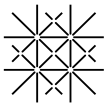

University of Basel

Variance and Resource Usage of Computational Experiments

Bachelor's Thesis

Department of Mathematics and Computer Science

Artificial Intelligence Group

University of Basel

Examiner: Prof. Dr. Malte Helmert

Supervisor: Clemens Büchner, PhD

Nillan Sivarasa

nillan.sivarasa@stud.unibas.ch

23-053-457

July 17, 2026

Abstract

This thesis examines how runtime variation and allocated resource usage can be measured for repeated Fast Downward experiments on a shared Slurm based High-Performance Computing (HPC) system. The final setup runs planner configurations in Apptainer containers through Slurm job arrays. For each experiment, it stores the generated task list, the Slurm script, the raw planner output and the parsed result files.

The analysis follows four questions. First, repeated runs of the same task and planner configuration are compared using mean runtime, standard deviation and coefficient of variation (CV). Second, Uniform Cost Search (UCS) and A* LM-cut are each compared between isolated and combined experiment groups on a common set of successfully solved tasks. Third, the thesis studies two implementation choices, memory binding and CPU allocation. The CPU allocation part is treated as an accounting experiment about requesting more CPUs than needed, not as a speedup experiment. Fourth, it identifies the experiment records that make the runs easier to inspect and repeat later.

Because direct energy measurements were unavailable, allocated CPU time is used to compare reserved computational resources and to derive a rough estimate of energy consumption.

The results suggest that runtime variation should be analyzed per task and configuration, rather than through one global runtime distribution. Very short successful tasks show the largest relative variation. Longer solved tasks are usually more stable. Running UCS and A* LM-cut in a combined experiment does not produce a clear systematic shift in runtime or runtime variation compared with the isolated experiments. Memory binding shows no clear systematic stability improvement in the tested setup. Assigning many CPUs to single process planner runs does not clearly reduce median runtime, but it increases allocated CPU time substantially. The thesis therefore recommends per task statistics, explicit failure filtering, separate reporting of allocated CPU time and complete experiment records.

Use of Generative Digital Tools

Generative digital tools were used as supporting tools during the preparation of this thesis. DeepL was used for translation support and language improvement. ChatGPT was used for research orientation, second opinions, feedback on wording and suggestions for improving the structure and clarity of self written text.

No generative AI tool was used as a scientific source or to conduct the experiments. The experimental design, implementation, analysis and final interpretation were carried out by the author. All factual claims and citations were checked against the cited sources or the generated experiment data. The author takes full responsibility for the final content of this thesis.

Table of Contents

1 Introduction

1.1 Motivation

Computational experiments are used to compare algorithms, implementations and parameter choices. However, reliable measurement is not only a matter of choosing suitable benchmarks and metrics. On shared High-Performance Computing (HPC) systems, the execution environment can affect runtime measurements through scheduling decisions, node placement and interference from other jobs. In classical planning, a benchmark task describes states, actions and goals. A planner then tries to find a plan. The usual measurements in planning (Ghallab et al., 2004) include coverage, runtime and solution quality. This thesis uses Fast Downward (Helmert, 2006) as the planning system.

Runtime numbers are not produced by the planner alone. They also depend on the machine on which the planner runs. Shared HPC systems add another layer, because jobs are scheduled on different nodes and may run while other users are active. Beyer et al. (2019) discuss several requirements for reliable benchmarking, including accurate measurements, resource limits, CPU core assignment, Non-Uniform Memory Access (NUMA) aware placement and isolation between runs. These requirements motivate the setup used in this thesis. The practical question is simple. If the same planner command is repeated on the same task, how stable is the measured runtime?

1.2 Problem Statement

This thesis treats a planning experiment as a concrete execution problem. The input is a planner configuration, a benchmark task, an execution setup and a set of resource limits. The output contains runtime, coverage, failure information and allocated CPU time. The aim is to measure these outputs in a way that is clear enough to interpret and repeat on a Slurm based cluster.

The final pipeline uses containers for the software environment and Slurm for job submission and resource allocation. It does not try to find the best planner. The selected configurations are used because they lead to different runtime behavior. This makes them useful test cases for studying variation, setup effects and resource usage.

1.3 Research Questions

The main question is how runtime and resource variation can be measured and interpreted for repeated Fast Downward experiments on a shared Slurm based HPC system.

The thesis considers four questions:

- **Q1.** For the same planner configuration and benchmark task, how much do runtime measurements vary across repeated runs?
- **Q2.** Do runtime and runtime variation change when UCS and A* LM-cut are run in a combined experiment instead of in separate experiments?
- **Q3.** How do implementation choices such as memory binding and CPU allocation affect observed runtime measurements and allocated resource usage?
- **Q4.** Which experiment records and practical recommendations help make such planning experiments easier to repeat on a shared Slurm based HPC system?

2 Classical Planning with Fast Downward

Classical planning (Ghallab et al., 2004) asks for a sequence of actions that changes an initial state into a state satisfying a goal. A planning task specifies the initial state, the goal and the available actions. A solution is a plan. Planning benchmarks are often written in Planning Domain Definition Language (PDDL) (McDermott et al., 1998), where a domain description is separated from concrete problem instances.

Fast Downward (Helmert, 2006) is a domain independent planning system. It translates PDDL input into a finite domain representation and solves the translated task with heuristic search. Heuristic search is also a useful setting for this thesis because small changes in search effort can lead to different runtime and resource allocation behavior on shared hardware. In this thesis, each run fixes one Fast Downward configuration and one benchmark task. The configurations are not ranked as planner submissions. They are used because they create different runtime profiles. The following configurations therefore represent different ways of solving the same type of planning task rather than different experimental goals.

2.1 Planner Configurations

The selected configurations cover simple optimal search, informed optimal search, satisficing search and LP based optimal planning. They are deliberately different, because the thesis is interested in how different search behavior affects measured runtime and variation.

Uniform Cost Search (UCS) is the simplest reference configuration. Fast Downward implements it as A* search (Hart et al., 1968) with the blind heuristic. Since this heuristic gives no useful estimates for non goal states, the search expands states mainly according to path cost. This means that the algorithm has little guidance about which states are closer to the goal. Harder tasks can therefore lead to large searches, timeouts and visible runtime differences. This makes the configuration useful as a baseline where runtime is mainly driven by search effort rather than by heuristic computation.

