





# An Analytical Bound for Choosing Trivial Strategies in Co-scheduling

Ruslan Kuchumov<sup>(✉)</sup>  and Vladimir Korkhov<sup>(✉)</sup> 

Saint Petersburg State University, 7/9 Universitetskaya nab.,  
St. Petersburg 199034, Russia  
st058444@student.spbu.ru, v.korkhov@spbu.ru

**Abstract.** Efficient usage of shared high-performance computing (HPC) resources raises the problem of HPC applications co-scheduling, i.e. the problem of execution of multiple applications simultaneously on the same shared computing nodes. Each application may have different requirements for shared resources (e.g. network bandwidth or memory bus bandwidth). When these resources are used concurrently, their resource throughputs may decrease, which leads to performance degradation.

In this paper we define application behavior model in co-scheduling environment and formalize a scheduling problem. Within the model we evaluate trivial strategies and compare them with an optimal strategy. The comparison provides a simple analytical criteria for choosing between a naive strategy of running all applications in parallel or any sophisticated strategies that account for applications performance degradation.

**Keywords:** High performance computing · Co-scheduling · Scheduling theory · Linear programming

## 1 Introduction

Commonly used job schedulers in HPC do not allow to oversubscribe the same computational resources with multiple jobs. The main reason for that is job performance degradation due to simultaneous access to shared resources, such as CPU cores, shared cache levels, memory bus. Requirements for such resources may depend on job input parameters, on external factors and may change over time, so it is difficult to provide them at jobs start time.

Applications may have different requirements for resources, and some resources may not be fully utilized by one application, but for others they would be a bottleneck. As schedulers in HPC do not allow over-subscription and allocate a whole cluster node to a single job, underutilized resources will be wasted even if there are jobs waiting in the queue for them.

The scheduling strategy of assigning multiple jobs to the same computational resources so that their interference with each other and the degradation of performance is minimal, in the literature is usually referred to as co-scheduling. It

has started to gain interest recently, due to advances in hardware and operating systems support [13].

In this research we are mostly focused on modelling part of co-scheduling. In our previous work [8] we have shown which metrics can be used to estimate application processing speed in co-scheduling environment. We have shown how they can be measured and how they relate to total processing time of the application.

In this paper we focus on the problem of scheduling applications on shared resources. In particular, we have defined application behavior model in co-scheduling environment and formalized a scheduling problem. An optimal strategy for this problem can be found by solving corresponding linear programming problem. Using the optimal solution as a reference we have evaluated two trivial strategies – a round-robin (RR) strategy and naive parallel (NP) strategy. The comparison allowed us to obtain boundaries on application processing speed that can be used for choosing between these strategies.

## 2 Related Work

Problem of co-scheduling started to gain interest in the scientific community recently in the context of HPC work scheduling. For example, there is a series of workshop proceedings papers [13, 14] dedicated to co-scheduling problem of HPC applications. Overall, these publications are mostly focused on the practical aspects, such as feasibility of this approach in general.

There is also a few publications on modelling HPC applications for co-scheduling. For example, in [1–3] author focuses on solving static scheduling problem with cache partitioning. Models in these papers are based on application speedup profile (that must be known in advance) as function of cache size.

Dynamic co-scheduling strategies in the context of HPC schedulers are not covered abundantly in scientific literature. Among few publications, there is [15], where authors provide supervised machine learning approach for estimating applications slowdown based on performance counters. Nevertheless, authors used this model for solving static scheduling problem. Another machine learning approach (with reinforcement learning) was used in [9] for dynamic collocation of services and HPC workloads.

Dynamic co-scheduling problem, on the other hand, is covered vastly in the context of thread scheduling in simultaneous multithreading (SMT) CPUs. There are [7, 10–12] to name a few. The general theme of approaches in these papers is to measure threads instruction-per-second (IPC) values when they were running alone on a core and when they were running in parallel with other threads. Then the ratio of these two values was used for making scheduling decisions.

In the paper [6] authors showed that optimal co-scheduling algorithm does not produce more than 3% gain in throughput (on Spec benchmarks) when compared to a naive strategy of running all threads in parallel in FCFS order. This was shown by gathering slowdown values for all threads combinations and solving linear programming problem for finding an optimal schedule that was later compared to a naive strategy.

In this paper we have done the similar work but in the context of HPC applications. Additionally, we have also provided theoretical boundaries for slowdown values when naive parallel strategy can not be applied and showed the form of deviation from the optimal strategy. These results were evaluated on numerical experiments.

### 3 Benchmarks

In the experiments below we have used benchmarks from NAS Parallel Benchmark (NPB) [4], Parsec [5], and a few of our own benchmarks. This set of benchmarks cover different examples of HPC applications as it includes CPU-, memory- and filesystem-intensive benchmarks, different parallelism granularities, data exchange patterns and workflow models. Datasets for all benchmarks were tuned to have approximately the same processing times.

