Expert Systems with Applications 40 (2013) 2799–2806

Contents lists available at SciVerse ScienceDirect

Expert Systems with Applications

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

The solution algorithms for the multiprocessor scheduling with workspan criterion

Radosław Rudek [a,*], Agnieszka Rudek [b], Andrzej Kozik [b]

[a] Wrocław University of Economics, Komandorska 118/120, 53-345 Wrocław, Poland
[b] Wrocław University of Technology, Wyb. Wyspiańskiego 27, 50-370 Wrocław, Poland

ARTICLE INFO

Keywords:
Multiprocessors
Scheduling
Learning
Deteriorating
Makespan
Workspan

ABSTRACT

In this paper, we consider multiprocessor scheduling problems, where each job (task) must be executed simultaneously by the specified number of processors, but the indices of the processors allotted to each job do not have to be contiguous (i.e., jobs can be fragmentable). Unlike other research in this domain, we analyse the problem under the *workspan* criterion, which is defined as the product of the maximum job completion time (makespan) and the number of used processors. Moreover, the job processing times can be described by non-increasing or non-decreasing functions dependent on the start times of jobs that model improvement (learning) or degradation (deteriorating), respectively. To solve the problems, we construct some polynomial time algorithms and analyse numerically their efficiency.

1. Introduction

The development of Massive Parallel Processor (MPP) systems implies that scheduling of multiprocessor jobs (tasks) has attracted significant interest in scheduling community (Drozdowski, 1996, 2009; Fan et al., 2012; Turek et al., 1996). In such systems jobs (tasks) can be performed parallel (in the same time) by more than one processor. However, multiprocessor jobs can model not only applications executed on MPP systems, but also many industrial processes, in which a single job (product) has to be processed in the same time by multiple machines or human workers (Caramia and Giordani, 2010). Namely, multiprocessor job scheduling problems occur for instance during production planning and project scheduling (Vizing, 1981), in chemical plants (Błażewicz et al., 1984) or during allocation of vessels to a berth with multiple quay cranes (Guan et al., 2002).

Usually, in such problems, the objective is to find a schedule (that includes start times of jobs and allotment of processors) that minimizes the maximum job completion time (makespan). Nevertheless, such time criteria can be insufficient for some practical cases, where the cost is significant, which is also related with the number of involved processor to complete the jobs (e.g., the financial cost of renting servers or hiring a crew, or the amount of used energy). Therefore, each processor (e.g., server, human worker) can be characterized by a rate, which is the cost (e.g., financial or energy) of using it per time unit. Thus, the total cost of processing jobs is also determined by the number of used processors. In other words, this cost criterion is the product of the maximum job completion time (makespan) and the number of used processors, which we will call the *workspan*.

On the other hand, in many real-life systems the efficiency of processors can change due to learning or deteriorating of processors (see Cheng et al., 2004; Gawiejnowicz, 2008; Janiak and Rudek, 2011; Kuo and Yang, 2007; Lai and Lee, 2010; Lee et al., 2009; Toksari and Güner, 2010; Wang et al., 2009; Yang and Kuo, 2007). Despite the existence of learning and deterioration effects is indisputable, they have never been considered in the context of scheduling multiprocessor jobs.

Therefore, in this paper, we will analyse multiprocessor job scheduling problems with the *workspan* criterion and processing times of jobs that can be constant or described by non-increasing (learning) or non-decreasing (deteriorating) functions dependent on job starting times. To the best of our knowledge, the results presented in this paper have never been investigated in the scheduling domain.

This paper is organized as follows. The next section contains model and problem formulation. Subsequently, description of approximation algorithms is provided, followed by an experimental verification of their efficiency. The last section concludes the paper.

2. Problem formulation

There are given a set of $J = \{1,\ldots,n\}$ of n jobs and m parallel identical processors $P = \{P_1,\ldots,P_m\}$. It is assumed that there are no precedence constraints between the jobs, they are non-preemptive and available for processing at time 0. Each processor can process (execute) one job at a time, however, each job $j \in J$ to

* Corresponding author. Tel.: +48 71 368 0378; fax: +48 71 368 0376.
E-mail addresses: radoslaw.rudek@ue.wroc.pl (R. Rudek), agnieszka.wielgus@pwr.wroc.pl (A. Rudek), andrzej.kozik@pwr.wroc.pl (A. Kozik).

