Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 231–238

ISBN 978-83-60810-22-4

Task Scheduling with Restricted Preemptions

Tomasz Barański
Email: tbaransk@poczta.onet.pl

***Abstract*—One of basic problems in task scheduling is finding the shortest schedule for a given set of tasks. In this paper we analyze a restricted version of the general preemptive scheduling problem, where tasks can only be divided into parts at least as long as a given parameter k. We introduce a heuristic scheduling method TSRP3. Number of processors m, number of tasks n and task lengths p_i are assumed to be known. If $n \geq 2m$ and k is sufficiently small, TSRP3 finds shortest possible schedule with $O(m)$ divisions in polynomial time. In addition we introduce a more robust algorithm TSRP4 based on combining TSRP3 with multi-fit.**

I. Introduction

UNRESTRICTED scheduling on parallel processors is solved in linear time using McNaughton's [7] algorithm. Unfortunately this approach often leads to processing of some tasks only for a very short time before or after preemption. Preemptions are usually costly in some way, so we introduced "granularity" threshold k, so that no part of any divided task can be shorter than k, which should be large enough to make any preemption times and costs negligible. Unless k is 0[1] this problem is in general NP-hard (see [4]). This paper is based on research thesis by Tomasz Barański [2].

II. The model

In this paper we consider deterministic scheduling problem of type[2]

$$P \quad | \quad k-pmtn \quad | \quad C_{MAX}$$

We schedule n tasks with known lengths p_i on m identical processors[3] numbered from 1 to m. Tasks can be divided, but no part of any task may be shorter than some given parameter k. Any processor can work on at most one task at any given time, and tasks cannot be executed in parallel. All tasks must be completed. The aim is to find a schedule of minimum makespan C_{MAX}. There is also a secondary goal of decreasing number of divisions, and out of 2 schedules with the same C_{MAX} we prefer the one with fewer divisions.

We assume task lengths p_i, division threshold k and lengths of all divided task parts to be integer. This doesn't make our solution less general, since real numbers within a few orders of magnitude can be scaled to integers with reasonable precession. Using integers eliminates rounding errors and simplifies calculations.

[1]Actually 1, since we assume lengths of all tasks and their parts are integer.

[2]See [6] or [8] for description of three-field notation of scheduling problems.

[3]By that we mean some abstract portions of work to be done on abstract machines, not actual microprocessors.

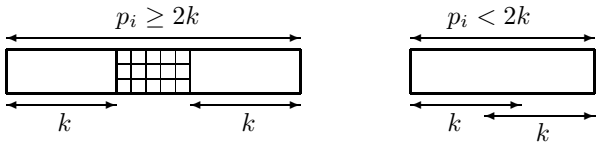

Fig. 1. Example of divisible task (left) and non-divisible task (right). First task can be divided anywhere within the grated area.

In [4] two similar models were presented. In first tasks could be preempted after being processed for at least k time. In second tasks could be preempted after being processed for a multiple of k, which allowed for easier task switching. Algorithms for solving both models were proposed, however there were no restriction on lengths of final parts of tasks, so in some cases they could be processed for a very short time after preemption.

III. TSRP3 algorithm

A. TSRP3 introduction

TSRP3 was developed by Tomasz Barański while working on research thesis [2] and based on earlier work by Michał Bakałarczyk [1]. It is basically a heavily modified version of McNaughton's algorithm with best-fit heuristics, and it never divides any task in more than 2 parts.

1) Sorting and grouping Tasks: Before TSRP3 begins scheduling all tasks are sorted and divided into four groups:

T_{SK}			p_j	$\leq$	k	Shorter than k
T_{ND}	k	$<$	p_j	$<$	$2k$	Not divisible
T_{DIV}	$2k$	$\leq$	p_j	$<$	$3k$	Divisible
T_{GD}	$3k$	$\leq$	p_j			Well divisible

Divisible tasks in groups T_{GD} and T_{DIV} are generally scheduled shortest to longest, while non-divisible T_{ND} and T_{SK} are generally scheduled longest to shortest. The longer a divisible task is, the easier it is to divide in such a way, that schedule of current processor has desired length. The shorter a non-divisible task is, the easier it is to fit into schedule on current processor. Indivisible tasks from T_{ND} generally cause the most problems while scheduling, so TSRP3 tries to schedule them first.

2) Estimating C_{MAX}: TSRP3 starts scheduling from the first processor and in each step either assigns some task or task fragment to current processor or closes it,

and moves to next processor. This decision is based on task lengths and current approximation of schedule length $worst \geq C_{MAX}$. After TSRP3 finishes scheduling tasks on a processor it moves to next one, and never returns to any previous processor. To explain *worst* we first need to introduce a few other estimates.

While scheduling tasks we always know the number of current processor *proc*, hence the number of processors on which we can still schedule tasks *prc* is (2). We also know the sum of lengths of all unscheduled tasks[4] σ and loads of all processors t_Z^j including current t_Z^{proc}.

$$C^* := \max\left\{\max_j\ p_j,\ \left\lceil\frac{1}{n}\sum_{j=1}^{n} p_j\right\rceil\right\} \tag{1}$$

McNaughton's lower bound on schedule length is defined in (1). Since only one processor can work on each task at any given time, no schedule can be shorter than the longest task. All schedules must also be at least as long as the sum of all task lengths divided by the number of processors.[5] C^* is calculated only once, while (2) to (6) must be recomputed once for each processor.

$$prc := m + 1 - proc \tag{2}$$

Number of processors where we can still schedule tasks *prc* must account for current processor as well, hence in (2) we add 1 to $m - proc$.

