SCHEDULING MULTIPLE DIVISIBLE LOADS

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

Abstract

In this paper we study the scheduling of multiple divisible loads on a star network of processors. We show that this problem is computationally hard. Special cases solvable in polynomial time are identified.

Key words: divisible loads, scheduling, computational complexity

The International Journal of High Performance Computing Applications, Volume 20, No. 1, Spring 2006, pp. 19–30
DOI: 10.1177/1094342006061879

Figure 6 appears in color online: http://hpc.sagepub.com

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) has emerged as a new paradigm in parallel processing, which links scheduling, communication optimization, and performance modeling. Surveys of DLT literature can be found in Bharadwaj et al. (1996), Drozdowski (1997), and Robertazzi (2003).

1.1 Problem Formulation

In this paper we consider scheduling multiple divisible loads in a star network. Each load, which is 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 use the word "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 the "originator" is located. In star topology, processors $P_1, \ldots, P_m$ communicate only with the originator P_0. The originator is not computing. Were it otherwise, the computing capability of the originator can be represented as an additional processor. Tasks in $\mathcal{T}$ may be reordered by the originator to achieve good performance of the computations. The originator splits the loads of the tasks into parts and sends them to processors $P_1, \ldots, P_m$ for remote processing. Only some subset $\mathcal{P}_j \subseteq \mathcal{P}$ of all processors may be used to process task T_j. We denote by α_{ij} the size of the part of task T_j 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 the load parts sum up to the task load, i.e. $\Sigma_{i=1}^{m} \alpha_{ij} = V_j$. 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.

Each processor is described by three parameters: computing rate, communication rate of the link to the originator, and 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

[1]INSTITUTE OF COMPUTING SCIENCE POZNAŃ UNIVERSITY OF TECHNOLOGY, POLAND (MACIEJ.DROZDOWSKI@CS.PUT.POZNAN.PL)

[2]POZNAŃ SUPERCOMPUTING AND NETWORKING CENTER POLAND

[3]LABORATOIRE D'INFORMATIQUE DU HAVRE UNIVERSITÉ DU HAVRE, FRANCE

heterogeneity of the computing environment, three forms of the star system can be distinguished (we use scheduling theory naming convention; Pinedo 1995; Błażewicz et al. 1996).

- 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 denote by C_{ij} the 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} denotes the processing rate of processor P_i perceived by task T_j. Computing for load α_{ij} lasts for $A_{ij}\alpha_{ij}$. The case of unrelated processors is the most general case. Both the processors and the tasks differ because of variations in the solved problems, 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, $\forall_{P_i \in \mathcal{P}} A_i = A$, $C_i = C$, $S_i = S$. Identical processors are a further specialization of the uniform processors. They represent, for example, the same parallel program executed in a homogeneous environment for different input data sets.

We assume that processors have sufficient memory buffers to store the received loads, 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 $\Sigma_{i=1}^{n} V_j$ because each task is a separate scheduling entity, separate memory object, 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 sizes of the load parts. Our objective is minimization of the schedule length, denoted by C_{max}. 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 (Bharadwaj et al. 1996; Drozdowski 1997; Robertazzi 2003). It has been observed in earlier DLT papers

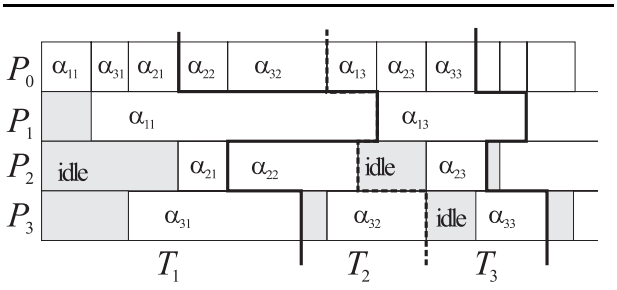

Fig. 1 An example of a permutation schedule.

that if the result returning time may be neglected, then the schedule for a single task is shortest when all the activated processors complete computations at the same moment. This requirement may be extended to the multiple loads case. We say that a schedule has a 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 finishing on different processors in widely scattered moments of time.

However, the process of result returning may be as time-consuming as load distribution and computations. In such cases we 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. We assume that transfer rates and startup times are the same for sending the load to the processors, and for returning the results.

It is assumed in this paper that the originator constructs permutation schedules (see, for example, Pinedo 1995 and Błażewicz et al. 1996 for a classic definition). By permutation schedule, we mean that a task is sent to the processors only once, that a processor executes the task only once, and the sequence of the tasks is the same on all processors. Consequently, communications and computations are non-preemptive, i.e. cannot be suspended and restarted later. If $P_i \notin \mathcal{P}_j$, then a dummy computation interval of length 0 is inserted on P_i. An example of permutation schedule is shown in Figure 1. When returning of the results is considered, we also assume permutation schedules, by which we mean that the order of the tasks in the distribution, computation, and result collection phases is the same.

1.2 Related Work

Scheduling multiple divisible loads has already been considered in DLT for communications without startup

times. In Bharadwaj et al. (1996) and Sohn and Robertazzi (1994), it was assumed that the task execution sequence was first-in first-out, processors were uniform, and task computations finished simultaneously. Furthermore, all processors were used by each task. In the multijob scheme proposed in Bharadwaj et al. (1996) and Sohn and Robertazzi (1994), communications of some task T_j overlap with computations of task T_{j-1} preceding T_j in the execution sequence. This allows computations to be started for T_j on some processors $P_1, \ldots, P_m$ immediately after the end of task T_{j-1}. Processors $P_{m'+1}, \ldots, P_m$ are idle until receiving their load share of T_j. Using the formulae provided in Bharadwaj et al. (1996) and Sohn and Robertazzi (1994) 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 $O(m^2n)$.

