Exploiting Parallelism in Interactive Theorem Provers

Roderick Moten

Colgate University, Hamilton, NY 13224 USA
rod@cs.colgate.edu

Abstract. This paper reports on the implementation and analysis of the *MP refiner*, the first parallel interactive theorem prover. The MP refiner is a shared memory multi-processor implementation of the inference engine of *Nuprl*. The inference engine of Nuprl is called the refiner. The MP refiner is a collection of threads operating as sequential refiners running on separate processors. Concurrent tactics exploit parallelism by spawning tactics to be evaluated by other refiner threads simultaneously. Tests conducted with the MP refiner running on a four processor Sparc shared–memory multi-processor reveal that parallelism at the inference rule level can significantly decrease the elapsed time of constructing proofs interactively.

1 Introduction

An interactive theorem prover is a computer program that employs automated deduction to construct proofs with the aid of a user. Many interactive theorem provers require users to supply programs, called *tactics*, to carry out inference. Tactics usually carry out multiple steps of inference and use heuristics to determine the inferences to employ. A tactic may construct an entire proof or a portion of a proof. Tactics were first used with the LCF interactive proof system [9]. Most interactive theorem provers in existence today use tactics [5, 8, 6, 20, 7, 11].

We believe that parallelism will improve the performance of interactive theorem provers. Previous research has shown employing parallelism in automatic theorem provers has lead to significant speedups. For instance, the parallel version of Otter [14], Roo [13, 24], has near linear speedups for many of the tests reported on in [13]. The tests were conducted on a shared–memory multi-processor with 24 processors using the benchmark of theorem proving problems in [30]. Parallel versions of SETHEO [12], PARTHEO [22] and SPTHEO [26, 25], out performed SETHEO. In addition, 75% of the tests with the Parthenon parallel theorem prover [4] based on Warren's SRI model of OR-parallelism of Prolog [29] had a linear speedup over the sequential version of the prover. The speedups were obtained using an Encore Multimax with 16 processors.

Static Partitioning with slackness [27] characterizes the model of parallelism employed in the above parallel theorem provers and several other parallel theorem provers [28, 23]. Static partitioning with slackness divides the search process into three sequential phases. During the first phase, an initial segment of the

search space is explored to obtain tasks that can be computed simultaneously using OR or independent AND-parallelism. During the second phase, the tasks are distributed amongst parallel processors. During the third phase, each of the parallel processors works on the tasks independently. After a processor completes a task, it reports its results to a master process that is responsible for global termination.

Interactive theorem provers that use tactics may also employ static partitioning with slackness because they construct proofs using AND-search and OR-search. AND-search and OR-search are obtained from constructing proofs using tactics created with `THEN` and `ORELSE`, respectively. Using a tactical, a new tactic can be created from one or more existing tactics. For example, given tactics t and t', the tactic (t `THEN` t') constructs a proof of the goal g by first applying t to g to produce subgoals $g_1, \ldots, g_n$. Afterwards, t' is used to construct proofs of each g_i. If t' constructs proofs of each g_i, then they are used to construct a proof of g. The proof of g obtained with the tactic (t `ORELSE` t') is the proof of g obtained with t. If t fails to produce a proof of g, then the proof of g obtained with (t `ORELSE` t') is the proof of g obtained with t'. `THEN` and `ORELSE` were first used in LCF and are implemented in many interactive theorem provers.

1.1 Exploiting Parallelism in Nuprl

To investigate the effects of parallelism in interactive theorem provers we developed the *MP refiner*. The MP refiner is a shared memory multi-processor implementation of the inference mechanism of the Nuprl Proof Development System [5]. The MP refiner provides concurrent tacticals for users to construct tactics that employ AND–parallelism and OR–parallelism. We call tactics that use parallelism *concurrent tactics*. We call the process of developing proofs using concurrent tactics *concurrent refinement*. AND–parallelism is obtained by creating tactics using the new tactical `PTHEN`. `PTHEN` is the concurrent version of the tactical `THEN`. Given tactics t and t', refining a goal g with the tactical (t `PTHEN` t') will first refine g using t to produce the subgoals $g_1, \ldots, g_n$. Then t' will be used to refine each g_i simultaneously. AND-search is being performed because proofs of each g_i are needed to construct a proof for g. OR-parallelism is obtained by creating tactics using the new tactical `PORELSEL` (pronounced p-or-else-l). `PORELSEL` is the concurrent version of `ORELSE`. Refining g with the tactic (t_0 `PORELSEL` $[t_1, \ldots t_n]$) results in refining g with each t_i simultaneously. One of the proofs is chosen deterministicly as the proof created by (t_0 `PORELSEL` $[t_1, \ldots t_n]$).