$$slack := prc \cdot C^* - \sigma - t_Z^{proc} \tag{3}$$

In (3) *slack* is the total amount of extra space that we could in theory allocate among remaining processors without going over C^*. To put it another way: *slack* tells how "sloppy" we can be scheduling tasks, while still (in theory) being able to complete optimal schedule.

$$hb := \begin{cases}\left\lceil\frac{\sigma}{prc}\right\rceil & \text{if } slack \leq 0,\\ C^* & \text{if } slack > 0.\end{cases} \tag{4}$$

As was shown in explanation of McNaughton's lower bound on C_{MAX}, any schedule must be at least as long as the sum of task lengths divided by the number of processors, hence form *prc* and σ we can derive approximation *hb* (4). If $slack < 0$ definition (4) is equivalent to $hb := C^* + \left\lfloor\frac{-slack}{prc}\right\rfloor$.

$$slack^{eff} := \begin{cases}\left\lfloor\frac{slack}{prc}\right\rfloor & \text{if } hb = C^* \text{ and } slack > 0,\\ 0 & \text{else.}\end{cases} \tag{5}$$

Effective slack $slack^{eff}$ (5) is *slack* divided among processors that we have left. It is assumed, that if on current processor we have $t_Z^{proc} \geq worst - slack^{eff}$, it is safe to close current processor and move to next one without increasing C_{MAX}.

$$lvb := \begin{cases}C^* - slack - (hb - C^*)(prc - 1) & \text{if } slack < 0,\\ C^* - slack & \text{if } slack \geq 0.\end{cases} \tag{6}$$

Approximation $lvb \geq 0$ in (6) is defined as "What is the shortest schedule on current processor that won't increase *hb* when we change processor to next one".

$$worst := \max\left\{C^*,\quad hb,\quad \max_{j\in 1..prc+1} t_Z^j\right\} \tag{7}$$

Our final estimate of C_{MAX} is *worst* defined in (7). It is the smallest number that is at least as big as C^* and current *hb* and greatest processor load so far. *TSRP3 tries to make load of current processor as close to worst as possible before closing it, and moving to next processor.* This is explained in detail in next subsection.

3) Load zones: TSRP3 behaves differently depending on the load of current processor t_Z^{proc}. Load zones are based on our current estimate *worst*. They are:

$$\begin{array}{lccccc}(S_1) & 0 & \leq & t_Z^{proc} & < & worst - 2k\\ (S_2) & worst - 2k & \leq & t_Z^{proc} & < & worst - k\\ (S_3) & worst - k & \leq & t_Z^{proc} & < & worst\\ (S_4) & worst & \leq & t_Z^{proc} & & \end{array}$$

4) Choosing the task that fits best on current processor and goal function dist(x): [6]

TSRP3 uses best-fit heuristics to determine which task or task part is the best candidate to schedule on active processor before moving to next one. It is generally best to close processor when $lvb \leq t_Z^{proc} \leq worst$ and t_Z^{proc} is as close to *worst* as possible. As can clearly be seen in (7) increasing t_Z^{proc} above *worst* will increase *worst* and thus possibly C_{MAX}. This is still the best course of action under some circumstances, but should be severely discouraged when it comes to choosing among other options. If current processor is closed when its load was below *lvb*, future *hb* will increase. This is almost as bad as going above *worst*, but its effects may be spread among all remaining $prc - 1$ processors. The "punishment" for schedule length beyond $[lvb, worst]$ can be some sufficiently big number, like $4k$ or $2k + worst - lvb$. The former is more convenient, and the latter more correct. From these considerations we get the following function that can choose the best candidate (or no candidate) to schedule on current processor:

$$dist(x) := \begin{cases}4k + \frac{lvb - x}{prc - 1} & \text{if } x < lvb,\\ worst - x & \text{if } x \in [lvb, worst],\\ 4k + x - worst & \text{if } worst < x.\end{cases} \tag{8}$$

[4] In some cases we may not change processor after task division, so σ has to include t_Z^{proc+1}

[5] We compute *ceiling* to get an integer value. For example if sum of task lengths is 31 and we have 3 processors, C_{MAX} with integer task parts' lengths cannot be less than 11.

[6] This is actually $dist(x, worst, lvb, prc)$, but we abbreviated it to $dist(x)$

TABLE I
EXPLANATION OF USED SYMBOLS

Symbol	Meaning	See	Upd.
n	Total number of tasks	—	No
m	Total number of processors	—	No
p_j	Durations of tasks $j \in 1 \dots n$	—	No
C_{MAX}	Schedule length	—	No
C^*	Lower bound on C_{MAX}	(1)	No
$proc$	Current processor $proc \in 1 \dots m$	—	Proc.
prc	Number of processors left	(2)	Proc.
t_Z^j	Load of processor j. Usually t_Z^{proc}	—	Yes
σ	Sum of unscheduled task durations	—	Yes
$slack$	Amount of "Slack space" under C^*	(3)	Proc.
hb	Estimate C_{MAX} from σ and prc	(4)	Proc.
lvb	Lowest t_Z^{proc} that won't increase hb	(6)	Proc.
$worst$	Current estimate of C_{MAX}	(7)	Yes.
$slack^{eff}$	Amortized slack space on $proc$	(5)	Proc.

