CMST 6(1) 25-40 (2000) DOI:10.12921/cmst.2000.06.01.25-40

COMPUTATIONAL METHODS IN SCIENCE AND TECHNOLOGY 6, 25-40 (2000)

SUBOPTIMAL APPROACHES TO SCHEDULING MALLEABLE TASKS

J. BŁAŻEWICZ[1], M. MACHOWIAK[1], G. MOUNIÉ[2], D. TRYSTRAM[2]

[1] *Institute of Computing Science, Poznań University of Technology*
ul. Piotrowo 3a, 60-965 Poznań, Poland

[2] *ID-IMAG, antenne de l'ENS IMAG*
ZIRST de Montbonnot, 51 rue Jean Kuntzman
38330 Montbonnot Saint Martin, France

Abstract: In the paper, the problem of scheduling a set of n malleable tasks on m parallel computers is considered. The tasks may be executed by several processors simultaneously and the processing speed of a task is a function of the number of processors alloted. The problem is motivated by real-life applications of parallel computer systems in scientific computing of highly parallelizable tasks. Starting from the continuous version of the problem (i. e. where the tasks may require a fractional part of the resources), we propose a general approximation algorithm with a performance guarantee equal to 2. Then, some improvements are derived that lead to a very good average behavior of the scheduling algorithm.

1. INTRODUCTION

Scheduling malleable tasks has been proved to be a promising way for an efficient implementation of some practical problems on parallel and distributed computers [3,9, 13-16].

The idea of a malleable task (in short MT) results in solving the problem at a different level of granularity in order to globally take into account communication costs and parallelization overheads with a simple penalty factor. The dependence of a malleable task processing time on a number of processors alloted is given as a (non-linear) function.

Most of the applications that have already been parallelized using the MT model have been decomposed by some expert users using the knowledge they have of the physical problem that is reflected in the program, and the penalty due to the management of the parallelism inside the malleable task (communication and synchronization overhead) is usually a monotonie function: using more processors makes the execution time decreasing while the global overhead increases. These assumptions are valid for the applications composed of computational kernels where the penalty can easily be predicted or estimated [3,9],

The above property distinguishes malleable tasks from the multiprocessor tasks, considered for example in [4] and [12], where the number of processors alloted to each task is known. The latter model has received a considerable attention in the literature. The problem of scheduling independent MT without preemption (it means that each task is computed on a constant number of processors from its start to completion) is NP-hard [10], thus, an approximation algorithm with

performance guarantee has been looked for. While the problem has an approximation scheme for any fixed value *m,* no practical polynomial approximation better than 2 is known [13]. The 2-approximation presented in [13] is based on a clever reduction of MT scheduling to the 2 dimensional bin-packing problem, using the earliest result of [17] that any λ-approximation for the bin-packing problem can be polynomially transformed into a λ-approximation for the MT scheduling. It is worth stressing that the algorithm of Ludwig [13] on average behaves similarly to its worst case behavior.

In this paper starting from the continuous version of the problem (i. e. where the tasks may require a fractional part of the resources), we propose a different approximation algorithm with a performance guarantee equal to 2. Then, some improvements are derived that lead to a very good average behavior of the scheduling algorithm.

The organization of the paper is as follows. In Section 2 the motivation for the scheduling problem and its formulation are given. Section 3 contains the first approximation algorithm together with an analysis of its worst case behavior. In Section 4 some improvements of the algorithm are presented and its mean behavior experimentally evaluated. Section 5 concludes the paper.

2. MOTIVATION AND FORMULATION OF THE SCHEDULING PROBLEM

2. 1. Motivation for the malleable tasks model

We start our considerations with few real-life applications of highly parallelizable tasks justifying the model used.

Simulation of molecular dynamics

The simulation of molecular dynamics is one of the most challenging problem in science. The computation of atom movements is irregular if interactions are spatially limited (cut-off). An efficient execution requires advanced techniques allowing to overlap communications by computations like asynchronous buffered communications and multithreading. In the case of protein behavior, computations may require to calculate interactions between hundreds of thousands of atoms [8], Needless to say, such an execution needs a large memory. On some of the top parallel computers, like Cray T3E, in order to simplify hardware and optimize communications, there is no virtual memory management, thus the available memory is strongly limited. Hence, when the instance of the problem does not fit into the memory of a processor, the execution cannot be performed directly. To complete the execution, the virtual memory management needs to be done "by hand" using out of core computations, that is loading and storing intermediate computations on a disk. Of course, this increases the time of an execution. Thus, when the number of processors is sufficient for storing the whole data in the memory of these processors, a superlinear speed-up will be observed, otherwise, a processing speed function is concave (see Fig. 1).

Operational oceanography

Numerical modeling of the ocean circulation started in the sixties and was continuously developed since that time for climate study and operational oceanography, i. e. near real-time forecast of the "oceanic weather", in a way similar to operational meteorology.