A* LM-cut uses an admissible landmark based heuristic (Helmert and Domshlak, 2009). The heuristic works with a delete relaxed view of the planning task, where actions are treated as if they do not remove facts that were already achieved. From this relaxation, LM-cut identifies sets of actions that every relaxed solution must use in order to make progress toward the goal. Such a set is called a cut. The minimum action cost in the cut is added

to the heuristic estimate and the action costs are reduced before the next cut is computed. Repeating this process gives a lower bound on the remaining goal cost, which keeps the heuristic admissible. This makes each heuristic evaluation more expensive than in UCS, but it can reduce the number of states that need to be explored. The tradeoff is important for this thesis. The runtime for a task can decrease when the additional heuristic work avoids a large amount of search, but it can also become slower when the heuristic computation is not compensated by enough pruning.

LAMA first is a satisficing configuration based on the first configuration of LAMA (Richter and Westphal, 2010). It combines heuristic search with landmarks and is designed to find plans efficiently without proving optimality. A landmark is a fact or condition that must become true at some point in every valid plan, so landmarks give the search additional intermediate targets. LAMA uses such landmark information together with heuristic estimates that guide the search toward a valid plan, rather than toward a proof of optimality. It also uses preferred operators, which are actions suggested by the heuristic as promising next steps. These mechanisms can improve coverage and reduce search effort on many tasks, but they also create a runtime profile that differs from the optimal configurations. This changes the role of runtime compared with the optimal configurations, because the run can stop after finding a valid plan. The search may therefore succeed on tasks where proving optimality would be much more expensive. It is included to cover a common satisficing setup with different coverage and runtime behavior from the optimal configurations. Only the first configuration is used, so the run definition remains fixed.

LP based operator counting (Pommerening et al., 2014) computes admissible heuristic estimates from operator counting constraints using linear programming. Operator counting methods reason about how often actions must be used in any valid plan. These constraints form a linear program whose objective gives a lower bound for the remaining cost. Instead of relying only on graph search operations, the configuration repeatedly calls an LP solver to obtain heuristic information. The implementation relies on CPLEX (IBM, 2026). This makes the configuration useful for the thesis because part of the runtime is spent in solver based heuristic computation. It can therefore produce different runtimes, timeouts and failure modes from the other configurations.

3 Benchmarking

Benchmarking compares tools, algorithms or configurations through measured runs. In reliable benchmarking (Beyer et al., 2019), recording the command output is not enough. The setup also has to define resource limits, CPU placement, measurement methods and isolation between runs.

One practical issue is process handling. Some tools start child processes. If the measurement only follows the main process, part of the resource usage can be missed. The final experiments in this thesis do not start additional worker processes in the experiment infrastructure. Each Slurm array task calls one Fast Downward command. Process tree accounting is therefore not the main uncertainty here. It still matters conceptually, because it explains why cgroup based benchmarking tools were considered.

The experiments run on a shared Slurm system. Slurm (SchedMD, 2026) allocates CPUs, time limits and job placement. It provides the execution framework, but it does not give full experimental isolation. Runs may be placed on different nodes and other jobs may affect the system at the same time.

NUMA placement is another possible source of variation. On multi socket machines, each socket is connected to its own local memory, forming a NUMA node together with the cores on that socket. A core can usually access local memory faster than memory attached to another socket, because remote memory access has to pass through a socket to socket connection. If the operating system or scheduler places a process on one socket while its memory pages are located on another, memory access times can increase and the runtime can vary between runs. Memory binding tries to keep a process close to local memory. For this reason, it is tested as a setup option. The thesis does not claim to control all hardware effects. It measures what remains visible under the feasible Slurm setup.

BenchExec (SoSy-Lab, 2026) would provide stronger Linux cgroup based measurement and isolation. Such isolation is one of the benchmarking requirements discussed by Beyer et al. (2019). Linux cgroups (Kerrisk, 2026) can account for process groups and enforce limits. On the cluster used here, full BenchExec isolation was not possible because Slurm jobs could not create the required sub cgroups. The final method therefore combines Slurm resource requests with containerized execution and records the remaining limitations explicitly.

Containers fix the software environment. The experiments use Apptainer (Apptainer Project, 2026), which is designed for HPC use. The container fixes Fast Downward and its dependencies. It does not remove scheduling effects, NUMA effects or hardware interference.

4 Experimental Infrastructure

The final infrastructure runs Fast Downward in Apptainer containers on the sciCORE Slurm cluster at the University of Basel. Each planner run is one element of a Slurm job array. This matches the structure of the experiments, where many runs differ only in task, configuration, repetition or resource setting.

4.1 Software Stack and Pipeline

The reported experiments use Fast Downward 24.06.1, Slurm, Apptainer 1.4.5, Python scripts and CPLEX 22.2.0 for operator counting. The reported results come from the containerized Slurm pipeline.

The pipeline has three steps. First, a JSON experiment definition is resolved into concrete run files. Then Slurm executes the array tasks inside the Apptainer container. Finally, parser and plotting scripts aggregate the raw outputs into CSV, JSON and figure files.

4.2 Reproducibility Records

Each final experiment directory contains the resolved configuration, generated task list, Slurm script, raw result directories, parsed CSV files and exported JSON files. The resource usage analysis also writes a CSV file with CPU hours and approximate kWh values. Examples of these records are shown in Appendix A.1.

These files cannot guarantee identical runtimes years later. Hardware, system load, Slurm behavior and software versions can change. They do make the experiment definition inspectable. The task list, planner configuration, resource request, execution command, container path and parsed measurements remain stored with the results.

4.3 Data Collection

The parser extracts status, coverage, exit code, runtime and selected planner statistics from each raw run directory. Runtime analysis uses only successful runs with a valid `total_time`. This value is the planner runtime used in the variation plots. Timeouts and other failures are counted separately. Slurm's `CPUTimeRAW` is used for allocated CPU time because direct energy accounting was unavailable.