Among these benchmarks we have selected only those that have constant or periodic processing speed profiles. This requirement comes from the model assumption that we will introduce later. The resulting list of benchmarks is shown in Table 1

**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
ua	NPB	Unstructured Adaptive mesh
sp	NPB	Scalar Penta-diagonal solver
freqmine	Parsec	Frequent Pattern Growth algorithm
swaptions	Parsec	HJM algorithm for pricing swap options
vips	Parsec	Image processing pipeline
streamcluster	Parsec	Online clustering problem in data mining
ffmpeg		Decoding of video file
raytracing		Ray tracing algorithm on CPU

We've run our benchmarks on a single Intel Xeon E5-2630 processor with 10 cores and 2 threads per core. Each benchmark was limited to any of 4 threads, and threads were assigned by Linux scheduler (they were not pinned manually). In some experiments we used only a half of all available cores by pinning threads to same 5 CPU cores (but within those cores threads were assigned by the Linux scheduler as well). In all of the experiments, each application had enough memory and swap was never used.

## 4 Application Processing Speed Metric

Degradation of application performance due to co-scheduling can be explained by concurrent use of shared resources, such as last level cache, memory bus, cpu time, network card etc. During application execution instructions that access shared resources may take more cycles to complete when underlying resources are performing operations for other applications. For example, instructions that require memory access may take more cycles, if required addresses are not in the cache as CPU would access memory bus. In turns, if memory bus is busy, that instruction would take ever more cycles to complete.

To define metric of application performance, we used amount of work per unit of time. We used CPU instructions as a unit of work as its rate is affected by all of the resources simultaneously, unlike resource-specific metrics (e.g. bus access rate or transmitted bytes to network card). As for the time unit, we have to take into account that CPU frequency is not constant and that application may be preempted from CPU core and suspended by OS scheduler.

CPU performance counters allow to measure cumulative values of the executed instructions ( $inst(t)$ ) and completed cycles ( $cycl(t)$ ) when application was running in the user space (as opposed to system space). Also, OS scheduler provides values for amount of time the processes was using CPU core ( $cpu(t)$ ). Dividing instructions by cycles during time period  $\Delta t$  would give processing speed during application active time ( $cpu(t) - cpu(t - \Delta t)$ ), commonly denoted in the literature as IPC (instructions per second). To be able to scale this value to the whole period we would assume that CPU cycles rate did not change when the application was not running. This gives us the formula for estimating application performance:

$$\nu(t) = \frac{inst(t) - inst(t - \Delta t)}{cycl(t) - cycl(t - \Delta t)} \frac{cpu(t) - cpu(t - \Delta t)}{\Delta t}$$

For our purposes we do not need absolute values of  $\nu(t)$ , but rather we need its change due to scheduling decisions that we make (between  $t_1$  and  $t_2$  time points):

$$f(t_1, t_2) = \frac{\nu(t_2)}{\nu(t_1)}$$

Assuming  $\Delta t$  is the same for all measurements, and that the number threads in the application do not change between measurements, this value can be computed as a ratio of product of IPC and cpu-time before and after scheduling decision. We will call this ratio as performance speedup value. When this value is less than 1, then it measures performance slowdown.

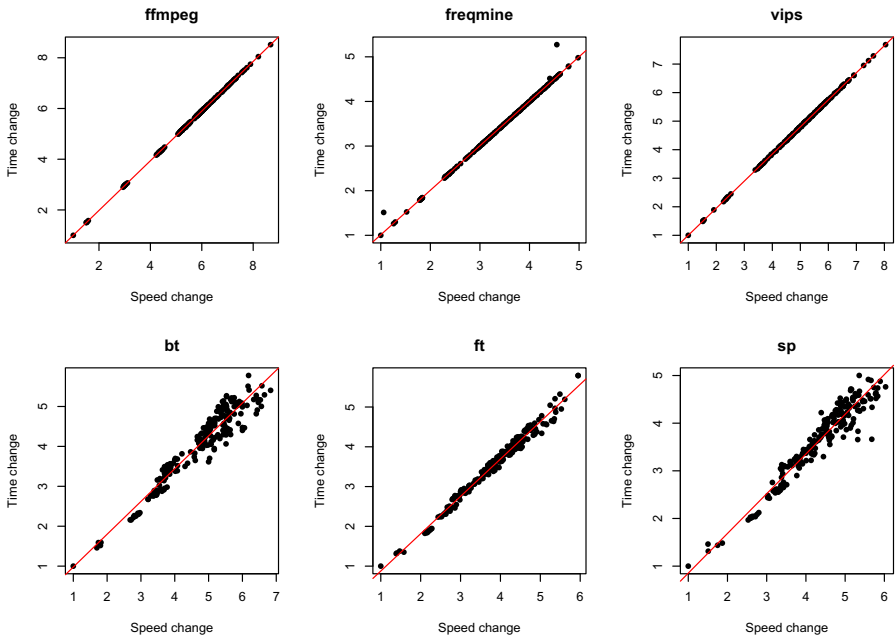
### 4.1 Evaluation on Experimental Data

To evaluate the defined metric we have compared it with change of application processing time in ideal and co-scheduling conditions. To do that we measured

