Increased Model-Realism can Make Processor-Selection Easier

N. Peter Drakenberg

Deutsches Klimarechenzentrum GmbH
Bundesstraße 45a
D-20146 Hamburg, Germany

Abstract. A formalization of the processor-set selection problem for parallel job-schedulers is presented and proven to be NP-hard in the strong sense. Nonetheless, a simple algorithm for the problem is presented, and is seen to perform well in practice when used in combination with more realistic, less uniform, cost-structures.

Keywords: parallel job scheduling, interconnection networks, topology awareness, resource allocation, processor-set selection.

1 Introduction

The problem of determining *when* and *where* to run a parallel program on a parallel computer is usually referred to as job scheduling, and is but one incarnation of scheduling and resource allocation problems which recur at a variety of levels and scales in the context of parallel processing.

Parallel programs running on parallel computers are susceptible to run-time delays due to communications congestion, and it therefore benefits each program (and other simultaneously running programs) if its communicating subtasks are mapped to processors that are located close together in the communication topology of the computer system being used. For reasons of efficiency/performance, a parallel program's subtasks should thus be mapped to processors such that intensively inter-communicating pairs of subtasks are as closely located as possible and vice versa, and a variety of methods to do so have been developed over the last 25 years [3–8, 36].

Unfortunately, current parallel job-schedulers tend to assign programs to run on more-or-less randomly assembled collections of processors [26, 34], and in such cases a carefully determined mapping of program subtasks to logical processors is essentially worthless. Making parallel job-schedulers select processor collections that are closely located (*i.e.*, 'compact') in the communication topologies of parallel computer systems is not only an efficiency/performance issue, however. For sites where single jobs never (or very very rarely) use an entire system, efficient and reliable selection of compact processor collections by the job-scheduler would enable hardware configurations with lower bisection bandwidths to remain competitive for the workloads in question, and could thereby strongly influence system procurement costs (and thus procurement decisions).

In this paper we present a flexible formalization of the processor-set selection problem for parallel job-schedulers. We show that the resulting optimization problem is NP-hard, and present results suggesting that the use of more realistic system models, with less homogenous cost-structures, may improve the quality of processor-set selections obtained for practically occurring problem instances.

2 Related Work

Parallel job scheduling and topology aware task mapping are well known problems that have been extensively studied since the early 1990s and early 1980s, respectively. Each field has an extensive collection of litterature associated with it, and the coverage in subsections below is necessarily very brief.

2.1 Parallel Job Scheduling

The dominant scheme for scheduling parallel jobs on parallel computers is known as *variable partitioning* [14], the meaning of which is that when scheduled to run, each job is assigned a set of processors (*i.e.*, a partition) of the requested size that it keeps and uses throughout its lifetime. In early parallel job schedulers partitions were allocated strictly in a first-come first-served (FCFS) manner to submitted jobs, with the result that system utilization most typically was below 60 % [18]. A widely used refinement of the scheme just described is to also apply a technique known as *backfilling* [23], whereby jobs that are not first in line to be started are nonetheless started, when doing so is possible without affecting the expected starting-time of the job that is currently first in line to be started. Through backfilling and other improvements, such as ordering jobs by priority rather than strictly FCFS, it has become entirely realistic to achieve utilization figures above 80 % [18, 25, 30].

All currently used parallel job schedulers (*e.g.*, PBS, LSF, LoadLeveler, SLURM, Sun/Oracle GE) use variable partitioning schemes with backfilling and order waiting jobs by priority. Additional features such as fair-share scheduling, consumable resources, and job preemption are also supported in many cases. Strategies for choosing the actual subsets of nodes to use for each job, are however essentially limited to selecting the least loaded nodes, selecting available nodes in some constant sequential order, or selecting nodes such that the number of consecutive sets of nodes is minimized. However, due to fragmentation and boundary effects, these strategies tend not to yield particularly encouraging results for recent generations of parallel computer systems.

2.2 Topology Aware Task Mapping

Substantial theoretical and practical research on interconnect topologies and topology-aware mapping of tasks to processors was performed from the early 1980s to the mid 1990s. Heuristic techniques such as pairwise exchange were initially suggested [6, 21], but subsequently found not to scale well. Instead,

methods were developed based on recursive partitioning [13] and graph contraction [3]. Yet other methods were developed based on techniques such as simulated annealing [7] and genetic algorithms [8]. The mapping problems considered were not always constrained to having predefined tasks. Mapping of recurrence relations [29] and loop iterations [11] onto regularly connected grids were other aspects of mapping being studied.

Due to the deployment in the mid 1990s and onwards of virtual cut-through and wormhole routing [19, 27] and the emergence of faster interconnects, message latencies became relatively unimportant and research in topology aware mapping of computations died down. However, with the reemergence of three-dimensional torus networks in recent top-of-the-line supercomputers [1, 10, 17], message latencies are again gaining in significance and interest in topology aware task mapping is increasing [4, 5, 26, 36].

Processor-set selection by job-schedulers has also been investigated previously [2, 33, 34], but to our knowledge only for mesh and torus topologies, whereas hierarchical topologies have received very little attention.

3 Problem Formulation and Notation

There exists a vast variety of communication network topologies for parallel computing, but because of the dominance of flat and hierarchically structured network topologies (*e.g.*, Clos networks [9] and fat-trees [22]) in current and recent Top500 rankings,[1] we only consider such network topologies in the present paper. For a fat-tree network with 256 compute nodes, overall performance differences of 200–400% due to bad processor selections have been reported [26], and our interest in flat and hierarchical topologies is thus not misplaced.

3.1 A Concrete Sample-System

Current high-performance computer systems are typically composed of several hundred or thousand compute-nodes, connected to one another by communication networks of various kinds, as well as power-distribution networks, cooling networks (for water-cooled systems), *etc.*

Figure 1 shows a fairly typical 256-node/2048-core system with nodes arranged in 10 racks with 25 nodes in each rack, and with the six nodes thereby not accounted for located in the center-rack (together with login-nodes, management-nodes, and a 288-port Infiniband switch). The limit of 25 nodes per rack is to prevent excessive floor loading, and as expected nodes 1–25 (counted from the bottom and upwards) are located in rack 1, nodes 26–50 in rack 2, and so on.

The system's job-scheduler communicates with the nodes over a dedicated control/monitoring network (plain ethernet) in which a separate switch is responsible for the nodes in consecutive pairs of racks. A yellow rectangle has

[1] November 2011: 41.8 % InfiniBand, 44.8 % Gigabit Ethernet,
November 2010: 45.2 % InfiniBand, 42.8 % Gigabit Ethernet.