A major practical problem to be dealt with in ocean general circulation models (OGCM), is their large computational cost, which is notably greater than the cost of corresponding atmospheric models, due to differences in the typical scales of motion. For example, the order of magnitude of the size of dynamic structures like fronts or eddies is a few tenths of kilometers in the ocean, while 5 or 10 times larger in the atmosphere. The horizontal resolution of OGCMs should allow the explicit representation of such structures, which leads to very important memory and CPU requirements (OGCMs typically use today a horizontal resolution of 1/6 ° to 1/10 ° - i. e. approximately 10 to 15 kilometers - and 20 to 50 vertical discretization levels, which represent more than several million grid points). Moreover the time scales in the ocean are one order of magnitude larger than in the atmosphere, which implies an integration of numerical models on longer time periods (from a few weeks to tenths of years, depending on the application), with a time step of few minutes (typically 1 to 30, depending on the horizontal resolution and the time integration scheme). The computations involved by these models are run on vector and/or parallel supercomputers and any simulation requires several hundred or thousand hours of CPU-time. The objective today is to use the low cost clusters of PC machines for solving these problems.

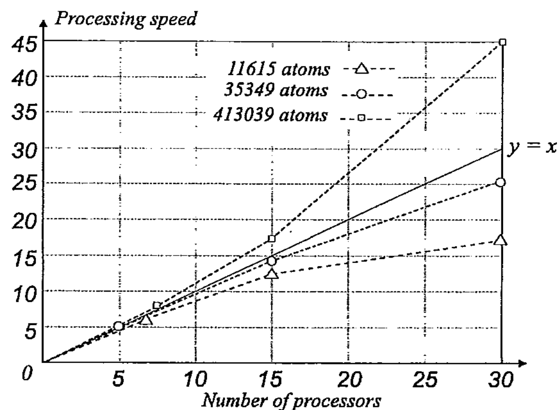

Fig. 1. Function: a processing speed *vs.* a number of processors

The parallelization of ocean models is performed by domain decomposition techniques. The geographical domain is divided into subdomains, each of them being allocated to a processor. Most of the existing works usually consider as many subdomains as processors. The computation of the explicit terms is mainly local; it requires only, at the beginning of each time step, some communications between processors corresponding to adjacent subdomains, to exchange model variables along the common interfaces. On the other hand, linear systems for the implicit tenns are not local, since they correspond to the discretized form of elliptic equations. Solving these global systems is performed for instance by a preconditioned conjuguate gradient or by domain decomposition techniques [7]. An important point for the purpose of this work is to emphasize

that ocean models are *regular* applications, in the sense that the volume of computations can be estimated quite precisely as a function of the grid size and the number of processors.

In the context of operational oceanography, adaptive meshing could be of great interest for ocean modelers. The basic principle of adaptive mesh refinement methods (AMR) consists in locally refining or coarsening the computation mesh, according to some mathematical or physical criteria (like error estimates or eddy activity). It could reduce the computational cost of models by taking advantage of the spatial heterogeneity of oceanic flows and thus using a fine mesh only where and when necessary. Such techniques are widely used with finite element codes, but rather rarely with finite differences because such refinements lead to non-homogeneous grids and thus complicate the handling of the code. However, Berger and Oliger [6] proposed an AMR algorithm which avoids this drawback, by considering a locally multigrid approach. In their method, the refinement is not performed on a unique non-homogeneous grid, but on a hierarchy of grids, i.e. a set of homogeneous embedded grids of increasing resolutions, and interacting among themselves (see the principle explained in Fig. 2). Without entering into details, the principle of the algorithm is as follows. Consider a hierarchy of grids, like the one depicted in Fig. 2: it consists in a root (or level-0) grid covering the entire domain of computations with coarse resolution Δh_0 and coarse time step Δt_0, and a number of subgrids (or level-1 grids) with a finer resolution $\Delta h_1 = \Delta h_0/r$ and a finer time step $\Delta t_1 = \Delta t_0/r$ focused only on some subdomains (r is an integer called the refinement ratio). This structure is recursive, in the sense that any level-l grid can contain finer level-(l + 1) subgrids, with $\Delta h_{l+1} = \Delta h_l/r$ and $\Delta t_{l+1} = \Delta t_l/r$ (of course, until a maximum level l_{max}).