When choosing task or task part z_i to assign to current processor, TSRP3 finds z_i, for which $dist(t_Z^{proc} + p_i)$ or $dist(t_Z^{proc} + length)$ is the lowest. If $dist(t_Z^{proc})$ is lower than for any task part we can schedule, TSRP3 moves to the next processor without scheduling anything on current one. It is more convenient to make $dist(x)$ a real function. Integer values can be used, but for $x < lvb$ you have to use $\lceil dist(x) \rceil$ and choose the longest task among those with the same (lowest) value of $dist()$.

Table I presents a list of symbols used by TSRP3, along with references to equations defining these symbols. Column Upd. says how often a variable is updated: No means a fixed parameter, Proc. means update once per processor, and Yes means update before scheduling another task.

B. TSRP3 Step by step

TSRP3 begins from the first processor. It assigns tasks to the current (open) processor, trying to make its load as close to $worst$ as possible. When current processor is sufficiently loaded, TSRP3 closes it, and moves to (opens) next processor, which in turn becomes the current processor. For each assignment it generally chooses some task or task part z_i, for which $dist(t_Z^{proc} + z_i)$ is lowest. This task or task part is called candidate, and may be compared to more tasks, scheduled, or discarded if $dist(t_Z^{proc} + z_i) > dist(t_Z^{proc})$ where t_Z^{proc} is the load of current processor. If a task is divided, it's remaining part is assigned to the next processor. TSRP3 can be presented in the following steps:

1) Sort and divide tasks in groups T_{SK}, T_{ND}, T_{DIV}, T_{GD}. Compute C^*.
2) If there are at least m well divisible tasks T_{GD}, schedule all tasks of length C^* on separate processors.
3) If there are no unscheduled tasks, finish.
4) If $m \geq n$, schedule all remaining tasks on separate processors and finish.
5) If $proc = m$, schedule all remaining tasks on $proc$, and finish.
6) Recompute $slack$, $slack^{eff}$, hb, lvb, $worst$, updating them once per processor.
7) Update t_Z^{proc} and $worst$. If $(worst - slack^{eff}) \leq t_Z^{proc}$ and $0 < t_Z^{proc}$ change processor to next one and go to step 5.
8) Take action depending on zone in which t_Z^{proc} is. This is explained in detail below, and may result in scheduling some task or task part, changing processor to the next one, or choosing a candidate task to schedule.
9) Choose candidate task to schedule. Take into account:
   - Candidate from step 8 if chosen
   - Longest and shortest tasks in each group.
   - Task division
   - *combo* (divides 2 or 3 tasks at once, explained later).

   If $dist(t_Z^{proc} + z_i) \leq dist(t_Z^{proc})$ for chosen candidate z_i, schedule it on current processor, otherwise move to next processor. Go to step 3

In step 2 we slightly reduce number of divisions in special cases, with lots of well divisible tasks. Step 4 takes care of some trivial cases. It is better to compare the number of unscheduled tasks to number of open processors except current one, rather than total numbers of processors and tasks. Steps 3 to 9 constitute the main loop of TSRP3. Step 7 is there to avoid divisions where not necessary. If $slack^{eff}$ is positive, we have a good chance to complete optimal schedule. Step 8 is the most complex, and is explained below.

Following subsections are divided by zones S_1 to S_4, and in one run of step 8 only one subsection gets executed depending on t_Z^{proc}. When a task is scheduled, it is considered to be removed from appropriate task set, therefore "$T_{SK} \neq \emptyset$" means "there is at least one unscheduled task shorter than k". In some cases we use term "divisible task" z_i. It means that $z_i \in T_{DIV} \cup T_{GD}$, and we generally use it, when we want to select as candidate the shortest task, that we can conveniently divide to make load of current processor equal to $worst$. Each subsection consists of sequentially checking some conditions and executing some instructions, such as scheduling a task. If conditions for an item are met, then its instructions are executed, including possibly jump to step 3 described above. If conditions for an item are not met, ignore it's instructions, and jump to next item on the same level. By making some task z_i a candidate to assign in step 9 we mean comparing it first to current candidate (if any) with $dist()$ and only making z_i candidate, if it's $dist(p_i + t_Z^{proc})$ is lower than for current candidate.

1) Step 8, zone S_1: $t_Z^{proc} \in [0,\ worst - 2k)$*:*

- If $T_{ND} \neq \emptyset$, choose as candidate the longest non-divisible task z_i such that either $p_i \leq (worst - k - t_Z^{proc})$ or $(worst - slack^{eff} - t_Z^{proc}) \leq p_i$. If the latter condition is met, or z_i is the longest task in T_{ND}, schedule that task on current processor and go to step 3.

