ML4AAD - Report Parallel Algorithm Configuration

Saskia, Francine, Hussein and Mohammad Ali
Albert Ludwigs University of Freiburg

Abstract

The report describes the implementation and the results of an Algorithm Configuration tool created during the laboratory course "Machine Learning for Automated Algorithm Design". The tool uses **SMAC** as a code base and uses parallelism and Bayesian optimization techniques in order to enhance the performance and the runtime.

Motivation

When trying to solve the algorithm configuration problem, it is likely to come across an infinitely large configuration space which would lead to an arbitrarily long parameter tuning phase when exploring all possible options for a parameter setting. Methods like SMAC Hutter, Hoos, and Brown (2011) have already came up with a concept of splitting the searching progress into the two main parts of finding promising configurations and then racing them against each other, to find the best possible parameter configuration in a given set of time without having to explore most of the configuration space. The project's goal is to use this idea and to add parallelization in order to expand the search and maybe find even better results in the same amount of time.

Approach

Similar to SMAC, the tool first searches the configuration space for promising configurations (the so called challengers) and then it would evaluate those challengers afterwards.
Three different approaches were used for finding such a set which can be chosen by an input parameter. Once a set of challengers is determined, they are evaluated on a set of instances, we call this racing them against each other. The one configuration that is performing best in the racing process, is the winner and is directly compared against the current best solution, our incumbent.
Two different methods are available for the racing part, which are also linked to an input parameter. After defining the current incumbent, a new set of challengers is searched and the process starts again until the predefined time limit is reached.

How to select challengers

Constant Liar

Constant Liar approach was introduced by Ginsbourger, Riche, and Carraro (2008). It was introduced as a heuristic to optimize models during iterations by adding a hallucinated value after each iteration and refitting the model with those added values.
There are three flavors to the Constant Liar approach. They are : **CL-min**, **CL-max** and **CL-pred** and they represent the minimal performance, maximal performance and the mean prediction so far, respectively. The exploration factor in the algorithm increases as the lie value increases and consequentially the exploitation factor increases as the lie value decreases.
The variation that was used in this tool was **CL-min** because of its exploitation factor. The implementation of the approach in the tool was as follows: The runs for each configuration were separated, then minimum cost the runs of each configuration is calculated. A "fake" Runhistory is created which is a copy of the existing current Runhistory. All the hallucinated values are added to the fake Runhistory along with the configuration they represent. The tool model (which is a Random Forest) is refitted using the fake Rushistory and the acquistion function is updated using the model. The new challengers are selected using the updated acquistion function.

Expectations across Fantasies

The Expectations across Fantasies approach is first introduced by Snoek, Larochelle, and Adams (2012). The approach is to calculate expected improvement over several fantasized values.
The approach is applied in the tool as follows: The runs of each competing configuration is separated. The posterior of those runs is computed using Linear Regression. A new EPM model is constructed using the newly computed posterior. A new acquistion function is computed using that EPM model. The acquistion function values is aggregated over all configuration. A new model is fitted with the aggregate acquistion values and an acquistion function is created using the aggregate model. New challengers are computed from acquistion optimizer calculated using the final acquisition function.

UCB

Parallel UCB or Parallel "Upper confidence bound" approach was introduced by Jones (2001). The goal of the approach is to satisfy the following formula:

$$arg_{\theta \in \Theta} min \quad \mu_\theta + \beta \cdot \sigma_\theta$$

μ_θ is the mean prediction of the configuration runs. σ_θ is the predictive uncertainty of the configuration runs. β is constant used to trade off exploitation and exploration.
The application of this approach in the tool is as follows: The runs of each configuration is separated. β is sampled from a uniform distribution between 0 and 1 randomly to vary the exploration. The UCB formula is computed for each configuration with the sampled β. The function returns the configuration with the minimum UCB score.

Parallel Racing

The main idea behind the parallel racing is to apply these different racing methods to determine the winner of challengers.
Therefore, the winner of a race is always compared to the incumbent, which is the current best configuration, to determine a new incumbent or keep the old one if it was better than the winner.
In order to do so, the emprical cost (emp_{cost}) is measured as well as the number of runs **N** of the incumbent and the winner. To identify which configuration is best, the one with smaller empirical cost is selected, which is defined as $\frac{emp_{cost}}{N}$. The best of these two is the new incumbent.
Note that one configuration may have smaller empirical cost but is tested on fewer instances, so small cost may not be our main target to select a good incumbent but also the number of runs.
It was decided to implement two different methods of racing:

Racing over a List of Instances

The main idea behind this approach is to select a list of instances instead of taking all of them into account. Each challenger is ran on that list of instances. The winner is the one that terminates first on that list. The incumbent and the winner of the racing are compared as described before.

