Exploiting parallelism in the $\mathcal{ME}$ calculus*

Tianyi Liang and Cesare Tinelli

Department of Computer Science
The University of Iowa

Abstract

We present some parallelization techniques for the Model Evolution ($\mathcal{ME}$) calculus, an instantiation-based calculus that lifts the DPLL procedure to first-order clause logic. Specifically, we consider a restriction of $\mathcal{ME}$ to the EPR fragment of clause logic for which the calculus is a decision procedure. The main operations in $\mathcal{ME}$'s proof procedures, namely clause instantiation and candidate literal generation, offer opportunities for MapReduce-style parallelization. This term/clause-level parallelization is largely orthogonal to the sort of search-level parallelization performed by portfolio approaches. We describe a hybrid parallel proof procedure for the restricted calculus that exploits parallelism at both levels to synergistic effect. The calculus and the proof procedure have been implemented in a new solver for EPR formulas. Our initial experimental results show that our term/clause-level parallelization alone is effective in reducing runtime and can be combined with a portfolio-based approach to maximize performance.

1 Introduction

The $\mathcal{ME}$ calculus [3] is an instantiation-based calculus that lifts to first-order logic without equality the popular DPLL procedure for propositional logic [6]. Like DPLL, it works with formulas in clause form, maintains at all times a *candidate model* for the input clause set, and keeps modifying that model until it finds one that satisfies all input clauses, or it determines that the clause set has no models. The main difference with DPLL is that the input clauses need not be ground and the candidate model is a Herbrand structure, represented finitely by a set of literals, called a *context*, instead of simple truth assignment. The calculus combines DPLL-style rules, such as unit propagation and splitting, with unification operations that generate instances of input clauses potentially not satisfied by the current candidate model. Such instances provide literals that can be added to the current context to obtain a new candidate model incrementally from the old one as needed. Implementations of $\mathcal{ME}$ benefit from enhancements similar to those developed for CDCL SAT solvers—the modern descendant of the DPLL procedure—such as conflict driven backjumping, lemma learning, and so on.

Roughly speaking, and ignoring those enhancements, a typical proof procedure for $\mathcal{ME}$ is a backtracking procedure relying on the following main data structures: a *context* M, a set of literals; a clause set F; a set R of candidate *decision (or split) literals*. The procedure starts with F consisting of the input clauses, R empty, and $M = \{\neg v\}$ denoting a Herbrand structure in which every ground atom is false. Then, it repeatedly performs the following. By unifying clauses in F with literals in M, it generates new *propagation* literals and adds them to M.[1] Then it identifies instances of clauses in F that are possibly falsified (by the structure denoted) by the context M. A simple syntactic check is used to determine if the context is *unrepairable*, i.e., both M and any of its enlargements definitely falsify one of those instances. If the context

*Work partially supported by NSF award 1049674.

[1] Intuitively, these are literals all of whose ground instances *must* be satisfied from that point on; their addition prevents future extensions of M that would break this requirement.

is instead repairable, the proof procedure uses the generated clause instances to compute new possible decision literals, and adds them to the candidate set R. Finally, it picks a literal from R according to some selection heuristic, creates a new decision point, and adds the chosen literal or its complement to the context. When the current context is not repairable, the procedure backtracks to a previous decision point, if any, and replaces the corresponding decision literal l, and all literals added after that, by the complement of l. The process ends when no more instances of F are falsified by M, which means that F is satisfiable, or when M is unrepairable but there are no decision points to backtrack to, which indicates that F is unsatisfiable.

1.1 Parallelizing $\mathcal{ME}$

In sequential implementations of $\mathcal{ME}$'s proof procedure, computing propagation and decision literals takes a considerable portion of the runtime. This computation, however, presents several opportunities for large scale parallelization of some of the term-level and clause-level operations involved. The work presented here was motivated by the conjecture that such parallelization would be effective in reducing execution times.

The proof procedure's exploration of the search space is determined in large part by a heuristic selection of decision literals—the other main factor being the backtracking heuristics. Decision literal selection offers *its own* parallelization opportunities that can be exploited for instance with a portfolio-style approach using concurrent proof procedures with different selection heuristics. The success of portfolio approaches in CDCL SAT solvers suggests that $\mathcal{ME}$ proof procedures could benefit from them as well. However, in our case it was not obvious how term/clause-level parallelization might interact with a portfolio-style one.

To investigate these opportunities and their interactions we designed and implemented a parallel proof procedure for $\mathcal{ME}$. For simplicity, with started with a restriction of $\mathcal{ME}$ to *EPR clauses*, (universal) clauses whose literals may contain variables, constants but no function symbols. While an extension of this work to the whole clause logic fragment is left to future work the restriction to the EPR case is interesting in its own right because (*i*) $\mathcal{ME}$ yields (practical) decision procedure for the satisfiability EPR formulas and (*ii*) many interesting problems can be recast as EPR satisfiability problems [16, e.g.]. Our initial experimental results show that both term/clause-level and portfolio-style parallelizations produce significant speed ups for $\mathcal{ME}$. Furthermore, the two have largely orthogonal effects and so combine nicely to produce better runtime results than either of them alone.

