average latency values in eleven of the twelve benchmarks.

Although a parallel computing cluster composed of powerful servers was employed in the pμ-CHC algorithm, no specialized hardware or parallel computing cluster was used in this study. On the contrary, as it will be explained in Sect. 5.2, a six-year-old moderate laptop computer that any end-user can easily access was utilized as the test environment.

5 Experimental setup

In this section of the article, the dataset formed by the Braun model will be introduced firstly. Then, the test environment will be explained.

5.1 Braun model

A dataset is a collection in which interrelated data are kept together in a specific order and suitable for multi-purpose use. In this study, a famous dataset created with the Braun model ([2]) is utilized. The dataset is divided into two subsets, and each subset contains twelve task scheduling benchmarks. The benchmarks in the first and the second subset consist of 512 tasks and 16 nodes, and 1024 tasks and 32 nodes, respectively. A benchmark is simply a text file that contains the ETC matrix, similar to Table 1.

A benchmark in a subset is categorized into three groups in terms of consistency, task heterogeneity, and machine heterogeneity. Furthermore, it is split into three subgroups as far as the consistency is concerned: consistent, inconsistent, and semi-consistent [30]. The following case holds when a benchmark is consistent: if a node n_j executes a task t_i faster than a node n_k, the node n_j executes all tasks faster than the node n_k. For example, the ETC matrix given in Table 1 is not consistent since n_2 executes t_4 faster than n_1, but n_2 executes t_3 slower than n_1. In other words, for an ETC matrix to be consistent, each row of the matrix must be sorted in ascending order. On the other hand, a benchmark is said to be inconsistent if a node n_j executes some tasks faster than a node n_k, but slower for others. Finally, if a node n_j is consistent with a node n_k, but it is inconsistent with a node n_x in terms of task execution speeds, this case indicates semi-consistency for the related benchmarks.

The variation on the execution times of a given task across all nodes is called the machine heterogeneity. In other words, the variation in the numbers of a given row in the ETC matrix specifies the machine heterogeneity. On the other hand, the amount of variance among the

execution times of tasks for a particular node defines the task heterogeneity. Unlike the machine heterogeneity, the variation in the numbers of a column in the ETC matrix is examined to determine the task heterogeneity.

Each benchmark in the dataset is named with the following notation: **u_x_yyzz.k**. In this nomenclature, **u** denotes an even distribution within the matrix, and **x, yy, zz**, and **k** represents the type of the consistency, the task heterogeneity, the machine heterogeneity, and the number of samples of the same type, respectively. For example, the benchmark named **u_s_lolo.0** is semi-consistent (**s**), and it has a low task (**lo**) and machine heterogeneity (**lo**).

Maximum 7-digit numerical values are used for the benchmarks with high task and machine heterogeneity. In contrast, the benchmarks with low task and machine heterogeneity consist of maximum 3-digit numerical values that represent the task execution times on the nodes. Therefore, the running time of a scheduling algorithm might change based on the sensitivity of the numbers in the relevant benchmark.

The computer program developed within the scope of this study first reads the benchmark file and determines the number of tasks (*NOT*) and the number of nodes (*NON*). Subsequently, the execution times of the tasks on the nodes are stored in a two-dimensional ETC Matrix.

5.2 Test environment

Serial and parallel versions of the simulated annealing-based optimization method developed in this study were run on a laptop computer with the configurations listed in Table 5. This single laptop computer with Intel® Core™ i7-4870HQ processor is chosen to make a relatively fair comparison. The launch date of this processor is Q3'14 (third quarter of 2014), and the study that has reported the best latency values for each of the benchmarks in the Braun dataset so far is the pμ-CHC ([13]), which is published in 2012.

6 Serial simulated annealing

In this section, the stages of the serial implementation of the proposed simulated annealing method and the associated experimental results will be discussed in detail.

The pseudocode for the serial version of the simulated annealing-based optimization approach developed to solve the cluster-based task scheduling problem is shown in Fig. 8. On the other hand, Fig. 9 illustrates the design flow of the proposed method. In Fig. 9, the numbers enclosed in a blue circle represent the steps of the flow, and the red-framed rectangles in each step indicate the preferences or search parameters that were found to improve the method in the corresponding step. For example, the solution representation is determined in step (1), and in this particular step, two different solution representations, namely task-oriented and node-oriented, are examined. As a result of this investigation, it was decided to use the task-oriented encoding. Therefore, the relevant box is enclosed in a red frame. The pseudocode given in Fig. 8 is obtained after each step of the design flow in Fig. 9 is optimized, and thus, it is a special case of the SA template given in Fig. 1. In other words, the problem at hand was first solved using the initial template (Version 1). It was then evolved to Fig. 8 (Version 2), thanks to the optimization steps in Fig. 9 to avoid exceeding the 90-s time limit and improve the quality of the solutions.

Figures 8 and 9 will be frequently referred to in the following subsections to ease the understanding of the proposed algorithm. Besides, *u_i_hilo.0* benchmark will also be used throughout the paper as the running example to show the effect of the associated optimization step.

6.1 Solution encoding (representation)

As can be seen in Fig. 8, the SA metaheuristic first declares two one-dimensional and empty arrays. The first array contains a solution (S_{cur}), while the second one stores the cumulative total of the execution times of the tasks assigned to each node ($Node\ Execution\ Times, NET$). The first array has as many elements as the number of tasks, whereas the second array's size is equal to the number of nodes. Since the cumulative sums will be stored in the second array, zero is used as the initial value of this array's elements.

As explained before, the proposed SA-based method aims to create a task-node mapping that minimizes the cost function. To achieve this goal, it starts with an initial solution and tries to bring this solution iteratively closer to

Table 5 Configurations of the test platform

Processor	Intel® Core™ i7-4870HQ Processor, 6 MB Cache, 2.5 GHz (launch date: Q3'14)
Memory	16 GB, 1600 MHz DDR3
Operating System	Partition 1: macOS High Sierra Partition 2: Windows 10 + Ubuntu 18.04 (WSL, Windows Subsystem for Linux)

Algorithm 2: Pseudocode of the Proposed Serial SA

Input: *Initial Temperature*(T), *Cooling Rate*(α), *Final Temperature*(T_{final}), *Inner Loop Constant*(ILC), *Number Of Tasks*(NOT), *Number Of Nodes*(NON), *Node Execution Times*(NET), *Expected Time to Compute*(ETC), *Thermal Equilibrium Counter Upper Limit*$(TECUL)$, *Exploitation Exploration Ratio*(EER), *Outer Loop Stop Counter Upper Limit*$(OLSCUL)$, *Threshold*, *Hot Enough Constant*(HEC)

Output: *Best Solution*(S_{best}), *Best Cost*(C_{best})

```
S_cur ← Create1DEmtpyArray(NOT)
NET ← Create1DEmtpyArray(NON, 0)
CreateARandomSchedule(S_cur, NON)
NET ← CalculateNodeExecutionTimes(ETC, S_cur)
C_cur ← FindMaximum(NET)
S_best ← S_cur
C_best ← C_cur
C_previous ← −1.0
outerLoopStopCounter ← 0
T ← FindInitialAnnealingTemperature()
while(T > T_final)
    thermalEquilibriumCounter ← 0
    for i ←1 to lLC by 1
        nextSolutionAccepted ← false
        r_01 ← GenerateARandomRealNumber(0,1)
        if(r_01 < EER)
            exploitationOrExploration ← true
            r_1 ← GenerateARandomInteger(1, NOT)
            r_2 ← GenerateARandomInteger(1, NOT)
            NET[S_cur[r_1]] ← NET[S_cur[r_1]] − ETC[r_1][S_cur[r_1]]
            NET[S_cur[r_2]] ← NET[S_cur[r_2]] − ETC[r_2][S_cur[r_2]]
            Swap(S_cur[r_1], S_cur[r_2])
            NET[S_cur[r_1]] ← NET[S_cur[r_1]] + ETC[r_1][S_cur[r_1]]
            NET[S_cur[r_2]] ← NET[S_cur[r_2]] + ETC[r_2][S_cur[r_2]]
        else
            exploitationOrExploration ← false
            r_1 ← GenerateARandomInteger(1, NOT)
            r_3 ← GenerateARandomInteger(1, NON)
            n_temp ← S_cur[r_1]
            NET[n_temp] ← NET[n_temp] − ETC[r_1][n_temp]
            S_cur[r_1] ← r_3
            NET[r_3] ← NET[r_3] + ETC[r_1][r_3]
        End if
        C_next ← FindMaximum(NET)
        ΔC ← C_next − C_cur
        if(ΔC < 0)
            thermalEquilibriumCounter + +
            nextSolutionAccepted ← true
            C_cur ← C_next
            if(C_next < C_best)
                C_best ← C_next
                S_best ← S_cur
            End if
        else
            r_01 ← GenerateARandomRealNumber(0,1)
            if(r_01 < e^(−ΔC/T))
                nextSolutionAccepted ← true
                C_cur ← C_next
            End if
        End if
        if(nextSolutionAccepted is equal to false)
            if(exploitationOrExploration is equal to true)
                NET[S_cur[r_1]] ← NET[S_cur[r_1]] − ETC[r_1][S_cur[r_1]]
                NET[S_cur[r_2]] ← NET[S_cur[r_2]] − ETC[r_2][S_cur[r_2]]
                Swap(S_cur[r_1], S_cur[r_2])
                NET[S_cur[r_1]] ← NET[S_cur[r_1]] + ETC[r_1][S_cur[r_1]]
                NET[S_cur[r_2]] ← NET[S_cur[r_2]] + ETC[r_2][S_cur[r_2]]
            else
                NET[r_3] ← NET[r_3] − ETC[r_1][r_3]
                S_cur[r_1] ← n_temp
                NET[n_temp] ← NET[n_temp] + ETC[r_1][n_temp]
            End if
        End if
        if(thermalEquilibriumCounter is equal to TECUL)
            break the inner for loop
        End if
    End for
    if(C_previous is equal to C_cur)
        outerLoopStopCounter + +
        if(outerLoopStopCounter is equal to OLSCUL)
            break the outer while loop
        End if
    else
        outerLoopStopCounter ← 0
    End if
    C_previous ← C_cur
    T ← α · T
End while
return S_best and C_best
```

Fig. 8 The pseudocode of the Serial SA

Fig. 8 continued

the global optimum. Therefore, how a solution is represented in the method is of great importance.

For this problem, two different solution representations can be utilized (①): task-oriented mapping (Fig. 10) and node-oriented mapping (Fig. 11).

When Figs. 10 and 11 are compared, it is seen that the task-oriented encoding offers four advantages: (1) The task-oriented solution benefits from a one-dimensional array, while the node-oriented solution uses a two-dimensional array. Therefore, the data structure to store a solution in the task-oriented representation is more straightforward. (2) Access to data is effortless in the task-oriented encoding. (3) The perturbation function that transforms the current solution into a new neighbor solution can be more easily realized. (4) The tasks do not need to be stored separately because array indices also represent the task numbers.

Due to the advantages listed above, the task-oriented solution representation was utilized within the scope of this study.

6.2 Determination of the data structure and data type

The necessity of solving the problem in a maximum of 90 s results in performing each step of the algorithm as time-effective as possible. Therefore, step ② aims to determine