The MP refiner is implemented in Standard ML of NJ (SML/NJ) version 1.09.07 with **MP** [16]. **MP** extends the runtime system of (SML/NJ) to provide parallelism by mapping ML threads to operating system threads that execute concurrently on separate processors. The runtime system manages memory for ML programs, provides system services, such as I/O and file system routines, manages the ML state, and handles signals and traps caused during the execution of ML code [2]. We call the runtime system using **MP** the *MP runtime system*. Although the MP runtime system deals with operating system and hardware

dependent features, it is extremely portable [16]. We use the port for Sparc multi-processors.

1.2 Outline

In Section 2, we give a brief introduction to Standard ML and **MP**. We give an overview of tactics in Nuprl and the refiner in Section 3. In Section 4, we describe the implementation of the MP refiner. In Section 5 and Section 6, we analyze the performance of the MP refiner. Section 7 contains the conclusion and a discussion on future work.

2 Overview of Standard ML and MP

In this section, we acquaint the reader with Standard ML (SML) [15], a dialect of the ML functional programming language [9]. For a thorough coverage of SML, we encourage the reader to read "ML for the Working Programmer" by Larry Paulson [21].

An SML program is a collection of declarations binding names to values. Running an SML program creates an environment containing the association of names to values represented by the program. Values are obtained by evaluating expressions. An example declaration binding the variable `x` to the value `7` is given below.

```
val x = 4 + 3
```

Functions are declared using the keyword `fun`. For example, the declaration

```
fun fact n = if n <= 0 then 1 else n * fact(n-1)
```

binds the name `fact` to the function value that computes the factorial of an integer. Functions may be defined to take multiple arguments. For example, the function `monus` defined below takes two integer arguments.

```
fun monus x y = if x < y then 0 else x - y
```

Functions may also be declared using the `fn` keyword. The keyword `fn` works like λ in the λ-calculus. For example, we can declare `monus` using the following declaration.

```
val monus = fn x => fn y => if x < y then 0 else x - y
```

If we were writing the declaration of `monus` in the λ-calculus we would write

```
val monus = λ x. λ y . if x < y then 0 else x - y
```

Let–expressions permit programs to define bindings with limited scope. For example, the scope of the function `aux` and variable `y` exists only within the definition of `sqrt`:

```
fun sqrt x = let
   fun aux r  =
        if r*r > x then r-1 else aux (r+1)
   val y = aux 1
in
   y
end
```

2.1 The MP Runtime System

MP [16] is a generic interface to a shared–memory multiprocessor for SML/NJ. **MP** extends the runtime system of SML/NJ to multiplex ML threads on top of threads provided by the operating system (OS threads). An ML thread is the computation unit of an ML expression. ML threads run on a *virtual ML processor*. A multi-threaded ML program may consist of several ML threads that run on a virtual ML processor. To provide parallelism in SML/NJ, we use several virtual processors. Each virtual processor is executed by an OS thread that runs on a separate processor. We obtain parallel computation in ML with

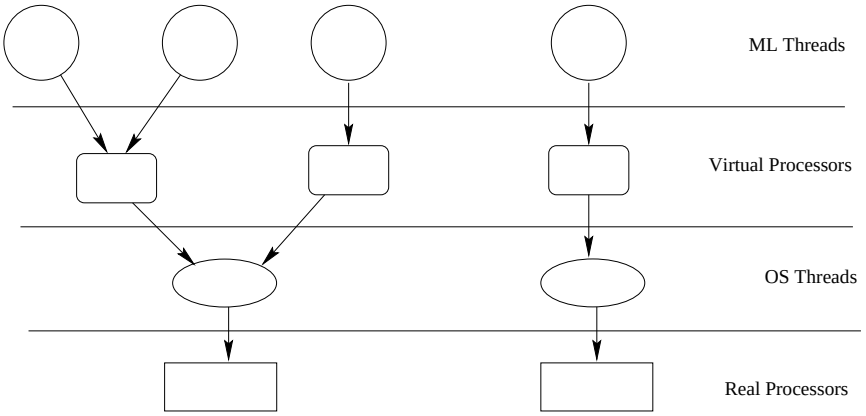

Fig. 1. MP Diagram

multiple virtual processors each mapped to an OS thread that runs on a seperate processor (see Figure 1).

