

Adapting MCP and HLFET Algorithms to Multiple Simultaneous Scheduling

Emilia Popa¹ · Mauro Iacono²  · Florin Pop^{1,3}

Received: 28 February 2017 / Accepted: 24 June 2017 / Published online: 3 July 2017
© Springer Science+Business Media, LLC 2017

Abstract We deal with the following scheduling problem: an infinite number of tasks must be scheduled for processing on a finite number of heterogeneous machines, such as all tasks are sent to execution with a minimum delay. The tasks have causal dependencies and are generated in the context of biomedical applications, and produce results relevant for the medical domain, such as diagnosis support or drug dose adjust measures. The proposed scheduling model had a starting point in two known bounded number of processors algorithms: Modified Critical Path and Highest Level First With Estimated Times. Several steps were added to the original implementation along with a merge stage in order to combine the results obtained for each of the previously scheduled tasks. Regarding the implementation, a simulator was used to analyze and design the scheduling algorithms.

Keywords Scheduling · Map-reduce · Medical systems

✉ Mauro Iacono
mauro.iacono@unicampania.it

Emilia Popa
emilia.popa@upb.ro

Florin Pop
florin.pop@cs.pub.ro; florin.pop@ici.ro

¹ Computer Science Department, University Politehnica of Bucharest, Splaiul Independentei 313, 060042 Bucharest, Romania

² Dipartimento di Matematica e Fisica, Università degli Studi della Campania “Luigi Vanvitelli”, viale Lincoln 5, 81100 Caserta, Italy

³ National Institute for Research and Development in Informatics (ICI), 8-10, Maresal Averescu, 011455 Bucharest, Romania

1 Introduction

The correct exploitation of parallel and distributed computing resources in scientific applications with specific time requirements relies on the capability of producing optimal schedules for the various tasks that compose the application, accounting for a correct evaluation of the interdependencies and of causal effects. Unfortunately, the problem of optimal scheduling is well known to be complex, and may only be reasonably faced, with the exclusion of a few cases, by means of heuristic approaches, that eventually leverage on specific characteristics of the workload. An appropriate tool to cope with this problem is provided by simulation based performance models of parallel and distributed systems and applications [33,38]. Many approaches consider heterogeneous environments (CPUs and GPUs [11,12]).

A relevant formulation of the problem may be given in the following terms: given a finite set of resources, a number of specific applications and the condition that every resource may be responsible of executing an infinite number of tasks, the problem consists in defining a (sub)optimal schedule to send to execution all tasks, while minimizing a local and global penalty measure.

In detail, an application is a sequence of specific tasks, each of which may require a specific set of resources (e.g. a specific computing node instead of any computing node in the system), that results in a set of constraints for the problem, and must be completed before a given deadline, that results in an additional constraint; resources may be used individually or in sets, that add further constraints; and are located on the nodes of the system, each of which may serve a given number of resource requests (and usage) simultaneously; finally, tasks follow a (partial) ordering, that is produced by the application logic, introducing additional constraints. In some cases, some resources may be alternative, and the choice between the alternatives may affect global and local performances (e.g. in-memory data vs data stored in external files), thus heavily impacting the efficiency of a schedule.

An additional aspect that characterizes modern computing infrastructures is the scale of available resources with respect to the past. Infrastructures are shared to a potentially very large number of concurrent users, and applications are structured to exploit, when possible, massive parallelism, to take advantage and to make profitable big data centers, or highly parallel architectures. The most popular approach to ensure scalability, performances and dependability on large scale computing infrastructures is based on the cloud architecture. In practice, the massive parallelism allowed by modern computing infrastructures suggests a coordinated scheduling of many concurrent computing processes: large parallelism is the most common situation, and scheduling problems may be analyzed by giving as a fact that the size of the schedules may be relevant. We hypothesized that, on these premises, there may be an increase in efficiency if scheduling is simultaneously performed for all the applications that are under the scheduler responsibility, instead of considering a separate scheduling for each application [1,5].

Consequently, in this paper we present two modified algorithms, stemmed from an application case that will be used as case study, that extend to the case of multiple simultaneous schedules two existing solutions in literature, namely MCP and HLFET. The effectiveness of our solution, with respect to the specific class of workloads

of interest, is evaluated by means of a simulative approach, that is based on a real case. These two algorithms, and their performance evaluation by means of simulation, constitute the main contribution of this paper: the solution generalizes a problem that stemmed from a real case study, in which the goal is the improvement of overall processing time for a healthcare system.

This paper is organized as follows: next Section presents some related works; Sect. 3 describes the approach followed to improve the two scheduling algorithms, followed by a case study in the subsequent Section and the evaluation of the results obtained on the case study; conclusions close the paper.

2 Related Works

2.1 The DAG Model

In the following, the application to be scheduled will be described in abstract form as a Direct Acyclic Graph (DAG). In our model the task graph is represented by $G(V, E, c, \tau)$ where: V is a set of nodes (tasks), so we have v nodes. We will refer to the nodes using the n_1, n_2, \dots notation; E is a set of directed edges (we have e number of dependencies), noted as $e(n_i, n_j)$; $w : V \rightarrow R_+$ is a function that associates a weight $w(n_i)$ to each node $n_i \in V$; $w(n_i)$ represents the execution time of the task T_i , which is represented by the node n_i in V ; ew_n is a function $ew_n : E \rightarrow R_+$ that associates a weight to a directed edge; if n_i and n_j are two nodes in V , then $ew_n(n_i, n_j)$ denotes the inter-tasks communication time between T_i and T_j (the time needed for data transmission between processors that execute tasks T_i and T_j). When two nodes are scheduled on the same processing element P , the cost of the connecting edge becomes zero. In this model a scheduler is considered efficient if the *makespan* is short and respects resource constrains, such as a limited number of processors, memory capacity, available disk space, etc. Many types of scheduling algorithms for DAGs are based on the list scheduling technique. Each task has an assigned priority, and scheduling is done according to a list priority policy: select the node with the highest priority and assign it to a suitable machine. According to this policy, two attributes are used to assign priorities:

- *t-level* (top-level) for n_i is the weight of the longest path from the source node to n_i :

$$t\text{-level}(n_i) = \max_{n_j \in \text{pred}(n_i)} \{t\text{-level}(n_j) + w(n_j) + ew_n(n_j, n_i)\},$$

- *b-level* (bottom-level) for n_i is the weight of the longest path from n_i to the exit node:

$$b\text{-level}(n_i) = w(n_i) + \max_{n_j \in \text{succ}(n_i)} \{b\text{-level}(n_j) + ew_n(n_i, n_j)\}.$$

The time-complexity for computing *t-level* and *b-level* is $O(v + e)$, so there are no penalties for the scheduling algorithms.

Table 1 Description of former classes for DAG scheduling algorithms

Former class	Description
LNP	Limited number of processors
BNP	Bounded number of processors
UNC	Unbounded number of clusters
TBD	Task duplication based scheduling algorithms
ANP	Arbitrary processors network scheduling algorithms

We can define the *ALAP* (*As Late As Possible*) attribute for a node n_i to measure how far the node's start-time, $st(n_i)$ can be delayed without increasing the *makespan*. This attribute will have an important role for load balancing constrains because it shows if we can delay the execution start of a task T_i :

$$ALAP(n_i) = \min_{n_j \in succ(n_i)} \{ALAP(n_j) - ew_n(n_i, n_j)\} - w(n_i)$$

The critical path (*CP*) is the weight of the longest path in the DAG and it offers an upper limit for the scheduling cost. Algorithms based on *CP* heuristics produce on average the best results. They take into consideration the critical path of the scheduled nodes at each step. However, these heuristics can result in a local optimum, failing to reach the optimal global solution [23]. The *t-level* and the *b-level* are bounded from above by the length of the critical path.