Veeravalli and Barlas (2002) made the same assumptions on the task sequence, processor selection, simultaneous computation completion, and zero startup time. Under the above assumptions a multi-installment load distribution strategy has been proposed to ensure that all processors work continuously on tasks $T_2, \ldots, T_n$. 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. As communications are shorter, all processors may receive some load earlier, and may work continuously on T_j. Unfortunately, it was observed in Veeravalli and Barlas (2002) that this strategy does not work for certain combinations of task and processor parameters. Four heuristics have been proposed in Veeravalli and Barlas (2002). It was demonstrated by a set of simulations that a multi-installment strategy gives the shortest schedule in most of the cases.

Ko and Robertazzi (2002) have given a probabilistic analysis for multiple loads arriving at multiple nodes of a fully connected network of identical processors. Marchal et al. (2004) have studied a steady state of multiple divisible loads executed in an arbitrary network. Sequencing of the communication and computation is ignored in the steady state. Instead of minimizing schedule length, the total load executed in a unit of time is maximized. Only the fraction of processor time or communication channel bandwidth dedicated to an application has to be determined.

In this paper we analyze the multiple divisible load scheduling problem along the lines of computational complexity. In the earlier literature DLT has been considered as a generally tractable linear model of distributed computations. The paper is organized as follows. In Section 2 we identify computationally hard cases. In Section 3 we present some polynomially solvable cases of the problem. We give bounds on the quality of approximation algorithms in Section 4.

2 Complexity

In this section we identify computationally hard (strictly speaking NP-hard, or NP-hard in a strong sense; see Garey and Johnson 1979) cases of the multiple divisible load scheduling problem. In our proofs of the computational complexity we use the NP-complete PARTITION problem, and the strongly NP-complete 3-PARTITION problem, defined as follows (Garey and Johnson 1979).

PARTITION
INSTANCE: A finite set $E = \{e_1, \ldots, e_q\}$ of positive integers.
QUESTION: Is there a subset $E' \subseteq E$ such that

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

3-PARTITION
INSTANCE: A finite set $G = \{g_1, \ldots, g_{3q}\}$ of positive integers, such that $\Sigma_{j=1}^{3q} g_j = Hq$ and $H/4 < g_j < H/2$ for $j = 1, \ldots, 3q$.
QUESTION: Can G be partitioned into q disjoint subsets $G_1, \ldots, G_q$ such that $\Sigma_{j \in G_i} g_j = H$ for $i = 1, \ldots, q$?

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

Proof. For $m = 1$ this problem is obviously in NP because NDTM has to guess the sequence of tasks execution. We show that our scheduling problem is NP-hard using the following polynomial time transformation from PARTITION:

$$n = q + 1,$$

$$V_j = 1, \beta_j = 1 \text{ for } j = 1, \ldots, n,$$

$$S_{1j} = 0 \text{ for } j = 1, \ldots, n,$$

$$C_{1j} = 0 \text{ for } j = 1, \ldots, q, C_{1n} = F,$$

$$A_{1j} = e_j \text{ for } j = 1, \ldots, q, A_{1n} = 1.$$

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 Figure 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 non-preemptive, each of the tasks $T_1, \ldots, T_q$ must fit into either interval $[0, F]$, or interval $[F + 1, 2F + 1]$. For the set of tasks $\mathcal{T}_{[0, F]}$ for which com-

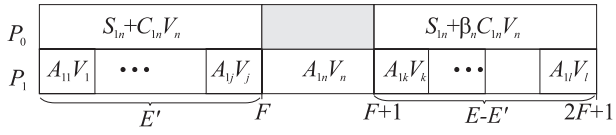

Fig. 2 Illustration to the proof of Theorem 1.

putations are performed in [0, F], we have $\Sigma_{T_j \in \mathcal{T}_{[0,F]}} A_{1j}V_j = \Sigma_{T_j \in \mathcal{T}_{[0,F]}} e_j \le F$. Analogously, for the tasks in interval [F + 1, $2F$ + 1]: $\Sigma_{T_j \in \mathcal{T}_{[F+1,2F+1]}} A_{1j}V_j = \Sigma_{T_j \in \mathcal{T}_{[F+1,2F+1]}} e_j \le F$. Thus, the PARTITION instance also has a positive answer. Consequently, the scheduling problem is NP-hard. □

Theorem 2. *If the result returning time is negligible, then the 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 $H > q$. Were it otherwise, all g_j can be multiplied by $q > 1$ to fulfill this requirement. The instance of the scheduling problem can be constructed as follows:

$$n = 4q + 1,\ V_j = 1 \text{ for } j = 1, \ldots, n,$$

$$S_{1j} = 0,\ S_{2j} = 0,\ C_{1j} = 1,\ C_{2j} = g_j,\ A_{1j} = \infty,$$

$$A_{2j} = H^3 g_j \text{ for } j = 1, \ldots, 3q,$$

$$S_{1,\,3q+1} = 0,\ S_{2,\,3q+1} = 0,\ C_{1,\,3q+1} = 1,\ C_{2,\,3q+1} = 1,$$

$$A_{1,\,3q+1} = H^4 + H,\ A_{2,\,3q+1} = \infty,$$

$$S_{1j} = 0,\ S_{2j} = 0,\ C_{1j} = H^4,\ C_{2j} = 1,\ A_{1j} = H^4 + H,$$

$$A_{2j} = \infty \text{ for } j = 3q + 2, \ldots, 4q,$$

$$S_{1,\,4q+1} = 0,\ S_{2,\,4q+1} = 0,\ C_{1,\,4q+1} = H^4,$$

$$C_{2,\,4q+1} = 1,\ A_{1,\,4q+1} = 1,\ A_{2,\,4q+1} = \infty,$$

$$y = q(H^4 + H) + 2.$$

We ask whether a schedule no longer than y exists. If the 3-PARTITION instance has a positive answer, then a feasible schedule of length y may look like that in Figure 3. Observe that processor P_2 can start executing tasks immediately after its first communication. Thus, there can also be other schedules no longer than y when a 3-PARTITION exists.