application processing time when it was running alone in the server and when it was running simultaneously with different combinations of other applications. In the second case, we made sure that the application that is being measured was the first one to finish, otherwise its conditions would change before completion and comparison would not be fair. For the same reason we had to select benchmark applications with constant or periodic (with a small period) speed profiles.

To collect the data we run all possible combinations with different number of applications in parallel. Each application required 4 threads, we run up to 5 applications on 10 CPU cores, so each CPU core was oversubscribed with up to 2 applications. Application threads were not pinned to the core and the OS scheduler could migrate them between cores dynamically.

Results for some benchmarks are shown in the scatter plots in the Fig. 1 and the Table 2 contains linear regression model for all benchmarks. Good fit of the linear model with coefficients close to 1, shows that changes in processing speed (measured as described above) matches exactly with changes in total processing time.



**Fig. 1.** Total time change vs processing speed change for different combinations of parallel tasks relative to ideal conditions

**Table 2.** Linear regression model parameters for processing time change as function of processing speed change (relative to ideal conditions)

	Coefficient	Intercept	R squared
ffmpeg	0.9791	0.0234	0.9999
freqmine	0.9976	0.0141	0.9947
vips	0.9491	0.0514	0.9999
bt	0.8217	0.1551	0.9199
ft	0.9387	-0.0627	0.9844
sp	0.8335	0.0187	0.9232
raytracing	1.0240	-0.0994	0.9999
streamcluster	0.6914	1.1861	0.7651
ua	1.7214	0.2477	0.6991
swaptions	1.0700	-0.1288	0.9928

## 5 Scheduling Problem

To formalize a scheduling problem, we introduce the assumption about application behaviour that the speed of processing is constant during application execution in ideal conditions. This implies that application do not have distinguished processing stages and its processing speed depends only on other applications running in parallel. In future work we plan to consider more general case without this assumption.

Let’s introduce the following notation for problem definitions. There are  $n$  applications (or tasks):  $T = \{T_1, \dots, T_n\}$ . Each task requires  $p_i, i = 1, \dots, n$  amount of work to be completed. Tasks can be executed simultaneously in any combination. Denote each combination as  $S_j \in 2^T, j = 0, \dots, (2^n - 1)$  – a subset of task indices.  $|S_j|$  is number of tasks in  $S_j$  combination, and  $|S_0| = |\emptyset| = 0$ .

Denote processing speed (amount of work done per unit of time) of  $T_i$  task when it is executed in combination  $S_j$  as  $f_{i,j} = f_i(S_j) \geq 0$ . For all tasks not belonging to  $S_j$  processing speed is zero ( $f_{i,j} = 0 \forall T_i \notin S_j$ ). We will define processing speed of a combination of tasks as the sum of all tasks’ processing speed in the combination ( $\sum_{i=1}^n f_{i,j}$ ).

A sequence of assigned processing times to combinations of tasks  $x_{j_1}, \dots, x_{j_m}, 0 \leq j_k \leq (2^n - 1)$  will be called a schedule when each tasks completes its required amount of work, i.e.  $\sum_{k=1}^m f_{i,j_k} x_{j_k} = p_i \forall i = 1, \dots, n$ . Tasks in a schedule can be preempted at any time, i.e. a tasks combination may repeat in a schedule.

The scheduling problem is to find a schedule that has a minimal makespan value ( $C_{max}$ ). Makespan is a completion time of the last task in a schedule.

### 5.1 Optimal Strategy

Makespan of a schedule can be written as a sum of assigned processing times to each tasks combination:  $C_{max} = \sum_{k=1}^m x_{j_k}$ . Since in the we can reorder terms in makespan sum and work amount sum without affecting their values, it allows us to consider only the schedules with non-recurring tasks combinations.

