Article

Analytical and numerical evaluation of co-scheduling strategies and their application

Ruslan Kuchumov [1,*] and Vladimir Korkhov [1]

[1] Saint Petersburg State University, St. Petersburg, Russia; kuchumovri@gmail.com, v.korkhov@spbu.ru
* Correspondence: kuchumovri@gmail.com

Abstract: Applications in high-performance computing (HPC) may not use all available computational resources, leaving some of them underutilized. By co-scheduling, i.e. running more than one application on the same computational node, it is possible to improve resource utilization and overall throughput. Some applications may have conflicting requirements on resources and co-scheduling may cause performance degradation, so it is important to take it into account in scheduling decisions. In this paper, we formalized co-scheduling problem and proposed multiple scheduling strategies to solve it: an optimal strategy, an online strategy and heuristic strategies. These strategies vary in terms of the optimality of the solution they produce and a priori information about the system they require. We showed theoretically that the online strategy provides schedules with a competitive ratio that has a constant upper limit. This allowed us to solve the co-scheduling problem using heuristic strategies that approximate this online strategy. Numerical simulations showed how heuristic strategies compare to the optimal strategy for different input systems. We proposed a method for measuring input parameters of the model in practice and evaluated this method on HPC benchmark applications. We showed high accuracy of measurement method, which allows to apply proposed scheduling strategies in scheduler implementation.

Keywords: Co-scheduling; HPC; scheduling theory; stochastic optimization

1. Introduction

Applications in high performance computing (HPC) rely heavily on the performance of the computational resources they are using. Such resources include, for example, memory cache size, memory bus bandwidth, network bandwidth, computational units in CPU, accelerator cards, etc. All of these resources are rarely fully utilized at the same time, instead, an application may reach maximum utilization of one resource, which would limit its use of other resources. These underutilized resources potentially may be used by other applications working at the same time.

Commonly used batch schedulers in HPC, for example SLURM or PBS, do not allow to implement this behaviour. Instead, they work by assigning the whole computational node to a single task, even if a task is using only a small portion of all available resources. In this case, when there are other tasks waiting in the queue, this reservation strategy causes underutilization of resources and an increase in queue wait time.

In aforementioned HPC batch schedulers it is also possible to split cluster nodes by CPU cores into slots that can run tasks simultaneously on the shared resources. This approach may improve utilization, but is still not a common practice, as applications may have bottleneck on the same resource and interfere with each others performance. In some cases, their performance degradation may be such that sequential execution would produce a better overall throughput, even though resources may not be fully utilized.

Deciding which tasks can be executed simultaneously with a minimal interference in advance, before starting them, is a challenging problem for several reasons. One of them is that the model of application degradation as a function of available resources can not be provided by the user or, in some cases, even by application developer. This dependency is required for making scheduling decisions, and in practice, it can only be estimated from the experiments. Other reasons are related to that this dependency should be reproducible

Citation: Kuchumov, R.; Korkhov, V. Analytical and numerical evaluation of co-scheduling strategies and their application. *Preprints* **2021**, *1*, 0. https://doi.org/

Received:
Accepted:
Published:

 doi:10.20944/preprints202109.0053.v1

as application behaviour may change due to different factors. For example, it may change for different values of application input parameters, for different hardware configurations, or it may change randomly when application is non-deterministic.

Instead, due to these practical limitations, in this paper we propose to approach this problem as an online scheduling problem. In these types of problems, scheduling decisions are made in the runtime based on the incomplete information that is available about each task. They are opposed to offline scheduling problems, where decisions are made before execution of any tasks based on complete information.

To evaluate the dependency of task degradation on the available resources, we propose a method of measuring task processing speed in its runtime. By comparing these values between different states of schedulers it allows us to estimate the change of task performance. In particular, task speed values that are measured twice, the first time when the task is running alone on the node (in ideal conditions), and the second time in when it is running in parallel with other tasks, can be used to quantify how task performance is being affected by other tasks.

In this paper, we evaluate the method of measuring task processing speed on several benchmark applications that mimic real HPC workloads. For the tasks that perform the same amount of work, regardless of the amount of available resources they have, the processing speed measured in their run-time must be inverse proportional to their total processing time. This property allows us to experimentally evaluate the method of measuring task speed by running tasks in different combinations and checking that the change of processing speed of a task is the same as the inverse change of its processing time.

The scheduling problem where more than one task can be executed on the same machine is often referred to as co-scheduling or co-allocation problem. In this paper we have formalized this problem as scheduling theory problem. We have limited our scope only to the static scheduling problems, where the number of tasks, their processing speed and required amount of work do not change in time. We also assumed that all tasks are available for execution at start, task preemption is allowed and the objective function is to minimize the completion time of the last task in the schedule (i.e. makespan).

In this paper we propose several scheduling strategies that can be applied to solve co-scheduling problem. These strategies vary in terms of optimality of the solution they produce and in terms of the information about each task that is available to them. By having all task information available at start, it is possible to define a strategy that constructs an optimal schedule. This optimal strategy can not be used in practice, but it can be useful for evaluation of online strategies, that make their decisions based on a limited information about the system.

We solve this co-scheduling problem by reducing to linear programming programming problem. Then, we propose an online strategy that works with an assumption that task processing speed value are still available, but the required amount of work for each task in unknown. For this strategy we theoretically show that the schedules that it produces can not be more than two times slower than the optimal schedule. This results allow us to define a set of heuristic strategies that approximate the online strategy. They work with an assumption that task processing speed values are initially unknown, but they can be measured in the scheduler run-time. We evaluate all of these strategies using numerical simulations on a randomly generated data that approximate experimentally measured values.

2. Related work

The problem of scheduling computational tasks on shared resources started to appear recently in scientific literature in HPC field. For example, there are workshop proceedings papers [1,2] dedicated to co-scheduling problem of HPC applications. These particular papers, along with others are mostly focused on feasibility of this approach in general and providing a proof of concept implementations. In [3] authors showed that by sharing

 doi:10.20944/preprints202109.0053.v1

nodes between two distributed applications instead of running these applications on dedicated nodes may improve performance by 20% in both their execution time and system throughput. In [4] this approach is referred to as job striping and it was shown to produce more than 22% increase system throughput on benchmark and real applications. In [5] authors presented scheduler implementation that applies co-scheduling strategy and reported up 35% increase in throughput. Paper [6] reports decrease of up to 26% of makespan due to co-scheduling of two applications.

There are also theoretical publications that are focused on modelling the co-scheduling problem. Some authors in their publications approach it as an offline scheduling problem, where all task data is available at start and an optimal schedule can be constructed in advance. Among these publications are [7,8] and [9], where authors are solving offline scheduling problem with resource constraint. They model CPU cache partition size as a controllable task resource and define task speedup as a function of cache partition size. These speedup functions are assumed to be known in advance and they are used for constructing an optimal schedule.

The problem of measuring task speedup profiles that later can be used for offline scheduling is described, for example, in [6]. In this paper authors used supervised machine learning approach for constructing a model of applications slowdown as a function of CPU performance counters values. Training dataset was obtained by measuring performance counters and execution times of applications in ideal conditions and in pairs with other applications. Authors fitted different models on the training data from 27 benchmark applications and reported that the best model provided 78% prediction accuracy for unseen applications.

Online co-scheduling problem for HPC applications is not covered in scientific literature to the same extent as offline scheduling and implementation-related problems. Among available publications there is [10] where authors propose to use reinforcement learning algorithm for dynamic colocation of services and batch HPC workloads. Computational resources for HPC workloads in the described setup are provided as opportunistic resources when service meets its target service level agreement (query latency). Authors proposed an algorithm that dynamically scales CPU clock frequency of the cores that run HPC workloads based on the values of CPU performance counters and the feedback reported by the service.

Similar online scheduling problem is covered in context of thread scheduling in simultaneous multithreading (SMT) CPUs. For example, there are [11–14], dating from the year 1999. The general idea, proposed in these papers, is to dynamically measure instruction per cycle (IPC) values of each thread, when it is running alone on a core and when it is running in parallel with other threads. Then the ratio of these two values is used for making scheduling decisions.

In the follow up paper [15] authors revisited this approach and showed that and experimentally showed that any strategy would not produce more than 3% gain in instructions throughput compared to a naive strategy of running all threads in parallel in the queue order. This results was achieved by measuring slowdown of each benchmark application thread in all possible combinations with other threads. These values were used to construct an optimal schedule which was later compared to a naive strategy.