The interface of **MP** provides a language level view of an OS thread as a *proc*. All of the procs share the same heap. However the allocation space of the heap is partitioned into chunks, and each proc is given a chunk to use as a private allocation space. Although the allocation space of the heap is partitioned, garbage collection occurs over the entire heap. In addition, garbage collection is sequential: Only one proc performs garbage collection while the others wait. When a proc wants to perform a garbage collection it notifies the other procs to

suspend themselves. After all of the other procs are suspended, the running proc performs gargage collection over the entire heap. When the garbage collection is completed, all the other procs are resumed. According to [16], the cost of synchronizing for garbage collection has little effect on performance.

3 Nuprl Tactics

In Nuprl, proofs are constructed using partial functions called tactics. A tactic constructs a proof using *primitive refinements*. A primitive refinement is a single step of primitive inference in Nuprl. Nuprl represents proofs as trees containing *unrefined* and *refined* goals. All the interior goals are refined. A leaf is either unrefined or refined. If a leaf of a proof is unrefined, then the proof is *incomplete*. On the other hand, if all the leaves of a proof are refined, then the proof is *complete* and the goal at the root of the proof has been proven. To obtain a complete proof from an incomplete proof, we apply tactics to each of the unrefined goals. Then we replace the unrefined goals with the proofs produced from the applications. We continue this process until we obtain a complete proof. To facilate this method of proof construction, the outcome from applying a tactic to a goal is a list of the unrefined goals and a *validation*. A validation is a function that constructs the proof of a goal from the proofs of its unrefined goals. For instance, if for tactic t and goal g

$$t(g) = ([g_1, \ldots, g_n], v),$$

then $v([g_1, \ldots, g_n])$ is the proof of g constructed by t.

Users create tactics in Nuprl using the functional programming language *Nuprl ML*, a variant of Cambridge ML [19]. Nuprl ML differs from Cambridge ML primarily in its base types and values. The base types and values in Nuprl ML are the data structures and operations of the refiner. For example, `proof`, the type of proofs, is a base type of Nuprl ML. A tactic is any Nuprl ML function with the following type.

```
proof -> proof list * (proof list -> proof)
```

Goals are represented as incomplete proofs with only one node. To evaluate the application of tactics to goals, the refiner is implemented as an interactive read–eval loop for Nuprl ML.

Users often build tactics modularly [10]. First, simple tactics are created that only perform one primitive refinement. These tactics use very simple heuristics or none at all. They can be described as follows.

1. Perform some heuristic.
2. Choose a primitive refinement rule based on the result of the heuristic.
3. Refine the goal with the primitive refinement rule.

Users build sophisticated tactics using simple tactics according to the following scheme.

1. Perform some heuristic.
2. Choose a collection of tactics based on the result of the heuristic.
3. With the aid of a *tactical*, create a new tactic using the collection of tactics.
4. Refine the goal with the new tactic.

A tactical is a combinator for creating new tactics from existing tactics. Two common tacticals are `THEN` and `ORELSE`. Given two tactics, t' and t'', (t' `THEN` t'') is the tactic t such that when t is applied to g, t' is applied first to g to produce the unrefined goals $g_1, \ldots, g_n$. Then t'' is applied to each unrefined goal g_i. Given two tactics t' and t'', (t' `ORELSE` t'') is the tactic t such that $t(g) = t'(g)$ as long as t' is defined on g; otherwise $t(g) = t''(g)$. A variant of `ORELSE` is `ORELSEL`. `ORELSEL` is the same as `ORELSE` except several alternatives may be given if t' fails. Like tactics, tacticals are user–defined.

4 Implementation of the MP Refiner

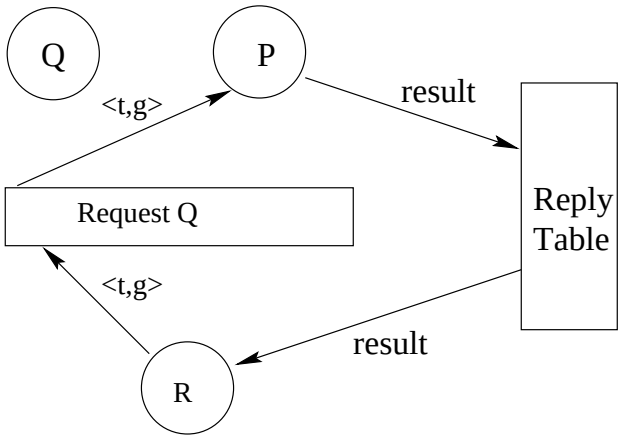

Fig. 2. MP Refiner

We designed the MP refiner as a collection of refiners that share memory. Any refiner in the collection may request that another refiner evaluate an application of a tactic to a goal. In Figure 2, for example, the refiner R *spawns* the application of the tactic t to the goal g by placing $\langle t, g\rangle$ in the *request queue*. The refiner P dequeues $\langle t, g\rangle$ from the request queue and applies t to g. Afterward, P places the result of the application in the *reply table*. R retrieves the results of the application from the reply table. The MP refiner implements parallelism by permitting a single refiner to spawn multiple tactic applications that are evaluated by multiple refiners simultaneously.