5 Experimental Design

5.1 Runtime Statistics and Filtering

For each benchmark task, planner configuration and execution setup, the successful repetitions form one runtime sample. The analysis computes the mean runtime, the standard deviation and the coefficient of variation (CV), which is defined as:

$$\mathrm{CV} = \frac{\text{standard deviation}}{\text{mean}}.$$

The mean is used for the CV based plots because CV is defined relative to the mean runtime. For the CPU allocation plots, the median is used instead, since it gives a more robust summary across the wider runtime distributions produced by different CPU counts. The CV relates variation to runtime scale. This is useful because a difference of a few milliseconds matters more for a task that runs for a tenth of a second than for a task that runs for several minutes.

Runtime statistics use successful runs with a valid runtime. Failed runs remain in the output and are used for coverage and failure analysis, but they are not converted into artificial runtime values. Direct comparisons use the tasks that provide the required successful repetitions in both compared settings. This avoids comparing different sets of solved tasks. The analysis remains descriptive, because the small repetition count gives limited statistical power for formal significance tests. The statistical consequences of this choice are discussed in Section 7.1.

5.2 Benchmarks and Configurations

The benchmark set contains 133 tasks from blocks, gripper, logistics00, pegsol-opt11-strips and sokoban-opt08-strips. These domains provide a mix of short runs and harder tasks under the chosen limits. The experiments use a strict wall clock time limit of 5 minutes and a memory limit of 4 GB per run. The exact problem list and limits are also stored in the generated experiment records. Very small tasks mainly show timing noise. Tasks that always fail mainly contribute to coverage information.

The planner configurations are UCS, A* LM-cut, LAMA first and LP based operator counting. Together, they provide different runtime scales, coverage behavior and failure modes.

5.3 Experiment Groups and Setups

The main repeated groups execute each selected planner configuration five times on each benchmark task. This gives 665 runs for one configuration and setup. The combined UCS versus A* LM-cut group contains 1330 runs, because it executes two configurations under the same experiment structure.

The setup dimensions are normal execution, memory bound execution and CPU allocation. In this thesis, normal execution denotes the baseline setup that requests exactly one CPU core and does not request memory binding. Memory bound execution uses the same one core baseline, but additionally requests local memory placement to test whether this NUMA related option changes the measurements. CPU allocation groups use 1, 2, 4, 8, 16, 32, 64 and 128 CPUs per task. The Fast Downward commands used here are single process runs. For that reason, the CPU experiments are not designed to show parallel speedup. They test how excessive CPU requests affect Slurm accounting and reserved cluster resources when a user assigns more CPUs than the application can effectively use.

5.4 Resource Usage and Energy Approximation

Slurm exposes `ConsumedEnergyRaw` and `ConsumedEnergy`. These would be the preferred source for electrical energy measurements. In this setup, both fields reported zero for the experiment jobs. A separate test job with `--acctg-freq=energy=30` also reported zero. Direct energy measurement was therefore unavailable.

Allocated resource usage is approximated with Slurm's `CPUTimeRAW`. The analysis sums CPU seconds over the main job array entries and excludes `.batch` and `.extern` steps to avoid double counting. CPU seconds are converted to CPU hours. A rough energy estimate multiplies CPU hours by an assumed average power draw per allocated CPU and divides the result by 1000 to convert watt hours to kilowatt hours:

$$\text{estimated kWh} = \frac{\text{CPU hours} \cdot \text{assumed W per CPU}}{1000}.$$

This is not measured electricity consumption. The absolute kWh values depend on the power assumption. The ranking by CPU hours is still useful, because it shows which experiment groups reserved the most scheduled cluster resources.

6 Results

The results follow the four questions from Chapter 1. The main text includes the figures needed for the argument and summarizes additional diagnostic plots in words. Table 6.1 uses the same notation as the experiment design.

All runtime plots use the filtering rules from Section 5.1. Where a comparison is paired, only common solved tasks are used. Table 6.1 gives the data basis for the main repeated experiments.

Experiment group	Normal (Solved)	Mbind (Solved)
UCS	380	380
A* LM-cut	497	500
LAMA first	660	660
Operator counting	385	385
UCS vs. A* LM-cut	880	880

Table 6.1. Coverage of repeated experiments.

Table 6.1 shows that coverage differs between configurations. Some configurations solve almost all selected runs, while others hit many timeouts or failures under the chosen limits. These differences matter when runtime plots are interpreted, because the available repeated samples are not identical for every configuration.

6.1 Q1 – Runtime Variation Across Repeated Runs

Q1 examines runtime variation for a fixed task, planner configuration and setup. The appropriate unit of analysis is therefore the task and configuration pair rather than the global runtime distribution. In Figure 6.1, each point represents a benchmark task and planner configuration with five successful repetitions. The x-axis shows their mean runtime, and the y-axis shows their coefficient of variation. Dataset specific plots indicate that the overall runtime scale is primarily determined by task and domain difficulty. Figure 6.1 therefore relates mean runtime to relative variation.