The former class of DAG scheduling algorithms are presented in Table 1.

Table 2 presents a critical analysis of existing DAG scheduling algorithms highlighting the type of the algorithm (list scheduling, clustering, etc), the complexity (where v represents the number of nodes, e represents the number of edges and p is the number of processors), the priority attribute used by the algorithm and the former class.

In this paper we focus on the two main reference algorithms used by our approach, namely Modified Critical Path (MCP) [39] and Highest Level First with Estimated Time (HLFET) [2], both belonging to the Bounded Number of Processors (BNP) [8] family. The BNP scheduling algorithms make the assumption of having a limited number of processors available and also require that the nodes are fully-connected, which means that no attention is paid to link contention or routing strategies used for communication.

2.2 The MCP Algorithm

The Modified Critical Path (MCP) algorithm is one of the most popular algorithms used to schedule DAG on a previously known number of processors. The idea of scheduling a problem represented by a DAG on a number of parallel processors in order to minimize the completion time is known to be as a NP-complete problem in its general form: this is the reason for which heuristic solutions were adopted [23]. The

Table 2 Critical analysis of existing DAG scheduling algorithms

Scheduling algorithm	Type	Complexity	Priority attribute	Former class
Highest Level First with Estimate Times (HLFET) [2]	List scheduling	$O(v^2)$	Static level	BNP
Highest Levels First with No Estimated Times (HLFNET) [2]	List scheduling	$O(v^2)$	Static level	BNP
Modified Critical Path (MCP) [39]	List scheduling	$O(v^2(\log v) + p)$	ALAP	UNC
Earliest Time First (ETF) [16]	List scheduling	$O(pv^2)$	Static level	BNP
Dynamic Level Scheduling (DLS) [34]	List scheduling	$O(pv^3)$	Dynamic level	ANP
Insertion Scheduling Heuristic (ISH) [19]	List scheduling	$O(v^2)$	Static level	BNP
Critical Node Parent Tree (CNPT) [14]	List scheduling	$O(v^2)$	Critical node	BNP
Polynomial-time algorithm (PTA) [15]	Tree-structured DAG	$O(v)$	Linear time algorithm	LNP
Coffman and Graham Algorithm (CGA) [8]	Arbitrary structured graphs	$O(v^2)$	Label numbers	BNP
Interval-Ordered DAGs (IOD) [29]	Arbitrary structured graphs	$O(v + e)$	Linear time algorithm	BNP
Sarkar algorithm (Sarkar) [32]	Clustering algorithm	$O(e(v + e))$	Edges' weights	BNP
Fast Load Balancing (improvement of EFT, w - width of the DAG) (FLB) [31]	List scheduling	$O(wp)$	Static level	BNP
Dominant Sequence Clustering (DSC) [40]	Clustering algorithm	$O((e + v) \log v)$	Static level	UNC
Clustering And Scheduling System II (CASS-II) [24]	Clustering algorithm	$O((e + v) \log v)$	Static level	UNC

Table 2 continued

Scheduling algorithm	Type	Complexity	Priority attribute	Former class
Dynamic Critical Path (DCP) [22]	Lookahead strategy	$O(v^3)$	Mobility attribute	BNP
Mobility Directed algorithm (MD) [23]	Relative mobility	$O(v^2)$	Smallest mobility	UNC
Hybrid Remapper algorithm (HRA) [26]	Dynamic list scheduling	$O(v^3)$	Block partition	BNP
Cluster ready Children First (CCF) [13]	Dynamic list scheduling	$O((e + v) \log v)$	Static level	UNC
Smallest Co-levels First with Estimated Times (SCFET) [2]	List scheduling	$O(v^2)$	Co-level	BNP
Smallest Co-levels First with No Estimated Times (SCFNET) [2]	List scheduling	$O(v^2)$	Co-level	BNP
Edge-zeroing algorithm (EZ) [32]	Clustering algorithm	$O(e(e + v))$	Edges' weights	BNP
Linear Clustering algorithm (LC) [17]	Clustering algorithm	$O(e(e + v))$	Critical path	BNP
Localized Allocation of Static Tasks (LAST) [4]	Not list scheduling	$O(e(e + v))$	Node priority	BNP
Papadimitriou and Yannakakis (PY) [30]	Approximation algorithm	$O(v^2(e + v \log v))$	e-value	TBD
Lower Bound algorithm (LBW) [9]	List scheduling	$O(v^2)$	Lower bound start-time	TBD
Duplication Scheduling Heuristic (DSH) [20]	List scheduling	$O(v^4)$	Node priority	TBD
Bottom-Up Top-Down Duplication Heuristic (extension of DSH) (BTDH) [7]	List scheduling	$O(v^4)$	Node priority	TBD
Linear Clustering with Task Duplication (LCTD) [6]	Clustering algorithm	$O(v^3 \log v)$	Edges among clusters	TBD

Table 2 continued

Scheduling algorithm	Type	Complexity	Priority attribute	Former class
Critical Path Fast Duplication (CPFD) [3]	Tree partitioning	$O(v^4)$	Critical path nodes	TBD
Mapping Heuristic algorithm (MH) [10]	List scheduling	$O(v(p^3v + e))$	Static level	ANP
Bottom-Up algorithm (BU) [27]	List scheduling	$O(v^2 \log v)$	Critical path nodes	ANP
Bubble Scheduling and Allocation (BSA) [21]	Incremental technique	$O(p^2ev)$	Critical path nodes	ANP
Improved Critical Path using Descendant Prediction (t) [35]	List scheduling	$O(v^2)$	ALAP	UNC

MCP algorithm is known to satisfy two important requirements: a good quality of the algorithm and also a reasonable low complexity. The quality refers to the completion time of the parallel program and the complexity is strictly related to its scheduling time [23].

One important notion used by this algorithm is the ALAP time of a node. This is a measure of how long the start-time of a node can be delayed without increasing the schedule length. The MCP algorithm uses the ALAP time of a node as the scheduling priority. The first step of the algorithm is to compute the ALAP times of all the nodes; then, in the second step, it builds a list of nodes, in increasing order of ALAP time. Ties are broken by considering the ALAP time of the children of a node. The MCP algorithm then schedules the nodes on the list one by one, such that a node is scheduled to a processor that allows the earliest start-time, using the insertion approach. The time-complexity of the MCP algorithm is $O(v^2 \log v + p)$ [23].

Simulations have shown that the complexity of the second step of the MCP algorithm may be reduced by using only one level of descendants in order to break ties, instead of using all the descendants. This particular change in the original algorithm did not modify the initial performance, but reduced the complexity of the second step of the algorithm to $O(e + v \log v)$ [23].

2.3 The HLFET Algorithm

The HLFET algorithm is a BNP algorithm which is considered to be one of the simplest algorithms of its kind and it is known as a list scheduling algorithm [23]. One important notion used by this algorithm is the *b-level* (bottom level, or static level) of a node. The *b-level* is bounded from above by the length of the critical path. The first step of HLFET is to compute the *b-level* of each node. The second step is to create a list of the nodes which are ready to be processed in descending order of the *b-level*. This list initially contains only the entry nodes of the graph, that are the nodes which have an in-degree equal to zero. The next two steps are to be computed until all the nodes have been scheduled. Step number four of the algorithm is to schedule the first node of the previously created list to a processing element which ensures the earliest execution. At this step a non-insertion procedure is used. The fifth and last step of the algorithm is to update the list of ready nodes with the nodes which are currently ready to be processed. The HLFET algorithm has a time-complexity of $O(v^2)$, which is lower than MCP time-complexity [23].