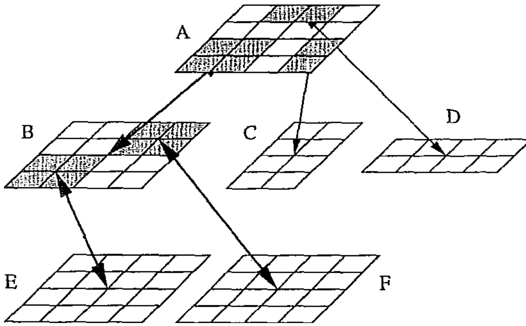

Fig. 2. Example of grid hierarchy

Time integration is performed recursively starting from the root grid. Any level-l grid is advanced forward one time step Δt_l. The solutions at time t and $t + \Delta t_l$ are used to provide initial and boundary conditions for the integration of the level-(l + 1) subgrids. These subgrids can then be advanced forward r time steps Δt_{l+1} to provide a more precise solution at time $t + r\,\Delta t_{l+1} = t + \Delta t_l$ on the regions covered by the subgrids. These solutions at level (l + 1) are then used to improve the solution at level l, *via* an update procedure.

The relevance of the grid hierarchy is checked regularly every N coarse time steps Δt_0. A criterion is evaluated at every grid point to determine whether the local accuracy of the solution seems sufficient or not. Subgrids can then be created, resized or removed.

Since the different grids at a given level l can be run simultaneously, they will be alloted to different groups of processors. The problem is then to determine, for a given hierarchy, which is the best grid-to-processor allocation (from the viewpoint of the computational efficiency). These assignments must be determined at every regridding step, because of the evolution of the grid hierarchy during the simulation.

The first simplified version of the approach discussed has been imlemented on a IBM-SP parallel system with 16 nodes [1], The ocean model used for this study was a simple quasi-geostrophic box model, since the intention was to validate the malleable task approach and to design a good scheduling heuristic before running the simulation on the operational models. This type of model has been widely used in the ocean modeling community, and is known as a simple prototype of eddy-active large scale circulation in the mid-latitudes. Blayo and Debreu [7] already implemented the AMR method in such a multi-layered model. They demonstrated that the use of this method results in a very significant gain in CPU time (by a factor of 3) while conserving, within a 10 to 20% range, the main statistical features of the solution obtained with an uniformly high resolution. Moreover, it appears that one only simulation with the AMR method leads to better local predictions than classical nested grid techniques on regions of particular interest, wherever the region of interest is located, and for a comparable amount of computation. A more sophisticated scheduling algorithm derived from the analysis we developed in the present paper should allow to solve larger and more complex instances.

Yet another example of the computations which can be modeled by malleable tasks are big matrix calculations used e. g. in Cholesky factorization. The speed of computations depends in this case on the fact whether or not a matrix fits into the cache memory [9].

The model of Malleable Tasks is an efficient tool for solving such problems. Typically, the total number of malleable tasks remains very small.

2. 2. Problem formulation

We consider a set of m identical processors $P = \{P_1, P_2, \ldots P_m\}$ used for executing the set $T = \{T_1, T_2, \ldots, T_n\}$ of n independent, nonpreemptable malleable tasks (MT). Each MT needs for its execution at least 1 processor but less than m. The number of processors alloted to a task is unknown in advance. The *processing speed* of a task depends on the number of processors alloted to it: namely, function f_i relates processing speed of task T_i, to a number of processors alloted. The criterion assumed is schedule length. Let us note that processing times of MT's are sometimes represented by some factor μ which determines the loss of time during a task processing using more than one processor, caused by communication delays or by synchronization needs. This factor, called *inefficiency factor,* is a discrete function of a number of processors and a type of a task and its geometrical interpretation is given in Fig. 3.

The relation between the speed function and the inefficiency factor is as follows:

$$\mu_i(r) = \frac{rt_i(r)}{t_i(1)} \rightarrow t_i(r) = \frac{t_i(1)}{r}\mu_i(r), \qquad \text{or} \qquad \frac{t_i(1)}{t_i(r)} = \frac{f_i(r)}{f_i(1)}. \tag{1}$$

Thus,

$$f_i(r) = \frac{rf_i(1)}{\mu_i(r)}$$

where: r - a number of processors used, r (0, m); $t_i(r)$ - processing time of task T_i on r processors; t_i-(1) - processing time of T_i on one processor; μ_i - inefficiency factor (a discrete function of r) for task T_i; f_i - processing speed function for T_i.

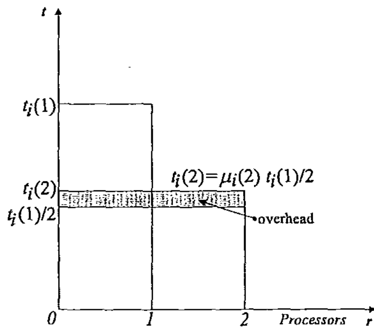

Fig. 3. Inefficiency factor

Now, the problem may be stated as the one of finding a schedule (a processor allocation to tasks) of minimum length ω, provided that the processing speed functions of the tasks are all concave. (Let us note, that this is a realistic case, more often appearing in practice. The case of only convex functions has been considered in [5]). We will denote this minimum value for m processors as ω^*_m.

