Scheduling multiple divisible loads

M.Drozdowski, M.Lawenda, F.Guinand

Technical Report RA-007/04

Scheduling multiple divisible loads

M.Drozdowski[1], M.Lawenda[2], F.Guinand[3]

Abstract

Scheduling multiple divisible loads on a star network of processors is studied in this paper. It is shown that this problem is computationally hard. Special cases solvable in polynomial time are identified.

Keywords: divisible loads, scheduling, computational complexity.

1 Introduction

Divisible loads are computations that can be divided into parts of arbitrary sizes and the parts can be processed independently in parallel. Divisible load theory (DLT) emerged as a new paradigm in parallel processing which links scheduling, communication optimization, and performance modeling. Surveys of DLT literature can be found in [1, 3, 8].

In this paper we consider scheduling multiple divisible loads in a star network. Each load, which represents a separate parallel application, will be called a task. The set of tasks is $\mathcal{T} = \{T_1, \ldots, T_n\}$. Each task T_j is represented by the volume of load V_j that must be processed.

The tasks (loads) are to be processed on a set of distributed computers interconnected by a star network. For the simplicity of presentation we will be using name processor when referring to a computer - communication link pair. The set of processors is $\mathcal{P} = \{P_1, \ldots, P_m\}$. In the center of the star a scheduling controller (or master, or server) P_0 called *originator* is located. Tasks in $\mathcal{T}$ may be reordered by the originator to achieve good performance of the computations. Originator splits the loads of the tasks into parts and

[1]Institute of Computing Science, Poznań University of Technology, ul.Piotrowo 3A, 60-965 Poznań, Poland. This research was partially supported by a grant of Polish State Committee for the Scientific Research. Corresponding author. Email: `Maciej.Drozdowski@cs.put.poznan.pl`

[2]Poznań Supercomputing and Networking Center, ul.Noskowskiego 10, 61-794 Poznań, Poland.

[3]Laboratoire d'informatique du Havre, UFR Sciences et Techniques, Universite du Havre, 25 rue P. Lebon, BP 540, 76058 Le Havre cedex, France.

sends them to processors $P_1, \dots, P_m$ for remote processing. Only some subset $\mathcal{P}_j \subseteq \mathcal{P}$ of all processors may be used to process task T_j. We will denote by α_{ij} the size of task T_j part sent to processor P_i. α_{ij} are expressed in load units (e.g. bytes). $\alpha_{ij} = 0$ implies that $P_i \notin \mathcal{P}_j$. The sizes of load parts sum up to the task load, i.e. $\sum_{i=1}^{m} \alpha_{ij} = V_j$. Not only $\mathcal{P}_j$ is selected by the originator, but also the sequence of activating the processors in $\mathcal{P}_j$ is chosen by the originator. In star topology processors $P_1, \dots, P_m$ communicate only with the originator P_0. Originator is not computing. Were it otherwise, the computing capability of the originator can be represented as an additional processor.

Each processor, is described by three parameters: computing rate, communication rate of the link to the originator, communication startup time. Computing and communication rates are expressed in time units per load unit (e.g. seconds per bytes), and are reciprocals of speeds. Startup time is expressed in time units. Depending on the heterogeneity of the computing environment, three forms of the star system can be distinguished (we use scheduling theory naming convention [2, 7]):

Unrelated processors – communication rates and startup times are specific for the communication link and for the task. Similarly, processor computing rates depend on the processor and task. We will denote by C_{ij} communication rate, and by S_{ij} the startup time, of the link to processor P_i perceived by task T_j. Transferring α_{ij} load units to P_i takes $S_{ij} + C_{ij}\alpha_{ij}$ time units. A_{ij} will denote the processing rate of processor P_i perceived by task T_j. Computing for load α_{ij} lasts $A_{ij}\alpha_{ij}$. The case of unrelated processors is the most general one. Both the processors, and the tasks are different due to the differences in the problems being solved, and the computer or network architecture,

Uniform processors – communication rates C_i, startup times S_i, and computing rates A_i are specific for the processors but are the same for all tasks. In other words $\forall_{T_j} A_{ij} = A_i, C_{ij} = C_i, S_{ij} = S_i$, for $P_i \in \mathcal{P}$. The class of uniform processors is a special case of the more general class of unrelated processors. Uniform processors represent identical, or similar, parallel programs executed on heterogeneous system.

Identical processors – communication rates, startup times, and computing rates are the same for all processors and tasks. Thus, $\forall_{P_i \in \mathcal{P}} A_i = A, C_i = C, S_i = S$. Thus, identical processors are further specialization of the uniform processors. Identical processors represent, e.g., the same parallel program executed in homogeneous environment for different input data.

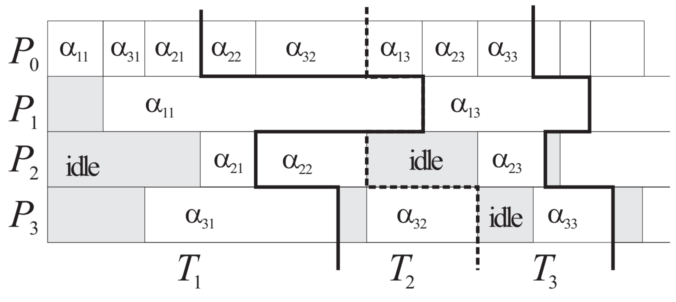
We assume that processors have sufficient memory buffers and computations do not have to start immediately after receiving the load. Note that even for uniform and identical processors n tasks are not equivalent to a single task with load $\sum_{j=1}^{n} V_j$ because each task is a separate scheduling entity and requires a separate set of communications.

By constructing a schedule the originator decides on: the sequence of the tasks, the sets of processors assigned to the tasks, the sequence of processor activation, and the load parts sizes. Let us now point out several possible assumptions on the structure of the schedule.

In some cases the time of returning the results may be so short in comparison with the load scattering and computing phases, that the result returning may be neglected in the construction of the schedule. This assumption is commonly used in modeling divisible load computations [1, 3, 8]. It has been observed in the earlier DLT papers that if the result returning time may be neglected, then the schedule for a single task is the shortest when all the processors complete computations at the same moment. This requirement may be extended to the multiple loads case. We will say that a schedule has *simultaneous completion* property if the computations on all parts of each task finish simultaneously. Simultaneous completion of the computations may be also justified by technological reasons: When a parallel application finishes at the same time on all processors, then managing it in a parallel computer batch system is simpler than if it were finished on different processors in different moments of time.

It is assumed in this paper that the originator constructs *permutation schedules* (see e.g. [2, 7] for classic definition). We mean by permutation schedule that a task is sent to the processors only *once*, and the sequence of the tasks is the same on all processors. Consequently, communications and computations are nonpreemptive, i.e. cannot be suspended and restarted later. If $P_i \notin \mathcal{P}_j$ and $\alpha_{ij} = 0$, then a dummy computation interval of length 0 is inserted on P_j. An example of permutation schedule is shown in Fig.1.

On the other hand, the process of result returning may be equally time consuming as load distribution and computations. In such cases we will assume that the amount of returned results is $\beta_j \alpha_{ij}$, which means that the volume of results is proportional to the amount of received load, and coefficient β_j is application specific. The result returning phase must be explicitly scheduled. Also in this case we will consider permutation schedules, by which we mean that the order of the tasks in distribution, computation, and result collection phases is the same. We assume that transfer rates and startup times are the same for sending the load to the processors, and for returning



P_0
P_1
P_2
P_3
α_{11} α_{31} α_{21} α_{22} α_{32} α_{13} α_{23} α_{33}
α_{11} α_{13}
idle α_{21} α_{22} idle α_{23}
α_{31} α_{32} idle α_{33}
T_1 T_2 T_3

Figure 1: An example of a permutation schedule.

of the results.