1.2 Related Work

Parallel approaches in first-order theorem proving have been categorized at three different levels: term, clause and search level [5]. In term-level approaches parallelize operations such as term matching or unification; clause-level approaches parallelize operations such as deduction of new clauses or backward subsumption; search-leval approaches parallelize the exploration of the search space. Because first-order calculi can generate thousands of clauses, each with tens of literals, parallelism at term or clause level requires sophisticated data structure and scheduling algorithms. Usually, the overhead of thread scheduling overcomes the benefit of concurrency. To our knowledge, with one exception [10], there has been no new work on term/clause level parallelism in first-order theorem proving after a number discouraging attempts (see [5] again) done in the 1990s.

Search level parallelism has received most of the attention, especially in SAT and SMT solving, with two approaches: Guiding Path and Portfolio. The Guiding Path approach, introduced in PSATO [20], follows a Master/Slave architecture. A Master process initiates several

sequential sub-solvers with partial models which partition the whole search space into several disjoint parts. If any sub-solver gets a satisfiable model, the problem is satisfiable; otherwise, it is unsatisfiable if all fail. This concept was further extended with a job stealing heuristics in ySAT[8]. By encoding QBF into SAT, QMiraXT became the first published parallel QBF solver[14]. A similar QBF solver can be found in [15, 9].

In the portfolio approach, introduced in ManySAT[11], a master process initializes several sub-solvers with different heuristics and makes them compete, of the same search space. The approach requires the various heuristics to be diverse enough. Complementary heuristics often generate super-linear speedups. This approach was also described as using different random seeds in [4]. Due to its success of SAT, the portfolio concept was also lifted to SMT solvers, with similar levels of success [19].

Lemma sharing is also an important factor in parallel SAT solvers. The portfolio approach benefits not only from the competition between subsolvers, but also from their cooperation through the exchange of lemmas [11]. There are two main shortcomings in lemma sharing. First, the set of shared lemmas may grow too large and possibly contain lemmas that are irrelevant to most subsolvers. Second, lemma communication can be a major source of overhead. Some SAT solvers minimize these problems by sharing only unit lemmas [4].

MapReduce is a software framework for processing large sets of data in a distributed system [7]. Very generally speaking, its main idea is to divide a large but highly distributable problem into small parts that can be processed completely independently (map step), and then compute the final result by combining the processed parts (reduce step). The sort of term/clause level parallelization that we describe in this paper could be seen as example of MapReduce.

1.3 Technical Preliminaries

The version of the $\mathcal{ME}$ calculus we consider here works in the *EPR fragment* of first-order clause logic without equality, which is restricted to clauses with no function symbols of positive arity. We use two disjoint sets of *variables*: a set $\mathcal{X}$ of *universal variables* and a set $\mathcal{P}$ of *parametric variables*.[2] We also use two disjoint sets of *constants*: a set $\mathcal{A}$ of *input constants* and a set $\mathcal{SK}$ of *Skolem constants*, with $\mathcal{C} = \mathcal{A} \cup \mathcal{SK}$. A *term* is either a constant or a variable. *Atoms*, *literals* (denoted by $k, l, l_0, k_0, \ldots$) and *clauses* ($C, C_0, \ldots$) over the set of terms above are defined as usual. We write $\neg l$ to denote the complement of a literal l; $l_0 \vee l_1 \vee \cdots \vee l_n$ to denote a clause C modulo AC of $\vee$; $|C|$ to denote the number of literals in C; and $\Box$ to denote the empty clause. A *Skolemization* of a literal l, denoted by l^{sko} is any literal obtained by replacing each *universal* variable in l by a fresh Skolem constant. A *substitution* σ is an idempotent function from variables $\mathcal{X} \cup \mathcal{P}$ to terms $\mathcal{X} \cup \mathcal{P} \cup \mathcal{C}$ such that the set $\mathcal{D}om(\sigma) = \{z \in \mathcal{X} \cup \mathcal{P} \mid z\sigma \neq z\}$ is finite. Substitutions extend to terms and clauses as usual. We use the standard notions of unifier and most general unifier. The *join* $\sigma \bowtie \rho$ of two substitutions σ and ρ is the most general simultaneous unifier of the set $\{\{z, z\sigma\} \mid z \in \mathcal{D}om(\sigma)\} \cup \{\{z, z\rho\} \mid z \in \mathcal{D}om(\rho)\}$ when such a unifier exists—otherwise it is undefined. For notational convenience, we will treat the join operator as left associative. A substitution σ is *p-preserving* if its restriction to $\mathcal{P}$ is a bijection onto $\mathcal{P}$. It is a *p-renaming* if it is p-preserving and its restriction to $\mathcal{X}$ is a bijection onto $\mathcal{X}$. A literal l' is a *p-variant* of a literal l if $l\sigma = l'$ for some p-renaming σ. For any literals l_0, l_1, we write $l_0 \geq l_1$ if $l_0\sigma = l_1$ for some p-preserving substitution σ; we call l_1 a *p-instance* of l_0. If L is a set of literals, we write $L \geq l_1$ if $l_0 \geq l_1$ for some $l_0 \in L$. We denote respectively by $Par(l)$ and $Var(l)$ the set of all parametric and all universal variables occurring in literal l. A literal l is *ground* if $Var(l) = Par(l) = \emptyset$; *universal* if $Par(l) = \emptyset$; and *pure* if either $Var(l) = \emptyset$

[2] Parametric variables were called *parameters* in earlier papers on $\mathcal{ME}$.

or $Par(l) = \emptyset$, or both.