Earlier in our research we have done the similar work but in the context of HPC applications. We have also provided theoretical boundaries for task speed values when naive parallel strategy can not be applied and showed how it deviated from the optimal strategy.

In the literature there are several approaches for controlling applications to implement co-scheduling. In [9] cache partitioning technology was used for assigning each core a specified amount of available shared cache. When application has a bottleneck in cache accesses, restricting its size allow to control application processing speed. In [10] the approach of changing voltage and frequency of individual CPU cores was used. By scaling the frequency of the cores that the same task it possible to change its processing speed.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 September 2021 doi:10.20944/preprints202109.0053.v1

In [5] authors used a different approach of migrating applications in virtual machines between nodes. This approach allows to implement task preemption, so one task that is in conflict with other tasks could be moved to a different node. There are also papers, e.g. [6] where authors do not control resources allocation or processing speed of each task in run-time, but instead decide on which task combinations to schedule in advance, before task processing starts.

3. Problem of tasks co-scheduling

In this section we will formalize task co-scheduling problem, provide an optimal solution that requires all a priori information, and an online solution that does not need compete information about the system at start. For the online solution we will provide an upper boundary value of the competitive ratio.

We will only consider the stationary problem definition, where the number of tasks, their processing speed in all combinations and required amount of work does not change in time. This assumption limits the scope of theoretical methods that can be used. In practice it may not always hold, but results obtained for stationary system can useful for dynamic systems as well.

We will use the following notation to formalize the scheduling problem. There are n tasks (applications) denoted as $T = \{T_1, \ldots, T_n\}$. Each T_i requires b_i work units to be performed before its completion. Any subset of tasks from T can be ran simultaneously on the same machine. We will refer to these subsets as tasks combinations. There are $m = 2^n - 1$ possible combinations (subtracting $\emptyset$ combination). We denote each combination as S_j, $S_j \in 2^T$, where 2^T is a set of all subsets of T (power set). $|S_j|$ is the number of tasks in S_j.

Tasks run in combinations at their own processing speed (measured in work unit per time units). Without loss of generosity, we will consider the speed of the task, when it was running in ideal conditions (in $S_j = T_i$ combination) to be equal to 1. Otherwise we can divide task required units of work (b_i) by this speed. Denote $a_{i,j}$ as a speedup (or acceleration) of the task T_i in combination S_j compared to ideal conditions. Due to this assumption, we will refer to $a_{i,j}$ as speedup and speed interchangeably (except for the sections related to experiments, where the difference between two terms matters).

Even thought task speed can not increase when it is run in combination with other tasks compared to ideal conditions, and thus values $a_{i,j} \leq 1$, we will still refer to these values as speedup values. This fact can be generalized in a constraint on values $a_{i,j}$ that the task speed decreases with an increase of the number of tasks in combination:

$$a_{i,p} \leq a_{i,q}, \forall (p,q) \mid S_p \subset S_q, i = 1, \ldots, n \tag{1}$$

When $T_i \notin S_j$, then $a_{i,j} = 0$, and when $T_i \in S_j$, then $0 < a_{i,j} \leq 1$. All values of $a_{i,j}$ form a matrix A with dimensions n by m. We will define the speed of a whole combination as a sum of all tasks speed in the combination ($a_j = \sum_{i=1}^{n} a_{i,j}$).

The scheduler during its work assigns combinations of tasks for execution following some scheduling strategy. That is, it produces a sequence of combinations and their processing times $(S_{j_1}, x_{j_1}), (S_{j_2}, x_{j_2}), \ldots$. We will call this sequence of pairs a feasible schedule, when each task completes exactly its required amount of work, i.e. $\sum a_{i,j_k} x_{j_k} = b_i \; \forall i = 1, \ldots, n$.

We consider the problem definition, where tasks preemption is allowed, that is, the scheduler may interrupt on execution of the current combination and run any other combination instead, even if all of the tasks in the current combination were not completed. Because of that, in a feasible schedule some combinations may be repeated multiple times in a schedule.

To compare schedules produced by different strategies, there are different objective functions that can be used. In this paper we will use only makespan objective function (C_{max}) which equals to a completion time of the last task in a schedule.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 September 2021 doi:10.20944/preprints202109.0053.v1

3.1. Optimal strategy

Makespan of a schedule, or a completion time of the last task, can be written as a sum of assigned processing times to each tasks combination: $C_{max} = \sum x_{j_k}$. Since it is the sum, we can reorder its terms and group together the terms corresponding to the same combination, denote $x_j = \sum x_{j_k}, j = 1, \ldots, m$. This allows us to write makespan as a sum of execution times of each combination, i.e. $C_{max} = \sum_{j=1}^{m} x_j$.

The problem of finding the schedule with the minimal makespan value then reduces to finding values of $x_j \geq 0, j = 1, \ldots, m$ which have the minimum sum and define a feasible schedule, i.e. $\sum_{j=1}^{m} a_{i,j} x_j = b_i \; \forall i = 1, \ldots, n$

This gives us a linear programming problem:

$$\begin{array}{ll} \text{minimize} & \sum_{j=1}^{m} x_j \\ \text{subject to} & Ax = b \\ & x \geq 0 \end{array} \tag{2}$$

Solving this problem gives us a vector x^* of an optimal time distribution between combinations. A schedule can be reconstructed from x^* by running each combination S_j for x_j^* time (if $x_j^* > 0$) in any order. In this case the schedule would be feasible and it will have the minimum makespan. In this paper we will refer to this strategy as an optimal (OPT) strategy, and its makespan value as C_{max}^{OPT}.

This optimal strategy is considered an offline strategy, as it requires all information (matrix A and vector b) to be known in advance before running any tasks. Because of that, it can not be used in practice but still it can be used as reference point for evaluating of other strategies.

3.2. Online strategy

Let's consider online formulation of this problem, when the amount of work required for completion of each task (i.e. vector b) is unknown. Values of matrix A are still known at the time 0.

To solve this, we propose to use an online strategy that always runs combination with the maximum speed. We will refer to this strategy in the paper as FCS – Fastest Combination Speed first.

This strategy always selects a combination of remaining tasks with a maximum sum of tasks processing speed. In case there are multiple combinations with the same speed, it may choose any one of them. The selected combination runs until completion of any of its tasks. After that, combinations that contain completed tasks are removed from the search domain, and a new combination with the maximum speed is found among the remaining combinations. Then, a new combination is scheduled for the next iteration and this process continues until completion of the last active task.

3.3. Competitive ratio of the FCS strategy

To evaluate FCS strategy we will derive an upper bound on its competitive ratio. We define it as a ratio of makespan of an online strategy to the makespan of the optimal strategy, when both strategies are computed on the same input.

The problem of finding an upper bound of competitive ratio of FCS strategy boils down to finding values of the matrix A and vector b for which ratio of FCS makespan and optimal makespan has the maximum value. Matrix A and vector b has constraints on their values as defined in the beginning of this section.

Denote the total amount of work of all task as $D = \sum_{i=1}^{n} b_i$, makespan of FCS schedule as C_u and makespan of an optimal schedule as C_v. Combination with the maximum speed will have the index j^* and its speed would be equal to a^*, i.e. $a_{j^*} = a^* = max_{j=1,\ldots,m} a_j$.

In general, FCS schedule (u) and optimal schedule (v) has forms as shown in figure 1. FCS schedule runs combination with maximum speed a^* for u^* time units, until completion of any of its tasks. After that, the schedule proceeds with the remaining combinations of active tasks until all work is completed. Denote $a_u \leq a^*$ as an average speed of all

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 September 2021 doi:10.20944/preprints202109.0053.v1

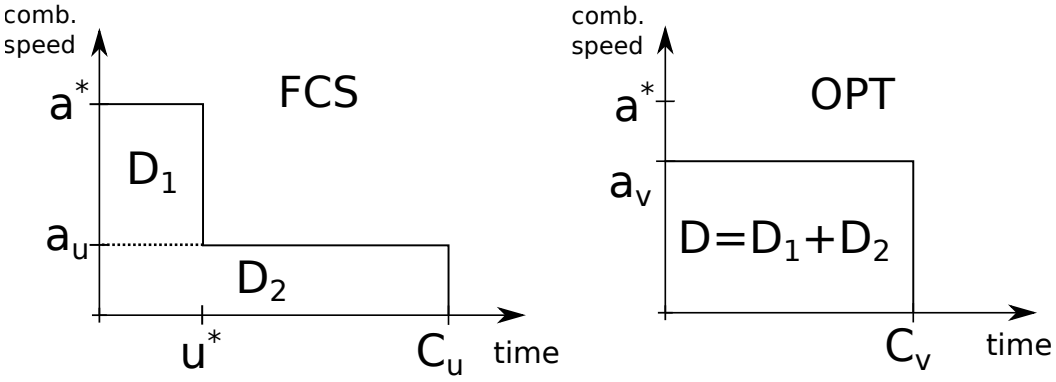

Figure 1. General forms of schedules produced by FCS (left) and OPT (right) strategies.

remaining combinations until the of the schedule at the time point C_u. Similarly, optimal schedule v runs combinations at the average speed a_v performing D units of work total until completion at time point C_v.

