Noname manuscript No.
(will be inserted by the editor)

On contiguous and non-contiguous parallel task scheduling

I.Błądek · M.Drozdowski · F.Guinand · X.Schepler

the date of receipt and acceptance should be inserted later

Abstract In this paper we study differences between contiguous and non-contiguous parallel task schedules. Parallel tasks can be executed on more than one processor simultaneously. In the contiguous schedules indices of the processors assigned to a task must be a sequence of consecutive numbers. In the non-contiguous schedules processor indices may be arbitrary. Nonpreemptive schedules are considered. Given a parallel task instance, optimum contiguous and non-contiguous schedules can be of different length. We analyze the smallest instances where such a difference may arise, provide bounds on the difference of the two schedules lengths and prove that deciding whether the difference in schedule length exists is **NP**-complete.

Keywords Parallel tasks, contiguous scheduling, non-contiguous scheduling

a)

task	T_1	T_2	T_3	T_4	T_5	T_6	T_7
$size_j$	1	3	2	1	2	1	3
p_j	7	3	3	5	1	3	1

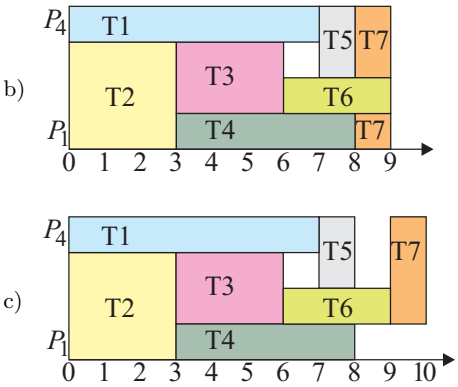

Fig. 1 Example of c/nc-difference. a) Instance data, b) optimum non-contiguous schedule, c) optimum contiguous schedule.

1 Introduction

Parallel tasks may require more than one processor simultaneously. The processors are granted either contiguously or non-contiguously. In the contiguous case indices of the processors are a range of consecutive integers. In the opposite case the indices may be arbitrarily scattered in the processor pool. In this paper we analyze the cases when for a parallel task instance, the lengths of optimum nonpreemptive contiguous and non-contiguous schedules are different. Non-contiguous schedules may be shorter because a contiguous schedule is also a feasible non-contiguous schedule, but not vice-versa. We will call such a situation *c/nc-difference*. An example of c/nc-difference is shown in Fig.1.

More formally our problem can be formulated as follows: We are given set $\mathcal{P} = \{P_1, \ldots, P_m\}$ of m parallel identical processors, and set $\mathcal{T} = \{T_1, \ldots, T_n\}$ of n parallel tasks. Each task $T_j \in \mathcal{T}$ is defined by its processing time p_j, and the number of required processors $size_j$, where $size_j \in \{1, \ldots, m\}$. For conciseness, we will be calling p_j task T_j length and $size_j$ task T_j size. Both processing times and task sizes are positive integers. We study two versions of the problem: either tasks are scheduled *contiguously*, or *non-contiguously*. In the former case the indices of the processors assigned to a

Corresponding author: Maciej Drozdowski. E-mail: Maciej.Drozdowski@cs.put.poznan.pl, Tel: (48) 616652981, Fax: (48) 618771525

I.Błądek, M.Drozdowski
Institute of Computing Science, Poznań University of Technology, Piotrowo 2, 60-965 Poznań, Poland

F.Guinand, X.Schepler
LITIS, University of Le Havre, 25, rue Philippe Lebon, BP 540, 76058 Le Havre Cedex, France

task must be consecutive. In the latter case, processor indices can be arbitrary in the range $[1, m]$. Scheduling is nonpreemptive and migration is disallowed. It means that task T_j started at time s_j must be executed continuously until time $s_j + p_j$ on the same set of processors. Schedule length (i.e. makespan) is the optimality criterion. We will denote by C^c_{max} contiguous, and by C^{nc}_{max} non-contiguous optimum schedules lengths for the given instance. In this paper we study the cases when $C^c_{max} > C^{nc}_{max}$. In the following discussion use "up"/"down" directions to refer to shifting tasks toward bigger/smaller processor indices. By "renumbering" a pair of processors we mean swapping the whole schedules on the two processors.

Since task widths $size_j$ are given and cannot be changed, we consider here a subclass of parallel task scheduling model called rigid tasks (Feitelson et al. 1997). According to the notation introduced in (Veltman et al. 1990; Drozdowski 2009), our scheduling problem may be denoted $P|size_j|C_{max}$. This problem is **NP**-hard, which follows from the complexity of the classic problem $P2||C_{max}$. The first publication on the rigid task scheduling seems to be (Lloyd 1981). The problem of scheduling various types of parallel task systems has been tackled in hundreds of publications and reporting them here is beyond the size and scope of this paper. Therefore, we direct an interested reader to surveys in (Dutot et al. 2004; Drozdowski 2009). The difference between contiguous and non-contiguous schedules has been first noticed in (Turek et al. 1992). Its existence has been acknowledged, e.g., in (Dutot et al. 2004; Baille et al. 2008). However, to the best of our knowledge the consequences of applying contiguous and non-contiguous schedules to the same instance have not been studied before.

The difference between contiguous and non-contiguous schedules has practical consequences in parallel processing. Parallel applications are composed of many threads running simultaneously and communicating frequently. It is advantageous to assign the threads of a single application to processors within a short network distance because communication delays are shorter and there are fewer opportunities for network contention with other communications (Bokhari and Nicol 1997; Bunde et al. 2004; Lo etal. 1997). In certain network topologies processor numbering schemes have been proposed to aid allocation of processors which are close to each other. In such network topologies ranges of consecutive processor indices correspond to locally connected processors. These can be various buddy processor allocation systems for 1-, 2-dimensional meshes, hypercubes and k-ary n-cube interconnection networks (Chen and Shin 1987; Li and Cheng 1991; Drozdowski 2009). Also other, more sophisticated processor numbering schemes have been proposed for this purpose (Leung et al. 2002). As a result, contiguous schedules correspond to assigning tasks to tightly connected processors. Thus, contiguous schedules are favorable for the efficiency of parallel processing. However, contiguous schedules are less desirable for a set of parallel tasks because it is harder to pack the tasks on the processors when they cannot be split between several groups of available processors. Hence, it is necessary to understand the difference between contiguous and non-contiguous schedules: When such a difference may arise. How much may be gained by going from a contiguous to a non-contiguous schedule.