We implemented the MP refiner in version 1.09.10 of SML/NJ [3] with the **MP** runtime system for Solaris. Currently, the MP refiner runs on a Sun Sparc

670 shared memory multi-processor with 4 processors and is capable of running on any Sun Sparc multi-processor with at least 3 processors under Solaris 2.5.

Each individual refiner of the MP refiner is implemented as a ML thread. One thread behaves as a normal refiner: It is an interactive ML top–loop with primitive support for developing proofs using tactic refinement. The other threads are ML read–eval loops that only evaluate tactic applications. Each thread runs on a unique processor. We call each thread a *refinement server*.

4.1 Implementation of Concurrent Tacticals

A user creates concurrent tactics using the concurrent tacticals `PTHEN` and `PORELSEL`. `PTHEN` is constructed from the function `PThenOnEach` (see Line 19 in Figure 3). `PTHENOnEach` is derived from `THENOnEach` for creating the various forms of `THEN` in the standard tactic collection of Nuprl 4.1 created by Paul Jackson [10]. We provide the code of `PThenOnEach` in Figure 3. The `else` portion of the `if`

```
(* some helper functions *)
1.  fun applyTac (t,g) = t(g)
2.  val reqHandler = requestHandler applyTac
(*****)
3.  fun PTHENOnEach tac extTac goal =
4.  let
5.      val (subgoals,validation) = tac goal
6.      val tactics = extTac subgoals
7.  in
8.      if not (length tactics = length subgoals)
9.        then
10.          fail "PTHENOnEach: Wrong number of tactics"
11.       else let
12.         val reqIds = enqRequests (tactics,subgoals)
13.         val replies = allReplies reqIds reqHandler
14.         val results = processReply replies
15.       in
16.         combineResults(results, validation)
17.       end
18.  end

19.  fun PTHEN (t,t') =  PTHENOnEach t  (map (fn _ => t'))
```

Fig. 3. Implementation of `PTHEN` using `PTHENOnEach`

statement in `PTHENOnEach` in Figure 3 implements AND-parallelism. On Line 12, `PTHENOnEach` spawns each of the applications of a tactic in the list `tactics` to a subgoal in the list `subgoals`. The tactics in `tactics` are implicitly passed to

```
1.  fun PFirst tactics goal = let
2.    val localTac = hd tactics
3.    val remoteTacs = tl tactics
4.    val requests = mkRequests remoteTacs goal
5.    val reqIds = enqRequests requests
6.    val (results,failedP) = applyTac (localTac(goal))
7.  in
8.    if failedP
9.      then
10.        retrieveFirstReply reqIds
11.     else
12.       (discardResults reqIds;
13.           results)
14. end
15. fun PORELSEL (tac, tacs) = PFirst (tac::tacs)
```

Fig. 4. Implementation of `PORELSEL` using `PFirst`

`PTHENOnEach` through the function `extTac` (see Line 6). The goals in `subgoals` are the subgoals produced from the application of `tac` to `goal` on Line 5. On Line 13, `PTHENOnEach` polls the reply table for the results of the tactic applications it spawned. To prevent deadlock, `allReplies` allows a refinement server to service requests from the request queue while it retrieves replies from the reply table. Once `PTHENOnEach` obtains all the results, it processes the results. Processing the results will cause `PTHENOnEach` to generate an exeception condition if an exeception condition occurred during the evalution of a spawned tactic application. On Line 16, `PThenOnEach` combines the results of the applications with `validation` to produce a list of unrefined goals and a single validation.

Notice that `PTHENOnEach` spawns all of the tactic applications and retains none to perform itself. Because `PTHENOnEach` uses `allReplies` to retrieve the results of the spawned applications, the refinement server may service some requests it placed in the request queue. However, this anomaly does not degrade the performance of concurrent tactics because the overhead of accessing the request queue and reply table is small, less than a millisecond.