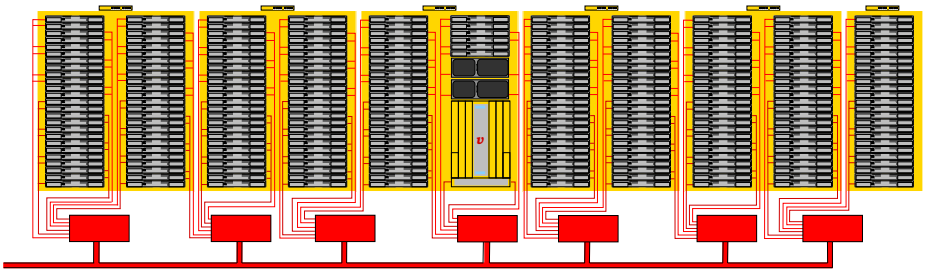

Fig. 1. A 256-node system, with its power- (red) and control- (yellow boxes) networks.

been placed behind each group of nodes managed by a common switch in Figure 1. The power distribution network is shown in red in Figure 1. The red boxes shown below the compute node racks represent fuse-blocks, and as can be seen in the figure, nodes 1, 4, 5, 8, 9, 12, 13 of each rack are connected to one power-line, while nodes 2, 3, 6, 7, 10, 11 of the same rack are connected to a different power-line, and in quite a few cases also to a different fuse-block.

The communication between compute-nodes is performed over InfiniBand. Internally, the 288-port InfiniBand switch used has a Clos topology [9, 32], and has 12 compute nodes connected to each line-card (local routing of messages is performed by each line card). Compute nodes are connected in-order to line-cards, so that nodes 1–12 are connected to line-card 1, nodes 13–24 to line-card 2, nodes 25–36 to line card 3 *etc.*

With all the different networks that are involved and their differing structures, it is clear that no single hierarchy (such as can be defined when using SLURM [35] or PBS Pro job schedulers) will suffice to simultaneously take the various networks of the system into account. To make matters concrete, it is desirable that a job makes use of as few InfiniBand line-cards as possible (for communication efficiency reasons), but at the same time it should ideally also make use of as few power-lines as possible (so that each given job is affected by fewer blown fuses), and it should be assigned to nodes below as few different control-network switches as possible (so that each given job can be harmed by as few failed switches as possible).

Finally, the system in Figure 1 is air-cooled, with cold air being provided from below on the system's font-side. For this reason the use of physically lower positioned nodes is preferrable over the use of physically higher positioned nodes, and because of flow of hot air around the sides of the system from rear to front, it is more desirable to use centrally positioned nodes than to use nodes nearer to the sides.

With all things considered, it is clear that the limitations and constraints of the various networks present in a cluster system can interact in non-trivial ways with respect to processor set selection, and that when cooling and energy consumption issues are considered, no two compute-nodes are truly identical unless they also occupy exactly the same physical location. These properties stand in noticeable contrast to most idealized scheduling and selection problems, in which sets of indentical resources and/or objects tend to be assumed.

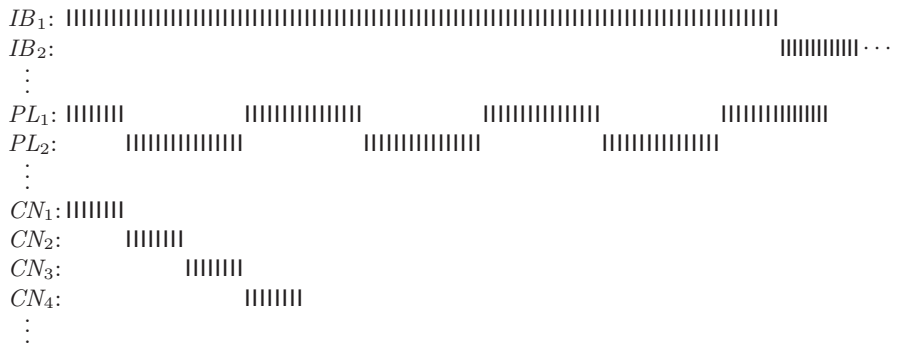

Fig. 2. Typical processor-set definitions, shown as bit-vectors.

3.2 Problem Formalization

To address the processor-set selection (PSS) problem such that the different types of constraints mentioned above can be accounted for, we define a set of processor numbers corresponding to each practical constraint that involves that particular collection of processors. For example, since compute nodes 1–12 are attached to line-card 1 on the InfiniBand switch and compute nodes 13–24 are attached to line-card 2, *etc.*, we define sets: $IB_1 = \{1, \ldots, 96\}$, $IB_2 = \{97, \ldots, 192\}$, *etc.* Similarly, we define sets $PL_1 = \{1, \ldots, 8, 25, \ldots, 40, 57, \ldots, 72, 89, \ldots, 104\}$, $PL_2 = \{9, \ldots, 24, 41, \ldots, 56, 73, \ldots, 88\}$, *etc.*, for processors connected to a common power-line, and continue in the same manner with processor-sets for control-network switches, as well as for each group of processors on the same compute-node (CN_1, CN_2, *etc.*). For the processor-set definitions given above it is assumed that each compute-node has eight processors, and an incomplete but nonetheless illustrative rendering of such processor-sets is given in Figure 2 (above). A complete example of processor-set definitions is also presented in the appendix.

Now, let $U = \{1, \ldots, m\}$ for some $m \in \mathbb{N}$ be the set of all processors that are under the job-scheduler's control. For each processor-set $P_i \subseteq U$, $i \in \{1, \ldots, n\}$, defined as described above (*i.e.*, $P_1 = IB_1$, $P_2 = IB_2$, *etc.*), an associated ℓ-element cost-vector $\mathbf{c}_i \in \mathbb{N}^\ell$ is defined. Given the collection of defined processor sets and their associated cost-vectors, we view the total cost of a processor-set selection $S \subseteq U$ as being given by the vector-sum (denoted by $\oplus$) of those cost-vectors $\mathbf{c}_i$ for which the corresponding processor-set P_i has at least one element in common with S. That is, the cost of a processor selection S can be defined as:[2]

$$cost(S) \;=\; \bigoplus_{i=1}^{n} \mathbf{c}_i \Big[(P_i \cap S) \neq \emptyset \Big] , \tag{1}$$

from which it follows that the cost of a selection is itself also a cost-vector.

[2] Using the notation introduced in [16, page 24] whereby a pair of brackets enclosing a boolean expression evaluates to 1 when the enclosed expression evaluates to true and evaluates to 0 when the enclosed expression evaluates to false.

Cost-vectors are compared lexicographically, which means that we consider the cost $\mathbf{c}_i \in \mathbb{N}^\ell$ as being lower than $\mathbf{c}_j \in \mathbb{N}^\ell$ when $\mathbf{c}_i \prec \mathbf{c}_j$, with the definition of the latter notation given by:

$$\begin{aligned}\mathbf{c}_i \prec \mathbf{c}_j \;\equiv\; & (\mathbf{c}_i)_1 < (\mathbf{c}_j)_1 \vee \big((\mathbf{c}_i)_1 = (\mathbf{c}_j)_1 \wedge (\mathbf{c}_i)_2 < (\mathbf{c}_j)_2\big) \vee \cdots \\ & \cdots \vee \big((\mathbf{c}_i)_1 = (\mathbf{c}_j)_1 \wedge \cdots \wedge (\mathbf{c}_i)_{\ell-1} = (\mathbf{c}_j)_{\ell-1} \wedge (\mathbf{c}_i)_\ell < (\mathbf{c}_j)_\ell\big)\,,\end{aligned}$$

wherein the notation $(\mathbf{c}_i)_k$ is used to indicate the kth element of $\mathbf{c}_i \in \mathbb{N}^\ell$.