Suppose a feasible schedule no longer than y exists. Due to the values of parameters A_{ij}, tasks T_1, …, T_{3q} can be executed on P_2 only, and tasks T_{3q+1}, …, T_{4q+1} on P_1 only. The total computing time on P_1 is $q(H^4 + H) + 1 = y - 1$, while the shortest load distribution operation lasts 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, the originator must communicate all the time with the exception of the last time unit when task T_{4q+1} must be executed on P_1.

The total computing requirement put on P_2 by tasks T_1, …, T_{3q} is qH^4. After excluding the first communication of T_{3q+1}, P_2 can be idle at most $qH + 1$ time units. To avoid idling on P_1, sending the load for the second task executed on P_1 must start at time $H + 1$ at the latest. Therefore, no further 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 from the end of T_j computations until the end of the communication operation of the second task executed by P_1. Let us calculate this idle time. $H^4 + g_j$ is the span of the interval from the end of T_i communication operation (the moment when P_2 can start computing) until the end of the communication operation of the second task executed by P_1 (when P_2 can start receiving any new load). $H^3(g_i + g_j)$ is the time of computing operations which can be executed on P_2 in this interval. The idle time on P_2

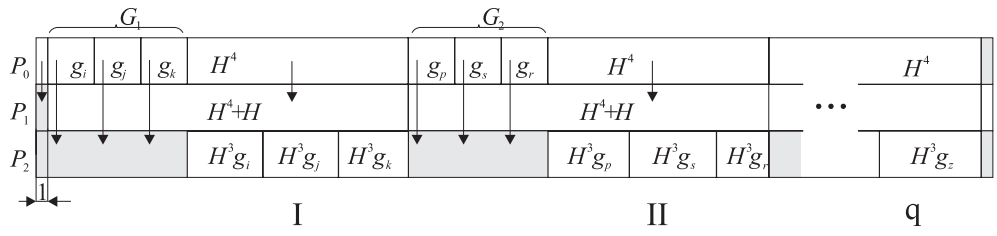

Fig. 3 Illustration to the proof of Theorem 2.

would be at least $H^4 + g_j - H^3(g_i + g_j)$. Since $H > q$ and $H > 1$ we have $H^4 + g_j - H^3(g_i + g_j) > H^4 - H^3(H-1) = H^3 \geq H^2 + H^2 > qH + 1$, while the idle time on P_2 cannot be greater than $qH + 1$. Hence, exactly three communications to P_2 must be carried out before sending the second task to P_1.

The sum of the computation times of the first three tasks T_i, T_j, T_k allocated to P_2 must be equal to H^4. If it is less, then it is at most $H^3(g_i + g_j + g_k) \leq H^4 - H^3$, which results in $H^3 > qH + 1$ 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 lasts longer than H and the sending operation of the second task allocated to P_1 cannot start at time $H + 1$, which results in additional idle time on P_1. Consequently, the three tasks must be processed in exactly H^4 time units. Otherwise the schedule of length y cannot exist.

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 $1 + iH^4 + (i+1)H$ for $i = 1, \ldots, q-1$. This creates a free time interval for at most three communications of the tasks assigned to P_2. Also, no fewer than three tasks can be started by the originator, otherwise there will be excessive idle time on P_2. The processing times of the three tasks must be equal to H^4, otherwise either P_2 or the originator must be idle for too long a time. We conclude that for each triplet of tasks assigned to P_2 their computing time is H^4. Hence, the 3-PARTITION instance has a positive answer. □

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

Proof. First we 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

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

The term $\Sigma_{i=1}^{m}(\tau/A_i)$ 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 processed load by $\Sigma_{j=i}^{m}(S_i/A_j)$. The term $\Sigma_{i=1}^{m}\Sigma_{j=i}^{m}(S_i/A_j)$ in equation (2) is the amount of the load that could not be processed due to the communication delays. Suppose that $(1/A_i) = S_i$ for all i. Equation (2) reduces to

$$V = \tau \sum_{i=1}^{m} S_i - \sum_{i=1}^{m} \sum_{j=i}^{m} S_i S_j$$

$$= \tau \sum_{i=1}^{m} S_i - \frac{1}{2}\left(\sum_{i=1}^{m} S_i \right)^2 - \frac{1}{2} \sum_{i=1}^{m} S_i^2. \qquad (3)$$