- If there are at least *prc* well divisible unscheduled tasks left in T_{GD}
  - Find the shortest divisible task $z_i \in T_{DIV} \cup T_{GD}$ such that either $(p_i + t_Z^{proc}) \in [worst - slack^{eff}, worst]$ or $(p_i + t_Z^{proc}) \leq (worst - k)$. If such a task exists schedule it and go to step 3.
  - Find the shortest divisible task z_i such that $(p_i + t_Z^{proc}) \geq (worst + k - slack^{eff})$ If it exists divide it, scheduling part of length $min\ \{worst - t_Z^{proc},\ p_i - k\}$ on processor *proc* and the rest on $proc + 1$, change current processor to next one and go to step 3.
- If there are more unscheduled tasks in T_{SK} than in $T_{DIV} \cup T_{GD}$, schedule the longest task from T_{SK} on current processor and go to step 3.
- If $T_{DIV} \cup T_{GD} \neq \emptyset$
  - If the shortest divisible task z_i is longer than $(worst - k - t_Z^{proc})$ and shorter than $(worst + k - slack^{eff} - t_Z^{proc})$ and $(p_i + t_Z^{proc}) \notin [worst - slack^{eff},\ worst]$ and there are unscheduled tasks in T_{SK}, keep scheduling them on current processor until they run out, or one of above conditions is no longer satisfied or scheduling even shortest task would cause t_Z^{proc} to increase above $(worst - k)$.[7]
  - Choose the shortest divisible task z_i as candidate.
  - If task z_i is longer than $(worst - k - t_Z^{proc})$ and shorter than $(worst + k - slack^{eff} - t_Z^{proc})$ and $(p_i + t_Z^{proc}) \notin [worst - slack^{eff},\ worst]$, choose as candidate z_i the shortest divisible task of length at least $(worst + k - slack^{eff} - t_Z^{proc})$. If it does not exist, then make the longest divisible task candidate. [8]
  - If $(p_i + t_Z^{proc}) \leq (worst - k)$ or $(p_i + t_Z^{proc}) \in [worst - slack^{eff}, worst]$, schedule z_i on current processor and go to step 3.[9]
  - If $(p_i + t_Z^{proc}) \geq (worst + k - slack^{eff})$ divide z_i, scheduling part of length $min\ \{worst - t_Z^{proc},\ p_i - k\}$ on processor *proc* and the rest on $proc + 1$, change current processor to next one and go to step 3.
  - If there are at least 2 unscheduled divisible tasks and $(worst - t_Z^{proc}) \geq 2k$ and $t_Z^{proc} \geq k$, try dividing 2 or 3 tasks using *combo*[10] heuristics. If successful, change processor to the next one and go to step 3.
  - Find the longest divisible task such that $(p_i + t_Z^{proc}) \leq worst$. If it exists, make it candidate to add z_i.
- Go to step 9.

[7]If there is a problem with division, t_Z^{proc} can be increased by scheduling some short tasks.

[8]By dividing task as short as possible, we preserve longer divisible tasks to be scheduled on another processor.

[9]We avoid zone S_3 with this check.

[10]Explained in III-C.

Operations in this zone are the most complex, because we are actively avoiding zones S_3 and S_4 except for range $[worst - slack^{eff}, worst]$. We also divide long tasks, if scheduling them would make $t_Z^{proc} \geq (worst + k - slack^{eff})$. Before going to step 9, where we do some operations common to all zones, we may or may not choose a task as candidate to schedule. If selected, it is compared in step 9 to other possibilities, including scheduling nothing and moving to next processor.

2) Step 8, zone S_2: $t_Z^{proc} \in [worst - 2k,\ worst - k)$*:*

- If $T_{ND} \neq \emptyset$, find the longest task $z_i \in T_{ND}$ such that $(p_i + t_Z^{proc}) \leq worst$. If it exists make it candidate to schedule. If it satisfies $(p_i + t_Z^{proc}) \geq (worst - slack^{eff})$ schedule it, and go to step 3.
- If there is some divisible unscheduled task left.
  - If the shortest divisible task is shorter than $(worst + k - slack^{eff} - t_Z^{proc})$, and there are unscheduled tasks in T_{SK}, keep scheduling them on current processor until they run out, or the above condition is no longer satisfied or scheduling even the shortest task would cause t_Z^{proc} to increase above $worst - k$.
  - Find the shortest divisible task z_i such that $(p_i + t_Z^{proc}) \geq (worst + k - slack^{eff})$. If it exists, divide z_i, scheduling part of length $min\ \{worst - t_Z^{proc},\ p_i - k\}$ on processor *proc* and the rest on $proc + 1$, change current processor to next one and go to step 3.
- Go to step 9.

In this zone we first try to find a non-divisible task of length $(p_i + t_Z^{proc}) \in [worst - slack^{eff}, worst]$. If that fails, we try to divide some task instead, possibly increasing t_Z^{proc} with tasks from T_{SK}. Other actions, common to all zones, are considered in step 9.

3) Step 8, zone S_3: $t_Z^{proc} \in [worst - k,\ worst)$*:*

- If there is at least one unscheduled task in T_{SK}, find the longest task z_i in T_{SK} such that $(p_i + t_Z^{proc}) \leq worst$. If it exists, make z_i the candidate to schedule. If $(p_i + t_Z^{proc}) \geq (worst - slack^{eff})$ schedule z_i, and go to step 3.
- Go to step 9.

Here the only actions that make sense are: fitting a task from T_{SK} or scheduling task part of length k or changing current processor to next one without scheduling anything. Only the first option is not covered in step 9.

4) Step 8, zone S_4: $t_Z^{proc} \geq worst$*:*

- Move to next processor, and go to step 9.

We never actually get to this zone, because condition $t_Z^{proc} \geq (worst - slack^{eff})$ is checked in step 7, but it is safer to include this check.

C. Dividing 2 or 3 tasks with combo

Description of TSRP3 algorithm has two references to *combo* heuristics, that tries to divide 2 or 3 tasks before changing current processor to next one. This is useful,

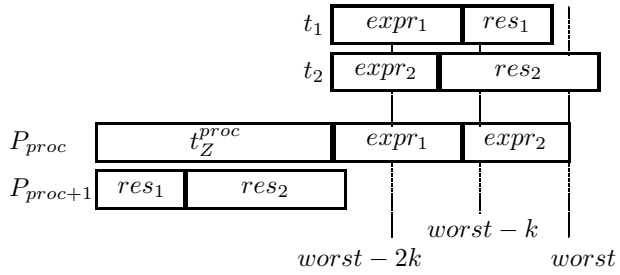

Fig. 2. Dividing two tasks with combo.