The problem then reduces to finding distribution of processing time among non-empty tasks combinations. Instead of a sequence of task combinations we will consider  $x_j \geq 0, j = 1, \dots, (2^n - 1)$  – a total processing time of combination  $S_j$ . This allows us to solve scheduling problem as linear programming problem:

$$\begin{aligned} &\text{minimize } \sum_{j=1}^{2^n-1} x_j \\ &\text{subject to } \sum_{j=1}^{2^n-1} f_{i,j} x_j = p, \quad i = 1, \dots, n \\ &\quad \quad \quad x_j \geq 0, \quad \quad \quad j = 1, \dots, 2^n - 1 \end{aligned}$$

By solving this problem, we will find values  $x_1^*, \dots, x_{2^n-1}^*$  that produce a minimum makespan. A schedule can be reconstructed from these values simply by running each task combination  $S_j$  for  $x_j^*$  time (if  $x_j^* \neq 0$ ) in any order without interruptions. We will denote an optimal makespan value as  $C_{max}^{LP}$ .

This approach produces optimal solution but it can not be used in schedulers as it requires all a priori information about each application. Instead it can be used as a reference for evaluation of other scheduling strategies.

### 5.2 Heuristic Strategies

We will consider two heuristics strategies (naive parallel and round-robin) and compare their solutions with the optimal strategy.

Naive parallel (NP) strategy disregards all information about applications and runs all available applications in parallel. This strategy does not require any a priori information about applications and thus its implementation is the simplest.

Another heuristic strategy that we consider is round robin (RR). It works by finding subsets of active tasks with the maximum speed and running each subset one after another in a loop until one of the tasks finishes. Each subset is run continuously for at most one unit of time (denoted as  $T$ ). Time unit may be smaller only when a task in a subset finishes earlier. We will call a sequence of subsets with the same speed executed in a single loop in RR strategy as a round. Round, in turn, consists of individual units of subset execution.

Unlike NP, RR strategy requires slowdown values for each subset of application. But, unlike the optimal strategy, it does not require amounts of works ( $p_i$ ) of each application.

## 6 Strategies Comparison

In this section we will compare RR and NP heuristics strategies with the optimal strategy and provide bounds on slowdown function parameters when NP performs not worse than RR strategy.

### 6.1 Additional Assumptions on the Application Behaviour

There we will introduce additional assumptions about application behaviour in order to obtain analytical solutions. The first assumption is that the processing speed decreases linearly with an increase of the number applications running in parallel. We have seen this dependency in our experiments (as shown in Fig. 2 and Table 3). This assumption is introduced as linear dependency is the simplest non-trivial form of speedup function. The results that are obtained below will also hold for convex functions as well.

Another assumption is that each application has the same slowdown function. This is equivalent to claiming that each application is the same. This is very restrictive assumption, but by choosing slowdown functions with minimum or maximum slope value, we can obtain lower or upper limits for the slope value for switching strategies.

With these two assumptions we can write speedup function as  $f_k = 1 - \alpha(k - 1)$  which only depends on  $k = 1, \dots, n$  - a number of tasks in the combination.  $\alpha$  is a slope of slowdown function such that  $0 < \alpha < \frac{1}{n-1}$  (since  $f_k$  should be  $0 < f_k \leq 1, k = 1, \dots, n$ ).

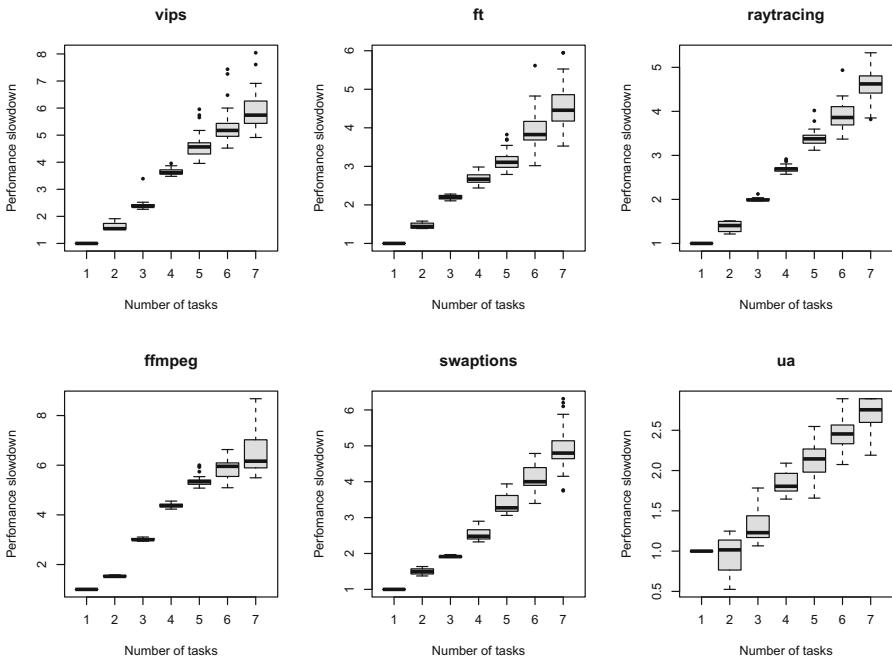


Fig. 2. Processing speed slowdown (relative to ideal conditions) as a function of the number of parallel applications



**Table 3.** Parameters of linear model of processing speed acceleration (relative to ideal conditions) vs. the number of parallel applications.

