Available online at www.sciencedirect.com
ScienceDirect
European Journal of Operational Research 183 (2007) 546–563
EUROPEAN JOURNAL OF OPERATIONAL RESEARCH
www.elsevier.com/locate/ejor

Discrete Optimization

On a resource-constrained scheduling problem with application to distributed systems reconfiguration ☆

Renaud Sirdey [a,b,*], Jacques Carlier [b], Hervé Kerivin [c], Dritan Nace [b]

[a] *Service d'architecture BSC (PC 12A7), Nortel GSM Access R&D, Parc d'activités de Magny-Châteaufort, 78928 Yvelines Cedex 09, France*
[b] *UMR CNRS Heudiasyc (Université de Technologie de Compiègne), Centre de recherches de Royallieu, BP 20529, 60205 Compiègne Cedex, France*
[c] *UMR CNRS Limos (Université de Clermont-Ferrand II), Complexe scientifique des Cézeaux, 63177 Aubière Cedex, France*

Received 18 October 2005; accepted 18 October 2006
Available online 13 December 2006

Abstract

This paper is devoted to the study of a resource-constrained scheduling problem, the *Process Move Programming problem*, which arises in relation to the operability of certain high availability real-time distributed systems. Informally, this problem consists, starting from an arbitrary initial distribution of processes on the processors of a distributed system, in finding the least disruptive sequence of operations (non-impacting process migrations or temporary process interruptions) at the end of which the system ends up in another predefined arbitrary state. The main constraint is that the capacity of the processors must not be exceeded during the reconfiguration. After a brief survey of the literature, we prove the *NP*-hardness of the problem and exhibit a few polynomial special cases. We then present a branch-and-bound algorithm for the general case along with computational results demonstrating its practical relevance. The paper is concluded by a discussion on further research.

Keywords: Combinatorial optimization; Scheduling; Branch and bound; Distributed systems; OR in telecommunications

1. Introduction

Let us consider a distributed system composed of a set U of *processors* and let R denote the set of *resources* they offer. For each processor $u \in U$ and each resource $r \in R$, $C_{u,r} \in \mathbb{N}$ denotes the amount of resource r

☆ This research was supported in part by ANRT grant CIFRE-121/2004. Part of this work was done while the third author was working at the Institute for Mathematics and its Applications (IMA), University of Minnesota, Minneapolis, USA.
* Corresponding author. Address: Service d'architecture BSC (PC 12A7), Nortel GSM Access R&D, Parc d'activités de Magny-Châteaufort, 78928 Yvelines Cedex 09, France. Tel.: +33 1 69 55 41 18; fax: +33 1 34 85 14 73.
E-mail addresses: renauds@nortel.com (R. Sirdey), carlier@hds.utc.fr (J. Carlier), kerivin@math.univ-bpclermont.fr (H. Kerivin), dnace@hds.utc.fr (D. Nace).

0377-2217/$ - see front matter
doi:10.1016/j.ejor.2006.10.011

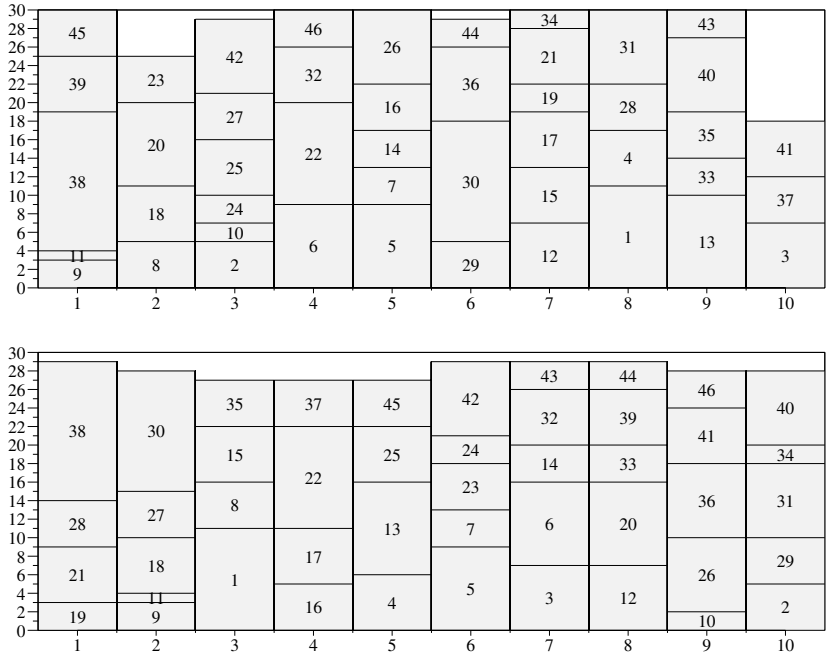

Fig. 1. Example of an instance of the PMP problem.

offered by processor u. We are also given a set P of applications, hereafter referred to as *processes*, which consume the resources offered by the processors. The set P is sometimes referred to as the *payload* of the system. For each process $p \in P$ and each resource $r \in R$, $w_{p,r} \in \mathbb{N}$ denotes the amount of resource r which is consumed by process p. Note that neither $C_{u,r}$ nor $w_{p,r}$ vary with time. Also, when $|R| = 1$, $C_{u,r}$ and $w_{p,r}$ are respectively denoted C_u and w_p (this principle is applied to other quantities throughout this paper).

An *admissible state* for the system is defined as a mapping $f : P \to U \cup \{u_\infty\}$, where u_∞ is a dummy processor having infinite capacity, such that for all $u \in U$ and all $r \in R$ we have

$$\sum_{p \in P(u;f)} w_{p,r} \leqslant C_{u,r}, \tag{1}$$

where $P(u;f) = \{p \in P : f(p) = u\}$. The processes in $\overline{P}(f) = P(u_\infty; f)$ are not instantiated, when this set is non-empty the system is in *degraded mode*.

An instance of the *Process Move Programming* (PMP) problem is then specified by two arbitrary system states f_i and f_t and, roughly speaking, consists in, starting from state f_i, finding the least disruptive sequence of operations at the end of which the system is in state f_t. The two aforementioned system states are respectively referred to as the *initial system state* and the *final system state* or, for short, the *initial state* and the *final state*.[1]

Fig. 1 provides an example of an instance of the PMP problem for a system with 10 processors, one resource and 46 processes. The capacity of each of the processors is equal to 30 and the sum of the consumptions of the processes is 281. The top and bottom figures respectively represent the initial and the final system states. For example, process number 23 must be moved from processor 2 to processor 6.

A process may be moved from one processor to another in two different ways: either it is *migrated*, in which case it consumes resources on both processors for the duration of the migration and this operation has virtually no impact on service, or it is *interrupted*, that is removed from the first processor and later restarted on the other one. Of course, this latter operation has an impact on service. Additionally, it is required that the

[1] Throughout the rest of this paper, it is assumed that $\overline{P}(f_i) = \overline{P}(f_t) = \emptyset$. When this is not true the processes in $\overline{P}(f_t) \setminus \overline{P}(f_i)$ should be stopped before the reconfiguration, hence some resources are freed, the processes in $\overline{P}(f_i) \setminus \overline{P}(f_t)$ should be started after the reconfiguration and the processes in $\overline{P}(f_i) \cap \overline{P}(f_t)$ are irrelevant.

capacity constraints (1) are always satisfied during the reconfiguration and that a process is moved (i.e., migrated or interrupted) at most once. The latter constraint is motivated by the fact that a process migration is far from being a lightweight operation (for reasons related to distributed data consistency which are out of the scope of this paper), as a consequence, it is desirable to avoid processes hopping around processors.

Throughout this paper, when it is said that a move is *interrupted*, it is meant that the process associated to the move is interrupted. This slightly abusive terminology significantly lightens our discourse. Additionally, it is now assumed that $|R| = 1$, unless otherwise stated.

For each processor u, a process p in $P(u;f_i)\backslash P(u;f_t)$ must be moved from u to $f_t(p)$. Let M denote the set of process moves. Then for each $m \in M$, w_m, s_m and t_m respectively denote the amount of resource consumed by the process moved by m, the processor from which the process is moved that is the *source* of the move and the processor to which the process is moved that is the *target* of the move. Lastly, $S(u) = \{m \in M : s_m = u\}$ and $T(u) = \{m \in M : t_m = u\}$.