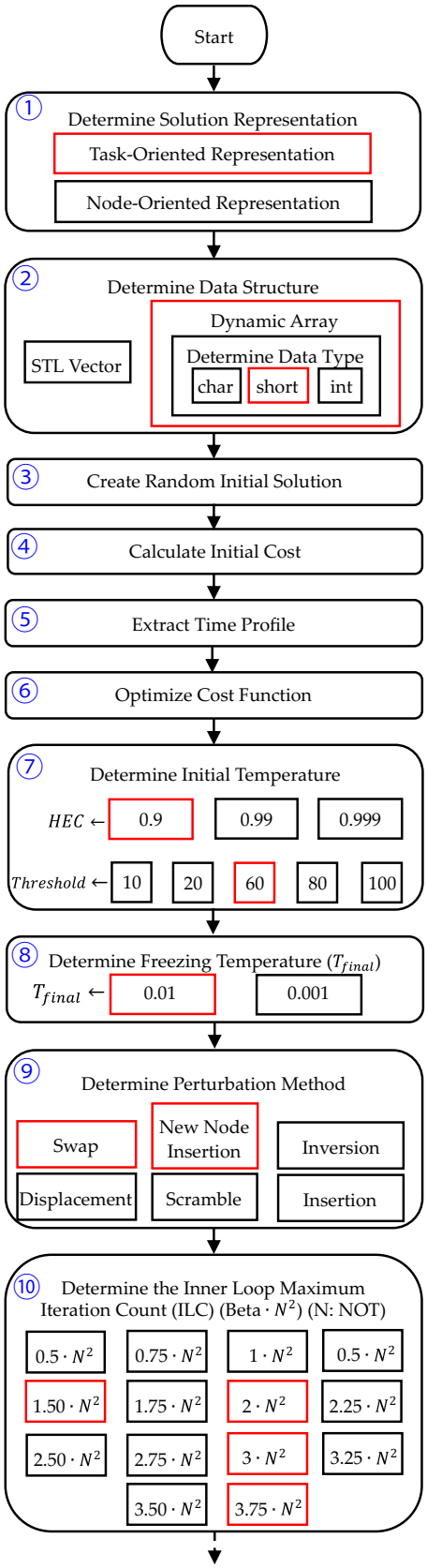


Fig. 9 The design flow of the simulated annealing-based optimization approach

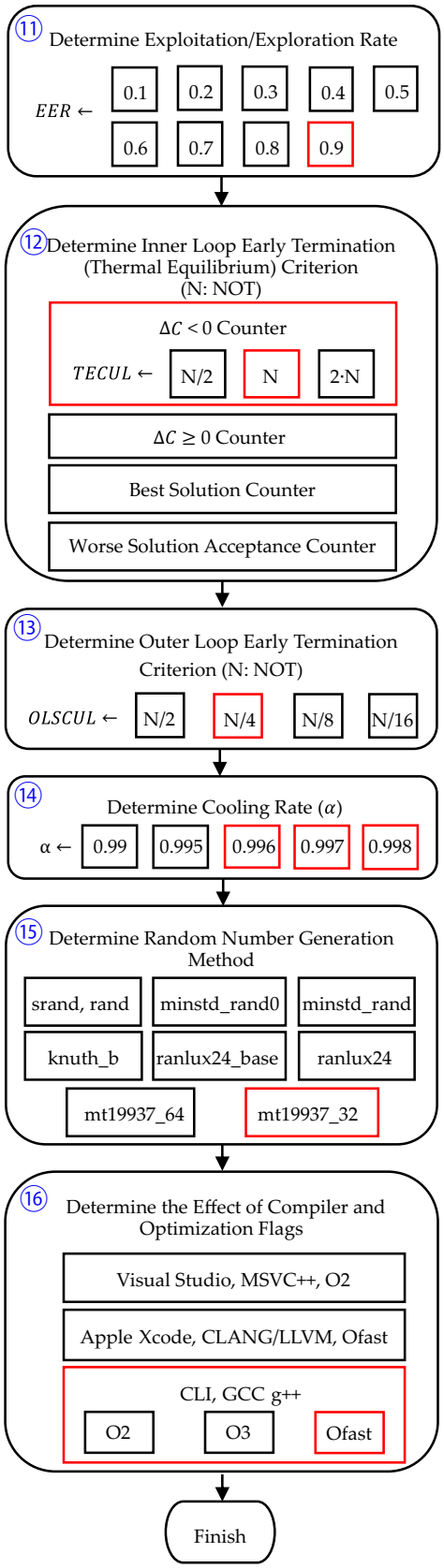


Fig. 9 continued

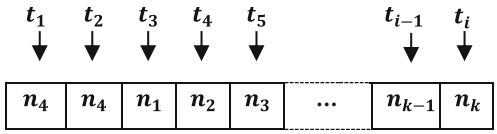


Fig. 10 Task-oriented solution representation

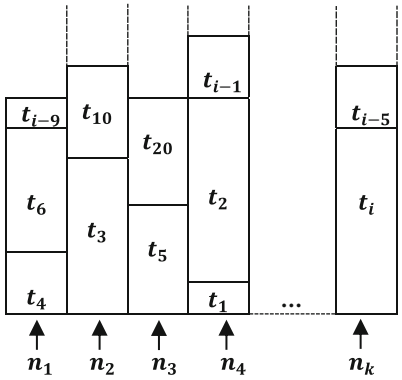


Fig. 11 Node-oriented solution representation

the most suitable data structure used to store the S_{cur} array and the type of data to be stored in this structure. At this step, a one-dimensional dynamic array and a vector container from the STL (Standard Template Library) were evaluated as the available data structures. Conducted tests have shown that the dynamic array runs much faster than the STL vector because it provides direct memory access. Therefore, the data structure preference has been used in favor of the one-dimensional dynamic array. Besides, since this array's elements will be filled with the values in the range $[1, NON]$, experiments have been carried out with *char*, *short,* and *int* primitive data types. Although the *char* data type of 1-byte capacity seems suitable for the problem, it was decided to utilize the *short* data type that stores 2 bytes due to the difficulties of the char encountered in mathematical operations. Utilizing a data type whose capacity is as low as possible is especially important since it affects the running speed in all copying situations as in the $S_{best} \leftarrow S_{cur}$ (line 42 in Fig. 8)

6.3 Creating a random initial solution and optimizing the cost function

In step ③, the SA metaheuristic creates the initial solution by assigning a random number (node number) in the closed range of $[1, NON]$ to each element of the array formed with the task-oriented encoding. Then, in step ④, similar to the creation of Tables 2, 3, and 4, the *NET* array is obtained by taking all tasks and the mapped nodes into account. The maximum element of this array gives the cost of the initial solution.

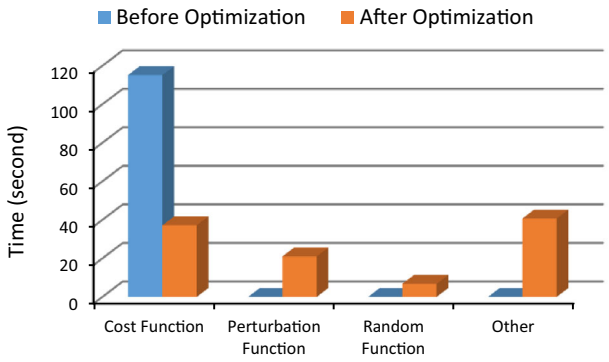


Fig. 12 The runtime profile of the SA metaheuristic for u_i_hilo.0

In the first version of the algorithm, the cost was calculated by checking all task assignments when necessary, as described in the previous paragraph. *Time Profiler* tool of *Xcode* software was used in step ⑤ to analyze the effect of both the cost calculation and other sub-blocks of the algorithm on the runtime. This tool revealed that the block of code to focus on for the time improvement was by far the cost function. As can be understood from Fig. 12, roughly 99% of the running time was spent on the cost calculation in the first version for *u_i_hilo.0*. A more in-depth analysis disclosed that the situation was caused by the cost calculation performed from scratch in all necessary cases in the first version of the algorithm (before the cost function optimization).

In step ⑨ of the design flow, it will be mentioned that the perturbation functions used to transform an existing solution into a next (neighbor) solution are determined as swap and new node insertion. Of these functions, the new node insertion only concerns one task and two nodes, while the swap pertains to two tasks and two nodes. Therefore, it is possible to realize the cost function much faster by considering only the tasks and nodes that cause change. The cost function has been optimized ⑥ by making use of this observation in the pseudocode given in Fig. 8. The values obtained with the runtime profile extracted after the optimization are also given in Fig. 12. This figure shows that the rate spent on the cost function is decreased from 99% to 37% due to the optimization.

6.4 Determination of the initial temperature

The mechanism that allows the SA metaheuristic to reach the global optimum without getting stuck at the local optimum is accepting the cost-increasing solutions probabilistically through a function depending on the temperature and cost difference. As seen in Fig. 8, one of the input parameters of SA is the initial temperature, and the determination of this temperature is vital. A low initial temperature ensures that the algorithm only accepts the cost-

Algorithm 3: Pseudocode of the Algorithm that Finds the Initial Annealing Temperature

Input: $Number\ Of\ Tasks(NOT), Number\ Of\ Nodes(NON),$
$Node\ Execution\ Times(NET), Expected\ Time\ to\ Compute(ETC),$
$Threshold, Hot\ Enough\ Constant(HEC)$
Output: T

```
1:  S_cur ← Create1DEmtpyArray(NOT)
2:  NET ← Create1DEmtpyArray(NON, 0)
3:  CreateARandomSchedule(S_cur, NON)
4:  NET ← CalculateNodeExecutionTimes(ETC, S_cur)
5:  C_cur ← FindMaximum(NET)
6:  totalCost ← 0
7:  counter ← 0
8:  while(true)
9:      S_next ← S_cur
10:     NNET ← NET
11:     r1 ← GenerateARandomInteger(1, NOT)
12:     r2 ← GenerateARandomInteger(1, NOT)
13:     NNET[S_next[r1]] ← NNET[S_next[r1]] − ETC[r1][S_next[r1]]
14:     NNET[S_next[r2]] ← NNET[S_next[r2]] − ETC[r2][S_next[r2]]
15:     Swap(S_next[r1], S_next[r2])
16:     NNET[S_next[r1]] ← NNET[S_next[r1]] + ETC[r1][S_next[r1]]
17:     NNET[S_next[r2]] ← NNET[S_next[r2]] + ETC[r1][S_next[r2]]
18:     C_next ← FindMaximum(NNET)
19:     ΔC ← C_next − C_cur
20:     if(ΔC > 0)
21:         counter + +
22:         totalCost ← totalCost + ΔC
23:         S_cur ← S_next
24:         NET ← NNET
25:         C_cur ← C_next
26:     End if
27:     if(counter is equal to Threshold)
28:         break the while loop
29:     End if
30: End while
31: T ← (− totalCost / counter) / ln(HEC)
32: return T
```

Fig. 13 Pseudocode that finds the initial temperature

decreasing solutions. Since this is ultimately the behavior of a greedy algorithm, SA is less likely to reach the global optimum in such a case. On the other hand, a high initial temperature increases the running time of the algorithm significantly. Therefore, the SA must start the search process with a fine-tuned initial temperature, called hot-enough in the literature [41] [42]. This temperature must be reduced carefully to explore the search space thoroughly.

In step ⑦, the initial temperature is determined, and the pseudocode in Fig. 13 is utilized for this process.

In order for the SA metaheuristic to accept the cost-increasing solutions ($\Delta C > 0$) at high temperatures, the condition given by (4) must be met. In this equation, $\boldsymbol{r}_{01}$ represents a randomly generated real number between 0 and 1, and $\boldsymbol{\Delta C}$ denotes the difference between the next cost and the current cost. Finally, $\boldsymbol{T}$ represents the current temperature.

$$\boldsymbol{r}_{01} < \boldsymbol{e}^{\frac{-\Delta C}{T}} \tag{4}$$

On the left side of (4), there is a randomly generated number between 0 and 1. To guarantee the condition's fulfillment, the right side of the formula must be equal to a constant value close to 1, as in (5).

$$\boldsymbol{e}^{\frac{-\Delta C}{T}} \cong 1 \tag{5}$$

In the proposed approach, this constant value is called *Hot Enough Constant*(HEC), and it is fed as an input parameter to the algorithm. Therefore, (5) turns into (6).

$$\boldsymbol{e}^{\frac{-\Delta C}{T}} \cong \boldsymbol{HEC} \tag{6}$$

Finally, the initial temperature is obtained approximately as (7) by manipulating (6).

$$\boldsymbol{T} \cong \frac{-\boldsymbol{\Delta C}}{\ln(\boldsymbol{HEC})} \tag{7}$$

Formula (7) indicates that the initial temperature is directly proportional to the cost difference. Therefore, in the cost-increasing moves, it is necessary to know the approximate value of ΔC. The proposed approach calls a function whose pseudocode is given in Fig. 13 to achieve this goal. The function first creates a random initial solution. Then, it generates neighbor solutions using the swap operator. In the meantime, it calculates the cumulative sum of the values of ΔC by taking only the cost-increasing solutions into account. When the number of iterative operations reaches a threshold fed as an input to the algorithm, the function terminates and returns the initial temperature.

SA-based algorithms, which are found in the literature, generally run all the benchmarks of the related problem using the same initial temperature and determine this temperature by trial and error [43]. In contrast, our design flow automatically determines the temperature for each benchmark through a function call detailed above.

Tests carried out in this step of the algorithm have shown that the optimal value is 0.9 for *HEC* and 60 for the *Threshold*.

6.5 Determination of the freezing temperature

If Figs. 1 and 8 are scrutinized, it is seen that the SA metaheuristic contains two repetition structures. Whereas the outer structure is a while loop that controls the temperature with the condition $T > T_{final}$, the inner one is a for loop in which the next solutions are generated through perturbation functions. The for loop is usually run at a fixed value for each temperature.

The condition of the outer loop depends on one of the inputs to the SA metaheuristic. It is called T_{final} and also known as the freezing temperature. T_{final} is a parameter that causes the algorithm to terminate. Therefore, like other SA

parameters, this temperature must be determined carefully. Choosing a small freezing temperature extends the algorithm's running time, while utilizing a large one may cause the algorithm to terminate before the current solution reaches the global optimum.

In this step, the tests carried out revealed that the most suitable value for the freezing temperature was 0.01. This choice was due to two disadvantages of smaller freezing temperatures than 0.01: (1) no positive contribution to solution quality was observed, (2) the runtime violated the 90-s constraint.

6.6 Determination of the perturbation method

The SA metaheuristic starts with an initial solution (S_{cur}), and in each iteration of the for loop, it transforms a current solution into a new neighbor solution to reach the global optimum. The functions used for this transformation are called perturbation. In step (9) of the proposed design flow, it is aimed to determine the perturbation method. For this purpose, six different perturbation functions are analyzed:

1 ***New Node Insertion (Renewal)***: As shown in Fig. 14a, this function randomly selects a task index on S_{cur} (e.g. t_2) and replaces the node value (e.g. n_4) in this index with another randomly generated node value (e.g. n_3).

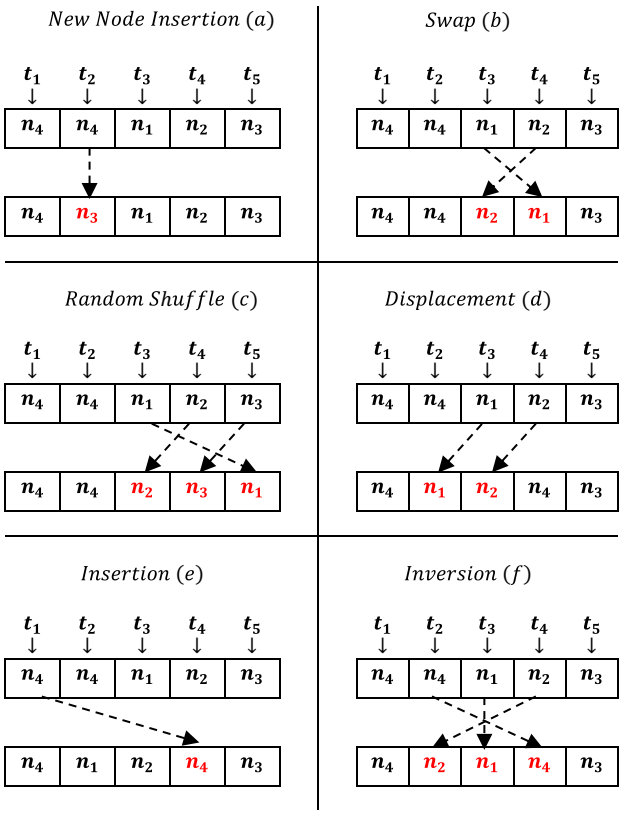


Fig. 14 Illustration of different perturbation methods

2 ***Swap***: As shown in Fig. 14b, this function randomly selects two task indices on S_{cur} (e.g. t_3 and t_4) and swaps the node values in these indices (e.g. n_1 ve n_2).