	Intercept	Coefficient	R squared
vips	0.7902	0.4483	0.8008
ft	0.6141	0.2133	0.7942
raytracing	0.6356	0.1496	0.9046
ffmpeg	0.8028	0.9821	0.7892
swaptions	0.7464	-0.3225	0.8771
ua	0.3239	0.4755	0.8215
bt	0.7315	0.5002	0.7941
freqmine	0.7315	0.5002	0.7941
sp	0.7315	0.5002	0.7941
streamcluster	0.7315	0.5002	0.7941

### 6.2 Comparison of Round-Robin and the Optimal Strategy

Since there’s no analytical solution for linear programming problem, we ran numerical experiments to compare RR strategy with optimal strategy. We performed parameter sweeps on the number of tasks ( $n$ ) and slowdown slope values ( $alpha$ ). For each set of number of tasks and slope values we have generated 100 cases of differential amounts of works ( $q_i = p_{i+1} - p_i$ ) that were drawn from uniform distribution (from 0 to 80 units bounds). For each case we solved linear programming problem (using lpsolve library) to find an optimal solution and found solution with RR strategy (with time unit  $T = 2$ ).

Figure 3 shows boxes and whiskers plots with competitive ratios of RR strategy for each slope value ( $alpha$ ) containing results for 100 sets of random  $q_i$  values. Competitive ratio is computed as ratio of RR makespan value to optimal makespan value. It can be seen that RR produces less than 15% deviation from the optimal strategy for 10 tasks.

### 6.3 Comparison of Round-Robin and Naive Parallel Strategy

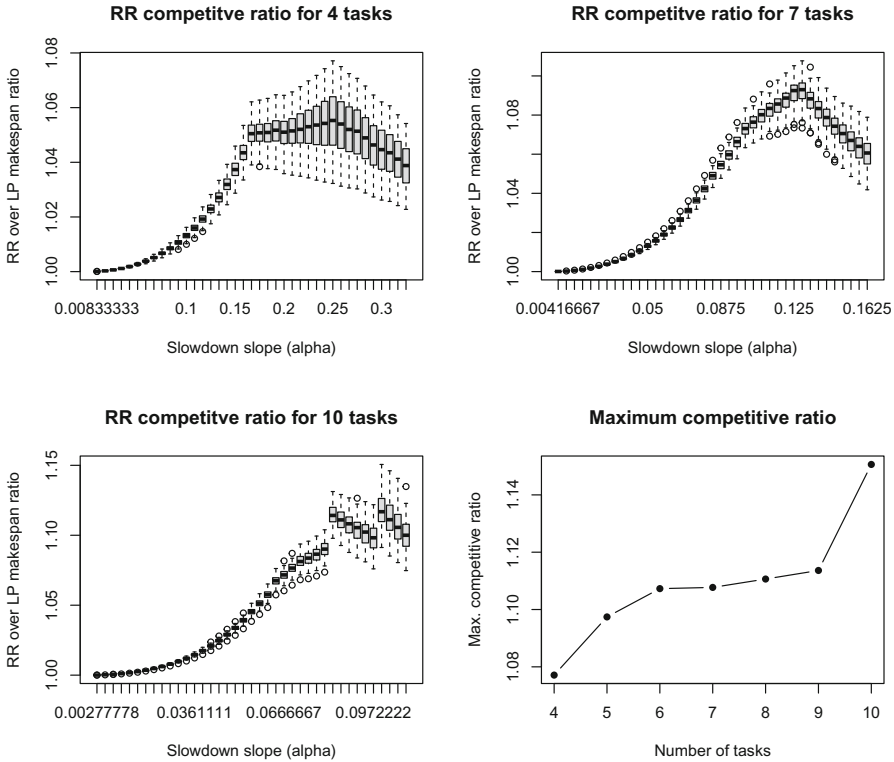
Since RR makespan value is close to the optimal and we can derive analytical formula for it, we will use it as a reference for evaluation of NP strategy.