As far as computer applications are concerned, functions f_i are discrete, i. e. they take values in discrete points only, which correspond to processor assignments to tasks. However, in general, it would be also possible that these functions are continuous (and they are concerned with continuous resources). Since in what follows we will use the results of optimal continuous resource allocation to construct good processor schedules, we will recall some basic results from the optimal continuous resource allocation theory [18], To distinguish it from a discrete case, an optimal schedule length of a continuous case will be denoted by C^*_{cont}.

Assume that a processing of task T_i is described by the following equation:

$$\dot{x}_i(t) = dx_i/dt = f_i(r_i(t)), \qquad x_i(0) = 0\ , \qquad x_i(C_i) = p_i \tag{2}$$

- $x_i(t)$ - the state of T_i at moment t,
- r_i(t) - a real number of resource units alloted to T_i at time t,
- f_i - a continuous, non-decreasing function, $f_i(0) = 0, f_i(r_i) > 0$,
- C_i - unknown in advance, finishing time of T_i,
- p_i - the final state or processing demand of T_i.

The total available resource amount is equal to m, i. e.

$$\sum_{i=1}^{n} r_i(t) \leq m \quad \text{for every } t$$

From (2) we have:

$$\int_0^{C_i} f(r_i)\,dt = p_i \tag{3}$$

and thus $p_i = C_i f_i(r_i)$ or $C_i = p_i / f_i(r_i)$

Denote by R the set of feasible resource allocations. Further, denote by U the set defined in the following way: $\mathbf{u} = (u_1, u_2, \ldots, u_n) \in U$ if $\mathbf{r} \in R$, where $u_i = f_i(r_i)$. Elements of U will be called *transformed resource allocations.*

Following [18] it may be proved that the minimum schedule length for a set of n independent tasks described by (2) can always be expressed by the formula:

$$C^*_{\max}(\mathbf{p}) = C^*_{\max}(\mathbf{p}, U) = \min\{C_{\max} > 0 : \mathbf{p}/C_{\max} \in conv\, U\} \tag{4}$$

where $\mathbf{p} = \{p_1, p_2, \ldots, p_n\}$ is the vector of processing demands of tasks, and *conv U* denotes the convex hull of U, i. e. the set of all convex combinations of elements of U.

Now, we describe geometrical interpretation of the problem (see Fig. 4). From the above Theorem we know that the minimum schedule length is determined by the intersection point of straight line $u_i = p_i/C_{\max}$, $i = 1, 2, \ldots, n$ and the boundary of set *conv U.* This means that the shape of boundary of *conv U* is of basic importance for the form of an optimal schedule. For the concave functions f_i one gets a convex set U equal to *conv U* (cf. Fig. 4). Using some algebraic calculations, a minimal length of a continuous schedule C^*_{cont}, can be calculated [18], In the following sections we will show how to use it to construct approximation algorithms for a discrete, computer scheduling problem.

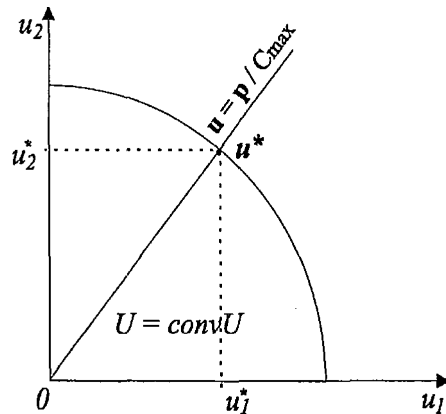

Fig. 4. Set U and the optimal solution of the continuous resource allocation problem in case of two tasks and concave functions

3. AN APPROXIMATION ALGORITHM AND ITS WORST CASE BEHAVIOR

In this section we will propose a way to transform a schedule obtained from the continuous version into a feasible schedule for the discrete MT model. We will prove that this transformation has a performance guarantee of 2 (as compared with C^*_{cont}).

3.1. Rounding scheme

To transform a continuous allotment greater than 1 for any task into a discrete one, it is sufficient to round off the quantities of processors used to the largest integer values smaller than the continuous ones. Then, the execution time of any task is not increased more than by a factor of 2. Moreover, the makespan of the discrete version is at most twice as long as the optimal continuous one.

When the number of processors assigned in the continuous solution is between 0 and 1, the speed function is super linear, and then, the continuous algorithm can produce schedules which are arbitrary far from the discrete optimal schedules. See for instance the following example: A set of $n > m$ independent tasks of unit length $t_i(1) = 1$, to be executed on m processors with the same speed function r^α, $0 < \alpha < 1$. Each task should be executed on m/n processors, leading to a schedule length equal to

$$C^*_{const} = \frac{1}{\left(\frac{m}{n}\right)^\alpha}.$$

In the MT model, each task will be executed on at least one (integer) processor. The total work cannot be less than the work of such an allotment. The n tasks will be scheduled in time at least $\omega_m^* \geq n/m$ (cf. Fig. 5).