2 A transition system for the $\mathcal{ME}$ calculus

To make the paper more self contained, we provide in this section a more formal description of the variant of $\mathcal{ME}$ used in this work. We refer the reader to [3] for more details on $\mathcal{ME}$.

As with formal and abstract treatments of the DPLL procedure and its extensions to Satisfiability Modulo Theories [17, 13, e.g.], one can formalize general classes of proof procedures for $\mathcal{ME}$ in a way that makes it easy to model and analyze operational features like backtracking and learning. An $\mathcal{ME}$ proof procedure can be described abstractly as a transition system over *states* of the form unsat, a distinguished fail state, or the form $\langle \mathsf{M}, \mathsf{F}, \mathsf{R}, \mathsf{A} \rangle$ where F is a clause set, M is a *context*, R is a set of *remainders*, and A is a set of *propagation* literals (see below). We model generic $\mathcal{ME}$ proof procedures by means of a set of states of the kind above together with a binary *transition relation* over these states defined by means of *transition rules*. For a given state S, a transition rule precisely defines whether there is a transition from S by this rule and, if so, to which state S'. A proof procedure can then be abstracted by a *transition system*, a set of transition rules defined over states, together with a strategy to generate executions in the system. We introduce a basic transition system for $\mathcal{ME}$ in the following.

Contexts and context unifiers A context M is a finite sequence of *decision points* ($\bullet$), and pure literals. A literal of M is *decision literal* if it immediately follows a decision point. Every maximal decision-point-free subsequence M_i of a context $M = M_0 \bullet M_1 \bullet \cdots \bullet M_n$ is a *decision level* of M. When convenient, we will treat a context as a set. A literal l is *contradictory* with a context M, written $l \perp M$, if $l\sigma = \neg k\sigma$ for some p-preserving substitution σ and p-variant k of a literal in M. We write $l \not\perp M$ if l is not contradictory with M.

Definition 1. *Let M be a context and $C = l_0 \vee \ldots \vee l_{m-1} \vee l_m \vee \ldots \vee l_n$ be a clause with $0 \leq m \leq n$. A substitution σ is a context unifier of C against M with remainder $r = l_m\sigma \vee \ldots \vee l_n\sigma$ if the following hold for some fresh p-variants $k_0, k_1, \ldots, k_n$ of literals in M:*

1. *(i) σ is a simultaneous most general unifier of $\{\{k_0, \neg l_0\}, \ldots, \{k_n, \neg l_n\}\}$,*
2. *(ii) $Par(k_i)\sigma \subseteq \mathcal{P}$ for $i = 0 \ldots m-1$,*
3. *(iii) $Par(k_i)\sigma \not\subseteq \mathcal{P}$ for $i = m \ldots n$.*

The context unifier σ is also admissible if the literals in r are pure and do not share universal variables. We write $M|_\sigma C$ to denote the remainder of an admissible context unifier σ of C against M.

We observe that any context unifier can be turned into an admissible one by renaming selected universal variables to parametric ones.

A clause C *conflicts* with a context M *via* a context unifier σ, written $C \perp_\sigma M$, if σ is an admissible context unifier of C against M with an empty remainder. We write $C \perp M$ (resp., $C \not\perp M$) if $C \perp_\sigma M$ for some (resp., no) σ.

2.1 The transition rules

The transition rules of the system are listed in Figure 1. Each rule operates on a current state of the form $\langle \mathsf{M}, \mathsf{F}, \mathsf{R}, \mathsf{A} \rangle$, and modifies some of its components. The $\mathsf{A\text{-}Add}$ rule identifies a

$$\textsf{A-Add}\ \frac{C \vee l \in \mathsf{F} \quad C \perp_\sigma \mathsf{M} \quad Par(l\sigma) = \emptyset \quad \mathsf{A} \not\geq l\sigma}{\mathsf{A} := \{a \in \mathsf{A} \mid l\sigma \not\geq a\} \cup \{l\sigma\}}$$

$$\textsf{A-Remove}\ \frac{\mathsf{A} = \{l\} \cup A \quad l \perp \mathsf{M} \text{ or } \mathsf{M} \geq l}{\mathsf{A} := A} \qquad \textsf{R-Add}\ \frac{C \in \mathsf{F} \quad |C| > 1 \quad r = \mathsf{M}|_\sigma C \quad r \notin \mathsf{R}}{\mathsf{R} := \mathsf{R} \cup \{r\}}$$

$$\textsf{Assert}\ \frac{\mathsf{A} = \{l\} \cup A \quad l \not\perp \mathsf{M} \quad \mathsf{M} \not\geq l}{\mathsf{A} := A \quad \mathsf{M} := \mathsf{M}\, l} \qquad \textsf{Decide}\ \frac{\mathsf{R} = \{l \vee C\} \cup R \quad l \not\perp \mathsf{M} \quad \neg l^{\mathrm{sko}} \not\perp \mathsf{M} \quad \mathsf{A} = \emptyset}{\mathsf{R} := R \quad \mathsf{M} := \mathsf{M} \bullet l}$$

$$\textsf{Backjump}\ \frac{\mathsf{M} = M' \bullet l\, M'' \quad \mathsf{F} \models C \quad C \perp \mathsf{M} \quad C \not\perp M'}{\mathsf{M} := M'\, \neg l^{\mathrm{sko}} \quad \mathsf{A} := \emptyset} \qquad \textsf{Fail}\ \frac{\bullet \notin \mathsf{M} \quad C \in \mathsf{F} \quad C \perp \mathsf{M}}{\mathsf{unsat}}$$