A pair (I,σ), where $I \subseteq M$ and where $\sigma : M \setminus I \rightarrow \{1,\ldots,|M \setminus I|\}$ is a bijection, defines an admissible *process move program*, if provided that the moves in I are interrupted (the interruptions are performed at the beginning) the other moves can be performed according to σ without inducing any violation of the capacity constraints (1). Formally, (I,σ) is an admissible program if for all $m \in M\backslash I$ we have

$$w_m \leqslant K_{t_m} + \sum_{\substack{m' \in I \\ s_{m'} = t_m}} w_{m'} + \sum_{\substack{m' \in S(t_m)\setminus I \\ \sigma(m') < \sigma(m)}} w_{m'} - \sum_{\substack{m' \in T(t_m)\setminus I \\ \sigma(m') < \sigma(m)}} w_{m'}, \tag{2}$$

where $K_u = C_u - \sum_{p \in P(u;f_i)} w_p$, thereby guaranteeing that the intermediate states are admissible.

Also note that because the final state is admissible, we have, for each processor $u \in U$

$$K_u + \sum_{m \in S(u)} w_m - \sum_{m \in T(u)} w_m \geqslant 0. \tag{3}$$

Let c_m denote the cost of interrupting m, the PMP problem then formally consists, given a set of moves, in finding a pair (I,σ) such that $c(I) = \sum_{m \in I} c_m$ is minimum.

After a brief survey of the literature, we study the complexity of the PMP problem and exhibit some polynomially solvable special cases. We then present a branch-and-bound algorithm for the general case along with computational results demonstrating its practical relevance.

2. Related work

The literature related to the present problem is quite scarce.

Coffman et al. [9,10] seem to be the first to study a problem relatively close to ours which consists in scheduling, without preemption, a collection of large file transfers (between storage devices) so as to minimize the makespan of the overall transfer process. Each device is assumed to have the ability to communicate directly with the others. However, they consider only a *port constraint* on the devices, that is they impose a bound on the number of simultaneous file transfers a given device can engage in, and implicitly assume that the devices have infinite capacity.

Carlier [6,7] studies a problem of scheduling debt payments. Although the context obviously differs, this problem is quite close to the PMP problem. Given that each person has an initial capital as well as both debts and credentials, the *debt payment problem* asks for an admissible debt payment program, that is an ordering of the debt payments such that the capital of each person always remains positive and such that all the debts end up being paid. Carlier then shows that if a payment must be performed in one go then the problem of finding such a program or deciding that none exists is strongly *NP*-complete and exhibits a polynomial algorithm which solves the problem when this constraint is relaxed (i.e., when the debts are *breakable*). In fact, it is possible to interpret a debt between two persons as a process move between two processors[2] (from the source

[2] It follows that the *NP*-completeness of the PMP problem (Section 3) can also be established by restriction to the debt payment problem. However, the *NP*-completeness result in [6,7] does not allow to establish the *NP*-completeness of the PMP problem for a system with only two processors, in that sense Proposition 1 is a stronger result as far as the PMP problem is concerned.

processor, associated to the creditor, to the target one, associated to the debtor) but *not* vice versa. Indeed, in Carlier's model, there can be only one debt from one person to another but not the other way around (otherwise the two debts partially cancel leaving either one or no debt at all). Furthermore, the other notions involved in the definition of the PMP problem (e.g., the interruption of a process) do not really have a counterpart in the work of Carlier. Lastly, it should be emphasized that Carlier's algorithm for the breakable debt payment problem can be used to design a polynomial algorithm which solves the homogeneous case studied in Section 4.2, in the special case where the digraph underlying the instance is asymmetric (i.e., under the constraint that when some processes must be transferred from a processor A to another processor B, no process has to be transferred from B to A). Additionally, Carlier's algorithm, which is based on network flow techniques, is in essence very different from the algorithm presented in Section 4.2, which exploits strong connectivity and eulerianity properties.

Gavish and Liu Sheng [13] study the problem of dynamically optimizing the performances of distributed systems, such as computerized airline reservation systems, using dynamic migrations of files or database fragments in reaction to temporary changes in usage patterns. They also stress that neither their study nor most studies anterior to theirs have taken capacity constraints into account and that an assessment of the impact of such constraints on distributed file management policies is an important open issue.

More recently, Hall et al. [14], Saia [20], and Anderson et al. [2] have studied various flavours of a problem, referred to as the *data migration* problem, which consists in computing an efficient plan for moving objects stored on devices in a fully connected network from one configuration to another. On top of requiring that each device is involved in the transfer of only one object at a time, they explicitly consider capacity constraints on each of the devices and assume both that the objects have the same size and that there is at least one free space on each storage device in the initial as well as in the final configuration. Lastly, they also introduce the notion of *bypass node*, which is an extra storage device that can be used to store objects temporarily, and study the influence of allowing indirect migrations (via a bypass node) on the makespan of the reconfiguration.

Aggarwal et al. [1] introduce the *load rebalancing* problem which, given a suboptimal assignment of jobs to processors, asks to relocate a subset of the jobs so as to decrease the makespan, that is the load of the heaviest loaded processor. Among other results, they propose several efficient approximation algorithms for a variant of the problem which asks to achieve the best possible makespan under the constraint that no more than k jobs are relocated. They do not, however, have to consider capacity constraints on the processors as the system reconfiguration is performed by removing all the relocated jobs and by subsequently restarting them on the appropriate processors.

It turns out that the PMP problem is quite different from the above problems. In most of the aforementioned studies the objective is to minimize the duration of the reconfiguration under a set of constraints on the legal parallelism and, sometimes, only under quite loose capacity constraints. On the contrary, in the PMP problem we are interested only in minimizing the impact the reconfiguration has on service under multidimensional capacity constraints, although most of this paper considers the monodimensional case.

3. Complexity

In this section, we study the computational complexity of the PMP problem and show, perhaps not surprisingly, that it is *NP*-hard in the strong sense.

Given a set of moves, say M, we focus on the decision problem, hereafter referred to as the *Zero-Impact Process Move Programming* (ZIPMP) problem, which asks whether or not there exists a bijection $\sigma : M \to \{1, \ldots, |M|\}$ such that for all $m \in M$

$$w_m \leqslant K_{t_m} + \sum_{\substack{m' \in S(t_m) \\ \sigma(m') < \sigma(m)}} w_{m'} - \sum_{\substack{m' \in T(t_m) \\ \sigma(m') < \sigma(m)}} w_{m'}. \tag{4}$$

Recall that the 3-partition problem is the decision problem which asks, given a set E of $3k$ items, an upper bound $W \in \mathbb{N}$ and a size $s : E \to \mathbb{N}$ such that $\frac{W}{4} < s(e) < \frac{W}{2}$ for all $e \in E$ and such that $\sum_{e \in E} s(e) = kW$, whether or not there exists a partition of E into k disjoint sets $E_1, \ldots, E_k$ such that for all $1 \leqslant i \leqslant k$

$$\sum_{e\in E_i} s(e) = W.$$

It is well known (see for example [12]) that the 3-partition problem is *NP*-complete in the strong sense.

Proposition 1. *The ZIPMP problem is NP-complete in the strong sense, even for a system with only two processors.*

Proof. Let us consider a system composed of two processors, A and B, such that the set of process moves from A to B, denoted M_A, contains $k-1$ moves which satisfy $w_m = W \in \mathbb{N}$ and such that the set of process moves from B to A, denoted M_B, contains $3k$ moves satisfying $\frac{W}{4} < w_m < \frac{W}{2}$ and $\sum_{m\in M_B} w_m = kW$. Additionally, $K_A = W$ and $K_B = 0$ (see Fig. 2).

All we need to prove is that the above instance is a yes-instance if and only if there exists a partition of M_B into k disjoint sets $M_1, \ldots, M_k$ such that for all $1 \leqslant i \leqslant k$

$$\sum_{m\in M_i} w_m = W. \tag{5}$$

First suppose that such a partition does exist. It is then easy to construct a solution by first performing all the moves in any one of the M_i (this is possible since $K_A = W$) and this frees enough room on processor B to perform any one of the moves in M_A. After performing this step $k-1$ times, all the moves in M_A have been performed, so have the moves in all but one of the M_i's and there are W free units on A. Hence, by Eq. (5), the moves in the last of the M_i's are possible.

Conversely, let us suppose that such a partition does not exist. Let k' denote the greatest integer such that there exists $M_1, \ldots, M_{k'}$ disjoint sets which satisfy Eq. (5) for all $1 \leqslant i \leqslant k'$. Necessarily $k' < k-1$ (otherwise the non-existence assumption is falsified), hence it is possible to realize k' of the $k-1$ moves in M_A. Then W free units are available on A but since there exists no more set satisfying Eq. (5) it is only possible to transfer less than W units from B to A, it is therefore impossible to free enough room on B to perform another of the remaining moves in M_A.