Lemma 1. *Schedules with $a_u = 1$ and $a_v = a^*$ have the maximum competitive ratio and it's value is equal to*

$$\rho = a^* - \frac{u^*(a^* - 1)}{D} \tag{3}$$

Proof. Since both FCS and optimal schedules complete the same amount of work D, we can write

$$D = a^*u^* + (C_u - u^*)a_u = C_v a_v$$

Then competitive ratio can be written as

$$\rho = \frac{C_u}{C_v} = \frac{a_v(D - u^*(a^* - a_u))}{a_u D} \tag{4}$$

To prove the lemma we can analyze its derivates on the variables a_u and a_v that are affected by strategies decisions.

$$\frac{\partial \rho}{\partial a_u} = -\frac{a_v(D - a^*u^*)}{D(u^*)^2}$$

$$\frac{\partial \rho}{\partial a_v} = \frac{D - u^*(a^* - a_u)}{Da_u}$$

The feasible region of these values is defined as $1 \leq a_u \leq a^*, 1 \leq a_v \leq a^*$. $\frac{\partial \rho}{\partial a_v}$ has no special points within a feasible region (otherwise $C_u = 0$). $\frac{\partial \rho}{\partial a_v}$ has a single feasible special point, that exists only if $D = a^*u^*$, i.e. all of the work is completed in the j^* combination. In this case ρ would take its minimum value ($\rho = 1$) as both schedules would be the same.

Within a feasible region $\frac{d\rho}{da_u} < 0$ and $\frac{d\rho}{da_v} > 0$. So the maximum value of ρ is reached at the borders of the region, when a_u has the minimum value (i.e. $a_u = 1$) and a_v has the maximum value ($a_v = a^*$).

By plugging the values $a_u = 1, a_v = a^*$ into (4) we can derive the formula (3) for the competitive ratio.

□

Function ρ is, in general, an unbounded function, as its derivatives do not have any special points withing a feasible region. For example, for fixed $u^* = 1$ (which requires $D > 1$ to be feasible) it can get arbitrary large in direction of D and a^*, as its derivatives would be positive.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 September 2021 doi:10.20944/preprints202109.0053.v1

In the proof of the following theorem we will consider only the schedules that has the form as defined by the lemma. That is, the optimal strategy runs combinations at the average speed a^* and FCS schedule runs remaining combinations (after the one a^* speed) with average speed 1. Along with the feasibility constraints, this form allow us to obtain bounds on the value of D as a function of a^*. Plugging these bounds in competitive ratio formula, in turn, will give us its upper bound.

Theorem 1. *Competitive ratio of FCS strategy is at most 2.*

Proof. To find the upper bound on ρ we will consider only the schedules with $a_u = 1, a_v = a^*$, as shown by the lemma.

Denote the index of the first combination of FCS schedule as j^* and let's assume without loss of generality that first $k > 0$ tasks finish in the S_{j^*} combination, i.e.

$$u^* = \frac{b_1}{a_{1,j^*}} = \cdots = \frac{b_k}{a_{k,j^*}}$$

Remaining tasks $T_{k+1}, \ldots, T_n$ run until completion in FCS in combinations with the speed $a_u = 1$. As FCS at every iteration selects combinations with the largest speed, any combination of $T_{k+1}, \ldots, T_n$ (including the one that were not selected by FCS) would have speed not greater than 1, otherwise they would be selected by FCS and then $a_u > 1$. So we can write that

$$a_j = \sum_{i=1}^{n} a_{i,j} = \sum_{i=k+1}^{n} a_{i,j} = 1, \forall j \mid S_j \in 2^{\{T_{k+1},\ldots,T_n\}} \tag{5}$$

Consider $v = (v_1, \ldots, v_m)^T$ to be a time distribution that forms an optimal schedule (i.e. $C_v = \sum_{j=1}^{m} v_j$) and B is a set of combination indices with non-zero time $B = \{j \mid v_j > 0\}$.

Optimal schedule runs all of the combinations at the maximum speed $a_v = a^*$, which is only possible when $a_j = a^*, \forall j \in B$. Also, because of, that optimal schedule can not include combinations containing only the tasks $T_{k+1}, \ldots, T_n$ as their speed is not greater than 1 (as showed in 5). So, any combination in B contains tasks from both $\{T_1, \ldots, T_k\}$ and $\{T_{k+1}, \ldots, T_n\}$ sets.

If we take any combination in the optimal schedule $S_j \mid j \in B$ and remove the tasks that complete in S_{j^*}, we will form another combination $S_p = S_j \cap \{T_{k+1}, \ldots, T_n\}$ and $S_p \neq \emptyset$. Combination S_p, by construct, has less tasks than S_j and $S_p \subset S_j$, so to the values of speeds of their tasks applies constraint (1), i.e. $a_{i,j} \leq a_{i,p}, i = 1, \ldots, n$. Also, we have showed above that to any subset of $T_{k+1}, \ldots, T_n$, including S_p applies 5, so we can write

$$\sum_{i=k+1}^{n} a_{i,j} \leq \sum_{i=k+1}^{n} a_{i,p} = 1$$
$$\forall j \in B, \text{where } p \mid S_p = S_j \cap \{T_{k+1}, \ldots, T_n\}$$

As vector v is an optimal feasible solution to linear programming problem (2), plugging its values to the system $Ax = b$ and adding together rows corresponding to $\{T_{k+1}, \ldots, T_n\}$ rows gives us:

$$\sum_{j \in B} v_j \sum_{i=k+1}^{n} a_{i,j} = \sum_{i=k+1}^{n} b_i$$

Lets define D_1 and D_2 as $D_1 = \sum_{i=1}^{k} b_i$ and $D_2 = \sum_{i=k+1}^{n} b_i$, so that $D = D_1 + D_2$. This allows us to obtain bounds on D:

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 September 2021 doi:10.20944/preprints202109.0053.v1

$$\begin{aligned} D &= D_1 + \sum_{j \in B} v_j \sum_{i=k+1}^{n} a_{i,j} \\ &\leq D_1 + \sum_{j \in B} v_j \\ &= D_1 + C_v \\ &= D_1 + \frac{D}{a^*} \\ D &\leq D_1 \frac{a^*}{a^* - 1} \end{aligned}$$

The amount of time required for completion of S_{j^*} in FCS schedule can also be expressed from D_1 as $u^* = \frac{D_1}{a^*}$. Plugging u^* and bounds on D into completive ratio formula (3) gives us:

$$\rho = a^* - \frac{u^*(a^* - 1)}{D} \leq 2 - \frac{1}{a^*}$$

Which is less than 2 for any $a^* > 1$. □

This result of this theorem tells us that the competitive ratio is bounded to a constant and it would not be greater than 2 for any number of tasks, required amount of work for each tasks or processing speed values. This also has a practical result, as FCS strategy can be transformed into a scheduling algorithm that implements a search of combination with the maximum speed.

4. Online optimization algorithm

FCS strategy can not be applied in practical implementation as it requires values of matrix A to be known before processing of any of the task begins. Instead, values of A can be measured when a task combination is running. Since tasks combinations can be preempted at any time, we can transform FCS strategy into a searching algorithm that finds combination with the largest speed and then executes it until completion of any of its tasks.

To obtain the value of any column of matrix A we need to run corresponding combination for at least some units of time to measure the speed of each task. As during this process tasks are running and completing some units work, it's important to reduce execution of tasks in slower combinations. This makes it a problem of online optimization. We assume additionally that speed measurements are not noisy.

The general idea of the online search algorithm is to iteratively run different tasks combinations as determined by acquisition function. When combination is run, its tasks speed are being measured and then at the following iterations this data is used to provide boundaries for the unknown values. Acquisition function matches a single value to every combination based the boundaries values of all combinations and the optimal value found on the previous iteration. Then, search algorithm selects a combination with the highest value of an acquisition function for running at the next iteration.