Figure 1: Transition Rules

propagation (or *assert*) literal and adds it to the set A of propagation literals, while removing from A all p-instances of the new literal. The A-Remove rule removes from A any literal that has become contradictory with or a p-instance of the current context M. The R-Add rule generates a new remainder from a non-unit clause C and adds it to the set R of remainders. The Assert rule moves a propagation literal to the current context provided that the literal is neither contradictory with nor a p-instance of the context. The Decide rule selects a literal l from the available remainders in R and adds it as a decision literal to context, provided that neither l nor its Skolemized complement is contradictory with the context.

The Backjump rule removes one or more decision levels from the current context and replaces the oldest of the removed decision literals by its Skolemized complement. A *backjump clause* C entailed by the clause set is used to determine how far to backjump. In actual implementations this clause can be computed from an input clause that conflicts with the current context, by using a conflict resolution mechanism similar to the one used in CDCL SAT solvers. The Fail rule applies, producing the distinguished state unsat, if an input clause conflicts with the current context and the context contains no decision points (to backtrack to).

An actual implementation of the transition system would remove from the set R when backjumping all remainders computed using literals no longer in the context M. We do not model this here just to simplify the description of the rules and because keeping stale remainders in R does effect correctness. Also for simplicity, we hardcode into the rules the heuristics that choses to process all current propagation literals before applying Decide. This is unnecessary for correctness if Backjump is modified to remove from A propagation literals computed using context literals no longer in M after the backjump.

Although we will not show it here, any fair execution of the transition system above starting with a state where $\mathsf{M} = \{\neg v\}$ and all the other fields except F are empty terminates in the unsat state if and only if the clauses in F are jointly unsatisfiable.

3 A Sequential Proof Procedure

In this section, we describe in some detail a sequential proof procedure we have implemented for the transition system in the previous section. The procedure mimics closely the behavior of the Darwin theorem prover [1], a full implementation of $\mathcal{ME}$ for clause logic without equality, when run on an EPR problem. The procedure uses these data structures for the main components of a state: a context, a priority queue of propagation literals, a priority queue of remainders, a

set of clauses. Its main loop consists of the following steps:

1. **Propagation.** The highest-priority literal from the propagation queue, if any, is removed from the queue. If it is neither p-subsumed by nor contradictory with the context, it is added to the context; otherwise, it is discarded and this step repeats.

2. **Decision.** If the context was unchanged by the previous step, the remainder with the highest-priority literal among all the remainder literals is removed from the remainder queue, if any. Then, that literal is added to the context if it is neither p-subsumed by nor contradictory with the context. Otherwise, the remainder is discarded and this step is repeated.

3. **Unit Context Unifier Calculation.** If the context was extended in the previous steps, the newly added context literal k is unified with the complement of each literal l in each clause in the clause set. Each most general unifier computed this way, is stored as a *unit context unifier* for l.

4. **Remainder Generation.** The procedure identifies all sets $\{\sigma_1, \ldots, \sigma_n\}$ of substitutions where (1) for $i = 1, \ldots, n$, σ_i is a unit context unifier for literal l_i in some clause $l_1 \vee \cdots \vee l_n$, and (2) one (or more) of the σ_i's was newly computed in the previous step. For each of these sets $\{\sigma_1, \ldots, \sigma_n\}$, the substitution $\sigma_1 \bowtie \cdots \bowtie \sigma_n$, when defined, is a context unifier of the corresponding clause $l_1 \vee \cdots \vee l_n$. The procedure computes all such context unifiers, makes them admissible, and adds their remainder to the remainder queue.[3] The process is interrupted, however, as soon as a context unifier with an empty remainder is computed. In that case, the procedure moves immediately to Step 6.

5. **Propagation Literal Generation.** A process similar to the one in the previous substep is used to generate propagation literals and add them to the propagation queue.[4]

6. **Backjumping.** After an analysis of the conflict represented by the empty remainder computed in the previous step, the procedures identifies a previous decision level d to backtrack to. If d is the top level, the procedure ends with an "unsatisfiable" result. Otherwise, it clears the propagation queue, undoes all additions to the context and to the remainder queue from that level on, adds to the context (at decision level $d-1$) the Skolemized complement l^{sko} of the decision literal l of d, and resumes the main loop from Step 1.

If neither of the first two steps is able to add literals to the context, the main loop aborts and the procedure terminates with success: the context denotes a model of the clause set.