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

We show the NP-hardness of the problem by a polynomial time transformation of the PARTITION problem. We assume that $e_i > 2$ for $i = 1, \ldots, q$. 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:

$$n = 2, m = q,$$

$$S_i = e_i, A_i = \frac{1}{S_i} = \frac{1}{e_i}, C_i = 0, \text{ for } i = 1, \ldots, q$$

$$V_1 = 4F^2 - \frac{1}{2} \sum_{i=1}^{m} e_i^2, V_2 = F$$

$$y = 3F + 1$$

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 the PARTITION problem. A feasible schedule for the instance of the scheduling problem is shown in Figure 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 equations (2) and (3), we obtain

$$3F \sum_{i=1}^{m} e_i - \frac{1}{2}\left(\sum_{i=1}^{m} e_i \right)^2 - \frac{1}{2} \sum_{i=1}^{m} e_i^2$$

$$= 6F^2 - \frac{1}{2}(2F)^2 - \frac{1}{2} \sum_{i=1}^{m} e_i^2 = V_1.$$

Thus, T_1 is executed feasibly. The communications of T_1 finish at time $2F$, and therefore communications of T_1 which take $\Sigma_{i \in E'} S_i = F$ fit in F time units of available time. In the last time unit of interval $[y-1, y]$ the selected processors process

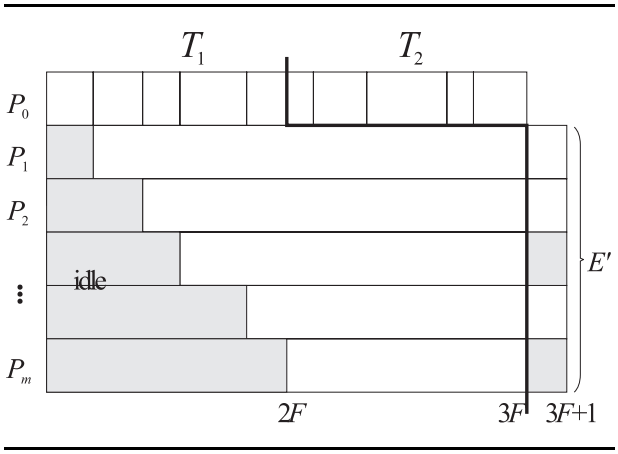

Fig. 4 Illustration of the proof of Theorem 3.

$$\sum_{i \in E'} \frac{1}{A_i} = \sum_{i \in E'} e_i = F$$

units of load. Hence, also T_2 is executed feasibly.

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 processor. Using equation (3) the volume of the processed load for T_1 is at most

$$\begin{aligned} V_1 &= y\sum_{i=1}^{m-1} S_i - \frac{1}{2}\left(\sum_{i=1}^{m-1} S_i\right)^2 - \frac{1}{2}\sum_{i=1}^{m-1} S_i^2 \\ &= (3F+1)\sum_{i=1}^{m-1} e_i - \frac{1}{2}\left(\sum_{i=1}^{m-1} e_i\right)^2 - \frac{1}{2}\sum_{i=1}^{m-1} e_i^2 \\ &= (3F+1)\sum_{i=1}^{m} e_i - (3F+1)e_m - \frac{1}{2}\left(\sum_{i=1}^{m} e_i\right)^2 \\ &\quad - \frac{1}{2}\sum_{i=1}^{m} e_i^2 + \frac{1}{2}\left(e_m^2 + 2e_m\sum_{i=1}^{m-1} e_i\right) + \frac{1}{2}e_m^2 \\ &= V_1 + 2F - (3F+1)e_m + e_m^2 + e_m\sum_{i=1}^{m-1} e_i \\ &= V_1 + 2F - e_m\left(3F + 1 - \sum_{i=1}^{m} e_i\right) \\ &= V_1 + 2F - e_m(F+1) < V_1 \end{aligned}$$

because $e_m > 2$. Hence, all m processors must be used by task T_1. 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 the computations of T_2. Note that $\forall_i S_i \geq 2$, 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 $\Sigma_{i \in \mathcal{P}'} S_i = \Sigma_{i \in \mathcal{P}'} e_i \leq F$. The load of T_2 processed in interval $[3F, 3F+1]$ must satisfy $\Sigma_{i \in \mathcal{P}'} \frac{1}{A_i} = \Sigma_{i \in \mathcal{P}'} e_i \geq F$. Thus, the answer is positive for the 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 no simpler. We explain it in the following observation.

Observation 4. *If the result returning time is negligible and simultaneous completion is required, then the 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. Communications of T_2 last at least $\min_{P_i \in \mathcal{P}}\{S_i\} = \min_{i \in E}\{e_i\} > 1$. The duration of T_1 processing is at least $3F$ (see the proof of Theorem 3). The schedule length is at least $3F + 2$. Thus, only sequence (T_1, T_2) allows for a schedule of length at most $3F + 1$, which by the proof of Theorem 3 exists if and only if PARTITION exists. □

Note that Theorems 1 and 2 can be proved also for non-permutation schedules. Preemption in computation and communication can be eliminated by sufficiently long communication or computation startup times. Theorem 3 relies on the simultaneous completion of T_1, and hence it holds also for non-permutation schedules.