Lets consider that applications are sorted in increasing order of  $p_i$ , i.e.  $p_i \leq p_{i+1}, i = 1, \dots, n - 1$ . Denote  $q_i$  as:

$$q_1 = p_1$$

$$q_k = p_k - p_{k-1}, k = 2, \dots, n$$

NP strategy runs all tasks in parallel, as they finish in the increasing order of  $p_i$ , the first task, will finish after  $q_1$  of work is completed, the second one after  $q_2$  of work is completed and so on. The processing speed would also increase as tasks complete from  $f_n$  to  $f_{n-1}$  and so on until  $f_1$ . Using this, we can write makespan value for this strategy:



**Fig. 3.** Ratio of RR makespan to LP makespan as a function of slowdown slope value (alpha) for 4, 7 and 10 tasks. Bottom right plot shows max ratio value (across all alphas) for different number of tasks

$$C_{max}^{NP} = \sum_{k=1}^n \frac{q_k}{f_{n-k+1}}$$

Now, let’s derive an estimate for makespan of RR strategy. We will do that by finding the sequence of tasks completion and number of rounds required to complete each task. This will allow us to derive exact makespan value, but it would contain rounding and modulo operations. To get rid of them, we will provide an upper bound instead. The deviation of the upper bound from the exact value would depend only on the time unit parameter ( $T$ ), so after limit transition ( $T \rightarrow 0$ ) we will get a close approximation for RR makespan value.

Denote  $s$  as the size of tasks combination with the maximum speed, i.e.  $s = \arg \max_{1 \leq k \leq n} \{k f_k\}$ . RR strategy at first will choose combinations with  $s$  tasks and will run each combination consequently for a single time unit ( $T$ ). These combinations will be executed until the task with smallest amount of work ( $q_1 = p_1$ ) will finish.

There are  $\binom{n}{s}$  combinations with  $s$  tasks, so until the first task finishes, each round would contain the same amount of time units. In a single round, each task would run  $\binom{n-1}{s-1}$  times. Since the first task requires  $q_1$  amount of work and  $Tf_s$  of work is completed per unit (processing speed of  $s$  tasks is  $f_s$ ). This gives us an upper bound of processing time until the first task finishes as:

$$g_1 \leq \left\lceil \frac{q_1}{\binom{n-1}{s-1}Tf_s} \right\rceil \binom{n}{s}T$$

After the first task finishes, we would be left with  $s - 1$  tasks and the next smallest task would require  $q_2 = p_2 - p_1$  amount of work before completion. If  $s$  was less than  $n$ , remaining tasks would still run in combinations of  $s$  tasks, or if  $s = n$ , then remaining tasks would run in combinations of  $s - 1$ .

Let's consider the first case ( $s < n$ ) when RR still chooses combinations with  $s$  tasks. Since one task is finished, there are now  $\binom{n-1}{s}$  available combinations and each task would run  $\binom{n-2}{s-1}$  times per round. This gives us the following upper bound for the second task:

$$g_2 \leq \left\lceil \frac{q_2}{\binom{n-2}{s-1}Tf_s} \right\rceil \binom{n-1}{s}T$$

The second case ( $s = n$ ) or case when there are no combinations left with  $s$  tasks are similar, because in both cases RR chooses combinations with less than  $s$  tasks. Since RR chooses combinations with the fastest speed to run next, it would always run the same (fastest) combination until the next task finishes. There's only such combination as there is only one way of choosing  $s - 1$  tasks from the subset of  $s - 1$  tasks. If this case occurs for the third task, the upper bound would be:

$$g_3 \leq \left\lceil \frac{q_3}{Tf_{n-s}} \right\rceil T$$

We can now write general formulae for all tasks:

$$g_k \leq \left\lceil \frac{q_k}{\binom{n-k}{s-1}Tf_s} \right\rceil \binom{n-k+1}{s}T, \quad k = 1, \dots, (s - 2)$$

$$g_k \leq \left\lceil \frac{q_k}{Tf_{n-k+1}} \right\rceil T, \quad k = (s - 1), \dots, n$$

Using the fact that  $\lceil \frac{a}{b} \rceil b < a + b$  for  $a, b > 0$  and that  $\binom{n-k+1}{s} = \frac{n-k+1}{s} \binom{n-k}{s-1}$  we can simplify the bounds as

$$g_k < \frac{n-k+1}{s} \left( T \binom{n-k}{s-1} + \frac{q_k}{f_s} \right), \quad k = 1, \dots, (s - 2)$$

$$g_k < T + \frac{q_k}{f_{n-k+1}}, \quad k = (s - 1), \dots, n$$

Which gives us the upper bound of makespan value:

$$C_{max}^{RR} < \sum_{k=1}^{n-s+1} \frac{n-k+1}{s} \left( T \binom{n-k}{s-1} + \frac{q_k}{f_s} \right) + \sum_{k=n-s+2}^n \left( T + \frac{q_k}{f_{n-k+1}} \right)$$