0957-4174/$ - see front matter © 2012 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.eswa.2012.11.005

be completed requires continuously in the same time the given number of processors δ_j. The indices of the processors allotted to each job do not have to be contiguous, thus, they are called *fragmentable* jobs (tasks). It models many real-life settings that occur in computer and manufacturing systems (e.g., Drozdowski, 2009; Turek et al., 1996). Each job is also characterized by the processing time p_j. Therefore, to complete job j, it is required to allot δ_j processors that are available continuously by time p_j.

Nevertheless, in many real-life systems the efficiency of processors can change with time due to their improvement or degradation, which are called *learning* and *deteriorating*, respectively (see Gawiejnowicz, 2008; Lai & Lee, 2010; Lee et al., 2009; Wang, 2009; Yang & Kuo, 2011). Furthermore, even if the efficiency of processors is constant, the jobs can deteriorate and require more time if they are processed later (see Cheng et al., 2004). Deteriorating and learning can be modeled by job processing times described by non-decreasing (deteriorating) and non-increasing (learning) functions dependent on start times of jobs. Therefore, to model more precisely multiprocessor problems by taking into consideration the above observations, in this paper, the processing time of each job j is described by the following function dependent on its start time S_j:

$$\tilde{p}_j(S_j) = p_j(1+S_j)^{\alpha_j}, \tag{1}$$

where p_j is the normal processing time that is defined as the time required to process (execute) a job if there is no learning nor deteriorating of the processors (i.e., $p_j \triangleq \tilde{p}_j(0)$) and α_j is the learning/deteriorating factor that describes the impact of these phenomena on the processing time of job j. Note that $\alpha_j < 0$ and $\alpha_j > 0$ represent learning and deteriorating factors, respectively, whereas $\alpha_j = 0$ models cases when there is no changing in the efficiency of processors (i.e., in job processing times) or it is negligible. For convenience we will also use $\tilde{p}_j$ instead of $\tilde{p}_j(S_j)$.

Since the jobs are fragmentable, then for the given number of processors m the feasible schedule S can be defined by the set $S = \{S_1,\ldots,S_j,\ldots,S_n\}$, where S_j denotes the start time of job j, and the jobs are assigned to processors according to the non-decreasing order of S_j. On this basis, the completion time of job j can be calculated $C_j = S_j + \tilde{p}_j$ and the maximum completion time (makespan) for fixed m and S is equal to $C_{\max}(S) = \max_{j\in J}\{C_j\}$.

Usually, the objective is to find such a schedule S that minimizes the makespan $C_{\max}(S)$. Nevertheless, such time criteria can be insufficient for some practical cases, where the cost related with the number of involved processor to complete the jobs (e.g., the financial cost of renting servers or hiring a crew, or the amount of used energy) is significant. Therefore, each processor (e.g., server, human worker) can be characterized by a rate, which is the cost (e.g., financial or energy) of using it per time unit. Thus, the total cost of processing jobs is also determined by the number of used processors.

Let us define the discussed problem formally. The objective is to determine the number of processors m and the feasible schedule of jobs S on these processors such that the following cost criterion value $W_{\max}$ is minimized:

$$W_{\max}(m,S) = m \cdot C_{\max}(S). \tag{2}$$

Note that $W_{\max}$ depends on the makespan $C_{\max}$ and on the other hand the product of a job processing time and the number of processors used to execute this job is often called the *workload* (i.e., $\delta_j\tilde{p}_j$), thus, we will call the defined criterion $W_{\max}$ as the *workspan*.

