Distributed Parallel Build for the Isabelle Archive of Formal Proofs

Fabian Huch
Technische Universität München, Garching, Germany

Makarius Wenzel
Augsburg, Germany

Abstract

Motivated by the continuously growing performance demands for the Isabelle Archive of Formal Proofs (AFP), we introduce distributed cluster computing to the Isabelle platform. Parallel build time on a single node has approached 4 h–8 h in recent years: by supporting multiple nodes, without shared memory nor shared file-systems, we target at a substantial speedup factor to get below the critical limit of 45 min total elapsed time. Our distributed build tool is part of the regular Isabelle distribution, but specifically adapted to cope with the structure of projects seen in the AFP.

In this work, we address two main challenges: (1) the distributed system architecture that has been implemented in Isabelle/Scala, and (2) the build schedule optimization problem for multi-threaded tasks on multiple compute nodes. We introduce a heuristic tuned to the typical AFP session structure, which can generate good schedules in a few seconds. We reached a total speedup factor of over 100, which is a milestone never before reached. Using this approach, we could build the Isabelle distribution in 8 min 10 s elapsed time, and the AFP in 35 min 40 s, or 1 h 59 min 13 s including very slow sessions.

2012 ACM Subject Classification Computer systems organization → Distributed architectures; Software and its engineering → Distributed systems organizing principles; Mathematics of computing → Discrete optimization

Keywords and phrases Interactive theorem proving, Isabelle, Archive of Formal Proofs, Theorem prover technology, Distributed computing, Schedule optimization

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.22

Supplementary Material
Software (Experiments and Data): https://github.com/Dacit/isabelle-distributed-build
archived at swh:1:dir:748bbe787e4431fca08e42ab4d8fd64a4a3e6970

Funding *Fabian Huch*: Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under the National Research Data Infrastructure – NFDI 52/1 – 501930651.

Acknowledgements We are immensely grateful to Rainer Kolisch, who shaped the research on project scheduling in operations research with his seminal PSPLIB library, and to Maximilian Kolter, for contributing their invaluable expertise to this project.

1 Introduction

Motivation. The Isabelle proof assistant is accompanied by the Archive of Formal Proofs (AFP) [1], an online journal of formal content produced with Isabelle. The AFP is continuously checked against the underlying Isabelle version (official release or development snapshots).

The AFP was founded 20 years ago: the first entry is from 19-Mar-2004 [21], and in June 2024 there are ≈ 830 entries. The steady growth of AFP content challenges the underlying Isabelle platform, since proposed changes to Isabelle need to be pushed through all entries of the AFP. Such changes could affect interfaces of Isabelle/ML, fundamental syntax of the Isabelle/Pure framework, library content in Isabelle/HOL (e.g. term notation, default proof

© Fabian Huch and Makarius Wenzel;
licensed under Creative Commons License CC-BY 4.0

rules). Often, the initial change idea fails: it requires iterated refinement, until it becomes acceptable to the AFP – not causing too many follow-up changes there.

To retain the very notion of *interactive* proof development and maintenance, AFP test runs need to be reasonably fast. The folklore of AFP maintenance tells us that 45 min is a critical limit of "waiting online" for results, e.g. while doing other maintenance work elsewhere. Most of its lifetime, though, AFP tests have required much longer than that: often a rather painful 2 h–4 h. To help a bit, there is a classification of some AFP entries as `slow` (sessions that may take ≈ 1 h CPU time on their own account), or `very_slow` (sessions that require many hours). These tagged groups are tested less frequently than the main bulk.

In recent years, Isabelle users have been so successful in producing AFP content that we are now at 4.4 million source lines,[1] and the build time has degraded to 4 h–8 h (on a high-end server machine with many cores). Thus it has become increasingly difficult to change Isabelle libraries significantly: known problems often remain open for a long time.

That means it is high time to catch up with Isabelle prover technology. This has already happened several times in the past, e.g. with the introduction of shared-memory parallelism in Isabelle/ML from 2009 to 2013 [29, 19, 30], and replacement of GNU `make` by the custom-made `isabelle build` tool (after 2012). The next stage is to support *distributed parallel build* directly in Isabelle, to regain the 45 min limit for AFP built time.

Problem. The overall task is to organize the Isabelle build process better, to achieve a significant speedup factor that accommodates interactive development and maintenance: there needs to be enough headroom for the coming years of AFP growth. One side-condition is that Isabelle/AFP developments can be very large and thus produce large build results that need to be handled efficiently. Other side-conditions are given by the underlying ML system of Isabelle, Poly/ML[2] by David Matthews, and its traditional concept of *persistent heap image* from old LISP times. That is a convenient programming model from the past, which allows a growing context of ML code and values produced incrementally, but there are built-in limitations. ML heaps can only be stacked-up linearly – there can be forks, but no joins (so the dependency graph of Isabelle sessions always is a tree). Thus, large developments may often contain redundantly processed theories, to simulate session merges.

Solutions. There are various axes to tackle the problem, and several solutions may be combined eventually. At first we distinguish *internal organization* about how Isabelle/ML works vs. *external organization* of running ML sessions; the latter is done in Isabelle/Scala.

The past 15 years have seen many internal improvements of Isabelle/ML, notably shared-memory multiprocessing (e.g. yielding a solid factor of ≈ 5–6 on 8 cores), but also runtime sharing of ML values. Those have already provided significant improvements to scalability, but also cause quite some complexity in the Poly/ML platform. Without exceedingly difficult engineering, significant gains cannot to be expected in this area, according to collected timings [16].

An external approach is to reorganize Isabelle session structure on the spot. The static description (via `ROOT` files) consists of session name, parent, used theories, and optionally re-used theories from other sessions. There is no need to build ML heaps precisely around this structure. The Isabelle/jEdit Prover IDE already allows to build a custom heap image, for the true requirements of the edited session (via `isabelle jedit -A -R`). The build tool

[1] `https://www.isa-afp.org/statistics`
[2] `https://www.polyml.org`

could do likewise, and load common library theories into one big session used as prerequisite for AFP, e.g. the union of `HOL-Library`, `HOL-Analysis`, and `HOL-Algebra`.

A less conservative variation is to impose the `skip_proofs` option on theories that are re-used from other sessions. This would imitate the theory (object) files known from the HOL4 prover, which is also based on Poly/ML. In any case, the expected speedup factor is limited to the currently observed waste factor of ≈ 1.75 in the AFP.

A more promising approach is to allow multiple build hosts, to open the game of distributed parallel computing: compute clusters may easily consist of hundreds or thousands of nodes, so a large speedup factor can be expected for the future. This merely requires to manage distributed builds properly, which is the plan of this work: for small clusters of 10–20 nodes.

Contribution. We introduce Isabelle technology for distributed parallel build of the AFP, with only minimal requirements on system infrastructure, and support for heterogeneous hardware that is often found in scientific environments, consisting of fast and slow machines. We provide build schedule optimizations based on an initial set of empirical data, and evaluate the resulting schedules and system performance with two more data sets

2 Distributed Build Architecture

Our overall approach is that *"Isabelle is the build system of Isabelle"*. Regular users often identify "Isabelle" with the Isabelle/HOL object-logic, but the reality is more complex: Isabelle is a platform of different languages with different purposes (although with a common emphasis on λ-calculus and functional programming). Two Isabelle languages are particularly important: (1) Isabelle/ML as *mathematical programming language* to implement formal logic, specification mechanisms, proof tools etc. – that provides only minimal system operations, e.g. to access external files. (2) Isabelle/Scala as *functional language for systems programming*, with full-scale access to file systems, database engines (SQLite or PostgreSQL), external processes (GNU Bash), operating system tools (rsync, SSH, XZ, Zstd), TCP/IP services etc. Scala runs on the Java/VM and acts like an abstract operating-system for organization of Isabelle applications: it works uniformly on Linux, macOS, or Windows (with Cygwin).