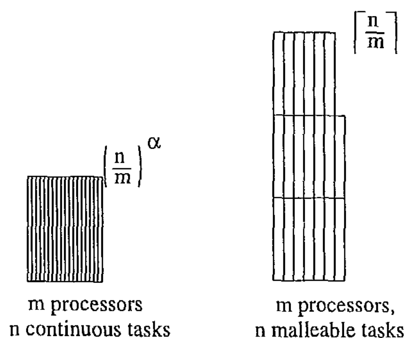

Fig. 5. Continuous and discrete schedules for independent malleable tasks

The ratio between the execution times of the continuous and the discrete versions is larger than $n/m\ (m/n)^\alpha$, namely

$$\left(\frac{n}{m}\right)^{1-\alpha}$$

The value of the ratio is arbitrary large as n increases. Thus, there exists no transformation from a continuous solution with a r^{α} speed function to a malleable task solution with a constant approximation ratio between continuous and discrete solutions.

Below, we present an algorithm using the information provided by the continuous solution to schedule the malleable tasks with a constant performance ratio to the optimal malleable scheduling.

3. 2. Transformation algorithm

Even if the continuous solution does not provide a time bound to the malleable task scheduling makespan, a transformation of the continuous solution with concave speed functions, provides a solution with constant ratio to the optimal malleable solution. In the malleable task model, the processor resource cannot be used at a rate smaller than 1. The minimum area of work used by a task is thus its execution time on one processor - $t_i(1)$.

The transformation algorithm is simple. Every task T_i with an allocation r_i- ≥ 1 in the continuous solution is scheduled on $\tilde{r}_i = \lfloor r_i \rfloor$ processors in the malleable scheduling. All parallel tasks (allotment strictly larger than 1) start at time 0. Other tasks, for which a continuous allotment $r_i < 1$, receive one processor and are scheduled in free time slots. The details are specified below in Algorithm 1.

Algorithm 1. Generic transformation algorithm

```
for i = 1 to n
    r̃_i = ⌊r_i⌋
if r_i < 1 then
    r̃_i = 1
for i = 1 to n
    if r̃_i > 1 then
        Start(T_i) = 0
for i = 1 to n
    if r̃_i = 1 then
        Start(i) = MinimumStart()
```

Note, that the complexity of the above algorithm is $O(n)$.

3. 3. Worst case analysis of Algorithm 1

Theorem 1 analyzes the worst case behavior of Algorithm 1.

Theorem 1. Algorithm 1 has a performance guarantee of 3.

Proof. The continuous solution consists in executing simultaneously all the tasks on a fractional number of processors so that they finish at the same time. This solution realizes the trade-off between the total work and the length of the tasks. Thus, the makespan of this solution is a lower bound on ω_m^* (a makespan of the optimal malleable solution).

For all the tasks which continuous allotment r_i 1, a malleable (discrete) allotment decreases with a ratio lower than 1/2. Using the concavity assumption, their duration does not increase more than twice. But their surface (*work*, defined as a product of a task duration and a number of processors allocated to it) decreases. Moreover, the sum of the processors alloted to these tasks after the transformation is lower than m.

The tasks which continuous allotment $r_i < 1$ are assigned one processor. Their execution times decrease but their work increases. This surface is the minimal one that these tasks can have in any malleable schedule.

The tasks which duration is between C^*_{cont} and $2C^*_{\text{cont}}$ can be executed on less than m processors starting at time 0. Each of the other tasks has a duration lower than C^*_{cont} and their total surface is less than $m\omega_m{}^*$.

The sum of the surfaces of the tasks is lower than $m\ C^*_{\text{cont}} + m\ \omega_m{}^*$. An analysis similar to Graham's one can be now applied [11], The last task that is alloted starts at a time when all the processors are busy (otherwise, it could have been started earlier). Thus, the schedule length of the malleable (discrete) schedule, ω_m is lower than

$$\max\left\{2C^*_{\text{cont}}, \frac{mC^*_{\text{cont}} + m\,\omega^*_m}{m} + C^*_{\text{cont}}\right\} = 2C^*_{\text{cont}} + \omega_m{}^* .$$

Since $C^*_{\text{cont}} \le \omega_m{}^*$, we obtain

$$\omega_m \le \omega^*_m + 2\,\omega^*_m .$$

This first bound can be improved by some refinement of the algorithm. The idea here is to start from the relation

$$\omega_m \le \omega^*_m + C^*_{\text{cont}}$$

and to refine it. If $\omega_m{}^* > 2C^*_{\text{cont}}$, we obtain directly a guarantee of 2.