In this paper we cover several scheduling strategies based on a search algorithm. Although these strategies work with values of the matrix A and vector b not being available at start, we presented them as theoretical results that can not be applied to the practical problem without additional model modifications. The reasons for that is, first, we are already working with a stationary problem, and, second, that we introduce an additional assumption of non-noisy measurements of task speed values. We still present these results in the paper as they can be used as a base for further model improvements.

4.1. Computing boundaries for incomplete data

To estimate missing values of matrix A we will use constraint (1) that processing speed of a task in a combination decreases when other tasks are added to it. Because of that, unknown speed of the task can be interpolated as a non-increasing function of the number of tasks.

Suppose we known the speed of T_i in combinations S_p and S_q and need to estimate a value in combination S_j, where $S_p \subset S_j \subset S_q$. Since speed is non-increasing, we have

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 September 2021 doi:10.20944/preprints202109.0053.v1

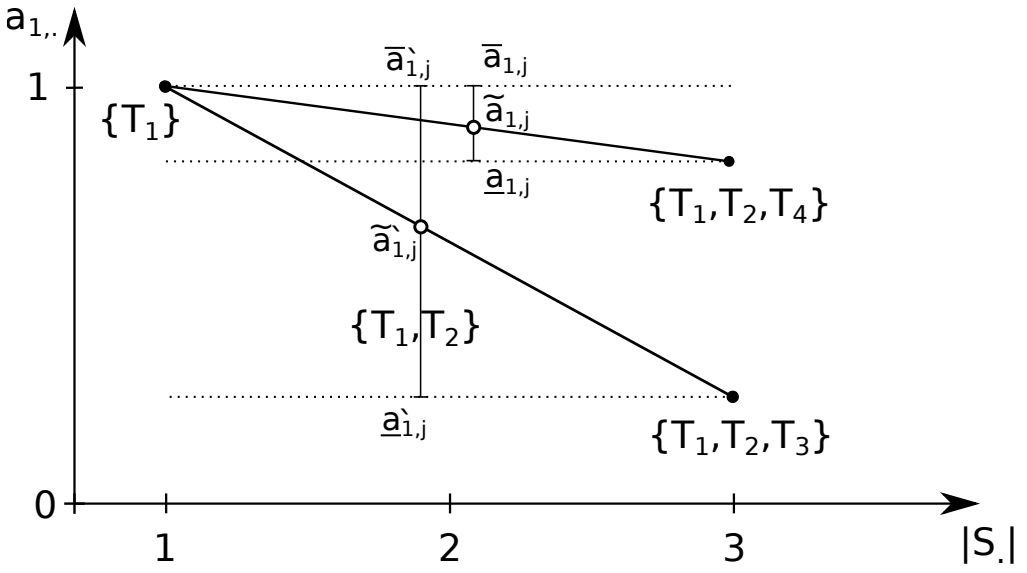

Figure 2. An example of ambiguity in interpolation of task speed values. Speed of T_1 in $S_j = \{T_1, T_2\}$ can be obtained from $\{T_1\}$ and either $\{T_1, T_2, T_3\}$ or $\{T_1, T_2, T_4\}$. Combination of $\{T_1, T_2, T_4\}$ tasks will be used for interpolation of the mean value as it produces a smaller boundaries interval.

$a_{i,p} \leq a_{i,j} \leq a_{i,q}$. We will use linear interpolation to find an approximate value, denoted as $\tilde{a}_{i,j}$:

$$\tilde{a}_{i,j} = a_{i,q} + \frac{a_{i,p} - a_{i,q}}{|S_p| - |S_q|}\Big(|S_j| - |S_q|\Big)$$

This linear dependency on the combination size can be noticed on experimental data (figure 4), which we will describe later in the paper.

With the assumption that values are measured without any noise and monotonicity constraint, we can claim that $a_{i,j} \in [a_{i,p}, a_{i,q}]$ almost surely. We will denote computed upper and lower boundary values as $\underline{a}_{i,j}$ and $\overline{a}_{i,j}$ respectively. For the sums of the corresponding values for all tasks in a combination, we will use $\underline{a}_j, \overline{a}_j, \tilde{a}_j$ notation.

Using this method, the value of $a_{i,j}$ can be obtained from multiple different pairs of combinations of S_p and S_q. To remove this ambiguity, we will use S_p with the largest number of tasks, such that $S_p \subset S_j$ and S_q with the smallest number of tasks, such that $S_j \subset S_q$. Then, among the possible S_p and S_q with the same size we will use the one with the minimal boundary interval width $(a_{i,p} - a_{i,q})$. An example when such ambiguity is possible is shown in figure 2.

4.2. Search algorithm acquisition functions

Implementing a search algorithm requires to define an acquisition function that would select the next task combination for evaluation. Acquisition function matches a single value to every combination based the confidence intervals values of all combinations and the decision on the previous iteration. Then, search algorithm selects a combination of highest value of an acquisition function for running at the next iteration.

We have implemented three acquisition functions: probability of improvement, expected improvement and upper confidence bound. To use these functions as random functions we will treat $a_{i,j}$ as random variables. Additionally, we will assume that $a_{i,j}$ are independent and have a normal distribution with the mean value $\tilde{a}_{i,j}$ and variance $\sigma^2_{i,j}$. As formally we can not apply these assumptions to the bounded and monotonic values, we will relax constrains on these values by defining variance as a function of interval bounds in the following way.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 September 2021 doi:10.20944/preprints202109.0053.v1

Since $a_{i,j}$ is a random variable with a normal distribution, it will exceed $\tilde{a}_{i,j} + \Phi^{-1}(k)\sigma_{i,j}$ with probability $1-k$, where $\Phi^{-1}(k)$ is a k^{th}-quartile of a standard normal distribution. We will denote the width of k^{th}-quartile confidence interval as $\Delta_{i,j}$ and compute it as

$$\Delta_{i,j} = \min\{\overline{a}_{i,j} - \tilde{a}_{i,j}, \tilde{a}_{i,j} - \underline{a}_{i,j}\}$$

This way, it will ensure that confidence interval will increase when the distance between available data points increases and that its shape has a form where monotonicity constraint holds. As $\delta_{i,j}$ can only be positive, we limit k to $0.5 < k < 1$, then, we can write variance of $a_{i,j}$ as

$$\sigma_{i,j} = \frac{\Delta_{i,j}}{\Phi^{-1}(k)} \tag{6}$$

Later in the paper we will use k as a tuning parameter for acquisition function. As task speed values has normal distribution, we can claim that combination speed values also have a normal distribution with mean value $\tilde{a}_j = \sum_{i \in S_j} \tilde{a}_{i,j}$ and variance $\sigma_j^2 = \sum_{i \in S_j} \sigma_{i,j}^2$.

We define probability of improvement (PI_j), expected improvement (EI_j) and upper confidence bound (UCB_j) of a combination S_j the in following way:

$$PI_j = \begin{cases} \Phi\left(\frac{\tilde{a}_j - a^+}{\sigma_j}\right) & \sigma_j \neq 0 \\ 0 & \sigma_j = 0, \text{ and } \tilde{a}_j < a^+ \\ 1 & \sigma_j = 0, \text{ and } \tilde{a}_j \geq a^+ \end{cases}$$

$$EI_j = \begin{cases} (\tilde{a}_j - a^+)\Phi\left(\frac{\tilde{a}_j - a^+}{\sigma_j}\right) + \sigma_j \phi\left(\frac{\tilde{a}_j - a^+}{\sigma_j}\right) & \sigma_j \neq 0 \\ \tilde{a}_j - a^+ & \sigma_j = 0 \end{cases}$$

$$UCB_j = \tilde{a}_j + \sigma_j$$

Where $\Phi(x)$ is a standard normal distribution cumulative distribution function, $\phi(x)$ is standard normal distribution density function and a^+ denotes the estimated maximum value from the previous iteration. A tuning parameter of these acquisition function is a probability of task speed value being outside of $\tilde{a}_{i,j} \pm \delta_{i,j}$ interval, i.e. value k. With an increase of parameter k the value of combination speed variance (σ_j^2) decreases.

Simulation results for of different acquisition function are described later in the paper in section 7.