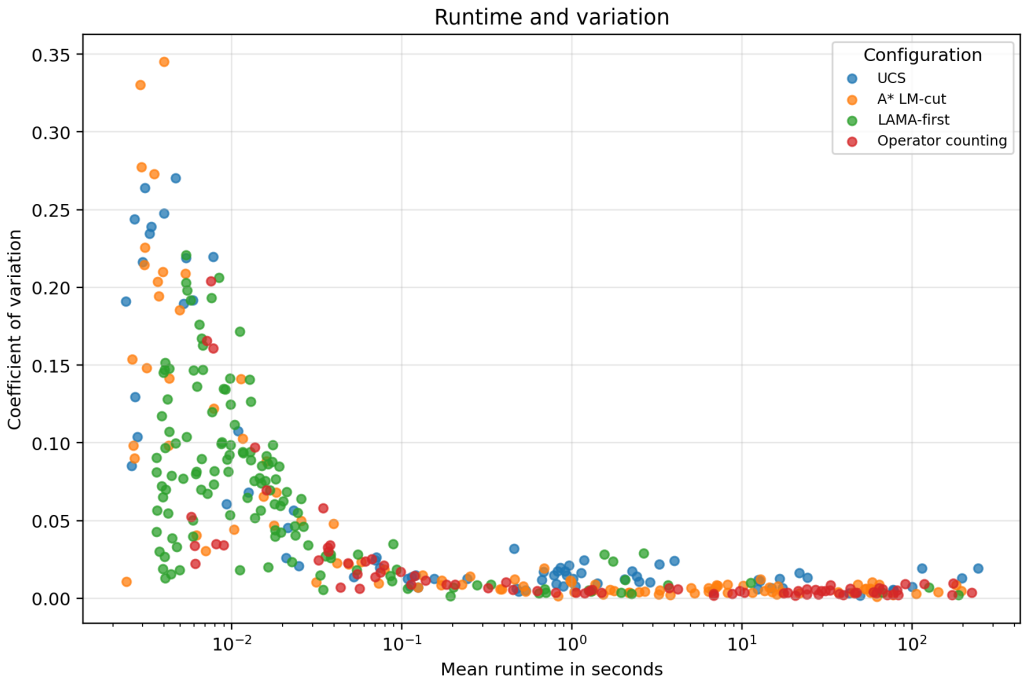

Figure 6.1. Runtime and relative variation. Each point is one task and configuration over five successful repetitions.

Figure 6.1 shows that high relative variation occurs mainly for very short tasks. For mean runtimes above roughly one second, the observed CV values remain below about 3%, indicating greater relative stability for longer solved tasks.

The repeated runs do vary. The variation is not spread evenly over all tasks. Very short runs react strongly to small timing differences. Longer successful runs are more stable in relative terms. This is why one global runtime plot can be misleading. It mixes task difficulty with measurement variation.

6.2 Q2 – Isolated Versus Combined Experiment Setup

Q2 examines whether including UCS and A* LM-cut in the same experiment affects their runtime measurements. Rather than comparing the two configurations directly, each configuration is compared between its isolated and combined experiment.

For each configuration, the isolated experiment is compared with the corresponding part of the combined UCS and A* LM-cut experiment. The analysis uses the same intersection of 76 tasks solved successfully in both settings. This paired design separates runtime differences from coverage differences and enables a direct comparison of the two plots.

Figures 6.2 and 6.3 sort the tasks by their isolated mean runtime. The isolated and combined measurements are then shown for the same task order. If the combined experiment changed the measurements in a systematic way, the combined line would consistently lie above or below the isolated line.

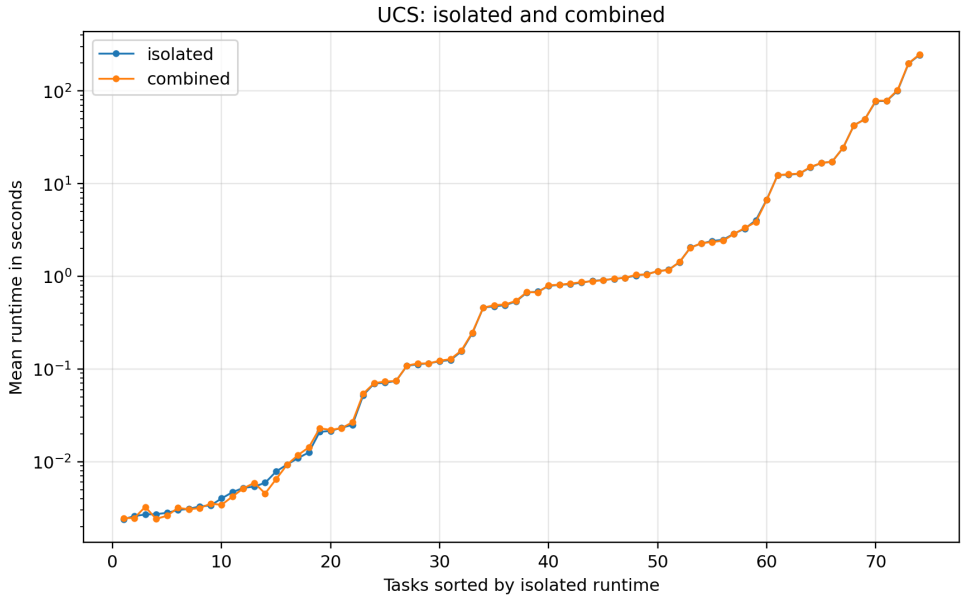

Figure 6.2. UCS in isolated and combined experiments.

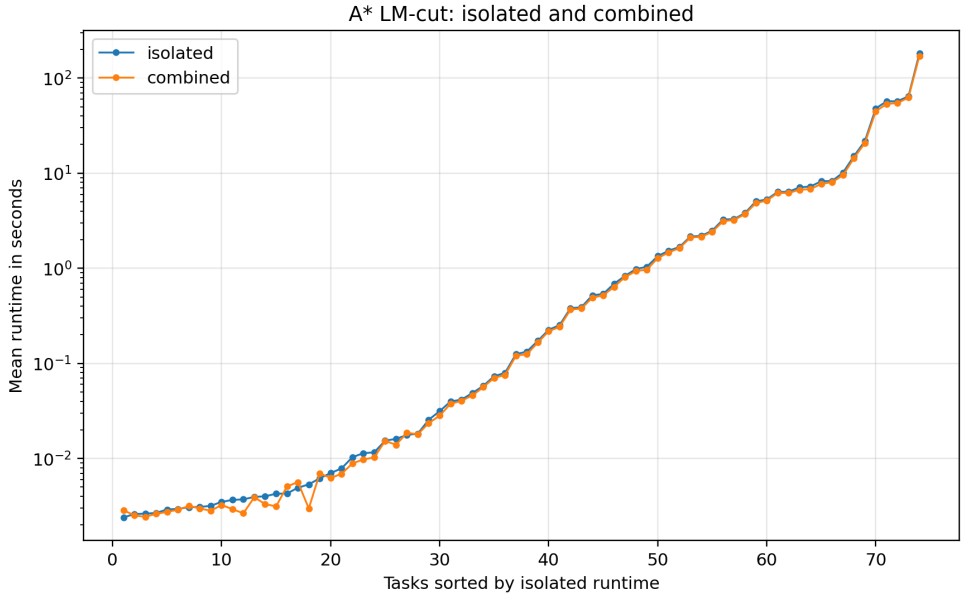

Figure 6.3. A* LM-cut in isolated and combined experiments.