3 ***Random Shuffle (Scramble)***: As shown in Fig. 14c, this function randomly selects two task indices on S_{cur} (e.g. t_3 and t_5) and randomly shuffles the nodes assigned to the tasks between these indices (e.g. n_1, n_2 and n_3).

4 ***Displacement***: As shown in Fig. 14d, this function randomly selects two task indices on S_{cur} (e.g. t_3 and t_4) and moves the node values between these indices to a randomly generated position (e.g. t_2).

5 ***Insertion***: As shown in Fig. 14e, this function randomly selects a task index on S_{cur} (e.g. t_1) and inserts the node value (e.g. n_4) in this index in a randomly generated position (e.g. t_4).

6 ***Inversion***: As shown in Fig. 14f, this function randomly selects two task indices on S_{cur} (e.g. t_2 ve t_4) and reverses the node values between these indices (e.g. n_4, n_1 and n_2).

The tests conducted revealed that the swap and the new node insertion were the most suitable perturbation methods that minimized the cost of the problem at hand. The results corresponding to the benchmark named *u_i_hilo.0* are given in Fig. 15.

By taking the values in Fig. 15 into account, it is inferred that the swap and the new node insertion functions can be utilized during the exploitation and exploration processes, respectively. Therefore, in Fig. 8, a neighbor solution is created using the swap block that lies between lines 17 and 24. Similarly, the code block located between lines 26 and 32 generates the next solution based on the new node insertion. An elaborate discussion will be held on how frequently these functions are called in step (11).

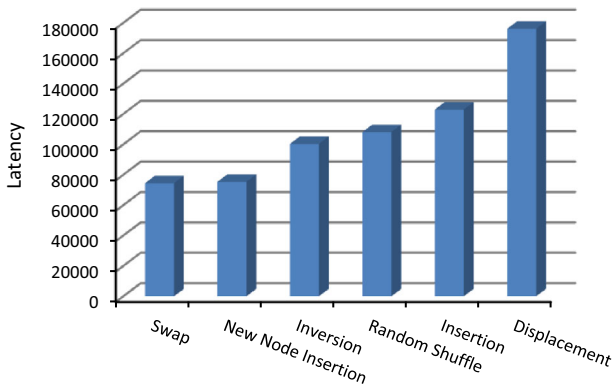


Fig. 15 Best latency values obtained by different perturbation functions for u_i_hilo.0

6.7 Determination of the maximum number of inner loop iterations

In step ⑩ of the proposed design flow, how many times the perturbation process will be performed at each temperature is determined. In other words, the upper limit of the inner loop, which is also called the *Inner Loop Constant*(ILC), is settled. The *ILC*, just like the initial temperature, is crucial for the search space to be adequately explored. Choosing a large *ILC* increases the runtime, while a small *ILC* may prevent the algorithm from reaching the thermal equilibrium. SA-based metaheuristics encountered in the literature usually utilize a constant *ILC* value independent of the parameters of the related problem [42]. On the other hand, in this study, tests were conducted to determine an *ILC* value that would allow the algorithm to achieve thermal equilibrium at each temperature. The heuristic formula given by (8) was obtained as a result of these tests.

$$\boldsymbol{ILC = \beta \cdot N^2} \tag{8}$$

In this formula, β and N represent a constant coefficient, and the input parameter *NOT*, respectively. Table 6 shows the β values that help to optimize the cost for serial and parallel versions of the proposed algorithm. As will be explained in Sect. 6, a parallel algorithm introduces some extra overhead, which corresponds to an increase in the execution time. Therefore, β values (*ILC* values) are reduced accordingly to compensate for this loss in the parallel implementations.

6.8 Determination of the exploitation and exploration rate

In step ⑨, it was decided to use the swap and the new node insertion functions for perturbation. The swap function seems to be the right candidate for the exploitation process since it creates a minor change in the existing solution. On the other hand, adding a new node index to a solution, which may not already be present, indicates a major change for the solution. Therefore, it can be employed during the exploration process. As discussed in Sect. 2.8, the utilization of these two functions together is called strategic oscillation (SO) in the literature, and it significantly contributes to the solution diversity [28]. The implementation of SO requires the determination of the *Exploitation Exploration Rate*(EER). The conducted experiments in step ⑪ of the design flow revealed that the best value for the *EER* was *0.9*. This value indicates that the SA metaheuristic performs exploitation for *90%* of the perturbation process. Similarly, exploration is utilized during the remaining *10%*.

Table 6 β values

Algorithm	β
Serial SA	3.75
Parallel SA(2 *Threads*)	3
Parallel SA(4 *Threads*)	2
Parallel SA(8 *Threads*)	1.5

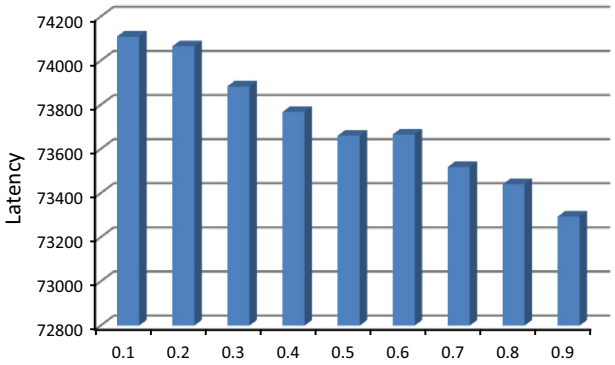


Fig. 16 Best latency values for u_i_hilo.0, corresponding to different EER rates

The best latency values corresponding to different EER rates for *u_i_hilo.0* are given in Fig. 16.

6.9 Determination of the inner and outer loop early termination criteria

During the design of the SA metaheuristic, several experiments were carried out by manipulating different search parameters to obtain better latency values. In parallel with the increase in the values of some parameters (e.g. *ILC*), an increase in the algorithm's running time was observed. Therefore, it was crucial to determine whether an early termination was possible for the loops, both inner and outer, to compensate for such increases.

As explained before, the SA metaheuristic contains two loops. If there is no external intervention, the outer loop that supervises the cooling process ends when the temperature falls below T_{final}, while the inner loop that controls the number of perturbations performed at each temperature terminates when the loop variable reaches the *ILC*.

In step ⑫, tests were conducted towards the early termination of the inner loop. The primary purpose of these tests is to determine whether the inner loop can be terminated early without introducing a negative effect on the solution quality before the relevant loop variable reaches the *ILC*. At this stage, four different analyses were made to identify a thermal equilibrium criterion that would lead to early termination:

Analysis 1 Early termination is performed by assuming that the thermal equilibrium is achieved when the number of the cost-decreasing solutions ($\Delta C < 0$) at any temperature reaches the upper limit fed as an input to the algorithm. In this regard, a counter is placed inside the if-block of the pseudocode given in Fig. 8 (line 36)

Analysis 2 Early termination is performed by assuming that the thermal equilibrium is achieved when the number of the cost-increasing solutions ($\Delta C \geq 0$) at any temperature reaches the upper limit fed as an input to the algorithm. In this respect, a counter is placed inside the else-block of the pseudocode given in Fig. 8 (line 44)

Analysis 3 Early termination is performed by assuming that the thermal equilibrium is achieved when the frequency of updating the best solution ($C_{next} < C_{best}$) at any temperature reaches the upper limit fed as an input to the algorithm. In this regard, a counter is placed inside the if-block of the pseudocode given in Fig. 8 (line 40)

Analysis 4 Early termination is performed by assuming that the thermal equilibrium is achieved when the frequency of accepting the cost-increasing solutions at any temperature reaches the upper limit fed as an input to the algorithm. In this respect, a counter is placed inside the if-block of the pseudocode given in Fig. 8 (line 46)

As a result of the analyses detailed above, it was understood that the best result was obtained thanks to Analysis 1. Subsequent tests revealed that the most reasonable value of the associated input parameter (*Thermal Equilibrium Counter Upper Limit*, *TECUL*) was *NOT*. In other words, the proposed algorithm switches to a new temperature due to the early termination of the inner loop when the number of the cost-decreasing solutions at any temperature is equal to the number of tasks.

On the other hand, in step ⑬ of the design flow, the following criterion was determined for the early termination of the outer loop. Early termination is assumed when the number of the consecutive temperature updates where the current solution does not change reaches the upper limit fed as an input to the algorithm. Subsequent experiments disclosed that the most reasonable value of the associated input parameter (*Outer Loop Stop Counter Upper Limit, OLSCUL*) was *NOT*/4.

Figure 17 shows the runtimes for each benchmark before and after the early termination of the loops. As can be seen from this figure, the running time was reduced to a range that would not violate the 90-sec constraint without causing any degradation in the solutions' quality, thanks to the early termination.

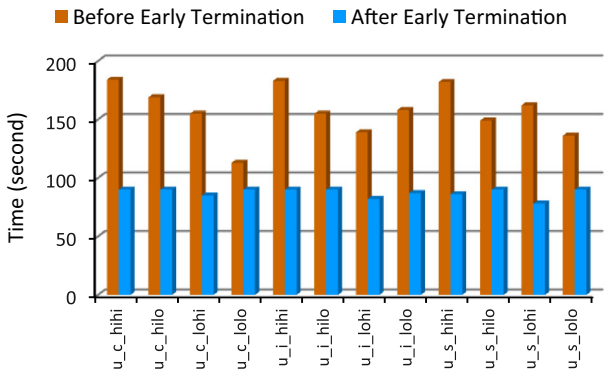


Fig. 17 Execution times before and after early termination

6.10 Determination of the cooling rate

The cooling schedule is the process of decreasing the temperature systematically through the cooling rate (α), which is one of the input parameters of the SA meta-heuristic, after each iteration. There are many different methods used in the literature for the cooling schedule, such as linear, exponential, and geometric. On the other hand, the most prominent of these is the geometric cooling, which allows the geometrical reduction of the temperature by multiplying it by a certain amount known as cooling rate, as shown in (9) [44].

$$\boldsymbol{T} \leftarrow \boldsymbol{\alpha} \cdot \boldsymbol{T} \tag{9}$$

As discussed in Sect. 5.1, the benchmarks of the Braun dataset have different degrees of difficulty and different task execution time sensitivities. Therefore, the running times of the benchmarks in different categories vary even when everything else is the same. In step ⑭ of the design flow, tests were carried out to determine the optimal cooling rate for each benchmark in various categories to most effectively use the 90-s limitation. The results of these tests are listed in Table 7. For example, the temperature is cooled faster for the benchmarks in the *hihi* category. On the other hand, for the benchmarks categorized as *lolo*, a slower cooling schedule is utilized.

Table 7 α Values

Benchmark Category	α
hihi	0.996
hilo, *lohi*	0.997
lolo	0.998

6.11 Determination of the random number generation method

A careful examination of the pseudocode given in Fig. 8 unveils that the proposed SA metaheuristic needs random numbers generated in the form of integers and real numbers for different purposes, such as creating the initial solution, determining the initial temperature, determining the perturbation method, implementing the exploitation-based perturbation, implementing the exploration-based perturbation, and accepting the cost-increasing solutions. Therefore, in step (15), tests were conducted to find a random number generator that not only improves the solution quality but also reduces the runtime.

Since the serial and parallel versions of the proposed approach were implemented in the C++ programming language, functions and classes that could be used for random number generation in this language were examined in the first stage of this step. The most commonly used random number generation function in C++ is the *rand* function that is located in the *cstdlib* library. The most significant disadvantage of this function from our point of view is that it cannot be used in the parallel implementation since it is not thread-safe and cannot be called independently by each thread in the shared memory model. Tests were carried out with seven generator classes located in the *random* library to overcome this adversity. These generators can be used in the serial version without any problem, and private copies of these generators can also be run simultaneously by each thread in the parallel implementation.

The tests conducted to analyze the effect of *rand* and seven other generators on the serial version of the proposed algorithm showed that the generator named mt19937_32 stands out because it both improves the quality of the solutions and reduces the runtime.

These tests were performed for all benchmarks in the Braun dataset, and the results for the benchmark named *u_i_hilo.0* are shown in Fig. 18a. In Fig. 18b, the results of the four generators with the lowest runtime are depicted to make a more meaningful comparison.