So we shall rather identify "Isabelle" with the Isabelle/Scala environment for systems programming: this is where our main implementation happens.[3] Thus we follow the tradition of the first (more modest) `isabelle build` tool from September 2012.[4]

2.1 Re-use the Slurm Cluster Workload Manager?

Quite a lot of systems for compute cluster management have appeared in the past 10–20 years. In the world of scientific computing, the most popular one is now Slurm [17]. Major centers for high-performance computing (HPC) often use Slurm as a "meta-operating system", to manage queues of user jobs. A running job appears on several compute nodes at the same time. The user program sees a conventional Linux system with network access; usually there are additional means to communicate with sibling nodes via message passing (e.g. OpenMPI).

The HPC Linux Cluster of the Leibniz Supercomputing Centre (LRZ) is next door to TU München in Garching. Thus we were motivated to explore this infrastructure carefully,

[3] `https://isabelle-dev.sketis.net/source/isabelle/browse/default/src/Pure/Build/;29f2b8ff84f3`

[4] `https://isabelle-dev.sketis.net/source/isabelle/browse/default/src/Pure/System/build.scala;aa1e2ba3c697`

with the help of a student project that produced an Isabelle workload manager based on Slurm [20], but running on our own Linux computers or virtual machines from the LRZ. The main lessons from this experiment are:

1. Slurm is primarily for Linux, and unavailable on macOS or Windows.[5] This is bad, because macOS is important in the ITP community, and we intend to test AFP on it eventually.
2. Slurm set-up is rather complicated, requires special Unix permissions, and is not easy to extend. In particular, Slurm needs a custom authentication service (system daemon), demands uniform user and group IDs, and has static cluster configuration that makes it difficult to add new machines. This is in conflict with typical compute resources found in academic environments: often there is only non-root access and standard system software.
3. Slurm does provide mechanisms for planning and scheduling jobs, but these are insufficient for our purpose, as they cannot use as much of our domain-specific information: For instance, Isabelle sessions can be built in different configurations (regarding ML threads and heap size). With a prediction model for the required computation time and resources, a build schedule could be planned in advance and optimized accordingly.
4. Although Slurm is very popular, its feature set turned out insufficient to handle high-end workstations with different kinds of CPU cores (hot “P-cores” versus cool “E-cores”).

Hence, we decided not to use an off-the-shelf component for cluster management, and instead do it ourselves in Isabelle/Scala, with minimal requirements and better results.

2.2 System Design based on Isabelle/Scala + SSH + PostgreSQL

Our design makes only minimal assumptions about the system environment:

1. Uniform platform: All build machines (master and worker nodes) operate on the same platform, after an initial decision if that is Linux, macOS, or Windows – and Intel or ARM.
2. Repository clones: official Mercurial repositories of Isabelle[6] and AFP[7] are required at the root of the overall build process, but regular build hosts get sources via `rsync`.
3. Local file systems: Working directories reside on fast solid-state storage. Network shares are neither required, nor desired: too slow and difficult to use concurrently.
4. SSH: The master node can reach all worker nodes via `ssh`, to access files and to run processes. SSH users and home directories may vary, but the shell needs to be GNU `bash`.
5. PostgreSQL: There is a dedicated PostgreSQL database server[8], which is accessible via LAN/WAN from all build nodes (using TCP/IP directly or tunneled through SSH).

SSH clients and servers are readily available on all platforms, usually from the single source of the OpenSSH project, which works reliably. In Isabelle/Scala, `ssh` works as stay-resident program: a single connection is multiplexed for many requests. The PostgreSQL database server is available in many forms on many systems, e.g. as a system package on Ubuntu Linux that requires approx. 10 min to set-up – we are using version 14, e.g. on Ubuntu 22.04

---

[5] `https://slurm.schedmd.com/platforms.html`
[6] `https://isabelle.in.tum.de/repos/isabelle`
[7] `https://foss.heptapod.net/isa-afp/afp-devel`
[8] `https://www.postgresql.org`

LTS. Accessing the database server through SSH tunneling (via Isabelle/Scala) avoids the need to configure extra SSL certificates. The `SQL` module in Isabelle/Scala provides a simple programming interface, based on PostgreSQL JDBC for Java. All other system dependencies are bundled with the Isabelle distribution, notably Scala 3 and OpenJDK 21. This now also includes the `rsync` tool, in just one version for all platforms, so that it can speak to itself reliably over a cluster of dissimilar machines.

These parts are fit together as the distributed parallel build infrastructure for Isabelle as shown in Figure 1. The cluster topology is given on the command-line via "`isabelle build -H` *hosts*`:`*options* ...", based on predefined hosts and host groups from a global registry file (in TOML format). The master oversees the build process, using database tables that are shared with all workers. Workers may in principle join and leave the build process freely, but presently we use a fixed arrangement determined at the build start. The master and workers all run the same Isabelle/Scala program, with a conventional programming model of *critical regions* and *functional update* of *shared state* (similar to *synchronized variables* in Isabelle/ML and Isabelle/Scala). The distribution of state information happens via the database server, using our own implementation of transactions with table locking, and efficient propagation of incremental updates (that works like a version control system, using a relational `OUTER JOIN` of recent updates against latest data). Since full transaction locking is rather costly (minimum $\approx 100\,\text{ms}$, median $\approx 1\,\text{s}$, sometimes much longer), we use the special PostgreSQL commands `LISTEN` and `NOTIFY` for asynchronous IPC messages. This reduces the frequency of state synchronization for critical regions: the master signals when relevant state updates have happened, so workers run silently most of the time without polling the database.

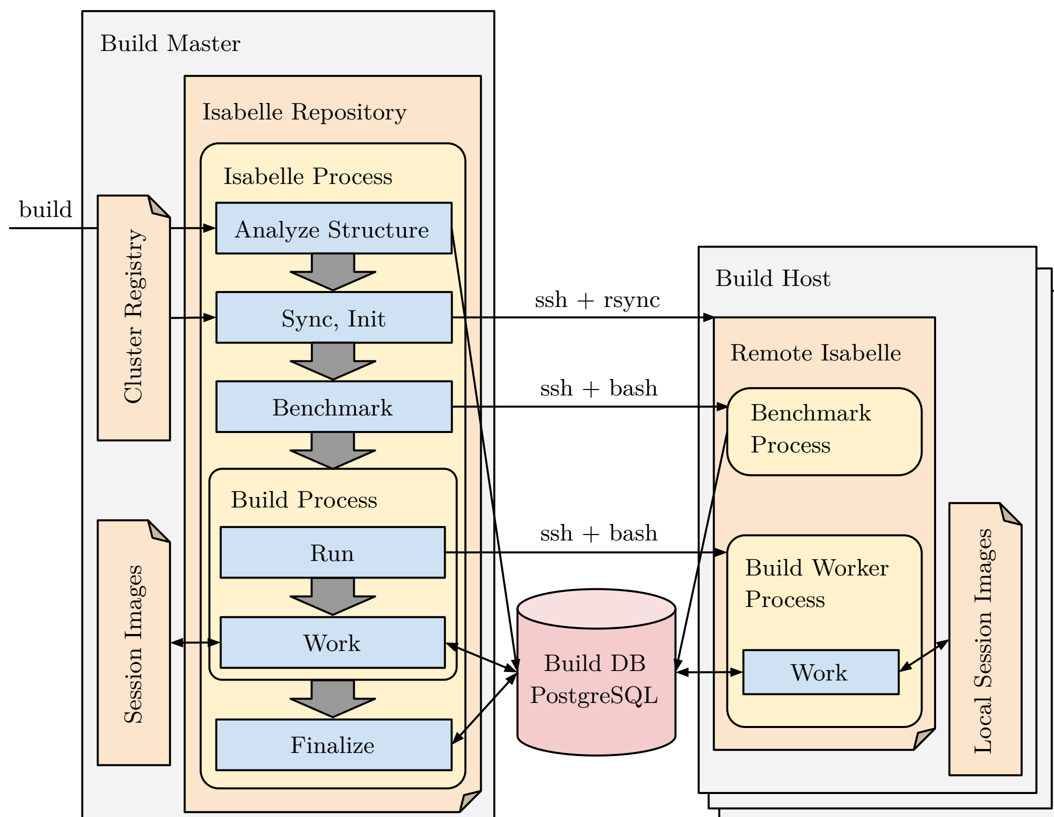

Figure 1 Overview of the distributed Isabelle build system.