3 Proposed Approach

Our approach deals with the simultaneous scheduling of multiple DAGs, exploiting the MCP algorithm and the HLFET algorithm. These algorithms were analyzed and their implementation was adapted to our case, by modifying the generation of the application schedules so that a single, global data structure to support scheduling is created with the information about the DAG nodes of all applications, and adding the necessary handling sections to create and manage this global structure. The rationale is that this collective organization of the schedule may enable an overall performance

increase, because of the presence of a higher number of mutually independent DAG nodes, that potentially allows a better mapping to the resources. As a result, a single, comprehensive scheduling algorithm is run for all the controlled workload, instead of an instance per application. We chose two different basic algorithm in order to verify the fitness of our assumptions, and to compare the effects of the modifications in the two cases.

Both MCP and HLFET have a time-complexity bounded to the number of vertices. Since the DAGs used for testing the final scheduling algorithms had a relative small number of vertices, the interesting issue is to analyze how the number of DAG structures influences the performance of the proposed algorithm. In both cases, the algorithm assumes that the set of all DAGs for the workload has been computed and is hence available.

3.1 The MCP Modified Algorithm

The proposed solution for adapting the MCP scheduling algorithm to schedule multiple DAG structures simultaneously is the addition of an extra step to the initial algorithm, to create a final list of nodes that contains all the nodes from all DAG structures; symmetrically, also the step in which a node is extracted from the list and scheduled to a processor using the non-insertion approach is replaced with the extraction of multiple nodes from the final task list at the same time. The main steps of the algorithm are:

1. **for** each DAG structure:
 - (a) **compute** the ALAP of each node;
 - (b) **create** a list of ALAP nodes and a list of corresponding node ids in ascending order of the ALAP values; ties are broken using the lowest ALAP value of the node children;
 - (c) **create** a hash map using the list of ALAP values as keys and the corresponding node ids as values; if two nodes happen to have the same ALAP, the first in the list is added to the hash map with the initial ALAP value, but the next one is added with a key equal to the initial ALAP + 1;
2. **create** a final hash by merging all the previously computed hash maps, by keeping the key values and creating a corresponding array for the nodes which have the same ALAP value;
3. **sort** the final hash by the key value;
4. **transform** the hash into an array of node arrays, keeping the order of the node lists;
5. **repeat until** all tasks have been selected:
 - (a) **take** the entry with the lowest index and **check** if the number of nodes is lower than the number of processing elements available; if this is the case, **complete** the list by extracting other nodes from the following list of nodes in order to have a minimum length equal to the number of processors;
 - (b) **distribute** the current task list to the processing elements using a round robin approach;
 - (c) **assign** to each new task a processing time using a uniform probability distribution;

- (d) at each processor level, **process** tasks using a FIFO scheduling algorithm to guarantee that the processing order is kept.

3.2 The HLFET Modified Algorithm

Analogously, we propose for the HLFET scheduling algorithm as well a version for scheduling multiple DAG structures simultaneously. As first, an extra step is added, in order to create the final list of tasks containing all the nodes from each of the causal models: this step is similar to the one used for adapting the MCP algorithm. As in the case of the MCP algorithm, the step in which a node was scheduled to an available processor is modified as well. In this case, since the HLFET scheduling algorithm used a non-insertion approach, no other alterations are made to the lists. The main steps of the algorithm are:

1. **for each** DAG structure:
 - (a) **compute** the b-level of each node;
 - (b) **create** an empty ready-list;
 - (c) **add** the root nodes of the graph to the ready-list;
 - (d) **create** lists of nodes with the same value of the b-level and **add** them to the final array of task lists in descending order of the b-level value;
2. **repeat until** all tasks have been scheduled:
 - (a) from the final array of task lists, **extract** the one with the lowest index;
 - (b) **create** a job for each of the tasks in the current task list and **distribute** the tasks to the processing elements using a round robin approach;
 - (c) **assign** each new task a processing time using a uniform probability distribution;
 - (d) at each processor level, **process** tasks using a FIFO scheduling algorithm to guarantee that the processing order is kept.

3.3 Implementation Details

All of the systems components are modeled as Java objects, benefiting of the advantages of Object Oriented Paradigm: the code is easy to be maintained and modified, the system has a modular structure and inheritance makes it easy to expand different functionalities and adapt them in order to fit the simulation requirements for the problem in cause.

The Graph Generator The generator is a module written in Java that can easily be extended with new functionalities and has the following main components:

- *Digraph* This class represents a directed graph with a number of vertices V and a number of edges E , and a list of adjacent nodes for each of the V vertices. It also holds an array of all the nodes in-degree. The private attributes of the class are the number of nodes, the number of edges, a vector of adjacency lists containing the adjacency information for every node in the directed graph and a vector of integer numbers which represent each of the vertices in-degree. In order to access the important attributes of the class mentioned above, the *Digraph* component has several getters defined, one for each of the private attributes. The class, however,

does not allow for this attributes to be changed by any external class, and this is why no setters have been defined. The *Digraph* class has three constructors, allowing the directed graph to be initialized as an empty digraph, as a deep-copy of another directed graph or as a graph created from an input stream which can be either a file or a URL. The used constructor when creating the test data is the one creating an empty directed graph, as the edges are later added taking into consideration the type of graph which is needed, specified in the *DigraphGenerator* component. However, the simulator makes use of this component to read data from the files given as input, and so the constructor which creates a directed graph from the information received as a file input shall be used. In order to make sure each graph is generated correctly, meaning no mathematical or DAG rules are broken, this component also throws errors if the given parameters are not correct. Such errors refer to the number of vertices which cannot be negative or the format of the input received by the constructor. Regarding the time-complexity of the operations, each of them take constant time. One exception is made by the functions which imply iterating over adjacent vertices of a vertex. This operations time complexity is directly proportional with the number of the adjacencies.

- *DigraphGenerator* This class is the main class of the Graph Generator, which sets the important parameters for each file generation: the number of files generated, the type of directed graph contained by each of the files, the number of vertices and edges of each of the DAGs. An inner class of this component is the *Edge* class, which holds the logic behind each of the directed graphs edges. This class implements the *Comparable* interface. This feature of the class is used when generating the graphs to guarantee an ascending order of the two vertex ids. There have been defined several methods for generating a DAG, all of them returning a *Digraph* component with a specific pattern. The first method generates a directed graph with a fixed number of vertices and edges. An empty *Digraph* is the starting point of the DAG and a new edge is added each time the id of the first node generated using the uniform probability distribution is different from the second id generated using the same method and the edge is not already contained in the graph. The second method returns a type of graph which is a random generated directed graph usually called the Erdos–Renyi model. This directed graph has a fixed number of vertices and a number of edges generated with a given probability. The method uses the simple constructor of the *Digraph* component where an empty DAG is generated, then adds edges to the graph using the Bernoulli distribution probability. Another method generates a complete DAG with a fixed number of vertices and uses the first method where the number of edges given is equal to $v(v - 1)$. There is also a method for generating a tournament graph. Given a number of vertices, this method creates a directed graph where there is a directed edge between every two edges. Two important methods are the ones for generating a rooted-in or a rooted-out directed graph with a fixed number of vertices and edges. Both of the methods take two integers as parameters, representing the two values. The first type of graph contains vertices which have the property that there is only one vertex reachable from any other node. The second type of graph contains nodes which have the property that each of them is reachable from only one other vertex. Another two similar methods are the ones for generating a rooted-in tree and a

rooted-out tree with a given number of nodes. These type of graphs are most suited for simulations on a Bayesian network, as they respect a causal model structure. A method for creating a complete binary tree with a fixed number of vertices also exists, this method being suitable for creating graphs which have the structure of a yes/no decision tree. These type of graphs may help scheduling input data received from a software which is meant to give an answer regarding whether a certain medication is appropriate for a patient.