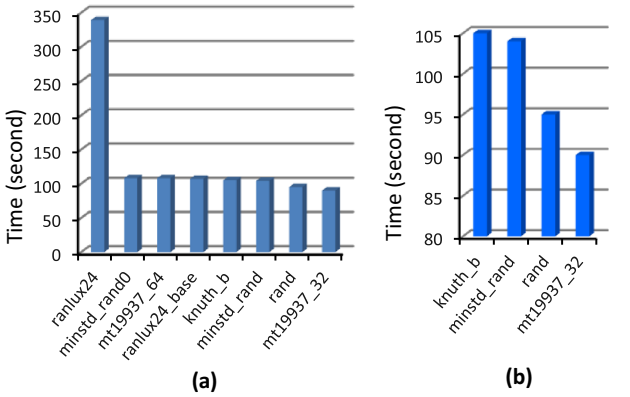


Fig. 18 Execution times with different random number generators for u_i_hilo.0

6.12 Determination of the compiler and optimization flags

In step (16) of the proposed design flow, the effect of different compilers and optimization flags of these compilers on the program's running time was examined. In this respect, three different compilers were taken into account:

1 ***GCC g++***: g++ is an open-source C++ compiler included in the GNU Compiler Collection (GCC) and uses the *ccache*. This cache uses MD4, which is a fast cryptographic hashing algorithm. Thanks to *ccache*, the compilation process is accelerated since the previous compilation results are used in a cache hit instead of a recompilation [45]. In the tests performed with this compiler, the optimization flags, namely *O2*, *O3*, and *Ofast* were inspected separately, and the program was run from the command-line interface (CLI). The version of the compiler was gcc 10.1.0.

2 ***CLANG/LLVM***: *CLANG/LLVM* is the default C++ compiler utilized by the integrated development environment called Apple Xcode and uses the *ccache*. In the tests carried out by this compiler, the optimization flag named *Ofast* was evaluated. The compiler version was Clang 11.0.0.

3 ***MSVC++***: *MSVC++* is the default C++ compiler utilized by the integrated development environment called Microsoft Visual Studio and uses the *clcache*. clcache was inspired by the ccache, but it is not as fast as the ccache in the compilation process. In the tests conducted with this compiler, the optimization flag called O_2 was taken into account. The compiler version was MSVC++ 14.28.

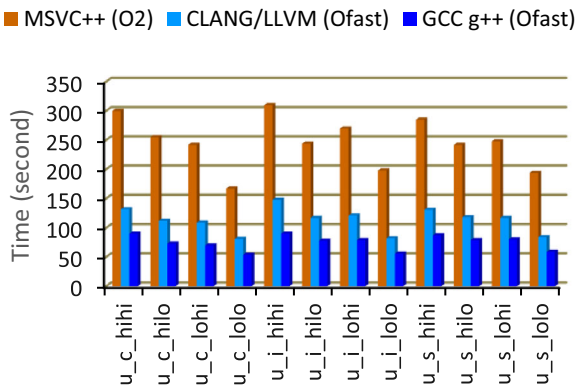


Fig. 19 Runtimes for different compilers

The runtimes for all benchmarks obtained by three different compilers mentioned above are shown in Fig. 19. As can be concluded from the figure, the compiler that leads to minimum runtime is *g++*, and the corresponding flag is *Ofast*.

6.13 Final words about the serial SA method

When the pseudocode given in Fig. 1 is examined, it is seen that the traditional SA metaheuristic first creates a temporary next solution by taking a copy of the current solution in each iteration of the inner loop. It then applies the perturbation to this duplicate solution. A reverse copying is performed if the current solution needs to be replaced by the next solution. Since this process iteratively repeats, the runtime of the first version of the SA metaheuristic is relatively high. Therefore, the proposed algorithm maintains a single solution instead of two and makes a temporary change in the current solution based on the perturbation. If this change is accepted, there is nothing to do. If it is not accepted, the change caused by the perturbation is reversed. This approach is similar to the technique called speculative execution used for the branch prediction in computer architecture [46].

As mentioned earlier, the cost function is also optimized thanks to the proposed algorithm, and the cost of a solution is calculated by only taking the corresponding change into account. For example, the cost calculation for a swap-based perturbation is performed in four steps: (1) the costs of task-node assignments corresponding to the swap operation are subtracted from the current cost, (2) the swap process is performed, (3) the costs of new task-node assignments caused by the swap function are added to the current cost, and (4) the *NET* array is scanned to determine the node with the maximum value. In this way, consideration of all tasks for the cost calculation is avoided, and a significant reduction in runtime is achieved.

6.14 Discussion of the serial implementation's results

The effectiveness of the proposed optimization approach was tested with a famous dataset, which is created with the Braun model and used to compare the performances of the task scheduling algorithms, within the 90-s time constraint. The results for the benchmarks in the dataset that consist of 512 tasks and 16 nodes will be discussed in this subsection, while the discussion of the benchmarks with 1024 tasks and 32 nodes will be held in Sect. 7.

Due to the random nature of the SA metaheuristic, each execution may produce a different result. Therefore, in such applications, the programs are run multiple times to analyze the relevant algorithms' efficiencies more accurately. Consequently, the best and average results are reported. In this study, the number of runs is selected as 30 since it is mostly used in the literature [47][48].

The results of the tests carried out were compared with other heuristics and metaheuristics in the literature. In this respect, NSGA-II ([36]), Sufferage ([29]), Min–Min ([29]), SGA ([35]), cMA ([31]), MA ([33]), MA + TS ([33]), GA ([34]), PGA ([34]), TS ([32]), CHC ([34]), p-CHC ([34]), and pμ-CHC ([13]) were taken into account. It was observed that NSGA-II ([36]), Sufferage ([29]), and Min–Min ([29]) algorithms were executed without a time constraint. On the other hand, the TS ([32]) and the remaining algorithms used 100 s and 90 s, respectively, as the upper limit of the runtime.

The NSGA-II ([36]), Sufferage ([29]), Min–Min ([29]), SGA ([35]), cMA ([31]), MA ([33]), and MA + TS ([33]) only report the best latency values, whereas both the best and average latencies are reported in the remaining studies.

In Table 8, the best latencies produced by the serial SA are compared with the NSGA-II ([36]), Sufferage ([29]), Min–Min ([29]), SGA ([35]), cMA ([31]), MA ([33]), and MA + TS ([33]).

On the other hand, in Tables 9 and 10, the serial SA is compared with the GA ([34]), PGA ([34]), TS ([32]), CHC ([34]), p-CHC ([34]), and pμ-CHC ([13]) by taking both the best and average latencies into account. In all three tables, the red and blue italic fonts indicate the methods that report the best and best average latencies, respectively. As can be noticed, the serial SA metaheuristic surpasses all methods in the literature in a comparison based on the best latencies. When the average latencies are in question, the serial SA still reports the best results for six benchmarks (*u_c_lolo.0, u_i_hilo.0, u_i_lolo.0, u_s_hihi.0, u_s_lohi.0, u_s_lolo.0*) and falls slightly behind the pμ-CHC ([13]) for the remaining benchmarks (*u_c_hihi.0, u_c_hilo.0, u_c_lohi.0, u_i_hihi.0, u_i_lohi.0, u_s_hilo.0*). As will be discussed in the next section, the average latency values are improved even for the five out of six benchmarks thanks to the parallel implementation of the proposed approach.

Figure 20 is included to visualize the data listed in Tables 8, 9, and 10 corresponding to the running benchmark *u_i_hilo.0*. It consists of three parts. In the left half of the figure, the best latency values of fourteen algorithms, including the serial SA, are shown. Four algorithms that report the best latencies among all algorithms are illustrated in the right half of the figure. At the top of the figure, the relative performance increase by the SA metaheuristic over the worst and best methods is depicted. For example, the percentage of improvement of the serial SA over the NSGA-II ([36]) and the pμ-CHC ([13]) are 67.3% and 0.11%, respectively, for *u_i_hilo.0*, as shown in Fig. 20. A similar figure is created for each of the remaining eleven

Table 8 Best latencies of different algorithms (benchmark size: 512 × 16)

Benchmark	NSGA-II [36]	Sufferage [29]	Min-Min [29]	SGA [35]	cMA [31]	MA [33]	MA+TS [31]	Serial SA
u_c_hihi.0	14071196.1	10908697.8	8460675.0	7752689.0	7700929.7	7669920.4	7530020.1	***7377378.2***
u_c_hilo.0	215624.7	167483.2	161805.4	156680.5	155334.8	154631.1	153917.1	***153088.7***
u_c_lohi.0	484360.5	349746.0	275837.3	253926.0	251360.2	249950.8	245288.9	***239069.9***
u_c_lolo.0	7022.6	5649.8	5441.4	5251.1	5218.1	5213.0	5173.7	***5145.5***
u_i_hihi.0	18981587.8	3391758.3	3513919.2	3161104.9	3186664.7	3058785.6	3058474.9	***2931630.9***
u_i_hilo.0	224802.5	78828.2	80755.6	75598.4	75856.6	74939.8	75108.4	***73294.3***
u_i_lohi.0	620779.0	125688.6	120517.7	111792.1	110620.7	107038.8	105808.5	***101746.3***
u_i_lolo.0	7462.8	2673.8	2785.6	2620.7	2624.2	2598.4	2596.5	***2539.1***
u_s_hihi.0	16179189.2	5574357.7	5160342.8	4433792.2	4424540.8	4327249.7	4321015.4	***4098580.9***
u_s_hilo.0	207484.6	103400.8	104375.1	98560.0	98283.7	97804.7	97177.2	***95764.9***
u_s_lohi.0	643263.1	153094.0	140284.4	130425.8	130014.5	127648.9	127633.0	***121484.8***
u_s_lolo.0	7288.9	3727.9	3806.8	3534.3	3522.0	3510.0	3484.0	***3425.9***

Table 9 Best/average latencies of different algorithms (benchmark size: 512 × 16)

Benchmark	GA ([34])		PGA ([34])		TS ([32])		Serial SA	
	Best	Average	Best	Average	Best	Average	Best	Average
u_c_hihi.0	7659878.7	7699080.1	7577921.9	7606613.0	7448640.4	7458864.4	***7377378.2***	***7400359.9***
u_c_hilo.0	155092.0	155300.1	154915.0	155036.5	153263.3	153438.0	***153088.7***	***153334.7***
u_c_lohi.0	250511.8	252568.7	248772.4	249687.9	241672.6	242385.3	***239069.9***	***240062.8***
u_c_lolo.0	5239.1	5248.6	5208.3	5224.7	5154.9	5155.7	***5145.5***	***5150.6***
u_i_hihi.0	3019844.3	3030564.2	2990517.8	3002119.3	2957854.0	2959029.3	***2931630.9***	***2953919.5***
u_i_hilo.0	74142.9	74568.4	74030.3	74102.8	73692.8	73734.8	***73294.3***	***73526.3***
u_i_lohi.0	104688.0	105048.1	103516.0	104078.6	103865.6	103867.1	***101746.3***	***102415.3***
u_i_lolo.0	2577.0	2587.8	2575.4	2577.0	2552.0	2559.9	***2539.1***	***2547.1***
u_s_hihi.0	4332248.2	4347835.5	4262337.5	4282920.5	4168795.8	4181985.8	***4098580.9***	***4121642.4***
u_s_hilo.0	97630.1	98026.1	97505.5	97585.5	96180.8	96432.1	***95764.9***	***96046.4***
u_s_lohi.0	126438.0	126840.8	125717.0	126100.1	123407.4	123600.5	***121484.8***	***122332.4***
u_s_lolo.0	3510.4	3516.5	3480.3	3487.2	3450.5	3454.0	***3425.9***	***3434.1***

benchmarks represented by Figs. 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31.

7 Multi-start parallel SA

The traditional form of computing in which a single operation is performed sequentially in each cycle at the machine level by a processing unit is called serial computing. On the other hand, parallel computing is another form of computation where more than one processing unit can work simultaneously, communicate, and cooperate when necessary. Unlike serial computing, several operations in a single transaction step are executed thanks to parallel computing. The processing unit referred to in these definitions denotes a single core of a multi-core processor or a computer with a single processor. For almost the last two decades, the end-users can quite easily access computers with multi-core processors that could only be found in the supercomputer centers in the past. Therefore, these users can take advantage of their multi-core computers, which fall into the category of shared-memory architecture, and effectively write parallel programs.

OpenMP is the most commonly used application development interface today, making it possible to develop parallel programs through shared-memory architecture [49]. This interface is in the form of a library consisting of compiler directives, library routines, and environment variables, and it supports the C++ programming language.

In parallel programs developed with OpenMP, the code blocks (parallel region) that will be run simultaneously by multiple threads are marked with a special compiler

Table 10 Best/average latencies of different algorithms (benchmark size: 512 × 16)

Benchmark	CHC ([34])		p-CHC ([34])		pμ-CHC ([13])		Serial SA	
	Best	Average	Best	Average	Best	Average	Best	Average
u_c_hihi.0	7599288.4	7681050.1	7461819.1	7481194.5	7381570.0	***7394702.7***	***7377378.2***	7400359.9
u_c_hilo.0	154947.0	155333.4	153791.9	153924.0	153105.4	***153193.7***	***153088.7***	153334.7
u_c_lohi.0	251194.3	251868.3	241513.2	243446.3	239260.0	***239706.2***	***239069.9***	240062.8
u_c_lolo.0	5225.9	5241.9	5177.5	5181.6	5147.9	5152.3	***5145.5***	***5150.6***
u_i_hihi.0	3015048.5	3024904.9	2952493.2	2956905.7	2938380.8	***2947896.4***	***2931630.9***	2953919.5
u_i_hilo.0	74240.9	74375.9	73639.8	73847.1	73378.0	73531.4	***73294.3***	***73526.3***
u_i_lohi.0	104546.0	104939.1	102123.1	102677.3	102050.6	***102402.8***	***101746.3***	102415.3
u_i_lolo.0	2576.7	2582.2	2548.9	2557.2	2541.4	2547.1	***2539.1***	***2547.1***
u_s_hihi.0	4299146.0	4320803.4	4198779.5	4239146.3	4103500.3	4123537.3	***4098580.9***	***4121642.4***
u_s_hilo.0	97888.2	98307.4	96623.3	96750.3	95787.4	***96020.5***	***95764.9***	96046.4
u_s_lohi.0	126238.0	126238.0	123236.9	123989.4	122083.3	122744.4	***121484.8***	***122332.4***
u_s_lolo.0	3492.1	3505.0	3450.1	3472.2	3433.5	3438.3	***3425.9***	***3434.1***

directive, defined as *#pragma omp*. Any code outside a parallel region is executed serially by a particular thread called the master thread whose thread ID is zero.