Seen from the master build host, the main build phases of Figure 1 are as follows:

1. *Analyze* the source structure of selected sessions (as defined in ROOT files), including full theory dependencies (given by theory headers). The resulting data is transferred to the database and can later be retrieved by workers. File names are stored with symbolic paths, relative to the root directories of Isabelle or the AFP.
2. *Synchronize* the master Isabelle distribution to all workers via `ssh` and `rsync`. This includes Scala/Java build artifacts and thus saves 1.5 min to initialize remote clones. The command-line `isabelle build -A:` treats the AFP directory structure analogously, but other project directories must already be present on each worker node.
3. *Initialize* the worker Isabelle distributions, by letting each node download required system components from the Isabelle components store, and finally ensure that all Scala/Java modules are up-to-date. Afterwards, workers are ready to run.
4. Run a short *benchmark* session by each worker, if explicit schedules should be used. This helps to predict overall run-times as discussed below.
5. *Run* all workers and let them *work* on the build autonomously, using queue and schedule information from the database. The master could participate as another worker, but is disabled by default, as it often resides on a slow workstation.
6. *Finalize* the build when all pending sessions are finished. This includes HTML and LaTeX presentation for all sessions, based on markup information found in session databases.

For the AFP, being more a scientific journal than a library of code, the presentation is the main outcome of a successful build. In contrast, the result of running an Isabelle/ML session is an opaque *heap image*, which is potentially large. So we compress heap images with Zstd (portably in Isabelle/Scala) and store them in separate tables of the database as raw blobs (avoiding extra compression by PostgreSQL). Remote workers check and restore required heaps from the shared store: local file systems act like a cache for heaps served by the database.

Care is required to manage ML heaps in a scalable manner. For example, the `HOL` image is at ≈ 200 MiB uncompressed, ≈ 50 MiB compressed, with 0.9 s/0.3 s compression/uncompression time (on a fast machine). To give a sense of scale: Committing the compressed blob requires roughly 1.5 s. Most Isabelle/AFP session images are much smaller, but can sometimes approach 1 GB. To allow very large applications in the future and circumvent the PostgreSQL limit of 2 GB for blobs, we store heap images as slices of ≈ 50 MiB.

After several rounds of performance tuning of our Isabelle/Scala implementation, the PostgreSQL engine works fairly well, both for small-scale synchronized program state, and large-scale storage of session images. Thus we can work without network shares, and really treat worker hosts as independent machines (apart from user-level SSH connections).

2.3 Prediction of Run-time per Configuration

A critical component for the schedule generation is an accurate estimation of run-time for a given configuration. Generally, there is little hope to estimate that without actually running the session: For instance, the AFP theory `Lorenz_C1` contains a single innocent-looking locale interpretation (three lines), whose run-time is about 50 h CPU-time on average test hardware. Still, once the run-time is known for some configuration, we can predict others. We use the `build_log` database of Isabelle to get data from historic builds. Our estimation is based on two main principles:

1. Given a job and number of threads, we estimate how the time changes across different build hosts via a factor derived from an initial benchmark. The benchmark session runs in single-threaded mode, it should be neither too short nor too long, its content should

rarely change, and it must be representative. Figure 2 shows the mean square error (MSE) of all sessions when used as individual benchmarks to scale the run-time of other sessions across hosts. There is a distinct separation in the build time since most sessions require `HOL`, which accumulates to 155 s already. `FOLP-ex` stands out as the session with lowest MSE amongst the faster sessions, and only takes about 15 s. All sessions with lower MSE are much slower. Further experiments with the initial fragment of HOL as benchmark all exhibited a far higher MSE, and the sessions that stick out have in common that they manipulate large terms. A likely explanation is that the typical workload seen in most sessions consists of such term manipulations, and the Isabelle/ML loading in Pure and bootstrapping of HOL is not representative.

2. If a job has been run with a different number of threads on the same host, we can interpolate the curve to the unknown point. However, the speedup has lots of variation, as illustrated by the speedup curve for `HOL-Analysis` shown in Figure 3. Even though the speedup factor varies, it always increases up to an optimum at 8–16 threads and then decreases again due to multithreading inefficiencies. Although one could try to train a sophisticated prediction model for the speedup, in practice only very few configurations are known most of the time and robustness is more important than accuracy. Hence we make use of the only underlying mechanic that we can safely assume: According to Amdahl's law [14], in any task only a fraction p of the total work T is parallelizable, so the total time for n threads is $\frac{p}{n}T + (1-p)T$. Thus, we select the prefix for which the speedup is monotone, estimate p by $pT = \frac{nm(T_n - T_m)}{m-n}$ for any two present timings T_n and T_m (for n and m threads, respectively), and interpolate according to that. If there is no such prefix, we use a simple rectified linear interpolation. This approach decreases the mean relative error (MRE) from 0.52 (linear only) to 0.42.

Our estimation then works as follows: If there is no historic build of a given job, we either use the session time-out if it is defined, or we take the average time of all jobs. If there are no other jobs to approximate from, we use dummy data to bootstrap the process. Otherwise, if there is data for the right job and threads, we just scale it to the right host. Should there be no data for the given number of threads, we try to interpolate. If there are not enough data points on the given host to do that, we interpolate on all other hosts individually and scale the results up to the current host. In case there is no host with enough data points, we unify the data and scale the individual points up to the current host, and then interpolate. Finally, if there is only one data point, we use the global speedup curve for the interpolation. Figure 4 shows boxplots of the relative error in the different scenarios outlined above. Unsurprisingly, the error is lower in the scenarios where less prediction is necessary.

In the final estimation, we are interested in producing a result that does not lead to underestimations which could be exploited by the schedule optimization – especially since a delayed task can cause problems for the whole schedule, but faster completion is not a problem. Hence, we increase the estimations slightly depending on the scenario.

3 The Scheduling Problem

The classic `isabelle build` tool immediately starts new session jobs whenever there is free capacity. If there are multiple sessions to select from, the session with the longest build time in its path of successors is chosen. To estimate the build time, the timing of the previous build is used, if available. Otherwise the static session timeout serves as a rough approximation of actual runtime. The configuration of the number of threads for the Isabelle/ML process is a static parameter, usually given on the command-line.

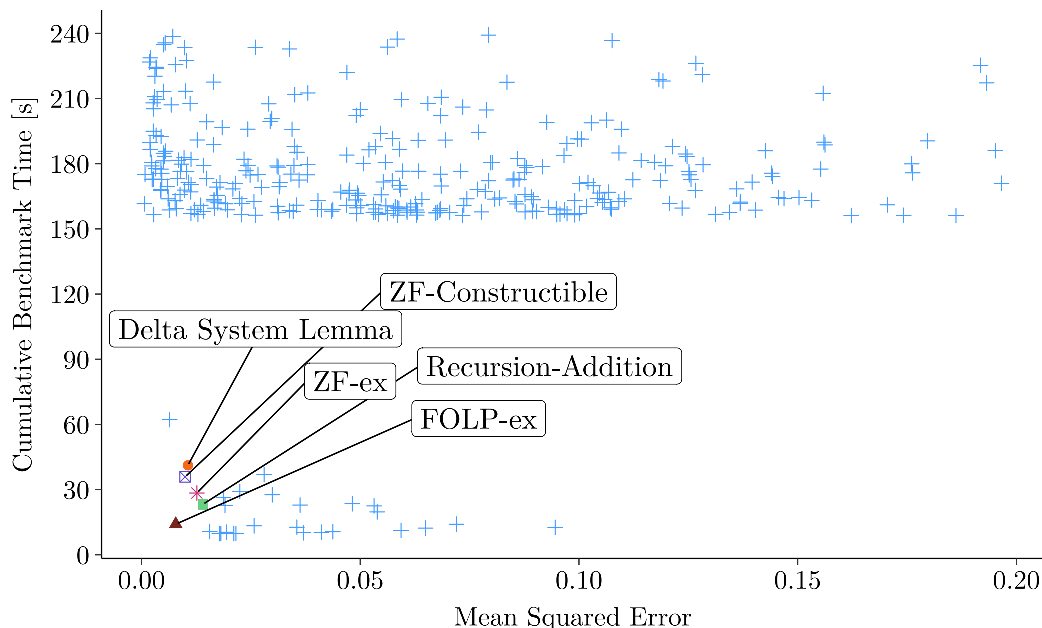

Figure 2 MSE vs. cumulative benchmark time (single-threaded build of benchmark session, after parallel build of its requirements), cut off at 240 s or MSE of 0.20. The MSE refers to the estimation error when scaling build times across different hosts, using on relative benchmark time as factor. Only sessions from the preliminary data set with at least 10 different data points are shown.