Hence, the 3-partition problem can be solved by an algorithm able to solve the ZIPMP problem. The *NP*-completeness of the latter problem therefore follows by restriction to the 3-partition problem, itself *NP*-complete in the strong sense. □

The strong *NP*-hardness of the PMP problem directly follows from the above proposition. As a consequence, there neither exists a polynomial nor a pseudopolynomial algorithm for the PMP problem unless $P = NP$.

Lastly, it is interesting to note that the complexity result in [6,7] implies that the PMP problem remains strongly *NP*-hard even when the digraph underlying the instance is asymmetric i.e., when there is at most one process to transfer in between each (unordered) pairs of processors.

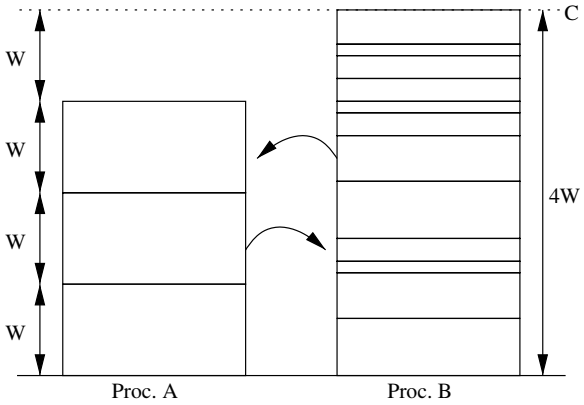

Fig. 2. Illustration of the kind of instances considered in the proof of Proposition 1.

4. Polynomially solvable special cases

This section is devoted to the study of two polynomially solvable special cases of the PMP problem.

To avoid any ambiguities we first recall a few basic notation and definitions regarding directed graphs. This terminology is borrowed from Bang-Jensen and Gutin [3]. Let $D = (V, A)$ denote a *directed multigraph* (that is parallel arcs are allowed but loops are forbidden). For a vertex $v \in V$, $N_D^+(v)$, $N_D^-(v)$, $d_D^+(v)$ and $d_D^-(v)$ respectively denote the *out-neighbourhood* (that is the set of vertices dominated by v), the *in-neighbourhood* (that is the set of vertices which dominate v), the *out-degree* (that is the number of arcs with tail v) and the *in-degree* (that is the number of arcs with head v) of v. A *walk* is an alternating sequence of vertices and arcs, say $v_1a_1v_2a_2v_3 \ldots v_{n-1}a_{n-1}v_n$, such that for $1 \leqslant i < n$ the tail of a_i is v_i and the head of a_i is v_{i+1}. A *closed walk* is a walk such that $v_1 = v_n$, a *trail* is a walk in which all arcs are distinct, a *path* is a trail in which all vertices are distinct and a *directed cycle* is a closed trail in which all vertices but the first and last are distinct (for short, the term *cycle* is used in the sequel).

Let M denote the set of process moves. To an instance of the PMP problem we associate a directed multigraph, denoted D and called the *transfer multigraph*, whose vertices are associated to the processors and such that an arc (s_m, t_m) is associated to each move $m \in M$. Given a transfer multigraph we also define the *transfer digraph*, denoted $\widetilde{D}$, as the directed graph obtained by contracting the parallel arcs in D.

4.1. Acyclic transfer digraphs

Our first concern is the case where the transfer multigraph is acyclic, without any constraint on the number of resources. Recall that every acyclic multigraph has a *topological ordering* of its vertices, that is there exists a bijection $\eta : V \to \{1, \ldots, |V|\}$ such that $\eta(v) < \eta(w)$ for all arcs $(v, w) \in A$.

Proposition 2. *If D is acyclic, a zero-impact process move program exists and can be found in linear time.*

Proof. By definition of a topological ordering $\eta^{-1}(|V|)$ has no out-neighbour. Equivalently, $S(\eta^{-1}(|V|)) = \emptyset$. Hence Eq. (3) becomes

$$\sum_{m \in T(\eta^{-1}(|V|))} w_m \leqslant K_{\eta^{-1}(|V|)},$$

which means that all the moves which target $\eta^{-1}(|V|)$ are possible.

Let $1 \leqslant i < |V|$, then, for all j such that $i < j \leqslant |V|$, assume that the moves in $T(\eta^{-1}(j))$ have been performed and that the corresponding arcs have been removed from D. Since, by definition of a topological ordering, $\eta^{-1}(i)$ can dominate only vertices $\eta^{-1}(j)$ with $i < j$, there is no arc with tail $\eta^{-1}(i)$ left in D. Equivalently, there remains no move with $\eta^{-1}(i)$ as source. Therefore, by Eq. (3), all the moves in $T(\eta^{-1}(i))$ can be performed and the corresponding arcs can be removed from D.

The claim follows from the well-known fact that a topological ordering can be obtained in linear time (e.g., [3]). □

Fig. 3 illustrates the resolution method. A topological ordering is $(5, 2, 7, 6, 1, 8, 3, 4)$, so the first set of moves performed (in an arbitrary order) is the set of moves which target vertex 4, then the move which targets vertex 3 is performed and so on.

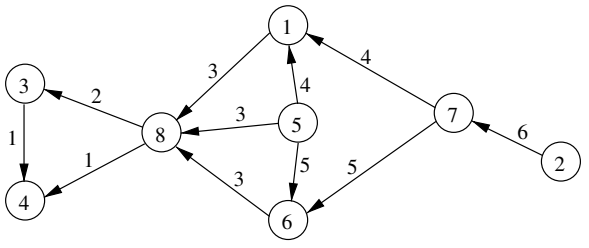

Fig. 3. Illustration of the resolution method for the acyclic case.

When D contains some cycles, it is still possible to derive a partial ordering of the process moves by looking at the strongly connected components of D. Recall that a directed multigraph is strongly connected either if $|V| < 2$ or if it contains a path from v to w and from w to v for each pair of distinct vertices v and w and that the strongly connected components of a directed multigraph are its maximal strongly connected subdigraphs.

Indeed, the following proposition suggests that the strongly connected components of D should be considered independently and in reverse topological order.

Proposition 3. *Let $C_1, \ldots, C_n$ denote the strongly connected components of D (assumed topologically ordered). Assume that given $1 < i \leqslant n$ the moves having both their source and target in $\bigcup_{j=i+1}^{n} C_j$ have been performed and that the corresponding arcs have been removed from D. Then a process move program which first schedules the moves having their source in C_i and target not in C_i, then the moves internal to C_i followed by the remaining moves, dominates any other program not satisfying this property.*

Proof. Since all the moves having both their source and target in $\bigcup_{j=i+1}^{n} C_j$ have been performed the vertices targeted by the moves having their source in C_i and target not in C_i are left without any out-neighbour. Hence, these moves are possible and performing such a move frees some resources on one of the vertices of C_i therefore easing the realization of the moves internal to C_i.

So assume that the moves having their source in C_i and target not in C_i have been performed. Performing a move, say m, having its source in $\cup_{j=1}^{i-1} C_j$ and target in C_i consumes some resources on one of the vertices of C_i. Hence, doing so before performing the moves internal to C_i can only harden the realization of these moves. Additionally, m is guaranteed to become possible after the moves internal to C_i have been either performed or interrupted (since the vertices in C_i are then left without out-neighbour).

Lastly, the realization of a move internal to $\cup_{j=1}^{i-1} C_j$ can be postponed as the realization of such a move neither eases nor hardens the realization of the moves internal to C_i and reciprocally. □

Fig. 4 illustrates the decomposition principle implied by the above proposition. First the moves targeting vertex *2* are performed in an arbitrary order, then the move targeting vertex *10*, then the moves internal to A, then the moves targeting vertices of A with their source in B, and so on.

Corollary 1. *Let $C_1, \ldots, C_n$ denote the strongly connected components of D (assumed topologically ordered), a process move program which interrupts a move such that $s_m \in C_i$ and $t_m \in C_j$ with $i \neq j$ is dominated.*

4.2. The homogeneous case

We now turn to the case where the consumption of each of the processes is equal to a constant, supposed equal to 1 without loss of generality.

Recall that a directed multigraph is eulerian if it is connected and if $d^+(v) = d^-(v)$ for all $v \in V$ and that such a multigraph possesses an eulerian tour, that is a closed trail which uses every arc exactly once.

First we have the following proposition.

Proposition 4. *If D is eulerian then the homogeneous case can be solved in linear time.*

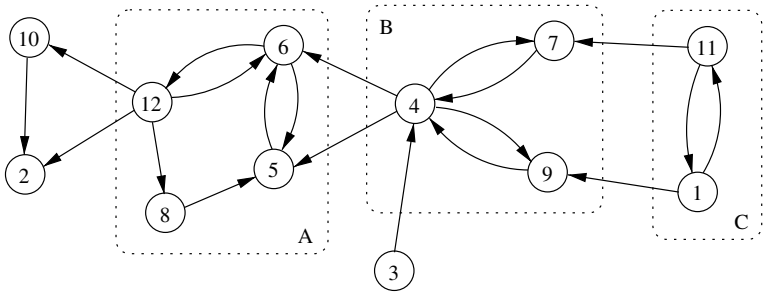

Fig. 4. Illustration of the decomposition principle implied by Proposition 3.