`PORELSEL` implements OR-parallelism by deterministically choosing a tactic from a list of tactics to refine a goal. We implement `PORELSEL` using the function `PFirst`. The code for `PFirst` is given in Figure 4. `PFirst` applies each tactic in the list `tactics` to `goal`. The application of the first tactic in `tactics`, `localTac`, to `goal` is performed on the refinement server running `PFirst` (see Line 6). Before applying `localTac` to `goal`, `PFirst` spawns the application of the other tactics to `goal` (see Lines 4 and 5). If the application of `localTac` to `goal` succeeds, then `PFirst` returns the results of the application and removes the results of the spawned applications from the reply table. If the results are not in the reply table when `PFirst` calls `discardReplies`, the reply table will

not allow the results to be entered in the table. If the application of `localTac` to `goal` fails, then `retrieveFirst` is called to obtain the results of the application of the leftmost tactic in `remoteTacs` that succeeds. If none of the applications succeed then `retrieveFirstReply` generates an exceptional condition which is used to indicated that `PFirst` failed.

5 Performance

We measure the comparisons of the running times of sequential tactics and concurrent tactics as speedup. Our sample of tactics were created using `THEN` and `ORESLEL`. The tactics used as arguments to `THEN` and `ORESLEL` are listed in Table 1 and Table 2 along with the elapsed time to complete their application to a goal. The elapsed running times of each tactic in Table 1 and Table 2 is the average of 25 runs. The elapsed running times were obtained using the single processor version of the refiner implemented in SML.

We use the following naming convention for our tactics. For a tactic `tac`i`_`j, where i and j are integers, i is the number of primitive refinements the tactic performs and j is the number of unrefined subgoals generated. For example, `tac65_10` performs 65 primitive refinements and generates 10 subgoals.

The tactics created with `THEN` used each of the tactics in Table 1 as the left hand side argument of `THEN`. We chose these tactics based on the cost model of concurrent tactics in [17]. The tactics used on the right hand side of `THEN` appear in Table 2.

Table 1. Tactics on the Left Hand Side of `THEN`

Tactic	Elapsed Time (Seconds)
tac1_10	0.006
tac1_50	0.022
tac1_100	0.047

We created tactics with `ORELSEL` for which only the last alternative succeeded.[1] . We only used one tactic as the left hand side tactic, namely `tac65_10`. The tactics used as alternatives are listed in Table 2. We used each tactic in Table 2 to create lists containing 1, 5, 20, and 50 alternatives. We used each list as the right hand side argument of `ORLSEL`.

To simplify creating tactics where only the last alternative succeeds, we used the tactical `ORELSELF` (pronounced or-else-l-f). `ORELSELF` is the same as `ORELSEL`

[1] In preliminary tests, tactics created with `ORELSEL` in which the first tactic succeeded out performed there concurrent versions. See [17] for details.

Table 2. The Right Hand Side Tactics

Tactic	Elapsed Time (Seconds)
tac1_1	0.002
tac65_10	0.01
tac400_100	0.649
tac2000_500	3.379
tac10000_100	16.567

except `ORELSELF` applies each alternative and throws aways the results of all the alternatives except the last.

For each sequential tactic, we created a concurrent tactic by replacing `THEN` with `PTHEN`, and `ORELSELF` with `PORELSELF` in the sample of sequential tactics. We obtained the elapsed times of each concurrent tactic to a goal using two, three, and four refinement servers. The MP refiner ran on a Sun Sparc 670 shared–memory multi-processor with four processors. We obtained the times using the timer supplied by SML/NJ. The timer uses gettimeofday to obtain the current time.

6 Results and Discussion

Our results show that concurrent refinement modestly improves the running times of tactics. Figure 5 depicts speedups of tactics using AND-parallelism. Figure 6 depicts speedups of tactics using OR-parallelism. None of the tactics achieve linear speedups—which we expected because of the shared memory bus between the processors. We obtained the best speedup of 2.825, from

```
tac1_100 PTHEN tac65_10                    (1)
```

using four refinement servers. The speed of 2.825 indicates that (1) on four processors is almost three times as fast as

```
tac1_100 THEN tac65_10.                    (2)
```

We obtained significant speedups primarily using AND parallelism and OR parallelism with `tac65_10`. This shows that the MP refiner works well using a fine–grain of parallelism. The running time of `tac65_10` is roughly the average running time of a refinement rule in Nuprl 4.1. Performance tests conducted by Rich Eaton show that in Nuprl 4.1 the average running time of a primitive refinement is 0.012 seconds. The running time of `tac65_10` is 0.1 seconds. Thus it is appears feasible to obtain significant speedups by employing parallelism at the primitive inference level in Nuprl. Originally we expected that parallelism would be more beneficial for tactics such as `tac10000_100`. However our results show that the MP refiner performed poorly on such large tactics.