Racing over Single Instance

For this method a set of instances is selected. This set is as large as the number of challengers we have. For each instance, all challengers are ran on them. The challenger that has the largest cost value on the instance is dropped. We stop when only one challenger is left, which is the winner of the racing. As before, the incumbent is compared with the winner of the racing by comparing the average cost as described before.

Experiments

Benchmark SATenstein

The benchmark is ran with the given scenario from **ACLIB** with a cutoff of 5 seconds and a wallclock limit of one hour.

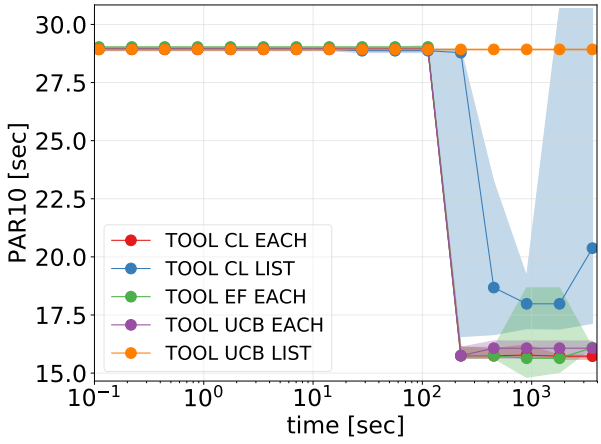

Figure 1: one core Satenstein

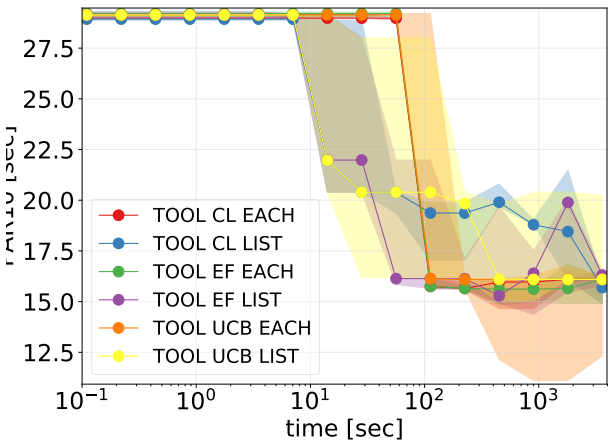

Figure 2: two cores Satenstein

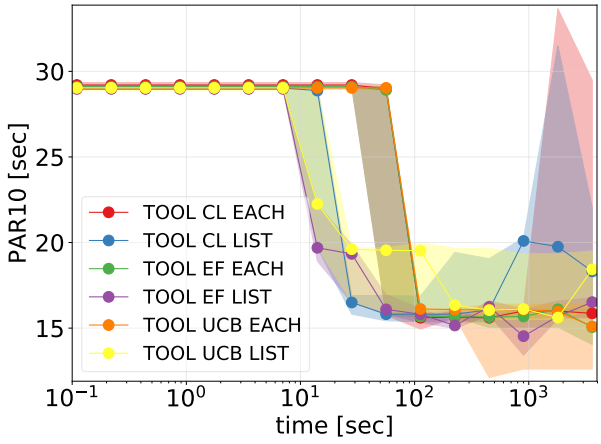

Figure 3: four cores Satenstein

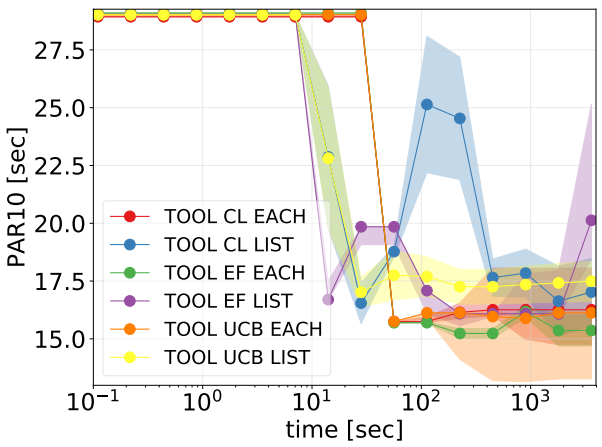

Figure 4: eight cores Satenstein

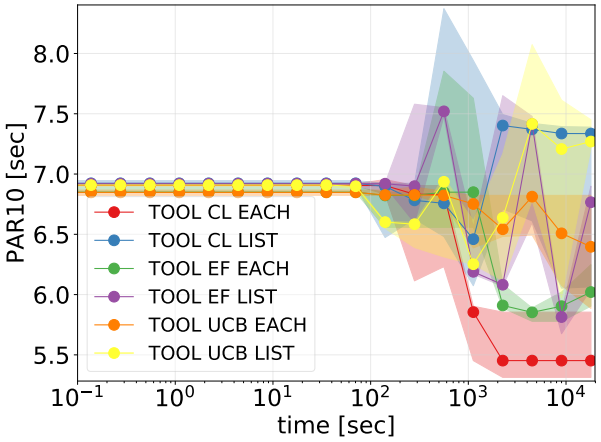

Figure 6: two cores Spear

Benchmark Spear