In the domain of harbor logistics, the berth assignment problem (BAP) is one of the most studied problem in container terminal operations (Lim 1998; Bierwirth and Meisel 2010). In the discrete BAP, a quay is partitioned into berths. In a common case one berth may serve one ship at a time, and one ship requires several contiguous berths. After relaxing some other constraints, the BAP reduces to a contiguous scheduling problem. A berth corresponds to a processor and a ship corresponds to a job. Depending on its size, a ship requires a given number of berths. A job processing time is given by the corresponding ship handling duration. This duration may be fixed or may depend on the berths. Since vessels cannot be partitioned into several pieces, non-contiguous schedules are not practically relevant. However, non-contiguous makespan values provide lower bounds on contiguous schedules and thus lower bounds for the relaxed versions of BAP.

Further organization of this paper is the following: In Section 2 the smallest instances for which a c/nc-difference may exist are analyzed. Section 3 contains a proof that deciding if a c/nc-difference appears is **NP**-complete. In Section 4 it is shown that the ratio of contiguous and non-contiguous schedule lengths is bounded. In Section 5 we report on the simulations conducted to verify whether c/nc-difference is a frequent phenomenon on average. The last section is dedicated to conclusions.

2 Smallest instances

Here we study conditions under which a c/nc-difference can arise. We start with a couple of observations. Obviously, $C^{nc}_{max} \leq C^c_{max}$ because each contiguous schedule is also a valid non-contiguous schedule, but not vice versa.

Observation 1 *For c/nc-difference to arise, there must be more than two tasks with* $size_j > 1$.

If there were only one task T_j with $size_j > 1$, then it would be possible to renumber processors in an optimum non-contiguous schedule such that T_j were executed contiguously. If there are exactly two tasks T_i, T_j : $size_i, size_j > 1$, then in the optimum non-contiguous schedule T_i, T_j either share some processors or they don't. In the latter case it is again possible to renumber processors such that T_i, T_j are scheduled contiguously. If T_i, T_j share processors, then they are executed sequentially. Therefore, it is also possible to renumber processors such that the processors used only by T_i are assigned contiguously above the shared processors, the shared processors are assigned contiguously while the processors used only by T_j are assigned contiguously below the shared processors. Thus, a contiguous schedule of the same length would be obtained.

Observation 2 *For c/nc-difference to arise, there must be more than two tasks with $p_j > 1$.*

If $\forall\ T_j, p_j = 1$, then it is always possible to arrange tasks in the optimum non-contiguous schedule into contiguous allocations in each time unit of a schedule separately (by sorting task indices). The cases with one or two tasks longer than a unit of time are analogous to Observation 1 but moving tasks within the time units of a schedule is necessary instead of processor renumbering.

Theorem 3 *For c/nc-difference to arise, $C^{nc}_{max} \geq 4$.*

Proof. First we will show that all non-contiguous schedules of length $C^{nc}_{max} \leq 3$ can be rearranged to contiguous schedules of the same length. Then, we show that there is an instance which non-contiguous schedule has $C^{nc}_{max} = 4$ and the contiguous schedule has $C^{c}_{max} = 5$.

For $C^{nc}_{max} = 1$ rearrangement into a contiguous schedule is always possible by Observation 2. If $C^{nc}_{max} = 2$, then tasks must have lengths $p_j \in \{1, 2\}$. The tasks with $p_j = 2$ can be reassigned contiguously on the lowest processor numbers, and the remaining tasks with $p_j = 1$ are handled as in Observation 2. For $C^{nc}_{max} = 3$ processing times are $p_j \in \{1, 2, 3\}$. The tasks with $p_j = 3$ can be moved down to the lowest processor numbers (cf. Fig 2a and b). The tasks with $p_j = 2$ can be shifted down or up by swapping whole intervals of the schedule on the processors. The tasks executed in time units 1,2 are moved down (just above any tasks with $p_j = 3$), and the tasks executed in time units 2,3 are shifted up to the biggest processor numbers (Fig.2c). After these transformations we obtained a schedule in which tasks that run in the same interval are stacked one on another without being interleaved by the tasks from other intervals. Consequently, it is possible to rearrange the tasks in their time intervals to be executed contiguously (Fig. 2d). Hence, we have $C^{c}_{max} = C^{nc}_{max} = 3$.

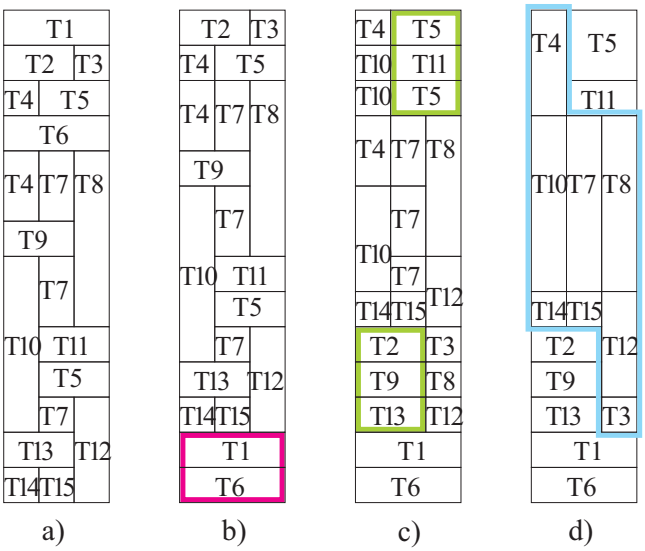

Fig. 2 Example of converting non-contiguous schedule of length $C^{nc}_{max} = 3$ to a contiguous schedule. a) Initial schedule. b) Shifting tasks with $p_j = 3$. c) Shifting tasks with $p_j = 2$. d) Rearranging tasks into contiguous allocations.