Our objective is minimization of the schedule length, denoted by C_{max}.

Scheduling multiple divisible loads has already been considered in DLT for communications without startup times. In [1, 9] it was assumed that task execution sequence was first-in first-out, processors were uniform, and task computations finish simultaneously. Furthermore, all processors were used by each task. In a multi-job scheme [1, 9] communications of some task T_j overlap with computations of task T_{j-1} preceding T_j in the execution sequence. This allows to start computations for T_j on processors $P_1, \dots, P_{m'}$ immediately after the end of task T_{j-1}. Processors $P_{m'+1}, \dots, P_m$ are idle until receiving their load share of T_j. Using the formulae provided in [1, 9] the distribution of the load for T_j can be found in $O(m)$ time, for a given m'. The actual value of m' can be found iteratively in at most m steps. Thus, for a sequence of n tasks the complexity of the algorithm is $O(m^2n)$.

In [10] the same assumptions on the task sequence, processor selection, simultaneous computation completion, and zero startup time were made. Under the above assumptions a multiinstallment load distribution strategy has been proposed to ensure that all processors work continuously on tasks $T_2, \dots, T_n$. When the overlap of computations on T_{j-1} with the communications of T_j is too short to send the whole load V_j to the processors, and thus avoid idle time (i.e. if $m' < m$), then the load is divided into multiple smaller installments. Since communications last shorter, all processors may receive some load earlier, and may work continuously on T_j. Unfortunately, it was observed in [10] that this strategy does not work for certain combinations of task, and processor parameters. Furthermore four heuristics have been proposed in [10]. It was demonstrated by a set of simulations that for processing multiple loads multiinstallment strategy gives the shortest schedule in most of the cases.

In [6] a probabilistic analysis is given for multiple loads arriving at multiple nodes of a fully-connected network of identical processors.

In this work we analyze multpile divisible load scheduling problem along the lines of computational complexity theory. Further organization of this paper is the following. In Section 2 computationally hard cases are identified. In Section 3 some polynomially solvable cases of the problem are presented. Bounds on the quality of approximation algorithms are given in Section 4.

2 Complexity

In this section we identify several cases of multiple divisible load scheduling problem which are computationally hard (strictly saying **NP**-hard, or **NP**-hard in a strong sense [4]). In our proofs of the computational complexity we will be using an **NP**-complete Partition problem, and strongly **NP**-complete 3-Partition problem, defined as follows [4]:

Partition
Instance: A finite set $E = \{e_1, \dots, e_q\}$ of positive integers.
Question: Is there a subset $E' \subseteq E$ such that

$$\sum_{j \in E'} e_j = \sum_{j \in E-E'} e_j = \frac{1}{2}\sum_{j=1}^{q} e_j = F? \tag{1}$$

3-Partition
Instance: A finite set $E = \{e_1, \dots, e_{3q}\}$ of positive integers, such that $\sum_{j=1}^{3q} e_j = Fq$ and $F/4 < e_j < F/2$ for $j = 1, \dots, 3q$.
Question: Can E be partitioned into q disjoint subsets $E_1, \dots, E_q$ such that $\sum_{e_j \in E_i} e_j = F$ for $i = 1, \dots, q$?

Theorem 1 *Multiple divisible load scheduling problem is* **NP***-hard even for one* $(m = 1)$ *unrelated processor, when result returning is considered.*

Proof. For $m = 1$ multiple divisible load scheduling problem is obviously in **NP** because NDTM has to guess the sequence of tasks execution. We will show that multiple divisible load scheduling problem is **NP**-hard by a polynomial time transformation from Partition, defined as follows:
$n = q + 1$,

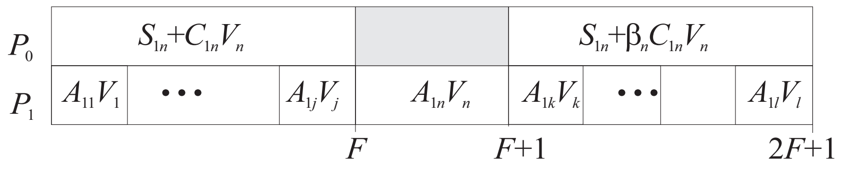


Figure 2: Illustration to the proof of Theorem 1.

$V_j = 1, \beta_j = 1$ for $j = 1, \ldots, n$,
$S_{1j} = 0$ for $j = 1, \ldots, n$,
$C_{1j} = 0$ for $j = 1, \ldots, q$, $C_{1n} = F$,
$A_{1j} = e_j$ for $j = 1, \ldots, q$, $A_{1n} = 1$.
We ask if a schedule with length at most $y = 2F + 1$ exists. Suppose, that the PARTITION instance has a positive answer. Then a feasible schedule of length $2F + 1$ can be constructed as shown in Fig.2.

Suppose the scheduling problem instance has a positive answer. Then task T_n is continuously performed because $S_{1n} + V_nC_{1n} + V_nA_{1n} + S_{1n} + \beta_nV_nC_{1n} = 2F + 1 = y$. As computations are nonpreemptive, tasks $T_1, \ldots, T_q$ must fit either into interval $[0, F]$, or interval $[F + 1, 2F + 1]$. For the set of tasks $\mathcal{T}_{[0,F]}$ which computations are performed in $[0, F]$ we have $\sum_{T_j\in\mathcal{T}_{[0,F]}} A_{1j}V_j = \sum_{T_j\in\mathcal{T}_{[0,F]}} e_j = F$. Analogously, for the tasks in interval $[F + 1, 2F + 1]$: $\sum_{T_j\in\mathcal{T}_{[F+1,2F+1]}} A_{1j}V_j = \sum_{T_j\in\mathcal{T}_{[F+1,2F+1]}} e_j = F$. Thus, a PARTITION instance also has a positive answer. Consequently, our scheduling problem is **NP**-hard. □

Theorem 2 *If result returning time is negligible, then multiple divisible load scheduling problem for two ($m = 2$) unrelated processors is* **NP***-hard in the strong sense.*

Proof. We prove the theorem by reduction from 3-PARTITION. We assume (without loss of generality) that $F > q, F > 1$. Were it otherwise, e_j can be multiplied by $q > 1$ to fulfil this requirement. The instance of the scheduling problem can be constructed as follows:
$n = 4q + 1$, $V_j = 1$ for $j = 1, \ldots, n$,
$S_{1j} = \infty, S_{2j} = 0, C_{1j} = \infty, C_{2j} = e_j, A_{1j} = \infty, A_{2j} = F^3e_j$ for $j = 1, \ldots, 3q$.
$S_{1,3q+1} = 0, S_{2,3q+1} = \infty, C_{1,3q+1} = 1, C_{2,3q+1} = \infty,$
$A_{1,3q+1} = F^4 + F, A_{2,3q+1} = \infty,$
$S_{1j} = 0, S_{2j} = \infty, C_{1j} = F^4, C_{2j} = \infty, A_{1j} = F^4 + F, A_{2j} = \infty$ for $j = 3q + 2, \ldots, 4q,$

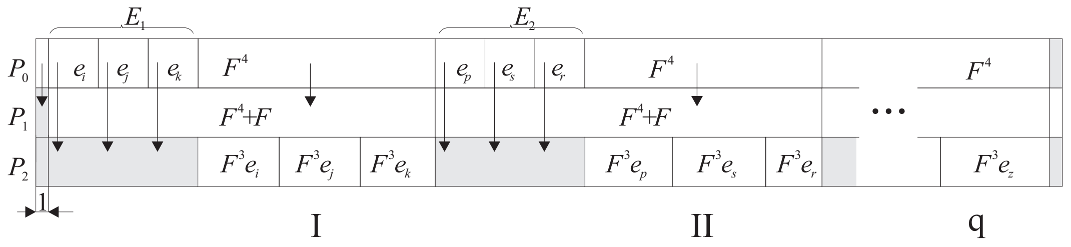


Figure 3: Illustration to the proof of Theorem 2.

