Computers & Operations Research 40 (2013) 2670–2676

Contents lists available at SciVerse ScienceDirect

Computers & Operations Research

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

An efficient implementation of the Min-Min heuristic

Pablo Ezzatti *, Martín Pedemonte, Álvaro Martín

Instituto de Computación, Universidad de la República, 11300—Montevideo, Uruguay

ARTICLE INFO

Available online 30 May 2013

Keywords:
Heterogeneous computing
Grid computing
Scheduling
Min—Min heuristic

ABSTRACT

Min—Min is a popular heuristic for scheduling tasks to heterogeneous computational resources, which has been applied either directly or as part of more sophisticated heuristics. However, for large scenarios such as grid computing platforms, the time complexity of a straightforward implementation of Min—Min, which is quadratic in the number of tasks, may be prohibitive. This has motivated the development of high performance computing (HPC) implementations, and the use of simpler heuristics for the sake of acceptable execution times. We propose a simple algorithm that implements Min—Min requiring only $O(mn)$ operations for scheduling n tasks on m machines. Our experiments show, in practice, that a straightforward sequential implementation of this algorithm significantly outperforms other state of the art implementations of Min—Min, even compared to HPC implementations. In addition, the proposed algorithm is at least as suitable for parallelization as a direct implementation of Min—Min.

1. Introduction

A *distributed heterogeneous computing* (*HC*) environment, such as a grid computing platform, can employ thousands or even millions of computational resources to solve several difficult problems, involving the simultaneous execution of a large number of tasks. In this context, an efficient use of the resources is a critical issue that calls for schedules with short makespan, defined as the time required to complete the execution of all the tasks according to the schedule [1]. It is well known, however, that computing a minimum makespan schedule, for a given estimate of the execution time of each task on each machine, is an NP-hard problem [2,3]. This has motivated a big research effort focused on heuristic methods, trying to find accurate solutions with moderate runtime (see, e.g., the comparison of heuristics in [1,4,5]).

One of the most popular techniques for scheduling tasks in HC environments is the deterministic heuristic *Min—Min* [6, Algorithm D], which, in general, yields good schedules in acceptable execution times [5]. For this reason, Min—Min has also been used as a building block in more sophisticated heuristics [7—9], and applied to produce an initial solution that is then successively improved, typically using some local search method [1,9].

Min—Min is a greedy algorithm; having constructed a partial schedule for a subset of the given set of tasks, the schedule is extended in such a way that the makespan increment is minimum. A direct implementation of Min—Min requires $O(mn^2)$ operations [6] to schedule n tasks on m machines. (In [6], m is regarded as a constant.) Although it is faster than other more complex heuristics (e.g. cellular memetic algorithms [10], ant colony optimization [11], and tabu search algorithm [12]), a time complexity that is quadratic in n may be prohibitive in real modern HC environments, where several thousands of tasks need to be scheduled. As a consequence, some authors have proposed simplifications of Min—Min [13]. Others have explored *high performance computing* (*HPC*) implementations, including the use of parallel techniques, and non-traditional hardware, such as *Graphics Processing Units* (*GPUs*) [14,15].

In this paper we propose an efficient implementation of Min—Min. The algorithm works in two phases: first, for each machine, the tasks are sorted in nondecreasing order of execution time estimate, and then, in a second phase, the Min—Min schedule is constructed. Using a *radix exchange sort* [16] for the first phase, and exploiting in the second phase the fact that, for each machine, the tasks have been sorted, we show that the algorithm only requires $O(mn)$ operations on registers of length $O(\log mn)$. Since, for each machine, the algorithm stores the permutation that sorts the set of tasks, some additional memory is required with respect to a direct implementation of Min—Min. The latter requires $O(mn)$ space, while the space complexity of the new algorithm is $O(mn \log n)$.

Our experiments demonstrate, in practice, that this new algorithm significantly outperforms the state of the art implementations of Min—Min, even compared to HPC implementations.

Summing up, the main contributions of this paper are:

- a new algorithm that efficiently implements the Min—Min heuristic,
- a theoretical analysis of the algorithm, and

* Corresponding author. Tel.: +598 27114244x125.
E-mail addresses: pezzatti@fing.edu.uy (P. Ezzatti), mpedemon@fing.edu.uy (M. Pedemonte), almartin@fing.edu.uy (Á. Martín).

0305-0548/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.cor.2013.05.014

- an experimental evaluation of the algorithm.

The rest of the paper is structured as follows. In Section 2, we review in more detail the related works. Later, in Section 3, we recall the formal definition of the Min—Min heuristic and establish the notation for the rest of the paper. In Section 4, we introduce our proposal, an efficient algorithm to compute the Min—Min heuristic, and analyze it theoretically. We also discuss, briefly, some practical considerations related to the choice of the sorting algorithm, and the potential parallelization of our implementation of Min—Min. In Section 5 we present some experimental results that validate our proposal and, finally, we discuss some conclusions and future work in Section 6.

2. Related works

Ibarra and Kim presented one of the pioneering works on static heterogeneous computing scheduling [6], where five different heuristics were evaluated, including Min—Min. Additionally, the authors also studied other strategies for two particular cases: when the tasks have to be scheduled on only two machines, and when the machines are identical.