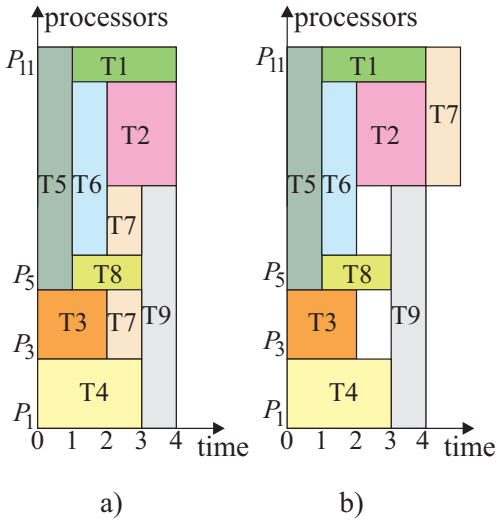

Fig. 3 C/nc-difference with the shortest possible C^{nc}_{max}. a) optimum non-contiguous schedule, b) optimum contiguous schedule.

In Fig.3 a schedule of length $C^{nc}_{max} = 4$ is shown. It can be verified, that there is no contiguous schedule shorter than $C^{nc}_{max} = 5$ as, e.g., presented in Fig.3b. □

A practical consequence of Theorem 3 is that if it is possible to build schedules in pieces of short length (at most 3 units of time), then rearrangement into a contiguous schedule of the same length is always possible.

Theorem 4 *For c/nc-difference to arise, $m \geq 4$.*

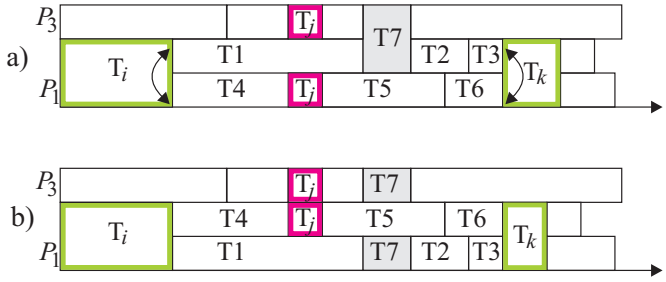

Fig. 4 Transforming a non-contiguous schedule on $m = 3$ to a contiguous schedule.

Proof. For $m = 2$ processors no task can be scheduled non-contiguously. For $m = 3$ a non-contiguous schedule can be converted to a contiguous schedule of the same length. The converting procedure scans a non-contiguous schedule from the beginning to the end for tasks scheduled on P_1, P_3, and then reschedules them such that they are executed on P_1, P_2 or P_2, P_3. Suppose T_j is the first task executed on P_1, P_3 (cf. Fig.4). We search for the latest task T_i preceding T_j and executed on two processors (or for the beginning of the schedule if such a task does not exist). Task T_i is scheduled contiguously (because T_j is the first task scheduled non-contiguously). We search for the earliest task T_k succeeding T_j and executed on the same processors as T_i (or for the end of the schedule if such a task does not exist). Then, we swap on the processors the intervals between T_i, T_k (Fig.4b). Consequently, all tasks until T_j are scheduled contiguously. The procedure proceeds until the last task executed on P_1, P_3.

In Fig.1 an instance with c/nc-difference on $m = 4$ processors is given. □

We finish this section with a conjecture motivated by the instance in Fig.1 with c/nc-difference with as few tasks as $n = 7$. No smaller instance have been found in our simulations (cf. Section 5).

Conjecture 1 For c/nc-difference to arise, $n \geq 7$ tasks are required.

3 Complexity of c/nc-difference

In this section we demonstrate that determining if a c/nc-difference exists is **NP**-complete. Informally speaking, given an instance of our scheduling problem checking if loosening or tightening processor-assignment rule results in a shorter/longer schedule, is computationally hard. More formally the c/nc-difference problem may be defined as follows:

C/NC-DIFFERENCE
Instance: Processor set $\mathcal{P}$, set $\mathcal{T}$ of parallel tasks.
Question: Does $C^c_{max} = C^{nc}_{max}$?

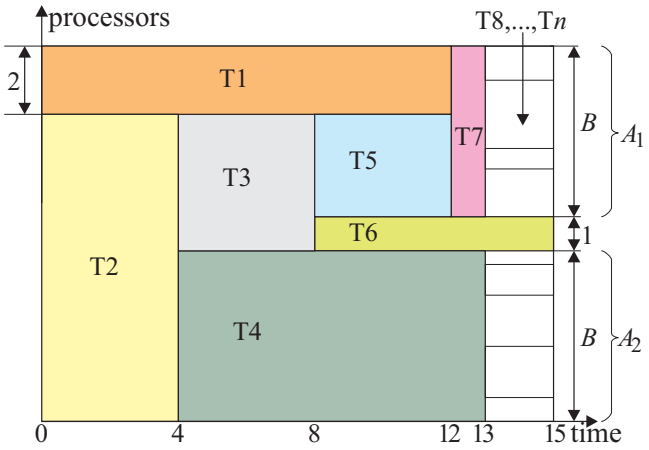

Fig. 5 Schedule for the proof of Theorem 5.

Theorem 5 *C/nc-difference is* **NP***-complete.*

Proof. C/nc-difference is in **NP** because NDTM may guess the contiguous and non-contiguous schedules and compare their lengths. However, the sizes of the strings encoding the schedules must be polynomial in the size of the input string. In the optimum schedules each task starts either at time zero, or at the end of some other task. For a contiguous schedule it is enough to determine a permutation of the tasks and the lowest processor index used by each task. The string encoding a contiguous schedule has length $O(n \log m)$. In a non-contiguous schedule a task may be assigned to arbitrary processors available at its start time. Hence, a string of length $O(n)$ encoding a task permutation is sufficient to determine a non-contiguous schedule. Thus, contiguous and non-contiguous schedules can be encoded in strings of polynomial size in the size of the input. We will use polynomial transformation from the partition problem defined as follows:

PARTITION
Instance: Set $A = \{a_1, \ldots, a_k\}$ of integers, such that $\sum_{j=1}^{k} a_j = 2B$.
Question: Is it possible to partition A into disjointed subsets A_1, A_2 such that $\sum_{i \in A_1} a_i = \sum_{i \in A_2} a_i = B$?

The transformation from partition to c/nc-difference is defined as follows: $m = 2B + 1, n = 7 + k$. The tasks are defined in Table 1.