The benchmark is ran with the given scenario from **ACLIB** with a cutoff of 5 seconds and a wallclock limit of 5 hours.

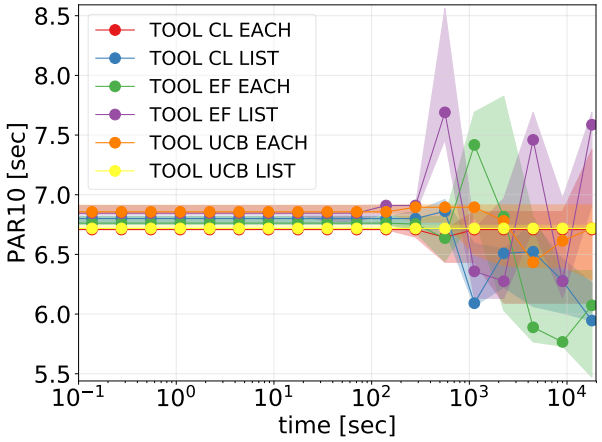

Figure 5: one core Spear

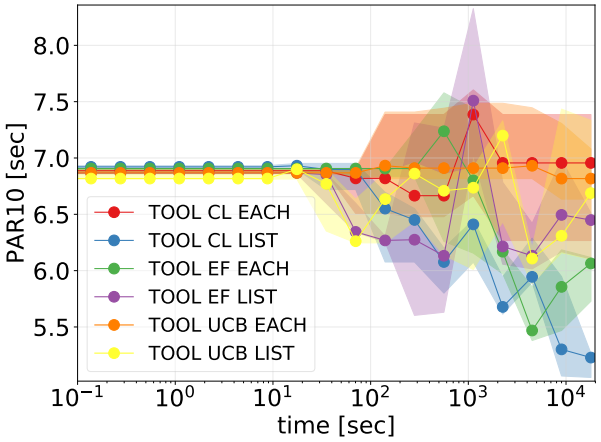

Figure 7: four cores Spear

Conclusion

Based on the previous section observations, multiple conclusions can be drawn. The most obvious one is the using **UCB** while racing on list of instances on just one core does not challenge the incumbent. That is understandable because racing over a list of instances require the number of challengers to be equal to the number of cores which is just one core in that case. So that lone challenger just races itself and then the same challenger is picked using the **UCB** function since it is the only available configuration and so on. It is clear that using **UCB** while racing over list of instances using just one core may not be the best course of action.

As for the rest of the results, it was not surprising to see that as the number of cores increases, the amount of time required to optimize decreases. That does not necessarily