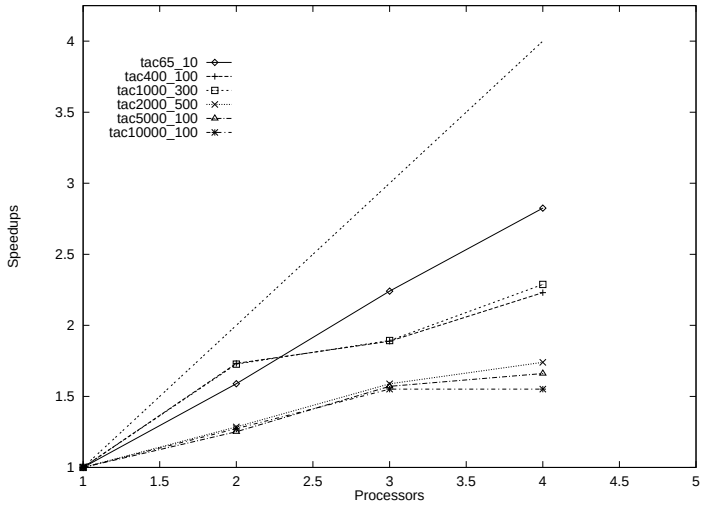

Fig. 5. AND-parallelism Speedups

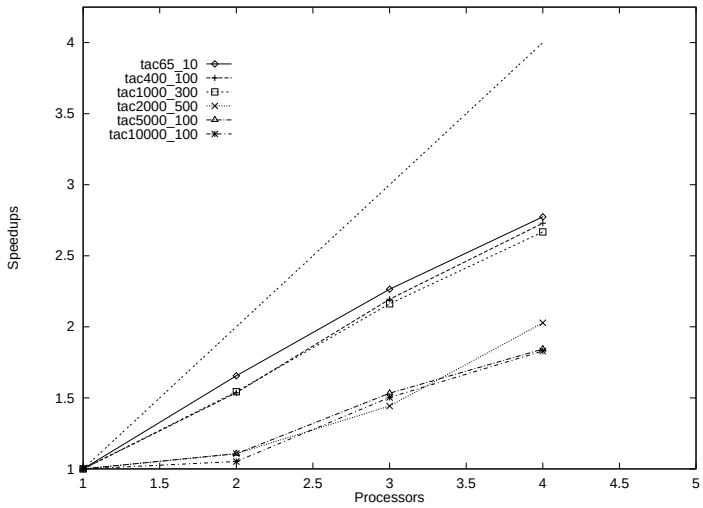

Fig. 6. OR-parallelism Speedups

We believe that memory management of Standard ML of NJ causes the MP refiner to perform poorly for large tactics. Morrisett and Tolmach [16] conjectured that contention between processors for the main-memory bus hinders performance in the MP runtime system. Morrisett and Tolmach used five benchmarks to test the performance of the MP runtime system. Of the five benchmarks, only one experienced near linear speedup[2]. This benchmark also gener-

[2] Actually, the benchmark had super–linear speedup.

ated the least bus traffic. We noticed in our tests that the tactics requiring the least memory had the greatest speedups: compare the speedups of (3), 2.825, and (4), 1.551.

```
tac1_100 PTHEN tac65_10                    (3)
```

```
tac1_100 PTHEN tac10000_100                (4)
```

The application of (3) to a goal requires less memory than the application of (4) to a goal. During a tactic application, the tactic creates a proof. A refinement server needs memory to store the proof. The size of the proof depends on the number of successful primitive refinements invoked by the tactic. (3) generates $1 + 65 \times 100$ primitive refinements, and (4) generates $1 + 100 \times 10000$ primitive refinements.

The MP refiner may behave poorly for large tactics because of contention between refinement servers for the request queue and the reply table. However, we do not believe contention between refinement servers significantly hinders performance of concurrent tactics. The probability of contention is based on the number of requests and the number of refinement servers. Thus contention should equally effect (3) and (4), but their speedups differ drastically.

The overhead of suspending the refinement servers to perform garbage collection may affect the performance of large concurrent tactics. When a refinement server wants to perform garbage collection, it must wait for each of the other refinement servers to enter a write barrier. During this time, the refinement server that wants to perform the garbage collection becomes idle. Also, each refinement server that enters the write barrier becomes idle. As a result, the possibility arises for only one refinement server to be running while the others are idle. However, the amount of time for the refinement servers to synchronize for garbage collection is negligible. Based on the implementation of **MP** and the rate of memory allocation in SML/NJ, a refinement server will enter the write barrier within five to seven machine instructions of the request for a garbage collection.