- *StdRandom* This Java class provides different methods for generating random numbers. This component is used by the DigraphGenerator to guarantee the randomness of the graphs edges generation. The numbers may be generated from several continuous and discrete probabilities: Gaussian, exponential, Bernoulli, and Poisson. The StdRandom class has two private attributes: the first one of type Random, for pseudo-random number generation and the second one is a long type attribute and represents a number used to initialize the first attribute.

DAGBroker is the distribution of tasks to processing nodes in the simulation. The next node to receive a task is chosen in a Round Robin manner, to guarantee that no node is overloaded while others remain unused. When a job is created, the length of the job is given with a uniform probability distribution to simulate tasks which have different time-complexity. This distribution can be easily changed with Weibull distribution to see how the processing time changes.

4 A Case Study

In order to verify the benefits of our proposal, a simulative approach has been chosen. The simulation has been based on a tool based on the HEPAR system, from which the scheduling workload have been abstracted and extracted.

4.1 The Simulated System

The HEPAR system is a medical expert system used to support diagnostic decisions based on data retrieved from physical examinations, medical interviews and non-invasive medical procedures. One advantage of this system is that it has been proved not to be sensitive to incompleteness of data [25]. This system contains a database of patient records of the Gastroenterological Clinic of the Institute of Food and Feeding in Warsaw, which is continuously enlarged with new cases [28]. With help of the HEPAR system, medical diagnosis and decision making are modeled using if-then rules. The system analyzes the given data and produces a subset of possible diagnoses, out of more than 70 types of disease categories. Regarding liver transplantation, the HEPAR system combines the risk scores with weights which can be varied dynamically in order to infer prognosis from previously given rules [18,25].

Knowledge in HEPAR is represented as a rule (e.g. Fig. 1), which has a measure of confidence in its correctness associated in order to express the uncertainty in the domain. This approach, however, does not give an exact meaning to these non-probabilistic measures of uncertainty [18]. On this system, a tool for liver disorders

```

if
  lessthan(biochemistry,AP,60) and
  greaterthan(biochemistry,gamma-GT,100) and
  between(biochemistry,ASAT,[50,200]) and
  between(biochemistry,ALAT,[50,200]) and
  lessequal(biochemistry,total_bili,17)
then
  conclude(patient,type-disorder,biliary_obstructive)
  with CF = 0.25
  conclude(patient,type-disorder,hepatocellular)
  with CF = 0.60
fi

```

Fig. 1 Example of a production rule for HEPAR

diagnosis [28] has been built, based on a Bayesian network model, managing causal relations among the various variables that are relevant in the HEPAR database for a diagnosis.

The application is executed on the MapR Converged Data Platform [<https://mapr.com/products/mapr-converged-data-platform/>], a software platform that provides real-time database management capabilities, global event streaming and scalable enterprise storage, integrating Hadoop and Spark. Database services are based on a NoSQL system, used to add real-time, operational analytics assets. MapR is composed of three main entities (see Fig. 2): the MapR Platform Services, responsible for storage, database functionalities and streaming; the Commercial Engines; and the Open Source Engines, that allow the integration of other ecosystem such as Hive, Apache, Mahout and others. In the Healthcare and Life Sciences domains, this software platform can provide access to important data, such as output from medical devices, lab results, imaging reports, medical correspondence and clinical data. This type of unstructured data represents almost 80% of information in the healthcare industry, and is in a continuous growth. The MapR Converged Data Platform is considered an efficient means of storing information about a big number of patients, which can be used to analyze the effects of new therapies, or even adjust drug doses, to minimize side effects and improve effectiveness, because of its support to Big Data and real-time data collection and processing, and has been chosen as support for genome processing and DNA sequencing, assisted diagnosis and personalized treatment planning applications.

4.2 The Simulation Testbed

The case study has been used to generate a number of DAG structured input data, representing the schedules of the system to be executed onto the simulated processing elements. These elements that may be seen as heterogeneous processors, which have the role of processing the tasks in the real system, according to the causal model of the various tasks. The simulations are parametrized, by means of the input DAGs, in

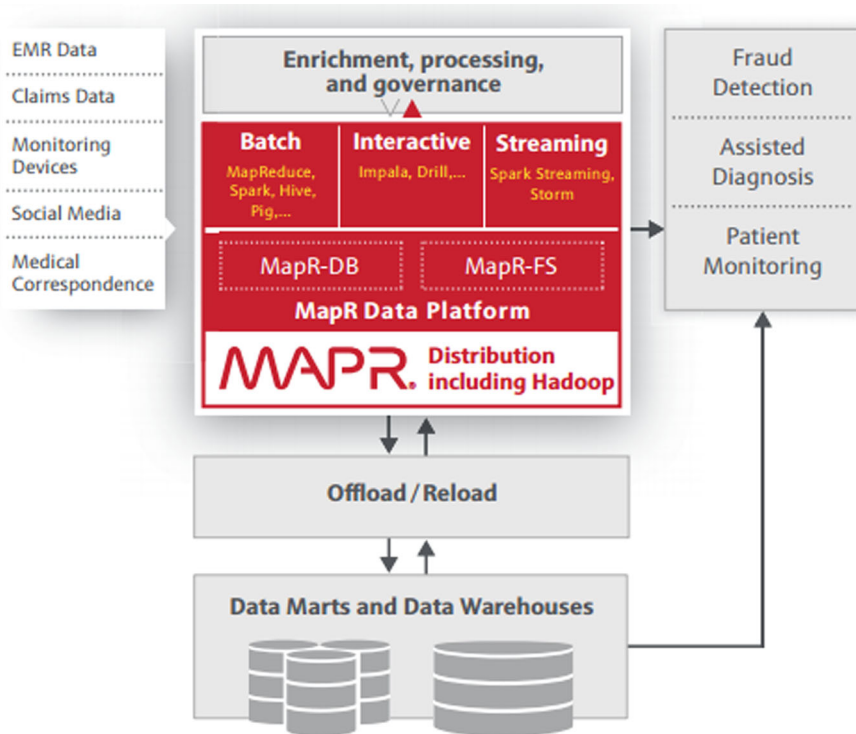


Fig. 2 MapR system architecture

the number of nodes in the graph, the number of edges, the processing time needed for each independent task and, obviously, the structure of the graph.

The simulation testbed is based on *MTS*² (Many Task Scheduling Simulator) [36], a task scheduling event-based simulator written in Java, which can be used to experimentally check or validate various classes of scheduling algorithms. It simulates events that happen inside of a cluster and offers a visual result after each simulation. This simulator solves both scalability problems: scalability in the number of simulated entities and scalability in the number of simulation steps. The first issue is solved by modeling entities as finite state machines with a set of accepted inputs, possible states and a state-transition function. A processing element cannot modify its state if it has not received any new input, that is the reason why the simulator run-time is linked to the number of events taking place during the simulation, instead of the clock ticks in the simulated environment. The second issue has been solved by introducing a PDES (Parallel Discrete Event Simulator) version of the simulator, to process events happening at the same time or at close intervals in parallel. Because the communication is done by message-passing, there are no issues concerning locking or synchronizing shared memory areas [37]. *MTS*² also offers a means for simulating fault tolerant algorithms and can anticipate the collapse of a processor by allowing new processing components to be added during the simulation. The architecture of the simulator is inspired by the

Scala actors model, whose backend API exposes the basic objects which can be easily extended in order to best fit new needs.

For both schedulers, a method for generating a reversed topological list of nodes has been implemented. This method is used for both ALAP and b-level computation. The implementation of the schedulers is based on the map-reduce paradigm, in which mapping is based on the use of the ALAP values as keys for the key-value pairs for MCP, and the reduction part combines all partial hashing together in both cases.