We conclude from the above results that the problem of scheduling multiple divisible loads is computationally hard. This means in practice that this problem has a hard combinatorial core which could not be expected from the earlier DLT literature. The main sources of the computational complexity are sequencing the tasks, selecting the processors to use, and 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 rational linear programming. Let us first study the case when simultaneous completion of the computations is not required, and the result 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 $j = 1, \ldots, n$ sent to processor $i = 1, \ldots, m$ is denoted by $\alpha_{ij} \geq 0$. The optimum distribution of the load can be found by the linear program: minimize C_{max} subject to

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

$$l = 1, \ldots, n, k = 1, \ldots, |\mathcal{P}_j| \qquad (4)$$

$$\sum_{i \in \mathcal{P}_j} \alpha_{ij} = V_j \qquad j = 1, \ldots, n. \qquad (5)$$

The term $\Sigma_{j=1}^{l-1} \Sigma_{i=1}^{|\mathcal{P}_j|} [S_{f(j,i)j} + \alpha_{f(j,i)j} C_{f(j,i)j}]$ in inequalities (4) is the time of sending the load for tasks $T_1, \ldots, T_{l-1}$. Sending the load to processors $f(l, i)$ activated as $i = 1, \ldots, k$ in the sequence of processors executing task T_l lasts $\Sigma_{i=1}^{k} [S_{f(l,i)l} + \alpha_{f(l,i)l} C_{f(l,i)l}]$. $\Sigma_{j=l}^{n} \alpha_{f(l,k)j} A_{f(l,k)j}$ is the time of computing the load parts of tasks $T_l, \ldots, T_n$, sent to processor $f(l, k)$, activated as kth for task T_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 equations (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 last two 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 obtain from equations (4)–(5) the optimum solution $\alpha_{11} \approx 18.333$, $\alpha_{21} \approx 9.333$, $\alpha_{31} \approx 4.333$, $\alpha_{12} \approx 1.667$, $\alpha_{22} \approx 0.333$, $C_{max} \approx 39.333$,

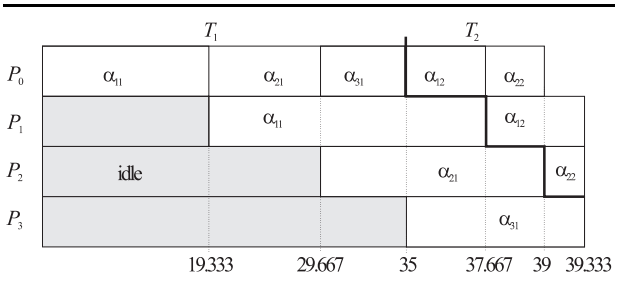

Fig. 5 Optimal schedule for example 1 (does not preserve proportion).

shown in Figure 5. The exclusion of both P_3 and P_2 from processing T_2 does not reduce schedule length any further. □

Observe that in the optimum schedule for example 1 computations on T_1 do not finish on all processors at the same time. This 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. each task uses all processors. We experimentally studied patterns that appear in the optimal solutions under the above conditions. When communication delays are big in comparison with computing time, then not all processors are exploited. This is the case when $C \gg A/m$. When communication delays are of similar order as computations, then the load of each task is distributed nearly equally between the processors. The exceptions are the leading and trailing tasks. In the leading tasks, the 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 Figure 6(a) where changes of α_{ij} from task to task are shown. Each line in Figure 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 $C \ll A/m$, then the 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 Figure 6(b). 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 equations (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

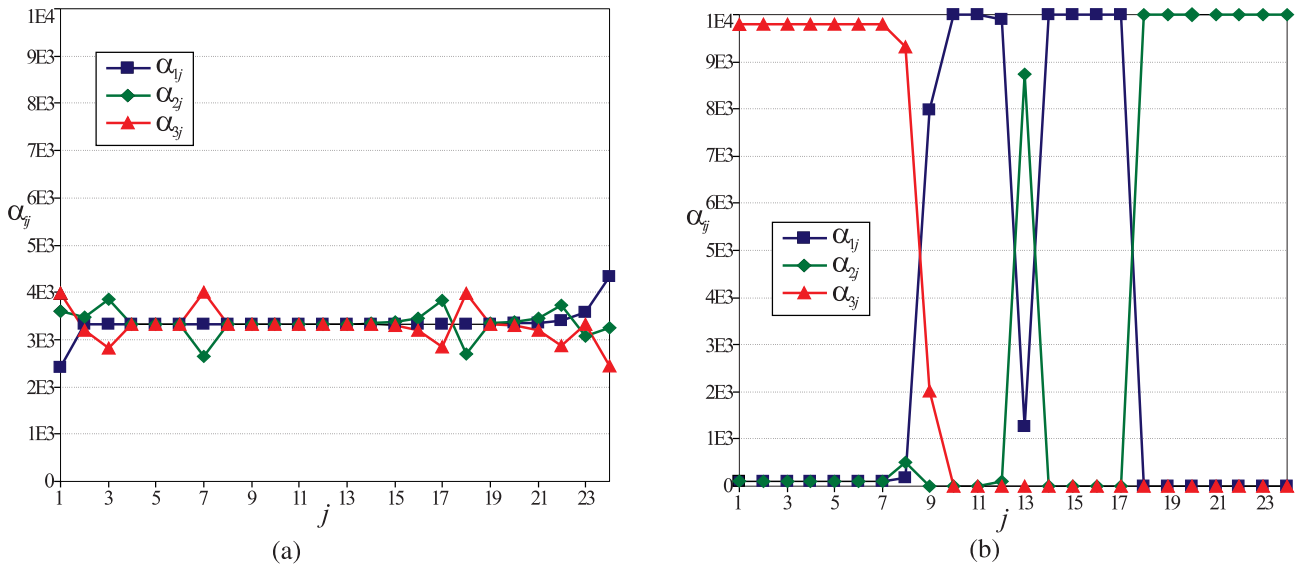

Fig. 6 Distribution of the load (α_{ij}) versus task number (j); $m = 3$, $n = 24$, $V = 1E4$, $C = S = 1$. (a) $A = 3$. (b) $A = 1E2$.

following linear program solves the case with simultaneous completion: minimize C_{max} subject to

$$z_{l-1} + \alpha_{f(l,k)l} A_{f(l,k)l} \le z_l$$

$$l = 2, \ldots, n, k = 1, \ldots, |\mathcal{P}_j| \quad (6)$$

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

$$l = 1, \ldots, n, k = 1, \ldots, |\mathcal{P}_j| \quad (7)$$

$$z_n = C_{max} \quad (8)$$

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

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 equation (8) the end of the last task is also the end of the schedule. The tasks are fully processed by equation (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 that presented in Figure 5. Hence, requiring simultaneous completion of computations on all processors may prevent obtaining an optimum schedule.