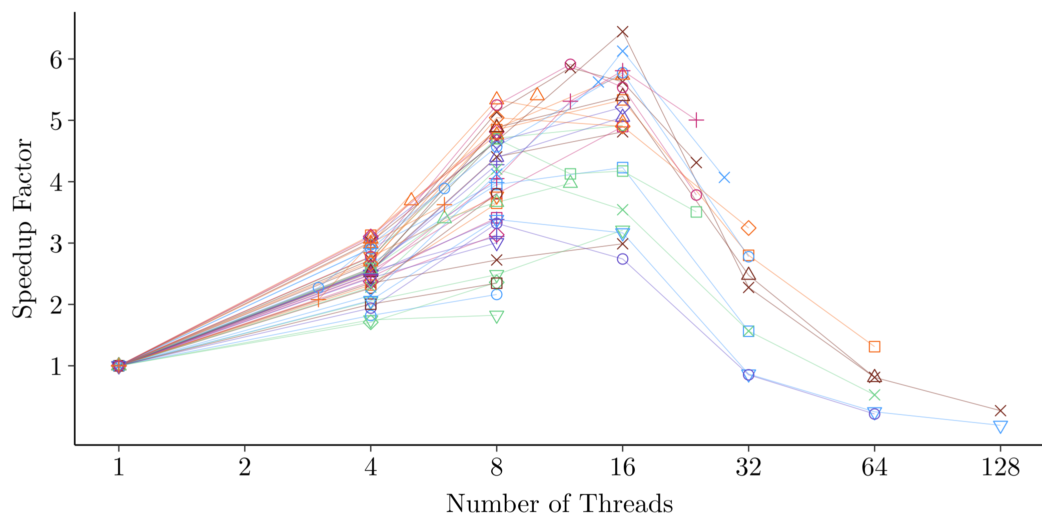

Figure 3 Number of threads vs. speedup factor for `HOL-Analysis` on various CPUs, using data from [16]. The speedup is the elapsed time, divided by single-threaded run-time on the same machine.

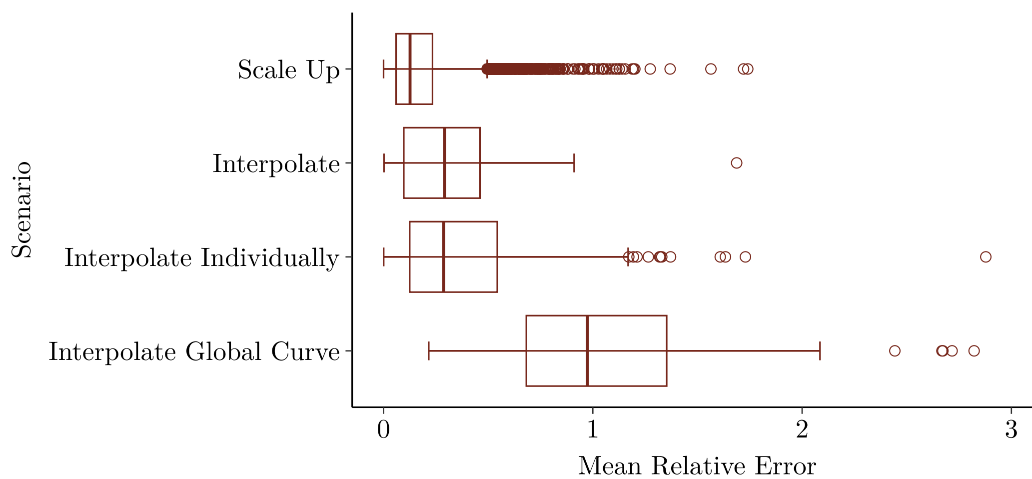

Figure 4 Box plots for MRE of interpolation in different scenarios on preliminary data. Whiskers mark 1.5 inter-quartile range.

This now works differently: the command-line option "`-o build_engine=...`" tells which implementation of build scheduling should be used. The default is merely an upgrade of the old approach, to work with multiple build hosts. In addition, there is now the "`build_schedule`" engine that implements advanced scheduling as described in the following.

While the old strategy was sufficient for a low degree of session-parallelism, it is far from optimal for parallel builds in a large heterogeneous cluster, where the compute power may vary greatly across machines. The two different levels of parallelism, as well as non-uniform memory access and different kinds of processor cores found in current CPUs, make for a very uneven scheduling problem. Hence, basic approaches to multiprocessor scheduling are not applicable. Instead, the scheduling problem at hand has been studied as the multi-mode resource-constrained project scheduling problem (MRCPSP) since the late 1960s [7]. Some variations of the exact formulation of this problem exist in the literature. Typically, it consists of finding a schedule for activities $\mathcal{A}$ in different modes $\mathcal{M}$, where each configuration (pair of activity and mode) takes $T\colon \mathcal{A}\times\mathcal{M} \to \mathbb{N}$ time, such that the overall *makespan* (i.e., the time it takes to process all activities once in any mode) is minimized. The problem is subject to precedence constraints according to some acyclic precedence relation $\prec\colon \mathcal{A}\times\mathcal{A} \to \mathit{bool}$, and there are constrained (renewable and non-renewable) resources $\mathcal{R}$. Each resource has an associated capacity $C\colon \mathcal{R} \to \mathbb{N}$, and a certain amount is required to run a configuration according to some cost function $R\colon \mathcal{A}\times\mathcal{M} \to \mathcal{R} \to \mathbb{N}$.

In our case, we need to schedule the build of a selection of sessions as activities, adhering to the dependency graph that induces the precedence relation. Each session can be run with different arguments such as the number of ML threads or the maximum ML heap size, though we only consider threads in this work. The build hosts (machines) define our resources: Each host has a number of CPU cores, which is a capacity that limits the sum of threads used. A second capacity is given by the maximum number of sessions that may run in parallel on each host. Finally, the run-time is determined by the session, build host, and number of threads as discussed in Section 2.3. Since in the MRCPSP formulation the run-time is independent of the resource used, we employ one mode per host and possible number of threads.

As we may not know the exact run-time and have to estimate it from historical data, we do introduce some uncertainty. Although stochastic approaches to the problem exist [26, 18], we work with the deterministic variant since it is simpler and more widely studied, and because the worst-case time is less of a concern – we are more interested in obtaining a solution quickly with good average time (which is a situation where this approach is appropriate [2]). Moreover, there is no need that the schedule stays fixed throughout the build: Should the actual times deviate (or a better solution be found due to the decreasing search space), the new schedule can be dynamically adopted. Hence, we re-compute the schedule periodically during the build.

3.1 Computational Complexity

Already the much simpler scheduling problem (considering identical processors instead of resources) is NP-complete [27]. Moreover, the MRCPSP (with resources and processors) remains strongly NP-hard even when restricting the problem simultaneously to one resource, uniform activity durations, a dependency graph that is a tree, and two processors (or three when the graph is omitted) [10]. It can also not be approximated by a constant factor in polynomial time even for a less general variant [9]. In our scheduling problem, in addition to the dependency graph being a tree, we can assume that T typically exhibits a monotone behavior: For all sessions, faster CPUs and more Isabelle/ML threads (up to a limit) usually correspond to less runtime. However, this does not improve complexity as the problem remains strongly NP-hard even with only two machines, a chain of dependencies, one single resource, as well as uniform mode, capacity, cost, time, and speed [4].