The implementation of the modified MCP algorithm is sketched in the following:

- **initialize** a hash map to collect the results;
- **repeat until** all DAGs have been processed:
 - **compute** the list of ALAPs and the list of corresponding node ID;
 - **if** counter equals 0
 - **for each** ALAP-node ID pair
 - **add** key-value pairs to the hash map
 - **else**
 - **for each** ALAP value in the ALAP list:
 - if the ALAP value exists in result keys:
 - **add** the corresponding node ID to the list of nodes having the same ALAP value;
 - **else**
 - **create** a new list and add the node and
 - **add** a new entry to the result hash map with the ALAP value as key and the new list as a value;
- return results;

The implementation of the modified HLFET algorithm is sketched in the following:

- **initialize** an array for task lists;
- **compute** the list of task lists for the current DAG;
- **copy** the list of task lists to the array for task lists;
- **repeat until** all DAGs have been processed:
 - **compute** the task list for the current DAG;
 - **for each** index of the array for task lists:
 - **merge** the corresponding task lists;
 - **if** length of current task list is greater than the array for task lists:
 - **add** the extra task list to the the array for task lists;
- return the array for task lists;

5 Evaluation

To analyze the implemented scheduling model, various simulations had to be conducted to see how the scheduling and processing time of the test data varied when some important parameters received different values.

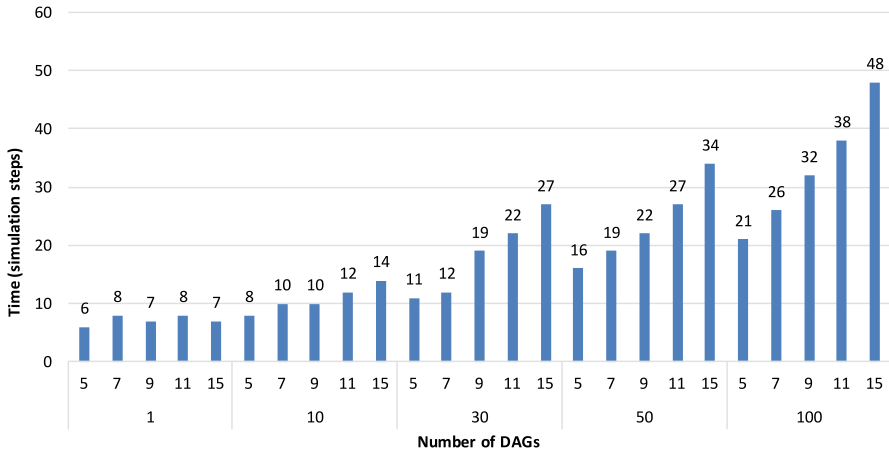


Fig. 3 Scheduling time by number of files and vertices

5.1 Testing the Modified MCP Scheduling Algorithm

The first set of tests was designed to see how the scheduling time of the test data varied when the number of DAG structures was changed starting from 10 graphs and ending with 100 graphs. The simulations were repeated using DAGs with different number of vertices, ranging from 5 to 15. For each of the graph structure sizes (with 5, 7, 9, 11 and 15 nodes) the scheduling of a single DAG structure has been simulated with an increasing number of input data files, in order to observe the relation between number of input data files and scheduling time.¹

The first conclusion which can be drawn from the graphic is that the average scheduling time of a single DAG structure is 6.7 ms. We can see that our algorithm which solves the problem of scheduling multiple DAG structures simultaneously, offers a much better performance than we would have had trying to schedule the DAGs one by one. Because the DAGs used for the simulations had a relatively small number of nodes, scheduling and processing only one DAG at a time on a number of 4 or more processors would have led to a waste of time, since the dependencies imposed by the DAG structures would not have allowed at some steps of the simulation, more than a single job processing. While waiting for some nodes to be processed in order to have the desired input for the current task, a processing element would have become a waste of resource. As visible in Fig. 3, the scheduling time increases with the number of DAG structures, but the increase is not directly proportional to the increase of the input data, but rather has a linear increase. Excluding the results of the simulations which implied testing the scheduling of a single DAG structure, the other results clearly show how the time taken by the scheduling algorithm is influenced by the number of vertices in each of the DAG structures. It was also important to see the relation between the time

¹ As the preprocessing stage produced only one hash in the case of these simulations, the algorithm has been modified by suppressing the step in which the hashes containing partial results were merged, without loss of generality with respect to the purposes of the simulation.

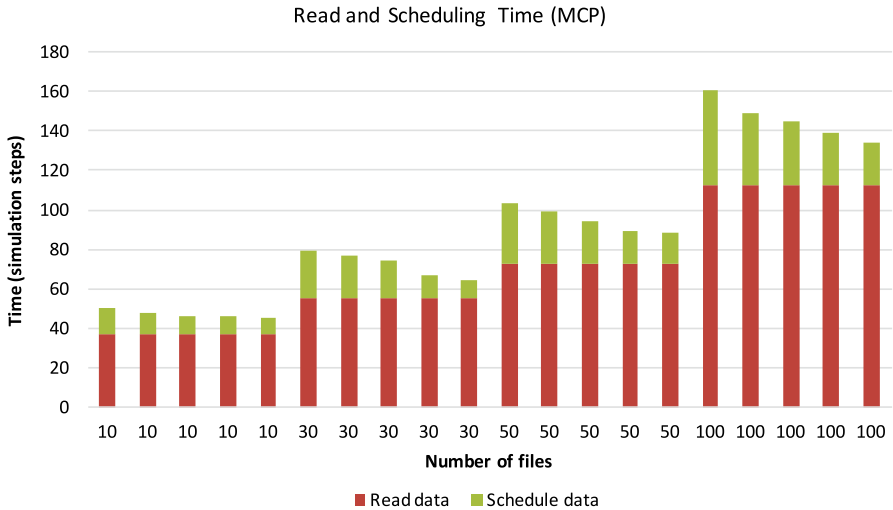


Fig. 4 Read and schedule time for modified MCP algorithm by number of files and vertices

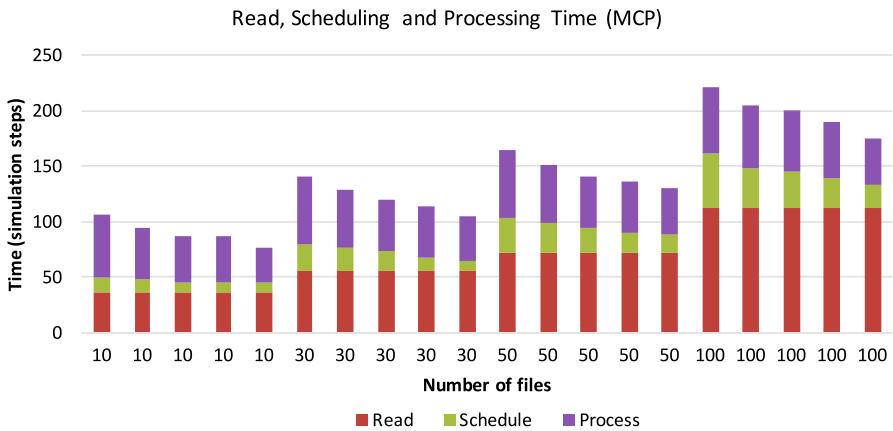


Fig. 5 Read, scheduling and processing time for four processors by number of files and vertices

needed to read all the data from the input files and the time need to schedule all the DAG structures received as input. The reason for this was to show that the scheduling algorithms are not so time consuming compared to the reading of the data. It is easy to deduce that when using a system like the MapR platform which has great performance regarding access to unstructured data, the time used to read the data may be reduced and so the overall time taken by the two operations drops.