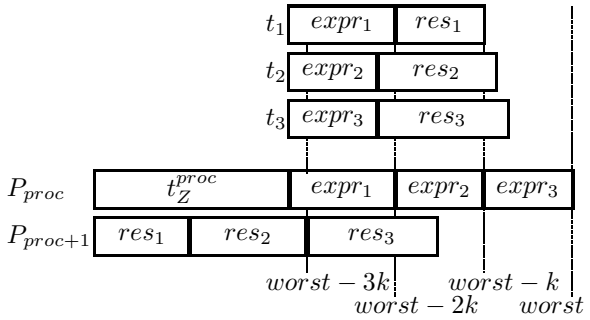

Fig. 3. Dividing three tasks with combo.

when there are some short divisible tasks, and scheduling any one task, divided or not, will guarantee t_Z^{proc} to be in either zone S_3 or S_4 and outside $[worst - slack^{eff},\ worst]$. While this is counter to our stated goal of reducing the number of divisions, our *primary* aim is still to minimize C_{MAX}.

Consider the following example: $slack^{eff} = 0$, $t_Z^{proc} = (worst - 2k - \epsilon)$, all indivisible tasks have lengths $p_i > (k + \epsilon)$, and all divisible tasks have lengths $p_i = (2k + \lambda_i)$, where $\epsilon, \lambda \geq 0$ and $\epsilon \neq \lambda_i$ and $\forall i\ \epsilon - k < \lambda_i < \epsilon + k$. No single task can be assigned or divided so as to avoid zones S_3 and S_4 or make load of current processor equal to $worst$. This can be done by increasing t_Z^{proc} by about k, or more precisely between $\epsilon + k - \lambda_i$ and $\epsilon + k$ for some i. It can be achieved by double division of tasks, as shown in Fig. 2.

If there are 2 divisible tasks z_1 and z_2 such that $p_1 + p_2 \geq 4k + \epsilon - slack^{eff}$ [11] and they can be divided in a way that satisfies (9) to (12), and scheduled like in Fig. 2, then it won't result in any conflicts (overlapping task execution on different processors).

$$t_Z^{proc} \geq k \tag{9}$$

$$expr_1,\ res_1,\ expr_2,\ res_2 \geq k \tag{10}$$

$$t_Z^{proc} \geq res_1 \tag{11}$$

$$res_1 + res_2 \leq t_Z^{proc} + expr_1 \tag{12}$$

$$res_1 + res_2 + expr_2 \leq worst \tag{13}$$

Inequality (9) is a sanity check derived from (10) and (11). Inequality (10) ensures that there are no task parts shorter than k. Inequality (11) is more precise than (9) and makes conflict between $expr_1$ and res_1 impossible. Inequality (12) does the same for $expr_2$ and res_2. Finally inequality (13) forbids *combo* on too long tasks, and makes sure that $t_Z^{proc+1} \leq (worst - k)$.

[11] This is a bit too restrictive. In some cases the next best course of action chosen with *dist*() is so bad, that it is better to use *combo* anyway, even if it makes $t_Z^{proc} \notin [worst - slack^{eff},\ worst]$. If this next best thing would be assigning whole task z_i longer than $(worst - t_Z^{proc})$, scheduling 2 task parts shorter than p_i may be a better option. On the other hand when there are few processors left, and we are to either move to next processor while $t_Z^{proc} < lvb$, or assign some task z_i and land in zone S_3 or S_4, scheduling 2 task parts whose sum is longer than p_i and shorter than the resulting hb, may also be better.

As it turns out, sometimes dividing 2 tasks with *combo* is not enough to make $t_Z^{proc} = worst$. Such situation arises when there are only divisible tasks left, there is at least 3 of them, $t_Z^{proc} = (worst - 3k + \epsilon)$ and $p_i = (2k + \lambda_i)$ where $\forall i\ 0 \leq \epsilon < \lambda_i < \frac{k+\epsilon}{2}$. In this case scheduling whole task or dividing 2 tasks causes t_Z^{proc} to be in zone S_3, which is undesirable. To solve this problem we find 3 tasks that satisfy inequalities (14) to (19) and assign their parts like in Fig. 3.

$$t_Z^{proc} \geq k \tag{14}$$

$$expr_1,\ res_1,\ expr_2,\ res_2,\ expr_3,\ res_3 \geq k \tag{15}$$

$$t_Z^{proc} \geq res_1 \tag{16}$$

$$t_Z^{proc} + expr_1 \geq res_1 + res_2 \tag{17}$$

$$t_Z^{proc} + expr_1 + expr_2 \geq res_1 + res_2 + res_3 \tag{18}$$

$$res_1 + res_2 + res_3 + exp_3 \leq worst \tag{19}$$

It is generally best to choose for *combo* as short a tasks as possible, divide shortest first and longest last, and always make *expr* as long as possible, while reserving k space for all future *expr*.

Scheduling res_i on different processors would relax some restrictions, but it would complicate computing *sigma*, *hb*, *lvb*, and require keeping track of loads on too many processors. We do not recommend it, and in practice task divisions between current and next processors suffice.

It is never necessary to divide more than three tasks at once. If scheduling any divisible task z_i would cause t_Z^{proc} to end up in S_3 or S_4, than increasing t_Z^{proc} by at most $2k$ by dividing at most two other tasks, makes $(t_Z^{proc} + p_i) \geq (worst + k)$. Therefore dividing at most 3 tasks will always suffice, provided it is possible.