Proof. If there exists a processor $u \in U$ such that $K_u \geqslant 1$ then a zero-impact process move program is obtained by performing the moves in the reverse order of an eulerian tour on D, starting with any of the moves targeting u.

Otherwise, any one move m such that $c_m = \min_{m' \in M} c_{m'}$ is interrupted and, since this frees one unit on s_m, the remaining moves can be performed in the reverse order of an eulerian tour on D, starting with any of the moves targeting s_m and preceding m in the eulerian tour.

The claim follows from the well-known fact that an eulerian tour can be obtained in linear time (e.g., [3]). □

We now suppose that D is strongly connected and not eulerian and demonstrate that in this case a zero-impact process move program exists and can be found in polynomial time. We do so by studying Algorithm 1.

Algorithm 1. An algorithm for the homogeneous case when D is strongly connected and non-eulerian.

While $V \neq \emptyset$
  Let C denote the set of vertices in the last of the (topologically ordered) strongly connected components of D.
  (a) If C contain only one vertex, say v, then perform all the moves targeting v in an arbitrary order, remove them from M, remove the corresponding arcs from A and remove v from V.
  (b) Else choose a vertex, say v_0, in C whose remaining capacity is non-zero and a maximal eulerian subdigraph rooted at v_0, perform the moves in the subdigraph in the reverse order of an eulerian tour, removing them from M and removing the corresponding arcs from A.
End.

Lemma 1. *The moves performed at step (a) of Algorithm 1 are possible.*

Proof. The first time the loop is executed we have $C = D$ and, hence, no move satisfies the premises of step (a).

Otherwise, when D is no more strongly connected, v is left without any out-neighbour. Hence, Eq. (3) implies that all the moves which target v are possible. □

Lemma 2. *The first time step (b) of Algorithm 1 is executed, there exists a vertex v_0 in C such that $K_{v_0} > 0$.*

Proof. The first time step (b) of the algorithm is executed we have $C = D$. Since D is not eulerian there exists v_0 such that $d^+(v_0) \neq d^-(v_0)$. So either $d^+(v_0) < d^-(v_0)$ or $d^+(v_0) > d^-(v_0)$ in which case since $d^+(v_0) + \sum_{v \neq v_0} d^+(v) = d^-(v_0) + \sum_{v \neq v_0} d^-(v)$ we have $\sum_{v \neq v_0} d^+(v) < \sum_{v \neq v_0} d^-(v)$ and, by the pigeon-hole principle, there exists a vertex, say v_0' such that $d^+(v_0') < d^-(v_0')$. By Eq. (3), a vertex such that $d^+(v_0) < d^-(v_0)$ is such that $K_{v_0} \geqslant d^-(v_0) - d^+(v_0) > 0$. □

Lemma 3. *Each time step (b) of Algorithm 1 is executed, there exists a vertex v_0 in C such that $K_{v_0} > 0$.*

Proof. A strongly connected component is said to be *terminal* if it has no out-neighbour.

The lemma is established by demonstrating that, each time the loop is executed, the terminal strongly connected components of the remaining transfer multigraph either contain one vertex or contain a vertex, say v_0, such that $K_{v_0} > 0$.

Lemma 2 proves that it is initially the case.

Assume this is true at a given iteration of the algorithm.

Then if step (a) is executed new terminal strongly connected components may appear but all of these components are such that there exists a vertex v_0 with $K_{v_0} > 0$ (regardless of their cardinality). This is so because for each of the newly introduced components at least one move having its source and target respectively in and not in the component has been performed.

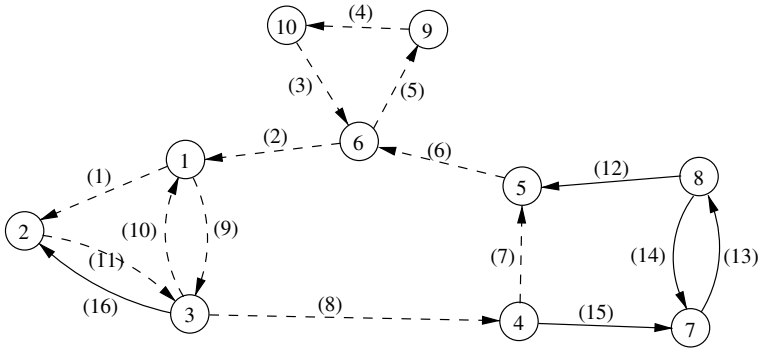

Fig. 5. Illustration of the functioning of Algorithm 1.

If step (b) is executed, then new terminal strongly connected components may appear but they all contain only one vertex. This is so because assuming otherwise would contradict the fact that the removed eulerian subdigraph was maximal for it would mean that at least one cycle encounters at least one vertex of the subdigraph. □

The following proposition is an immediate consequence of Lemmas 1 and 3.

Proposition 5. *If D is non-eulerian and strongly connected, Algorithm 1 outputs a zero-impact process move program.*

We are now able to solve the homogeneous case.

Corollary 2. *Assume that D is connected*[3] *then, unless D is eulerian and $K_u = 0$ for all $u \in U$, a zero-impact admissible process move program exists and can be found in polynomial time.*

Proof. If D is eulerian then we proceed as in the proof of Proposition 4. So let us assume that D is connected and not eulerian and let $C_1, \ldots, C_n$ denote its strongly connected components (topologically ordered). Algorithm 1 considers the strongly connected components of D as implied by Proposition 3. Assume that $|C_n| > 1$. If the transfer multigraph, say D'_n, associated to the moves internal to C_n is not eulerian then Proposition 5 shows how to find a zero-impact process move program. Otherwise if D'_n is eulerian then $d^+_{D'_n}(v) = d^-_{D'_n}(v)$ for all vertices of D'_n however since D is connected then at least one vertex in C_n, say v_0, is the head of an arc whose tail is not in C_n it follows that $d^-_D(v_0) > d^+_D(v_0)$ and, hence, that $K_{v_0} > 0$. This provides a vertex from which an eulerian tour can be started.

When the moves internal to C_i ($i < n$, $|C_i| > 1$) are considered then, since D is connected, at least one move with source in C_i and target not in C_i has been performed, therefore ensuring that one unit of load is free on at least one of the vertices of C_i. Let D'_i denote the transfer digraph associated to the moves internal to C_i. It follows that a zero-impact process move program is given either by an eulerian tour (if D'_i is eulerian) or by Proposition 5 otherwise.

The claim follows from the fact that Algorithm 1 is clearly polynomial. □

Fig. 5 illustrates the functioning of the algorithm. Initially, a maximal eulerian subdigraph rooted at *2* is chosen (dashed arcs). This is so because $d^+(2) < d^-(2)$. The moves are then performed in the reverse order of an eulerian tour on the subdigraph. After, this initial step, the remaining graph has two connected components ($\{2,3\}$ and $\{4,5,7,8\}$) which can be considered independently. The latter is considered first on the example. It has 3 strongly connected components ($\{4\}$, $\{7,8\}$ and $\{5\}$, in topological order). So $\{5\}$, the last, is considered first and the move from *8* to *5* is scheduled, which frees one unit on *8* which is chosen as the root of the small maximal eulerian subdigraph (*7* could have been chosen as well because $d^+(7) < d^-(7)$). The remaining graph is acyclic so we are done.

[3] If this assumption is not satisfied, then the argument can be repeated for each of the connected components of D.

5. A branch-and-bound algorithm

In this section, we present a branch-and-bound algorithm for the PMP problem. The algorithm initially starts with the worst possible solution, which consists in interrupting all the moves. Then an admissible program is built, each branching decision consisting in choosing an interrupted process to concatenate to the program ordering, among those for which doing so preserves the admissibility of the program. A leaf is obtained when no such process exists. This scheme is complemented by a lower bound as well as dominance relations.

We first describe each of the algorithm building blocks separately and then sketch how to integrate them in a practical branch-and-bound algorithm. Section 6 reports on computational results.

5.1. Branching scheme

A node of the search tree is denoted by as a quadruplet $N = (I, J, \sigma_J, R)$ where I, J and R respectively denote the sets of moves which are interrupted, ordered or yet neither interrupted nor ordered and where $\sigma_J : J \to \{1, \ldots, |J|\}$ is an ordering of the moves in J.