Selection Heuristics In our current implementation, the priority function used in the propagation queue is determined by a ranking of literals where propositional literals[5] are preferred over (i.e., have a higher rank than) universal ones, which in turn are preferred over parametric literals. Among universal literals, those with more variables are preferred. Among parametric literals, those with less parameters are preferred. After the criteria above, literals introduced in the propagation queue in an earlier decision level are preferred. Any ties after that are broken in an arbitrary, but fixed, way. Something similar is done for the remainder queue. There, the

[3] We ignore here the enhancement that considers only *productive* context unifiers (see [3] for details).
[4] In reality, this step and Step 4 are interleaved. We present them here as sequential for simplicity.
[5] That is, literals with a 0-arity predicate symbol.

same literal ranking as the one in the propagation queue is used first locally in each remainder, to select a literal, and then globally to chose among those selected literals.

For comparison, we also implemented two random selection heuristics: controlled random and totally random. The controlled random heuristics modifies the priority functions above by breaking the final ties randomly (as opposed to a fixed way). With the totally random heuristics no priority function is actually used. The dequeue operation in both queues simply picks an element from the queue at random.

4 A Parallel Proof Procedure

The sequential proof procedure highlighted above presents several opportunities for parallelization. We have focused on parallelizing two main aspects: the computation of context unifiers and the exploration of the search space.

Term/Clause-level Parallelization At each iteration of the sequential proof procedure most of the computation is spent in the generation of context unifiers. Since the individual unit context unifiers computed in Step 3 are completely independent from each other, they can all be computed in parallel, MapReduce style. The computation of the context unifiers done in Step 4 (by joining unit context unifiers) can be parallelized in a similar way for each clause in the clause database.

Search-level Parallelization As in DPLL, in $\mathcal{ME}$ the exploration of the search space is driven by the selection of the next decision literal. In fact, since empty remainders trigger a backjump as soon as they are generated, the exploration is also driven by the order in which propagation literals are chosen. Our experiments on candidate selection confirm that, again as in DPLL, the selection heuristics can have a significant impact on performance for some problems. To account for that we also implemented a portfolio-based approach [11] where the input problem is given to several subsolvers running independently from each other. The subsolvers differ only for the candidate selection heuristics they use for the propagation and the remainder queues. They run completely independently except that they are all stopped once one of them proves or disproves the input problem.

4.1 General Architecture

Our parallel proof procedure follows the actor model of computation, and relies on a small number of actor classes. All actors communicate asynchronously via message queues and run in their own computation thread. The actor model considerably simplifies the implementation of parallel systems with respect to the shared-memory model. It also minimizes synchronization needs, leading to less overhead and greater scalability with the increase of computational resources.

Main Actors The main actors in our architecture, sketched in Figure 2, are one Main Solver, one Context Manager, one or more Clause Managers, one Unification Pool, and one or more Candidate Generators. Roughly, the **Main Solver** parses the input formulas, sets ups a number of data structures, creates the other actors, and then passes control to the Context Manager. The **Context Manager** manages the context data structure, and is the one effectively applying the rules of the calculus by selecting literals to add to the context, analyzing conflicts caused by empty remainders, deciding where to backjump, and shrinking the context accordingly. A

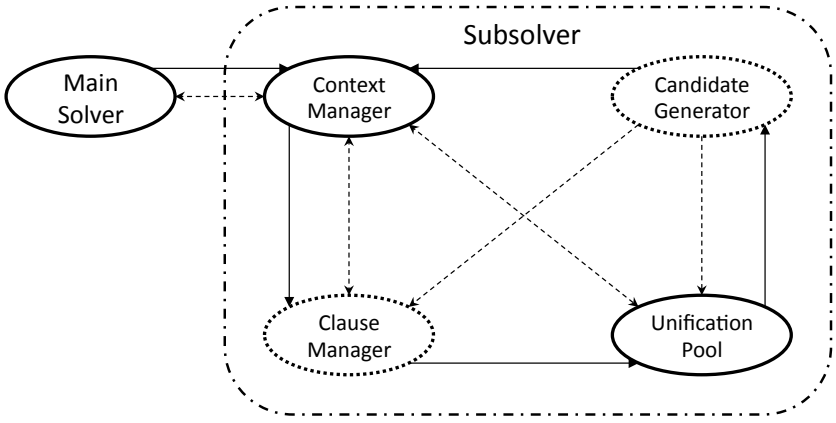

Figure 2: Architecture of the parallel EPR Solver. The solid lines represent data flow, the dash lines represent control flow. The dotted ovals stand for multiple actors.

Clause Manager is responsible for one or more input clauses and for the computation of unit context unifiers for them. In an ideal situation, with the system running on a unlimited number of parallel computational (e.g., cores), we would have one clause manager per clause. In reality, the clause set is partitioned so that each Clause Manager manages several clauses sequentially, to reduce the number of threads running on the same core. The **Unification Pool** is in charge of computing context unifiers, by merging unit context unifiers received from the Clause Manager, as well as computing propagation literals or remainders from those context unifiers. It performs its functions by delegating them to one or more **Candidate Generators** it manages. It is essentially a scheduler, creating and assigning unification tasks to the Candidate Generators as they became available.

The portfolio extension adds several subsolver actors below the Main Solver, each relying on its own Context Manager, Clause Managers, Unification Pool, and Candidate Generators according to the architecture above.