The UCS measurements are nearly unchanged. Across these 76 tasks, the median combined to isolated runtime ratio is 1.0041. This means that the combined experiment produced nearly the same median runtime as the isolated UCS experiment.

For A* LM-cut, the median combined to isolated runtime ratio is 0.9634. The combined experiment is therefore slightly faster in the median, but the plot does not show a uniform shift over all tasks. The differences remain task dependent.

The CV comparison gives the same general picture. The median combined minus isolated CV difference is close to zero for both configurations, with -0.0016 for UCS and -0.0010 for A* LM-cut. This suggests that the combined experiment setup does not introduce a clear systematic change in runtime variation.

Overall, the combined setup does not appear to dominate the measurements. It changes some task level results, but not in one consistent direction. For this thesis, the result supports the interpretation that runtime and variation should still be analyzed per task and configuration, rather than only as one global effect of the experiment batch.

6.3 Q3 – Implementation Choices for Memory Binding and CPU Allocation

Q3 considers two implementation choices, memory bound execution and excessive CPU allocation. In the memory binding comparison, Normal denotes the one core baseline without memory binding. In the CPU allocation experiment, normal denotes the setup without memory binding, while the requested CPU count is varied explicitly.

6.3.1 Memory Binding

Memory bound runs are compared with normal runs for the same planner configuration and common successful tasks.

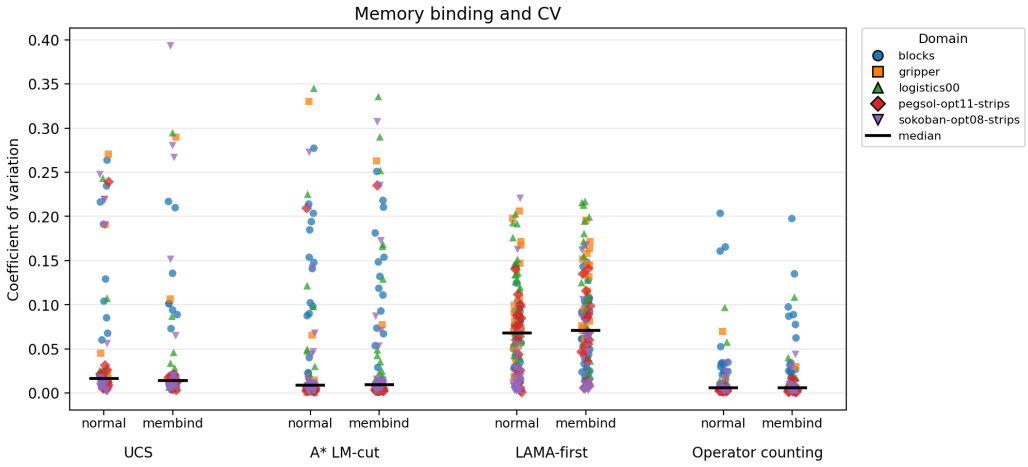

Figure 6.4. Memory binding and relative variation.

Figure 6.4 does not show a systematic reduction in runtime variation from memory binding. Median CV values are broadly similar in both setups. The setup plots for mean runtime and standard deviation lead to the same interpretation. Setup effects are smaller than task and configuration differences.

The larger CV values for LAMA first are likely related to its short runtimes rather than to memory binding itself. As shown in Figure 6.1, short successful tasks tend to have higher relative variation, because small absolute timing differences become large relative differences.

This does not mean that NUMA placement never matters. It only describes the benchmark set and cluster setup tested here.

6.3.2 CPU Allocation

The CPU allocation experiments vary the number of CPUs assigned to each run. The planner commands are single process runs, so no parallel speedup is expected. The experiment is included to show the accounting impact of requesting too many CPUs on a shared cluster. It asks what happens to runtime and allocated CPU time when a user requests far more CPUs than the application can effectively use.

For this analysis, the aggregation is based on common solved tasks. For each planner configuration and setup, the task set is restricted to tasks that were solved successfully at all CPU counts from 1 to 128. The plotted value is then the sum of the mean runtimes over this fixed task set.

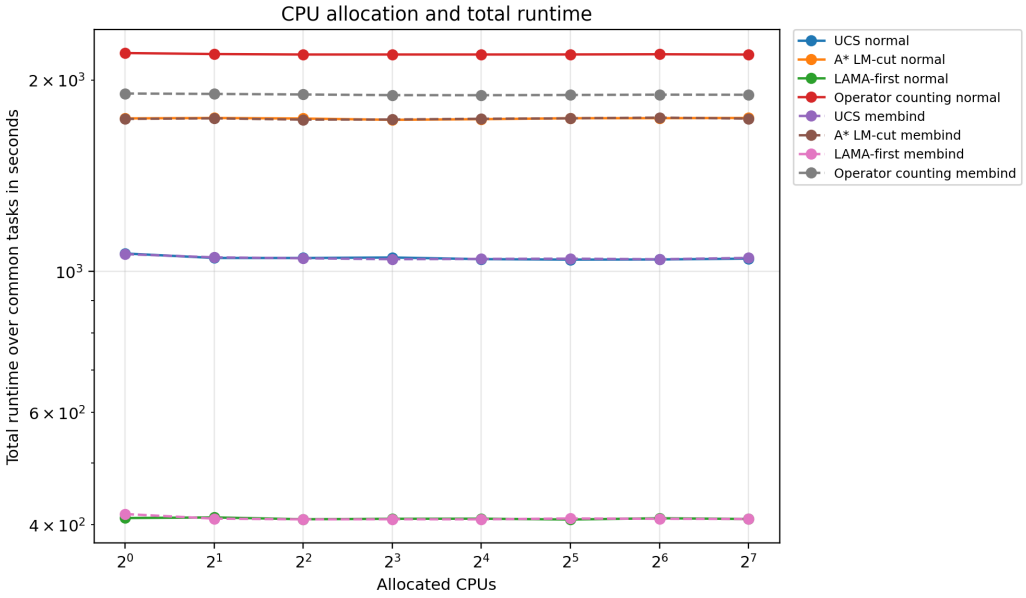

Figure 6.5. CPU allocation and summed runtime over common solved tasks.