For such a quadruplet to define an admissible node, it is required that the sets I, J and R are both mutually exclusive (that is $I \cap J = I \cap R = J \cap R = \emptyset$) and collectively exhaustive (i.e., $I \cup J \cup R = M$) as well as for $(I \cup R, \sigma_J)$ to be an admissible process move program. Stated in plain English, this latter requirement expresses the fact that as long as the moves in $I \cup R$ are interrupted, the moves in J can be performed according to σ_J without inducing any violation of the capacity constraints.

Given a node N and a processor u, let

$$\ell_u(N) = \min_{i=1,\ldots,|J|} \left(K_u + \sum_{m \in S(u) \cap (I \cup R)} w_m + \sum_{\substack{m \in S(u) \cap J \\ \sigma_J(m) \leqslant i}} w_m - \sum_{\substack{m \in T(u) \cap J \\ \sigma_J(m) \leqslant i}} w_m \right) \tag{6}$$

and

$$L_u(N) = K_u + \sum_{m \in S(u)} w_m - \sum_{m \in T(u) \cap J} w_m. \tag{7}$$

Informally, $\ell_u(N)$ is the minimum remaining capacity of u during the execution of $(I \cup R, \sigma_J)$ and $L_u(N)$ is the remaining capacity of u after the execution of $(I \cup R, \sigma_J)$.

Proposition 6. *Let $N = (I, J, \sigma_J, R)$ be a node of the search tree and let $m \in R$, if $w_m \leqslant \ell_{s_m}(N)$ then $N' = (I, J \cup \{m\}, \sigma_{J \cup \{m\}}, R \backslash \{m\})$ is an admissible node for the search tree, where $\sigma_{J \cup \{m\}}$ is an ordering of the moves in $J \cup \{m\}$ such that $\sigma_{J \cup \{m\}}(m') = \sigma_J(m')$ for all $m' \in J$ and $\sigma_{J \cup \{m\}}(m) = |J| + 1$.*

Proof. By definition of ℓ_u, the fact that $w_m \leqslant \ell_{s_m}(N)$ implies that the process associated to m can remain on s_m during the entire execution of the program $(I \cup R, \sigma_J)$. After its execution, the remaining capacity on t_m is equal to

$$L_{t_m}(N) = K_{t_m} + \sum_{m' \in S(t_m)} w_{m'} - \sum_{m' \in T(t_m) \cap J} w_{m'}$$

and, from Eq. (3), we have

$$K_{t_m} + \sum_{m' \in S(t_m)} w_{m'} - \sum_{m' \in T(t_m) \cap J} w_{m'} \geqslant \sum_{m' \in T(t_m) \cap (I \cup R)} w_{m'} \geqslant w_m.$$

Hence, after all the moves in J have been performed, there is enough capacity on t_m to host the process associated to m. □

Note that the following relationships hold

$$\ell_{s_m}(N') = \ell_{s_m}(N) - w_m, \tag{8}$$

$$L_{t_m}(N') = L_{t_m}(N) - w_m, \tag{9}$$

$$\ell_{t_m}(N') = \min(\ell_{t_m}(N), L_{t_m}(N')). \tag{10}$$

Our branching scheme can then be stated as follows. The root node is $(\emptyset, \emptyset, \sigma_\emptyset, M)$ and is associated to the process moves program $(M, \sigma_\emptyset)$ which interrupts all the moves. At a node $N = (I, J, \sigma_J, R)$ of the search tree, let $I' = \{m \in R : w_m > \ell_{s_m}(N)\}$. By definition of ℓ_u, a process associated to a move m in I' cannot remain on s_m during the execution of $(I \cup R, \sigma_J)$ without inducing a violation of the capacity constraints. Hence, a move in I' cannot be added to J and concatenated to σ_J, and it will remain so in the branch rooted at N since ℓ_u is a non-increasing function of $|J|$ (from Eqs. (8) and (10)). It follows that for each $m \in R \backslash I'$ the nodes $N' = (I \cup I', J \cup \{m\}, \sigma_{J\cup\{m\}}, R\backslash(I' \cup \{m\}))$ are generated.

Hence, when branching from a node, the number of ordered moves is increased by one whereas the number of interrupted moves is increased by a number in $\{0, \ldots, |R| - 1\}$.

5.2. Lower bounds

At a node $N = (I, J, \sigma_J, R)$, let $\mathrm{KP}(u)$ denote the value of an optimal solution to the following knapsack problem

$$\begin{cases} \text{Maximize} & \displaystyle\sum_{m\in S(u)\cap R} c_m x_m & \text{(11a)} \\ \text{s.t.} & \displaystyle\sum_{m\in S(u)\cap R} w_m x_m \leqslant \ell_u(N), & \text{(11b)} \\ & x_m \in \{0,1\}, m \in S(u) \cap R. \end{cases} \tag{11}$$

We refer the reader to Kellerer et al. [17] for details regarding the knapsack problem.

Proposition 7. *A lower bound on the values of the solutions which can be obtained by exploring the branch rooted at N is provided by*

$$\mathrm{LB}(N) = \sum_{m\in I} c_m + \sum_{u\in U} \mathrm{LB}(u), \tag{12}$$

where $\mathrm{LB}(u) = W_u - \mathrm{KP}(u)$ *and* $W_u = \sum_{m\in S(u)\cap R} c_m$.

Proof. Since ℓ_u is a non-increasing function of $|J|$, the sum of the weights of the moves in $R \cap S(u)$ which can further be concatenated to σ_J cannot exceed ℓ_u. This is captured in the knapsack constraint (11b). Hence, $\mathrm{KP}(u)$ provides an upper bound on the sum of the costs of the moves in $R \cap S(u)$ which can further be concatenated to σ_J. □

Fortunately, the knapsack problem is one of the easier *NP*-hard problems (see [19] for a recent survey regarding the relative easiness of the knapsack problem) and, in particular, it can be solved in pseudopolynomial time. For example, lower bound (13) can be obtained in $O\left(\sum_{u\in U} \mid S(u) \cap R \mid \ell_u(N)\right)$ using the well-known Bellman recursion [5]. Moreover, if the results of the individual knapsack problems are memorized at each depth, computing the bound at a given depth requires solving only two knapsack problems: one for the source and one for the target processor of the last move in the schedule.

When the size of the coefficients prevents the use of dynamic programming, a tight upper bound on $\mathrm{KP}(u)$ can be obtained using any FPTAS[4] for the knapsack problem leading to a slightly weaker lower bound (see for example [16]).

[4] Recall [17] that given $\varepsilon \in]0,1[$, an ε-approximation scheme for a maximization problem is an algorithm which produces solutions of value greater than or equal to $(1-\varepsilon)\mathrm{OPT}(I)$ for all instances I of the problem. A Fully Polynomial Time Approximation Scheme (FPTAS) is an ε-approximation scheme whose running time is polynomial in the natural size of the instance as well as in $\frac{1}{\varepsilon}$.

Also, computationally cheaper, but weaker, lower bounds can be obtained from any upper bound for problem, the so-called Dantzig bound obtained by solving the linear relaxation of the knapsack problem would be an example. Note that when $c_m = w_m$, problem becomes a subset sum problem leading to the following lower bound

$$\mathrm{LB}'(N) = \sum_{m\in I} w_m + \sum_{u\in U} \max(0, W_u - \ell_u(N)).$$

Lastly, LB(N) can be generalized to the multiple resource case. Problem then becomes a multidimensional knapsack problem which is still reasonable to tackle using dynamic programming for a small enough number of resources (say less than or equal to 3). When the number of resources increases, however, it is likely that only upper bounds on KP(u) will be available. The reader is referred to Kellerer et al. [17] for details on how to solve the multidimensional knapsack problem using dynamic programming as well as on how to obtain upper bounds.

5.3. Dominance relations

The following lemma is stated without proof.

Lemma 4. *If $a \geqslant c$ and $b \geqslant d$ then $\min(a,b) \geqslant \min(c,d)$.*

Proposition 8. *Let $N_1 = (I_1, J_1, \sigma_{J_1}, R_1)$ and $N_2 = (I_2, J_2, \sigma_{J_2}, R_2)$ be two nodes of the search tree, then N_1 dominates N_2 if the following conditions hold*:

1. $R_1 = R_2 = R$.
2. $\sum_{m\in I_1} c_m \leqslant \sum_{m\in I_2} c_m$.
3. $L_u(N_1) \geqslant L_u(N_2)$, $\forall u \in U$.
4. $\ell_u(N_1) \geqslant \ell_u(N_2)$, $\forall u \in U$.