$S_{1,4q+1} = \infty, S_{2,4q+1} = 0, C_{1,4q+1} = \infty, C_{2,4q+1} = F^4, A_{1,4q+1} = \infty, A_{2,4q+1} = 1,$
$y = q(F^4 + F) + 2.$

We ask whether a schedule not longer than y exists. If 3-PARTITION instance has positive answer then a feasible schedule of length y may look like the one in Fig.3. Observe that P_2 can start processing tasks immediately after its first communication. Thus, there can be also other schedules not longer than y when a 3-PARTITION exists.

Suppose, a feasible schedule not longer than y exists. Due to the values of parameters A_{ij}, C_{ij}, S_{ij}, tasks $T_1, \dots, T_{3q}$ can be executed on P_2 only, and tasks $T_{3q+1}, \dots, T_{4q+1}$ on P_1 only. The total time of computing on P_1 is $q(F^4 + F) + 1 = y - 1$, while the shortest load distribution operation last one unit of time. As a result, P_1 must compute all the time with the exception of the first time unit when the load of T_{3q+1} is sent. The sum of all communication times is equal to $y - 1$. Thus, originator must communicate all the time with the exception of the last time unit when task T_{4q+1} must be executed.

Total computing requirement put on P_2 by tasks $T_1, \dots, T_{3q}$ is qF^4. After excluding the first communication of T_{3q+1}, P_2 can be idle at most $qF + 1$ time units. To avoid idling on P_1, sending the load for the second task executed on P_1 must start at time $F + 1$ at the latest. Therefore, no more load can be sent to P_2 than for three tasks. Suppose that two tasks T_i, T_j are started on P_2 before sending the load for the second task on P_1, and T_i is started first. Then, there would be excessive idle time on P_2 since the end of T_j computations till the end of the communication operation of the second task executed by P_1. $F^4 + e_j$ is the span of the interval since the end of T_i communication operation till the end of the communication operation of the second task executed by P_1. $F^3(e_i + e_j)$ is the time of computing operations which can be executed on P_2 in this interval. The idle time on P_2 would be at least $F^4 + e_j - F^3(e_i + e_j)$. Since $F > q$ and $F > 1$ we have

$F^4 + e_j - F^3(e_i + e_j) > F^4 - F^3(F-1) = F^3 \geq F^2 + F^2 > qF + 1$, while the idle time on P_2 cannot be greater than $qF + 1$. Hence, exactly three communications to P_2 must be done before sending the second task to P_1.

The sum of computation times of the three tasks allocated to P_2 must be equal to F^4. If it is less, then it is at most $F^4 - F^3$ which results in $F^3 > qF+1$ idle time on P_2 while communication of the second task allocated to P_1 with the originator. Suppose it is more, then sending their loads last longer than F and the reading operation of the second task allocated to P_1 cannot start in time, which results in additional idle time on P_1. Consequently, schedule of length y cannot exist. We conclude that the three tasks must be processed in exactly F^4 time units.

The same reasoning can be applied to the following tasks assigned to P_1. The load distribution operations of these tasks cannot be started later than by $1 + iF^4 + (i+1)F$ for $i = 1, \ldots, q-1$. This creates free time interval for at most three communications of the tasks assigned to P_2. Also no less than three tasks can be started by the originator, otherwise there will be excessive idle time on P_2 during the next load sending operation of a task assigned to P_1. The processing times of the three tasks must be exactly equal to F^4, otherwise either P_2 or the originator must be idle. We conclude that for each triplet of tasks T_i, T_j, T_k assigned to P_2 the processing time satisfies $F^3(e_i + e_j + e_k) = F^4$. Hence, 3-PARTITION instance also has a positive answer. $\Box$

In the following theorem we consider a simpler case of uniform processors, but with simultaneous completion required, i.e. each task must be finished at the same time on all used processors.

Theorem 3 *If result returning time is negligible and simultaneous completion is required, then multiple divisible load scheduling on uniform processors is* **NP***-hard already for two* ($n = 2$) *tasks, even if the sequence of the tasks is known.*

Proof. First we will calculate the amount of a single application load that can be distributed, and processed on a star network with $C_i = 0$, until time τ. Without loss of generality, let us assume that the sequence of processor activation is $P_1, \ldots, P_m$. The amount of load V that can be distributed, and processed in time τ is

$$V = \sum_{i=1}^{m} \frac{\tau}{A_i} - \sum_{i=1}^{m} \sum_{j=i}^{m} \frac{S_i}{A_j} \tag{2}$$

Term $\sum_{i=1}^{m} \frac{\tau}{A_i}$ is the amount of load that could be processed if all processors were activated simultaneously at time 0. Startup time S_i of the selected processor P_i delays the activation of all processors P_j for $j \geq i$. Therefore, S_i decreases the total load that could be processed by $\sum_{j=i}^{m} \frac{S_i}{A_j}$. Term $\sum_{i=1}^{m} \sum_{j=i}^{m} \frac{S_i}{A_j}$ in (2) is the amount of the load that could not be processed due to the communication delays. Suppose that $\frac{1}{A_i} = S_i$ for all i. Formula (2) reduces to

$$V = \tau \sum_{i=1}^{m} S_i - \sum_{i=1}^{m} \sum_{j=i}^{m} S_i S_j = \tau \sum_{i=1}^{m} S_i - \frac{1}{2}(\sum_{i=1}^{m} S_i)^2 - \frac{1}{2} \sum_{i=1}^{m} S_i^2 \qquad (3)$$

Note that V in (3) does not depend on the sequence of processor activation.

We will show **NP**-hardness of the problem by a polynomial time transformation of PARTITION problem. Assume that $e_i > 2$ for $i = 1, \dots, q$. Were it otherwise, all e_i may be multiplied by 2 without changing the answer to the PARTITION instance. The transformation of a PARTITION instance to a scheduling problem instance is as follows:
$n = 2, m = q$,
$S_i = e_i, A_i = \frac{1}{S_i} = \frac{1}{e_i}, C_i = 0$, for $i = 1, \dots, q$
$V_1 = 4F^2 - \frac{1}{2} \sum_{i=1}^{m} e_i^2, V_2 = F$
$y = 3F + 1$.
As already mentioned the sequence of task execution is given: T_1 precedes T_2. We ask if a schedule of length at most y exists.

