Minimizing the overheads of dependent AND-parallelism

Peter Wang[1] and Zoltan Somogyi[2]

1 **Mission Critical Australia**
novalazy@gmail.com

2 **Department of Computer Science and Software Engineering**
University of Melbourne, Australia and
National ICT Australia (NICTA)
zs@unimelb.edu.au

Abstract

Parallel implementations of programming languages need to control synchronization overheads. Synchronization is essential for ensuring the correctness of parallel code, yet it adds overheads that aren't present in sequential programs. This is an important problem for parallel logic programming systems, because almost every action in such programs requires accessing variables, and the traditional approach of adding synchronization code to all such accesses is so prohibitively expensive that a parallel version of the program may run more slowly on four processors than a sequential version would run on one processor. We present a program transformation for implementing dependent AND-parallelism in logic programming languages that uses mode information to add synchronization code only to the variable accesses that actually need it.

1998 ACM Subject Classification D.3.3 Concurrent Programming Structures, D.3.4 Compilers

Keywords and phrases synchronization, program transformation

1 Introduction

The usual motivation for parallelism is higher performance. Parallelizing imperative programs is notoriously errorprone, whereas in some declarative languages, introducing parallelism into a program can be as easy as adding a directive asking for it. Unfortunately, parallel implementations of declarative languages have a big enemy: overheads.

Parallel systems have overheads for the mechanisms used to manage the parallelism itself, as well as the overheads present in the sequential system they are based on. To get speedups over the best sequential systems, overheads must be kept to a minimum. Unfortunately, most sequential implementations of logic programming languages have quite high overheads. The best way to minimize overheads is to design a language that removes the need for them as far as possible, which is best done by moving all possible decisions to compile time. The only logic programming language designed with this objective is Mercury. Its implementation has consequently long been the fastest among sequential implementations of logic programming languages [8], though recently systems like YAP [3] have been catching up. This speed makes Mercury a good starting point for a parallel logic programming implementation.

Most parts of most Mercury programs are deterministic, so Mercury needs AND-parallelism, not OR-parallelism, and since most goals in predicate bodies depend on the goals preceding them, it needs *dependent* AND-parallelism. If you want to use this parallelism to achieve higher speeds than would be possible on one processor in *any* language (which may be the "killer application" of declarative languages), then you need to be very aggressive in keeping down the overheads of the synchronization required to implement the dependencies between parallel goals. Our paper presents a way to do this.

The structure of the paper is as follows. Section 2 briefly reviews the Mercury language. Sections 3 and 4 present the main contributions of the paper, our synchronization and specialization transformations respectively. Section 5 gives some performance results, while section 6 concludes with comparisons to related work.

2 Background

The part of Mercury relevant to this paper can summarized by this abstract syntax:

pred P	$: \ p(x_1, \ldots, x_n) \ \leftarrow \ G$	predicates
goal G	$: \ x = y \mid x = f(y_1, \ \ldots, \ y_n)$	unifications
	$\mid p(x_1, \ \ldots, \ x_n) \mid x_0(x_1, \ \ldots, \ x_n)$	first and higher order calls
	$\mid (G_1, \ \ldots, \ G_n) \mid (G_1 \ \& \ \ldots \ \& \ G_n)$	seq and par conjunctions
	$\mid (G_1; \ldots; G_n) \mid$ switch $x \ (\ldots; \ f_i : G_i; \ldots)$	disjunctions and switches
	$\mid (\mathit{if} \ G_c \ \mathit{then} \ G_t \ \mathit{else} \ G_e) \mid \mathit{not} \ G$	if-then-elses and negations
	$\mid \mathit{some} \ [x_1, \ldots, x_n] \ G$	quantifications

The atomic constructs of Mercury are unifications, plain first-order calls, and higher-order calls. The composite constructs include sequential disjunctions, if-then-elses, negations and existential quantifications. These should all be self-explanatory. A parallel conjunction is a conjunction in which all the conjuncts execute in parallel. They may be independent or dependent; if dependent, the dependencies must be respected. (The mechanism for enforcing the dependencies is the subject of this paper.) A switch is a disjunction in which each disjunct unifies the same variable, whose value is known, with a different function symbol.

Mercury has a strong mode system. The mode system classifies each argument of each predicate as either input or output; there are exceptions, but they are not relevant to this paper. If input, the caller must pass a ground term for that argument. If output, the caller must pass a distinct free variable, which the predicate will instantiate to a ground term. A predicate may have more than one mode; we call each one a *procedure*. The Mercury compiler generates different code for different procedures. The compiler is responsible for reordering conjuncts in both sequential and parallel conjunctions as necessary to ensure that for each variable shared between conjuncts, the goal that generates the value of the variable (the *producer*) comes before all the goals that use this value (the *consumers*). This means that for each variable in each procedure, the compiler knows exactly where in that procedure that variable gets grounded.

The mode system both establishes and requires some invariants. The conjunction invariant says that on any execution path consisting of conjoined goals, each variable that is consumed by any one of the goals is produced by exactly one goal. The branched goal invariant says that in disjunctions, switches and if-then-elses, each branch of execution must produce the exact same set of variables that are visible from outside the branched goal, with one exception: a branch of execution that cannot succeed may produce a subset of this set.

Each procedure and goal has a determinism, which puts limits on the number of its possible solutions. Goals with determinism *det* succeed exactly once, *semidet* goals at most once, *multi* goals at least once, while *nondet* goals may succeed any number of times. Each procedure's mode declaration declares the procedure's determinism.