5. Measuring task processing speed in experiments

When tasks running in parallel use shared resources their performance may decrease as bandwidth for these resources is limited and it is being shared between tasks. Examples of such shared resources may include memory bus, shared cache levels, disk bandwidth, network card bandwidth, etc. Such situations can be observed when CPU instruction requiring accesses to the shared resources may take more CPU cycles to complete, i.e. rate of finishing instructions (instruction per cycle, IPC) would decrease. IPC is also not specific to any shared resources in particular, but shows overall speed of instructions processing.

Another reason of performance degradation due to co-scheduling is when operating system scheduler runs multiple threads on the same CPU core sharing cputime between them. In this case, IPC of an active threads may not change, but their portions of cputime may decrease resulting in increase of overall execution time.

To measure task processing speed (in an absolute value) we propose to use such metrics as IPC multiplied by cputime as it is affected by both the operating system scheduler and concurrent access to shared resources. Values of IPC and cputime can be measured during tasks run-time with a low overhead by accessing CPU performance counters and thread-specific data provided by operating systems (e.g, ProcFS in Linux).

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 September 2021 doi:10.20944/preprints202109.0053.v1

When processing speed is measured in this way at time t, the unit of work would be CPU instructions ($inst(t)$) and unit of time would be CPU cycles $cycl(t)$. Both of them are provided by CPU performance counters for the time when task was running in user space. Cputime $cpu(t)$ over sampling period Δt would an unitless value. Assuming that all of these values are cumulative, then we can write the formula for estimating task processing speed:

$$v(t) = \frac{inst(t) - inst(t - \Delta t)}{cycl(t) - cycl(t - \Delta t)} \frac{cpu(t) - cpu(t - \Delta t)}{\Delta t} \tag{7}$$

6. Benchmark applications

To collect experimental data of task processing speed values we have used NAS Parallel Benchmark (NPB) [16] and Parsec [17] test suites along with our own test applications. These benchmarks were chosen as they mimic workload of applications that are common for HPC field and cover many different types of parallelism patterns granularities, inter-thread data exchange, synchronisation patterns and have bottlenecks on different types of resources (there are CPU-, memory- and IO-intensive applications).

Due to the assumption of stationary scheduling problem, among these benchmarks we have selected only those with constant or periodic speed profiles. The resulting of benchmarks is presented in table 1. Datasets of each benchmark were tuned to so that each task runs for the same amount of time on the same computational resources.

Table 1: List of benchmarks used in experiments.

Name	Suite	Description
bt	NPB	Block Tri-diagonal solver
ft	NPB	Discrete 3D fast Fourier Transform
lu	NPB	Lower-Upper Gauss-Seidel solver
sp	NPB	Scalar Penta-diagonal solver
fm	Parsec	Frequent Pattern Growth algorithm (freqmine)
so	Parsec	HJM algorithm for pricing swap options (swaption)
vp	Parsec	Image processing pipeline (VIPS library)
sc	Parsec	Online clustering problem in data mining (streamcluster)
ff		Decoding of video file (ffmpeg)
rt		Ray tracing algorithm on CPU

For collecting experimental data we used a single node with Intel Xeon E5-2630 processor with 10 cores and 2 threads per core. In experiments each benchmark was limited to a certain number of threads by changing parallelism parameters in the benchmark application. Threads were not bound to specific cores and could be migrated between cores by operating system scheduler (Linux CFS). In all of the experiments, each application had enough memory and swap was never used.

6.1. Evaluation of task processing speed measurements

We have used Linux perf (perf_event_open system call) to access CPU performance counters to monitor number of instructions and number of CPU cycles of each thread in its run-time. Counters values were reported only for the instructions that were running in the user-space (as opposed to kernel-space). Values of cputime in both user-space and kernel-space of each thread were obtained from ProcFS pseudo-filesystem provided by Linux kernel. These values were used to compute processing speed of each thread of running benchmark application. Then, to find processing speed the task itself, values of different threads were averaged.

This approach still allows to measure processing speed of each task in the user-space and in the kernel-space, but with less accuracy. When an instruction takes more CPU cycles due to the shared access to the same resource from another task, IPC value would be affected. When application issues an interrupt for a system call that is done is the kernel

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 September 2021 doi:10.20944/preprints202109.0053.v1

space, and it takes more time due to the concurrent access to the shared resources, then this situation would be reflected only by the change in kernel-space cputime. This was done deliberately to avoid using two separate processing speed metrics for kernel-space and user-space. Additionally, it did not have any noticeable affect on overall results of experiments.

To show that formula 7 measures speed of processing, we have compared the change of its average value with the change of the total processing time of each benchmark in different conditions. At first, we measured an absolute value of processing speed and total execution time of each benchmark in ideal conditions, and then, when it was running in combinations with other benchmark applications. After that we have compared the ratio of processing speed in ideal conditions to the processing speed in co-scheduling conditions with the ratio of execution time in co-scheduling conditions to the execution time in ideal conditions. These ratios should be equal for tasks that perform the same total amount of work units (CPU instructions) regardless of the processing speed.

Benchmark tasks that we used perform the same number of CPU instructions regardless of the available resource bandwidth they have, i.e. the amount of work units does not change when benchmarks are running in different combinations (this was shown in our previous paper [18]). This property may not hold, when a task performs active waiting on its resources in user-space. In this case, proposed experiments can not be used as the number of reported CPU instructions increase when the task was waiting on a resource and not advancing its state. For some benchmarks that use OpenMP for parallelisation, we had to ensure that active waiting on threads synchronizations primitives is disabled.

We run these experiments in the following way. For each task that we were measuring, we have generated a set of all possible combinations with other tasks. In these combinations some tasks may be repeated multiple times, and the number of tasks in each combination could be from 1 to 10. We did not measure all of the combinations, as there are a lot of them and we need to run each task until completion, instead we run 300 of randomly sampled combinations. For each run we made sure that the task we are measuring finishes the first, otherwise we would measure it multiple combinations and its speed would not be consistent.

We run experiments for situations when each task requires the same number threads and when tasks require random number of threads. For the same number of threads, we ran two cases, when each task uses 2 threads and when they use 6 threads. In the first case there are enough CPU cores to run all threads without cputime being shared, and in the second case cputime is shared between threads. When cputime is not shared, its value remains constant and task speed changes only due to the changes in IPC. When cputime is shared, its value may change in different scheduler states and task speed value would change accordingly. For the situation with the random number of threads, each task may require form 2 to 6 threads.

Results of experiments with a random number of threads per tasks are shown in figure 3. Since plots for other experiment results look the same, we present them as table 2 with linear regression values. Results show that for all benchmarks tasks the ratio of processing times in co-scheduling and ideal conditions matches exactly with task speedup. That is, task speed measured using formula (7) in run-time is proportional to speed measured after completion (by diving work units by total time).

7. Numerical simulation results

To evaluate search algorithm and FCS strategy and to compare them with an optimal strategy, we have implemented simulation software ([19]). It works by generating random task speedup values in each combination ($a_{i,j}$) and task work units (b_i). Generated values complies to constraints defined in section 3. These values are then used to find makespan of an optimal strategy by solving linear programming problem, and makespan of FCS and online search strategies by running their simulations.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 September 2021 doi:10.20944/preprints202109.0053.v1

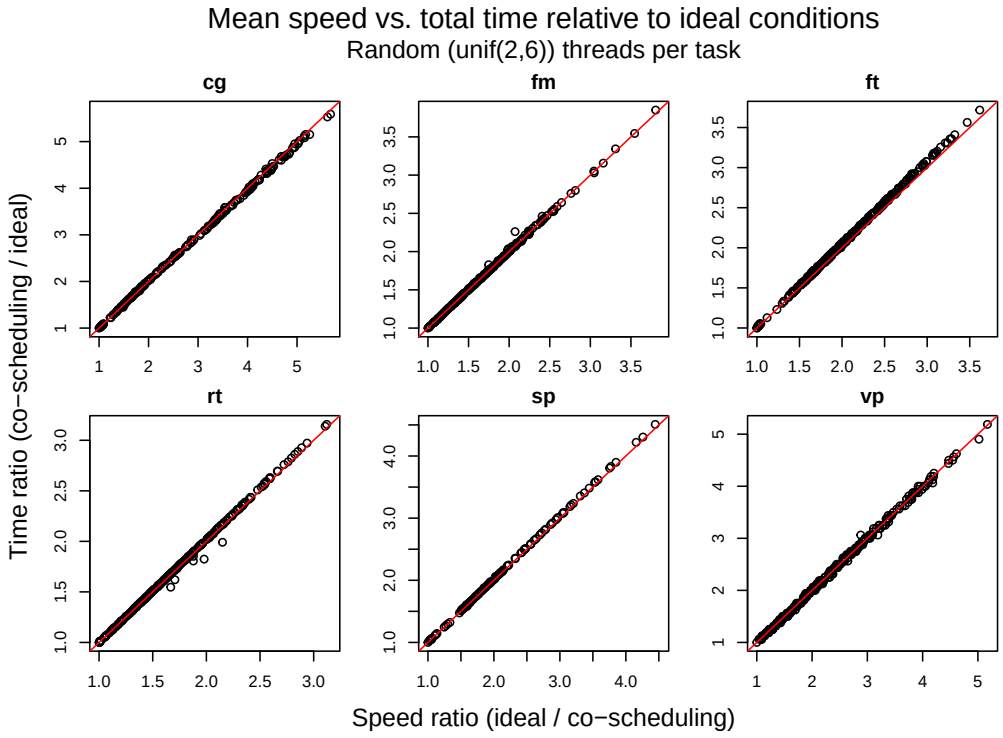

Figure 3. Scatter plots of processing time ratio versus processing speed ratio for some benchmark tasks. Each point corresponds to task measurement in a combination with other tasks. Red line is region where ratios are equal.