Suppose the answer is positive for PARTITION problem. A feasible schedule for the instance of the scheduling problem is shown in Fig.4. Processors corresponding to set E' in PARTITION are used by T_2. Let us check that the schedule is feasible. T_1 completes computations at time $\tau = 3F$. If we supply the values of startup times S_i, and processing rates A_i into equation (2), we get equation (3), and further $3F \sum_{i=1}^{m} e_i - \frac{1}{2}(\sum_{i=1}^{m} e_i)^2 - \frac{1}{2} \sum_{i=1}^{m} e_i^2 = 6F^2 - \frac{1}{2}(2F)^2 - \frac{1}{2} \sum_{i=1}^{m} e_i^2 = V_1$. Thus, T_1 is executed feasibly. The communications of T_1 finish at time $2F$, therefore communications of T_1 which take $\sum_{i \in E'} S_i = F$ fit in $F + 1$ time units of available time. In the last time unit of interval $[\tau, y]$ the selected processors process $\sum_{i \in E'} \frac{1}{A_i} = \sum_{i \in E'} e_i = F$ units of load. Hence, also T_2 is executed feasibly.

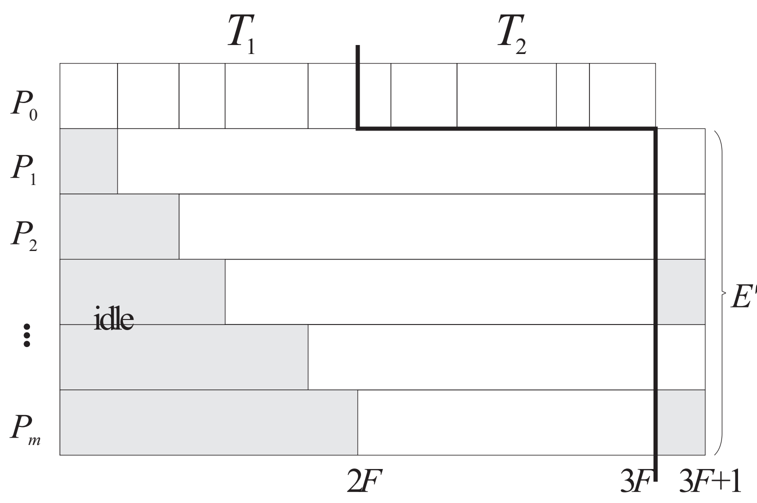


Figure 4: Illustration to the proof of Theorem 3.

Suppose that a schedule of length y exists. Task T_1 is executed first. All m processors must be used by T_1. Suppose it is otherwise, and some processor is not exploited. Without loss of generality we can renumber the processors such that P_m is the unused processors. By (3) the volume of the processed load for T_1 is at most $V_1' = y\sum_{i=1}^{m-1} S_i - \frac{1}{2}(\sum_{i=1}^{m-1} S_i)^2 - \frac{1}{2}\sum_{i=1}^{m-1} S_i^2 = (3F+1)\sum_{i=1}^{m-1} e_i - \frac{1}{2}(\sum_{i=1}^{m-1} e_i)^2 - \frac{1}{2}\sum_{i=1}^{m-1} e_i^2 = (3F+1)\sum_{i=1}^{m} e_i - (3F+1)e_m - \frac{1}{2}(\sum_{i=1}^{m} e_i)^2 - \frac{1}{2}\sum_{i=1}^{m} e_i^2 + \frac{1}{2}(e_m^2 + 2e_m\sum_{i=1}^{m-1} e_i) + \frac{1}{2}e_m^2 = V_1 + 2F - (3F+1)e_m + e_m^2 + e_m\sum_{i=1}^{m-1} e_i = V_1 + 2F - e_m(3F+1-\sum_{i=1}^{m} e_i) = V_1 + 2F - e_m(F+1) < V_1$ because $e_m > 2$. Hence, all m processors must be used. If all processors are used then T_1 communications complete by $2F$, and due to simultaneous completion requirement, its computations finish at time $3F$. This leaves interval $[2F, 3F+1]$ free for communications, and interval $[3F, 3F+1]$ for computations on T_2. Note that $\forall_i S_i \geq 1$, and any communication in interval $[3F, 3F+1]$ gives no contribution to the processed load of task T_2. Consequently communications of set $\mathcal{P}'$ of the processors selected for executing T_2 must satisfy $\sum_{i\in\mathcal{P}'} S_i = \sum_{i\in\mathcal{P}'} e_i \leq F$. The load of T_2 processed in interval $[3F, 3F+1]$ must satisfy $\sum_{i\in\mathcal{P}'} \frac{1}{A_i} = \sum_{i\in\mathcal{P}'} e_i \geq F$. Thus, the answer is positive for PARTITION instance if the elements corresponding to the processors in set $\mathcal{P}'$ are selected to set E'. □

The case of arbitrary processor sequence is not simpler. We explain it in the following observation.

Observation 4 *If result returning time is negligible and simultaneous completion is required, then multiple divisible load scheduling on uniform processors is* **NP***-hard even for two* ($n = 2$) *tasks, and arbitrary sequence of the tasks.*

Proof. The proof for the previous problem can be adjusted to the current situation. If the sequence of tasks is (T_2, T_1), then the length of the schedule is at least the length of the communications of T_2 plus the length of the schedule for T_1. The duration of T_1 processing is at least $3F$ (see the proof of Theorem 3). Communications of T_2 last at least $\min_{P_i \in \mathcal{P}}\{S_i\} = \min_{i \in E}\{e_i\} > 1$, and schedule length is at least $3F + 2$. Thus, only sequence (T_1, T_2) allows for a schedule of length at most $3F + 1$, the proof of Theorem 3 applies, and a schedule of length $3F + 1$ exists if and only if PARTITION exists. □

We can conclude from the above results that scheduling multiple divisible loads is computationally hard. The main source of the computational complexity are sequencing the tasks, selecting the processors to use, sequencing processor activation.

3 Polynomial cases

3.1 Fixed activation order, no result returning

When the task execution sequence, the set of used processors, and the processor activation orders are known, then the optimum distribution of the load can be found by using linear programming. Let us first study the case when simultaneous completion of the computations is not required, and results returning time can be ignored. For the sake of notation simplicity, and without loss of generality, let us assume that the order of task execution coincides with task numbers. The set of processors exploited by T_j is $\mathcal{P}_j$. The order of processor activation can be different for each task. Let the number of the ith processor activated for task T_j be given by function $f(j, i)$. The amount of load from task $j = 1, \dots, n$ sent to processor $i = 1, \dots, m$ is denoted by $\alpha_{ij} \geq 0$. The optimum distribution of the load can be found using the following linear program:
minimize C_{max}
subject to

$$\sum_{j=1}^{l-1}\sum_{i=1}^{|\mathcal{P}_j|}(S_{f(j,i)j}+\alpha_{f(j,i)j}C_{f(j,i)j})+\sum_{i=1}^{k}(S_{f(l,i)l}+\alpha_{f(l,i)l}C_{f(l,i)l})+$$
$$+\sum_{j=l}^{n}\alpha_{f(l,k)j}A_{f(l,k)j}\le C_{max}\quad l=1,\dots,n,\ k=1,\dots,|\mathcal{P}_j| \tag{4}$$
$$\sum_{i\in\mathcal{P}_j}\alpha_{ij}=V_j\quad j=1,\dots,n \tag{5}$$

In inequalities (4) term $\sum_{j=1}^{l-1}\sum_{i=1}^{|\mathcal{P}_j|}(S_{f(j,i)j}+\alpha_{f(j,i)j}C_{f(j,i)j})$ is the time of sending the load for tasks $1,\dots,l-1$. Sum $\sum_{i=1}^{k}(S_{f(l,i)l}+\alpha_{f(l,i)l}C_{f(l,i)l})$ is the time of sending the load to processors $f(l,i)$ activated as $i=1,\dots,k$ in the sequence of processors executing task l. $\sum_{j=l}^{n}\alpha_{f(l,k)j}A_{f(l,k)j}$ is the time of computing the load parts of tasks $l,\dots,n$, sent to processor $f(l,k)$, activated as k-th for task l. Thus, inequalities (4) ensure that computations complete before the end of the schedule. By constraints (5) all tasks are fully processed. Let us consider an example.