D. Sufficient conditions of optimality for TSRP3

With $k = 0$ (or $k = 1$ for all task parts' lengths integer) optimal schedule can be completed in O(n) time with McNaughton's algorithm. Scheduling indivisible tasks is in general an NP-hard problem (although there are reasonably good rough algorithms). The question arises of where is the threshold value k_l between these two cases or how many well divisible tasks are needed for guaranteed

completion of optimal schedule in polynomial time. This subsection addresses these questions by showing sufficient conditions for TSRP3 to complete an optimal schedule. As shown in section V these conditions are very restrictive, and in fact TSRP3 often completes optimal schedule with fewer than m divisible tasks.

Case a)

$$\text{For } 0 \leq l \leq m \text{ tasks } p_i = C^* \quad (20)$$

$$\text{For other tasks: } p_i \leq C^* - 2k \quad (21)$$

$$\text{There are } 2(m-l-1) \text{ tasks in } T_{GD} \quad (22)$$

$$\text{For pairs of them } p_a + p_b \leq (C^* + k) \quad (23)$$

All tasks (20) of length C^* are assigned to separate processors. This way we will decrease number of divisions while still getting optimal schedule. Condition (23) stands for this: We have at least $2(m-l-1)$ well divisible tasks, and we can form $(m-l-1)$ pairs, such that sum of their lengths is $\leq (C^* + k)$. This may be done by choosing some subset M of T_{GD}, and pairing longest and shortest tasks in M. They may be used for *combo*. Let us consider what actions TSRP3 will take depending on t_Z^{proc}. If $t_Z^{proc} = 0$, then after scheduling any task we have $t_Z^{proc} \leq C^* - k$ because (21), so we avoid zones S_3 and S_4. If $t_Z^{proc} \leq C^* - 2k$ and by scheduling some task z_i we would end up in zone S_3, we can do a double division instead. If $C^* - 2k < t_Z^{proc} \leq C^* - k$, we can divide one task from (22). Therefore we never get to zones S_3 and S_4, and we always have enough long tasks to make $t_Z^{proc} = C^*$ by dividing one or two of them.

Case b)

$$\text{For } 0 \leq l \leq m \text{ tasks } p_i = C^* \quad (24)$$

$$\text{For other tasks: } p_i \leq C^* - 2k \quad (25)$$

$$\text{There are } (m-l-1) \text{ well divisible tasks} \quad (26)$$

$$\text{And } 2(m-l-1) \text{ more divisible tasks} \quad (27)$$

$$\text{Sum of 1 (26) and 2 (27) is } \leq C^* + 2k \quad (28)$$

Now we have at least $(m-l-1)$ well divisible tasks and additionally at least $2(m-l-1)$ divisible tasks. This is different from above case in that we can have fewer well divisible tasks, but need more divisible tasks overall. We also need to be able to combine these tasks in triplets satisfying (28). Reasoning proceeds as above, but if necessary we divide 3 tasks.

Case c)

$$\text{For } 0 \leq l \leq m \text{ tasks we have } p_i = C^* \quad (29)$$

$$\text{For other tasks: } p_i \leq C^* - 2k \quad (30)$$

$$\text{There are } (m-l-1) \text{ well divisible tasks} \quad (31)$$

$$\text{Sum of tasks in } T_{SK} \text{ is } \geq 2(m-l-1) \quad (32)$$

If by scheduling a well divisible task z_i we get $t_Z^{proc} > (C^* - k)$, then we need to extend t_Z^{proc} by at most 2k to divide z_i and make $t_Z^{proc} = C^*$. Tasks shorter than k are useful for this. Since zone of possible division as shown in Fig. 1 for well divisible task is at least k long, and T_{SK} are at most k long, there never will be any trouble fitting some T_{SK} on current processor to divide z_i as long as there are enough of them.

Case d)

$$\text{For } 0 \leq l \leq m \text{ tasks } p_i = C^* \quad (33)$$

$$\text{For other tasks: } p_i \leq C^* - 2k \quad (34)$$

$$\text{We have } (m-l-1) \text{ tasks of length } p_i \geq 4k \quad (35)$$

$$\text{And } (m-l-1) \text{ tasks of length } p_i \geq k \quad (36)$$

Tasks (36) are there to avoid some malicious cases. Long tasks (35) have division zone of length at least $2k$ This makes it easy to make loads of all processors at most C*. If by scheduling an indivisible task, we would get to zone S_3, we can instead divide one of tasks (35). If by scheduling a divisible task we would get to zone S_3, we can do a double division instead.

Case e)

$$|a| + |b| + |c| + |d| \geq (m-l-1) \quad (37)$$

We have a combination of cases a) b) c) d) and the sum of number of processors, for which at least one of them can be used to make $t_Z^{proc} = C^*$ is at least $(m-l-1)$. We use different methods to complete optimal schedule.

1) Conclusions: These are sufficient, but not necessary conditions for completion of optimal schedule by TSRP3. As shown in section V, in practice TSRP3 will usually find optimal schedule as long as there are at least $m/2$ divisible tasks and $2m$ other tasks. Having at most l, where $0 \leq l \leq m$ tasks of length C^* and other tasks no longer than $C^* - 2k$ is forced just to exclude malicious data sets.

A simple rule for finding k_l for which TSRP3 will complete optimal schedule is as follows (38):

$$\begin{aligned} L(i) &:= \text{Length of } i\text{-th longest task} \\ k_a &= \left\lfloor \frac{L(2m)}{3} \right\rfloor \\ k_b &= \min \left\{ \left\lfloor \frac{L(3m)}{2} \right\rfloor, \left\lfloor \frac{L(m)}{3} \right\rfloor \right\} \\ k_l &= \max \{k_a, k_b\} \end{aligned} \quad (38)$$

As shown in [2], computational complexity of TSRP3 is $O(n \cdot \log(n))$, which is comparable to sorting. Memory requirements depend on data structures used for schedule, but are around 100b or less per task.