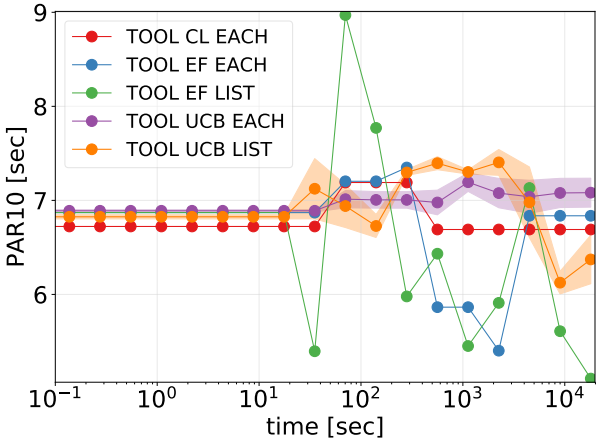

Figure 8: eight cores Spear

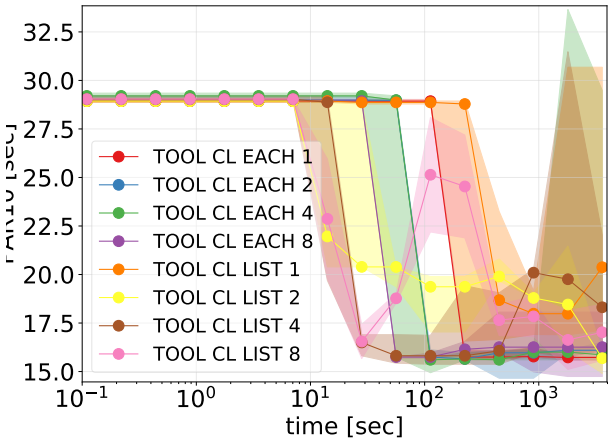

Figure 9: compare Performance from CL on Satenstein with different number of cores

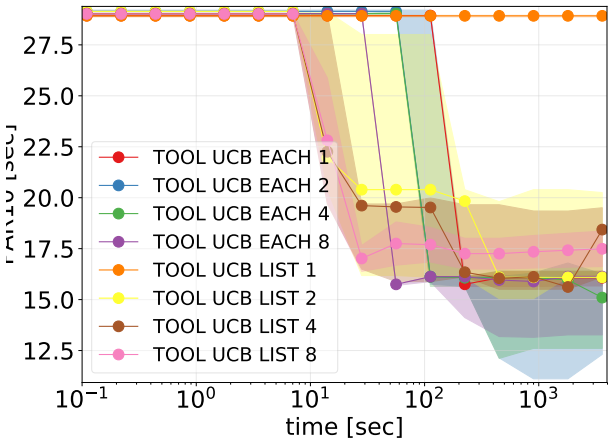

Figure 10: compare Performance from UCB on Satenstein with different number of cores

mean that the final solution is better using high number of cores because it can just stagnate after reaching good configuration early on. However, that can be useful in case of some problems where it would be sufficient to find the configuration below certain performance threshold.

The variation of the results between the racing strategies was not crystal clear though. In case of just one core, racing over single instance was better than racing over a list of instances. As the number of cores grew, the results were mixed. Although the general trend showed that SATenstein is doing better while racing over single instance while Spear is doing better using List of instances.

The variation between the Bayesian optimization techniques was not too obvious as well. It seems that the racing strategies had more impact on the performance than the Bayesian optimization techniques did. Using For SATenstein, using **UCB** or **Expectations across fantasies** looked to achieve better results but **Constant Liar** had better results using Spear.

The overall best performance for SATenstein happened while using **UCB** optimization technique while racing over single instance and using 4 cores. **Expectations across fantasies** with the same conditions had a close performance. As for Spear benchmark, The best performance was using **Constant Liar** over list of instance over 4 cores. Using **Expectations across fantasies** over list of instance but with 8 cores achieved almost similar performance.

These conclusions do not offer a universal combination of techniques to solve the problem at hand, but it reaffirms the logical assumption that using parallelism would enhance the time need to improve over the incumbent. As for the Bayesian optimization techniques and racing strategies, the results would depend on the Algorithm Configuration problem itself. So the magical universal combination does not exist according to the tool, however for a certain benchmark, certain combinations can be recommended.

References

Ginsbourger, David, Riche, Rodolphe Le, and Carraro, Laurent. 2008. A Multi-points Criterion for Deterministic Parallel Global Optimization based on Gaussian Processes. In *HAL preprint hal-00260579*.

Hutter, Frank, Hoos, Holger H., and Brown, Kevin Leyton. 2011. Sequential Model-Based Optimization for General Algorithm Configuration. In *Learning and Intelligent Optimization 5*.

Jones, Donald R. 2001. A taxonomy of global optimization methods based on response surfaces. In *Journal of Global Optimization*.

Snoek, Jasper, Larochelle, Hugo, and Adams, Ryan P. 2012. Practical Bayesian optimization of Machine Learning Algorithms. In *NIPS'12 Proceedings of the 25th International Conference on Neural Information Processing Systems*.