Figure 6.5 shows no clear runtime improvement from assigning more CPUs. Within each line, the same common task set is used for all CPU counts. The summed runtime therefore compares the CPU allocations on a fixed set of solved tasks. The lines remain mostly flat, which supports the expectation that these single process planner runs do not benefit from additional allocated CPUs.

For Operator counting, the normal line is higher than the memory bound line. This should not be interpreted as a clear memory binding effect, because each line is based on the tasks solved successfully for all CPU counts within that setup. Since Operator counting has many failed runs, the common task sets for normal and memory bound execution can differ. The main result is therefore the increase in allocated CPU time with higher CPU counts, not the

absolute difference between these two Operator counting lines.

The resource effect becomes visible when the same summed runtimes are multiplied by the allocated CPU count. This gives the allocated CPU time for the common task set.

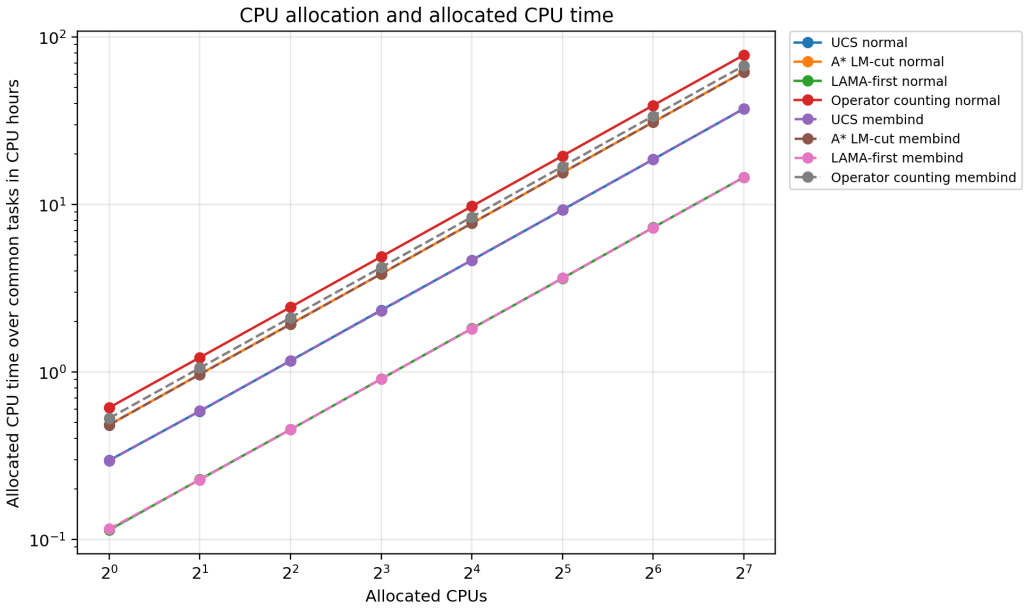

Figure 6.6. CPU allocation and allocated CPU time over common solved tasks.

Figure 6.6 shows a clearer pattern than the runtime plot. Because the runtime stays mostly flat while the CPU count increases, allocated CPU time grows almost proportionally with the number of requested CPUs. This is the main result of the CPU allocation experiment. Requesting more CPUs than needed does not systematically reduce planner runtime, but it strongly increases the amount of cluster resources reserved for the runs.

6.3.3 Resource Usage and Energy Approximation

The CPU allocation results also explain which experiment groups dominate resource usage. Resource usage is summarized with `CPUTimeRAW` as described in Section 5.4. Table 6.2 lists the largest experiment groups by allocated CPU hours.

The kWh values use 10 W per allocated CPU as a conservative reference value. This is based on the hardware information available for sciCORE compute nodes. For example, an Intel Xeon 6430 CPU (Intel, 2023) has a Thermal Design Power (TDP) of 270 W for 32 cores, which corresponds to about 8.4 W per core for the processor alone. This excludes node level overhead such as RAM, motherboard components and networking. The same assumption is used for all groups, so the ranking follows the allocated CPU hours.

Experiment group	CPU hours	Est. kWh at 10 W/CPU
Mbind CPU allocation, 128 CPUs	1668.20	16.68
Normal CPU allocation, 128 CPUs	1665.28	16.65
Mbind CPU allocation, 64 CPUs	832.00	8.32
Normal CPU allocation, 64 CPUs	831.45	8.31
Mbind CPU allocation, 32 CPUs	417.01	4.17
Normal CPU allocation, 32 CPUs	415.48	4.15
Mbind CPU allocation, 16 CPUs	208.13	2.08
Normal CPU allocation, 16 CPUs	208.03	2.08
Normal CPU allocation, 8 CPUs	103.99	1.04
Mbind CPU allocation, 8 CPUs	103.83	1.04

Table 6.2. Largest groups by allocated CPU time. Estimated kWh values are derived from CPU hours and an assumed 10 W per allocated CPU, not from direct energy measurements. Here Normal means no memory binding; the CPU count is specified in the group name.

Table 6.2 confirms the pattern from Figure 6.6. Runs with high CPU allocations dominate allocated resource usage. The 128 CPU groups are by far the largest entries, followed by the 64, 32 and 16 CPU groups. This shows that the estimated energy values mainly follow the allocated CPU hours. CPU allocation therefore changes the resources reserved for the experiment much more clearly than it changes planner runtime.

6.4 Q4 – Experiment Records and Repeatability

Q4 asks what helps when such experiments have to be inspected or repeated later. The records connect the experiment definition to the jobs that actually ran and to the plots that were produced. They include the resolved configuration, generated task list, Slurm script, raw outputs, parsed result tables, failure table, exported JSON summaries and CPU time resource summary.

These files show which tasks were run, which command was executed, which limits were requested, which runs succeeded or failed and which measurements entered the plots. They do not promise identical runtimes on a future cluster. They do make the experiment definition reconstructable. That is the practical form of repeatability used in this thesis.

The results also put the cluster setup into perspective. Memory binding and larger CPU allocations did not show a strong systematic runtime effect in the tested setting. This suggests that the setup choices studied here influenced allocated resources more clearly than planner runtime. Detailed records are therefore useful because they separate the experiment definition, the execution setup and the measurements actually observed.