In Fig. 4 for each number of files used in the simulation the variations of the number of vertices are shown. Here, for each of the 4 sections (10, 30, 50 and 100), we can see the results for 15, 11, 9, 7 and 5 vertices, from left to right. In Fig. 5 we can see the same data, with the added information of simulation steps taken by each simulation. The processing time of the DAG structures is not measured in milliseconds like all

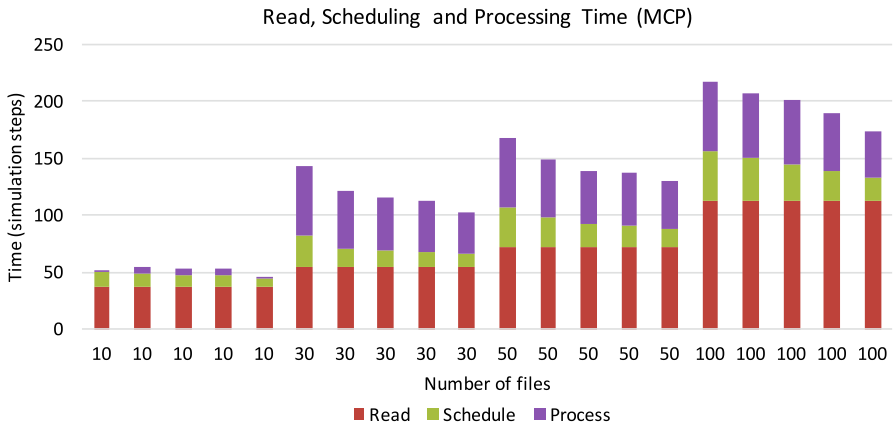


Fig. 6 Read, scheduling and processing time for ten processors by number of files and vertices

the others, but in messages exchanged by the nodes, as explained in Chapter 2. The simulations for Fig. 5 were made using a simulated cluster of four processing elements and a Broker.

Similar simulations were made for a number of ten processing elements, to see how the processing time changed in this case (Fig. 6). The biggest decrease in the processing time was observed when running simulations for a number of 10 DAG structures. This is due to the step of the algorithm in which task lists are distributed to processors using an insertion approach. At this step, the increased number of processing elements guarantees that lists of tasks which have a length equal or smaller than 10 are processed at the same time. However, when the number of files increases, the lists of tasks increase in length and their processing does not depend anymore on the scheduling algorithm, but on the length of the lists and processing time of each job.

5.2 Testing the Modified HLFET Scheduling Algorithm

Simulations were also conducted to analyze the performance of the second implemented algorithm. As in the case of the modified MCP algorithm, a parametric analysis has been performed. As we can see in Fig. 6, a variable number of DAG structures has been considered, increasing from 10 to 100. For each of these simulations, the number of vertices of each causal model was also varied with values ranging from 5 to 15. Fig. 7 shows how the scheduling time increases with the number of files received as input, each file containing data about a single DAG structure. By a comparison with Fig. 3, that is related to the first algorithm, the scheduling time appears to be lower. This is an expected result, as the original HLFET algorithm had a better time-complexity than the original MCP algorithm. An important difference between the results shown in Figs. 3 and 7 is that we can no longer see a linear increase of the scheduling time with the number of vertices of each DAG structure, for each set of tests ranging from 10 to 100 files. Also, the results obtained when simulating an input of 30 DAGs is easily comparable to the results seen when simulating an input of 50 DAGs.

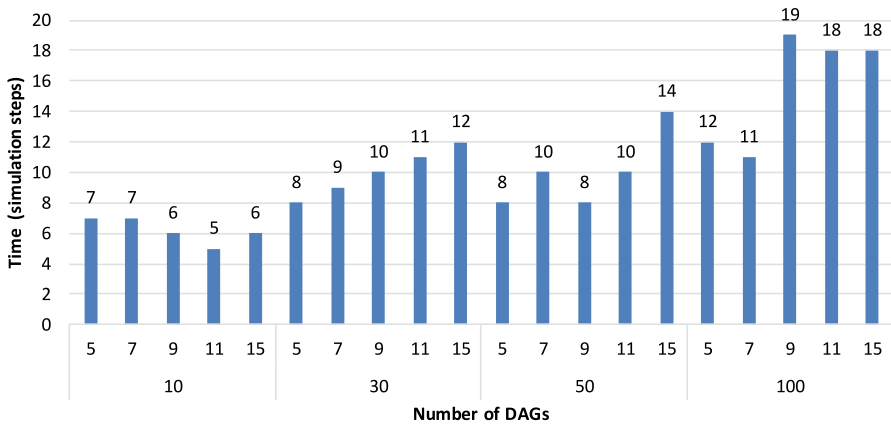


Fig. 7 Scheduling time by number of files and vertices

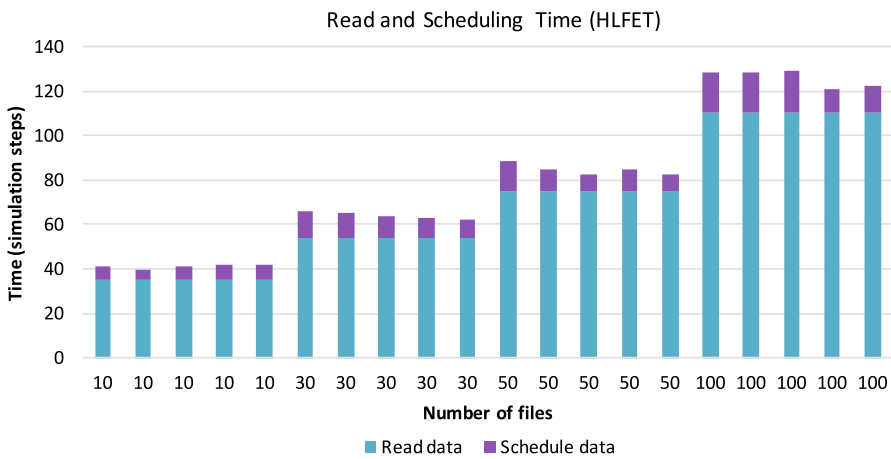


Fig. 8 Read and schedule time for modified HLFET algorithm by number of files and vertices

As in the case of the first set of simulations, the modified HLFET algorithm was tested on a single DAG structure also to see how the scheduling time differed from the first analyzed algorithm. An average of 5.54 ms was obtained, which is lower than the value for the modified MCP algorithm, as expected.

Simulations have proved a good scalability with the increase of DAG structures. The maximum scheduling time for 100 input files was 19 ms and the scheduling time of 500 input files with different DAG sizes (5, 7, 9, 11 and 15 vertices) was 50 ms, a good value in relation with the average scheduling time of a single graph.

During the simulations, the time taken by the process of reading the data from the files was measured. Figure 8 shows how the scheduling of data takes significantly less time than reading data from the input files. The processing time of the graphs was also measured, but, as in the case of the first scheduling algorithm, the number of messages exchanged by the nodes as been used as measurement unit, as the simulator

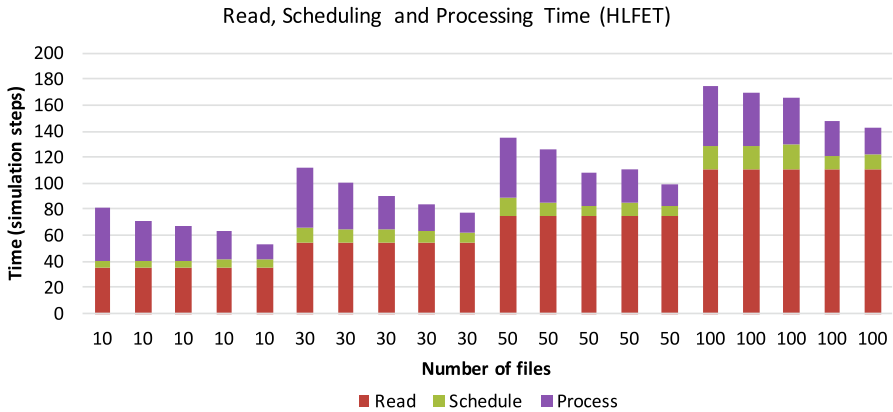


Fig. 9 Read, scheduling and processing time for four processors by number of files and vertices