The methods used above can be extended to deal with the returning of the results. Linear programming formulations for returning of the results can be found in Drozdowski, Lawenda, and Guinand (2004).

3.2 Continuous Computing

In this section we assume simultaneous completion, use of 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, until the end of the last task. We call this situation continuous computing. An example of a continuous computing is shown in Figure 7. Let us start with some observations.

Observation 5. *When computing is continuous, only schedule for the first task decide on the length of the whole schedule.*

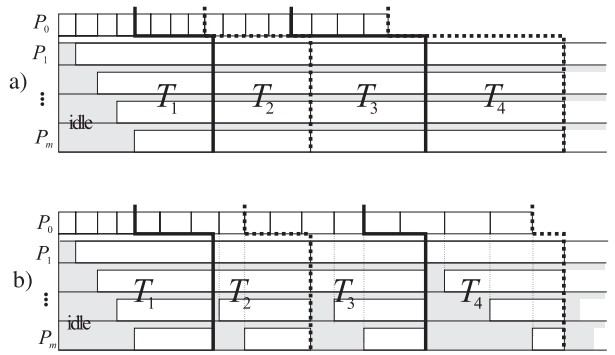

Fig. 7 Example of (a) continuous and (b) non-continuous computing.

Proof. Since all processors are used by each task, selection of the processor set is immaterial. With the exception of the first task, the sequence of processor activation can be arbitrary because processors are used in the same interval due to continuous computing and simultaneous completion. With the exception of the first task, the load assigned to processor P_i for task T_j is

$$\alpha_{ij} = \frac{V_j}{A_{ij}\sum_{l=1}^{m}(1/A_{lj})}$$

and a decision on task chunk sizes is not necessary. □

Note that also the sequence of the tasks matters because only some task sequences may result in continuous computing (cf. the proofs of Theorem 3, and Observation 4).

Observation 6. *If the first task has the shortest possible completion time, and computing is continuous, then the schedule for all tasks is optimum.*

Proof. The schedule cannot be shorter because all processors work in parallel from the end of the first task, and the first task is completed in the shortest possible time. □

Continuous computing is possible when the load of any task T_j is distributed to the processors before the computations on T_j are started. This leads to the following greedy approach to the construction of continuous schedules.

1. For $i = 1$ to n: select T_i as the first task, and construct for it an optimum schedule $\mathcal{S}_i$.
2. Select the shortest schedule $\mathcal{S}$ in $\mathcal{S}_1, \ldots, \mathcal{S}_n$. Set $\mathcal{T}' = \mathcal{T} - \{T_i\}$.
3. While $\mathcal{T}' \neq \emptyset$ do the following.
   3.1. Select task T_j whose communication interval $\Sigma_{l=1}^{m}(S_{lj} + C_{lj}\alpha_{lj})$ fits in the interval between the end of communication and the end of computation of the preceding task, and which maximizes interval between T_j ends of communications and computations, i.e. $A_{1j}\alpha_{1j} - \Sigma_{l=1}^{m}(S_{lj} + C_{lj}\alpha_{lj})$, where

$$\alpha_{lj} = \frac{V_j}{A_{lj}\sum_{k=1}^{m}(1/A_{kj})}$$

   if there is no task satisfying the above conditions then stop.
   3.2. Append T_j to the end of schedule $\mathcal{S}$; set $\mathcal{T}' = \mathcal{T}' - \{T_j\}$.

If the resulting schedule has a continuous computing property, then it is optimal by Observation 6. Beyond the construction of the optimum schedule for the first task the above algorithm can be implemented to run in $O(nm + n^2)$ time.

Unfortunately, it is hard to claim that the above algorithm builds optimum schedules in polynomial time in general. To the best of our knowledge, the complexity of scheduling single divisible load (step 1 in the above algorithm) on a heterogeneous star remains open in the general case. Two decisions must be made: the set of used processors and the sequence of their activation must be selected. If $\forall_{ij} S_{ij} = 0$, then it can be shown that all processors take part in computations, and they should be activated according to increasing C_{ij} for task T_j (Bharadwaj et al. 1996; Błażewicz and Drozdowski 1997; Beaumont et al. 2005). If $\forall_{ij} S_{ij} \neq 0$, then it can be shown that processors participating in the computation should be activated according to increasing C_{ij} for task T_j (Beaumont et al. 2005). Yet, it is not known which processors should be used (cf. Figure 4). Hence, it is not guaranteed that the above method constructs a schedule with a continuous computing property if such a schedule exists.

We propose sufficient conditions under which an optimum schedule can be constructed by the above algorithm. This means that in the set of optimum schedules with continuous computing there is a subset satisfying our conditions. Suppose the processors are identical. Executing the tasks according to the increasing sizes (V_j) is called the shortest processing time (SPT) sequence. Let us assume that tasks are ordered according to the SPT rule, i.e. $V_1 \leq V_2 \leq \ldots \leq V_n$.