4.2 Synchronization

The parallel proof procedure has been designed to minimize the need for the various actors to synchronize with one another while also minimizing runtime differences between identical runs when no randomized selections heuristics are used. The main synchronization points are discussed below.

Synchronization between sub-solvers. Once started, the subsolvers are completely independent from the main solver and each other. Synchronization with the main solver occurs only once a subsolver solves the input problem, i.e., determines if it is satisfiable or not, at which point the main solver terminates all subsolvers, outputs a response, and quits.

Synchronization before candidate selection. Before selecting a candidate as a new context literal, the Context Manager waits for all context unifier computations in each Candidate Generator to terminate, to make sure that all possible candidates are available for selection in the propagation and the remainder queue.

	Seq	S1C1U1	S1C3U20	S4C1U1	S4C3U20	CR	TR
EPT Solved	161	175	199	197	222	213	186
EPS Solved	125	125	128	129	134	135	128
Total Solved	286	300	327	326	356	348	314
Median runtime (s)	1.48	1.82	1.61	1.52	1.81	1.77	1.43
Average runtime (s)	11.83	19.93	27.29	27.46	27.23	32.62	18.74
Med speedup		0.93	1.27	1.21	1.38	1.35	1.35
Avg speedup		1.29	1.67	1.88	2.34	2.16	1.96
Max speedup		11.99	14.27	45.50	43.62	42.36	39.26
Med speedup ($\geq$ 5s)		1.75	1.93	2.16	2.93	2.61	2.48
Avg speedup ($\geq$ 5s)		2.64	3.21	4.18	5.59	5.10	4.25

Figure 3: Results for several solver configurations over the EPS and EPT problems of TPTP. The columns represent the tested solver configurations: the sequential solver (Seq); the parallel solver with i subsolvers, j clause managers, and k candidate generators in the unification pool (SiCjUk); the parallel solver with 4 subsolvers each using the controlled random heuristics (CR) or the totally random heuristics (TR).

Synchronization when backjumping. When an empty remainder is generated, the Context Manager is immediately notified. In turn, it immediately instructs all Clause Managers and the Unification Pool to abort their computation, and waits for an acknowledgment from them before backjumping and sending a new context literal to the Clause Managers. Waiting for an acknowledgment is needed because the actor model does not guarantee that messages received by an actor in the same order they were sent. Note that an actor's computation cannot be interrupted by another actor. Since Candidate Generators take a while to complete their tasks, they are not notified about a backjump and just let run to completion. However, they are required to time-stamp, with the number of backjumps so far, the candidates they compute. This way, such candidates can be discarded if they arrive to the Context Manager when they are no longer current because of a later backjump.

In our current implementation of the parallel proof procedure several parallelization parameters such as the number of subsolvers, clause managers and candidate generators, are controlled by user-configurable options. Since we focus on parallel strategies, we did not implement at this time any preprocessing simplifications on the input clause set.

5 Experimental Evaluation

We evaluated the performance of our implementation of the sequential and the parallel proof procedures described above against the EPR benchmarks of the TPTP library [18] which are divided into EPS problems (satisfiable clause sets) and EPT problems (unsatisfiable clause sets).[6] Since our current solvers do not include inference rules for equality yet, we focused on the 483 clausal problems without equality. Of those, 318 are EPT problems and 165 are EPS problems.

All tests were run on a computer with two 12-core AMD Opteron 6172 processors and 32Gb of memory, and running under Ubuntu 11.10. The solvers were developed in Scala, a language

[6]Detailed results together with the benchmark problems and our implementation can be found at `http://www.cs.uiowa.edu/~tiliang/paar12/`.

based on the Java Virtual Machine. We used OpenJDK 64-Bit Version 1.6.0 as the JVM engine. All experiments were run using the JVM option "-XX:+UseCompressedOops". Since CPU time is not very meaningful when measuring the *runtime* performance of parallel programs, we used wall clock time[7], with a timeout limit of 300 seconds.

5.1 Results

Our experimental results are summarized in Figure 5 for the baseline sequential solver and for several configurations of the parallel solver. The results refer to a single run of each configuration. Configuration S1C1U1 of the parallel solver uses a single subsolver, one clause group (managing all clauses), and one candidate generator (again for all clauses) in the unification pool. It is parallel only in that its various actors run concurrently. In contrast, S1C3U20 is more properly MapReduce-style for having 3 clause managers and 20 candidate generators. Configuration S4C1U1 uses 4 subsolvers which differ from each other only in the fixed way they break the final ties in their selection heuristics. Each subsolver has just one clause manager and one candidate generator, making this configuration essentially a pure portfolio-style solver. Configuration S4C3U20 is a hybrid resulting from the combination of the previous two. Configuration CR (resp. TR) is like S4C3U20 except that the subsolvers use the controlled (resp., totally) random selection heuristic, each with a different seed.

For each configuration, averages and median runtime values are computed over the problems solved by that configuration. Speedup factors are with respect to the runtimes of the Seq configuration, and computed for each problem solved by Seq. The rows marked with ($\geq$ 5s) remove from consideration *easy problems*, defined as problems solved by Seq in less than 5s (216/483). Focusing on those rows for the parallel configurations is instructive because for easy problems the overhead caused by thread initialization and scheduling generally cancels out most of the performance improvement due to parallel execution—in fact, it actually increases overall runtimes for most problems solved by Seq within 1 second.