Given a set $P_\mathrm{a} \subseteq U$, of currently available processors and a number of required processors $r \in \mathbb{N}$, the processor-set selection problem is to find a solution, X, to the following optimization problem:

$$\min_{\substack{\prec \\ X \subseteq P_\mathrm{a}}} \Big\{ cost(X) \;\Big|\; |X| \geq r \Big\}, \tag{2}$$

or determine that no such solution X exists.

3.3 Motivations

Compared to plain scalar values, lexicographically ordered cost-vectors have the advantages that different concerns (*e.g.*, communication costs, electrical reliability, energy consumption, *etc.*) are easy to keep separate and differently prioritized, and that cost-constraints to decide the acceptability of selections can be defined in terms of individual cost-vector components. When the cost of each processor-set is limited to being a single scalar value, such tasks become more difficult and thus error-prone.

As described above, a processor-set is defined for a group G of processors to indicate that it is desirable (with respect to some criterion) to let the processors of G be selected for jobs of size $\leq |G|$. That is, selecting processors from two different processor-sets P_1 and P_2, with cost-vectors $\mathbf{c}_1$ and $\mathbf{c}_2$, when any one of P_1 or P_2 alone would suffice, leads to a total selection cost of $\mathbf{c}_1 \oplus \mathbf{c}_2$ instead of $\min_\prec(\mathbf{c}_1, \mathbf{c}_2)$. In many real cases, however, the cost implied by a processor-set definition only arises when the processor-set is straddled by a processor selection and not when the requested number of processors can all be selected within the processor-set, suggesting that common and essential problem-features are not captured by the model.

The idea behind the processor-set/cost-vector model for processor selection, however, is that the *surplus* costs incurred by undesirable processor selections will cause minimization procedures to find more suitable selections whenever such exist,[3] and that it therefore should be irrelevant whether the base (*i.e.*, minimum) total cost of each particular processor selection is zero or some other value. In the case of P_1 and P_2 just above, for example, it is not important that the minimum selection cost is $\min_\prec(\mathbf{c}_1, \mathbf{c}_2)$ instead of $\mathbf{0}$, but what is important is that any attempt at selecting processors from both sets will lead to a total selection cost that lies as much above this minimum selection cost as is given by the (lexicographically) larger value of $\mathbf{c}_1$ and $\mathbf{c}_2$.

[3] In this context we consider such minimization procedures to be exact, despite the results in Section 4 and Section 6.

4 Complexity of the Processor-Set Selection Problem

The theorem below establishes that PSS is NP-hard in the strong sense, which in turn implies that an algorithm that efficiently delivers exact answers for all conceivable problem instances is unlikely to exist.

Theorem 1. *The PSS problem is NP-hard in the strong sense.*

Proof: The proof is by reduction from the set-union knapsack problem (SUKP). The SUKP is defined as follows:

> There is a universe of m elements, denoted by $1, \ldots, m$, and n items, with the set of elements constituting item i denoted by P_i, and such that the union of all items is the set of all elements, $\bigcup_{i=1}^{n} P_i = \{1, \ldots, m\}$. The value of item i is denoted by v_i and the weight of element j is denoted by s_j. The capacity of the knapsack is b. For any $K \subseteq \{1, \ldots, n\}$, we define P_K as $P_K = \bigcup_{i \in K} P_i$. The objective of the SUKP is to find a solution (K) to the following mathematical program:
>
> $$\max \left\{ \sum_{i \in K} v_i \;\middle|\; \sum_{j \in P_K} s_j \leq b,\ K \subseteq \{1, \ldots, n\} \right\}, \tag{3}$$
>
> or less formally expressed; to find a collection of items of maximum total value such that the weight of their constituent elements does not exceed the knapsack capacity b.

In [15], Goldschmidt *et al.* show that SUKP is NP-hard in the strong sense, even in the case when $|P_i| = 2$, $s_j = 1$ and $v_i = 1$, for all $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$.

Given $b \geq m$, the problem in Eq. (3) is trivial. The solution K is simply chosen to contain all items P_i, $i = 1, \ldots, n$. Given $b < m$, one or more elements (and the items containing these elements) must be removed from the solution for $b \geq m$. The elements to remove in order to satisfy Eq. (3) are those for which the items that are consequentially removed contribute the least to the overall value of the knapsack. With R denoting the set of elements to remove, and assuming that $s_j = 1$ for all $j \in \{1, \ldots, m\}$, this can be expressed as:

$$\min_{R \subseteq U} \left\{ \sum_{i=1}^{n} v_i \Big[(R \cap P_i) \neq \emptyset \Big] \;\middle|\; |R| \geq m - b \right\}, \tag{4}$$

where $U = \{1, \ldots, m\}$. Assuming single-element cost-vectors, and that $P_{\mathrm{a}} = U$, the minimization problem in Eq. (4) is identical to that in Eqs. (1–2), and the proof is thereby complete. ■

5 A Simple Processor-Set Selection Algorithm

As should be expected from the results arrived at in Section 4, the simple algorithm presented below is by no means guaranteed to find optimal solutions.

The main idea behind the algorithm SelectProcessorSet (in Figure 3) is to start with the set of all currently available processors P_{a} being the set of selected processors, and then successively remove subsets of P_{a} that correspond to defined processor sets (*i.e.*, $P_i, i \in \{1, \ldots, n\}$), until removing any further such subsets would leave less than the required number, r, of processors selected.

The way in which processor-sets P_i are removed from the set of selected processors has some similarities to the concept of *reaching* as used in dynamic programming [12], in which case solutions to subproblems are computed before it is known whether these solutions will be of use in obtaining the final solution.

The algorithm begins by populating the array *remove* such that *remove*$[k]$ holds the defined processor set (*i.e.*, one of $P_i, i \in \{1, \ldots, n\}$) of size k with the lexicographically largest associated cost-vector (among processor-sets of size k). The algorithm then proceeds to its main phase, in which *remove*$[k]$ is processed in sequence (for $k = 1, \ldots, |P_{\mathrm{a}}| - r - 1$), by forming the union of *remove*$[k]$ with each element of C_{p} (*i.e.*, the defined processor-sets). The size of each processor-set, s', so obtained is determined, and when the total cost of s' is lexicographically greater than that of the current value of *remove*$[|s'|]$, the value of s' will replace the current value of *remove*$[|s'|]$.