For convenience and to keep an elegant description of the considered problems we will use the three field notation scheme $X|Y|Z$ (see Graham et al., 1979), where X describes the machine/processor environment, Y describes job characteristics and constraints and Z represents the minimization objectives. According to this notation, the problems analysed in this paper will be denoted as follows: $P|size_j|W_{\max}$, $P|size_j$, $det|W_{\max}$, $P|size_j$, $le|W_{\max}$, where $\alpha_j = 0$, $\alpha_j > 0$ and $\alpha_j < 0$, respectively, and $size_j$ is used to denote that each job has the fixed number of processors required to execute (process) this job.

3. Algorithms

Note that the problem for fixed m is NP-hard even for constant job processing times ($\alpha_j = 0$ for $j = 1,\ldots,n$) (see Du & Leung, 1989). Thus, the analysed problem is not less complex. Thereby, it is highly unlikely to find its optimal solution in polynomial time. It is hard, what to do. Nevertheless, we propose some heuristics.

The main idea of the presented algorithms is to construct a schedule S by determining the sequence of jobs $\pi = \langle\pi(1),\ldots,\pi(i),\ldots,\pi(n)\rangle$, where $\pi(i)$ denotes the index of a job in position i in π. Namely, for the given number of available processors m, jobs are assigned to processors in the given sequence π and for each of them the minimum feasible start time is calculated. Thereby, for each $j = \pi(i)$ (for $i = 1,\ldots,n$), the start time S_j has to be found such that at least δ_j processors are idle during time $[S_j, S_j + \tilde{p}_j]$. In the further part of the paper, an idle time of a processor will be called a *gap*, and determining the earliest feasible S_j is based on finding gaps on δ_j processors that cover range $[S_j, S_j + \tilde{p}_j]$. Note that we operate on rational numbers, thus, we are not limited to integer numbers only. An example allotment of processors to jobs is shown in Fig. 1.

To describe formally the method of determining S_j for each job in a sequence π, let us define the following structures (`GapList` and `PointerList`) and corresponding functions (in alphabetical order for each structure):

- $\texttt{GapList}_i$ – doubly linked list structure that stores idle times (called *gaps*) for processor P_i; each element in the list (i.e., gap) is characterized by its start time S_g, its end time C_g and the index of corresponding processor P_g (for this case $P_g = P_i$); the list is ordered according to the non-decreasing order of S_g;
- $\texttt{GapList} = \{\texttt{GapList}_1,\ldots,\texttt{GapList}_m\}$ – the set of all lists storing idle times (gaps) for processors;
  - $\texttt{AddGap}(P_i, S_g, C_g)$ – add at the end of $\texttt{GapList}_i$ of processor P_i a new gap with the start time S_g and end time C_g;
  - $\texttt{ClearGapList}(P_i)$ – clear the list of gaps corresponding to processor P_i;
  - $\texttt{DeleteGap}(g)$ – delete gap g from the list corresponding to processor P_g;
  - $\texttt{GapEndTime}(g)$ – return the end time C_g of gap g;
  - $\texttt{GapProcessor}(g)$ – return the index of the processor P_g corresponding to gap g;
  - $\texttt{GapStartTime}(g)$ – return the start time S_g of gap g;
  - $\texttt{Head}(\texttt{GapList}, P_i)$ – return the first gap on the list of gaps corresponding to processor P_i;
  - $\texttt{InsertGapAfter}(g, S_g, C_g)$ – insert a new gap with the given start time S_g and end time C_g after gap g; index of the corresponding processor P_g is the same as for g;
  - $\texttt{NextGap}(g)$ – return a gap from `GapList` that is a successor of g; the corresponding processor is unambiguously defined by gap g;
  - $\texttt{SetGapEndTime}(g, S_g)$ – set the end time C_g of gap g;
  - $\texttt{SetGapStartTime}(g, S_g)$ – set the start time S_g of gap g;
- `PointerList` – doubly linked list of m elements; each of them stores pointer p to a gap from `GapList` such that each p corresponds to a gap on a different processor;
  - $\texttt{AddPointer}(g)$ – add at the end of `PointerList` a new element that points to gap g;
  - $\texttt{AssignGapToPointer}(p, g)$ – set that element p from `PointerList` points to gap g;