5.2 Analysis

As we conjectured, the sequential solver exhibits the lowest success rate, measured as the percentage of problems solved within the time limit, solving only 59% of the 483 problems. All the parallel configurations solve a superset of the problems solved by Seq, with an increased success rate that goes from 62% for the minimally parallel S1C1U1 to 75% for the hybrid S4C3U20. The improvement provided by S1C1U1 shows that just computing unit context unifiers in parallel with joining such unifiers to generate candidates is already advantageous.

We experimented with a number of additional configurations (not reported here) differing from S1C1U1 only in the number of clause managers and the size of the unification pool. Our main conjecture was that increasing those parameters would lead to a greater success rate because of the relative independence of the computations performed by each clause group and each candidate generator. Our general findings confirm that conjecture, with a sweet spot provided by S1C3U20. Adding more clause managers or more candidate generators usually leads to a degraded performance, possibly because then, as we have verified, the number of threads significantly exceeds the number of physical cores. These experiments, together with additional ones on machines with fewer cores, show that the success rate of the S1C*U* configurations increases lineraly with the number of cores. This strongly suggest that our solver will scale up

[7] Measured with the same utilities used at SMT-COMP 2011 (`http://www.smtcomp.org/2011/`).

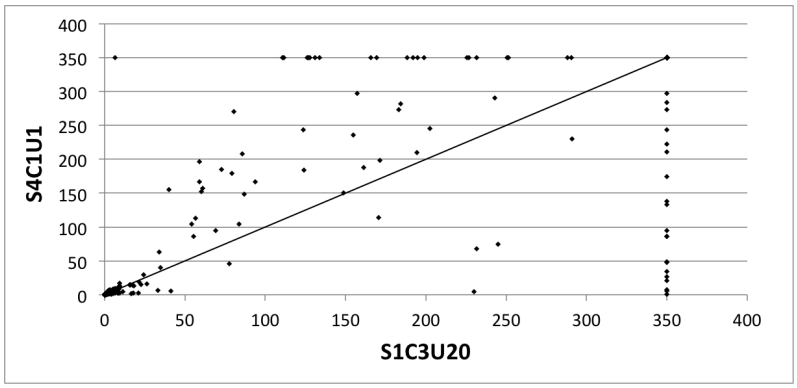

Figure 4: Comparative runtime performance of a MapReduce (S1C3U20) and a portfolio strategy (S4C1U1). Times are in seconds.

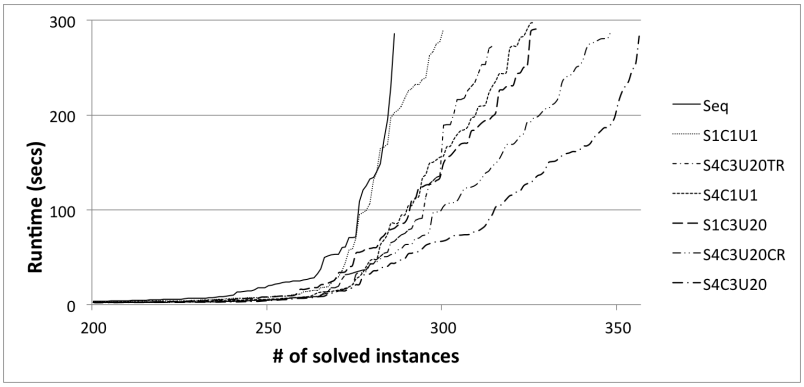

Figure 5: Runtime performance of all configurations.

well, within the limits of Amdahl's law [12], as processors with more and more cores become available.

The very simple portfolio approach implemented by configuration S4C1U1 impressively achieves almost the same success rate (67%) as that of the more sophisticated MapReduce configuration S1C3U20 (68%). Its superiority to the sequential solver is consistent with similar findings by others on parallelizing SAT and SMT solvers. What is interesting in our case is that the MapReduce and the portfolio strategies are complementary to a certain extent, as shown in the scatter plot of Figure 4. In particular, each solves about 20 problems that the other cannot solve. The same plot also shows that for problems solved by both, S1C3U20 is superior to S4C1U1 in terms of runtimes. The overall superiority of S1C3U20 is confirmed by a Wilcoxon rank-sum test on the whole set, which allows us to accept the alternative hypothesis that S1C3U20 is faster than S4C1U1 with a p-value smaller than 0.001.

We obtained similar results also for configuration S1C3U10 (not shown), which creates about as many threads as S4C1U1. It is possible that the subsolvers of S4C1U1 are not diverse enough to fully exploit the advantages of a portfolio approach. However, additional controlled experiments with one subsolver using random selection heuristics, for greater variability, did not improve the overall performance of S4C1U1.