When *remove*$[|P_{\mathrm{a}}| - r - 1]$ has been processed in the manner just described, the algorithm's suggestion for the best set of processors to remove from P_{a} such that q processors remain (where $q \geq r$) is stored in *remove*$[|P_{\mathrm{a}}| - q]$, and consequently, the value given by $P_{\mathrm{a}} \setminus (\mathit{remove}[|P_{\mathrm{a}}| - r])$ is returned as the result of the SelectProcessorSet algorithm.

Note that actual implementations (*vide infra*) of the algorithm in Figure 3 compute cost-vectors for processor-sets when forming the unions $s \cup (p_k \cap P_{\mathrm{a}})$, or shortly thereafter, and do not as in Figure 3 repeatedly redo this calculation (in MaxCostSet). This is done in Figure 3 for the purpose of simplifying the presentation.

6 Algorithm Implementation and Evaluation

The algorithm in Figure 3 and a worst-case exponential-time algorithm for the set-union knapsack problem[4] described by Goldschmidt *et al.* [15] were first implemented in Common Lisp [31], along with a simple framework to perform processor-set selection driven by the workload trace described below.

Having observed that the Lisp implementation of the algorithm in Figure 3 behaved and performed reasonably, it was reimplemented in ANSI C [20] and interfaced to the parallel environment queue selection (PQS) API of the Sun Grid Engine (SGE) job-scheduler, and evaluated as described in Section 6.3 below.

[4] Similarly to how the algorithm in Figure 3 operates internally, the set-union knapsack algorithm was used to determine which processors *not* to select.

Algorithm: SELECTPROCESSORSET(r, P_a, C_p, C_v)
Inputs: $r \in \mathbb{N}\backslash\{0\}$: the number of processors requested,
$P_\mathrm{a} \in \mathbb{P}(\mathbb{N})$: the set of currently available processors,
$C_\mathrm{p} = \langle p_1, \ldots, p_n \rangle$: a sequence of processor-sets,
$C_\mathrm{v} = \langle \mathbf{v}_1, \ldots, \mathbf{v}_n \rangle$: a sequence of corresponding cost-vectors.
Output: S: a set of processors ($\emptyset$ if selection impossible).

```
begin
  if r ≤ |P_a| then
    for k ← 1 to n do
      q ← p_k ∩ P_a;
      remove[|q|] ← MaxCostSet( q, remove[|q|], C_p, C_v );
    od
    for j ← 1 to |P_a| − r − 1 do
      if remove[j] ≠ ∅ then
        s ← remove[j];
        for k ← 1 to n do
          s' ← s ∪ (p_k ∩ P_a);
          if |s'| > |s| then
            remove[|s'|] ← MaxCostSet( s', remove[|s'|] , C_p, C_v );
          fi
        od
      fi
    od
    S ← P_a \ remove[|P_a| − r];
  else
    S ← ∅;
  fi
end
```

```
proc MaxCostSet( s_1, s_2, ⟨p_1, ..., p_n⟩, ⟨v_1, ..., v_n⟩ ) : P(N) ≡
  c_1 ← 0;
  c_2 ← 0;
  for k ← 1 to n do
    c_1 ← c_1 ⊕ [(p_k ∩ s_1) ≠ ∅] v_k;
    c_2 ← c_2 ⊕ [(p_k ∩ s_2) ≠ ∅] v_k;
  od
  if c_1 ≺ c_2 then
    return s_2;
  elif c_2 ≺ c_1 then
    return s_1;
  else
    return (if rectrand(0.0, 1.0) < 0.5 then s_1 else s_2 fi);
  fi
end
```

Fig. 3. The simple processor-set selection algorithm.

6.1 Workload Details

The workload used for evaluation is a trace of submitted and executed jobs corresponding to one week (24×7 hours) of wall-clock time on the system described in Section 3.1 (and depicted in Figure 1). The trace begins at a time directly following a restart of the entire system, and jobs started prior to the window of observation need therefore not be considered.

Looked upon in further detail, the workload used can be seen to have the following characteristics:

> 21902 jobs in total with a mean job-size of 44.6 ± 27.4 processors (5.57 ± 3.43 nodes), and with 6692 jobs using 8 or fewer processors (*i.e.*, using at most one node). For jobs using 8 processors or less, the mean run-time is 243 ± 390 sec. The mean run-time of jobs using more than 8 processors is 848 ± 3170 sec., and the mean run-time of jobs using more than 8 processors and more than 900 sec. of run-time is 8375 ± 6445 sec. (*i.e.*, 2.3 ± 1.8 hours).

In all cases, above mentioned run-times refer to wall-clock times and have thus not been scaled by the number of participating processors. Note that for evaluation of processor-set selection algorithms it is important that the workload exhibits substantial variation in the number of unused processors. This criterion is satisfied by the described workload.

On the system in question, the (wall-clock) run-time is limited to 8 hours for all parallel jobs. For this reason it is common practice to use rather long *job-chains* (*i.e.*, the last action of a running job is usually to submit a new instance of itself). In order to avoid obtaining misleading results, the use of job-chains must be properly accounted for when replaying the job-submission traces.

6.2 Prototype Evaluation

For the evaluation of Lisp algorithm implementations, the workload described above was simplified by considering compute-nodes to have only one processor (instead of eight), and dividing the processor counts of all requests by eight.

The processor-set/cost-vector configuration comprised a total of 334 processor-set definitions and associated (5-element) cost vectors, corresponding to InfiniBand line-cards(22), control-network switches (7), power-lines (41), fuse-blocks (8) and compute-nodes (256). The Figure-3-algorithm was run with two different sets of cost-vectors for the mentioned processor-set definitions. In the first such set of cost-vectors, the cost of compute-node processor-sets has been set to reflect the relative desirability of using some compute nodes over others from an energy- and cooling-perspective, as described in Section 3.1. In the second set of such cost-vectors, all compute-node processor-sets have been assigned identical cost-vectors.

Runs with the described workload-trace were performed with the simple algorithm using both sets of cost-vectors for all processor-selection requests, whereas the exact algorithm was run only when the problem size was small enough ($|P_a| - r \leq 12$) that its execution-time was still reasonable (see Figure 4). The selection decisions made by the simple algorithm using the more

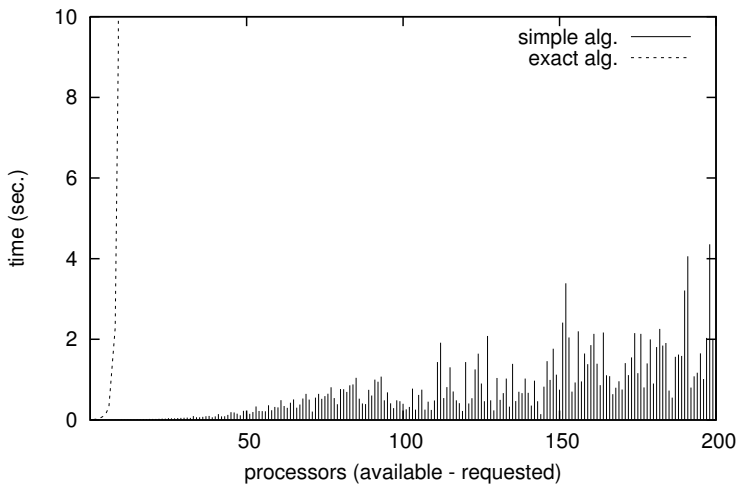

Fig. 4. Mean execution-times of the Lisp implementations of the simple and the exact processor-set selection algorithms, as a function of $|P_{\mathrm{a}}| - r$, for each request. All measurements were made using compiled Lisp code with CMUCL [24] (release 19d) running on an Intel Core2 Duo T1700 1.8 GHz processor with 2Gb physical memory.

realistic (*i.e.*, non-uniform) cost-vectors were the ones actually used to change the selected-state of nodes maintained by the simulation, and thus influencing the starting conditions for subsequent processor selections.