Theorem 7. *If computing is continuous, and*

$$\forall_{T_j \in \mathcal{T}} V_j > \frac{Sm}{(A/m) - C},$$

then the SPT maximizes the interval between the completion of the task communication, and starting of its computations on identical processors.

Proof. The requirement

$$\forall_{T_j \in \mathcal{T}} V_j > \frac{Sm}{(A/m) - C}$$

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. The proof based on pairwise interchange can be found in Drozdowski, Lawenda, and Guinand (2004). □

Thus, if it is possible to maintain continuous computing at all, then the SPT will also do it, provided that

$$\forall_{T_j \in \mathcal{T}} V_j > \frac{Sm}{(A/m) - C}.$$

The conditions of the optimality of the SPT sequence in continuous computing are the following.

Theorem 8. *The SPT is the optimum task sequence on identical processors if*

$$\forall_{T_j \in \mathcal{T}} V_j > \frac{Sm}{(A/m) - C},$$

and $x_j > Sm + V_{j+1}C$ *for* $j = 1, \ldots, n-1$, *where*

$$x_1 = \frac{CV_1 - (SA/C)\{[1 + (C/A)]^m - [1 + (C/A)]\} + (m-1)S}{[1 + C/A]^m - 1},$$

$$x_j = x_{j-1} + \frac{V_j A}{m} - Sm - V_j C \text{ for } j = 2, \ldots, n-1$$

Proof. If it is possible to maintain continuous computing on identical processors at all, then according to Theorem 7, the SPT sequence will also have this property because the SPT maximizes the distance between task communication completion and computation start. Using Observation 6 the first task must be finished in the shortest possible time. On identical processors, task T_1 with the smallest load V_1 satisfies this condition. Hence, the SPT task sequence is optimal among schedules with a continuous computing property on identical processors.

It still remains to ensure that continuous computing is possible. This 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 (see Figure 8). This condition demands 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 $x_j = x_{j-1} + (V_j A/m) - Sm - CV_j$ for $j = 2, \ldots, n-1$. The 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

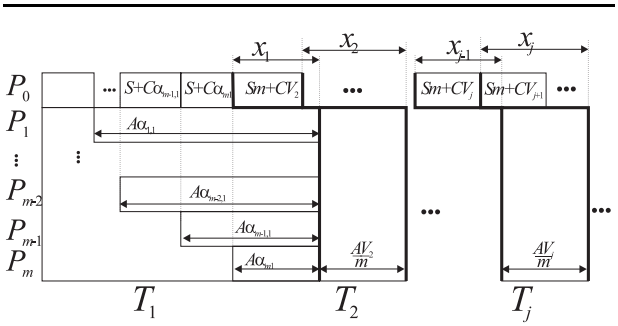

Fig. 8 Illustration of the proof of Theorem 8.

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

α_{i1}, for $i = 1, \ldots, m-1$, can be expressed as a function of α_{m1}:

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

The size of the first task is

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

from which we derive

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

□

3.3 m = 1

Observation 9. *If the result returning time is negligible, then the multiple divisible load scheduling problem for one* ($m = 1$) *unrelated processor is solvable in O(nlogn) time using the algorithm of Johnson (1954).*

Proof. If the results are not returned, and only one machine ($m = 1$) is available, then execution of a task reduces to two operations: the communication operation involving originator P_0, followed by the computation operation involving P_1. This situation is equivalent to a two-machine flowshop. The two-machine flowshop is solvable in $O(n\log n)$ time using the algorithm of Johnson (1954) (or see, for example, Pinedo 1995; Błażewicz et al. 1996). □

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 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.

Although this special case may seem trivial, it represents practical situations when parallel computations both start and complete in roughly the same time on all processors. In such situations, all processors are working in parallel, and behave as a single processing facility, i.e. a single processor.

4 Approximability

In this section we study the bounds on the quality of approximation algorithms for the 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 a 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 10. *The length C^H_{max} of a schedule built by any greedy heuristic H solving the multiple divisible load scheduling problem on identical processors satisfies:*

$$\frac{C^H_{max}}{C^*_{max}} \le 2m,$$

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 the initiator performs communications, and intervals of total length E_A when initiator does not perform any communications because all processors compute. In the case of identical processors $E_C = \Sigma_{j=1}^{n}(\Sigma_{P_i \in \mathcal{P}_j} S + CV_j)$. Note that $nS + C\,\Sigma_{j=1}^{n} V_j \le C^*_{max}$ because each load must be sent. In the worst case, some heuristic may activate all processors while only a single processor is necessary for each task. Consequently, $\Sigma_{j=1}^{n}\Sigma_{P_i \in \mathcal{P}_j} S - nS = S\Sigma_{j=1}^{n}(|\mathcal{P}_j| - 1) \le Sn(m-1) \le (m-1)C^*_{max}$. Some heuristic may also tend to use fewer processors than necessary. In the worst case $|\mathcal{P}_j| = 1$, and $E_A \le \Sigma_{j=1}^{n} AV_j$. Note that

$$\sum_{j=1}^{n} \frac{AV_j}{m} \le C^*_{max}.$$

Hence, $E_A \le mC^*_{max}$. Altogether we have $C^H_{max} = E_C + E_A \le C^*_{max} + (m-1)C^*_{max} + mC^*_{max}$. □

The results of Theorem can be further strengthened. If $S = 0$, then in the above proof $\Sigma_{j=1}^{n}\Sigma_{P_i \in \mathcal{P}_j} S - nS = 0$, and the ratio of schedule lengths can be narrowed to

$$\frac{C^H_{max}}{C^*_{max}} \le m + 1.$$

If

$$\forall_{T_j \in T} CV_j + mS > \frac{V_j A}{m},$$

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 $(C^H_{max}/C^*_{max}) \le m + 1$.