IV. TSRP4 Algorithm

TSRP3 works well when there are many long divisible tasks, but noticeably worse when there are few or none. TSRP4 is a method for making TSRP3 more robust with indivisible tasks. As can be concluded from (7) on page 2 *worst* either increases or stays the same when we move to the next processor. TSRP3 tries to make the load of current processor as close to current *worst* as possible, which may lead to overloading last processors while underloading first processors. To address this we introduce parameter $C_{OGR} \geq C^*$, and modify (7) to make sure that $worst \geq C_{OGR}$. Our goal then becomes to find such C^*_{OGR} that modified TSRP3 can complete a schedule of length C^*_{OGR}, but not $C^*_{OGR} - 1$.

This is accomplished by first setting C_{OGR} to C^*, and running TSRP3. If the length of resulting schedule len is C^* then finish, else $C^* < C^*_{OGR} \leq len$ and C^*_{OGR} can be found with bisection of $(C^*, len]$ by completing $O(\log(len - C^*))$ schedules with modified TSRP3. Schedule completed with TSRP4 is never longer than completed with TSRP3 for the same input.

V. Experimental comparison of TSRP3 with other scheduling algorithms

To show behavior of TSRP3 and TSRP4 depending on k, they were tested for 10 randomly generated data sets. Each contained 500 tasks with Gauss distribution with $\mu = 100$ and $\sigma = 30$ to schedule on 100 processors. If there are fewer tasks than processors then scheduling is trivial, and if there are many (10+) tasks of wildly varying lengths, there is usually a good schedule without task divisions. Scheduling is most problematic, when there are few tasks per processor. Even statistically small sample of 10 data sets is enough to form opinion on behavior of TSRP3 and TSRP4 for changing k. The results were averaged and are shown in Fig. 4. They were taken from [2], where a slightly different version of TSRP3 was implemented.

Some labels in Fig. 4 require explanation. C^* is McNaughton's lower bound on schedule length (1). LPT and Multi-fit are algorithms scheduling tasks without dividing them. LPT assigns the longest task to the first processor that becomes idle. Multi-fit was our inspiration for TSRP4. It has "superior" and "subordinate" parts. Superior part manipulates parameter C_{OGR} to find C^*_{OGR} as described in IV. Subordinate part starts from the first processor and in each step either schedules on current processor the longest task that fits under C_{OGR} or moves to the next processor.

In Fig. 4 there are two main areas. Below $k = 60$ there are at least m divisible tasks, and there are some well divisible tasks. Above $k = 80$ the number of divisible tasks drops rapidly.

In III-D it was shown, that with sufficient number of divisible tasks TSRP3 can complete optimal schedule of length C^*. There may however exist some optimal schedules with fewer task divisions. Sufficient conditions for completing optimal schedule given in III-D turn out to be very restrictive. During tests and barring maliciously chosen data sets TSRP3 was generally not failing at completing optimal schedules until number of divisible tasks dropped below $m/2$. When there are few or none divisible tasks, TSRP3 usually completes schedules longer than LPT, which does not divide tasks at all. When there are no divisible tasks, and k changes, length of TSRP3 schedule fluctuates, because border between T_{SK} and T_{ND} moves, so tasks are scheduled in different order. This leads to a somewhat paradoxical observation that sometimes a shorter schedule may be completed by increasing k. These fluctuations generally stop well before all tasks are classified as T_{SK}.

Algorithm TSRP4 was written to mitigate shortcomings of TSRP3 when there are few or none divisible tasks. As you can see in Fig. 4 schedule completed with TSRP4 is never longer than with TSRP3. For relatively short k TSRP4 schedule length is optimal, and as k grows, C_{MAX} approaches length of schedule completed with Multi-fit. This shows similarity of these two algorithms. Fluctuations with changing k and no divisible tasks are also smaller. This is an improvement over TSRP3, but it may require completing several schedules to find C^*_{OGR}.

In paper [2] an even better scheduling algorithm was devised by completing 2 schedules: one with TSRP4 and another with a good scheduling algorithm that doesn't divide tasks: "LPT + PSS", and choosing shorter one, or one with fewer divisions, if both schedule lengths were the same. In short LPT + PSS uses local search to balance loads of processors in schedule completed by LPT.

Algorithm LPT + PSS doesn't divide tasks, and was developed by Tomasz Barański and presented in [2]. It has proven to be superior to both multi-fit and LPT in terms of C_{MAX}, but takes longer to compute. Several speed improvements were proposed to mitigate this. LPT + Presistent Simple Swap starts by computing a schedule with LPT and sorting processors by load. In each step it chooses the most loaded processor P_{MAX}, and the least loaded P_{MIN} and tries to balance their loads by either moving one task from P_{MAX} to P_{MIN} or switching a pair of tasks between them. It tries to balance loads of these two processors as much as possible. If successful it updates P_{MAX} and P_{MIN} and continues balancing loads, otherwise it changes P_{MIN} to another processor with schedule shorter than that of P_{MAX}. PSS stops when C_{MAX} reaches C^* or no further balancing can be done. Any feasible schedule without tasks division can be used as starting point for PSS, such as multi-fit or even scheduling all tasks on a single processor, but in [2] LPT + PSS produced the best results. For data sets used in Fig. 4 in 9 out of 10 cases LPT + PSS produced schedule with $C_{MAX} = C^*$. It may seem superior to TSRP4, but in some cases optimal schedule can only be obtained by dividing tasks.