For the limited subset of selection problems that could be handled by the exact algorithm, the simple algorithm using the more realistic cost-vectors arrived at a different selection than the exact algorithm (also using realistic cost-vectors) for only 23 out of 4272 multi-node jobs/requests. When comparing results obtained for the two different sets of cost-vectors and using the uniform cost-vectors to judge selection quality, selections of equal cost were obtained with both sets of cost-vectors for 15155 out of 15210 multi-node jobs. For the remaining 55 jobs, better processor selections were obtained using realistic cost-vectors than using uniform cost-vectors for 41 jobs ($\sim 75\,\%$), even though solution quality was judged according to the uniform cost-vectors. Correspondingly for the 4272 requests that could be handled by the exact algorithm, the exact algorithm (now using uniform cost-vectors) arrived at better processor selections than the simple algorithm using realistic cost-vectors for 4 jobs, and the simple algorithm using realistic cost-vectors in turn arrived at better processor selections than the same algorithm using uniform cost vectors for 4 jobs and worse for only 1 job, with solution quality again judged according to the uniform cost-vectors. We view this as indications that artificially introduced non-uniformness in cost-structure definitions may contribute to improved processor-set selection quality.

Prototyping the algorithms in Lisp allowed us to focus efforts on key issues, and postpone less immediately relevant matters such as file-formats for describing processor-sets and cost-vectors, implementations of bit-vectors, and dynamic memory management, that need to be addressed when using a language such as C. However, as can be inferred from Figure 4, showing mean execution-times (without error-bars, since these would have completely cluttered the diagram), the execution-times fluctuate noticeably, and one of the main reasons for this variability is that bit-vectors have been implemented as `bignum`s [31], the sizes of which vary in correspondance with the most significant bit that is set.

6.3 SGE-implementation Evaluation

The evaluation of the SGE-interfaced algorithm implementation used the workload described in Section 6.1 without modification (*i.e.*, assuming 2048 processors in total and 8 processors per node). Compared to the description in Section 6.2, processor-set definitions were expanded as is implied by having 8 processors per node instead of only 1, and cost-vectors were expanded by adding a new first element, through which the selection of processors on as few different nodes as possible was made the primary objective.

A distinct SGE-master was set up on one of the management hosts of the system described in Section 3.1, and its 256 compute nodes were cloned and subsequently simulated through Xen hypervisors and virtual machines on 16 compute nodes of the same system (*i.e.*, with 16 virtual compute nodes on each real compute node). Since the wall-clock execution-time of each job is known from the workload-trace, each job simply sleeps an amount of time corresponding to its execution-time,[5] and therefore no substantial load arises on the virtual machines. This method of replaying the workload-trace enabled us to observe the described algorithm and its implementation under very authentic conditions, with a very modest impact on the physical system.

The mean recorded execution times of the SGE-interfaced processor-set selection algorithm as a function of the number of processors *not* to select for each corresponding job (*i.e.*, $|P_{\mathrm{a}}| - r$) is shown in Figure 5 (a). We find the observed running-times of the algorithm to clearly be within acceptable limits, particularly in view of the fact that systems of the kind in question should preferably be sufficiently heavily used that the number of idle processors only rarely can be counted in the hundreds or thousands, and that in the common case that parallel jobs are always given complete nodes for themselves, problem sizes can be reduced as was done in Section 6.2. Finally, Figure 5 (a) does indicate some execution-time irregularities, but the exact sources of these are currently not clear to us.

As explicitly stated and as implied, respectively, in the previous discussion, the algorithm was run with a processor-set and cost-vector configuration such

[5] In order to ensure proper treatment by the SGE job-accounting machinery, the sleep operations were performed by letting an `mpiexec`-command in each job-script invoke `sleep` commands (in parallel) that slept for the appropriate length of time.

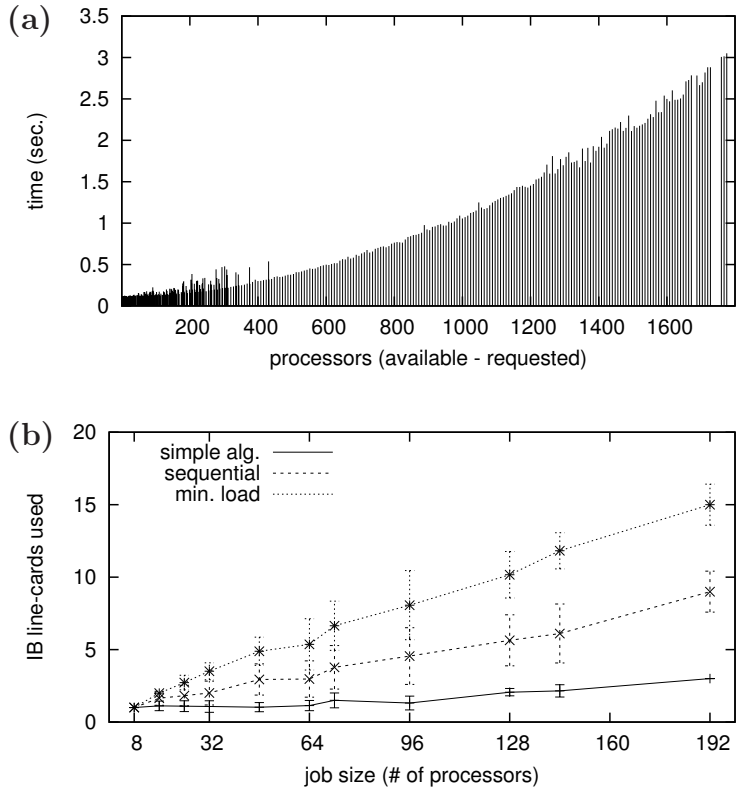

Fig. 5. (a) Mean execution-times of the SGE-interfaced processor-set selection algorithm as a function of $|P_{\mathrm{a}}| - r$, measured on a Sun Fire X4100 with 2.4 GHz AMD Athlon processors and 4 Gb physical memory. (b) Average number of different InfiniBand line-cards used by jobs of various sizes for three different processor-set selection methods (error-bars indicate standard deviations).

that the primary objective was to select processors on as few different nodes as possible and the secondary objective was to select processors connected to as few different InfiniBand line-cards as possible. The third-, fourth-, fifth-, and sixth-level objectives were to minimize the use of control-network switches, power-lines, fuse-blocks, and compute-node energy/cooling costs, respectively. The primary objective was achieved for all jobs (in part because all jobs in the workload request a multiple of 8 processors). Figure 5 (b) presents the outcome with respect to the secondary objective, compared to the two processor selection strategies that the SGE has built-in (*i.e.*, sequentially by increasing node numbers, and least loaded nodes). With respect to communication locality, as can be seen in Figure 5 (b), the processor-set selection method described in this paper represents a clear improvement over both of the strategies provided in the SGE, and which are widely used in practice (also by other job-schedulers).