Example 1. $m=3, n=2, |\mathcal{P}_j|=m, f(j,i)=i$, for $j=1,2$, i.e., all processors are used, and the order of processor activation coincides with processor numbers for both tasks. Processors are identical: $\forall_{i,j}A_{ij}=1$, $\forall_{i,j}C_{ij}=1$, $\forall_{i,j}S_{ij}=1$. $V_1=32, V_2=2$. For these values the solution from (4)-(5) is: $\alpha_{11}=18.5, \alpha_{21}=9.75, \alpha_{31}=3.75, \alpha_{12}=2.0, \alpha_{22}=0, \alpha_{32}=0, C_{max}=40$. The two last communications of T_2 contain no load, because $\alpha_{22}=0, \alpha_{32}=0$, but still contribute startup times $S_1=S_2=1$. Thus, this is not the best solution, and processor P_3 need not be used in processing T_2. After removing P_3 from $\mathcal{P}_2$ we get from (4)-(5) the optimum solution: $\alpha_{11}\approx 18.333, \alpha_{21}\approx 9.333, \alpha_{31}\approx 4.333, \alpha_{12}\approx 1.667, \alpha_{22}\approx 0.333, C_{max}\approx 39.333$, shown in Fig.5. Exclusion of both P_3, and P_2 from processing T_2 does not reduce schedule length anymore. $\Box$

Let us observe that in the optimum schedule for Example 1 computations on T_1 do not finish on all processors at the same time. It demonstrates that simultaneous completion of the computations on all processors for all tasks is not necessary for the optimality of the solution.

Suppose that tasks are of equal size $\forall_{T_j}V_j=V$ processors are identical, and $\forall_{T_j}\mathcal{P}_j=\mathcal{P}$, i.e., are all processors are used by all tasks. We experimentally studied patterns that appear in the optimal solutions under the above

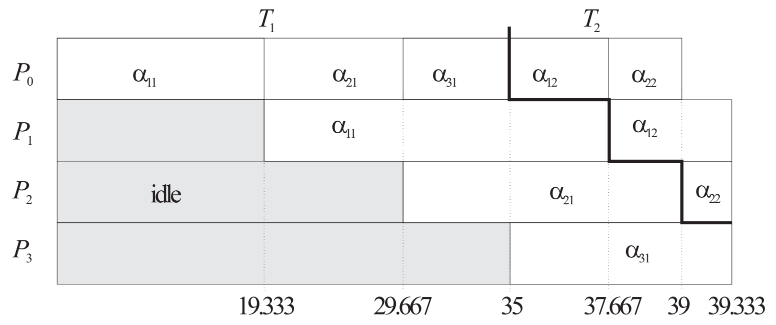


Figure 5: Optimal schedule for Example 1 (does not preserve proportion).

conditions. When communication delays are big in comparison with computing time then not all processors are exploited. It is the case when $C \gg \frac{A}{m}$. When communication delays are of similar order as computations then load of each task is distributed nearly equally between the processors. The exceptions are the leading and trailing tasks. In the leading tasks distribution is unequal so that waiting for the first load chunk to process is minimized. In the trailing tasks the distribution is also unequal such that processors stop computing at the same time. This is demonstrated in Fig.6a where changes of α_{ij} from task to task are shown. Each line in Fig.6 represents the load from the consecutive tasks assigned to a certain processor. When communication delays are short in comparison with computing times, e.g. when $C \ll \frac{A}{m}$, then total load of all tasks is distributed nearly equally between the processors, but computations of each task are concentrated on one processor. This is demonstrated in Fig.6b. Such a situation is not very comfortable for a user of a parallel application because a distribution optimal globally (for all tasks) is not a solution which is using parallelism.

It was assumed in (4)-(5) that the computation completion times are arbitrary. If simultaneous completion is required, then a linear programming formulation can be given to deal with the simultaneous completion. Let z_l denote the completion of computations on task T_l. The following linear program solves the case with simultaneous completion:
minimize C_{max}
subject to

$$z_{l-1} + \alpha_{f(l,k)l} A_{f(l,k)l} \le z_l \quad l = 2, \dots, n,\ k = 1, \dots, |\mathcal{P}_j| \tag{6}$$

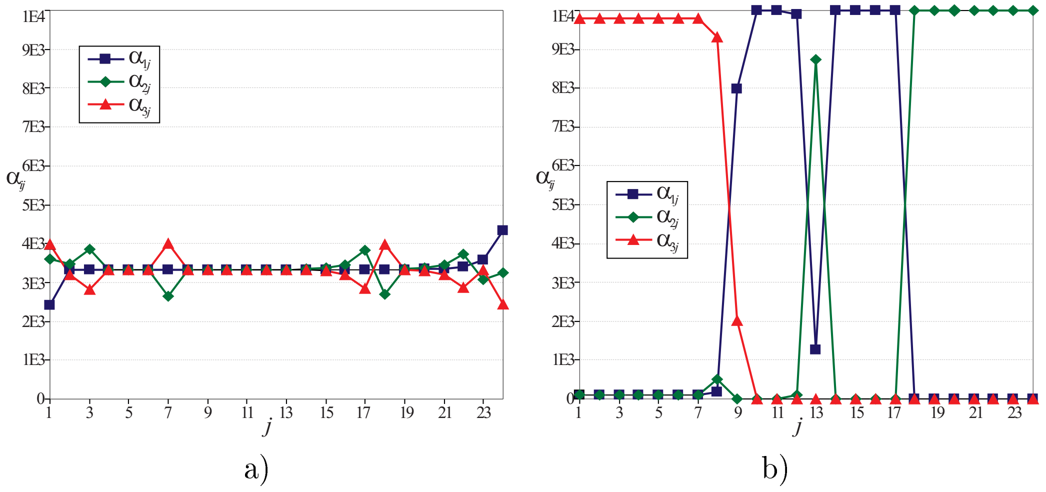


Figure 6: Distribution of the load (α_{ij}) vs task number (j); $m = 3, n = 24, V = 1E4, C = S = 1$. a) $A = 3$, b) $A = $1E2.

$$\sum_{j=1}^{l-1}\sum_{i=1}^{|\mathcal{P}_j|}(S_{f(j,i)j} + \alpha_{f(j,i)j}C_{f(j,i)j}) + \sum_{i=1}^{k}(S_{f(l,i)l} + \alpha_{f(l,i)l}C_{f(l,i)l}) +$$

$$+\sum_{j=l}^{n}\alpha_{f(l,k)j}A_{f(l,k)j} \leq z_l \quad l = 1,\ldots,n,\ k = 1,\ldots,|\mathcal{P}_j| \tag{7}$$

$$z_n = C_{max} \tag{8}$$

$$\sum_{i\in\mathcal{P}_j}\alpha_{ij} = V_j \quad j = 1,\ldots,n \tag{9}$$

By inequalities (6), the computations of task T_l can be feasibly performed in interval $[z_{l-1}, z_l]$. Inequalities (7) ensure that communications and computations of task T_l are completed by time z_l. By (8) the end of the last task is also the end of the schedule. The tasks are fully processed by (9).