7 Conclusion

In this paper, we described the implementation of the first parallel interactive theorem prover. Our prover, the MP refiner, uses AND-parallelism and OR-parallelism on a shared–memory multi-processor. The MP refiner is a multi-processor implementation in SML of the Nuprl refiner. The MP refiner makes significant improvement on the throughput of tactics when using a fine-grain of parallelism and only modest improvements using a coarse-grain of parallelism. Granularity is measured in the number of successful primitive refinements invoked by a tactic. We obtained the best results parallelizing tactics that have running times close to the average running time of a step of primitive inference in Nuprl. We believe the modest improvements with coarse-grain parallelism results from contention between the processors for the main-memory bus due to

the memory management of SML/NJ. Concurrent tactics employing coarse-grain parallelism have higher memory requirements than concurrent tactics employing fine-grain parallelism.

7.1 Non-determinism in Concurrent Refinement

Many tactic theorem provers support *replaying* of proofs. Replaying a proof reconstructs the proof using the same tactics that created it. Replaying a proof may produce a proof different from the original proof. In other words, a user can construct two different proofs for a goal g using the exact same tactics. This is possible because the user may construct the proofs within two different states. For example, suppose a tactic t makes use of a theorem T to provide a complete proof of a goal g. If we apply t to g in a state with T present, then t will completely prove g. In a state without T, however, t will not completely prove g. As long as the state is the same, a user will construct the same proof for a goal g if he uses the same tactics (see [17]). However, with concurrent tactics, a user can construct different proofs for a goal using the same tactics with respect to a single state. The outcome of concurrent tactics, like many concurrent programs, is effected by the order of execution of its threads. For example, suppose t is the concurrent tactic (s `PORELSEL` $[s', s'']$). Suppose that s fails if the mutable shared variable x is 1 and succeeds otherwise. Futhermore, suppose s' changes x to 1 and s'' changes x to 0. For tactics t and t', let $t \rightarrow t'$ mean that t executes before t'. Suppose that the execution order of s', s'', and s is $s' \rightarrow s'' \rightarrow s$. Then the outcome of t is the outcome of s. However if the execution order of s', s'' and s is $s'' \rightarrow s' \rightarrow s$. Then the outcome of t is either the outcome of s' or s''. In [17], we describe a way to have determinstic concurrent tactics that have shared mutable variables.

7.2 Future Work

In the future, we intend to experiment with pipelining in the MP refiner. Pipelining is obtain by creating concurrent tactics using multiple compositions of `PTHEN`. To understand pipelining, consider applying the tactic (t `THEN` (t' `THEN` t'')) to the goal g. First, the tactic t is applied to g to produce the subgoals $g_1, \ldots g_n$. Afterward, t' is applied to each g_i producing the subgoals $g_{i,1}, \ldots, g_{i,m_i}$. Pipelining involves applying t'' to $g_{1,1}, \ldots, g_{1,m_1}$, for example, and t' to g_2 simultaneously. Multiple compositions of `PTHEN`, for example (t `PTHEN` (t' `PTHEN` t'')), implements pipelining.

Additional future work includes creating a parallel interactive theorem prover using a master–slave design. The master prover is a typical interactive theorem prover that uses one or more automatic theorem provers as slaves. The master prover spawns goals for the slaves to prove completely and automatically. The master prover only spawns goals it believes the slave can prove automatically. While the slaves are proving the goals, the master continues with its own work. This approach would probably be more favorable to interactive theorem provers such as IMPS [7] or PVS [18] that employ decision procedures as part of the

inference strategy. Nuprl can also use this type of structure for proving well-formedness goals [1]. Well-formedness goals are generated by the Nuprl type theory refinement rules to validate that a term is a type. These goals usually can be proven automatically by the `Auto` tactic [10]. A slave prover simply refines all goals it receives using `Auto`.