7 Summary and Conclusions

We have presented a model for processor selection by parallel job-schedulers that is conceptually simple, easy to understand, and flexible with respect to the kinds of constraints that can be accounted for. The model is also easily extended such that different processor-set and cost-vector definitions can be given for different ranges of job sizes.[6] In this way, jobs of different sizes can be steered towards different regions of a system, constraints only affecting jobs of specific sizes can be accounted for, and processor-set/cost-vector definitions can be kept shorter.

The resulting minimization problem for optimal processor selection was proven to be NP-hard in the strong sense. A simple (approximative) algorithm is nonetheless presented and shown to run sufficently fast to be practically useful and to yield processor selections of acceptable quality and that represent a clear improvement over processor selection strategies that are currently in widespread practical use. The quality of solutions obtained by the algorithm appears to also benefit slightly from less uniformly defined processor-set costs, suggesting that more realistic cost models can bring both direct and indirect advantages.

Concerning job-scheduler based processor-set selection in general, it has been observed on multiple occasions that imposing topology-related constraints on processor selection usually leads to longer waiting times and reduced overall system utilization (*e.g.*, see [2, 28]), suggesting that such mechanisms bring little benefit. On the other hand, by successfully imposing topology-related constraints on processor selection, it may be possible to purchase a larger number of processors (because of a less expensive communications network), in which case a higher total throughput may be delivered despite reduced system utilization.

Acknowledgements

The author would like to thank Dipl. Ing. Frank Rambo for explaining the electrical wiring of the 256-node/2048-core cluster shown in Figure 1, Dipl. Ing. Christian Schwede for setting up the Xen-based virtual replica of the already mentioned cluster, and Dipl. Inf. Gerald Vogt for patiently answering questions concerning energy consumption, cooling, and air-flow in the machine-rooms.

References

1. Adiga, N.R., et al.: Blue Gene/ L torus interconnection network. IBM J. Res. Develop. 49(2/3), 265–276 (Mar/May 2005)
2. Aridor, Y., et al.: Resource allocation and utilization in the Blue Gene/ L supercomputer. IBM J. Res. Develop. 49(2/3), 425–436 (Mar/May 2005)
3. Berman, F., Snyder, L.: On mapping parallel algorithms onto parallel architectures. J. Parall. Distr. Comput. 4(5), 439–458 (1987)

[6] In fact, job-size differentiated processor-set and cost-vector definitions are supported by all algorithm implementations mentioned in this paper. These facilities were not used in the described evaluation runs, however.

4. Bhanot, G., et al.: Optimizing task layout on the Blue Gene/L supercomputer. IBM J. Res. Develop. 49(2/3), 489–500 (Mar/May 2005)
5. Bhatelé, A., Bohm, E., Kalé, L.V.: Optimizing communication for Charm++ applications by reducing network contention. Concur. Pract. Exp. 23(2), 211–222 (Feb 2011)
6. Bokhari, S.H.: Assignment Problems in Parallel and Distributed Computing. Kluwer Academic Publishers, Norwell, MA (1987)
7. Bollinger, S.W., Midkiff, S.F.: Heuristic technique for processor and link assignment in multicomputers. IEEE Trans. Comput. C-40(3), 325–333 (Mar 1991)
8. Chokalingam, T., Arunkumar, S.: Genetic algorithm based heuristics for the mapping problem. Comput. Oper. Res. 22(1), 55–64 (Jan 1995)
9. Clos, C.: A study of non-blocking switching networks. Bell Sys. Tech. J. 32(2), 406–424 (Mar 1953)
10. Cray Inc., Seattle, WA 98104, U.S.A.: Cray XT System Overview (2009), publication No. S–2423–22
11. Darte, A., Robert, Y.: Mapping uniform loop nests onto distributed memory architectures. Parallel Computing 20(5), 679–710 (May 1994)
12. Denardo, E.V.: Dynamic Programming: models and applications. Dover Publications, Mineola, NY 11501 (2003)
13. Ercal, F., Ramanujam, J., Saddayappan, P.: Task allocation onto a hypercube by recursive mincut bipartitioning. J. Parallel Distrib. Comput. 10(1), 35–44 (Sep 1990)
14. Feitelson, D.G., Rudolph, L.: Parallel job scheduling: Issues and approaches. In: Feitelson, D.G., Rudolph, L. (eds.) Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science, vol. 949, pp. 1–18. Springer-Verlag, Berlin, Germany (1995)
15. Goldschmidt, O., Nehme, D., Yu, G.: Note: On the set-union knapsack problem. Nav. Res. Logist. 41(6), 833–842 (Oct 1994)
16. Graham, R.L., Knuth, D.E., Patashnik, O.: Concrete Mathematics. Addison-Wesley, Reading, Massachusetts, 2nd edn. (1994)
17. IBM Blue Gene Team: Overview of the IBM Blue Gene/ P project. IBM J. Res. Develop. 52(1/2), 199–220 (Jan/Mar 2008)
18. Jones, J.P., Nitzberg, B.: Scheduling for parallel supercomputing: A historical perspective of achievable utilization. In: Feitelson, D.G., Rudolph, L. (eds.) Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science, vol. 1657, pp. 1–16. Springer-Verlag, Berlin, Germany (1999)
19. Kermani, P., Kleinrock, L.: Virtual cut-through: A new computer communication switching technique. Computer Networks 3(4), 267–286 (Sep 1979)
20. Kernighan, B.W., Richie, D.M.: The C Programming Language. Prentice-Hall, Englewood Cliffs, NJ 07632, 2nd edn. (1988)
21. Lee, S.Y., Aggarwal, J.K.: A mapping strategy for parallel processing. IEEE Trans. Comput. C-36(4), 433–442 (Apr 1987)
22. Leiserson, C.E.: Fat-Trees: Universal networks for hardware-efficient supercomputing. IEEE Trans. Comput. C-34(10), 892–901 (Oct 1985)
23. Lifka, D.: The ANL/IBM SP scheduling system. In: Feitelson, D.G., Rudolph, L. (eds.) Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science, vol. 949, pp. 295–303. Springer-Verlag, Berlin, Germany (1995)
24. MacLachlan, R.A.: CMUCL User's Manual. Carnegie-Mellon University (Nov 2006), release 19d