Let us now return to Example 1. For linear program (6)-(9) a solution $\alpha_{11} = 19, \alpha_{21} = 9, \alpha_{31} = 4, \alpha_{12} = 1, \alpha_{22} = 1, C_{max} = 40$ is obtained, which is longer than in Fig.5. Hence, requiring simultaneous completion of computations on all processors may prevent obtaining an optimum schedule.

3.2 Fixed activation order, with result returning

The methods used in the previous section can be extended to deal with the returning of the results. Without loss of generality we assume that tasks are

executed in the order of their numbers, and such is the order of sending the loads from the originator to the processors. Yet, the set of processors used by a task, the sequence of processor activation, and the sequence of result collection can be arbitrary. Let us denote by:
$T_{j^{dd}}$ - the last task which distributes the load before task T_j distributes its load,
$T_{j^{rd}}$ - the last task which returns its results to the originator before task T_j distributes its load,
$T_{j^{dr}}$ - the last tasks which distributes the load before task T_j returns its results,
$T_{j^{rr}}$ - the last task which returns its results to the originator before returning task T_j results,
t_j^D - the time moment when distribution of task T_j load starts,
t_j^R - the time moment when collection of task T_j results starts,
t_{ij} - the time moment when P_i finishes computing load α_{ij},
$f(j,i)$ - the number of the ith processor activated for task T_j,
$g(j,i)$ - the processor returning results as ith in the sequence, for task T_j.

Optimum distribution of the load can be found by solving the following linear program:
minimize C_{max}
subject to

$$t_j^D \geq t_{j^{dd}}^D + \sum_{i=1}^{|\mathcal{P}_{j^{dd}}|} (S_{f(j^{dd},i)j^{dd}} + \alpha_{f(j^{dd},i)j^{dd}} C_{f(j^{dd},i)j^{dd}}) \quad j = 1,\dots,n, \tag{10}$$

$$t_j^D \geq t_{g(j^{rd},k)j^{rd}} + \sum_{i=k}^{|\mathcal{P}_{j^{rd}}|} (S_{g(j^{rd},i)j^{rd}} + \beta\alpha_{g(j^{rd},i)j^{rd}} C_{g(j^{rd},i)j^{rd}}) \quad j = 1,\dots,n,\ k = 1,\dots,|\mathcal{P}_{j^{rd}}| \tag{11}$$

$$t_j^R \geq t_{j^{dr}}^D + \sum_{i=1}^{|\mathcal{P}_{j^{dr}}|} (S_{f(j^{dr},i)j^{dr}} + \alpha_{f(j^{dr},i)j^{dr}} C_{f(j^{dr},i)j^{dr}}) \quad j = 1,\dots,n \tag{12}$$

$$t_j^R \geq t_{g(j^{rr},k)j^{rr}} + \sum_{i=k}^{|\mathcal{P}_{j^{rr}}|} (S_{g(j^{rr},i)j^{rr}} + \beta\alpha_{g(j^{rr},i)j^{rr}} C_{g(j^{rr},i)j^{rr}}) \quad j = 1,\dots,n,\ k = 1,\dots,|\mathcal{P}_{j^{rr}}| \tag{13}$$

$$t_j^R \geq t_{g(j,1)j} \quad j = 1,\dots,n,\ k \in \mathcal{P}_l \tag{14}$$

$$t_{kj} \geq t_j^D + \sum_{i=1}^{k}(S_{f(j,i)j} + \alpha_{f(j,i)j}C_{f(j,i)j}) + \alpha_{f(j,k)j}A_{f(j,k)j}$$
$$j = 1, \dots, n,\ k \in \mathcal{P}_j \qquad (15)$$
$$t_{kj} \geq t_{kj^{dd}} + \alpha_{kj}A_{kj} \quad l = 1, \dots, n,\ k \in\ \mathcal{P}_j \cap \mathcal{P}_{j^{dd}} \qquad (16)$$
$$C_{max} \geq t_{g(n,k)n} + \sum_{i=k}^{|\mathcal{P}_n|}(S_{g(n,i)n} + \beta\alpha_{g(n,i)n}C_{g(n,i)n}) \;\; k = 1, \dots, |\mathcal{P}_n| \qquad (17)$$
$$\sum_{i\in\mathcal{P}_j} \alpha_{ij} = V_j \quad j = 1, \dots, n \qquad (18)$$

Inequalities (10), (11) guarantee that distribution of T_j load may take place only after all the preceding communication operations. Similarly, (12), (13) ensure that collection of T_j results follow after the preceding communication operations. By inequalities (14) returning of the results can start when any results are available. Computation of some part of the task T_j on processor k can finish only after receiving the load part and computing it by inequalities (15). By inequalities (16) computations exploiting the same processor do not overlap. The end of the schedule is set by the end of returning the results of the last task by inequalities (17). All the load is processed by equation (18).

3.3 Continuous computing

In this section we assume identical processors, simultaneous completion, using all processors by each task, and negligible result returning time. Moreover, it is assumed that the tasks occupy processors continuously from the start of computations on the first task, till the end of the last task. We will call this situation *continuous computing.* We are going to propose conditions under which an optimum schedule can be constructed for the continuous computing. The conditions we propose are sufficient but not necessary. This means that in the set of optimum schedules with continuous computing there is a subset satisfying our conditions. Let us start with some observations.

Observation 5 *When computing is continuous, only the sequence of the tasks decides on the optimality of the schedule.*

Proof. Since all processors are used by each task, the selection of processor set is immaterial. For each task the sequence of processor activation can be arbitrary because processors are identical, and used in the same interval

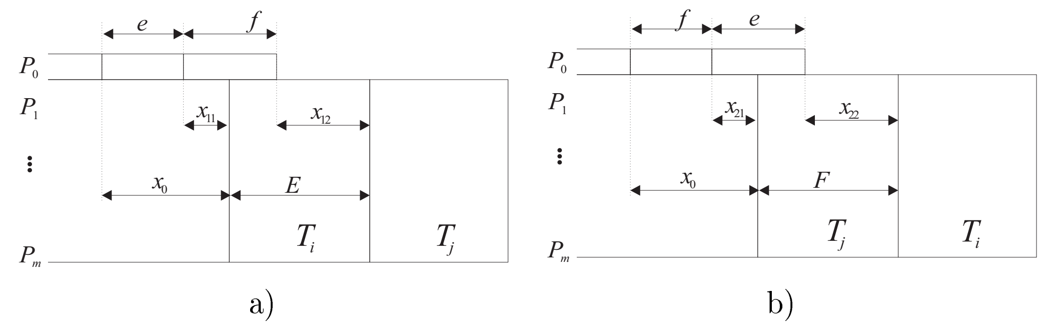


Figure 7: Illustration to the proof of Theorem 6.

due to simultaneous completion. With the exception of the first task in the sequence the load assigned to any processor is $\frac{V_j}{m}$ for task T_j, and a decision on task chunk sizes is not necessary. □

Continuous computing is possible when the load of any task T_j is distributed to the processors before starting of the computations on T_j. Executing the tasks according to the increasing sizes (V_j) will be called SPT (for Shortest Processing Time) sequence.

Theorem 6 *If computing is continuous, and* $\forall_{T_j \in \mathcal{T}} V_j > \frac{Sm}{\frac{A}{m}-C}$ *then SPT maximizes the interval between the completion of the task communication, and starting of its computations.*

Proof. The requirement $\forall_{T_j \in \mathcal{T}} V_j > \frac{Sm}{\frac{A}{m}-C}$ can be rewritten as $\forall_{T_j \in \mathcal{T}} \frac{AV_j}{m} > Sm + CV_j$, which means that load distribution time is shorter than computation using all processors in the same interval. This requirement should be satisfied by real parallel applications which have high computing demands.