If the answer to the partition problem is positive, then a contiguous schedule of length $C^c_{max} = 15$ as shown in Fig.5 can be constructed. Note that tasks $T_8, \ldots, T_n$ can be scheduled contiguously in interval [13,15] either on processors $P_1, \ldots, P_B$ or $P_{B+1}, \ldots, P_{2B+1}$ because the answer to partition problem is positive. It is also a feasible non-contiguous schedule. The schedules are optimum because they have no idle time. Hence,

Table 1 Task set for the proof of Theorem 5

task	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_{7+j}, $j = 1, \ldots, k$
$size_j$	2	$2B-1$	$B-1$	B	$B-2$	1	B	a_j
p_j	12	4	4	9	4	7	1	2

$C^c_{max} = C^{nc}_{max}$ and the answer to c/nc-difference is also positive.

Suppose now that the answer to c/nc-difference is positive, i.e. $C^{nc}_{max} = C^c_{max}$. We will show that the answer to partition problem must be also positive. An optimum non-contiguous schedule of length $C^{nc}_{max} = 15$ is shown in Fig.5. The position of task T_6 in the schedule is crucial for the proof because T_6 divides the range of available processors into two equal parts. We will ascertain that the pattern depicted in Fig.5 is necessary for $C^c_{max} = 15$. Let us make some observations.

• T_1 cannot be scheduled sequentially (i.e. one after another) with any of tasks $T_2, \ldots, T_6$, otherwise $C^c_{max} > 15 = C^{nc}_{max}$. Hence, T_1 and $T_2, \ldots, T_6$ must be executed at least partially in parallel.

• T_2 must be executed sequentially with T_3, T_4, T_5, T_7, otherwise more than m processors would run in parallel. Since T_1 and T_2 must run in parallel and T_6 can't be run sequentially with T_1 then also T_6 with T_2 must be executed sequentially.

• Considering sequential execution of T_2, T_4, task T_4 can't be scheduled sequentially with any of tasks T_3, T_5, T_6 (otherwise $C^c_{max} > 15 = C^{nc}_{max}$). Hence, T_4 must be executed at least partially in parallel with each of tasks T_3, T_5, T_6.

• Since T_1 must run in parallel with $T_3, \ldots, T_6$ and T_4 in parallel with T_3, T_5, T_6, then T_3 and pair T_5, T_6 must be executed sequentially (otherwise more than m processors would run in parallel).

• Consequently, T_2, T_3, T_6 are executed sequentially and T_2 cannot be the second in the sequence because T_4 would have to be preempted or $C^c_{max} > 15 = C^{nc}_{max}$.

• Since the non-contiguous schedule has no idle time, then also the contiguous schedule of length $C^c_{max} = C^{nc}_{max}$ has no idle time.

• T_1 must be scheduled sequentially with T_7 because $p_1 = 12$, $C^c_{max} = 15$, and all p_j except p_4, p_6, p_7 are even (otherwise there is an idle time on the processors running T_1, moreover T_1, T_4, T_6 must run in parallel).

• Since T_1, T_7 are scheduled sequentially, task T_6 cannot be the second in the sequence of T_2, T_3, T_6, otherwise more than m processors would be used in parallel with T_7.

• Thus, only two sequences are feasible: either (T_2, T_3, T_6), or (T_6, T_3, T_2).

• Assuming sequence (T_2, T_3, T_6), task T_7 must be executed after T_1. Consequently, T_7 runs in parallel with T_6. As T_6 runs in parallel also with T_4, task T_6 must be executed on processor P_{B+1}, otherwise some tasks would be scheduled non-contiguously.

• This creates a box of 11 time units and B processor wide for T_4 after T_2. There must be some subset of tasks from $T_8, \ldots, T_n$ in this box (otherwise there is an idle time on the processors running T_4).

• Since the schedule is nonpreemptive, contiguous, without migration and idle time, the tasks selected from $T_8, \ldots, T_n$, executed in the box require simultaneously exactly B processors. Thus, the answer to partition problem must be also positive.

• Sequence (T_6, T_3, T_2) results in a mirror symmetry of the schedule and can be handled analogously. □

It follows from the above theorem that it is hard to decide whether $C^c_{max}/C^{nc}_{max} < 1 + 1/15$ (Garey and Johnson 1979).

4 The ratio of c/nc schedule lengths

In this section we study bounds on the worst case ratio of the non-contiguous and contiguous schedules for a given instance.

Theorem 6 $5/4 \leq \max\{C^c_{max}/C^{nc}_{max}\} \leq 2$.

Proof. The instance shown in Fig.3 demonstrates that the ratio of c/nc-different schedules is $C^c_{max}/C^{nc}_{max} \geq 5/4$. In the following we show that it is also bounded from above. The proof is constructive in nature. An algorithm converting any non-contiguous schedule S of length C^{nc}_{max} into a contiguous schedule of length at most $2C^{nc}_{max}$ will be presented. The main idea is to shift parts of the non-contiguous schedule aside so that the tasks in any part can be rearranged on the processors into a contiguous allocation. Further discussion will be illustrated with figures in which the parallel tasks are represented in a simplified form as just segments in time (cf. Fig.6c).

In any non-contiguous schedule S it is possible to identify intervals when the set of executed tasks is constant. Let there be l such intervals. We will denote by A_i for $i = 1, \ldots, l$ the set of tasks executed in parallel in interval i. For example, in Fig.6a: $A_1 = \{T_3, T_4, T_5\}$, $A_2 = \{T_1, T_3, T_4, T_6, T_8\}$, $A_3 = \{T_1, T_2, T_4, T_7, T_8\}$. The tasks in A_i, if put out of schedule S, can be rearranged into contiguous allocation because in interval i they

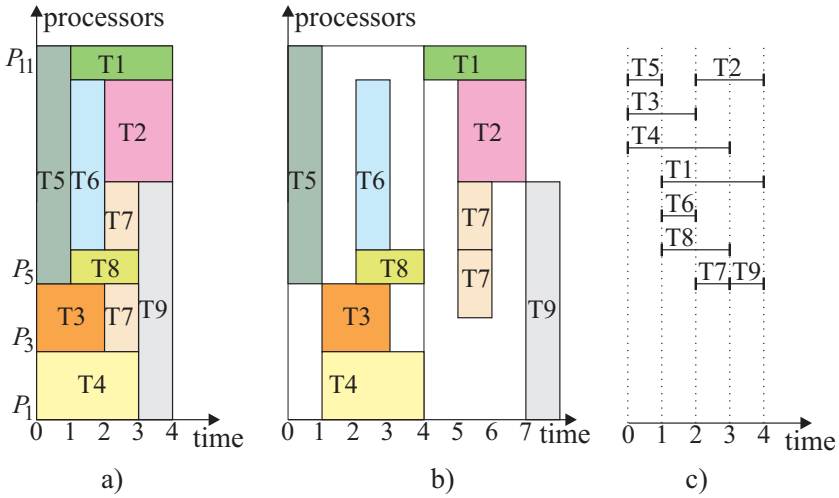

Fig. 6 Proof of Theorem 6. a) Original non-contiguous schedule. b) Partitioning into sets for contiguous rearrangement. c) Segment representation of the original schedule.