When the program flow reaches a parallel region (body), other threads in the thread team are activated. In other words, parallel computing starts. After all the statements in the parallel body are executed by all threads, the body ends. Afterward, only the master thread starts executing the following code serially until a new parallel body is encountered. This programming model is known as fork-join, and the default value of the number of threads in OpenMP programs is the number of cores in the processor.

In OpenMP, a private copy of a variable (primitive or user-defined data type) declared in a parallel region is created for each thread. On the other hand, variables declared outside a parallel body are shared by all threads within the associated region.

In this section, a parallel version of the SA metaheuristic, known as the Multi-Start SA (MSSA), was implemented as per the shared-memory model to improve the average latency values obtained by the serial SA. The MSSA is based on the principle that an SA implementation may run simultaneously on independent multiprocessors/cores with the same (different) initial temperatures and initial solutions [50]. In this way, the search space is expanded, and multiple solutions can be produced thanks to the cores that work concurrently in the same period. The best of each solution is reported as the output of the method.

The pseudocode of the MSSA metaheuristic developed within the scope of this study is given in Fig. 32. In this parallel implementation, all threads use the same initial temperature; but, each thread creates a different initial solution randomly. Two additional arrays are utilized to determine which thread finds the best solution when the algorithm ends. The first array (S_{best}) is two-dimensional and used to store the best solution found by each thread (line 2). On the other hand, the second array (C_{best}) is one-dimensional and stores the cost of the best solution each thread finds (line 3). All threads share both of these arrays, but each thread modifies a particular array element by using its thread ID as the index.

As seen in Fig. 32, OpenMP library is used in the MSSA. Unlike the serial SA, the OpenMP directive named *#pragma omp parallel num_threads* that marks the beginning of a parallel region is utilized. This directive takes the number of threads as its parameter. Besides, a unique thread ID is assigned to each thread inside the region by the function named *omp_get_thread_num* command. On the other hand, the initial solution (S_{cur}) is also declared in the parallel block to make it private for each thread. Similarly, a private copy of the initial temperature is supplied to the method to ensure the independent updates of the temperature. The values of other search parameters, such as the cooling rate (α) and inner loop constant (*ILC*), are also kept the same for each thread. Then, the serial SA is run independently by each thread. Each thread accesses the C_{best} array using its thread ID corresponding to a particular index and records the best cost it has found during the search. When all threads are finished ($T_{threadID} \leq T_{final}$), the parallel SA algorithm is terminated.

Tests were performed using three different thread numbers (2, 4, and 8) in the parallel version. As will be discussed in the following subsection, the best results were not necessarily achieved by utilizing eight threads. The differences in quality in the results were caused by the additional cost of managing threads (parallel overhead), which is one of the most critical factors affecting the runtime of parallel programs. Parallel overhead is defined as the amount of time required to coordinate parallel tasks,

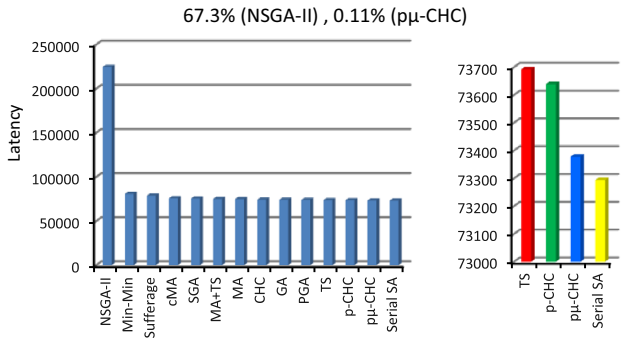


Fig. 20 Best latency values of different algorithms for u_i_hilo.0

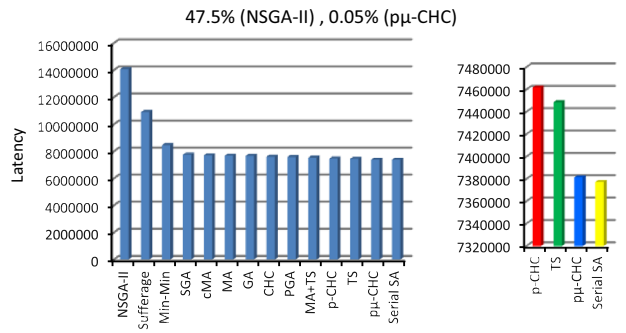


Fig. 21 Best latency values of different algorithms for u_c_hihi.0

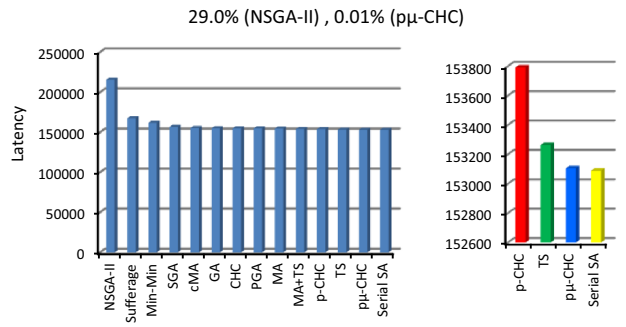


Fig. 22 Best latency values of different algorithms for u_c_hilo.0

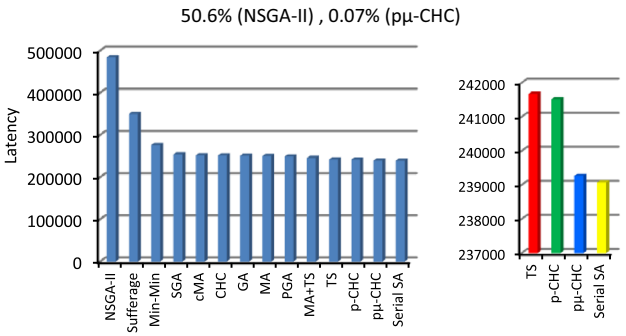


Fig. 23 Best latency values of different algorithms for u_c_lohi.0

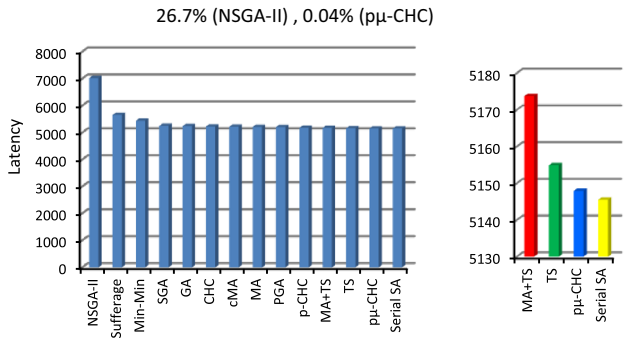


Fig. 24 Best latency values of different algorithms for u_c_lolo.0

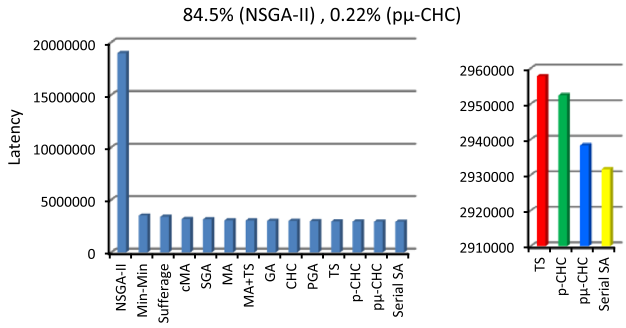


Fig. 25 Best latency values of different algorithms for u_i_hihi.0

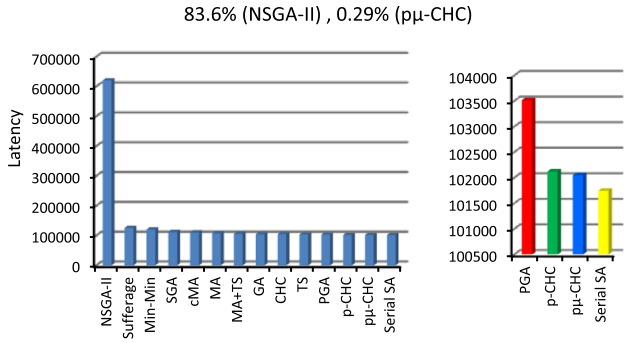


Fig. 26 Best latency values of different algorithms for u_i_lohi.0

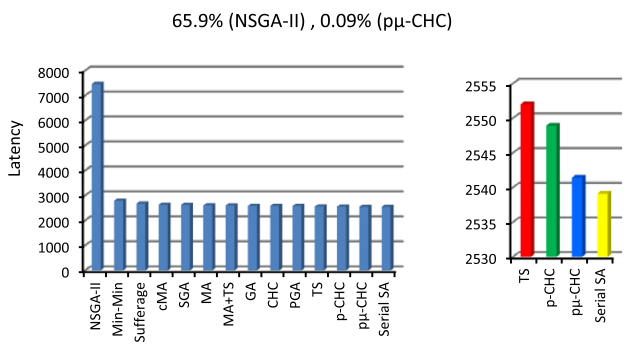


Fig. 27 Best latency values of different algorithms for u_i_lolo.0

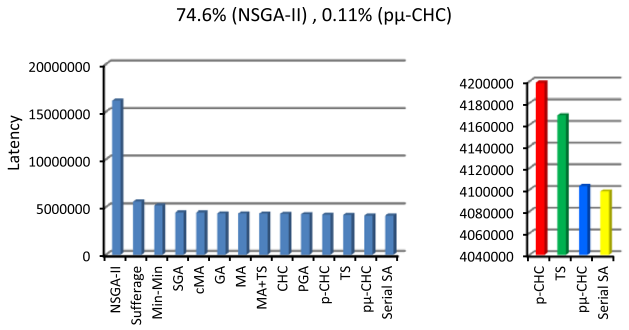


Fig. 28 Best latency values of different algorithms for u_s_hihi.0

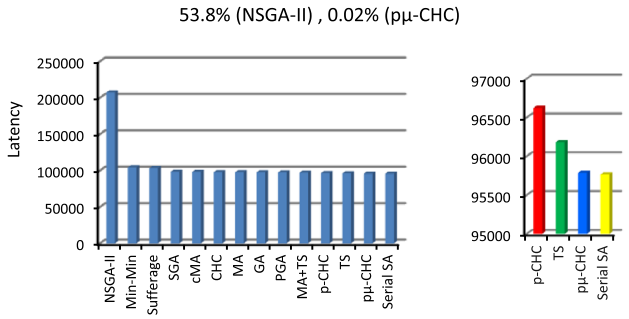


Fig. 29 Best latency values of different algorithms for u_s_hilo.0

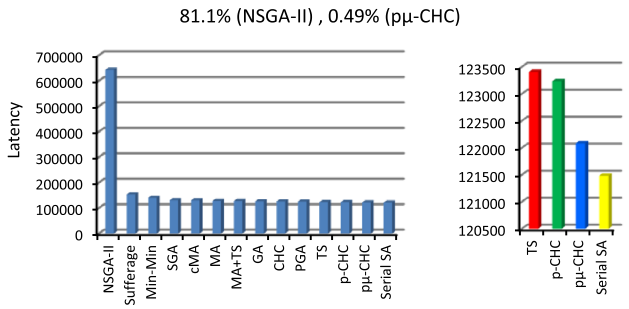


Fig. 30 Best latency values of different algorithms for u_s_lohi.0

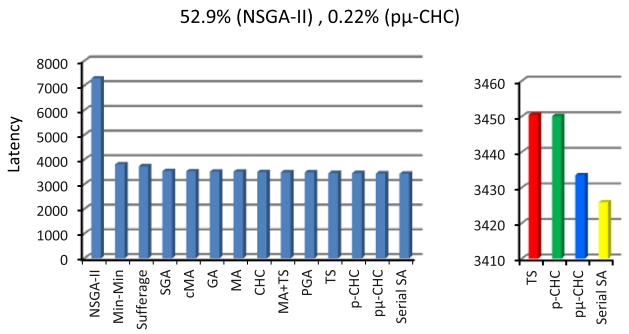


Fig. 31 Best latency values of different algorithms for u_s_lolo.0

rather than doing useful work, and it includes factors such as task start-up and termination times, synchronization, the software overhead introduced by the parallel libraries and operating system. Not surprisingly, this overhead increases

Algorithm 4: Pseudocode of the Multi-Start Parallel SA

Input: $Initial\ Temperature(T), Cooling\ Rate(\alpha),$
$Final\ Temperature(T_{final}), Inner\ Loop\ Constant(ILC),$
$Number\ Of\ Tasks(NOT), Number\ Of\ Nodes(NON),$
$Node\ Execution\ Times(NET), Expected\ Time\ to\ Compute(ETC),$
$Thermal\ Equilibrium\ Counter\ Upper\ Limit(TECUL),$
$Exploitation\ Exploration\ Ratio(EER),$
$Outer\ Loop\ Stop\ Counter\ Upper\ Limit(OLSCUL),$
$Threshold, Hot\ Enough\ Constant(HEC), Number\ Of\ Threads$
Output: $Best\ Solution(S_{best}), Best\ Cost(C_{best})$