The difficult point is when $\omega^* < 2\ C^*_{\text{cont}}$. For this case, we propose to modify the previous algorithm as follows: decrease the number of processors alloted to the tasks which initial allotment was greater than 1, until reaching an execution time of $2\ C^*_{\text{cont}}$. As $2\ C^*_{\text{cont}}$ is greater than $\omega_m{}^*$ the allotment chosen for these tasks is lower than the allotment in the optimal malleable schedule. It means that the work surface of these tasks is lower than $m\ \omega_m{}^*$. The sum of the surfaces of these tasks is then lower than the optimal surface. Thus, we obtain the following bound

$$\omega_m \le \frac{m\,\omega^*_m}{m} + C^*_{\text{cont}} .$$

After this modification, we again obtain:

$$\omega_m \le 2\,\omega^*_m .$$

We see that the refined version of Algorithm 1 has the worst case behavior bounded by 2. The cost of the rounding algorithm is linear. To schedule the tasks we need also a good mapping

algorithm which can be in this case Largest Processing Time First (LPTF) [2], It requires to sort the tasks and to maintain a sorted list ofthe processor load. Thus, a reasonnable implementation has a complexity of $O(n \log(n) \log(m))$.

4. AN IMPROVED ALGORITHM WITH BETTER AVERAGE BEHAVIOR

Algorithm 1 in the last section has the worst case bound equal to 2. It appears that on the average it will behave similarly, most often approaching this bound. For this reason we propose a slightly more sophisticated algorithm. Its main idea for refinement is to pack more cautiously small tasks (requiring one processor only) and to use several steps of rounding off. These changes do not lengthen the discrete (malleable) schedule in the worst case, while allowing for a good average behavior (as will be demonstrated in the computational experiment).

Algorithm 2

- calculate C^*_{cont} and the optimal continuous processor allocation r_i for all tasks.
- round the continuous allocation of processors to the integer values.

  if $r_i \le 1$ **then**

  $\tilde{r}_i := 1$

  else

  if $(r_i > 2)$ or $(r_i < 1.5)$ **then**

  $\tilde{r}_i := \lfloor r_i \rfloor$ {As in the general rounding scheme}

  else

  $\tilde{r}_i := 2$ {refinement when $1.5 \le r_i \le 2$}
- calculate the new processing times of the tasks.
- calculate a discrete number of processors used $\tilde{m} = \sum_{i=1}^{n} (\tilde{r}_i)$.
- find a task with the biggest completion time C_i and assign $C := \max\{C_i\}$

  if $\tilde{m} = m$ **then** *Go To End*

  if $\tilde{m} < m$ **then** allocate an excess of processors to the longest task - *Go To End*

  if $\tilde{m} > m$ **then** assign the remaining tasks after the tasks already scheduled till the $\max\{C, \sum_{i=1}^{n} t_i(1)\}$

  if $\tilde{m} = in$ **then** *Go To End*

  else take the schedule of minimum length constructed by either **A** or **B**

- A the remaining tasks schedule after the tasks already scheduled treating them as new instance.

- B **while** $\tilde{m} > in$ reduce a number of processors assigned to the task (group of tasks) with the biggest number of processors alloted and assign the tasks on the freed processor and on the other processors till the finishing time of the longest task.

END $\omega_m := \max\{C_i\}$.

Let us consider the following example of the application of Algorithm 2.

Example 1

Let us consider the following data: n = 10, m = 5 , set of tasks T = $\{T_1, T_2, \ldots, T_{10}\}$ with vector $t_i(1)$ = [1, 2, 3, 4, 5, 6, 7, 8, 9, 20] of processing times. Processing speed function $f(r) = r^{1/5}$, the same for each task. The solution of the continuous problem is C^*_{cont} = 14.60.

Continuous processor allocations r^*_{cont} = [$1.5*10^{-6}$, $4.8 * 10^{-5}$, $3.6 * 10^{-4}$, $1.5 * 10^{-3}$, $4.7* 10^{-2}$, 0.011, 0.025, 0.049, 0.088, 4.82], From this solution we proceed according to the rounding algorithm. The consecutive partial schedules are depicted in Fig. 6.

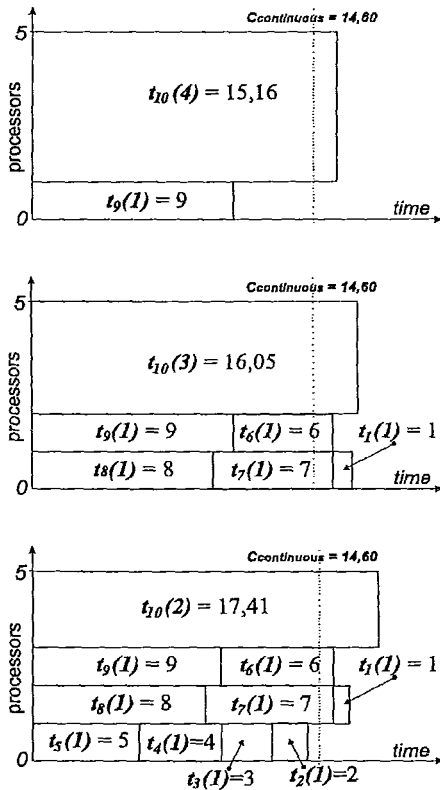

Fig. 6. Consecutive partial schedules generated according to Algorithm 2 for the set of tasks given in Example 1

As we mentioned, Algorithm 2 will not have worse behavior than Algorithm 1, however, its mean behavior seems to be better as demonstrated by the following set of experiments.