References

1. Stuart F. Allen. *A Non-type-theoretic Semantics for a Type Theoretic Language.* PhD thesis, Cornell University, 1987.
2. Andrew W. Appel. A runtime system. *Journal of Lisp and Symbolic Compuation*, 3, 1990.
3. Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In *Programming Language Implementation and Logic Programming: 3rd International Symposium*, pages 1–13. Springer-Verlag, 1991.
4. Soumitra Bose et al. Parthenon: A parallel theorem prover for non–horn clauses. *Journal of Automated Reasoning*, 8:153–181, 1989.
5. Robert L. Constable, Stuart F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, Douglas J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, Englewood Cliffs, NJ, 1986.
6. C. Cornes, J. Courant, J. Filliatre, G. Huet, P. Manoury, C. Munoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saibi, and B. Werner. The Coq Proof Assistant Reference Manual: Version 5.10. Unpublished, 1995.
7. William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. IMPS: A system description. In *Eleventh Conference on Automated Deduction (CADE)*, volume 607 of Lecture Notes in Computer Science, pages 701–705. Springer–Verlag, 1990.
8. M. J. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge Univesity Press, 1993.
9. Michael Gordon, Arthur Milner, and Christopher Wadsworth. *Edinburgh LCF.* Number 78 in *Lecture Notes in Computer Science.* Springer-Verlag, New York, 1979.
10. Paul Jackson. *Enhancing the Nuprl Proof Development System and Applying it to Computational Abstract Algebra.* PhD thesis, Cornell University, 1995.
11. Matt Kaufmann. A user's manual for an interactive enhancement to the Boyer-Moore theorem prover. Technical Report 19, Computational Logic, Inc., 1988.
12. R. Letz, J. Schumann, S. Bayerl, and W. Bibel. SETHEO: A high performance theorem prover. *Journal of Automated Reasoning*, 8:183–212, 1992.
13. Ewing L. Lusk and William W. McCune. Experiments with Roo, a paralled automated deduction system. In *Parallelization in Inference Systems*, number 590 in *Lecture Notes in Artificial Intelligence*, pages 139–162. Springer-Verlag, 1990.
14. W. W. McCune. OTTER 1.0 user's guide. Technical Report ANL-88-44, Argonne National Laboratory, 1989.
15. Robin Milner, Mads Tofte, and Robert Harper. *The Definition Of Standard ML.* MIT Press, Cambridge, MA, 1990.
16. J. Gregory Morrisett and Andrew Tolmach. Procs and locks: A portable multiproceessing platform for standard ml of new jersey. In *Proceedings of the Fourth*

ACM SIGPLAN Symposium on Principles of Practice of Parallel Programming, pages 198–207. ACM, 1994.
17. Roderick Moten. *Concurrent Refinement in Nuprl*. PhD thesis, Cornell University, 1997.
18. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *Eleventh International Conference on Automated Deduction (CADE)*, number 607 in *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer Verlag, 1992.
19. Larry Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge University Press, Cambridge, 1987.
20. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Number 828 in *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1994.
21. Lawrence C. Paulson. *ML for the Working Programmer (Second Edition)*. Cambridge University Press, Cambridge, 1996.
22. J. Schumann and R. Letz. Partheo: A high–performance parallel theorem prover. In *Tenth International Conference on Automated Deduction (CADE)*, number 449 in *Lecture Notes in Artificial Intelligence*, pages 40–56. Springer-Verlag, 1990.
23. Johann M. Ph. Schumann. Parallel theorem provers: An overview. In *Parallelization in Inference Systems*, number 590 in *Lecture Notes in Artificial Intelligence*, pages 26–50. Springer-Verlag, 1990.
24. John K. Slaney and Ewing L. Lusk. Parallelizing the closure computation in automated deduction. In *Tenth International Conference on Automated Deduction (CADE)*, number 449 in *Lecture Notes in Artificial Intelligence*, pages 28–39. Springer Verlag, 1990.
25. Christian B. Suttner. A parallel theorem prover with heuristic work distribution. In *Parallelization in Inference Systems*, number 590 in Lecture Notes In Computer Science, pages 243–253. Springer Verlag, 1992.
26. Christian B. Suttner. *Parallelization of Search–based Systems by Static Partitioning with Slackness*. PhD thesis, Institut für Informatik, TU München, 1995.
27. Christian B. Suttner and Manfred B Jobmann. Simulation analysis of static partitioning with slackness. In *Parallel Processing for Artificial Intelligence 2*, number 15 in Machine Intelligence and Pattern Recognition, pages 93–105. North–Holland, 1994.
28. Christian B. Suttner and Johann Schumann. Parallel automated theorem proving. In *Parallel Processing for Artificial Intelligence 1*, number 14 in Machine Intelligence and Pattern Recognition, pages 209–257. North–Holland, 1994.
29. D. H. D. Warren. The SRI model for OR-parallel execution of Prolog: Abstract design and implementation issues. In *International Symposium on Logic Programming*, pages 92–102. North–Holland, 1987.
30. G.A. Wilson and J. Minker. Resolution, refinements, and search strategies: A comparative study. *IEEE Transactions on Computers*, C-25:782–801, 1976.

This article was processed using the LaTeX macro package with LLNCS style