$T \leftarrow FindInitialAnnealingTemperature()$
$S_{best} \leftarrow Create2DEmtpyArray(numberOfThreads, NOT)$
$C_{best} \leftarrow Create1DEmtpyArray(numberOfThreads)$
#pragma omp parallel num_threads$(numberOfThreads)$
  $threadID \leftarrow$ ***omp_get_thread_num***$()$
  $T_{threadID} \leftarrow T$
  $S_{cur} \leftarrow Create1DEmtpyArray(NOT)$
  $NET \leftarrow Create1DEmtpyArray(NON, 0)$
  $CreateARandomSchedule(S_{cur}, NON)$
  $NET \leftarrow CalculateNodeExecutionTimes(ETC, S_{cur})$
  $C_{cur} \leftarrow FindMaximum(NET)$
  $S_{best}[threadID] \leftarrow S_{cur}$
  $C_{best}[threadID] \leftarrow C_{cur}$
  $C_{previous} \leftarrow -1.0$
  $outerLoopStopCounter \leftarrow 0$
  while$(T_{threadID} > T_{final})$
    $thermalEquilibriumCounter \leftarrow 0$
    for $i \leftarrow 1$ ***to*** ILC ***by*** 1
      $nextSolutionAccepted \leftarrow false$
      $r_{01} \leftarrow GenerateARandomRealNumber(0,1)$
      if$(r_{01} < EER)$
        $exploitationOrExploration \leftarrow true$
        $r_1 \leftarrow GenerateARandomInteger(1, NOT)$
        $r_2 \leftarrow GenerateARandomInteger(1, NOT)$
        $NET[S_{cur}[r_1]] \leftarrow NET[S_{cur}[r_1]] - ETC[r_1][S_{cur}[r_1]]$
        $NET[S_{cur}[r_2]] \leftarrow NET[S_{cur}[r_2]] - ETC[r_2][S_{cur}[r_2]]$
        $Swap(S_{cur}[r_1], S_{cur}[r_2])$
        $NET[S_{cur}[r_1]] \leftarrow NET[S_{cur}[r_1]] + ETC[r_1][S_{cur}[r_1]]$
        $NET[S_{cur}[r_2]] \leftarrow NET[S_{cur}[r_2]] + ETC[r_2][S_{cur}[r_2]]$
      else
        $exploitationOrExploration \leftarrow false$
        $r_1 \leftarrow GenerateARandomInteger(1, NOT)$
        $r_3 \leftarrow GenerateARandomInteger(1, NON)$
        $n_{temp} \leftarrow S_{cur}[r_1]$
        $NET[n_{temp}] \leftarrow NET[n_{temp}] - ETC[r_1][n_{temp}]$
        $S_{cur}[r_1] \leftarrow r_3$
        $NET[r_3] \leftarrow NET[r_3] + ETC[r_1][r_3]$
      End if
      $C_{next} \leftarrow FindMaximum(NET)$
      $\Delta C \leftarrow C_{next} - C_{cur}$
      if$(\Delta C < 0)$
        $thermalEquilibriumCounter++$
        $nextSolutionAccepted \leftarrow true$
        $C_{cur} \leftarrow C_{next}$
        if$(C_{next} < C_{best})$
          $S_{best}[threadID] \leftarrow S_{cur}$
          $C_{best}[threadID] \leftarrow C_{cur}$
        End if
      else
        $r_{01} \leftarrow GenerateARandomRealNumber(0,1)$

Fig. 32 The pseudocode of the MSSA.

```
51:                 if(r01 < e^(−ΔC / T_threadID))
52:                     nextSolutionAccepted ← true
53:                     C_cur ← C_next
54:                 End if
55:             End if
56:             if(nextSolutionAccepted is equal to false)
57:                 if(exploitationOrExploration is equal to true)
58:                     NET[S_cur[r1]] ← NET[S_cur[r1]] − ETC[r1][S_cur[r1]]
59:                     NET[S_cur[r2]] ← NET[S_cur[r2]] − ETC[r2][S_cur[r2]]
60:                     Swap(S_cur[r1], S_cur[r2])
61:                     NET[S_cur[r1]] ← NET[S_cur[r1]] + ETC[r1][S_cur[r1]]
62:                     NET[S_cur[r2]] ← NET[S_cur[r2]] + ETC[r2][S_cur[r2]]
63:                 else
64:                     NET[r3] ← NET[r3] − ETC[r1][r3]
65:                     S_cur[r1] ← n_temp
66:                     NET[n_temp] ← NET[n_temp] + ETC[r1][n_temp]
67:                 End if
68:             End if
69:             if(thermalEquilibriumCounter is equal to TECUL)
70:                 break the inner for loop
71:             End if
72:         End for
73:         if(C_previous is equal to C_cur)
74:             outerLoopStopCounter + +
75:             if(outerLoopStopCounter is equal to OLSCUL)
76:                 break the outer while loop
77:             End if
78:         else
79:             outerLoopStopCounter ← 0
80:         End if
81:         C_previous ← C_cur
82:         T_threadID ← α · T_threadID
83:     End while
84: End pragma
85: threadIndex ← FindIndexOfMaximum(C_best)
86: return S_best[threadIndex] and C_best[threadIndex]
```

Fig. 32 continued

the program's runtime. Moreover, the overhead is directly proportional to the number of threads utilized. Therefore, the number of inner loop iterations (*ILC*), one of the essential search parameters of the simulated annealing method, has been reduced to compensate for increased runtime. For example, the β values are selected as 3.75 and 1.5 for the serial SA and MSSA (8 threads), respectively, as listed in Table 6.

7.1 Discussion of the parallel implementation's results

In Table 11, the best and average latency values of the serial and parallel SA (2, 4, and 8 threads) for the benchmarks containing 512 tasks and 16 nodes are compared with the results of the pμ-CHC ([13]) algorithm. A similar comparison is made in Table 12, with only one difference, where Table 12 lists the results for the benchmarks composed of 1024 tasks and 32 nodes. In both tables, the red and blue italic fonts indicate the methods that report the best and best average latencies, respectively. As can be noticed from both tables, an SA implementation (either serial or parallel) surpasses the pμ-CHC ([13]) in a comparison based on the best latencies. In other words, there is not even a single red value in columns 2 of both tables, and the red values are only scattered throughout the SA columns. A similar situation is observed when the average latencies are in question. An SA implementation finds the best average latencies for eleven benchmark. However, it falls slightly behind the pμ-CHC ([13]) for only one benchmark (*u_c_hilo.0*), according to Table 11. On the other hand, the SA method outperforms the pμ-CHC ([13]) in terms of the best and average latencies, as shown in Table 12. It can be concluded from both tables that the serial SA mostly finds the best latencies, whereas a parallel implementation is good at obtaining the best average latencies.

Figure 33 is included to visualize the data listed in Table 11 corresponding to the running benchmark *u_i_hilo.0*. It consists of two parts. In the lower half of the figure, the best latency values of the pμ-CHC ([13]) and the SA implementations (both serial and parallel) are shown. On the other hand, in the upper half the figure, the relative performance increase by the associated SA metaheuristic over the pμ-CHC ([13]) is depicted. For example, the percentage of improvement of the serial SA over the pμ-CHC ([13]) is 0.11% for *u_i_hilo.0,* as shown in Fig. 33. A similar figure is created for each of the remaining eleven benchmarks represented by Figs. 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44.

The theoretical optimal values (the minimal latency values) obtained for the benchmarks in the Braun dataset with a dynamic programming approach are called lower bound or LB [34]. The gap between the best latency of an SA implementation (serial or parallel) and the optimum LB is calculated using the formula given by (10).

$$\boldsymbol{GAP(\%)} = \frac{\boldsymbol{SA - LB}}{\boldsymbol{SA}} \cdot \mathbf{100} \tag{10}$$

Tables 13 and 14 list the gap values for the benchmarks with the size of 512×16 and 1024×32, respectively. According to Table 13, the gap in all SA implementations is less than 1% for all benchmarks. On the other hand, the gap increases when the benchmark size gets bigger. For example, the gap values range from 0.70% to 1.80% for the serial SA, according to Table 14. This observation indicates that the 90-s time constraint needs to be relaxed for bigger benchmarks to allow the SA approach to discover the search space more thoroughly.

Table 11 Best/average latencies of pμ-CHC and various SA implementations (benchmark size: 512 × 16)

Benchmark	pμ-CHC ([13])		Serial SA		Parallel SA (2 Threads)		Parallel SA (4 Threads)		Parallel SA (8 Threads)	
	Best	Average	Best	Average	Best	Average	Best	Average	Best	Average
u_c_hihi.0	7381570.0	7394702.7	7377378.2	7400359.9	7381181.3	7393090.7	7381036.3	7394480.0	***7376385.2***	***7391308.5***
u_c_hilo.0	153105.4	***153193.7***	153088.7	153334.7	153090.9	153345.5	153095.9	153292.3	***153071.5***	153314.0
u_c_lohi.0	239260.0	239706.2	***239069.9***	240062.8	239117.9	239765.0	239165.5	239651.5	239114.9	***239575.6***
u_c_lolo.0	5147.9	5152.3	***5145.5***	5150.6	5147.0	5151.1	5145.9	***5150.4***	5147.4	5150.8
u_i_hihi.0	2938380.8	2947896.4	***2931630.9***	2953919.5	2933767.7	2947620.8	2933858.1	2945171.5	2934715.8	***2944859.4***
u_i_hilo.0	73378.0	73531.4	***73294.3***	73526.3	73338.2	73488.1	73366.1	73478.1	73367.4	***73472.0***
u_i_lohi.0	102050.6	102402.8	101746.3	102415.3	***101737.7***	102234.9	101841.8	102116.1	101773.9	***102056.1***
u_i_lolo.0	2541.4	2547.1	***2539.1***	2547.1	2539.5	2546.2	2540.4	***2545.5***	2540.9	2545.9
u_s_hihi.0	4103500.3	4123537.3	4098580.9	4121642.4	***4093541.4***	***4113577.9***	4097940.9	4115372.3	4096688.9	4113997.2
u_s_hilo.0	95787.4	96020.5	***95764.9***	96046.4	95778.6	95970.2	95770.3	***95942.2***	95786.8	95973.7
u_s_lohi.0	122083.3	122744.4	***121484.8***	122332.4	121694.5	122197.5	121605.4	122031.5	121589.1	***121996.8***
u_s_lolo.0	3433.5	3438.3	3425.9	3434.1	3428.9	3433.1	***3425.6***	***3431.8***	3428.3	3433.3

Table 12 Best/average latencies of pμ-CHC and various SA implementations (benchmark size: 1024 × 32)

Benchmark	pμ-CHC ([13])		Serial SA		Parallel SA (2 Threads)		Parallel SA (4 Threads)		Parallel SA (8 Threads)	
	Best	Average	Best	Average	Best	Average	Best	Average	Best	Average
B.u_c_hihi.0	6049220.5	6052322.9	***6027322.9***	6038444.2	6028312.1	***6037901.8***	6027409.9	6037909.3	6038758.9	6047291.0
B.u_c_hilo.0	59679.5	59730.6	59392.4	***59486.0***	59381.0	59487.2	***59377.0***	59489.9	59449.1	59542.5
B.u_c_lohi.0	210005.1	210370.4	209472.4	209908.4	***209420.0***	***209848.1***	209645.7	209929.1	209891.4	210105.6
B.u_c_lolo.0	2100.0	2103.3	***2092.8***	2096.6	2093.4	***2096.4***	2093.9	2096.8	2094.9	2097.8
B.u_i_hihi.0	1616697.4	1621628.0	***1594892.5***	1609082.3	1595807.3	1606520.9	1595878.2	***1604198.6***	1595897.7	1607967.9
B.u_i_hilo.0	14993.2	15047.8	***14822.1***	14974.6	14830.3	***14924.2***	14832.8	14926.6	14857.6	14927.3
B.u_i_lohi.0	49060.5	49351.9	***48334.6***	48807.0	48392.7	48960.3	48444.4	***48755.2***	48451.2	48793.4
B.u_i_lolo.0	487.5	491.8	485.0	487.7	485.3	487.9	***484.7***	***487.2***	485.4	489.1
B.u_s_hihi.0	3255266.8	3272088.3	***3235500.9***	3270337.5	3249974.3	3264401.5	3250374.2	***3264274.1***	3255071.6	3269501.7
B.u_s_hilo.0	34675.2	34747.2	34560.2	34694.1	34560.4	***34671.9***	***34556.5***	34677.8	34610.6	34777.9
B.u_s_lohi.0	110749.7	111068.5	***110326.2***	***111058.2***	110404.7	111061.2	110492.3	110975.9	110602.3	111292.6
B.u_s_lolo.0	*1153.1*	*1158.0*	1143.9	1148.6	***1142.8***	***1148.0***	1144.9	1148.5	1147.5	1152.7

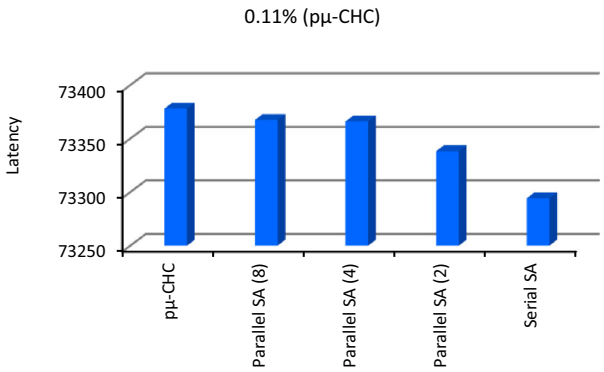


Fig. 33 Best latency values for u_i_hilo.0.