Consider two tasks T_i, T_j executed continuously one after another. For the simplicity of presentation let us denote by $e = Sm + CV_i, E = \frac{AV_i}{m}, f = Sm + CV_j, F = \frac{AV_j}{m}$. Note that $e < E, f < F$. A task preceding T_i, T_j completes its communication x_0 units of time before the end of its computations (cf. Fig.7). Suppose that T_i precedes T_j (cf. Fig.7a). The time from the end of T_i communication to the start of T_i computation is $x_{11} = x_0 - e$. The length of the interval since the end of T_j communication till the beginning of T_j computation is $x_{12} = x_0 + E - e - f$. The worse of the two interval lengths is $\min\{x_{11}, x_{12}\}$. Now suppose that the order of the two tasks is inverted. Then the lengths of the intervals are (cf. Fig.7b) $x_{21} = x_0 - f, x_{22} = x_0 + F - e - f$.

The smaller of the intervals is $\min\{x_{21}, x_{22}\}$. Let us analyze the conditions under which it is better to execute the two tasks in the order (T_i, T_j), than in the order (T_j, T_i), i.e. when $\min\{x_{11}, x_{12}\} > \min\{x_{21}, x_{22}\}$. Note that changing the order of the two tasks does not influence the rest of the schedule.

Let us assume that $\min\{x_{11}, x_{12}\} = x_{11}$ then $x_0 - e < x_0 + E - e - f$, hence $f < E$. Suppose that $\min\{x_{21}, x_{22}\} = x_{21}$ then $x_0 - f < x_0 + F - e - f$, hence $e < F$. Sequence (T_i, T_j) is better when $x_{11} = x_0 - e > x_{21} = x_0 - f$ from which we get $e = Sm + V_iC < f = Sm + V_jC$, and $V_i < V_j$. Thus, SPT sequence is desired. Suppose that $\min\{x_{21}, x_{22}\} = x_{22}$ then $e > F$. Sequence (T_i, T_j) is better if $x_{11} = x_0 - e > x_{22} = x_0 + F - e - f$, and $f > F$ which is in contradiction with $f = Sm + CV_j < \frac{AV_j}{m} = F$.

Let us assume that $\min\{x_{11}, x_{12}\} = x_{12}$ then $f > E$. If $\min\{x_{21}, x_{22}\} = x_{21}$ then $e < F$. Sequence (T_i, T_j) is better when $x_{12} = x_0 + E - e - f > x_{21} = x_0 - f$, from which we get $E > e$. Altogether we have $e < E < f$, and $V_i < V_j$. Finally, if $\min\{x_{21}, x_{22}\} = x_{22}$, and $e > F$. If we put together the conditions for this case we have: $e < E, e > F, f < F, f > E$. Using $e < E < f$, and $e > F > f$ we get a contradiction. We may conclude that by using interchanges between all pairs of consecutive tasks (excluding the first task), any continuous computing sequence can be changed to a continuous computing SPT sequence. □

Let us assume that tasks are ordered according to SPT rule, i.e. $V_1 \le V_2 \le \ldots \le V_n$. The conditions of the optimality of SPT sequence in continuous computing are the following.

Theorem 7 *SPT is the optimum task sequence for continuous computing if* $\forall_{T_j \in \mathcal{T}} V_j > \frac{Sm}{\frac{A}{m} - C}$, *and* $x_j > Sm + V_{j+1}C$ *for* $j = 1, \ldots, n-1$, *where* $x_1 = \frac{CV_1 - \frac{SA}{C}[(1+\frac{C}{A})^m - (1+\frac{C}{A})] + (m-1)S}{(1+\frac{C}{A})^m - 1}$, $x_j = x_{j-1} + \frac{V_jA}{m} - Sm - V_jC$ *for* $j = 2, \ldots, n-1$.

Proof. It follows from Observation 5 that only the task sequence has to be chosen for continuous computing. According to Theorem 6 SPT sequence maximizes the distance between task communication completion and computation start. Thus if it is possible to maintain continuous computing at all, then SPT will also do it, provided that $\forall_{T_j \in \mathcal{T}} V_j > \frac{Sm}{\frac{A}{m} - C}$. Yet, it has not been determined by Theorem 6 what is the first task in the sequence (i.e. the one which precedes the first pair (T_i, T_j) satisfying SPT order). Note that the longer the first task is, the longer communication delays are, and the longer the processors must wait idle for the initiation of the computations.

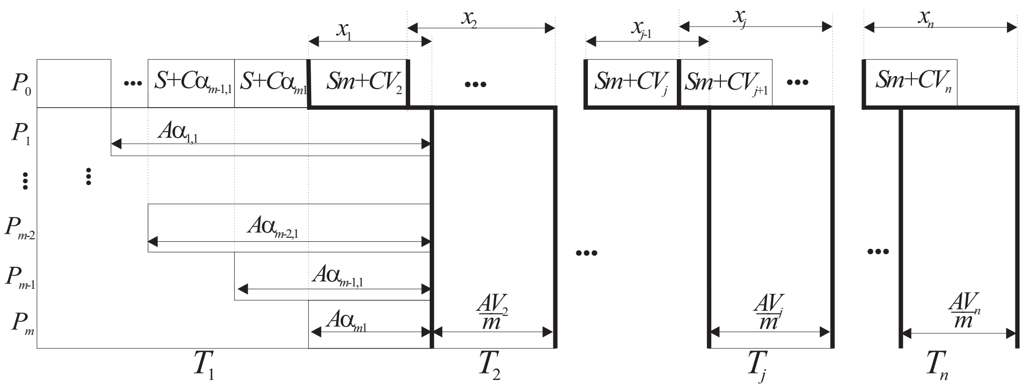


Figure 8: Illustration to the proof of Theorem 7.

If the first task in the sequence has the smallest load, then the idle area in the whole schedule is the shortest possible because there is no idle time after the first task. Hence, the SPT sequence $V_1 \leq V_2 \leq \ldots \leq V_n$ is optimal.

It still remains to ensure that continuous computing is possible. It is the case if $x_j > Sm + V_{j+1}C$, for $j = 1, \ldots, n-1$, where x_j is the time between end of task T_j communication, and the start of its computation (cf. Fig.8). It means that communication of T_{j+1} finishes before its computation has to start. The length x_j of the interval for the communication of T_{j+1} is equal to $x_j = x_{j-1} + \frac{V_j A}{m} - Sm - CV_j$ for $j = 2, \ldots, n-1$. Length x_1 of the first interval is $x_1 = A\alpha_{m1}$. Now we calculate α_{m1}. Since computations on T_1 must finish simultaneously on all processors we have

$$A\alpha_{i1} = S + \alpha_{i+1,1}(A + C) \qquad i = 1, \ldots, m-1$$

α_{i1} can be expressed as a function of α_{m1}:

$$\alpha_{i1} = \alpha_{m1}(1 + \frac{C}{A})^{m-i} + \frac{S}{A}\sum_{j=0}^{m-i-1}(1 + \frac{C}{A})^j \qquad i = 1, \ldots, m-1$$