3.2 Schedule Generation Methods

Due to the practical importance in many fields and industries, a vast number of approaches exist to the problem (cf. [25] for a recent comprehensive resource). Exact solutions are often branch-and-cut methods which iteratively constrain the LP-relaxation of the mixed integer program for the optimal solution [33]. However, these integer programs explode in size: Due to the linear nature, one variable is required for every possible configuration of every activity at every time step. In another approach, Bofill et al. [5] use iterative SAT solving to find the lower bound for which an encoding of the problem is still satisfiable such that the satisfying assignments correspond to a schedule. Both this implementation as well as a recent implementation of the linear programming based-approach (as well as [12] and our own implementation of another mixed integer approach) could find the optimal solution for a small problem with 28 activities[9], but found no reasonable bounds for larger problems.

Meta-heuristic optimization approaches such as genetic algorithms, local search, or simulated annealing (where neighbor solutions are constructed for an initial feasible solution to find optima) have become quite popular in the field [23]; these can find approximative solutions for larger problems. We observed that a local search implementation [11] and one of the original genetic algorithm approaches [13] found solutions for the small problem very fast, but struggled for our medium-sized problem (132 activities)[10] due to technical limitations. Using OptaPlanner[11] (which combines several meta-heuristics), we could generate a good solution for this problem in short time, but did not find a solution for our large problem

[9] Building all sessions in the Isabelle distribution excluding `HOL` on one machine, with 1 or 8 threads.
[10] Building the whole Isabelle distribution of a cluster of 4 fast machines, with 1 or 8 threads.
[11] `https://www.optaplanner.org`

(923 activities)[12]. We found that only some constraint programming solvers were able to tackle problems of this size. While the open-source Google OR-Tools[13] could solve some of the large problem instances given enough time, IBM ILOG CPLEX[14] excelled for this task, finding a reasonable solution for the large problem in a few minutes.

However, since CPLEX is proprietary software that cannot be bundled with Isabelle, we focus on heuristic-based *schedule generation schemes* (SGS) in the present work. These can also provide an initial solution to aid further optimization, e.g. by providing an upper bound on the makespan for the constraint programming formulation. SGS yield very fast solutions at the expense of solution quality by iteratively building a schedule, maintaining a decision set of activities whose precedence constraints are fulfilled. In the *serial* SGS, at every step a configuration is selected according to some *priority rule* (e.g., selecting the activity with the most successors in the fastest mode) and scheduled in the earliest possible time. In contrast, in every time step of the *parallel* SGS, a priority rule is employed to select from the set of currently feasible configurations until no more can be scheduled, at which point the scheme proceeds to the next time step. This latter approach is greedy since activities are scheduled immediately when they are feasible, which limits the search space, but also means that the optimal solution may not be found (independently of the priority rule).

3.3 Priority Rules for Isabelle builds

In the literature, a vast amount of priority rules have been explored, and ultimately it depends on the problem structure which rules perform best. Our exact problem structure can vary depending on which sessions need to run, and which hardware is available. Still, we can make some general observations that hold for AFP builds on a typical cluster:

1. Session sizes and in-degrees in the dependency graph (the number of other uses of a session) weakly follow a scale-free distribution [15], i.e. there are few long-running or frequently-used sessions, and many sessions which are fast or without dependencies.
2. As machines in research environments often accumulate over the years, there will be relatively few of the fastest machine, but plenty of slower ones. Moreover, machines with many cores (server CPUs) typically have far lower clock speed than high-end consumer CPUs with fewer cores, so the more cores a CPU has the slower they are.
3. As the number of threads for a single session increases, the elapsed time decreases (up to some limit, typically 16 threads), but the parallel efficiency also degrades.

Hence, it is desirable to build sessions that greatly influence the overall makespan with the optimal amount of threads on the hardware with fastest individual cores, and sessions that have little to no impact in single-threaded mode on slow hardware. Due to the scale-free nature, we can hope that constructing a greedy solution that takes this into account yields a near-optimal solution. Our heuristics hence employ the notion of *critical* sessions, which are sessions that are on a path in the dependency graph whose accumulated time (for optimal amount of threads) is longer than some cut-off. The cut-off is taken either at some absolute value, or as a factor of the critical path through the dependency graph. A number of the fastest hosts is reserved for these critical sessions: Either one for every critical session that needs to be built in parallel, a fixed fraction of the fastest machines, or all machines faster than some cut-off. Finally, the uncritical sessions are run with a number of threads that increases for slower sessions (or is always constant).

[12] Building the AFP (without slow sessions) on an experimental cluster of 14 machines, with 1 or 8 threads.
[13] `https://developers.google.com/optimization`
[14] `https://www.ibm.com/products/ilog-cplex-optimization-studio`

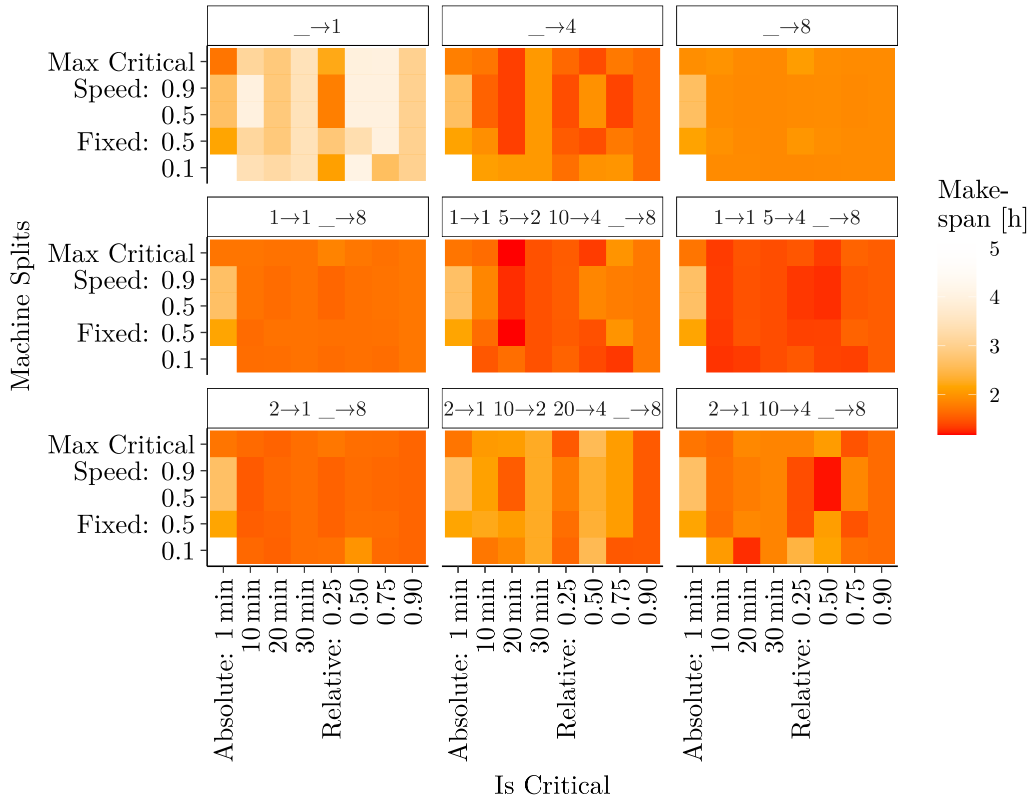

Figure 5 Solution quality (on preliminary data) for the three different parameters of our heuristic. The x-axis shows the criterion for a critical session, the y-axis how the available machines are split for critical/uncritical sessions, and the group label shows the threads parameter: The arrow notation maps run times less than the specified number of minutes on the left hand side to the number of threads on the right hand side. Darker is better (shorter makespan).

Figure 5 shows a comparison of the resulting makespan for an AFP build, depending on concrete values for the parameters explained above. The thread distribution is clearly the most important parameter for this build, and best results are generally achieved with 1 thread up to 1 min, then 4 threads up to 5 min, and 8 threads above that. Which sessions should be considered critical depends on the other parameters, but an absolute path length of 10 min to 20 min works well. The machine split only plays a minor role most of the time.

For the final heuristic, we additionally exploit that the number of sessions that may be scheduled on each host is typically much smaller than the number of CPU cores available (due to memory requirements), so we construct a parallel SGS where sessions are scheduled with a larger amount of threads than the minimum if there are unused cores. Additionally, if there is more capacity than sessions can use in parallel, we immediately schedule all sessions in their optimal mode. Finally, we sweep over the parameter space to find the best schedule, which can be done within a few seconds even for our largest problems.