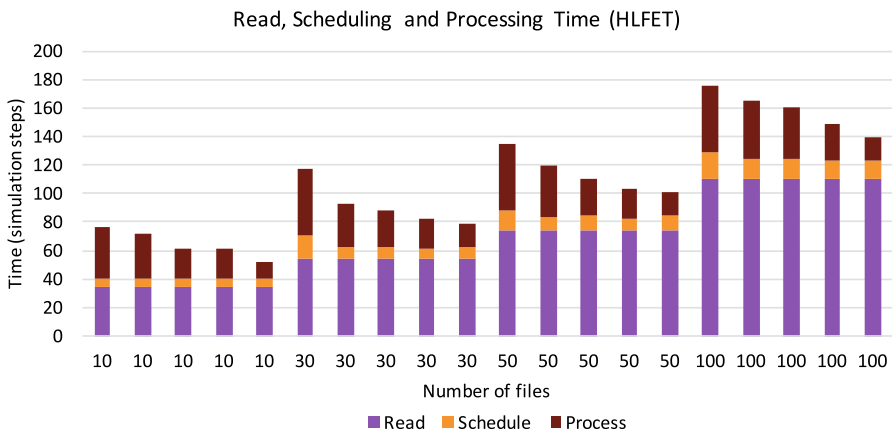


Fig. 10 Read, scheduling and processing time for ten processors by number of files and vertices

run time is linked to the number of events happening inside the simulated cluster. This is visible in Fig. 9, that also shows the relation between the time taken by the read, the scheduling and the processing of the information. The simulations for the results showed in Fig. 9 were run on a ten processors simulated cluster.

Figure 10, however, shows the simulation results for a four processors cluster. It is interesting to see how the differences between the overall time taken by the reading, scheduling and processing are not so visible, meaning our algorithm can manage to schedule tasks in such manner that a reduced amount of resources can offer a similar performance. This last observation may be the result of using a non-insertion approach when distributing the tasks to the processors. Even if the number of processing elements increases, the task lists, which have a smaller size than the number of processors and are distributed exactly as they arrive, cause an increased number of messages, that translates into a higher processing time.

5.3 Comparing the Two Algorithms

When analyzing the performance of both algorithms, we start by simulating the scheduling of a single DAG structure, to see if the initial time-complexity order is maintained. Regarding the modified MCP algorithm, tests have shown that the average scheduling time of a single DAG structure is 6.7 ms, which is higher than 5.54 ms, the time obtained for the modified HLFET algorithm. Moreover, let us consider the scheduling time taken for 500 input data files which contain DAG structures of different sizes, ranging from 5 to 15 nodes each. For the first scheduling algorithm, the average scheduling time was 81.5 ms, while, for the second algorithm, the result obtained was 50 ms. This is the second evidence that the HLFET algorithm proved to be more efficient when processing graphs with a relatively small number of nodes. Last, but not least, the simulations for the modified MCP algorithm indicated that the highest scheduling time was obtained for scheduling 100 DAG structures with 15 vertices each, and the maximum value was 48 ms. For the second algorithm, the maximum value obtained was 19 ms for scheduling 100 graphs with 9 vertices each.

6 Conclusions

We presented two adapted scheduling algorithms for applications represented as a causal model, with relevance in the medical industry, having as starting point the MCP and HLFET scheduling algorithms and the *MTS*² simulator as support for our analysis and algorithm design. Our contribution also includes an extension of the *MTS*² simulator with a scheduling layer for tasks with dependencies.

The simulations have shown that our scheduling algorithms provide a performance improvement when simultaneously dealing with a large number of DAGs. The actual scheduling of the tasks takes a lower time than the reading and processing of tasks. We have also shown in our simulations that the scheduling model is scalable and the time improvement between processing a single DAG and a 500 DAGs workload is significant. For the modified MCP algorithm we see an increase of approximately 12 times for 500 DAGs, as for the modified HLFET algorithm we observed an increase of approximately 9 times regarding the scheduling time. Another important observation was that the time-complexity of the modified MCP algorithm was higher than the modified HLFET algorithms, maintaining the ranking of the initial algorithms. As further work, we will implement the analyzed algorithms in a real medical environment and integrate the scheduling layer with a diagnosis tool to help hospitals with low resource benefit from the diagnosis support of a performant tool without the costs implied by software platforms like the MapR Converged Platform.

Acknowledgements This article is based upon work from COST Action IC1406 High-Performance Modelling and Simulation for Big Data Applications (cHiPSet), supported by COST (European Cooperation in Science and Technology). Work has been partially supported by project DataWay - Real-time Data Processing Platform for Smart Cities: Making sense of Big Data (Platform de Procesare n Timp Real pentru Big Data n Orae Intelligent), with a grant of the Romanian National Authority for Scientific Research and Innovation, CNCS UEFISCDI, Project Number PN-II-RU-TE-2014-4-2731.