The size of the first task is

$$V_1 = \sum_{i=1}^{m}\alpha_{m1}(1 + \frac{C}{A})^{m-i} + \frac{S}{A}\sum_{i=1}^{m-1}\sum_{j=0}^{m-i-1}(1 + \frac{C}{A})^j$$

this can be reduced to

$$V_1 = \alpha_{m1}\frac{A}{C}[(1 + \frac{C}{A})^m - 1] + \frac{SA}{C^2}[(1 + \frac{C}{A})^m - (1 + \frac{C}{A})] - \frac{(m-1)S}{C}.$$

Hence

$$x_1 = A\alpha_{m1} = \frac{V_1 - \frac{SA}{C^2}[(1+\frac{C}{A})^m - (1+\frac{C}{A})] + \frac{(m-1)S}{C}}{\frac{1}{C}[(1+\frac{C}{A})^m - 1]}.$$

□

3.4 $m = 1$

Observation 8 *If result returning time is negligible, then multiple divisible load scheduling problem for one ($m = 1$) unrelated processor is solvable in $O(n \log n)$ time by Johnson's algorithm [5].*

Proof. If the results are not returned, and only one machine ($m = 1$) is available, then execution of a task reduces to two operations: communication operation involving originator P_0, followed by computation operation involving P_1. This situation is equivalent to two-machine flowshop. Two-machine flowshop is solvable in $O(n \log n)$ time by Johnson's algorithm [5] (or see e.g. [2, 7]). □

For the completeness of the presentation let us note that in our case Johnson's algorithm divides the set of tasks into two subsets: $\mathcal{T}_1$ comprising the tasks for which $S_{1j}+C_{1j}V_{1j} < A_{1j}V_{1j}$, and set $\mathcal{T}_2$ comprising the remaining tasks. Tasks in $\mathcal{T}_1$ are executed in the order of increasing $S_{1j} + C_{1j}V_{1j}$, while tasks in $\mathcal{T}_2$ are ordered according to decreasing $A_{1j}V_{1j}$. $\mathcal{T}_1$ is executed first.

This special case can be also applied if for technical reasons, the parallel applications should start and finish on all processors simultaneously.

4 Approximability

In this section we study the bounds on the quality of approximation algorithms for multiple divisible load scheduling problem. By a greedy heuristic we mean an algorithm which is not unnecessarily delaying communications and computations. This means that if there is some load to be distributed and communication medium is available, then the load is immediately distributed, if there is some load already at a processor and the processor is free, then the computation on the load is immediately started.

Theorem 9 *Length C^H_{max} of a schedule built by any greedy heuristic H solving multiple divisible load scheduling problem on identical processors satisfies:*

$$\frac{C^H_{max}}{C^*_{max}} \le mn + m - n + 1,$$

where C^*_{max} is the optimum schedule length.

Proof. Intervals of two types can be distinguished in any schedule for our problem: Intervals of total length E_C when initiator performs communications, and intervals of total length E_A when initiator is free because all processors compute. In the case of identical processors $E_C = \sum_{j=1}^{n}(\sum_{P_i \in \mathcal{P}_j} S + CV_j)$. Note that $nS + C\sum_{j=1}^{n} V_j \le C^*_{max}$ because the loads must be sent to at least one processor. In the worst case some heuristic may try activating all processors while only a single processor is necessary for each task. Consequently, $\sum_{j=1}^{n}\sum_{P_i \in \mathcal{P}_j} S - nS = S\sum_{j=1}^{n}(|\mathcal{P}_j| - 1) \le (m-1)nC^*_{max}$. Some heuristic may also tend to use less processors than necessary. In the worst case $|\mathcal{P}_j| = 1$, and $E_A \le \sum_{j=1}^{n} AV_j$. Note that $\sum_{j=1}^{n} \frac{AV_j}{m} \le C^*_{max}$. Hence $E_A \le mC^*_{max}$. Altogether we have $C^H_{max} = E_C + E_A \le C^*_{max} + n(m-1)C^*_{max} + mC^*_{max}$, from which the theorem follows. □

The results of Theorem 9 can be further strengthened. If $S = 0$, then in the above proof $\sum_{j=1}^{n}\sum_{P_i \in \mathcal{P}_j} S - nS = 0$, and the ratio of schedule lengths can be narrowed to $\frac{C^H_{max}}{C^*_{max}} \le m + 1$. If $\forall_{T_j \in \mathcal{T}} CV_j + mS < \frac{V_j A}{m}$, i.e. when computations dominate in the parallel application, then $E_C \le \sum_{j=1}^{n}(mS + CV_j) \le \sum_{j=1}^{n} \frac{V_j A}{m} \le C^*_{max}$. Consequently $\frac{C^H_{max}}{C^*_{max}} \le m + 1$.

5 Conclusions

In this paper we studied some combinatorial aspects of scheduling multiple divisible loads. It has been demonstrated that this problem is computationally hard for dedicated processors, and uniform processors with simultaneous completion requirement. Polynomially solvable cases have been presented: when the order of task execution, the used processors and their activation sequence are given, the optimum distribution can be found by applying linear programming. The case of a single processor boils down to a well known operations research problem of scheduling in two-machine flowshop. Though it may seem trivial, this special case may be useful in practical situations when parallel computations both start and complete in roughly the same time on all processors. Finally, bounds on the performance of heuristics for the problem have been searched for.

Still, some problems remain open: the complexity status remains unknown for the problems of scheduling on uniform processors without simultaneous completion, and scheduling on identical processors. Improving the bounds on approximability can be subject of the further study. From the practical point of view, the problems considered in this paper have to be solved in reasonable time using little information about the parallel system and the applications. Algorithms for scheduling with a limited knowledge can be the subject of the future research.

References

[1] V.Bharadwaj, D.Ghose, V.Mani, T.Robertazzi, *Scheduling divisible loads in parallel and distributed systems.* IEEE Computer Society Press, Los Alamitos CA, 1996.

[2] J.Błażewicz, K.Ecker, E.Pesch, G.Schmidt, J.Węglarz, *Scheduling Computer and Manufacturing Processes*, Springer-Verlag: Heidelberg, 1996.

[3] M.Drozdowski, *Selected problems of scheduling tasks in multiprocessor computer systems*, Series: Monographs, No.321, Poznań University of Technology Press, Poznań, (1997), (see also `http://www.cs.put.poznan.pl/~maciejd/h.ps`).

[4] M.R.Garey, D.S.Johnson, *Computers and Intractability: A guide to the theory of NP-completeness*, Freeman, San Francisco, 1979.

[5] S.M.Johnson, Optimal two- and three-stage production schedules with setup times included, *Naval Research Logistics Quarterly* 1 (1954) 61-67.

[6] K.Ko, T.G.Robertazzi, Scheduling in an Environment of Multiple Job Submission, *Proceedings of the 2002 Conference on Information Sciences and Systems*, Princeton University, Princeton NJ, March 2002.

[7] M.Pinedo, *Scheduling: theory, algorithms, and systems*, Prentice Hall, Englewood Cliffs, 1995.

[8] T.Robertazzi, Ten reasons to use divisible load theory, *IEEE Computer* 36 (2003) 63-68.

[9] J.Sohn, T.Robertazzi, A Muli-Job Load Sharing Strategy for Divisible Jobs on Bus Networks, Department of Electrical Engineering, SUNY at Stony Brook, Stony Brook, New York, Technical Report 697, 1994.

[10] B.Veeravalli, G.Barlas, Efficient Scheduling Strategies for Processing Multiple Divisible Loads on Bus Networks, *Journal of Parallel and Distributed Computing* 62, 132-151 (2002)