Values of b_i were drawn from the uniform distribution from 100 to 200 work units. Values of task speed in ideal conditions were fixed to 1, so that task speed in any combination would be the same as task speedup value. Task speedup values were generated by the following procedure. It works by iterating over all combinations in the order of increasing combination sizes, starting from 2. At each iteration the maximum allowed speedup of the task is found as a minimum speedup among the combinations with the lesser size. Then, this value is multiplied by uniform random number generated between α and 1. The resulting value gives a random speedup value that complies to monotonicity constraint (1).

The aforementioned α constant corresponds to the maximum slope of task speed value when the size of combination increase. We measured task speed as a function of combination size in experiments, and results for some benchmarks are shown in figure 4. Each plot corresponds to benchmark application running on 6 cores (out of 20) in combinations with other benchmarks running on a random number of cores from 2 to 6. The data was collected in the following way. The combination with the maximum number of tasks was picked at random and the speed of the target task was measured in this combination. Then, a random task was removed from this combination and the target task was measured in a new combination. This process was repeated until the target task was the only task in the combination, i.e. it was running in ideal conditions. At least 30 combinations with the largest size were generated. Each plot shows how distribution speed value of the task changes depending on the combination size.

Task speedup and combination speedup values corresponding for different α values are shown in figure 5. The smaller values of α correspond to the smaller values of task and combinations speeds. While all task speed values decrease, their sum, i.e. combination speed, may increase and may reach a peak value for combinations with less than n tasks.

In simulations we used multiple different ranges of α values. Within each range we have chosen a value for each task uniformly at random. The range from α from 0.75 to 0.85

 doi:10.20944/preprints202109.0053.v1

Task	2 Threads		6 Threads		From 2 To 6 Threads	
	Model	R^2	Model	R^2	Model	R^2
bt	0.984 x + 0.017	0.9997	0.969 x + 0.03	1	0.97 x + 0.029	0.9999
cg	0.984 x + 0.013	0.9996	1.016 x - 0.038	0.9992	1.02 x - 0.042	0.9997
ft	0.995 x + 0.005	0.9996	0.958 x + 0.026	0.9998	0.963 x + 0.032	0.9996
sp	0.994 x + 0.007	0.9998	0.977 x + 0.018	0.9999	0.982 x + 0.017	0.9999
rt	1.015 x - 0.018	0.9943	0.974 x + 0.034	0.9993	0.983 x + 0.02	0.9982
fm	0.993 x + 0.006	0.9999	0.995 x - 0.003	0.9999	0.995 x + 0.005	0.9989
sc	0.958 x + 0.039	0.9986	0.837 x + 0.22	0.9995	0.858 x + 0.188	0.997
vp	1.02 x - 0.008	0.9958	1.016 x + 0.035	0.9501	0.999 x + 0.005	0.9986
ff	0.925 x + 0.065	0.9963	0.843 x + 0.165	0.9956	0.974 x + 0.07	0.9986
so	1.111 x - 0.114	0.9868	0.984 x + 0.025	0.9992	0.984 x + 0.035	0.9964

Table 2: Linear regression models fitted on experimental data of processing speed ratio as a function of time ratio. The table contains model coefficient and intercept values with R squared value. Data is fitted for the measurement of each benchmark task run in combinations with other benchmarks.

was also simulated as it produces the task speed values that are closest to task speedup values measured in experiments.

We have generated 50 systems with parameters as described above. Each system has 10 tasks, similar to experiments setup. We did not vary the maximum number of tasks in the system, as possible curves of combination speed values (right plot in 5) could be generated by changing α and n changes only the discretization of these curves.

For these generated systems we simulated FCS and search-based strategies (UCB, PI and EI) and computed their competitive ratios to the optimal strategy. In the search-based strategies, each combination required at least 10 time unit of run time to measure tasks speed values, which equals to 5-10% of the amount of work in ideal conditions. This value was made deliberately large to exaggerate an effect of sub-optimal decisions of the search algorithm.

Results of the simulations for these systems are shown in figures 6 and 7. Competitive ratios of UCB, PI and EI strategies as a function speedup value ranges (α) for different values of variance parameter (k^{th}-quartile in (6)) are shown in figure 6. For UCB strategy, it can be noticed that its competitive ratio decrease with an increase of k, i.e. with a decrease of confidence interval width. Competitive ratio of PI and EI strategies are not affected by the changes in k. The effect of the α on the competitive ratio is not monotonic for all strategies and the larger deviation from the optimal strategies is reached at the border values of α ranges.

Figure 7 compares minimal competitive ratio (which is achieved at $k = 0.99$) as a function of α range. As expected, search based strategies performed worse than FCS strategy. The maximum competitive ratio of 2.6 was reached by UCB strategy for $\alpha \in [0.45, 0.45]$. For the lower ranges of α the difference between search-based strategies is less pronounced. Based on these simulation results, we can claim that PI strategy produces better results for all tested ranges of task speed values.

8. Discussion

In this paper with formalized co-scheduling problem. We proposed several strategies: an optimal strategy, an online strategy (FCS) and search based strategies (PI, EI, UCB). An optimal strategy requires all task information to be available at start and it solves linear programming problem to construct a schedule. FCS strategy schedules next the combination with the maximum processing speed, it works with assumption that required amounts of work for each task are unknown.

We showed theoretically that FCS strategy provides schedules with a competitive ratio that are less than 2. That is, makespan of FSC strategy can not be larger than makespan of an optimal strategy by more than two times. This allowed us to solve co-scheduling

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 September 2021 doi:10.20944/preprints202109.0053.v1

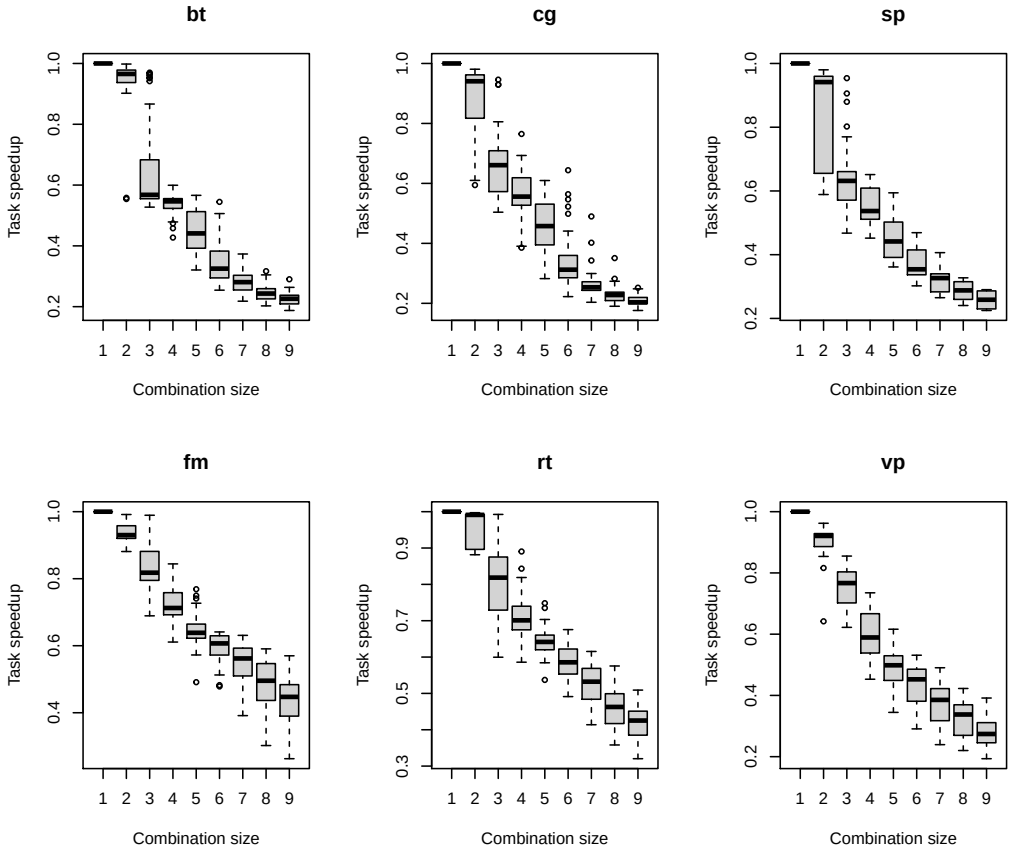

Figure 4. Box with whiskers plots of task speed as a function of combination size measured on benchmark applications.