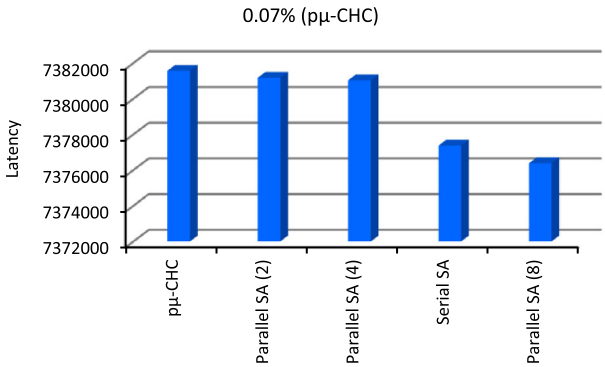


Fig. 34 Best latency values for u_c_hihi.0

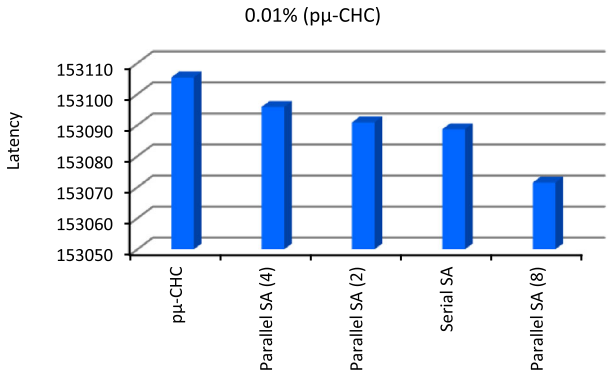


Fig. 35 Best latency values for u_c_hilo.0

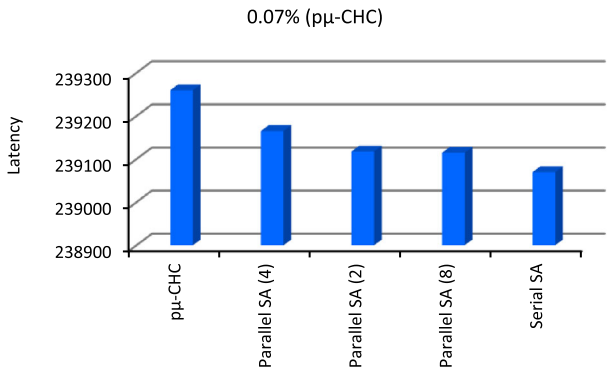


Fig. 36 Best latency values for u_c_lohi.0

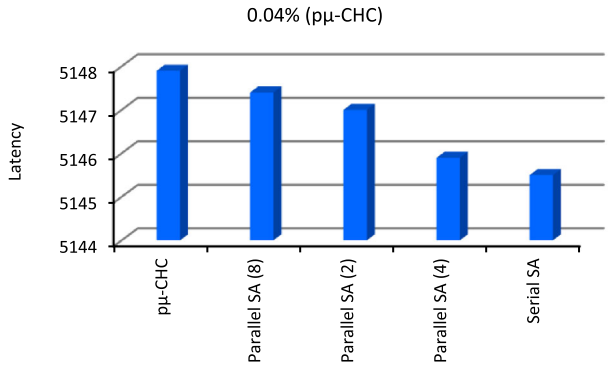


Fig. 37 Best latency values for u_c_lolo.0

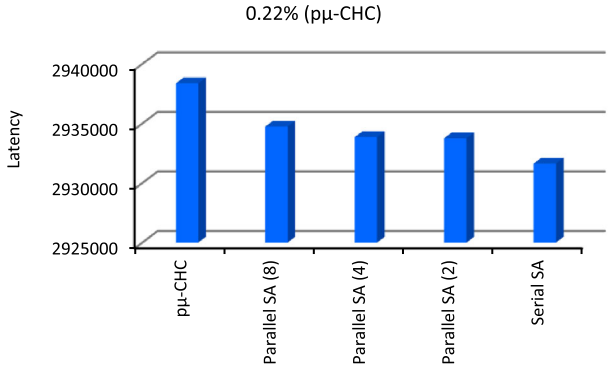


Fig. 38 Best latency values for u_i_hihi.0

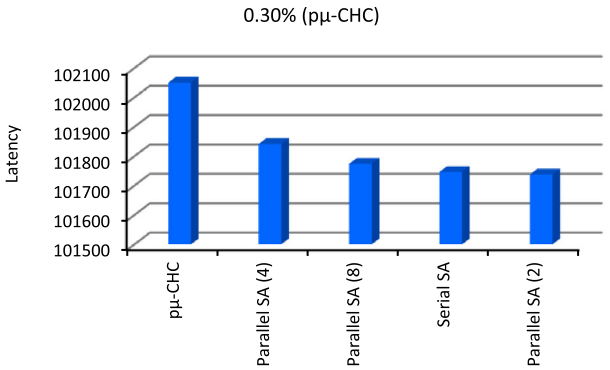


Fig. 39 Best latency values for u_i_lohi.0

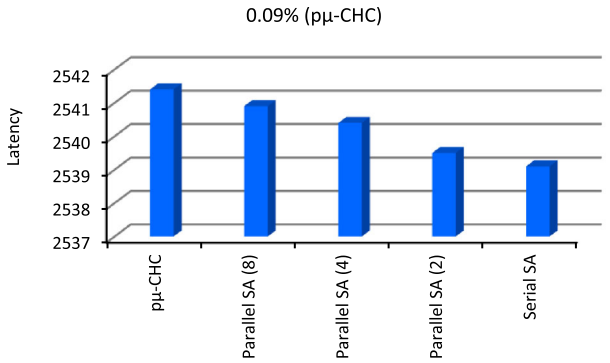


Fig. 40 Best latency values for u_i_lolo.0

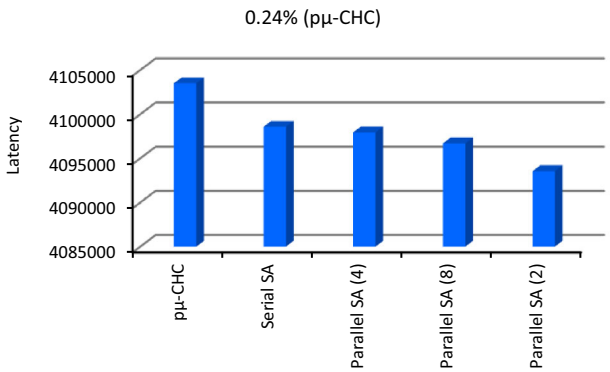


Fig. 41 Best latency values for u_s_hihi.0

8 Conclusion

The advancement of technologies, the global spread of the Internet and the ease of access to the information through the Internet, the widespread use of computers, mobile and IoT devices, economical access to storage areas, and the increase in the use of high-speed networks have brought the concept of big data that we frequently hear. This so-called big data is useless in its raw form and cannot be processed with traditional algorithms/hardware. Therefore,

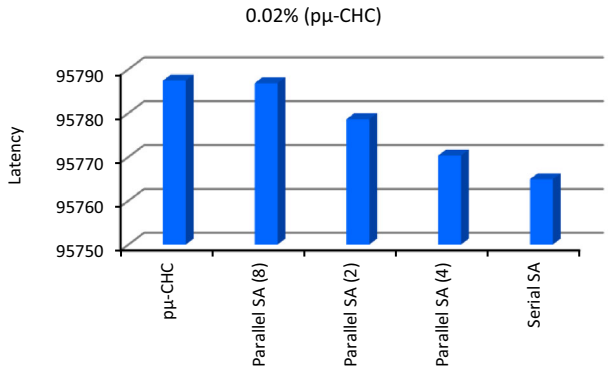


Fig. 42 Best latency values for u_s_hilo.0

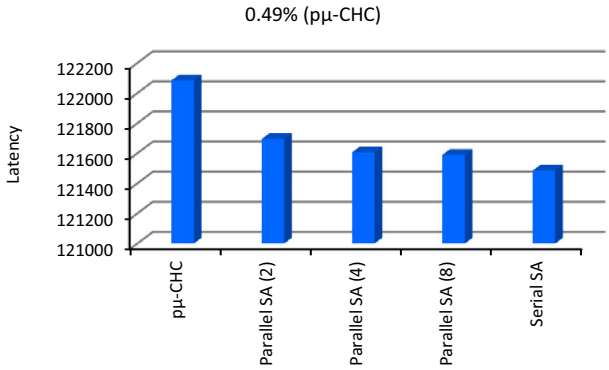


Fig. 43 Best latency values for u_s_lohi.0

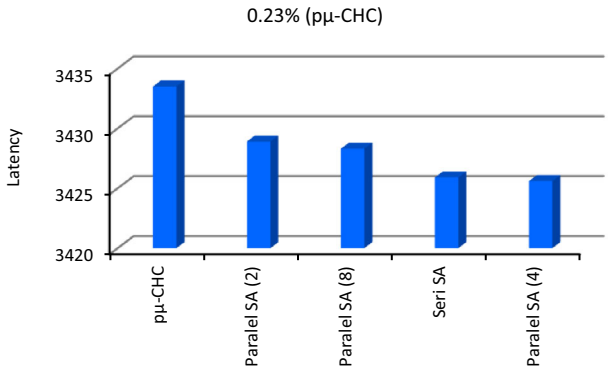


Fig. 44 Best latency values for u_s_lolo.0

storing, processing, and analyzing this data in a timely manner requires distributed data clusters that lean towards heterogeneous in terms of the computational capabilities of the cluster nodes in the long run. Such data clusters are frequently encountered in the infrastructures that provide cloud services. One of the most critical problems facing these cloud platforms is referred to as cluster-based task scheduling. In this context, task scheduling is simply the name of the task-cluster node mapping process that will allow the last task to complete its execution as early as possible. Due to the heterogeneous nature of the clusters mentioned above, the execution time of a task depends on the cluster node that is assigned. Not surprisingly, this dependency increases the complexity of the scheduling considerably, and consequently, exploring the entire search space by testing all possible assignments using a brute force algorithm becomes impractical. Due to the limitations of the other related approaches, heuristics or metaheuristics are mostly employed to find the optimal or a near-optimal solution in a reasonable time.

In this study, a simulated annealing-based metaheuristic for cluster-based task scheduling was developed, and its effectiveness was shown through rigorous experiments. The serial and parallel (shared memory) versions of the proposed approach were converted into a computer program using the C++ programming language. For the parallel version, the OpenMP library was also utilized. The efficiency of the proposed technique was tested with twelve popular benchmarks created by the Braun model. It was observed that both the serial and parallel versions of the approach produced much better results within the 90-s time constraint than the best latency values ever reported in the literature for all comparisons. For example, the percentage of improvement of the serial version ranges from 0.01% to 0.49%. To decrease the execution time of the developed computer program and improve the quality of the scheduling solutions, different random number generation and perturbation techniques, data structures, early loop termination conditions, exploitation-exploration rates, and compiler effects were also analyzed in detail within the scope of this study. We believe that such a comprehensive methodology will be a guide for other simulated annealing-based research.

On the other hand, the proposed technique suffers from the limitation caused by the benchmark files filled with the static ETC times. In other words, the ETC values that do not change dynamically are read from a file and processed by our scheduling tool. This assumption poses a restriction on real cloud environments, where tasks (jobs) that need to be scheduled on the cloud resources (virtual machines) arrive stochastically and their execution time cannot be obtained or predicted in advance. Therefore, a new benchmark set with task execution times measured on a real cloud-based system is needed for this particular research field.

Given the efficiency of the proposed approach supported by the experimental results, new avenues for possible future research are available. In the current implementation, the initial solution is created randomly. Analyzing the effect of the utilization of a heuristic to form the initial solution might be a promising area of future research. The current parallel implementation is asynchronous. In other words, no communication happens between the threads during the execution. On the other hand, there are also

Table 13 The gap between the LB and various SA implementations (benchmark size: 512 × 16)

Benchmark	LB ([2])	Serial SA		Parallel SA (2 threads)		Parallel SA (4 threads)		Parallel SA (8 threads)	
	Optimum	Best	GAP (%)	Best	GAP (%)	Best	GAP (%)	Best	GAP (%)
u_c_hihi.0	7346524.2	7377378.2	0.41	7381181.3	0.46	7381036.3	0.46	7376385.2	0.40
u_c_hilo.0	152700.4	153088.7	0.25	153090.9	0.25	153095.9	0.25	153071.5	0.29
u_c_lohi.0	238138.1	239069.9	0.38	239117.9	0.40	239165.5	0.42	239114.9	0.40
u_c_lolo.0	5132.8	5145.5	0.24	5147.0	0.27	5145.9	0.25	5147.4	0.28
u_i_hihi.0	2909326.6	2931630.9	0.76	2933767.7	0.83	2933858.1	0.83	2934715.8	0.86
u_i_hilo.0	73057.9	73294.3	0.32	73338.2	0.38	73366.1	0.42	73367.4	0.42
u_i_lohi.0	101063.4	101746.3	0.67	101737.7	0.66	101841.8	0.76	101773.9	0.69
u_i_lolo.0	2529.0	2539.1	0.39	2539.5	0.41	2540.4	0.44	2540.9	0.46
u_s_hihi.0	4063564.7	4098580.9	0.85	4093541.4	0.73	4097940.9	0.83	4096688.9	0.80
u_s_hilo.0	95419.0	95764.9	0.36	95778.6	0.37	95770.3	0.36	95786.8	0.38
u_s_lohi.0	120452.3	121484.8	0.84	121694.5	1.02	121605.4	0.94	121589.1	0.93
u_s_lolo.0	3414.8	3425.9	0.32	3428.9	0.41	3425.6	0.31	3428.3	0.39

Table 14 The gap between the LB and various SA implementations (benchmark size: 1024 × 32)