Proof. Let $N_2^\star = (I_2 \cup I^\star, J_2 \cup J^\star, \sigma_{J_2\cup J^\star}, \emptyset)$ denote the best leaf of the branch rooted at N_2 and let $m = \sigma^{-1}_{J_2\cup J^\star}(|\, J_2 \,| + 1)$ (assuming $|J^\star| \geqslant 1$).

Let $N_2^{(m)} = (I_2, J_2 \cup \{m\}, \sigma_{J_2\cup\{m\}}, R \setminus \{m\})$, since $\ell_u(N_1) \geqslant \ell_u(N_2)$ for all $u \in U$ the node $N_1^{(m)} = (I_1, J_1 \cup \{m\}, \sigma_{J_1\cup\{m\}}, R \setminus \{m\})$ is admissible. Using Condition 4 and Eq. (8) we have

$$\ell_{s_m}(N_1^{(m)}) = \ell_{s_m}(N_1) - w_m \geqslant \ell_{s_m}(N_2) - w_m = \ell_{s_m}(N_2^{(m)}).$$

Using Condition 3 and Eq. (9) we have

$$L_{t_m}(N_1^{(m)}) = L_{t_m}(N_1) - w_m \geqslant L_{t_m}(N_2) - w_m = L_{t_m}(N_2^{(m)}). \tag{13}$$

Lastly, using Condition 4, Eqs. (10) and (13) as well as Lemma 4 we have

$$\ell_{t_m}(N_1^{(m)}) = \min(\ell_{t_m}(N_1), \quad L_{t_m}(N_1^{(m)})) \geqslant \min(\ell_{t_m}(N_2), \quad L_{t_m}(N_2^{(m)})) = \ell_{t_m}(N_2^{(m)}).$$

Hence, for all $u \in U$ we have $L_u(N_1^{(m)}) \geqslant L_u(N_2^{(m)})$ as well as $\ell_u(N_1^{(m)}) \geqslant \ell_u(N_2^{(m)})$.

The above argument can be applied iteratively until the node $N_1^\star = (I_1, J_1 \cup J^\star, \sigma_{J_1\cup J^\star}, I^\star)$ is obtained. Then the best leaf of the branch rooted at N_1 has value at most equal to

$$\sum_{m\in I_1} c_m + \sum_{m\in I^\star} c_m,$$

which is, by Condition 2, smaller than or equal to $\sum_{m\in I_2} c_m + \sum_{m\in I^\star} c_m$. □

The dominance relation of Proposition 8 generalizes several other relations.

Provided that many equivalent total orderings of a set of non-interrupted moves can be obtained by combining a given set of per-processor orderings, it is expected that a significant amount of redundancy can be removed from the search tree by considering the following special case of the dominance relation of

Proposition 8. Consider two nodes $N_1 = (I, J, \sigma_J^{(1)}, R)$ and $N_2 = (I, J, \sigma_J^{(2)}, R)$. If $\sigma_J^{(1)}$ and $\sigma_J^{(2)}$ are such that, for all $u \in U$, the ordering of the moves in $J \cap (S(u) \cup T(u))$ induced by $\sigma_J^{(1)}$ is equivalent to the one induced by $\sigma_J^{(2)}$ then N_1 dominates N_2 and reciprocally. This is so because $L_u(N_1) = L_u(N_2)$ and $\ell_u(N_1) = \ell_u(N_2)$ for all $u \in U$.

The strong-connectivity-based dominance relation discussed in Section 4.1 is also taken into account by the rule of Proposition 8. For example, consider two nodes $N_1 = (I, J, \sigma_J^{(1)}, R)$ and $N_2 = (I, J, \sigma_J^{(2)}, R)$. Then for $i = 1, \ldots, |J|$ let $m = \sigma_J^{(1)-1}(i)$ and let $C_n \subseteq U$ denote the last (topologically ordered) strongly connected component of the transfer digraph induced by the moves in $\{m' \in J : \sigma(m') \geqslant i\}$. Assuming that $\sigma_J^{(1)}$ and $\sigma_J^{(2)}$ induce equivalent orderings of the moves in C_n, if m is always internal to C_n when $|C_n| > 1$ then we have $L_u(N_1) = L_u(N_2)$ as well as $\ell_u(N_1) \geqslant \ell_u(N_2)$ for all $u \in U$. Hence N_1 dominates N_2.

5.4. Subproblem selection

Subproblem selection is performed in a greedy fashion. At a node $N = (I, J, \sigma_J, R)$ of the search tree, the immediate profit associated to the decision of using a move $m \in R$ such that $w_m \leqslant \ell_{s_m}(N)$ for branching is defined as

$$p_m = c_m - (W_s - \mathrm{KP}_s - \mathrm{LB}(s_m)) - (W_t - \mathrm{KP}_t - \mathrm{LB}(t_m)),$$

where $W_s = \sum_{m' \in S(s_m) \cap R \setminus \{m\}} c_{m'}$, $W_t = \sum_{m' \in S(t_m) \cap R} c_{m'}$ and where KP_s and KP_t respectively denote the value of an optimal solution to knapsack problems

$$\begin{cases} \text{Maximize} & \displaystyle\sum_{m' \in S(s_m) \cap R \setminus \{m\}} c_{m'} x_{m'} \\ \text{s.t.} & \displaystyle\sum_{m' \in S(s_m) \cap R \setminus \{m\}} w_{m'} x_{m'} \leqslant \ell_{s_m}(N) - w_m, \\ & x_{m'} \in \{0,1\}, \quad m' \in S(s_m) \cap R \setminus \{m\} \end{cases}$$

and

$$\begin{cases} \text{Maximize} & \displaystyle\sum_{m' \in S(t_m) \cap R} c_{m'} x_{m'} \\ \text{s.t.} & \displaystyle\sum_{m' \in S(t_m) \cap R} w_{m'} x_{m'} \leqslant \min(\ell_{t_m}(N), L_{t_m}(N) - w_m), \\ & x_{m'} \in \{0,1\}, \quad m' \in S(t_m) \cap R. \end{cases}$$

The right-hand sides of the capacity constraints of the above two problems are justified by Eq. (8) as well as (9) and (10), respectively.

Hence, the increment in the lower bound is taken into account when evaluating branching decisions, the moves inducing the biggest immediate profits being used for branching first.

Note that this subproblem selection scheme can be used as the basis of a simple pseudopolynomial greedy algorithm for the PMP problem.

5.5. Putting it all together

We have implemented a DFS branch-and-bound algorithm based on the ideas discussed in the previous sections, namely lower bound (13), the dominance relations of Proposition 8 as well as the subproblem selection strategy of Section 5.4.

The resolution of the knapsack problems involved in both the calculation of lower bound (13) and the subproblem selection scheme is performed using the Bellman Algorithm (see for example [17]).

The exploitation of the dominance relation of Proposition 8 deserves more comments.

Indeed, there are three main ways of exploiting dominance relations within a branch-and-bound algorithm:

1. Exclude a node from consideration if it is dominated by a node which *has already been* considered (e.g., [15]).

2. Exclude a node from consideration if there exists a node which dominates it, *regardless of whether or not* the latter has already been considered (e.g., [4]).
3. *Replace* a node by another node which dominates it, if such a node exists and can be found (e.g., [8]).

All of these strategies have pros and cons. Strategy 1 requires memorizing (at least partially) the set of nodes considered so far and may result in the exploration of redundant branches: for example if the branching procedure considers N_1 before N_2 and if N_2 dominates N_1. Strategy 2 does not require memorizing the set of nodes considered so far (as long as the dominance relation has been supplemented so as to guarantee unicity) but may result in delaying the improvement of the upper bound: for example if the branching procedure considers nodes N_1, N_2 and N_3 (in that order) and if N_3 dominates N_1 then the algorithm explores only the branches rooted at N_2 and N_3 it is however possible that exploring the branch rooted at N_1 improves the upper bound enough so that there is no need to consider N_2, so it comes down to whether it is computationally more interesting to explore the branch rooted at N_1 and the branch rooted at N_3 (despite of the fact that it is known to be redundant) or the branches respectively rooted at N_2 and N_3. Lastly, strategy 3 requires memorizing (at least partially) the set of nodes considered so far but, thanks to the fact that replacement is performed, it avoids both redundancy and delayed upper bound improvement, it however requires being able to find dominating nodes from a given node and this problem might be as hard as the problem the branching procedure is solving.

As long as the memory is managed efficiently, memorizing the set of nodes considered so far is not an issue: if the branching procedure is to succeed it must not consider too many nodes and workstations nowadays usually have fairly huge amounts of memory. Additionally, it should be emphasized that the branching procedure discussed in this paper is not destined to be embedded in a real-time system, see the discussion in Section 7.