As the difference between each  $g_k$  and its upper bound is  $O(T)$  by using very small time unit, we will get an approximation ( $C_{max}^{RR*}$ ) of an exact value. After limit transition  $T \rightarrow 0$  we'll get:

$$C_{max}^{RR} \approx C_{max}^{RR*} = \sum_{k=1}^{n-s+1} \frac{n-k+1}{s} \left( \frac{q_k}{f_s} \right) + \sum_{k=n-s+2}^n \frac{q_k}{f_{n-k+1}}$$

We can notice that the last sum in  $C_{max}^{RR*}$  upper bound matches exactly with the one in  $C_{max}^{NP}$  for  $k > n - s + 1$  and when  $s = n$  these two value are the same. Because of that, we can claim that when the largest subset is the fastest, then round-robin strategy performs the same as naive parallel strategy.

Assuming that slowdown function is linear ( $f_k = 1 - \alpha(k - 1)$ ,  $k = 1, \dots, n$ ) we can find the slope threshold ( $\alpha^*$ ) after which round-robin produces smaller makespan:

$$n = s = \arg \max_{0 \leq k \leq n} \{k f_k\} = \left\lceil \frac{\alpha^* + 1}{2\alpha^*} \right\rceil = \left\lfloor \frac{\alpha^* + 1}{2\alpha^*} + \frac{1}{2} \right\rfloor$$

Which gives the following bounds to  $\alpha^*$ :

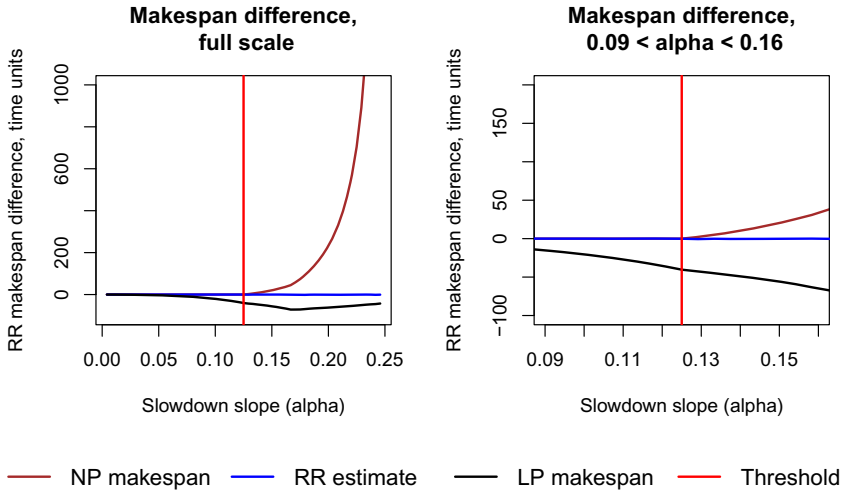
$$\frac{1}{2n} < \alpha^* \leq \frac{1}{2(n-1)}$$

So, when  $f_k$  slope is greater than  $\alpha^* = \frac{1}{2(n-1)}$  the difference between  $C_{max}^{NP}$  and  $C_{max}^{RR*}$  is non-zero. We can estimate the first term in makespan difference:

$$C_{max}^{NP} - C_{max}^{RR*} = \frac{q_1}{f_n} - \frac{nq_1}{sf_s} = \dots = -q_1 \frac{(2n-1)^2\alpha^2 + (2-4n)\alpha + 1}{(n-1)\alpha^3 + (2n-3)\alpha^2 + (n-3)\alpha - 1}$$

Other non-zero terms will be similar, with the only difference in the first coefficient ( $q_k$ ). Each term is a ratio of two polynomials with  $\alpha$  variable and the polynomial in denominator has a larger degree and the one in numerator. Because of that, with increase of  $\alpha$  value the difference in makespan values has hyperbolic growth.

The results obtained here can be seen on numerical to simulations. In the Fig. 4 there are plots corresponding to  $C_{max}^{NP} - C_{max}^{RR}$  (brown line),  $C_{max}^{RR*} - C_{max}^{RR}$  (blue line) and  $C_{max}^{LP} - C_{max}^{RR}$  (black line). Red line show threshold value  $\alpha^*$ . As it can be seen, blue line is almost zero for all slopes values, which shows that approximate formulae match with exact value. NP makespan value also matches with RR strategy exactly until for  $\alpha < \alpha^*$  and after threshold value (red line) it starts to increase significantly. Deviation of RR from the optimal strategy (black line) is negligible, when compared to NP strategy.