Benchmark	LB ([2])	Serial SA		Parallel SA (2 threads)		Parallel SA (4 threads)		Parallel SA (8 threads)	
	Optimum	Best	GAP (%)	Best	GAP (%)	Best	GAP (%)	Best	GAP (%)
B.u_c_hihi.0	5980871.9	6027322.9	0.77	6028312.1	0.78	6027409.9	0.77	6038758.9	0.95
B.u_c_hilo.0	58942.5	59392.4	0.75	59381.0	0.73	59377.0	0.73	59449.1	0.85
B.u_c_lohi.0	207892.8	209472.4	0.75	209420.0	0.72	209645.7	0.83	209891.4	0.95
B.u_c_lolo.0	2078.0	2092.8	0.70	2093.4	0.74	2093.9	0.75	2094.9	0.80
B.u_i_hihi.0	1567178.7	1594892.5	1.73	1595807.3	1.79	1595878.2	1.79	1595897.7	1.79
B.u_i_hilo.0	14582.3	14822.1	1.61	14830.3	1.67	14832.8	1.68	14857.6	1.85
B.u_i_lohi.0	47606.9	48334.6	1.50	48392.7	1.62	48444.4	1.72	48451.2	1.74
B.u_i_lolo.0	477.4	485.0	1.56	485.3	1.62	484.7	1.50	485.4	1.64
B.u_s_hihi.0	3178482.2	3235500.9	1.76	3249974.3	2.19	3250374.2	2.21	3255071.6	2.35
B.u_s_hilo.0	33948.7	34560.2	1.76	34560.4	1.76	34556.5	1.75	34610.6	1.91
B.u_s_lohi.0	108330.1	110326.2	1.80	110404.7	1.87	110492.3	1.95	110602.3	2.05
B.u_s_lolo.0	1128.1	1143.9	1.38	1142.8	1.28	1144.9	1.46	1147.5	1.69

synchronous parallel simulated annealing methods in the literature that allow exchanging information. Evaluating the effectiveness of these synchronous methods is another area for further investigation. As discussed in the previous paragraph, our technique utilizes the static ETC times from the Braun dataset while performing a task-node mapping. We believe that our methodology can be employed to provide the required data to train a machine learning algorithm that maps the tasks to the nodes in a real cloud-based system where only a dynamic mapping is suitable. This might be an interesting topic for another future work.

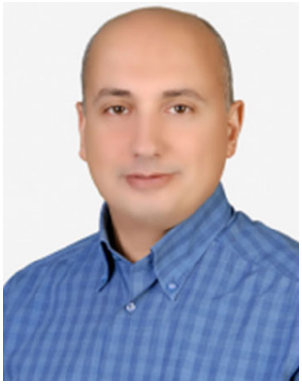
References

1. Gogos, C., Valouxis, C., Alefragis, P., Goulas, G., Voros, N., Housos, E.: Scheduling independent tasks on heterogeneous processors using heuristics and column pricing. Future Generat. Comput. Syst. **60**, 48–66 (2016)
2. Heterogeneous Computing Scheduling Problem. [Online], Available:https://www.fing.edu.uy/inco/grupos/cecal/hpc/HCSP/index.htm (2020)
3. Massobrio, R., Dorronsoro, B., Nesmachnow, S.: Virtual savant for the heterogeneous computing scheduling problem. *International Conference on High Performance Computing and Simulation (HPCS)*, pp. 821–827, (2018)
4. Wang, X., Qing-dao-er-ji, R.: Application of optimized genetic algorithm based on big data in bus dynamic scheduling. Clust. Comput. **22**(9), 15439–15446 (2019)

5. Attaran, M., Woods, J.: Cloud computing technology: improving small business performance using the internet. J. Small Bus. Entrepren. **31**(6), 495–519 (2019)
6. Fernández, A., del Río, S., López, V., Bawakid, A., del Jesus, M., Benítez, M.J., Herrera, F.: Big data with cloud computing: an insight on the computing environment, MapReduce, and programming frameworks. Wiley Interdisc. Rev.: Data Min. Knowledge Discov. **4**(5), 380–409 (2014)
7. Rekha, P.M., Dakshayini, M.: Efficient task allocation approach using genetic algorithm for cloud environment. Clust. Comput. **22**(21), 1241–1251 (2019)
8. Li, J., Qiu, M., Ming, Z., Quan, G., Qin, X., Gu, Z.: Online optimization for scheduling preemptable tasks on iaas cloud systems. J. Parall. Distribut. Comput. **72**(5), 666–677 (2012)
9. Syed, H.A., Kamran, R., Usman, A., Syed, S. S. A., Manzoor, H.: Cloud task scheduling using nature inspired meta-heuristic algorithm. *International Conference on Open Source Systems and Technologies (ICOSST)*, (2015)
10. Wolfram Alpha. [Online], Available: http://www.wolframalpha.com (2020)
11. Date, K., Nagi, R.: GPU-accelerated hungarian algorithms for the linear assignment problem. Parall. Comput. **57**, 52–72 (2016)
12. Sivapuram, R., Picelli, R.: Topology optimization of binary structures using integer linear programming. Finite Elem. Anal. Des. **139**, 49–61 (2018)
13. Nesmachnow, S., Cancela, H., Alba, E.: A parallel micro evolutionary algorithm for heterogeneous computing and grid scheduling. Appl. Soft Comput. **12**(2), 626–639 (2012)
14. Erbay, H., Kör, H.: Big data and its analysis. *International Conference on Science and Technology (ICONST)*, (2016)
15. Patel, A.B., Birla, M., Nair, U.: Addressing big data problem using hadoop and map reduce. *Nirma University International Conference on Engineering (NUICONE)*, (2012)
16. Arora, R., Parashar, A.: Secure user data in cloud computing using encryption algorithms. Int. J. Eng. Res. Appl. **3**(4), 1922–1926 (2013)
17. Hashem, I.A.T., Yaqoob, I., Anuar, N.B., Mokhtar, S., Gani, A., Khan, S.U.: The rise of big data on cloud computing: review and open research issues. Inform. Syst. **47**, 98–115 (2014)
18. Okutucu, B.O.: Cloud computing and cloud technologies. Master Thesis, Dept. of Computer Eng., Okan Univ., Istanbul (2012)
19. Vincy, V.A.G., Karthija, T., Sunil, J.: Understanding hadoop framework through single-node cluster installation. *International Conference on Recent Advances in Energy-efficient Computing and Communication (ICRAECC)*, (2019)
20. Diedhiou, C., Carpenter, B., Esmeli, R.: Comparison of platforms for recommender algorithm on large datasets. *Imperial College Computing Student Workshop (ICCSW)*, (2018)
21. Senthilkumar, M., Ilango, P.: A survey on job scheduling in big data. Cybern. Inform. Technol. (CIT) **16**(3), 1311–9702 (2016)
22. Nagina, D., Dhingra, S.: Scheduling algorithms in big data: a survey. Int. J. Eng. Comput. Sci. **5**(8), 17737–17743 (2016)
23. Flórez, E., Barrios, C. J., Pecero, J. E.: Methods for job scheduling on computational grids: review and comparison. *High Performance Computing,* Communications in Computer and Information Science, Springer, pp. 19–33, (2015)
24. Mishra, A., Trivedi, P.: Benchmarking the contention aware nature inspired metaheuristic task scheduling algorithms. Clust. Comput. **23**(1), 537–553 (2020)
25. Merendino, S., Celebi, M.E.: A simulated annealing clustering algorithm based on center perturbation using gaussian mutation. *Twenty-Sixth International Florida Artificial Intelligence Research Society Conference*, (2013)
26. Kirkpatrick, S., Gelatt Jr., C.D., Vecchi, M.P.: Optimization by Simulated Annealing. Science **220**(4598), 671–680 (1983)
27. Yang, X.-S. Nature-inspired optimization algorithms. *Elsevier*, 2nd Edition, (2020)
28. Alobaedy M.M., Ku-Mahamud, K. R.: Strategic oscillation for exploitation and exploration of ACS algorithm for job scheduling in static grid computing. *Second International Conference on Computing Technology and Information Management (ICCTIM),* pp. 87–92, (2015)
29. Xhafa, F., Durresi, A., Barolli, L.: Batch mode scheduling in grid systems. Int. J. Web Grid Serv. (IJWGS) **3**(1), 19–37 (2007)
30. Braun, T.D., Siegel, H.J., Beck, N., Bölöni, L.L., Maheswaran, M., Reuther, A.I., Robertson, J.P., Theys, M.D., Yao, B., Hensgen, D., Freund, R.F.: A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. J Parall. Distrib. Comput. **61**(6), 810–837 (2001)
31. Xhafa, F., Alba, E., Dorronsoro, B., Duran, B., Abraham, A.: Efficient batch job scheduling in grids using cellular memetic algorithms. *Metaheuristics for Scheduling in Distributed Computing Environments*, Springer, Berlin pp. 273–299, (2008)
32. Xhafa, F., Carretero, F. J., Alba, E., Dorronsoro, B.: Design and evaluation of tabu search method for job scheduling in distributed environments. *22nd IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, (2008)
33. Xhafa, F.: A Hybrid Evolutionary Heuristic for Job Scheduling on Computational Grids, Hybrid Evolutionary Algorithms, Springer-Verlag pp.269–311 (2007)
34. Nesmachnow, S., Cancela, H., Alba, E.: Heterogeneous computing scheduling with evolutionary algorithms. Soft. Comput. **15**(4), 685–701 (2010)
35. Xhafa, F., Duran, B., Abraham, A., Dahal, K.P.: Tuning struggle strategy in genetic algorithms for scheduling in computational grids. Neural Netw. World **18**(3), 209–225 (2008)
36. Subashini, G., Bhuvaneswari, M.C.: A fast and elitist bi-objective evolutionary algorithm for scheduling independent tasks on heterogeneous systems. ICTACT J. Soft Comput. **1**, 9–17 (2010)
37. Panda, S.K., Jana, P.K.: Efficient task scheduling algorithms for heterogeneous multi-cloud environment. J. Supercomput. **71**(4), 1505–1533 (2015)
38. Panda, S.K., Pande, S.K., Das, S.: Task partitioning scheduling algorithms for heterogeneous multi-cloud environment. Arab. J. Sci. Eng. **43**(2), 913–933 (2018)
39. Panda, S.K., Nag, S., Jana, P. K.: A smoothing based task scheduling algorithm for heterogeneous multi-cloud environment. *3rd IEEE International Conference on Parallel, Distributed and Grid Computing (PDGC),* (2014)
40. Al-Qadhi, A.K., Ariffin, A.A., Latip, R., Al-Zubaidi, A., Hamid, W.A.: Two stages transfer algorithm (TSTT) for independent tasks scheduling in heterogeneous computing systems. J. Phys. Conf. Ser. **1018**, 28 (2018)
41. Chen, C., Tiong, L.K.: Using queuing theory and simulated annealing to design the facility layout in an AGV-based modular manufacturing system. Int. J. Prod. Res. **57**(17), 5538–5555 (2019)
42. Li, Y., Qiao, C.: A route optimization method based on simulated annealing algorithm for wind-assisted ships. IOP Conf. Ser. **295**(4), (2019)
43. Vincent, F.Y., Redi, A.P., Hidayat, Y.A., Wibowo, O.J.: A simulated annealing heuristic for the hybrid vehicle routing problem. Appl. Soft Comput. **53**, 119–132 (2017)
44. Peprah, A.K., Appiah, S.K., Amponsah, S.K.: An optimal cooling schedule using a simulated annealing based approach. Appl. Math. **8**(8), 1195–1210 (2017)
45. Tridgell, A., Rosdahl, J.: ccache-a Fast C/C++Compiler Cache, Available: http://ccache.samba.org, (2020)

46. Mcilroy, R., Sevcik, J., Tebbi, T., Titzer, B.L., Verwaest, T.: Spectre is here to stay: an analysis of side-channels and speculative execution. *ArXiv*, (2019)
47. Ahonen, H., de Alvarenga, A.G., Amaral, A.R.: Simulated annealing and tabu search approaches for the corridor allocation problem. Eur. J. Oper. Res. **232**(1), 221–233 (2014)
48. Gülcü, A., Akkan, C.: Robust university course timetabling problem subject to single and multiple disruptions. Eur. J. Oper. Res. **283**(2), 630–646 (2020)
49. Gava, J., Bandiera, V., Reis, R., Ost, L.: Evaluation of compilers effects on OpenMP soft error resiliency. *IEEE Computer Society Annual Symposium on VLSI (ISVLSI),* pp. 259–264, (2019)
50. Balaji, A.N., Porselvi, S., Jawahar, N.: Particle swarm optimization algorithm and multi-start simulated annealing algorithm for scheduling batches of parts in multi-cell flexible manufacturing system. Inte. J. Serv. Operat. Manag. **32**(1), 83–129 (2019)

Esra Celik received her B.Sc. and M.Sc. degree in Computer Engineering from Ataturk University, Erzurum, Turkey, in 2015 and 2018, respectively. Since 2019, she has been pursuing her Ph.D. degree in the Department of Computer Engineering at Ataturk University. Her current research interests include high-level synthesis of digital circuits, combinatorial optimization, metaheuristics, high performance computing, deep learning and machine learning.



Deniz Dal received his B.Sc. degree in Electrical Engineering from Istanbul Technical University, Istanbul, Turkey, in 1996 and his M.Sc. and Ph.D. degrees in Computer Engineering from Syracuse University, Syracuse, NY, in 2001 and 2006, respectively. Since 2007, he has been with the Department of Computer Engineering at Ataturk University, Erzurum, Turkey. His current research interests include high-level synthesis of digital circuits, combinatorial optimization, metaheuristics, and high performance computing.