In the latter case a better bound can be obtained by a heuristic *CC* attempting to build a schedule with a continuous computing property. Divide the load of each task into m equal parts and send them to the processors. For each task, start computations synchronously on all processors as soon as all processors have received their share of the load. Since all processors compute in parallel, we have $E_A \le C^*_{max}$. Hence, $E_C + E_A \le 2C^*_{max}$, and $(C^{CC}_{max}/C^*_{max}) \le 2$.

5 Conclusions

In this paper we have studied the combinatorial aspects of scheduling multiple divisible loads. It has been demonstrated that this problem is computationally hard for unrelated processors, and for uniform processors with simultaneous completion requirement. When the order of task execution, the used processors and their activation sequence are given, then the optimum distribution can be found in polynomial time by applying linear programming. The case of a single processor boils down to a well-known operations research problem of scheduling in a two-machine flowshop. Finally, bounds on the performance of heuristics for the problem have been sought. Still, the complexity of scheduling on uniform processors without simultaneous completion, and scheduling on identical processors remains unknown. Improving the bounds on approximability can be the subject of further study.

Acknowledgments

This research was partially supported by a grant from the Polish State Committee for Scientific Research. We are very grateful to Olivier Beaumont and Frédéric Vivien for their comments which helped to improve the quality of this paper.

Author Biographies

Maciej Drozdowski received an M.Sc. in control engineering in 1987, and a Ph.D. in computer science in 1992. In 1997, he defended his habilitation in computer science. Currently, he is an associate professor at the Institute of Computing Science, Poznań University of Technology. His research interests include design and analysis of algorithms, complexity analysis, combinatorial optimization, scheduling, and computer performance evaluation. He is a member of the IEEE Computer Society, and the Polish Information Processing Society.

Marcin Lawenda graduated from Poznań University of Technology and received an M.Sc. in computer science (parallel and distributed computation speciality) in 2000. He currently works for the Poznań Supercomputing and Networking Center in the project manager position for the Virtual Laboratory project. He is also a Ph.D. candidate in the Department of Computer Science, Poznań University of Technology. His research interests include parallel and distributed environments, scheduling and Grid technologies. He is author and coauthor of several reports and papers (20+) in conference proceedings and journals. He has been a member of the Polish Information Processing Society since 2000.

Dr. Frederic Guinand received an M.Sc. in computer science in 1991, and a Ph.D. in computer science in 1995 from the University of Grenoble. After a postdoc at the Swiss Federal Institute of Technology of Lausanne (EPFL, Switzerland), he held an Assistant Professor position at Le Havre University in 1997. Currently, he is a Professor in the Department of Applied Mathematics and Computer Science at Le Havre University, France. His research activities concern optimization for problems with uncertainties arising in high performance computing, bioinformatics and complex systems fields.

References

Beaumont, O., Casanova, H., Legrand, A., Robert, Y., and Yang, Y. 2005. Scheduling divisible loads on star and tree networks: results and open problems, *IEEE Transactions on Parallel and Distributed Systems* 16:207–218.

Bharadwaj, V., Ghose, D., Mani, V., and Robertazzi, T. 1996. *Scheduling Divisible Loads in Parallel and Distributed Systems*, IEEE Computer Society Press, Los Alamitos, CA.

Błażewicz, J. and Drozdowski, M. 1997. Distributed processing of divisible jobs with communication startup costs. *Discrete Applied Mathematics* 76:21–41.

Błażewicz, J., Ecker, K., Pesch, E., Schmidt, G., and Weglarz, J. 1996. *Scheduling Computer and Manufacturing Processes*, Springer-Verlag, Heidelberg.

Drozdowski, M. 1997. *Selected Problems of Scheduling Tasks in Multiprocessor Computer Systems*, Series Monographs, No. 321, Poznań University of Technology Press, Poznań (http://www.cs.put.poznan.pl/ mdrozdowski/h.ps).

Drozdowski, M., Lawenda, M., and Guinand, F. 2004. Scheduling multiple divisible loads. Technical Report RA-007/04, Institute of Computing Science, Poznań University of Technology (http://www.cs.put.poznan.pl/ mdrozdowski/rapIIn/RA07-04ps.zip).

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

Johnson, S. M. 1954. Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly* 1:61–67.

Ko, K., and Robertazzi, T. G. 2002. Scheduling in an environment of multiple job submission. *Proceedings of the Conference on Information Sciences and Systems*, Princeton University, Princeton NJ, March.

Marchal, L., Yang, Y., Casanova, H., and Robert, Y. 2004. A realistic network/application model for scheduling divisible loads on large-scale platforms. Research Report 2004-21, École Normale Supérieure de Lyon, Laboratoire de l'Informatique du Parallélisme.

Pinedo, M. 1995. *Scheduling: Theory, Algorithms, and Systems*, Prentice-Hall, Englewood Cliffs, NJ.

Robertazzi, T. 2003. Ten reasons to use divisible load theory, *Computer* 36:63–68.

Sohn, J. and Robertazzi, T. 1994. A multijob load sharing strategy for divisible jobs on bus networks. Technical Report 697, Department of Electrical Engineering, SUNY at Stony Brook, Stony Brook, New York.

Veeravalli, B. and Barlas, G. 2002. Efficient scheduling strategies for processing multiple divisible loads on bus networks. *Journal of Parallel and Distributed Computing* 62:132–151.