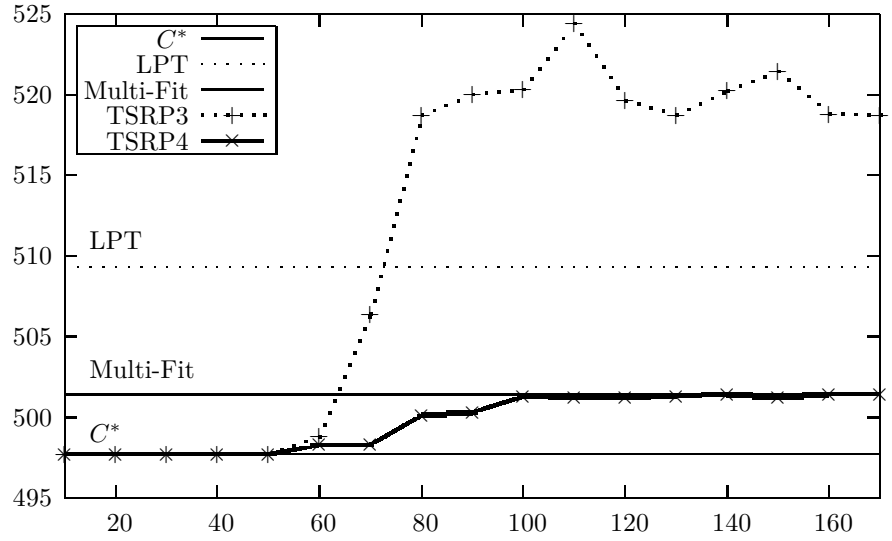

Fig. 4. Relation between k (horizontal axis) and C_{MAX} (vertical axis) for TSRP3 and TSRP4. Less is better. Results are averaged for 10 data sets with similar C*. While k was gradually increased, Task durations stayed the same. C^* is lower bound on schedule length. LPT and Multi-fit don't divide tasks, and are shown for comparison. As k increases, number of divisible tasks drops, and TSRP3 produces worse results. TSRP4 is much more robust.

VI. Concluding remarks

In this article we proposed TSRP3, a heuristic algorithm for completing schedules with partial task division. As we have shown in (III-D) it is guaranteed to find optimal schedule if there are at least $2m$ tasks $3k$ long or longer.

While schedules completed with TSRP4 may be satisfactory in general, and optimal with enough divisible tasks, in some cases dividing tasks in at most two parts is not enough. We will illustrate this with a simple example: We have $m = 3$ processors and $n = 7$ tasks, with $k = 4$. There is one task of length $p_1 = 12$ and six tasks $p_i = 6$. To get optimal schedule we need to divide the longest task in 3 parts of length 4 and schedule one on each machine with two indivisible tasks. This way we get $C_{MAX} = C^* = 16$. If we divide the longest task in at most 2 parts, then no part of it will be on one processor P_3. Therefore we either schedule 2 or 3 indivisible tasks on that processor. If we schedule 3 tasks, their combined length is $6 \cdot 3 = 18$. If we schedule only 2 tasks, than we need to divide tasks of length $12+6\cdot 4 = 36$ among 2 processors, so schedule won't be shorter than 18. Therefore in this example we need to cut some task in more than 2 fragments to get optimal schedule. A method of doing this is hinted in [2].

TSRP3 or TSRP4 may be used to solve one-dimensional bin-packing problem, like cut-weld problem, by changing m, running TSRP4 and comparing resulting C_{MAX} to target bin size. This is similar to method used in TSRP4 to find C^*_{OGR}. Please note, however, that in bin-packing problem there are no conflicts. We recommend using a dedicated algorithm for bin-packing rather than TSRP4. We suggest introducing some cost of dividing tasks for further study of TSRP3. Divivision of tasks in more than two parts is also worth investigating.

References

[1] Michał Bakałarczyk, "Szeregowanie zadań z oraniczoną podzielnością na procesorach równoległych", *Praca inżynierska*, PW EiTI, Warszawa 2006.

[2] Tomasz Barański, "Szeregowanie zadań z częściową podzielnością na procesorach równoległych", *Praca magisterska*, PW EiTI, Warszawa 2010.

[3] Krzysztof Trakiewicz "Modele i algorytmy optymalizacji rozkroju i spawania kształtowników", *Praca magisterska*, PW EiTI, Warszawa 2004.

[4] K. Ecker, R. Hirschberg "Task scheduling with restricted preemptions", p. 464-475, LNCS vol. 694, Springer-Verlag 1993.

[5] Nir Menakerman, Raphael Rom "Bin packing with item fragmentation", WADS 2001, LNCS 2125, p. 213-324, Springer-Verlag, Heidelberg 2001.

[6] A. Janiak "Wybrane problemy i algorytmy szeregowania zadań i rozdziału zasobów", Akademicka Oficyna wydawnicza PLJ, Warszawa 1999.

[7] Robert Mc Naughton, "Scheduling with Deadlines and Loss Functions", Management Science, Vol. 6, No. 1, 1959.

[8] B. Chen, C.N. Potts, G.J. Woeginger. "A review of machine scheduling: Complexity, algorithms and approximability.", Handbook of Combinatorial Optimization, Vol. 3, 1998, 21-169.

[9] Manfred Kunde "A multifit algorithm for uniform multiprocessor scheduling", Lecture Notes in Computer Science, 1982, Volume 145/1982, 175-185.