References

1. Achim, O.M., Pop, F., Cristea, V.: Reputation based selection for services in cloud environments. In: 2011 14th International Conference on Network-Based Information Systems (NBIS), pp. 268–273. IEEE (2011)
2. Adam, T.L., Chandy, K.M., Dickson, J.R.: A comparison of list schedules for parallel processing systems. *Commun. ACM* **17**(12), 685–690 (1974). doi:[10.1145/361604.361619](https://doi.org/10.1145/361604.361619)
3. Ahmad, I., Kwok, Y.K.: On exploiting task duplication in parallel program scheduling. *IEEE Trans. Parallel Distrib. Syst.* **9**(9), 872–892 (1998). doi:[10.1109/71.722221](https://doi.org/10.1109/71.722221)
4. Baxter, J., Patel, J.H.: The last algorithm: a heuristic-based static task allocation algorithm. In: ICPP (1989)
5. Bessis, N., Sotiriadis, S., Cristea, V., Pop, F.: Modelling requirements for enabling meta-scheduling in inter-clouds and inter-enterprises. In: 2011 Third International Conference on Intelligent Networking and Collaborative Systems (INCoS), pp. 149–156. IEEE (2011)
6. Chen, H., Shirazi, B., Marquis, J.: Performance evaluation of a novel scheduling method: linear clustering with task duplication. In: Proceedings of the 2nd International Conference on Parallel and Distributed Systems, pp. 270–275 (1993)
7. Chung, Y.C., Ranka, S.: Applications and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed memory multiprocessors. In: Proceedings Supercomputing '92, pp. 512–521 (1992). doi:[10.1109/SUPERC.1992.236653](https://doi.org/10.1109/SUPERC.1992.236653)
8. Coffman, E.G., Graham, R.L.: Optimal scheduling for two-processor systems. *Acta Inform.* **1**(3), 200–213 (1972). doi:[10.1007/BF00288685](https://doi.org/10.1007/BF00288685)
9. Colin, J.Y., Chrtienne, P.: C.p.m. scheduling with small communication delays and task duplication. *Oper. Res.* **39**(4), 680–684 (1991). doi:[10.1287/opre.39.4.680](https://doi.org/10.1287/opre.39.4.680)
10. El-Rewini, H., Lewis, T.: Scheduling parallel program tasks onto arbitrary target machines. *J. Parallel Distrib. Comput.* **9**(2), 138–153 (1990). doi:[10.1016/0743-7315\(90\)90042-N](https://doi.org/10.1016/0743-7315(90)90042-N). <http://www.sciencedirect.com/science/article/pii/074373159090042N>
11. Farina, R., Cuomo, S., De Michele, P., Piccialli, F.: A smart GPU implementation of an elliptic kernel for an ocean global circulation model. *Appl. Math. Sci.* **7**(61–64), 3007–3021 (2013)
12. Farina, R., Cuomo, S., Michele, P.D.: A CUBLASCUDA implementation of PCG method of an ocean circulation model. In: AIP Conference Proceedings, vol. 1389, no. 1, pp. 1923–1926 (2011)
13. Forti, A., Bavikadi, S., Bigongiari, C., Cabras, G., De Angelis, A., De Lotto, B., Frailis, M., Hardt, M., Kornmayer, H., Kunze, M., et al.: Grid services for the magic experiment. In: Sidharth, B., Honsell, F., De Angelis, A. (eds.) *Frontiers of Fundamental Physics*. Springer, Dordrecht (2006)
14. Hagra, T., Janecek, J.: A high performance, low complexity algorithm for compile-time task scheduling in heterogeneous systems. In: Proceedings of 18th International Parallel and Distributed Processing Symposium, 2004, pp. 107– (2004). doi:[10.1109/IPDPS.2004.1303056](https://doi.org/10.1109/IPDPS.2004.1303056)
15. Hu, T.C.: Parallel sequencing and assembly line problems. *Oper. Res.* **9**(6), 841–848 (1961). doi:[10.1287/opre.9.6.841](https://doi.org/10.1287/opre.9.6.841)
16. Hwang, J.J., Chow, Y.C., Anger, F.D., Lee, C.Y.: Scheduling precedence graphs in systems with inter-processor communication times. *SIAM J. Comput.* **18**(2), 244–257 (1989). doi:[10.1137/0218016](https://doi.org/10.1137/0218016)
17. Kim, S., Browne, J.: General approach to mapping of parallel computations upon multiprocessor architectures. In: Proceedings of International Conference on Parallel Processing vol. 3, pp. 1–8 (1988)
18. Korver, M., Lucas, P.J.F.: Converting a rule-based expert system into a belief network. *Med. Inform.* **18**, 219–241 (1993)
19. Kruatrachue, B., Lewis, T.: Duplication Scheduling Heuristics (DSH): A New Precedence Task Scheduler for Parallel Processor Systems. Oregon State University, Corvallis (1987)
20. Kruatrachue, B., Lewis, T.: Grain size determination for parallel processing. *IEEE Softw.* **5**(1), 23–32 (1988). doi:[10.1109/52.1991](https://doi.org/10.1109/52.1991)
21. Kwok, Y.K., Ahmad, I.: Bubble scheduling: a quasi dynamic algorithm for static allocation of tasks to parallel architectures. In: Proceedings of Seventh IEEE Symposium on Parallel and Distributed Processing, pp. 36–43 (1995). doi:[10.1109/SPDP.1995.530662](https://doi.org/10.1109/SPDP.1995.530662)
22. Kwok, Y.K., Ahmad, I.: Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors. *IEEE Trans. Parallel Distrib. Syst.* **7**(5), 506–521 (1996). doi:[10.1109/71.503776](https://doi.org/10.1109/71.503776)
23. Kwok, Y.K., Ahmad, I.: Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.* **31**(4), 406–471 (1999). doi:[10.1145/344588.344618](https://doi.org/10.1145/344588.344618)

24. Liou, J.C., Palis, M.A.: An efficient task clustering heuristic for scheduling dags on multiprocessors. In: Proceedings of the Workshop on Scheduling and Resource Management for Parallel and Distributed Processing, pp. 152–156 (1996)
25. Lucas, P.J.F., Segaar, R.W., Janssens, A.R.: Hepar: an expert system for the diagnosis of disorders of the liver and biliary tract. *Liver* **9**(5), 266–275 (1989). doi:[10.1111/j.1600-0676.1989.tb00410.x](https://doi.org/10.1111/j.1600-0676.1989.tb00410.x)
26. Maheswaran, M., Siegel, H.J.: A dynamic matching and scheduling algorithm for heterogeneous computing systems. In: Proceedings of the Seventh Heterogeneous Computing Workshop, HCW '98, pp. 57. IEEE Computer Society, Washington, DC, USA (1998). <http://dl.acm.org/citation.cfm?id=795689.797882>
27. Mehdiratta, N., Ghose, K.: A bottom-up approach to task scheduling on distributed memory multiprocessors. In: 1994 International Conference on Parallel Processing, vol. 2, pp. 151–154 (1994). doi:[10.1109/ICPP.1994.14](https://doi.org/10.1109/ICPP.1994.14)
28. Onisko, A., Druzdzal, M.J., Wasyluk, H.: A probabilistic causal model for diagnosis of liver disorders. In: In Proceedings of the Seventh International Symposium on Intelligent Information Systems (IIS-98), pp. 379–387 (1998)
29. Papadimitriou, C.H., Yannakakis, M.: Scheduling interval-ordered tasks. *SIAM J. Comput.* **8**(3), 405–409 (1979). doi:[10.1137/0208031](https://doi.org/10.1137/0208031)
30. Papadimitriou, C.H., Yannakakis, M.: Towards an architecture-independent analysis of parallel algorithms. *SIAM J. Comput.* **19**(2), 322–328 (1990). doi:[10.1137/0219021](https://doi.org/10.1137/0219021)
31. Radulescu, A., van Gemund, A.J.C.: FLB: Fast load balancing for distributed-memory machines. In: Proceedings of the 1999 International Conference on Parallel Processing, pp. 534–541 (1999). doi:[10.1109/ICPP.1999.797442](https://doi.org/10.1109/ICPP.1999.797442)
32. Sarkar, V.: Partitioning and Scheduling Parallel Programs for Multiprocessors. MIT Press, Cambridge (1989)
33. Sfrant, A., Pop, F.: Asymptotic scheduling for many task computing in big data platforms. *Inf. Sci.* **319**, 71–91 (2015)
34. Sih, G.C., Lee, E.A.: A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Trans. Parallel Distrib. Syst.* **4**(2), 175–187 (1993). doi:[10.1109/71.207593](https://doi.org/10.1109/71.207593)
35. Simion, B., Leordeanu, C., Pop, F., Cristea, V.: A hybrid algorithm for scheduling workflow applications in grid environments (icdpd). In: Proceedings of the 2007 OTM Confederated International Conference on On the Move to Meaningful Internet Systems: CoopIS, DOA, ODBASE, GADA, and IS—Volume Part II, OTM'07, pp. 1331–1348. Springer, Berlin (2007). <http://dl.acm.org/citation.cfm?id=1784707.1784728>
36. Stratulat, A., Oncioiu, R., Pop, F., Dobre, C.: MTS2: Many task scheduling simulator. In: ECMS, pp. 586–593 (2015)
37. Stratulat, A., Oncioiu, R., Pop, F., Dobre, C.: MTS2: Many task scheduling simulator. In: Mladenov, V.M., Spasov, G., Georgieva, P., Petrova, G. (eds.) ECMS, pp. 586–593. European Council for Modeling and Simulation (2015)
38. Vasile, M.A., Pop, F., Tutueanu, R.I., Cristea, V., Kołodziej, J.: Resource-aware hybrid scheduling algorithm in heterogeneous distributed computing. *Future Gener. Comput. Syst.* **51**, 61–71 (2015)
39. Wu, M.Y., Gajski, D.D.: Hypertool: a programming aid for message-passing systems. *IEEE Trans. Parallel Distrib. Syst.* **1**(3), 330–343 (1990). doi:[10.1109/71.80160](https://doi.org/10.1109/71.80160)
40. Yang, T., Gerasoulis, A.: DSC: scheduling parallel tasks on an unbounded number of processors. *IEEE Trans. Parallel Distrib. Syst.* **5**(9), 951–967 (1994). doi:[10.1109/71.308533](https://doi.org/10.1109/71.308533)