4 Experimental Results

To empirically evaluate our schedules and strategies, we used three data sets with different hardware and configuration, as shown in Table 1: A preliminary data set used during development (cf. Section 3), one set collected on a heterogeneous cluster, and a final data set where we used the Isabelle2024 release on our current hardware.

Table 1 Experimental setup for data sets used in evaluation. CPU slices have access to 10 cores.

Data Set	Preliminary	Heterogeneous	Release
Measure Period	Nov/Dec 2023	Mar 2024	Jun 2024
Isabelle Revision	various	08b83f91a1b2	Isabelle2024
Builds/Data Points (Sessions)	38/19 263	41/30 510	36/30 203
Cluster Hardware:			
Xeon Silver 4316 Slice, 44 GB DDR4	10	10	–
Intel Core i9-12900K, 64 GB DDR5	4	4	4
Intel Core i9-13900K, 128 GB DDR5	–	–	4

We compare four scenarios in the following: Building the Isabelle distribution (medium-sized problem in Section 3.2), the AFP (without `slow`, corresponds to the large problem), the slow AFP (including `slow`, but not `very_slow`), or the full AFP. In doing so, we want to answer the following research questions empirically:

- RQ1: Which timings do we achieve, and when should one use explicit schedules?
- RQ2: How good is our heuristic compared to the best solver-generated solution?
- RQ3: Do the actual builds follow the generated schedule closely?

RQ1. To answer our first question, we compare the resulting build timings for different scheduling modes of the heterogeneous cluster in Table 2a, and the final results for the release version on our current hardware in Table 2b. On our current (homogeneous) cluster, the

Table 2 Final build timings from heterogeneous and release data sets, with different build engines. The solver engine refers to a scheduled build with an initial schedule generated by CPLEX in 5 min.

(a) Elapsed time for heterogeneous data set, without benchmarks.

(b) Elapsed and CPU time for release data set, including benchmarks.

	(a) Classic	(a) Heuristic	(a) Solver	(b) Classic		(b) Heuristic	
	Elapsed	Elapsed	Elapsed	Elapsed	CPU	Elapsed	CPU
Distribution	0:16:23	0:09:16	0:09:29	0:08:10	2:49:51	0:09:12	3:58:49
AFP	1:00:25	0:43:11	0:56:12	0:42:16	37:22:37	0:35:40	64:57:11
Slow AFP	0:58:16	0:48:03	0:50:13	0:49:09	45:52:17	0:40:50	73:49:22
Full AFP	5:04:32	2:07:56	2:12:01	2:07:10	50:53:02	1:59:13	77:01:21

classic build mode performs best for shorter builds, finishing the distribution in 8 min 10 s. For longer builds, using the schedule generated by our heuristic is 7 min to 8 min faster than the classic build, finishing the AFP in 35 min 40 s, and 40 min 50 s including slow sessions. This corresponds to a factor of over 100 between CPU time and elapsed time, though the CPU time does increase with the amount of parallelism (since the parallelism consumes additional CPU time). These timings are all within our critical limits of 10 min for the distribution and 45 min for the AFP, which is a huge improvement. The difference between the strategies is much more pronounced on the large heterogeneous cluster, as important jobs can end up on a slow host if assigned blindly. Since this is the case for the classic build, finishing the full AFP (with very slow sessions) took more than 5 h, compared to about 2 h otherwise. Finally, starting with a solver-generated schedule (CPLEX in 5 min) resulted in slightly worse times compared to the heuristic, although the solver always generated a better initial schedule.

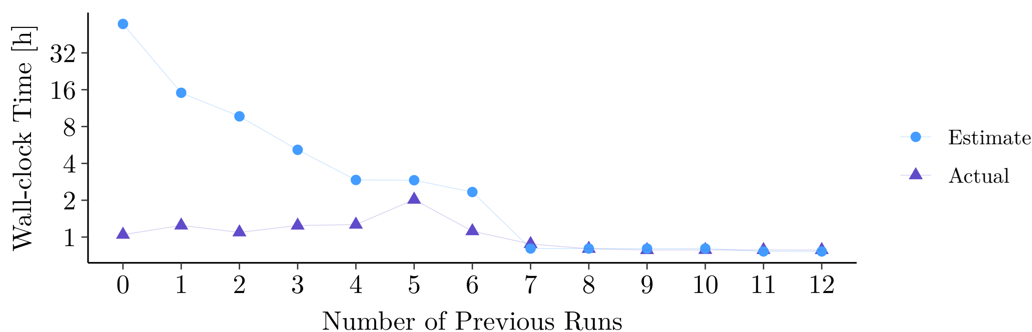

Figure 6 Estimate and actual build time (log-scaled) for consecutive builds of the slow AFP, using the release version and our current hardware (starting with an initially empty build log database).

Even though the heuristic was faster than the classic build for the AFP, it requires bootstrapping by collecting data from several builds until the estimation can be accurate. Figure 6 shows how much the actual time deviates from the initially predicated time when starting with an empty database. After 7–8 builds, the estimate was very close to the actual build time, and both remained stable consequently. While the bootstrapping could still be improved by generating schedules that explore more modes, it is sufficient in automated build environments – for one-shot builds, the classic build engine is better suited anyway.

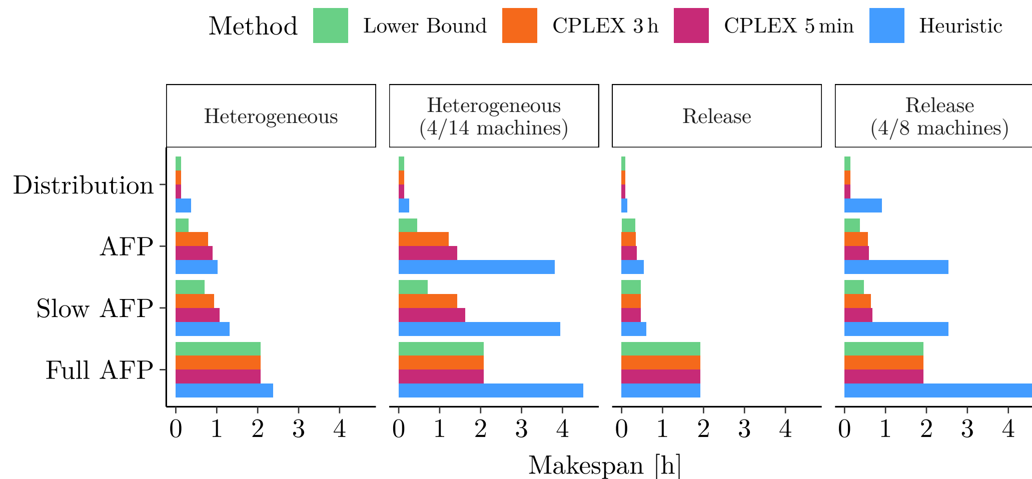

Figure 7 Comparison of estimated makespan (and best computed lower bound) in different scenarios on heterogeneous and release data sets, using the full cluster or only the four fastest hosts.

RQ2. Although the true build duration is often better than initially estimated due to the optimization every 5 min (by default), computing a good schedule is still critical. Hence, for our second research question we compare the makespan of the initially generated schedules. We consider the schedule found by our heuristic as well as the best schedule and lower bound we could generate via CPLEX (on a slow machine) in 5 min, and 3 h, respectively. Figure 7 shows the results. For both data sets, the schedules are reasonably close to each other on

the full cluster (in the expected ordering). There is still a significant gap to the best lower bound for the harder scheduling problems (AFP and slow AFP). The solution quality of the heuristic drops off dramatically when scheduling only for the four fastest machines, where our assumption is violated that plenty of slower hardware would be available.