Since the pioneer work of Ibarra and Kim, many researchers have shown the benefits of Min—Min heuristic for heterogenous computing scheduling, since it makes possible to obtain good quality solutions in an acceptable runtime. Some of these works are summarized below. Braun et al. [1] studied experimentally 11 heuristics for static scheduling in HC environments, including a wide range of simple greedy constructive heuristic approaches and Min—Min. Then, Xhafa et al. [4] have also evaluated several static scheduling strategies, including Min—Min. In the same line of work, Luo et al. [5] analyzed and compared a set of 20 greedy heuristics under different conditions.

In another line of work, researchers have proposed several extensions to Min—Min or new algorithms with several points of contact with this heuristic. Segmented Min—Min is an algorithm closely related to Min—Min proposed by Wu et al. [7]. In this algorithm, tasks are sorted according to some score function of the expected time to compute in all machines (it could be the maximum, minimum or average expected time to compute among all machines). Then, the ordered sequence is segmented in groups, and finally Min—Min is applied to each of these groups of tasks.

Another common idea used by researchers is to apply a local search method in order to improve the quality of the solution obtained with Min—Min. Ritchie et al. [11] proposed a local search method that selects the best task movement from a neighborhood of solutions, which includes the solutions where a single task from the most loaded machine is moved to another machine or it is swapped with another task that executes on another machine. On the other hand, Pinel et al. [9] proposed the H2LL local search operator that chooses randomly a task from the most loaded machine and moves it to one of the least loaded machines.

Other interesting extensions are briefly described next. He et al. [17] proposed a QoS (Quality of Service) guided Min—Min heuristic which can guarantee QoS requirements of certain tasks while minimizing the makespan at the same time. He and Sun [18] integrated the time associated to data movement into traditional scheduling on grid environments using the Min—Min heuristic. Finally, Chauhan and Joshi [8] combined Min—Min with the weighted mean time-Min heuristic.

Despite the good results of Min—Min, when considering large scenarios (both tasks and machines), the complexity of the algorithm makes impracticable its direct application. This is particularly important when working in grid environments because they consist of thousands of machines, and the number of tasks to be scheduled is of the order of several thousands. For this reason, two different approaches have been proposed to reduce the execution time of this heuristic: to modify the original heuristic in order to run faster or to implement the algorithm in parallel.

Diaz et al. [13] proposed a two-phase heuristic for the energy-efficient scheduling of independent tasks on computational grids that can be considered as a simplification of Min—Min. In the first phase, the tasks are sorted by a certain criteria (average, minimum, maximum expected time to compute). In the second phase, the tasks are processed in order, searching for the best machine assignation. This heuristic is in fact a smart extension of the B heuristic described by Ibarra and Kim [6].

The parallel implementation of Min—Min on GPU platforms [15] was studied considering three different approaches: a single GPU version, and synchronous and asynchronous versions that execute on four GPUs concurrently. Pinel et al. [14] also addressed the parallelization of Min—Min for solving large scenarios proposing both a multicore and a GPU implementation.

From all the reviewed works it is clear that Min—Min heuristic has a wide applicability and it is widely used by the community to solve scheduling problems. On the other hand, the resolution of large scenarios, specially in grid environment contexts, poses an important challenge to this heuristic due to its computational complexity. For these reasons, a significant improvement in the computational efficiency of the algorithm could be welcomed and adopted by the community.

3. The Min—Min scheduling heuristic

In this section we formally define the scheduling problem and the Min—Min heuristic. We follow, loosely, the definitions and notation from [6]. Consider a set M of m machines and a set T of n tasks. We assume that an *expected time to compute* (*ETC*) of a task J on a machine i, denoted $\mu_i(J)$, is available for all $J \in T$ and all $i \in M$. A *schedule* $S = \{L_i\}_{i \in M}$, assigns a set L_i of tasks to be executed on machine i, for each $i \in M$. The *makespan* of S, denoted $\hat{f}(S)$, is the largest amount of execution time assigned by S to a machine i, among all $i \in M$, i.e.

$$\hat{f}(S) = \max_{i \in M} \left\{ \sum_{J \in L_i} \mu_i(J) \right\}. \tag{1}$$

Ideally, we look for a schedule with minimum makespan. As mentioned, however, this is computationally unaffordable for problem instances of large size, which motivated the definition of several heuristics in [6], including Min—Min.

Algorithm 1. Min—Min heuristic.

input: A set T of tasks, a set M of machines, and the ETC $\mu_i(J)$ for all $J \in T, i \in M$
output: A schedule $S = \{L_i\}_{i \in M}$
1 **foreach** $i \in M$ **do**
2 | Set $L_i = \emptyset$, $t_i = 0$
3 **end**
4 Set $U = T$
5 **while** $U \neq \emptyset$ **do**
6 Let $(i,J) = \arg\min_{(i,J)} \{t_i + \mu_i(J) : i \in M, J \in U\}$. //Solve ties arbitrarily
7 Set $L_i = L_i \cup \{J\}$, $t_i = t_i + \mu_i(J)$, $U = U \setminus \{J\}$.
8 **end**

An algorithmic definition of Min—Min is shown in Algorithm 1. The algorithm iterates through a loop in Step 5,