25. Mu'alem, A.W., Feitelson, D.G.: Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. IEEE Trans. Parall. Distr. Sys. 12(6), 529–543 (Jun 2001)
26. Navaridas, J., et al.: Effects of job and task placement on parallel scientific applications performance. In: Proc. 17th Euromicro Int'l Conf. on Parallel, Distributed and Network-based Processing. pp. 55–61 (Feb 2009)
27. Ni, L.M., McKinley, P.K.: A survey of wormhole routing techniques in direct networks. Computer 26(2), 62–76 (Feb 1993)
28. Pascual, J.A., et al.: Effects of topology-aware allocation policies on scheduling performance. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science, vol. 5798, pp. 138–156. Springer-Verlag, Berlin, Germany (2009)
29. Quinton, P., van Dongen, V.: The mapping of linear recurrence relations on regular arrays. J. VLSI Signal Process. 1(2), 95–113 (Oct 1989)
30. Skovira, J., et al.: The EASY – LoadLeveler API project. In: Feitelson, D.G., Rudolph, L. (eds.) Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science, vol. 1162, pp. 41–47. Springer-Verlag, Berlin, Germany (1996)
31. Steele, Jr., G.L.: Common Lisp: the Language. Digital Press, Burlington, Massachusetts, 2nd edn. (1990)
32. Voltaire Ltd., Herzliya, Israel: Voltaire GridVision Integrated Grid Directors User Manual (May 2007), part Number: 399Z00038
33. Wan, M., et al.: A batch scheduler for the Intel Paragon MPP System with a noncontiguous node allocation algorithm. In: Feitelson, D.G., Rudolph, L. (eds.) Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science, vol. 1162, pp. 48–64. Springer-Verlag, Berlin, Germany (1996)
34. Weisser, D., et al.: Optimizing job placement on the Cray XT3. In: Cray User Group 2006 Proceedings. Lugano, Switzerland (2006)
35. Yoo, A.B., Jette, M.A., Grondona, M.: SLURM: Simple linux utility for resourse management. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science, vol. 2862, pp. 44–60. Springer-Verlag, Berlin, Germany (2003)
36. Yu, H., Chung, I.H., Moreira, J.: Topology mapping for Blue Gene/L supercomputer. In: SC'06: Proc. 2006 ACM/IEEE Conf. on Supercomputing. ACM, New York, NY (2006)

Appendix: Configuration File Format

The decisions made by the presented algorithm for processor set selection are governed by a configuration file, that specifies the collection of available queues, the collections of processor sets, the corresponding cost-vectors, and the cost criteria any actual processor-set selection must satisfy in order to be seen as being acceptably good.

A concrete example of such a configuration file is provided in Figure 6. The hypothetical target system is assumed to be deployed analogously to that shown in Figure 1, but in order to keep the configuration file within the allotted page-limits, it comprises only 6 racks with 7 nodes in each rack, for a total of 42 nodes,

with a relative positioning as follows, when looking at the system's front side:

n07	n14	n21	n28	n35	n42
n06	n13	n20	n27	n34	n41
n05	n12	n19	n26	n33	n40
n04	n11	n18	n25	n32	n39
n03	n10	n17	n24	n31	n38
n02	n09	n16	n23	n30	n37
n01	n08	n15	n22	n29	n36

As seen in Figure 6, a configuration file consists of five separate parts, each beginning with a distinct keyword. The first part (line 1 in Fig. 6), simply names the queues in which jobs may run for which processor-sets are to be selected. The second part (lines 3–11) specifies how many processors that are available on each host and for which queues. More complicated cases than what is shown on lines 3–11 can also be handled. For example, by writing

```
[cluster@k007=1,3,5|standby@k007=2,4,6]
```

it is specified that processors 1, 3 and 5 are available through the queue `cluster` and processors 2, 4 and 6 through the queue `standby`, on the node `k007`.

Processor-set definitions

The processor-sets (such as described in Section 3.2 above) that are to be used are specified following the keyword `groups`. Distinct processor-sets for the processors of each indivdual compute node are defined on lines 13–21 in Fig. 6. The 42 nodes of our hypothetical target system are assumed to be connected to 7 different 6-port line-cards. Processor-sets corresponding to the processors whose nodes are directly attached to each line-card are defined on lines 21–27 of Fig. 6. For example, through

```
ib1={n01,n02,n03,n04,n05,n06},
```

a processor-set named `ib1` is defined, that comprises all processors on nodes `n01`, ..., `n06`, and this in turn is exactly the set of processors attached to line-card 1 on the switch.

Analogously, the remaining processor-set definitions on lines 28–40 in Fig. 6, correspond to other factors such that it may be desirable that they influence the processor-set selection for jobs, and as discussed in Section 3.

Processor-set cost definitions

As shown in Figure 6, the definitions of cost-vectors associated with processor sets can depend on the size of the requesting job. This can be used for a variety of purposes, such as imposing stricter communication locality requirements for small jobs and selecting processor-sets for differently sized jobs starting from different physical regions of a machine, *etc.*

In Figure 6, processor-set costs for 1–4 processor jobs are defined separately from those for jobs of all other sizes. Although this has been done mainly to make it clear that doing so is possible, it also provides an opportunity to discuss the costs choosen for the node processor sets. Even though the system is composed of logically identical nodes, they are obviously not physically in exactly the same location, and each node thereby has a slightly different physical environment. The different physical environments of the nodes can induce an ordering according to which the use of some nodes is preferable over the use of other nodes, in the absence of other constraints, and assuming the system is not currently operating under full load (*i.e.*, that some nodes need not be used).

In the present case, with cooling by cold air provided from below at the front-side of the system, the use of lower placed nodes is preferable to the use of higher placed nodes, and because of flow hot air around the sides of the system from rear to front it is more desirable to use centrally positioned nodes than nodes nearer to the sides. The cost-vector definitions on lines 44–64 in Figure 6 are intended to express precisely the cooling related preferences w.r.t. processor-set selection that have just been described.

Constraint definitions

Finally, the fifth and final part of a configuration file, contains definitions of cost constraints that must be satisfied by processor-set selections. As shown on lines 103–107 in Figure 6, each constraint is simply a boolean-valued expression, and when this expression does not evaluate to true for a proposed processor-set selection, the job scheduler is informed that the job in question should not be allowed to start.

Just as for cost-definitions, constraints can be differently defined for jobs of different sizes. In combination with job-size and queue differentiated processor-set costs this enables a rather precise control over when a potential processor-set selection is considered to be of sufficiently high quality to be used.