problem using heuristic strategies that approximate FSC. They work by implementing stochastic search algorithm which samples different task combinations until it find the one with the larges processing speed. Different implementation of the search algorithm are possible depending on the acquisition function they are using.

To formalize co-scheduling problem as a stochastic optimization problem, we represented task processing speed as a random variable with a normal distribution. Mean values and variance parameters of this normal distribution are derived as a relaxation of monotonicity constraints defined by co-scheduling problem. We defined three acquisition functions: probability of improvement (PI), expected improvement (EI) and upper confidence bound (UCB). These acquisition functions had a tuning parameter which affected scale of variance values.

We implemented numerical simulations to find competitive ratios of search-based strategies. For simulations input, we generated random values of task processing parameters that comply with model constraints. Random task processing speed values had a parameter which affected the rate of task speedup when combination size increases. Simulations were run for different ranges of speedup rates and different values of acquisition function parameter.

Simulation results showed that there is no effect of variance parameters for PI and EI strategies and that UCB works better with smaller variances (i.e. tighter upper bounds). In general, PI produced better results for all simulated ranges of speed speedup rates. UCB and EI showed worse competitive ratios for larger speedup rates ranges.

Considering assumptions on the model, PI strategy can be used for schedulers implementations, according to simulations. In practice, model definition must first be improved to account for noisy measurements and the change of task speed values in time. Search-

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 September 2021 doi:10.20944/preprints202109.0053.v1

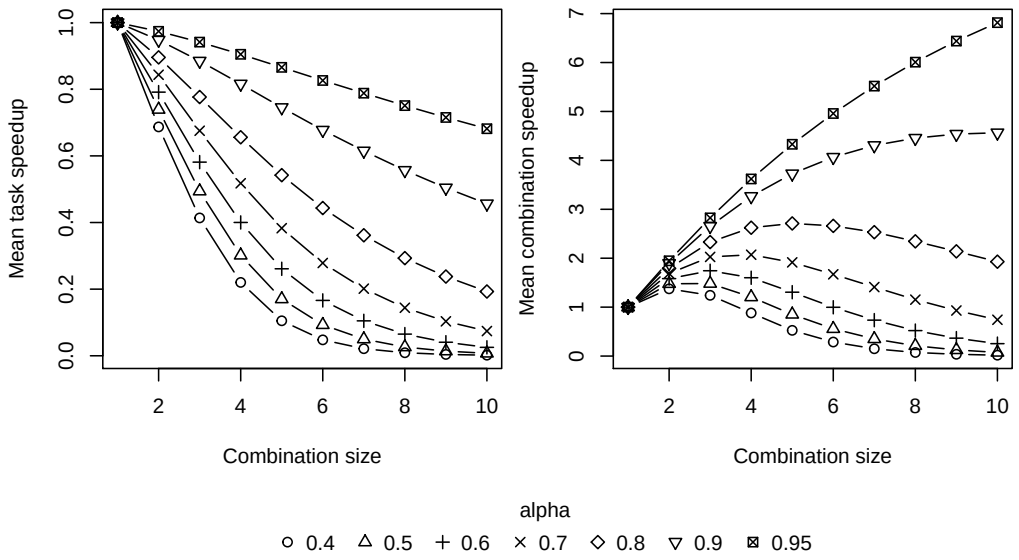

Figure 5. Generated task speedup values and corresponding combination speedup values for different speedup rate (alpha) parameter. Averaged values across all combinations with the same size shown as a single data point.

based strategies can be easily adapted to take into account theses assumptions by changing acquisition functions, but showing that competitive ratio of FCS strategy obtained in this paper holds for these assumptions is not trivial. Now, we leave at as a future work.

We proposed a method of measuring task processing speed in run-time using CPU performance counters (IPC) and operating system scheduler information (cputime fraction). To show that the value being measured represent processing speed, we used a set of benchmark application that mimic real HPC workloads. Experimental results showed that between different task combinations the change of this value matched with the inverse change of processing time. For applications with a constant amount of work (that does not depend on available resources) this implies that this value measured in run-time proportional to the processing speed. Experimental data provided us with with a shape of speedup curve that we aimed to approximate when generated input data for numerical simulations.

Implementation of search-based scheduling strategies relies on the ability of the operating system scheduler of controlling tasks preemption. That is, it should provide an interface for suspending and continuing an execution of running tasks. In Linux, it is possible by using freezer control group, which is available in kernel releases starting from version 2.6.

The disadvantage of this approach is that all tasks that potentially can be run simultaneously, must in a runnable state when the scheduler starts, so that their processing speed can be measured. As main memory of the node is limited, this limits the number of tasks in a combination. Some task combinations can be practically infeasible, even if they have a small number of tasks. These practical restrictions can be represented in the model by changing a set of combinations from 2^T to a custom set of subset. This would reduce the number of subset, but would not affect results obtained in the paper.

Other scheduling strategies that potentially may lead to practical results can also be considered within this model definition and assumptions. For example, values of total mount work for each task (vector b) may be considered a known quantity. In practice, it can be done by analysing historical data of previous runs of the same task. With the assumptions that each task processes constant amount of work units, these values can be reliably used for making scheduling decisions.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 September 2021 doi:10.20944/preprints202109.0053.v1

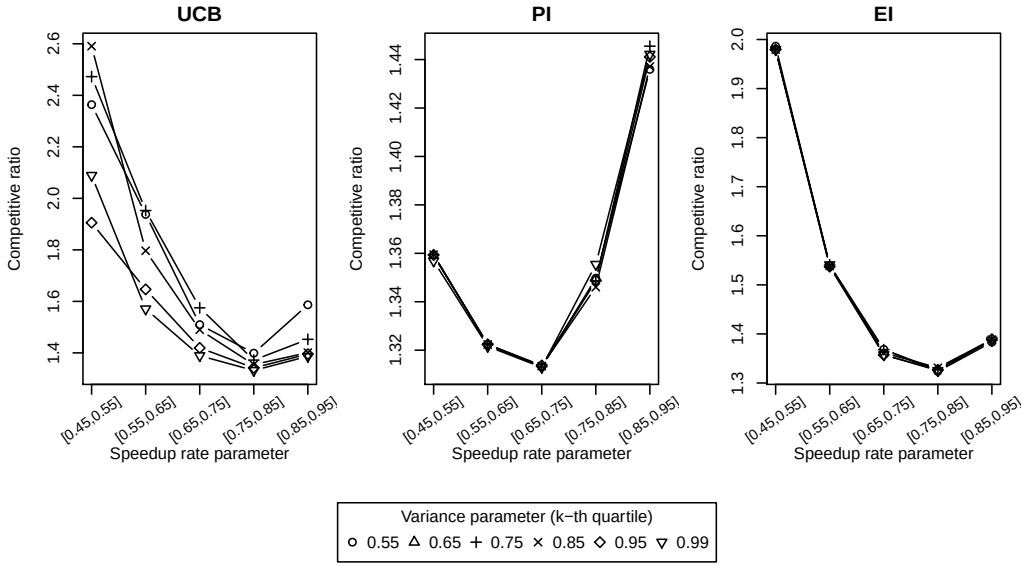

Figure 6. Competitive ratio of search algorithms with different acquisition functions. Values are presented for each range of alpha values and confidence intervals tuning parameters.