On empirical grounds, strategy 1 appears to be the most suited to exploit the dominance relation of Proposition 8. This is performed using a balanced binary search tree (see for example [18]) keyed on the binary representation of the set R of a node $N = (I, J, \sigma_J, R)$, each key being associated to a list of triplets $\{c(N), L(N), \ell(N)\}$. When a node is considered, the list associated to R is searched for a triplet which dominates the node. If such a triplet is found the branch rooted at the node is pruned. Otherwise, the branch is explored. Then the list is searched for triplets which are dominated by the triplet associated to the node, which are removed, and the latter is added at the front of the list.

6. Computational experiments

In this section, we report on computational experiments carried out so as to assess the practical relevance of the branch-and-bound algorithm of Section 5. These experiments have been performed on a Sun Ultra 10 workstation with a 440 MHz Sparc microprocessor, 512 MB of memory and the Solaris 5.8 operating system.

6.1. Instance generation

Given U the set of processors, C the processor capacity and W an upper bound on the process consumption, an instance is generated as follows.

First, the set of processes is built by drawing consumptions uniformly in $\{1, \ldots, W\}$ until $\sum_{p \in P} w_p \geqslant C \mid U \mid$. The initial state, f_i, is then generated by randomly assigning the processes to the processors: the processor to which a process is assigned is drawn uniformly from the set of processors whose remaining capacity is sufficient (note that not all processes necessarily end up assigned to a processor). The final state, f_f, is built in the very same way with the exception that only the processes which are assigned to a processor in the initial state are considered. An instance is considered valid only if all the processes assigned to a processor in the initial state are also assigned to a processor in the final state. Invalid instances are discarded and the construction process is repeated until a valid instance is obtained (the rejection rate depends on the parameters, as an example, coarse estimates for $|U| = 10$, $C = 100$ as well as $W = 10$ and $W = 50$ respectively are 29% and 41%). The set of moves is then built as explained in Section 1.

It should be emphasized that the above scheme generates instances for which the capacity constraints are extremely tight, instances which can be expected to be hard and, in particular, significantly harder than those

Table 1
Illustration of the performance impact of each of the algorithm components on a small set of moderate size instances (5 processors of capacity 100, processes weights drawn uniformly in {1,...,40})

N.	$\|M\|$	OPT	LB	Dom.			LB & dom.		
			#nodes	#nodes	#keys	#items	#nodes	#keys	#items
01	22	6	>18,500,000	>15,900,000	>316,729	>606,958	177,542	6738	7905
02	21	17	16,647,308	>15,500,000	>224,009	>454,493	189,618	7255	11,178
03	16	23	12,726	319,552	8905	16,232	2679	220	244
04	20	10	575,391	>16,100,000	>210,796	>510,507	34,829	2093	2573
05	17	26	1,243,750	488,432	10,821	22,253	23,635	1354	1968
06	19	25	265,197	13,217,379	136,421	480,749	29,891	1808	2162
07	18	5	14,972,721	5,876,920	66570	153,435	55,209	2685	4116
08	23	23	>23,600,000	>15,000,000	>334,966	>627,169	457337	18783	24,298
09	20	19	1,526,411	>15,700,000	>215,828	>481,464	55,045	2996	3611
10	17	47	143,800	1,609,022	38,846	86,350	25,814	1475	1971

occurring in practice. As an example, for $|U| = 10$, $C = 100$ and $W = 10$ only 1.28% of free capacity remains, on average, on each of the processors. However, for the system to which this work is to be applied (see [22]) the maximum *theoretical* load of a processor ranges (nonlinearly) from at most 50% (for a system with 2 processors) to at most around 93% (for a system with 14 processors, which is the maximum). This is so because some spare capacity is provisioned for fault tolerance purpose and this spare capacity is spread among all the processors. Additionally, it should be stressed that the system carries at most 100 processes and that a preprocessing technique, based on the fact that the properties of a system state are invariant by a permutation of the processors, is used to decrease the number of moves by around 25% on average. It turned out that our algorithm was able to solve virtually all practical instances within a few seconds and that, as a consequence, we had to design more aggressive instance generation schemes, such as the above, in order to push the algorithm to its limits.

Lastly, we have supposed that $c_m = w_m$, which is quite natural for our application as it is reasonable to assume that the amount of service provided by a process is proportional to the amount of resources it consumes.

6.2. Influence of the algorithm building blocks

For a small set of moderate size instances generated using the scheme of Section 6.1, Table 1 provides the number of nodes explored by the algorithm (''#nodes''), the number of entries in the binary search tree discussed in Section 5.5 (''#keys'') as well as the total number of items stored in it[5] (''#items''), that is the sum over the set of entries of the length of the associated list, when only the lower bound is activated (column ''LB''), when only the dominance relation is activated (column ''Dom.'') and when both the lower bound and the dominance relation are activated (column ''LB & Dom.'').

Table 1 illustrates that both the lower bound and the dominance relations significantly contribute to the reduction of the search space. It also illustrates the fact that the size of the data structure used to exploit the dominance relation grows mildly with the number of nodes.

6.3. Computational results

In order to reasonably explore the (practically relevant part of the) problem space we have used the scheme of Section 6.1 to generate a set of 10 instances for each $|U| \in \{2,\ldots,14\}$,[6] each $W \in \{10,20,\ldots,90,100\}$ and

[5] Because this quantity is measured at the end of the execution of the algorithm it provides only an order of magnitude. This is so because the algorithm tries to remove dominated triplets from a list each time a new triplet is added, as explained in Section 5.5.

[6] The choice for the values of $|U|$ is motivated by the fact that the system to which this work is to be applied contains at least *2* and at most *14* processors [22].

$C = 100$. Hence a total of 1300 instances, amongst which only 1020 were considered of non-trivial size (from around 10 up to 254 moves). For each of these sets of 10 instances, Table 2 indicates the average problem size (i.e., the average number of moves), denoted $\overline{|M|}$, as well as the number of instances in the set that the algorithm has been able to solve in less than 20 min, denoted n. Additionally, Table 3 provides for each value of $|U|$, the size of the biggest instance the algorithm was able to solve in less than 20 min, the size of the smallest instance the algorithm was *not* able to solve in less than 20 min as well as the size of the biggest instance on which the algorithm was tried.

Our intent, in performing this experiment, has been to obtain an idea, when the capacity constraints are extremely tight, on the kind of instances which are within the reach of the algorithm in a relatively short time for practically relevant values of $|U|$.

In the range $5 \leqslant |U| \leqslant 12$ the algorithm is able to solve most instances of size below or slightly above 40, generally in a fairly small fraction of the 20-min limit. In this range, the algorithm is also able to solve a bunch of fairly big instances, culminating in the resolution of an instance with 11 processors and 190 moves in a bit more than 3 min.

Instances in the range $2 \leqslant |U| \leqslant 4$ appear to be more difficult. This is presumably due to the fact that the difficulty ends up concentrated among the few processors. As an example, for $|U| = 2$, the algorithm failed to solve an instance with 22 moves and took a bit more than 7 min to solve another instance with only 20 moves.

Table 2
Average instance size, denoted $\overline{|M|}$, and number of instances solved in less than 20 min, denoted n, for each of the 10 instances sets generated

W	$\lvert U\rvert$	2		3		4		5		6		7		8	
		$\overline{\lvert M\rvert}$	n	$\overline{\lvert M\rvert}$	n	$\overline{\lvert M\rvert}$	n	$\overline{\lvert M\rvert}$	n	$\overline{\lvert M\rvert}$	n	$\overline{\lvert M\rvert}$	n	$\overline{\lvert M\rvert}$	n
10		17.3	9	37.3	1	54.4	4	73.1	2	86.8	4	110.1	4	125.8	2
20		8.2	10	19.5	10	26.7	9	35.0	4	46.7	6	56.5	4	64.1	2
30		6.4	10	12.9	10	19.5	10	23.9	10	30.4	9	37.2	8	44.6	3
40				9.9	10	12.5	10	19.3	10	22.9	10	28.1	9	33.9	8
50						10.6	10	12.9	10	19.5	10	22.0	10	25.9	10
60								13.2	10	14.6	10	18.1	10	21.4	9
70										13.3	10	15.2	10	18.6	10
80												11.9	10	15.4	10
90														12.9	10
		9		10		11		12		13		14			
10		150.1	2	159.2	0	179.5	3	198.5	0	215.8	0	237.6	0		
20		75.6	3	82.1	5	92.5	1	102.6	0	111.1	0	122.4	0		
30		47.2	5	56.7	2	64.6	2	71.2	1	77.5	0	80.6	0		
40		37.3	7	45.7	3	48.0	4	51.6	3	56.8	3	58.6	2		
50		30.1	8	33.5	8	37.8	5	41.8	4	43.7	5	53.0	0		
60		25.8	9	29.5	6	29.2	8	31.8	8	35.3	6	40.8	1		
70		22.1	9	23.2	9	25.7	8	28.1	9	32.2	4	36.3	4		
80		17.3	10	19.0	10	21.2	9	25.1	9	25.5	10	28.4	5		
90		16.3	10	18.9	9	20.8	10	23.6	10	22.8	8	26.4	7		
100		12.8	10	15.8	10	17.9	10	18.2	10	19.6	9	22.7	9		