are executed in parallel and are using at most m processors. These are the tasks from interval $i-1$, and $i+1$ which may prevent rearranging the tasks in A_i into a contiguous allocation. Thus, if the tasks in A_i were moved such that they were executed neither in the same time moment as the tasks in A_{i-1}, nor in the same time moment as the tasks in A_{i+1} then A_i could be rearranged into a contiguous processor allocation. Such move should be done so that the relative positions in time of the tasks in set A_i remain fixed, i.e. the same as in the original non-contiguous schedule. Shifting the task sets aside increases schedule length. We have to show that the increase is not greater than C^{nc}_{max}. Note that some task T_j may be present in more than one time interval. If we include T_j in set A_i, then we remove T_j from all the earlier $(\ldots, A_{i-2}, A_{i-1})$ and all the later $(A_{i+1}, A_{i+2}, \ldots)$ sets (cf. Fig.6b). Consequently, it must be allowed to divide sets A_i, and shift aside subsets of A_is. Let us denote by $B_1, \ldots, B_k$ the sets of tasks into which sets $A_1, \ldots, A_l$ are repartitioned such that contiguous schedule is possible. The algorithm converting a noncontiguous schedule to a contiguous one must decide how to move tasks from sets $A_1, \ldots, A_l$ (where a task may appear several times), to sets $B_1, \ldots, B_k$ (where each task appears once). For instance, in Fig.6b: $B_1 = \{T_5\}$, $B_2 = \{T_3, T_4, T_6, T_8\}$, $B_3 = \{T_1, T_2, T_7\}$, $B_4 = \{T_9\}$. Note that there exists an order of sets $A_1 \prec \ldots \prec A_l$ imposed by their sequence in the original non-contiguous schedule. Analogously, a sequence of sets $B_1 \prec \ldots \prec B_k$ also exists. This sequence follows from the order of the original non-contiguous schedule, and the decision made by the algorithm separating tasks for the contiguous rearrangement.

Assume some partition into sets and their sequence $(B_1, \ldots, B_k)$ are given. Let us now consider the extension of the original schedule (see Fig.7a). To separate sets B_j from B_{j+1}, such that further non-contiguous rearrangement is possible, all the tasks in B_{j+1} must be delayed such that the first task in B_{j+1} starts after the last task in B_j. The cost of such an operation may be represented as an interval with two arrows shown in Fig.7a. This interval starts at the beginning of the first task in B_{j+1}, and ends with the last task in B_j. For simplicity of exposition, we will call such time shift intervals necessary to separate sets B_j, B_{j+1}, *arrow intervals*. The sum of the lengths of all arrow intervals is equal to the extension of the original non-contiguous schedule. Note that if no pair of arrow intervals overlaps, then the time cost of separating the tasks in sets $B_1, \ldots, B_k$ for contiguous rearrangement does not exceed the length of the non-contiguous schedule C^{nc}_{max}. Thus, we can prove an arithmetic hypothesis (that a schedule after the transformation is not longer than $2C^{nc}_{max}$) by a geometric argument (that arrow intervals do not overlap).

The situation that two arrow intervals overlap, for a given $(B_1, \ldots, B_k)$, will be called a *conflict*. We will propose a method of removing conflicts from $(B_1, \ldots, B_k)$ by redistributing the tasks. If a conflict arises, then it means that some original set A_i has been partitioned into more than two subsets (see Fig.7a). In more detail there are sets of tasks $B_j, \ldots, B_{j+h}$, where $h \geq 2$, which must be separated and each of them comprises tasks from some set A_i, i.e. from the same interval of the original schedule. For simplicity of presentation let us assume that $h = 2$, and exactly three sets $B_j, \ldots, B_{j+2}$ are in conflict. If $h > 2$ then the conflicts can be removed by considering groups of three consecutive sets as explained further on. When $B_j, \ldots, B_{j+2}$ are in conflict (Fig.7a), then there are two arrow intervals over-

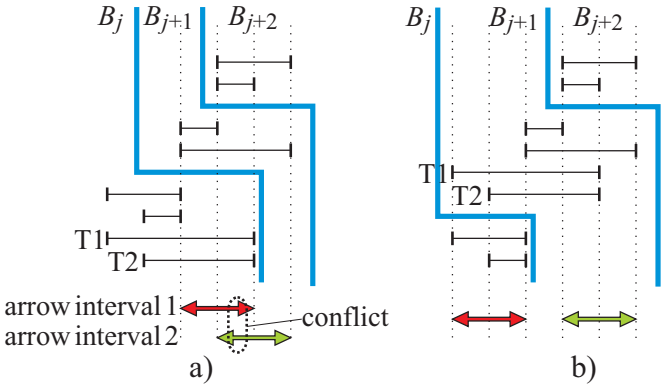

Fig. 7 Proof of Theorem 6. a) Arrow intervals for separating sets B_j, B_{j+1}, B_{j+2}. b) Resolving the conflict.

lapping: arrow interval 1 representing overlapping of the tasks from B_j and B_{j+1}, and arrow interval 2 representing overlapping of the tasks in B_{j+1} and B_{j+2}. Arrow interval 1 finishes with the end of the last task in B_j which is necessarily executed in parallel with the first task form B_{j+2}. The conflict can be removed by shifting all conflicting tasks from B_j to B_{j+1}. It means that tasks in B_j which are executed in parallel with the tasks in B_{j+2} are moved to B_{j+1}. This is feasible because we have conflict, so the tasks that cause it in B_j, are executed in parallel with some tasks in B_{j+1} and rearranging them together into a contiguous schedule is possible. Again, arrow interval 1 represents the delay of B_{j+1} necessary to allow rearrangement of B_{j+1} into a contiguous layout without interfering with B_j. Arrow interval 1 stretches from the earliest beginning of a task in B_{j+1} till the end of the last task in B_j. As after the moving of the tasks from B_j to B_{j+1}, no task remaining in B_j is executed in parallel with the tasks in B_{j+2}, the end of arrow interval 1 is moved earlier in time, and there is no more conflict with arrow interval 2 (see Fig.7b). Thus, the conflict of overlapping arrow interval 1 and arrow interval 2 is resolved.