To evaluate the mean behavior of Algorithm 2 we use the following measure:

$$S_{\text{Alg2}} = \min\left\{\omega_m/C^*_{\text{cont}}, \omega_m/C_{\text{area}})\right\}$$

where

- ω_m- a schedule length obtained by Algorithm 2,
- C^*_{cont} - an optimal schedule length of the continuous solution,
- $C_{\text{area}} = \sum_{i=1}^{n} t_i(1)/m$ - a schedule length for the uniprocessor allocation for all the tasks.

Clearly, the maximum of the two values C^*_{cont} and C_{area} is the lower bound on the optimal schedule length ω_m^* for malleable tasks (discrete case), thus, S_{Alg2} indicates properly a behavior of Algorithm 2.

Table 1. Average behavior of Algorithm 2 for different numbers of processors and the shapes of speed function (n = 100)

Processors	a – different for each task			$a = 4$		
	ω_m/C^*_{cont}	ω_m/C_{area}	S_{alg2}	ω_m/C^*_{cont}	ω_m/C_{area}	S_{alg2}
4	7.96	1.00	1.00	8.48	1.01	1.01
8	5.16	1.02	1.02	5.22	1.02	1.02
16	3.35	1.28	1.28	3.12	1.18	1.18
32	3.26	1.54	1.54	3.11	1.29	1.29
64	1.93	1.48	1.48	2.98	1.32	1.32
128	1.60	1.41	1.41	1.87	1.36	1.36
256	1.21	2.12	1.21	1.30	2.03	1.30
512	1.12	3.79	1.12	1.10	3.09	1.10

Table 2. Average behavior of Algorithm 2 for varying number of tasks and the shape of speed function (m = 64)

Tasks	a – different for each task			$a = 4$		
	ω_m/C^*_{cont}	ω_m/C_{area}	S_{alg2}	ω_m/C^*_{cont}	ω_m/C_{area}	S_{alg2}
20	1.09	4.26	1.09	1.07	3.19	1.07
30	1.10	3.80	1.10	1.21	3.27	1.21
40	1.11	2.60	1.11	1.33	3.05	1.33
50	1.13	1.87	1.13	1.13	1.87	1.13
60	1.19	1.62	1.19	1.15	1.83	1.15
70	1.21	1.56	1.21	1.19	1.61	1.19
80	1.17	1.68	1.17	1.28	1.52	1.28
90	1.27	1.36	1.27	1.31	1.35	1.31
100	1.51	1.47	1.47	1.30	1.42	1.30
110	1.45	1.51	1.45	1.31	1.18	1.18
120	1.48	1.41	1.41	1.43	1.18	1.18
130	1.63	1.26	1.26	1.36	1.19	1.19
140	1.61	1.40	1.40	1.45	1.19	1.19
150	1.89	1.41	1.41	1.49	1.18	1.18
160	1.78	1.25	1.25	1.57	1.18	1.18
170	2.02	1.14	1.14	1.73	1.19	1.19
180	2.23	1.26	1.26	1.85	1.09	1.09
190	2.10	1.18	1.18	2.07	1.01	1.01
200	2.05	1.10	1.10	6.01	1.00	1.00

Table 3. An influence of the speed function on an average behavior of Algorithm 2 $(m = 32)$

Tasks	a – different for each task			$a = 7$		
	ω_m/C^*_{cont}	ω_m/C_{area}	S_{alg2}	ω_m/C^*_{cont}	ω_m/C_{area}	S_{alg2}
10	1.05	4.67	1.05	1.02	4.47	1.02
20	1.11	2.58	1.11	1.05	3.17	1.05
30	1.37	1.66	1.37	1.10	1.85	1.10
40	1.51	1.53	1.51	1.14	1.48	1.14
50	1.38	1.24	1.24	1.21	1.24	1.21
60	1.94	1.41	1.41	1.38	1.10	1.10
70	1.90	1.13	1.13	1.53	1.16	1.16
80	1.88	1.16	1.16	1.52	1.10	1.10
90	2.12	1.18	1.18	1.86	1.10	1.10
100	1.96	1.09	1.09	1.99	1.10	1.10

To test mean behavior of Algorithm 2 an extensive computational experiment has been conducted in the following way.

Task processing times $t_i(1)$ have been generated from a uniform distribution in interval [1..100].