Minor implementation details

Due to the size and complexity of configuration files, they are not read as such by the processor-set selection machinery. Instead, configuration files are parsed and validated by a separate program, that for valid configuration files create corresponding (machine independent) binary configuration files, and these in turn are read by the processor-set selection machinery.

At regular time-intervals (every 5 min. currently), the processor-set selector checks the last modification time of its binary configuration file, and reloads it if it has changed. It is thereby easily arranged to make use of different processor-set selection strategies at different times of day or during weekends *vs.* workdays, *etc.* The separately performed configuration-file validation (and conversion into binary form), prevents simple mistakes (*e.g.*, syntactic errors) made when preparing and/or modifying configuration files from influencing processor-set selection behaviour and decisions.

```
queues: cluster,standby

slots: [n01=1..4],[n02=1..4],[n03=1..4],[n04=1..4],[n05=1..4],
       [n06=1..4],[n07=1..4],[n08=1..4],[n09=1..4],[n10=1..4],
          :          :          :          :          :
       [n36=1..4],[n37=1..4],[n38=1..4],[n39=1..4],[n40=1..4],
       [n41=1..4],[n42=1..4]

groups: h01={n01},h02={n02},h03={n03},h04={n04},h05={n05},
        h06={n06},h07={n07},h08={n08},h09={n09},h10={n10},
           :         :         :         :         :
        h36={n36},h37={n37},h38={n38},h39={n39},h40={n40},
        h41={n41},h42={n42},ib1={n01,n02,n03,n04,n05,n06},
        ib2={n07,n08,n09,n10,n11,n12},
                  :         :
        ib7={n37,n38,n39,n40,n41,n42},
        en1={n01,n02,n03,n04,...,n11,n12,n13,n14},
        en2={n15,n16,n17,n18,...,n25,n26,n27,n28},
        en3={n29,n30,n31,n32,...,n39,n40,n41,n42},
        ps1a={n07,n06,n03,n02},ps1b={n05,n04,n01},
        ps2a={n14,n13,n10,n09},ps2b={n12,n11,n08},
        ps3a={n21,n20,n17,n16},ps3b={n19,n18,n15},
        ps4a={n28,n27,n24,n23},ps4b={n26,n25,n22},
        ps5a={n35,n34,n31,n30},ps5b={n33,n32,n29},
        ps6a={n42,n41,n38,n37},ps6b={n40,n39,n36},
        pwr1={n01,n02,n03,n04,n05,n06,n07,n09,n10,n13,n14},
        pwr2={n08,n11,n12,n15,n16,n17,n18,n19,n20,n21},
        pwr3={n22,n23,n24,n25,n26,n27,n28,n30,n31,n34,n35},
        pwr4={n29,n32,n33,n36,n37,n38,n39,n20,n41,n42}

costs:
    when PEs in [1,4]:
        cluster@h01=[0,0,0,0,129],cluster@h02=[0,0,0,0,135],
        cluster@h03=[0,0,0,0,141],cluster@h04=[0,0,0,0,147],
        cluster@h05=[0,0,0,0,153],cluster@h06=[0,0,0,0,159],
        cluster@h07=[0,0,0,0,165],cluster@h08=[0,0,0,0,115],
        cluster@h09=[0,0,0,0,121],cluster@h10=[0,0,0,0,127],
        cluster@h11=[0,0,0,0,133],cluster@h12=[0,0,0,0,139],
        cluster@h13=[0,0,0,0,145],cluster@h14=[0,0,0,0,151],
        cluster@h15=[0,0,0,0,101],cluster@h16=[0,0,0,0,107],
        cluster@h17=[0,0,0,0,113],cluster@h18=[0,0,0,0,119],
        cluster@h19=[0,0,0,0,125],cluster@h20=[0,0,0,0,131],
        cluster@h21=[0,0,0,0,137],cluster@h22=[0,0,0,0,102],
        cluster@h23=[0,0,0,0,108],cluster@h24=[0,0,0,0,114],
        cluster@h25=[0,0,0,0,120],cluster@h26=[0,0,0,0,126],
```

Fig. 6. Configuration file for processor-set selection.

```
57          cluster@h27=[0,0,0,0,132],cluster@h28=[0,0,0,0,138],
58          cluster@h29=[0,0,0,0,116],cluster@h30=[0,0,0,0,122],
59          cluster@h31=[0,0,0,0,128],cluster@h32=[0,0,0,0,134],
60          cluster@h33=[0,0,0,0,140],cluster@h34=[0,0,0,0,146],
61          cluster@h35=[0,0,0,0,136],cluster@h36=[0,0,0,0,130],
62          cluster@h37=[0,0,0,0,136],cluster@h38=[0,0,0,0,142],
63          cluster@h39=[0,0,0,0,148],cluster@h40=[0,0,0,0,154],
64          cluster@h41=[0,0,0,0,160],cluster@h42=[0,0,0,0,166]
65      otherwise:
66          cluster@h01=[0,0,0,0,129],cluster@h02=[0,0,0,0,135],
 ⋮                    ⋮                          ⋮
86          cluster@h41=[0,0,0,0,160],cluster@h42=[0,0,0,0,166],
87          cluster@ib1=[4,0,0,0,0],cluster@ib2=[4,0,0,0,0],
88          cluster@ib3=[4,0,0,0,0],cluster@ib4=[4,0,0,0,0],
89          cluster@ib5=[4,0,0,0,0],cluster@ib6=[4,0,0,0,0],
90          cluster@ib7=[4,0,0,0,0},cluster@en1=[0,1,0,0,0],
91          cluster@en2=[0,1,0,0,0],cluster@en3=[0,1,0,0,0],
92          cluster@ps1a=[0,0,2,0,0],cluster@ps1b=[0,0,2,0,0],
93          cluster@ps2a=[0,0,2,0,0],cluster@ps2b=[0,0,2,0,0],
94          cluster@ps3a=[0,0,2,0,0],cluster@ps3b=[0,0,2,0,0],
95          cluster@ps4a=[0,0,2,0,0],cluster@ps4b=[0,0,2,0,0],
96          cluster@ps5a=[0,0,2,0,0],cluster@ps5b=[0,0,2,0,0],
97          cluster@ps6a=[0,0,2,0,0],cluster@ps6b=[0,0,2,0,0],
98          cluster@pwr1=[0,0,0,4,0],cluster@pwr2=[0,0,0,4,0],
99          cluster@pwr3=[0,0,0,4,0],cluster@pwr4=[0,0,0,4,0]
100     end-costs
101
102 constraints:
103     when PEs in [1,4] : cost <= [0,0,0,0,166],
104     when PEs in [5,16] : cost <= [8,1,6,8,628],
105     when PEs in [17,32] : cost <= MINCOST(32) + [0,0,0,0,64],
106     when PEs in [33,64] : cost <= MINCOST(64) + [4,1,2,0,38],
107     otherwise : cost <= ceil(1.2*MINCOST(PEs)) + [4,0,2,0,28]
```

Fig. 6. Configuration file for processor-set selection (continued).