However, after the tasks redistributing set B_{j+1} comprises more tasks, and the earliest beginning of a task in B_{j+1} may have moved earlier in time. Hence, the beginning of arrow interval 1 moved back in time, and a new conflict may have arisen between sets B_{j-1}, B_j, B_{j+1}. Such a conflict may be removed in an analogous way by moving tasks from set B_{j-1} to set B_j. This procedure should be continued until either removing the conflict or reaching the beginning of the initial non-contiguous schedule and sets B_1, B_2, B_3. The conflict cannot move any earlier in time because we already reached the beginning of the schedule. Thus, the conflict which originated in $B_j, \ldots, B_{j+2}$ is removed from $(B_1, \ldots, B_{j+1})$. By proceeding forward with increasing j, the sets of tasks can be repartitioned such that arrow intervals do not overlap. As they do not overlap, the total extension of the initial non-contiguous schedule is not greater than C^{nc}_{max}. Consequently, the contiguous schedule is not longer than $2C^{nc}_{max}$.

Overall, the algorithm converting a non-contiguous schedule into a contiguous one can be summarized as follows.

1. Scan the initial non-contiguous schedule from the beginning toward the end, and build sets B_j greedily by appending new tasks to the current set B_j as long as no task in B_j has finished. If some task in B_j has finished, close B_j and start building B_{j+1}. Thus, sets $B_1, \ldots, B_k$ are constructed.
2. Scan the sets from B_1 to B_k and verify if any conflict has arisen. If conflicts appeared, then resolve them one by one from the earliest to the latest by redistributing tasks between sets $B_1, \ldots, B_k$ as described above.
3. Shift sets $B_1, \ldots, B_k$ aside such that pairs of sets B_j, B_{j+1} are not executed in parallel.
4. Rearrange tasks in sets $B_1, \ldots, B_k$ into a contiguous assignment. □

Example. We finish this section with an example showing operation of the algorithm given in the proof of Theorem 6 on the example shown in Fig.6a. The process of repartitioning the tasks is shown in Fig.8. The initial greedy partitioning is:
$B_1 = \{T_3, T_4, T_5\}$, $B_2 = \{T_1, T_6, T_8\}$, $B_3 = \{T_2, T_7\}$, $B_4 = \{T_9\}$.
The first two arrow intervals in Fig.8a overlap which indicate that sets B_1, B_2, B_3 are in conflict because tasks $T_4 \in B_1, T_1 \in B_2, T_7 \in B_3$ are executed in parallel. It means that set $A_3 = \{T_1, T_2, T_4, T_7, T_8\}$ of tasks executed in the third time unit has been partitioned into more than two sets. According to the algorithm given in Theorem 6, task T_4 is shifted to B_2. Hence, we have $B_1 = \{T_3, T_5\}$, $B_2 = \{T_1, T_4, T_6, T_8\}$, $B_3 = \{T_2, T_7\}$, $B_4 = \{T_9\}$.
In Fig.8b the second and third arrow intervals overlap because $T_1 \in B_2, T_2 \in B_3, T_9 \in B_4$ are executed in parallel. Therefore, T_1 is moved to B_3, and we have:
$B_1 = \{T_3, T_5\}$, $B_2 = \{T_4, T_6, T_8\}$, $B_3 = \{T_1, T_2, T_7\}$, $B_4 = \{T_9\}$.
Again (Fig.8c) a conflict between B_1, B_2, B_3 arises because $T_3 \in B_1, T_4 \in B_2, T_1 \in B_3$ are executed in parallel. We move T_3 to B_2 and obtain partitioning:
$B_1 = \{T_5\}$, $B_2 = \{T_3, T_4, T_6, T_8\}$, $B_3 = \{T_1, T_2, T_7\}$, $B_4 = \{T_9\}$
for which arrow intervals do not overlap (cf. Fig.8d). The parts of the schedule shifted aside are shown in Fig.6b. It can be verified that the schedule in Fig.6b is of length $2C^{nc}_{max} = 8$ and tasks can be rearranged into contiguous processor assignment, which is necessary only in set $B_3 = \{T_1, T_2, T_7\}$.

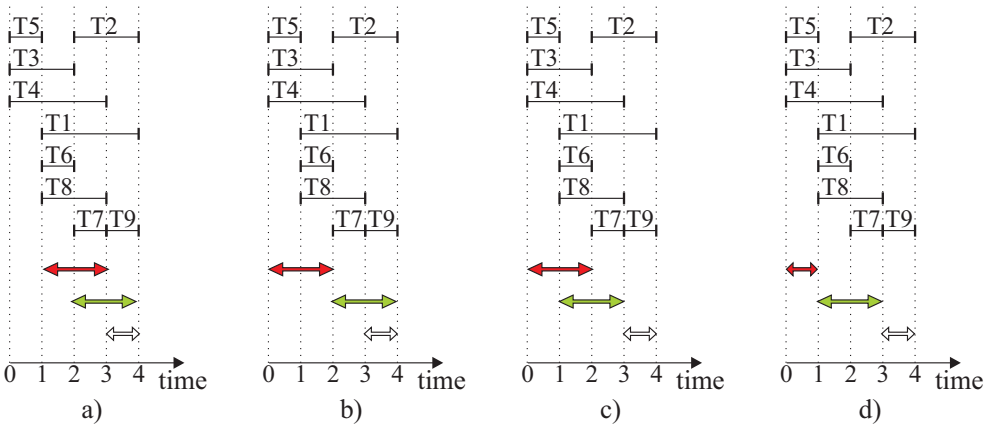

Fig. 8 Example of redistributing tasks as indicated in Theorem 6

Theorem 6 has practical consequences. If one constructs a schedule while disregarding possible non-contiguity of the assignments, then to be sure that a conversion to a contiguous schedule is always feasible a margin in time of [25%,100%] of non-contiguous schedule length should be included. However, in the simulations described in the next section no difference exceeding 25% of non-contiguous schedule length was found. Hence, we finish this section with a conjecture:

Conjecture 2 $\max\{C^c_{max}/C^{nc}_{max}\} = 5/4$.