**Fig. 4.** Difference of makespan values of NP, LP, RR estimation with RR makespan for different slowdown slopes value ( $\alpha$ ). Vertical line shows threshold valued computed from derived formula. Both plots show the same data, but in different scales

## 7 Conclusion

In this paper we have defined application behavior model in co-scheduling environment, when applications can be executed simultaneously on shared resources. We have proposed to use application processing speed (measured based on IPC and cpu time) as a metric of performance degradation and have validated it on HPC benchmarks applications.

Based on that we have formalized a scheduling problem and found an optimal solution by reducing it to a linear programming problem. Optimal solution can not be implemented in schedulers as it requires a priori information about application performance slowdown values for all possible combinations of parallel applications. Besides that, linear programming problem does not have an analytical solution, it can only be solved iteratively.

To obtain more practical solution, we have considered two heuristic strategies, round-robin (RR) and naive parallel (NP). The first one (RR), takes into account application slowdown values and iterates over combinations of applications with the lowest slowdown in RR fashion. The second one (NP) simply runs all available applications in parallel disregarding slowdown completely. We have showed using numerical experiments that RR produces results very close to the optimal strategy and NP strategies matches with RR until some point.

We have derived an analytical bound for applications performance degradation value until which NP strategy matches exactly with RR strategy, and thus it is very close to the optimal strategy. When this threshold values is reached, the difference between an NP and RR starts to increase significantly (with hyperbolic growth).

The threshold value has a very simple analytical formula and  $t_i$  depends only on the number of jobs (given model assumptions), so can be computed easily in practice. Possible scheduler implementation may be based on online version of RR strategy (that was described in the paper) solving multi-armed bandit problem [9, 12]. This strategy would periodically probe multiple applications combinations to evaluate their processing speed, then it would pick one combination and would run it for some amount of time. Results from this paper allow to probe only the largest combination one time and then to run it immediately, if its speed is below a threshold value. In this case, this decision would be a part of an optimal schedule.

**Acknowledgements.** Research has been supported by the RFBR grant No. 19-37-90138.

## References

1. Aupy, G., et al.: Co-scheduling Amdahl applications on cache-partitioned systems. *Int. J. High Perform. Comput. Appl.* **32**(1), 123–138 (2018)
2. Aupy, G., Benoit, A., Goglin, B., Pottier, L., Robert, Y.: Co-scheduling HPC workloads on cache-partitioned CMP platforms. *Int. J. High Perform. Comput. Appl.* **33**(6), 1221–1239 (2019)
3. Aupy, G., Benoit, A., Pottier, L., Raghavan, P., Robert, Y., Shantharam, M.: Co-scheduling high-performance computing applications. In: *Big Data: Management, Architecture, and Processing*, May 2017. <https://hal.inria.fr/hal-02082818>
4. 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)
5. Bienia, C.: *Benchmarking Modern Multiprocessors*. Ph.D. thesis, Princeton University, January 2011
6. Eyerman, S., Michaud, P., Rogiest, W.: Revisiting symbiotic job scheduling. In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 124–134. IEEE (2015)
7. Jain, R., Hughes, C.J., Adve, S.V.: Soft real-time scheduling on simultaneous multithreaded processors. In: *23rd IEEE Real-Time Systems Symposium, RTSS 2002*, pp. 134–145. IEEE (2002)
8. Kuchumov, R., Korkhov, V.: Collecting HPC applications processing characteristics to facilitate co-scheduling. In: Gervasi, O., et al. (eds.) *ICCSA 2020*. LNCS, vol. 12254, pp. 168–182. Springer, Cham (2020). [https://doi.org/10.1007/978-3-030-58817-5\\_14](https://doi.org/10.1007/978-3-030-58817-5_14)
9. Li, Y., Sun, D., Lee, B.C.: Dynamic colocation policies with reinforcement learning. *ACM Trans. Architect. Code Optim. (TACO)* **17**(1), 1–25 (2020)
10. Parekh, S., Eggers, S., Levy, H., Lo, J.: *Thread-sensitive scheduling for SMT processors* (2000)
11. Snively, A., Mitchell, N., Carter, L., Ferrante, J., Tullsen, D.: Explorations in symbiosis on two multithreaded architectures. In: *Workshop on Multi-Threaded Execution, Architecture, and Compilers* (1999)

12. Snaveley, A., Tullsen, D.M.: Symbiotic jobscheduling for a simultaneous multi-threaded processor. In: Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 234–244 (2000)
13. Trinitis, C., Weidendorfer, J.: Co-scheduling of HPC Applications, vol. 28. IOS Press (2017)
14. Trinitis, C., Weidendorfer, J.: First workshop on co-scheduling of HPC Applications (COSH 2016)
15. Zacarias, F.V., Petrucci, V., Nishtala, R., Carpenter, P., Mossé, D.: Intelligent colocation of workloads for enhanced server efficiency. In: 2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), pp. 120–127. IEEE (2019)