Table 3
For each value of $|U|$, row "A" indicates the size of the biggest instances solved by the algorithm in less than 20 min, row "B" the size of the smallest instance *not* solved by the algorithm in less than 20 min and row "C" provides the size of the biggest instance on which the algorithm was tried

$\lvert U\rvert$	2	3	4	5	6	7	8	9	10	11	12	13	14
A	20	36	60	71	88	116	124	149	80	190	65	53	58
B	22	34	31	34	39	33	26	25	24	25	31	25	26
C	22	46	60	78	101	121	139	157	165	190	213	232	254

Also, in the range $13 \leqslant |U| \leqslant 14$, instances with extremely high cost optimal solutions start to appear. The algorithm seems to have difficulties in dealing with these instances as it failed to close a few relatively small instances (see Table 3) or required an important fraction of the allowed 20 min to solve a few other such instances. As an example, an instance with 13 processors and 24 moves was solved in a bit more than 8 min, this instance required the interruption of nearly 16% of the moved payload. Having said that, the practical relevance of these instances may be challenged as systematically having instances with high cost optimal solutions would be a con against embedding a reconfiguration procedure such as the present one within the design of a system. At the end of the day, what really matters is whether or not the amount of payload usually impacted by the reconfiguration is acceptable (typically below a few percent).

Lastly, it should be emphasized that when W is small enough (typically less than or equal to 30), small cost solutions almost always exist and can be found by the algorithm, generally within a small fraction of the 20-min limit. For example, with $|U| = 14$ and $W = 10$, the algorithm terminated with solutions situated, on average, at less than 1.2% from an hypothetical zero cost solution (given a solution of value z, distance to optimality was measured using the ratio $d(z) = \frac{z-\mathrm{OPT}}{S-\mathrm{OPT}}$, where OPT and $S = \sum_m c_m$ respectively denote the value of an optimal solution and of the worst possible one, which simply consists in interrupting all the moves,[7] when unknown OPT was replaced by a lower bound e.g., 0). Overall, on the set of instances with $W \leqslant 30$ which the algorithm failed to solved in less than 20 min, solutions situated, on average, at 2.07% from an hypothetical zero cost solution were obtained.

Overall, 659 of the 1020 "hard" instances have been solved.

7. Conclusion

In this paper, we have introduced the Process Move Programming problem which consists, starting from an arbitrary initial process distribution on the processors of a distributed system, in finding the least disruptive sequence of operations (non-impacting process migrations or temporary process interruptions) at the end of which the system ends up in another predefined arbitrary state. The main constraint is that the capacity of the processors must not be exceeded during the reconfiguration. This problem has applications in the design of high availability real-time distributed switching systems such as the one discussed in [22].

We have shown that the PMP problem is *NP*-hard in the strong sense and exhibited some polynomial special cases, the most notable of which being the homogeneous case where all the processes have a constant consumption in a unique resource.

We have proposed a branch-and-bound algorithm for the general case. From an industrial perspective, it can be considered that the PMP problem is solved by this algorithm as it is able to close virtually all practical instances within a few seconds. Additionally, we have performed computational experiments demonstrating the algorithm's perspective when used to solve instances significantly harder than those occurring in practice, in terms both of size and tightness of the capacity constraints. Indeed, our algorithm was able to solve more than 64% of our such test instances within a 20-min limit, including some instances with more than 100 moves. Also, our experiments suggest that the truncated version of the algorithm has fairly reasonable heuristic capabilities.

Nevertheless, our branch-and-bound procedure is not destined to be embedded in a real-time system. This is so mainly because the behaviour of such an algorithm may be quite sensitive to changes in the kind of instances it is asked to solve. Hence, the main purpose of our algorithm is to allow building a database of instances with known optimal solutions so as to empirically assess the quality of the solution obtained using efficient approximate resolution algorithms suitable for use in a real-time context. Efficient approximate resolution algorithms for the PMP problems are presently discussed in [21].

[7] This measure is quite natural as $1 - d(z)$ can be interpreted either as a *differential approximation ratio* (recall that differential approximation is concerned with how far the value of a solution is from the worst possible value [11]) or as a *conventional approximation ratio* [12] for the maximization problem complementary to the PMP problem which asks to maximize the sum of the costs of the moves which are *not* interrupted.

Acknowledgement

The authors wish to thank the anonymous referee for several suggestions that led to improvements in the paper.

References

[1] G. Aggarwal, R. Motwani, A. Zhu, The load rebalancing problem, in: Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures, 2003, pp. 258–265.
[2] E. Anderson, J. Hall, J. Hartline, M. Hobbs, A.R. Karlin, J. Saia, R. Swaminathan, J. Wilkes, An experimental study of data migration algorithms, in: Proceedings of the 5th International Workshop on Algorithm Engineering, Lecture Notes in Computer Science, Springer, 2001, p. 145.
[3] J. Bang-Jensen, G. Gutin, Digraphs—Theory, algorithms and applications, Springer-Verlag, 2002.
[4] P. Baptiste, J. Carlier, A. Jouglet, A branch-and-bound procedure to minimize total tardiness on one machine with arbitrary release dates, European Journal of Operational Research 158 (2004) 595–608.
[5] R. Bellman, Dynamic Programming, Princeton University Press, 1957.
[6] J. Carlier, Le problème de l'ordonnancement des paiements de dettes, RAIRO—Operations Research 18 (1) (1984).
[7] J. Carlier, Problèmes d'ordonnancement à contraintes de ressources: algorithmes et complexité, Méthodologie & Architecture des Systèmes Informatiques, vol. 40, Université P. et M. Curie et CNRS, 1984.
[8] J. Carlier, P. Chrétienne, Problèmes d'ordonnancements: modélisation, complexité et algorithmes, Études et Recherches en Informatique, Masson, 1988.
[9] E.G. Coffman, M.R. Garey, D.S. Johnson, A.S. Lapaugh, Scheduling file transfers in distributed networks, in: Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing, 1983, pp 254–266.
[10] E.G. Coffman, M.R. Garey, D.S. Johnson, A.S. Lapaugh, Scheduling file transfers, SIAM Journal on Computing 14 (3) (1985).
[11] M. Demange, V.T. Paschos, On an approximation measure founded on the links between optimization and polynomial approximation theory, Theoretical Computer Science 158 (1996) 117–141.
[12] M.R. Garey, D.S. Johnson, Computers and Intractability—A Guide to the Theory of NP-Completeness, W.H. Freeman and Company, 1979.
[13] B. Gavish, O.R. Liu Sheng, Dynamic file migration in distributed computer systems, Communications of the ACM 33 (1990) 177–189.
[14] J. Hall, J. Hartline, A. R. Karlin, J. Saia, J. Wilkes, On algorithms for efficient data migration, in: Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms, 2001, pp. 620–629.
[15] T. Ibaraki, The power of dominance relations in branch-and-bound algorithms, Journal of the ACM 24 (2) (1977) 264–279.
[16] H. Kellerer, U. Pferschy, Improved dynamic programming in connection with an FPTAS for the knapsack problem, Journal of Combinatorial Optimization 3 (1999) 59–71.
[17] H. Kellerer, U. Pferschy, D. Pisinger, Knapsack Problems, Springer, 2004.
[18] D.E. Knuth, Sorting and Searching, International Network Optimization Conference, second ed.The Art of Computer Programming, vol. 3, Addison-Wesley, 1998.
[19] D. Pisinger, Where are the hard knapsack problems? Computers & Operations Research 32 (2005) 2271–2284.
[20] J.C. Saia, Data migration with edge capacities and machine speeds, Technical report, University of Washington, 2001.
[21] R. Sirdey, J. Carlier, D. Nace, Approximate resolution of a resource-constrained scheduling problem, Technical Report PE/BSC/INF/016550 V01/EN, Service d'architecture BSC, Nortel GSM Access R& D, France, submitted for publication.
[22] R. Sirdey, D. Plainfossé, J.-P. Gauthier, A practical approach to combinatorial optimization problems encountered in the design of a high availability distributed system, in: Proceedings of International Network Optimization Conference, 2003, pp. 532–539.