5 Simulations

In this section we study by simulation whether c/nc-difference is a frequent phenomenon and how big is the difference between contiguous/non-contiguous schedule lengths.

Two branch and bound (B&B) algorithms were constructed to solve contiguous and non-contiguous versions of the problem. Branching schemes of both algorithms assume that a schedule consists of two parts: a part that is already constructed, and a part still remaining to be built. The branching schemes enumerate all possible completions of the existing partial schedule with the yet unscheduled tasks. Consider a partial non-contiguous schedule. The set of optimum schedules comprises active schedules, i.e. the schedules in which a task starts at time zero or at the end of some other task. To schedule task T_j feasibly $size_j$ arbitrary processors must be available. Hence, given some partial schedule it is enough to know the earliest moment s_j when $size_j$ processors are available to determine starting time of T_j. If scheduling of T_j creates an idle interval before s_j on some processor(s) which could have been exploited by some other task T_i, then a schedule in which T_i is using this interval is constructed by considering T_i before T_j. Thus, in order to define a non-contiguous schedule it is enough to determine a permutation of the tasks. The branching scheme of the algorithm for contiguous schedules must determine not only the sequence of the tasks but also the processors executing a task, e.g. by defining the smallest index of a processor used by the task. However, enumeration of all processor assignments for a task is not necessary by the following observation:

Observation 7 *There are optimum contiguous schedules, such that each task is in corner-contact with the envelope of the preceding tasks.*

Consider some task T_j in a given schedule S. If $size_j$ contiguous processors are free before T_j starting time s_j, then it is possible to shift T_j to the earliest time moment when $size_j$ processors are free contiguously without increasing schedule length. Let us call such a move an *l-shift*. Suppose that more than $size_j$ processors are free contiguously at time s_j. Then it is possible to shift T_j to the set of processors with the smallest, or the biggest numbers. We will call such a transformation a *v-shift*. Consider, the area in time×processor space that can be reached by the lower-left corner of T_j by performing l- and v-shifts. The border of this area will be called *envelope* of the tasks preceding T_j. The length of schedule S does not change if we schedule T_j in a corner of the envelope. We will call such an assignment a *corner-contact* assignment of a task.

The test instances were generated in the following way. Task numbers n were increasing from $n = 5$ to $n = 11$. The processor numbers were $m \in \{10, 20, 50, 100, 200, 500\}$. For each n, m pair at least 1E4 instances were generated. Processing times of the tasks were chosen with discrete uniform distribution in range [1,100]. Widths of the tasks were chosen with discrete uniform distribution from range $[1, m-1]$. The experiments were conducted on a cluster of 15 PCs with with Intel Core 2 Quad CPU Q9650 running at 3.00GHz, with 8 GB

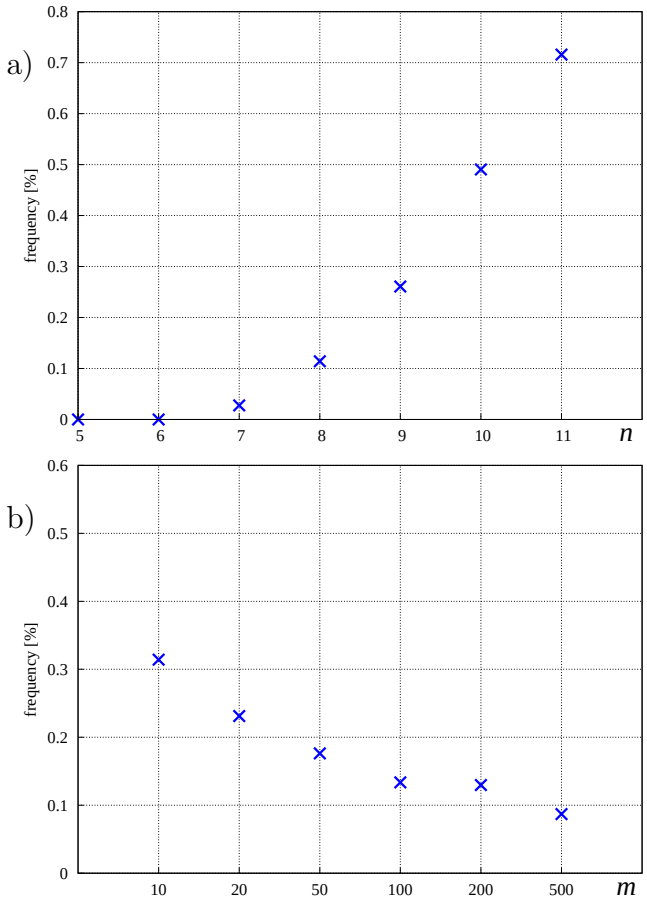

Fig. 9 Frequency in % of c/nc-differences in simulation. a) vs n b) vs m.

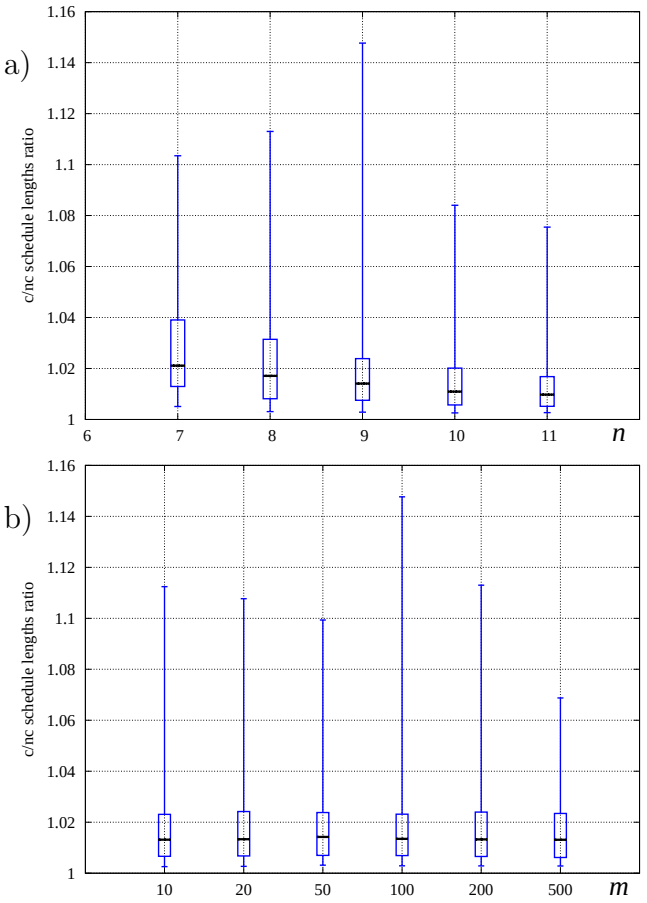

Fig. 10 Dispersion of c/nc-different schedule lengths. a) vs n, b) vs m.