RQ3. For our last research question, we want to assess whether generated schedules can be followed closely in the actual build. To that end, we analyze deviation of session run-time and session delay (i.e., the difference between planned and actual completion) for builds of the AFP with pre-generated schedules. All optimizations that would change the schedule during the build are disabled for this experiment. Figure 8 shows the result. Overall, the individual run-time deviations were quite low. However, with the solver-generated schedule a few sessions took several minutes longer, delaying a significant chunk of sessions such that the schedule finished 13 min late. The heuristically generated schedule had less (and mostly negative) deviations and accumulated not much delay, finishing within a minute of the estimated time. While the solver-generated schedule might have accumulated more delay as it is more tightly packed, the mean deviation was also three times as high compared to the heuristic, indicating that the run-time estimation was worse. A likely reason for this is that the data was collected by running builds with the heuristic strategy, so there is more accurate data for its frequently selected configurations. This also explains why using the solver resulted in worse timings even though the initially estimated makespan was shorter.

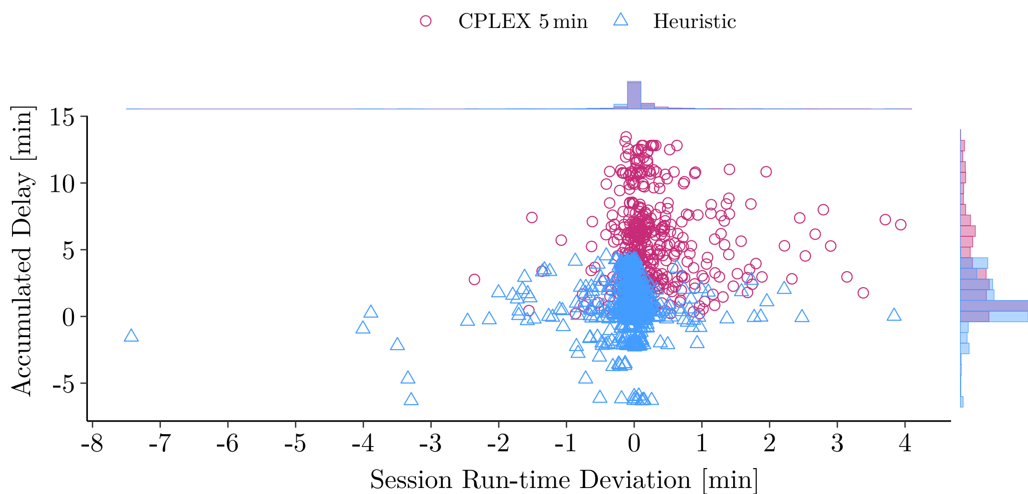

Figure 8 Accumulated delay and session run-time deviation for AFP builds with fixated schedules (release data), with histograms.

5 Conclusions

In this work, we introduced a distributed build system for Isabelle tailored to the requirements of the AFP. Based on the Isabelle/Scala framework in a standard repository clone, the system requires only SSH access, a fast local file system, and a single PostgreSQL database server – fully platform-independent. To utilize large heterogeneous clusters as well, we developed a sophisticated scheduling approach that incorporates build time estimation based

on previous timings. This involves finding a good solution to the computationally difficult multi-mode resource-constrained project scheduling problem in a few seconds, which remains NP-complete under major relaxations [10] (and does not admit constant-factor approximation in polynomial time [9]). After several rounds of tuning, the scheduling now works exceedingly well, and with our small clusters of 8–14 machines, we could achieve a speedup factor of over 100 vs. CPU time. This means that the AFP, which (through sheer size) had accumulated a multithreaded build time of 4 h–8 h on a single high-end server machine, can now be run in under 45 min again including slow sessions. Even with the `very_slow` group, the AFP now takes less than 2 h.

5.1 Related work

Any successful ITP system will eventually struggle with build times, as users are pushing more and more material into the libraries and archives. Coping with that requires technical efforts that usually remain unnoticed and unpublished. Subsequently we can only sketch important work for other proof assistants, with the focus on three kinds of parallelism: (1) multithreading within a single process, (2) multiprocessing on a single machine, (3) cluster computing on multiple machines.

The **Mizar** system and **Mizar Mathematical Library** have many similarities to Isabelle and AFP, concerning sizes of content, and the idea of a formally checked journal. The software technology is quite different, though: Mizar is implemented in Pascal and works like a multi-pass compiler on intermediate files. This does open possibilities for parallel processing: Urban [28] uses Perl and GNU `make -j` to manage the Mizar process for multiprocessing on a single machine. It might be worth revisiting that approach 10 years later, e.g. with the help of a distributed `make` tool. The resulting architecture would be unlike ours, where build data is stored in the database instead of a (shared) file-system.

Coq lacks a curated collection of applications, and it lacks a uniform build tool. Instead, every project needs to stand on its own for hosting and tooling. Reichel et al. [24] have scrutinized Coq build processes of projects found on public repositories: the motivation was to replay builds on intermediate versions, in order to collect data for automatic proof repair via machine-learning. As preliminary stage, huge efforts were required to “capture” Coq builds (they report that 68 % of commits in open-source Coq projects could not be built without problems). This lack of uniform tooling for Coq makes difficult to foresee eventual moves towards cluster computing.

The OCaml runtime (on which Coq runs) did not support parallel execution of concurrent code until recently, so Coq parallelization had to rely on parallel processes without shared-memory, e.g. via parallel make for independent files. More involved efforts use local multiprocessing with message passing for proofs that are irrelevant (opaque), notably Barras et al. [3]: an explicit directed acyclic graph of Coq proofs tells which parts can be independently checked in parallel forks of `coqc`. Coq proof state information is passed along with the opaque proof task. A distributed version of that will probably require homogeneous software setup, but this has not been pursued so far. Instead, *regression proof selection* was introduced by Celik et al. [6] and later refined and parallelized by Palmskog et al. [22], where the fine-grained dependency graph of theorems and definitions is analyzed so that only proofs affected by changes need to be re-checked. However, they do not consider changes in the logic, which would be very hard to track via dependency graph. For instance, if any tactic changes, only checking the selected proofs would not be sufficient. Despite those technological efforts, changes are often not checked well enough.

The **Lean** prover [8] and its companion library **Mathlib** are closer to Isabelle and AFP than Coq, but Lean makes every effort to do everything in just one language. Consequently, the "Lean Make" or **Lake** tool has been implemented in the monadic functional programming language of Lean. Lake is a sophisticated build and package manager for Lean projects, and is part of the main code base of Lean 4.[15] The corresponding Mathlib 4[16] is organized as a Lake project: its current source size is 1.4 million lines or 62 MiB (the AFP has 4.4 million lines or 216 MiB). Proof-level parallelism is part of regular Lean, with built-in multithreading. Further scaling in Lake is unclear from skimming through the documentation: we did not find dedicated papers to explain its concepts.

In practice, users of Lean and Lake usually rely on cached object files for theories rather than building them afresh, and the build time of the continuous integration for isolated changes in Mathlib is usually less than 1 h. We could run the whole of Mathlib 4 in 28 min on one of our fastest machines, plus another 7 min for add-on tests (which in Isabelle/AFP would be part of regular sessions). As Mathlib grows further towards the current size of AFP, it will be interesting to see which improvements on Lean build management will emerge.

5.2 Future Work

We can imagine many directions for further improvement of the Isabelle build system. Some alternative solutions have already been sketched in the introduction: notably ad hoc **reorganization of session structure**, similar to existing options `isabelle jedit -A -R`. In fact, the Prover IDE opens many further questions of working with **quasi-interactive builds**: the user could repair failures incrementally in the IDE, while the build keeps running, and changed sources would be uploaded to the build database without stopping sessions that are unaffected. This ultimately means combining ideas from the PIDE editor model [31] with the build tool, to scale-up editing to the AFP, as was postulated naïvely in 2014 [32].

Revisiting the build schedule problem, we could support **near-optimal schedules** as follows. The heuristic developed above can find appropriate schedules in a very short time, but they are usually not optimal. In contrast, a state-of-the-art CP-solver can yield schedules with tangibly shorter makespan, but finding optimal solutions often takes much longer than our target of 45 min for the whole build. We could do the scheduling occasionally (e.g. once per week), and re-use the result for subsequent versions of the sources. With an optimal solution generated offline, and only small ongoing changes of the AFP, we can expect even better quantitative results.