The complementarity of S1C3U10 and S4C1U1 suggests that combining them could have a synergistic effect on performance. This is confirmed by the results obtained by S4C3U20 which can solve not only all the problems solved by S1C3U10 and by S4C1U1 individually, but also a few more. In fact, S4C3U20 is also faster than any other configuration on the problems solved by both. This is reflected by the average speed up factors in Figure 3. As with the pure portfolio strategies, adding randomization in the selection process, both in a controlled or uncontrolled fashion, did not improve the performance of S4C3U20 further, actually resulting in worse performance. The superiority S4C3U20 in general to all other configurations listed in Figure 3 is clearly shown in the chart of Figure 5, indicating again that clause-level and search-level parallelism can be combined to great effect in $\mathcal{ME}$.

Finally, we observe that the best average speedup factor we achieved for hard problems (5+) seems to be low with respect to the number of cores used. On the one hand, this contrasts with results achieved by the best portfolio SAT solvers [11, e.g.] whose average speedups versus a sequential version can be superlinear in the number of cores. On the other hand, our portfolio implementation is fairly unsophisticated yet and lacks crucial features such as lemma sharing. Also, EPR satisfiability is a much harder problem that propositional satisfiability (NEXPTIME vs. NP) and so it is possibly correspondingly harder to parallelize. So, while our results could be considered a good first step, more work and experimental evaluations are still needed.

6 Conclusion and Future Work

We have described concurrent proof procedures for the $\mathcal{ME}$ calculus that rely on term/clause-level as well as search-level parallelism. Our experiments provide initial evidence that the former is effective in reducing runtimes in instantiation-based theorem proving when using MapReduce-style approaches which minimize the interactions between concurrent threads. Our results show that, in addition to improving performance by themselves, such approaches also combine synergistically with the traditional portfolio approaches.

We are working on an enhancement of the proof procedure with lemma learning, and lemma sharing in the portfolio case. Lemma learning in $\mathcal{ME}$ is similar to lemma learning in SAT solvers but has its own distinct features for exploring at the first-order, as opposed to the propositional, level [2]. Further work will involve conducting further experimental evaluations on the effectiveness of lemma sharing between subsolvers in our parallel implementation.

References

[1] P. Baumgartner, A. Fuchs, and C. Tinelli. Implementing the model evolution calculus. *International Journal of Artificial Intelligence Tools*, 15(1):21–52, Feb. 2006.

[2] P. Baumgartner, A. Fuchs, and C. Tinelli. Lemma learning in the model evolution calculus. In M. Hermann and A. Voronkov, editors, *Proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'06), Phnom Penh, Cambodia*, volume 4246 of *Lecture Notes in Computer Science*, pages 572–586. Springer, 2006.

[3] P. Baumgartner and C. Tinelli. The model evolution calculus as a first-order DPLL method. *Artificial Intelligence*, 172:591–632, 2008.

[4] A. Biere. Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. FMV Reports Series Technical Report 10/1, Institute for Formal Models and Verification, Johannes Kepler University, 69, 4040 Linz, Austria, August 2010.

[5] M. P. Bonacina. A taxonomy of parallel strategies for deduction. *Annuals of Mathematics and Artificial Intelligence*, 29(1-4):223–257, 2000.

[6] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5:394–397, July 1962.

[7] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, Jan. 2008.

[8] Y. Feldman. Parallel multithreaded satisfiability solver: Design and implementation, 2005.

[9] M. Fränzle, C. Herde, T. Teige, S. Ratschan, and T. Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *Journal on Satisfiability, Boolean Modeling and Computation*, 1:209–236, 2007.

[10] J. Gaillourdet, T. Hillenbrand, B. Löchner, and H. Spies. The new Waldmeister loop at work. In F. Baader, editor, *Proceedings of the 19th International Conference on Automated Deduction, CADE-19*, volume 2741 of *Lecture Notes in Artificial Intelligence*, pages 317–321. Springer, 2003.

[11] Y. Hamadi and L. Sais. ManySAT: a parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6, 2009.

[12] M. Hill and M. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, July 2008.

[13] S. Krstić and A. Goel. Architecting solvers for SAT modulo theories: Nelson-Oppen with DPLL. In B. Konev and F. Wolter, editors, *Proceeding of the Symposium on Frontiers of Combining Systems (Liverpool, England)*, volume 4720 of *Lecture Notes in Computer Science*, pages 1–27. Springer, 2007.

[14] M. Lewis, T. Schubert, and B. Becker. QMiraXT – A Multithreaded QBF Solver. In *GI/IT-G/GMM Workshop "Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen"*, 2009.

[15] B. D. Mota, P. Nicolas, and I. Stéphan. A new parallel architecture for QBF tools. In *HPCS'10*, pages 324–330, 2010.

[16] J. A. Navarro Peréz. *Encoding and Solving Problems in Effectively Propositional Logic*. PhD thesis, The University of Manchester, 2007.

[17] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Abstract DPLL and abstract DPLL modulo theories. In F. Baader and A. Voronkov, editors, *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'04), Montevideo, Uruguay*, volume 3452 of *Lecture Notes in Computer Science*, pages 36–50. Springer, 2005.

[18] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[19] C. M. Wintersteiger, Y. Hamadi, and L. Moura. A concurrent portfolio approach to SMT solving. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 715–720, Berlin, Heidelberg, 2009. Springer-Verlag.

[20] H. Zhang, M. P. Bonacina, M. Paola, Bonacina, and J. Hsiang. PSATO: a distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21:543–560, 1996.