RAM memory, and OpenSuSE Linux. The algorithms were implemented in Gnu C++.

The relative frequency of the c/nc-difference in the instance population is shown in Fig.9a vs task number n, and in Fig.9b vs processor number m. In Fig.9b, results for instances with $n \geq 7$ are shown for better readability. It can be verified in Fig.9 that on average the emergence of c/nc-difference is not very frequent situation. Fewer than 0.8% instances had c/nc-difference in our simulations. Our results support Conjecture 1 because no c/nc-differences were observed for $n \leq 6$. The frequency is increasing with n and decreasing with m.

The magnitude of c/nc-differences, measured as the ratio of contiguous to non-contiguous schedule lengths, is shown vs n in Fig.10a and vs m in Fig.10b. Only instances with $n \geq 7$ are depicted in Fig.10b. It can be seen that the biggest difference in the random instances is $\approx$1.15 which is far below 1.25 observed in Fig.3. Thus, the results support Conjecture 2. On average the difference between contiguous and non-contiguous schedule lengths decreases with the number of tasks n. No apparent tendency can be observed in the ratios of the c/nc-different schedule lengths vs m.

Intuitively, the tendencies in Fig.9,10 can be justified in the following way: On the one hand, the number of opportunities for creating c/nc-difference is growing with the number of the tasks. Hence, the frequency of c/nc-differences is growing. On the other hand, also the flexibility of constructing contiguous schedules is growing with the number of the tasks. Therefore, the difference in contiguous/non-contiguous schedule lengths is decreasing with n. With growing number of processors m, the relative differences between task sizes (e.g. $size_j/m - size_i/m$, for tasks T_i, T_j) has more and more possible realizations in the stochastic process defining the instances. This gives more flexibility and makes enforcing c/nc-difference more difficult with growing m. Consequently, with growing m fewer c/nc-different instances were generated.

6 Conclusions

In this paper we analyzed differences between optimum nonpreemptive contiguous and non-contiguous schedules for parallel tasks. The requirements on the smallest instances allowing the c/nc-difference were pointed out. Determining whether a c/nc-difference emerges is computationally hard. However, all non-contiguous schedules can be converted to a valid contiguous schedule twice as long as the original schedule. We leave two open questions: What is the minimum number of tasks necessary for c/nc-difference to arise (Conjecture 1)? Is it always possible to build a contiguous schedule not longer than 25% of the non-contiguous schedule for the same instance (Conjecture 2)?

Acknowledgements This research has been partially supported by a grant of Polish National Science Center, a grant of French National Research Ministry and PHC Polonium project number 28935RA.

References

Baille, F., Bampis, E., Laforest, C., Rapine, C. (2008). Bicriteria scheduling for contiguous and non contiguous parallel tasks. *Annals of Operations Research*, 159, 97–106.

Bierwirth, C., Meisel, F. (2010). A survey of berth allocation and quay crane scheduling problems in container terminals. *European Journal of Operational Research*, 202(3), 615-627.

Bokhari, S.H., Nicol, D.M. (1997). Balancing contention and synchronization on the Intel Paragon. *IEEE Concurrency*, 5(2), 74–83.

Bunde, D.P., Leung, V.J., Mache, J. (2004). Communication patterns and allocation strategies. Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04), page p.248b.

Chen, M.-S., Shin, K.G. (1987). Processor allocation in an n-cube multiprocessor using gray codes. *IEEE Transactions on Computers*, 36(12), 1396–1407.

Drozdowski, M. (2009). *Scheduling for Parallel Processing*. London: Springer-Verlag.

Dutot, P.F., Mounié, G., Trystram, D. (2004). Scheduling parallel tasks: Approximation algorithms. In J.Y.Leung (Ed.), *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, (pp.26.1–26.24). Boca Raton: CRC Press.

Feitelson, D.G., Rudolph, L., Schwiegelshohn, U., Sevcik, K., Wong, P. (1997). Theory and practice in parallel job scheduling. In D. G. Feitelson, L. Rudolph (Eds), Job Scheduling Strategies for Parallel Processing. LNCS, volume 1291 (pp. 1–34). Springer-Verlag, 1997.

Garey M.R., Johnson, D.S. (1979). *Computers and Intractability: A Guide to the Theory of NP-completeness*. San Francisco: Freeman.

Leung, V.J., Arkin, E.M., Bender, M.A., Bunde, D., Johnston, J., Lal, A., Mitchell, J.S.B., Phillips, C., Seiden, S. (2002). Processor allocation on Cplant: Achieving general processor locality using one-dimensional allocation strategies. Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'02), pp. 296–304.

Li, K., Cheng, K.-H. (1991). Job scheduling in a partitionable mesh using a two-dimensional buddy system partitioning scheme. *IEEE Transactions on Parallel and Distributed Systems* 2(4), 413–423.

Lim, A. (1998). The berth planning problem, *Operations Research Letters*, 22(2-3), 105-110.

Lloyd, E.L. (1981). Concurrent task systems. *Operations Research* 29(1), 189–201.

Lo, V., Windisch, K.J., Liu, W., Nitzberg, B. (1997). Noncontiguous processor allocation algorithms for mesh-connected multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 8(7), 712–726.

Turek, J., Wolf, J.L., Yu, P.S. (1992). Approximate algorithms for scheduling parallelizable tasks. In Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'92), pp.323–332. ACM.

Veltman, B., Lageweg, B.J., Lenstra, J.K. (1990). Multiprocessor scheduling with communications delays. *Parallel Computing*, 16:173–182.