6.5 Summary

The results address the four research questions. Runtime variation is best assessed per task and configuration using mean runtime, standard deviation and CV. The combined UCS and A* LM-cut experiment shows no strong systematic change relative to the isolated runs, and memory binding has no clear systematic effect in the tested setup. By contrast, requesting more CPUs than needed substantially increases allocated CPU time without a clear runtime benefit. The generated records document the tasks, commands, limits, outputs and parsing steps, making the experiments inspectable even though identical future runtimes cannot be guaranteed. When direct energy accounting is unavailable, `CPUTimeRAW` provides the most transparent proxy for resource usage.

7 Discussion

The results show that runtime variation should be interpreted at task level. Figure 6.1 shows that high relative variation mainly occurs for short successful runs. For tasks with mean runtimes above roughly one second, CV values are much lower. A global runtime plot would mostly show task difficulty and hide this pattern. For variance analysis, the useful unit is therefore a fixed task, planner configuration and setup measured over repeated runs.

The isolated versus combined experiment in Section 6.2 does not show a strong systematic shift. UCS and A* LM-cut behave similarly when compared with their isolated runs. Some task level differences remain, but the combined setup does not dominate runtime or CV. This supports the view that variation should be analyzed per task rather than as one global batch effect.

The same applies to memory binding. Figure 6.4 does not show a clear systematic reduction in CV. The higher CV values for LAMA first are more likely related to its short runtimes than to memory binding itself. This result does not rule out NUMA effects in general. It only shows that memory binding was not a dominant factor in the tested setting.

The clearest setup effect appears in the CPU allocation experiment. Figure 6.5 shows that summed runtime over common solved tasks stays mostly flat across CPU counts. Figure 6.6 shows that allocated CPU time grows strongly with the requested CPU count. Requesting more CPUs than needed therefore affects reserved cluster resources much more clearly than planner runtime.

Table 6.2 leads to the same conclusion for the energy approximation. The largest estimated values come from the high CPU allocation groups. Since the estimate is based on `CPUTimeRAW`, it mainly reflects allocated CPU time. It should therefore be used to compare resource usage between experiment groups, not as a direct measurement of electricity consumption.

Overall, the tested cluster setup choices did not create one simple runtime shift. Memory binding had no clear systematic effect, and larger CPU allocations did not improve these single process runs. The main effect of the setup appears in resource accounting. This distinction is important for future experiments, because planner runtime and allocated cluster resources answer different questions.

7.1 Limitations

The main limitation is infrastructure control. BenchExec could not be used with full cgroup based isolation because the Slurm environment did not allow the required sub cgroups. The final setup relies on Slurm requests and containers, but it cannot fully control low level interference, node placement or scheduler decisions.

The results depend on the selected configurations, benchmark domains, limits and sciCORE setup. Other planners, domains, limits or clusters may behave differently.

The experiments use five repetitions per task and configuration. This is enough to observe whether runtime instability occurs, but it is still a small sample size for estimating runtime variation. With $N = 5$, a single unusually fast or slow run can noticeably affect the standard deviation and the coefficient of variation. The variance statistics should therefore be interpreted as descriptive indicators of broad trends rather than precise estimates of the underlying runtime distributions.

Failure filtering also affects comparisons. Runtime statistics use successful runs only, so configurations with many timeouts have fewer comparable tasks. Pairwise comparisons reduce this issue by using commonly solved tasks. The CPU allocation analysis also uses common solved tasks for each configuration and setup, but these task sets can still differ between configurations.

Resource usage comparisons depend on the CPU time proxy defined in Section 5.4. Direct Slurm energy measurements were not available, so the estimated kWh values remain an approximation derived from allocated CPU time.

7.2 Practical Recommendations

The observations from Chapter 6 lead to three recommendations. First, runtime variation should be reported per task and configuration. Figure 6.1 shows that short tasks can have high relative variation, while longer solved tasks are more stable.

Second, runtime and coverage should be reported together. Table 6.1 shows that configurations differ strongly in solved runs. Without coverage information, runtime stability can be misread.

Third, planner runtime and allocated CPU time should be reported separately. The CPU allocation plots show that requesting more CPUs than needed mainly increases reserved cluster resources. It does not systematically improve runtime for the single process planner runs.

The generated records support this analysis by making the experiments inspectable. They show the resolved configuration, task list, Slurm script, container reference, raw outputs, parsed results, failure tables and analysis summaries. They do not guarantee identical future runtimes, but they document what was run and how the reported measurements were obtained.

8 Conclusion

This thesis studied how runtime and resource variation can be measured for repeated Fast Downward experiments on a shared Slurm based HPC cluster. The final setup used a containerized Slurm pipeline and several fixed planner configurations with different runtime profiles.

8.1 Answers to the Questions

Q1. Runtime measurements do vary across repeated runs of the same task and configuration. The size of the variation depends strongly on runtime scale. Very short successful tasks show the highest relative variation. Longer solved tasks usually have low CV values. Mean runtime, standard deviation and CV are therefore most useful when they are computed per task and configuration.

Q2. The combined UCS and A* LM-cut experiment does not show a strong systematic change compared with the isolated runs. Each configuration was compared with itself in isolated and combined experiment groups. Some task level differences remain, but there is no clear global shift in runtime or CV. Including both configurations in one experiment setup therefore did not dominate the measurements.

Q3. Memory binding does not show a clear systematic improvement in runtime stability for the tested setup. Increasing CPU allocation does not clearly reduce median runtime for the single process runs. It does, however, increase allocated CPU time. This separates planner runtime from reserved cluster resources.

Q4. Repeatability is improved by storing the resolved experiment configuration, task list, Slurm script, raw outputs, parsed CSV files, failure tables, exported analysis files and CPU time resource summaries. These records make the experiment definition transparent, even though identical runtimes cannot be guaranteed on a future shared cluster.

8.2 Overall Conclusion

Runtime and resource variation should be measured at the level of fixed task, configuration and setup groups. The results should then be interpreted together with coverage, failures, experiment setup and allocated CPU time. Global runtime summaries are still useful, but mainly for runtime scale. They do not by themselves explain measurement variation.

The combined UCS and A* LM-cut experiment did not show a strong shift compared with the isolated runs. The stronger setup effect was found in resource usage since requesting more CPUs than needed strongly increased allocated CPU time without a clear runtime benefit.