Processing speed functions have been chosen as

$$f_i(r) = r^{1/a}, \quad a \geq 1 .$$

Values of parameter a have been generated from a uniform distribution in interval [1..10],

The results of the experiment are gathered in Tables 1 through 3. Each entry in these tables is a mean value for 10 instances randomly generated. Table 1 illustrates an influence of a number of processors on the average behevior of Algorithm 2. Table 2 shows an impact of a number of tasks, while Table 3 illustrates an influence of the speed function, respectively, on the performance of Algorithm 2.

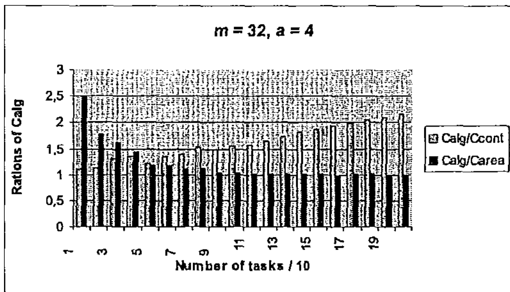

Fig. 7. An impact of a number of tasks on behavior of Algorithm 2

Figure 7 illustrates an impact on the behavior of Algorithm 2 by a number of tasks with varying speed functions.

From the experiments conducted we see that the mean behavior of the algorithm (as obtained in the above computational experiments) does not exceed value 1.54 of the assumed lower bound of the optimal schedule length for the discrete case.

The algorithm behaves well for a wide range of task and processor parameters. The experiments show also that when a number of tasks greatly exceeds a number of processors, the optimal continuous solution does not approximate well the discrete malleable one. In the latter, each task must receive at least one processor. On the other hand, for a number of tasks being close to a number of processors, the continuous solution may be a good starting point for a construction of an optimal malleable schedule. Since the first is constructed in polynomial time, the second (of a good quality) may be also constructed in a short time.

5. CONCLUSION

In the paper, the problem of scheduling malleable tasks has been considered. Starting from the continuous version of the problem (i. e. where the tasks may require a fractional part of the resources), we proposed a general approximation algorithm with a performance guarantee equal to 2. Then, some improvements were derived that led to a very good average behavior of the scheduling algorithm. Further investigations could take into account a construction of the algorithm with a better worst case performance guarantee, as well as, an analysis of some special (but practically important) cases, involving few parallel tasks in the system only, each requiring many processors at the same time.

References

[1] E. Blayo, L. Debreu, G. Mounié. D. Trystram, Engineering Simulation, **22,** 8 (2000).

[2] A. Barak, and O. La'adan, Journal of Future Generation Computer Systems, 1998.

[3] P. E. Bernard, *Parallelisation et multiprogrammation pour line application irreguliere de dynamique moleculaire operationnelle,* Mathematiques appliques, Institut National Polytechnique de Grenoble, 1997.

[4] J. Błażewicz, M. Drabowski, J. Węglarz, IEEE Transactions on Computers 35, 389 (1986).

[5] J. Błażewicz, M. Machowiak, G. Mounié, D. Trystram, J. Węglarz, Revista Iberoamericana de Computación, 2000, to appear.

[6] M. Berger and J. Oliger, J. Comp. Phys. **53**, 484 (1984).

[7] E. Blayo and L. Debreu, J. Phys. Oceanogr. **29,** 1239 (1998).

[8] P. E. Bernard, T. Gautier, D. Trystram, *Proceedings of Second Merged Symposium IPPS/SPDP 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing,* San Juan, Puerto Rico, 1999.

[9] J. Dongarra, L. Duff, D. Danny, G. Sorensen, H. van der Vorst, Society for Industrial & Applied Mathematics, 1999.

[10] J. Du, J.Y-T. Leung, SIAM Journal on Discrete Mathematics, **2**, 473 (1989).

[11] R. L. Graham, Bell System Tech. J. **45**, 1563 (1966).

[12] E. Lloyd, Journal of the ACM, **29,** 781 (1982).

[13] W. T. Ludwig, *Algorithms for scheduling malleable and nonmalleable parallel tasks,* PhD thesis, University of Wisconsin-Madison, Department of Computer Sciences, 1995.

[14] G. Mounié, C. Rapine, D. Trystram, *Efficient approximation algorithms for scheduling malleable tasks,* In: *Eleventh ACM Symposium on Parallel Algorithms and Architectures (SPAA'99),* ACM, 23 (1999).

[15] G. N. S. Prasanna and B. R. Musicus, Algorithmica (1995).

[16] U. Schwiegelshohn, W. Ludwig, J. Wolf, J. Turek, P. Yu, SIAM Journal on Computing **28,** 237 (1999).

[17] J. Turek, J. Wolf, P. Yu, *Approximate algorithms for scheduling parallelizable tasks*, In: *4th Annual A CM Symposium on Parallel Algorithms and Architectures,* 323 (1992).

[18] J. Węglarz, *Modelling and control of dynamic resource allocation project scheduling systems*, In: S. G. Tzafestas (ed.), *Optimization and Control of Dynamic Operational Research Models*, North-Holland, Amsterdam, 1982.