It is possible to implement proposed strategies as an extension to a batch scheduler (e.g. as a SLURM extension) or as a stand-alone scheduler, that would work on top of existing batch scheduler. The later approach may require user-writable control group to implement task preemption. Alternatively, task preemption can be implemented using signals, which does not require interventions from privileged system users. Linux performance (perf) counters can be used for measuring task processing speed, which in general does not require any privileged access. There is no restrictions on the applications that can be run using this scheduling approach, as it does not require any code or binary files modifications. The implementation of the scheduler we leave outside of the scope of this paper as a future work.

9. Conclusion

In this paper we defined a model for solving co-scheduling problem and proposed multiple scheduling strategies: an optimal strategy, an online strategy (FCS) and heuristic strategies (EI, PI and UCB). The optimal solutions is found by reducing the problem to a linear programming problem, which requires all task processing speed and amounts of work to be known in advance. FCS strategy is defined with an assumption that only task processing speed is available, while required work units are unknown. Heuristic strategies work with an assumption that no information is available at start, but task processing speed can be measured in tasks run-time.

We showed theoretically that FCS strategy produces schedules that are at most 2 times worse than an optimal strategy. This allowed us to solve co-scheduling problem using heuristic strategies that approximate FCS. We defined these strategies as implementations of stochastic optimization algorithms with different acquisition functions. To apply these optimization algorithms we defined non-deterministic version of co-scheduling problem by treating each task processing speed value as a random variable and relaxing constraints on its value by defining its variance.

We used numerical simulations to compare all strategies with an optimal strategy and to show how heuristic strategies behave for different values of tuning parameter and problem inputs. Results showed that PI strategy produced a lower values of completive ratios than EI and UCB strategies for almost all problem input data.

We also proposed a method for measuring task processing speed in its runtime and evaluated it using benchmark HPC applications in different environments. We showed

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 September 2021 doi:10.20944/preprints202109.0053.v1

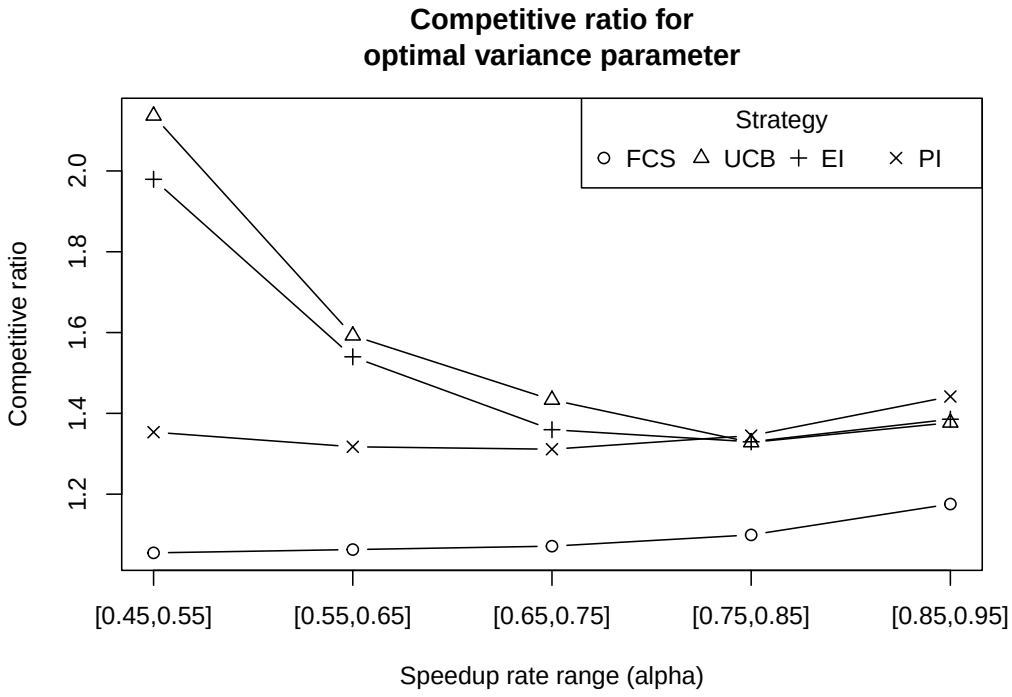

Figure 7. Comparison of the lowest competitive ratio values for each slowdown rate range.

that this methods measures values with a high accuracy, which allows to apply proposed scheduling strategies in scheduler implementations.

Acknowledgments: Research has been supported by the RFBR grant No. 19-37-90138.

1. Trinitis, C.; Weidendorfer, J. *Co-scheduling of HPC applications*; Vol. 28, IOS Press, 2017.
2. Trinits, C.; Weidendorfer, J. First Workshop on Co-Scheduling of HPC Applications (COSH 2016).
3. de Blanche, A.; Lundqvist, T. Node Sharing for Increased Throughput and Shorter Runtimes–an Industrial Co-Scheduling Case Study. Proceedings of the 3rd Workshop on Co-Scheduling of HPC Applications (COSH 2018), 2018.
4. Breslow, A.D.; Porter, L.; Tiwari, A.; Laurenzano, M.; Carrington, L.; Tullsen, D.M.; Snavely, A.E. The case for colocation of hpc workloads. *Concurrency and Computation: Practice and Experience Preprint* **2012**.
5. Breitbart, J.; Pickartz, S.; Lankes, S.; Weidendorfer, J.; Monti, A. Dynamic Co-Scheduling Driven by Main Memory Bandwidth Utilization. *2017 IEEE International Conference on Cluster Computing (CLUSTER)* **2017**. doi:10.1109/cluster.2017.59.
6. Zacarias, F.V.; Petrucci, V.; Nishtala, R.; Carpenter, P.; Mossé, D. Intelligent Colocation of Workloads for Enhanced Server Efficiency. 2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD). IEEE, 2019, pp. 120–127.
7. Aupy, G.; Benoit, A.; Dai, S.; Pottier, L.; Raghavan, P.; Robert, Y.; Shantharam, M. Co-scheduling Amdahl applications on cache-partitioned systems. *The International Journal of High Performance Computing Applications* **2018**, *32*, 123–138.
8. Aupy, G.; Benoit, A.; Goglin, B.; Pottier, L.; Robert, Y. Co-scheduling HPC workloads on cache-partitioned CMP platforms. *The International Journal of High Performance Computing Applications* **2019**, *33*, 1221–1239.
9. Pottier, L. Co-scheduling for large-scale applications: memory and resilience. PhD thesis, Université de Lyon, 2018.
10. Li, Y.; Sun, D.; Lee, B.C. Dynamic colocation policies with reinforcement learning. *ACM Transactions on Architecture and Code Optimization (TACO)* **2020**, *17*, 1–25.
11. Snavely, A.; Mitchell, N.; Carter, L.; Ferrante, J.; Tullsen, D. Explorations in symbiosis on two multithreaded architectures. Workshop on Multi-Threaded Execution, Architecture, and Compilers, 1999.
12. Snavely, A.; Tullsen, D.M. Symbiotic jobscheduling for a simultaneous multithreaded processor. Proceedings of the ninth international conference on Architectural support for programming languages and operating systems, 2000, pp. 234–244.
13. Parekh, S.; Eggers, S.; Levy, H.; Lo, J. Thread-sensitive scheduling for SMT processors, 2000.
14. Jain, R.; Hughes, C.J.; Adve, S.V. Soft real-time scheduling on simultaneous multithreaded processors. 23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002. IEEE, 2002, pp. 134–145.

15. Eyerman, S.; Michaud, P.; Rogiest, W. Revisiting symbiotic job scheduling. 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). IEEE, 2015, pp. 124–134.
16. Bailey, D.; Harris, T.; Saphir, W.; Van Der Wijngaart, R.; Woo, A.; Yarrow, M. The NAS parallel benchmarks 2.0. Technical report, Technical Report NAS-95-020, NASA Ames Research Center, 1995.
17. Bienia, C. Benchmarking Modern Multiprocessors. PhD thesis, Princeton University, 2011.
18. Kuchumov, R.; Korkhov, V. Collecting HPC Applications Processing Characteristics to Facilitate Co-scheduling. International Conference on Computational Science and Its Applications. Springer, 2020, pp. 168–182.
19. Kuchumov, R.; Korkhov, V. Co-scheduling numerical simulation source code. https://gitlab.com/mildlyparallel/co-scheduling-simulations, 2021.