Looking further at the big picture, an important question is how to use compute cluster resources efficiently within a typical research group, with several **independent builds** potentially by **multiple users**. This could mean to have different categories and priorities of builds, e.g. *responsive* ones for quasi-interactive maintenance via the Prover IDE front-end vs. more traditional *batch builds* for continuous integration scenarios (occasional updates by AFP authors). Our cluster resources could be maintained in a global database, and Isabelle/Scala build process would work with this information to re-arrange schedules on the spot. Ultimately, we would like to regain good reactivity for interactive work, and avoid disappointing "CI-clouds" where the queue-time is often longer than the actual build time.

[15] `https://github.com/leanprover/lean4/blob/master/src/lake/README.md`
[16] `https://github.com/leanprover-community/mathlib4`

References

1 Archive of Formal Proofs (AFP). ISSN 2150-914x. URL: `https://www.isa-afp.org/about`.

2 Francisco Ballestín. When it is worthwhile to work with the stochastic RCPSP?, 2007. `doi:10.1007/s10951-007-0012-1`.

3 Bruno Barras, Carst Tankink, and Enrico Tassi. Asynchronous processing of Coq documents: From the kernel up to the user interface. In *Interactive Theorem Proving (ITP 2015)*, volume 9236 of *LNCS*. Springer, 2015. `doi:10.1007/978-3-319-22102-1_4`.

4 J. Blazewicz, J. K. Lenstra, A. Kan, and H. G. Rinnooy. Scheduling subject to resource constraints: classification and complexity. *Discrete Applied Mathematics*, 5, 1983. `doi:10.1016/0166-218X(83)90012-4`.

5 Miquel Bofill, Jordi Coll, Josep Suy, and Mateu Villaret. Solving the Multi-Mode Resource-Constrained Project Scheduling Problem with SMT. In *International Conference on Tools with Artificial Intelligence (ICTAI 2016)*. IEEE, 2016. `doi:10.1109/ICTAI.2016.0045`.

6 Ahmet Celik, Karl Palmskog, and Milos Gligoric. ICoq: Regression proof selection for large-scale verification projects. In *Automated Software Engineering (ASE 2017)*. IEEE, 2017. `doi:10.1109/ASE.2017.8115630`.

7 Edward Wilson Davis. *An exact algorithm for the multiple constrained-resource project scheduling problem*. Yale University, 1968.

8 Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In *Conference on Automated Deduction (CADE 2015)*, volume 9195 of *LNCS*. Springer, 2015. `doi:10.1007/978-3-319-21401-6_26`.

9 Evgeny R. Gafarov, Alexander A. Lazarev, and Frank Werner. Approximability results for the resource-constrained project scheduling problem with a single type of resources. *Annals of Operations Research*, 213, 2014. `doi:10.1007/s10479-012-1106-5`.

10 M. R. Garey and D. S. Johnson. Complexity Results for Multiprocessor Scheduling under Resource Constraints. *SIAM Journal on Computing*, 4, 1975. `doi:10.1137/0204035`.

11 Martin Josef Geiger. A multi-threaded local search algorithm and computer implementation for the multi-mode, resource-constrained multi-project scheduling problem. *European Journal of Operational Research*, 256, 2017. `doi:10.1016/j.ejor.2016.07.024`.

12 Daniel Geiss. Optimal Build Strategies for Isabelle. Technical report, Technische Universität München, 2023.

13 Sönke Hartmann. Project Scheduling with Multiple Modes: A Genetic Algorithm. *Annals of Operations Research*, 102, 2001. `doi:10.1023/A:1010902015091`.

14 Mark D. Hill and Michael R. Marty. Amdahl's Law in the Multicore Era. *Computer*, 41, 2008. `doi:10.1109/MC.2008.209`.

15 Fabian Huch. Formal Entity Graphs as Complex Networks: Assessing Centrality Metrics of the Archive of Formal Proofs. In *Conference on Intelligent Computer Mathematics (CICM 2022)*. Springer, 2022. `doi:10.1007/978-3-031-16681-5_10`.

16 Fabian Huch and Vincent Bode. The Isabelle Community Benchmark. In *Practical Aspects of Automated Reasoning (PAAR 2022)*, volume Vol-3201. CEUR-WS, 2022. `doi:10.48550/arXiv.2209.13894`.

17 Morris A. Jette and Tim Wickberg. Architecture of the Slurm Workload Manager. In *Job Scheduling Strategies for Parallel Processing (JSSPP 2023)*, volume 14283 of *LNCS*. Springer, 2023. `doi:10.1007/978-3-031-43943-8_1`.

18 Brian Daniel Keller. *Models and methods for multiple resource constrained job scheduling under uncertainty*. PhD thesis, University of Arizona, 2009.

19 David Matthews and Makarius Wenzel. Efficient parallel programming in Poly/ML and Isabelle/ML. In *Declarative Aspects of Multicore Programming (DAMP 2010)*. ACM, 2010. `doi:10.1145/1708046.1708058`.

20 Mokhtar Riahi Mohamed. Development of a Distributed Build Platform for Isabelle. Technical report, Technische Universität München, 2022.

21 Tobias Nipkow and Cornelia Pusch. AVL Trees. *Archive of Formal Proofs*, 2004. , Formal proof development. URL: `https://isa-afp.org/entries/AVL-Trees.html`.

22 Karl Palmskog, Ahmet Celik, and Milos Gligoric. PiCoq: Parallel regression proving for large-scale verification projects. In *International Symposium on Software Testing and Analysis (ISSTA 2018)*, volume 12. ACM, 2018. `doi:10.1145/3213846.3213877`.

23 Robert Pellerin, Nathalie Perrier, and François Berthaut. A survey of hybrid metaheuristics for the resource-constrained project scheduling problem, 2020. `doi:10.1016/j.ejor.2019.01.063`.

24 Tom Reichel, R. Wesley Henderson, Andrew Touchet, Andrew Gardner, and Talia Ringer. Proof Repair Infrastructure for Supervised Models: Building a Large Proof Repair Dataset. In *Interactive Theorem Proving (ITP 2023)*, volume 268 of *LIPIcs*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. `doi:10.4230/LIPIcs.ITP.2023.26`.

25 Christoph Schwindt and Jürgen Zimmermann. *Handbook on project management and scheduling*, volume 1. Springer, 2015. `doi:10.1007/978-3-319-05443-8`.

26 Frederik Stork. *Stochastic Resource-Constrained Project Scheduling*. PhD thesis, Technische Universität Berlin, 2001.

27 Jeffery David Ullman. Polynomial complete scheduling problems. In *Proceedings of the Fourth ACM Symposium on Operating System Principles*, volume 7 of *SIGOPS*. ACM, 1973. `doi:10.1145/800009.808055`.

28 Josef Urban. Parallelizing Mizar. In *Trends in Contemporary Computer Science*. Bialystok University of Technology Publishing Office, 2014. `doi:arXiv:1206.0141v2`.

29 Makarius Wenzel. Parallel Proof Checking in Isabelle/Isar. In *Programming Languages for Mechanized Mathematics Systems (PLMMS 2009)*, 2009. URL: `https://files.sketis.net/papers/parallel-isabelle.pdf`.

30 Makarius Wenzel. Shared-Memory Multiprocessing for Interactive Theorem Proving. In *Interactive Theorem Proving (ITP 2013)*, LNCS. Springer, 2013. `doi:10.1007/978-3-642-39634-2_30`.

31 Makarius Wenzel. Asynchronous User Interaction and Tool Integration in Isabelle/PIDE. In *Interactive Theorem Proving (ITP 2014)*, LNCS. Springer, 2014. `doi:10.1007/978-3-319-08970-6_33`.

32 Makarius Wenzel. System description: Isabelle/jEdit in 2014. In *User Interfaces for Theorem Provers (UITP 2014)*, volume 167 of *EPTCS*, 2014. `doi:10.4204/EPTCS.167.10`.

33 Guidong Zhu, Jonathan F. Bard, and Gang Yu. A branch-and-cut procedure for the multimode resource-constrained project-scheduling problem. *Journal on Computing*, 18, 2006. `doi:10.1287/ijoc.1040.0121`.