Allocated CPU time remains a transparent resource proxy. Its ranking shows which experiment groups reserved the most scheduled cluster resources.

8.3 Future Work

Future work could extend these experiments in three main directions:

First, to improve statistical robustness, increasing the number of repetitions per task would yield more stable variance estimates, enabling stronger conclusions about runtime distributions beyond descriptive trends.

Second, regarding energy validation, repeating the experiments on a cluster with direct energy accounting would allow a direct comparison between estimated and measured energy consumption.

Third, to achieve better hardware isolation, testing across different cluster partitions or node types would help to separate planner specific behavior from underlying hardware and scheduler interference.

Bibliography

Apptainer Project. Apptainer User Guide. Official documentation, 2026. URL https://apptainer.org/docs/user/main/. Accessed: 2026-06-04.

Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable benchmarking: Requirements and solutions. *International Journal on Software Tools for Technology Transfer*, 21(1):1–29, 2019. doi: 10.1007/s10009-017-0469-y. URL https://doi.org/10.1007/s10009-017-0469-y.

Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004. ISBN 978-1-55860-856-6.

Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. doi: 10.1109/TSSC.1968.300136. URL https://doi.org/10.1109/TSSC.1968.300136.

Malte Helmert. The Fast Downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006. doi: 10.1613/jair.1705. URL https://doi.org/10.1613/jair.1705.

Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling*, ICAPS 2009, pages 162–169. AAAI Press, 2009. URL https://ojs.aaai.org/index.php/ICAPS/article/view/13370.

IBM. IBM ILOG CPLEX Optimization Studio documentation. Official documentation, 2026. URL https://www.ibm.com/docs/en/icos. Accessed: 2026-06-04.

Intel. Intel Xeon Gold 6430 Processor Specifications. Official documentation, 2023. URL https://ark.intel.com/content/www/us/en/ark/products/232501/intel-xeon-gold-6430-processor-60m-cache-2-10-ghz.html. Accessed: 2026-07-16.

Michael Kerrisk. cgroups(7): Linux manual page. Linux man-pages project, 2026. URL https://man7.org/linux/man-pages/man7/cgroups.7.html. Accessed: 2026-06-04.

Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL: The planning domain definition language. Technical report, The AIPS-98 Planning Competition Committee, 1998. URL https://www.cs.cmu.edu/~mmv/planning/readings/98aips-PDDL.pdf.

Florian Pommerening, Gabriele Röger, Malte Helmert, and Blai Bonet. LP-based heuristics for cost-optimal planning. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling*, ICAPS 2014, pages 226–234. AAAI Press, 2014. URL https://ojs.aaai.org/index.php/ICAPS/article/view/13857.

Silvia Richter and Matthias Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39:127–177, 2010. doi: 10.1613/jair.2972. URL https://doi.org/10.1613/jair.2972.

SchedMD. Slurm Workload Manager documentation. Official documentation, 2026. URL https://slurm.schedmd.com/. Accessed: 2026-06-04.

SoSy-Lab. BenchExec: A framework for reliable benchmarking and resource measurement. Software documentation, 2026. URL https://github.com/sosy-lab/benchexec. Accessed: 2026-06-04.

Appendix

A.1 Experiment Record Examples

The final experiment directories contain the resolved experiment configuration, generated task list, Slurm script, raw outputs, parsed CSV files, failure tables, exported JSON files and optional CPU time resource summary. These files document the experiment definition and support later inspection.

The following snippets show the kind of information stored with each experiment. They are shortened for readability, but they correspond to the files used by the pipeline.

A.1.1 Resolved experiment configuration

A resolved JSON file records the selected task set, planner configuration, repetition count and resource limits.

```
{
  "name": "container_lmcut_all",
  "benchmarks": ["blocks", "gripper", "logistics00",
                 "pegsol-opt11-strips", "sokoban-opt08-strips"],
  "configs": ["astar_lmcut"],
  "repetitions": 5,
  "time_limit_minutes": 5,
  "memory_limit_mb": 4096,
  "cpus_per_task": 1,
  "container": "fd_24_06_1_cplex.sif"
}
```

A.1.2 Generated Slurm script

The generated Slurm script stores the resource request and the command structure used for the array tasks.

```
#SBATCH --job-name=fd_container
#SBATCH --cpus-per-task=1
#SBATCH --time=00:06:00
#SBATCH --mem=4096M

apptainer exec "$CONTAINER" fast-downward.py \
  --overall-time-limit 300 \
  --overall-memory-limit 4096M \
  "$DOMAIN" "$PROBLEM" "$PLANNER_ARGS"
```

A.1.3 Parsed result row

The parsed CSV files connect the raw run directory to the values used in the analysis.

Domain	Problem	Config	Status	Repetition	Total time
blocks	probBLOCKS-4-0.pddl	`astar_lmcut`	ok	1	0.42

Table A.1. Example of a shortened parsed result row.

University of Basel

Faculty of Science

Declaration on Scientific Integrity

(including a Declaration on Plagiarism and Fraud)
Translation from German original

Title of Thesis: Variance and Resource Usage of Computational Experiments

Name Assessor: Clemens Büchner

Name Student: Nillan Sivarasa

Matriculation No.: 23-053-457

I attest with my signature that I have written this work independently and without outside help. I also attest that the information concerning the sources used in this work is true and complete in every respect. All sources that have been quoted or paraphrased have been marked accordingly.

Additionally, I affirm that any text passages written with the help of AI-supported technology are marked as such, including a reference to the AI-supported program used. This paper may be checked for plagiarism and use of AI-supported technology using the appropriate software. I understand that unethical conduct may lead to a grade of 1 or "fail" or expulsion from the study program.

Place, Date: Basel, 17.07.26 Student: 

Will this work, or parts of it, be published?

No

Yes. With my signature I confirm that I agree to a publication of the work (print/digital) in the library, on the research database of the University of Basel and/or on the document server of the department. Likewise, I agree to the bibliographic reference in the catalog SLSP (Swiss Library Service Platform). (cross out as applicable)

Publication as of: _____

Place, Date: Basel, 17.07.26 Student: 

Place, Date: _____ Assessor: _____

Please enclose a completed and signed copy of this declaration in your Bachelor's or Master's thesis.