Before we started this work, Mercury already supported independent AND-parallelism [2]. This meant that programmers who wanted to use parallelism could indicate *which* conjunctions should be executed in parallel by writing the parallel connective '&' instead of the sequential connective ',' between the conjuncts. The compiler then checked that the

```
:- pred p(int, int, int).       % after transformation
:- mode p(in, in, out) is det.  p(A, B, C) :-
                                  new_future(FutureD),
% before transformation           (
p(A, B, C) :-                       q(A, D),    % produces D
  (                                 signal_future(FutureD, D),
    q(A, D),     % produces D       r(D, E)     % produces E
    r(D, E)      % produces E     ) & (
  ) & (                             s(B, F),    % produces F
    s(B, F),     % produces F       wait_future(FutureD, D'),
    t(D, F, G)   % produces G       t(D', F, G) % produces G
  ),                              ),
  C = D + E + G. % produces C     C = D + E + G.  % produces C
```

Figure 1 Example of synchronizing a parallel conjunction

code satisfied the restrictions imposed on parallel conjunctions. The first restriction required all conjuncts in parallel conjunctions to be det, and the second required them to be all independent (no conjunct could produce any variable consumed by any other conjunct). The reason for the first check is that while there are ways to run nondeterministic conjuncts in parallel, the overheads of the algorithms required to assure correctness in such cases make it almost impossible to achieve speedups, even on several cores, over the best possible sequential version of the program at hand. We therefore keep the first restriction, even though our work obviously removes the second.

3 Synchronization for dependent AND-parallelism

Our main objective is to minimize the cost of synchronization by incurring it only when it is truly needed. In Mercury, this is relatively easy to do, due to the compiler's complete knowledge of the data flow in every predicate. (This was one of the design criteria for the language, even though the original implementation was sequential.) Given two or more conjuncts in a conjunction, the compiler knows not only which variables they have in common (i.e. which variables they *share*), but for each shared variable, it also knows which conjunct produces the variable and which conjuncts consume it.

The main parts of our implementation are the synchronization transformation (described in this section) and the specialization transformation (in section 4) . They both operate on a representation of the source code, which is based on the Mercury abstract syntax (section 2).

The synchronization transformation processes each parallel conjunction in the program independently. For each parallel conjunction, it computes the set of variables that need synchronization because they are produced by one conjunct and consumed by later conjuncts. For each such variable, it creates a new variable that we call a *future*. It then modifies all conjuncts to use these futures to synchronize all accesses to the shared variables.

We will introduce the synchronization transformation using the example in figure 1. The body of `p` contains a parallel conjunction, whose first conjunct is the sequential conjunction `q(A, D), r(D, E)` and whose second is the sequential conjunction `s(B, F), t(D, F, G)`. The comments indicate the modes of the calls.

The basic idea of our work is that

- the producer of a shared variable will signal the availability of a value for the shared variable *immediately after* that value has been computed, and
- all consumers of a shared variable will wait for the availability of a value for the shared

```
add_sync_par_conj(Conjuncts, Purity) returns Goal:
let SharedVars be the set of variables that are
  (a) produced by one conjunct and
  (b) consumed by one or more later conjuncts
if SharedVars = ∅:
  Goal := par_conj(Conjuncts)
else:
  FutureMap := ∅
  AllocGoals := []
  for each Var in SharedVars:
    create a new var FutureVar
    FutureMap := FutureMap ∪ {Var → FutureVar}
    AllocGoals := [new_future(FutureVar) | AllocGoals]
  ParConjuncts := []
  for each Conjunct in Conjuncts:
    ParConjuncts := ParConjuncts ++
      [add_sync_par_conjunct(Conjunct, SharedVars, FutureMap, yes)]
  Goal := seq_conj(AllocGoals ++ [par_conj(ParConjuncts)])
```

Figure 2 Synchronizing a parallel conjunction

variable *immediately before* they use that value for the first time.
No consumer ever accesses the shared variable directly. Instead, each consumer waits on the future, and creates its own copy (`D'` in this case).

The data structures we use for this synchronization are the futures. (We got the name from MultiLisp [4]; our futures resemble theirs.) A future consists of these four fields:

- a boolean that says whether the future has been signalled yet;
- the value of the variable, if it has been signalled;
- a list of the contexts suspended waiting for this future, if it has not;
- a mutex to allow only one engine to access the future at a time.

The implementations of the three operations on futures are quite straightforward.

- `new_future` allocates a new future on the heap, and initializes its fields to the obvious values: not yet signalled, no waiting contexts.
- `wait_future` locks the mutex. If the future has been signalled, it picks up the its value and unlocks the mutex. Otherwise, the thread suspends itself on the future, unlocking the mutex. It will pick up the future's value when it resumes.
- `signal_future` locks the mutex, sets the value of the shared variable, records that the value is now available, makes all the threads suspended on the future runnable, and unlocks the mutex.

The synchronization transformation traverses the bodies of all procedures looking for parallel conjunctions. When it finds one, it invokes the algorithm in figure 2. In our algorithmsall variable names start with an upper case letter, names starting with lower case letters represent the names of functions or data constructors, square parentheses represent lists as in Prolog syntax, and ++ means list concatenation. The `seq_conj` and `par_conj` data constructors create conjunctions of the named kinds out of a list of conjuncts.

If none of the conjuncts consumes a variable produced by one of the other conjuncts, then the conjunction represents independent AND-parallelism, and requires no synchronization (beyond the barrier at the end, which we will ignore from now on). Otherwise, the conjunction

```
add_sync_par_conjunct(Goal, SharedVars, FutureMap, Rename) returns NewGoal:
ConsumedVars := SharedVars ∩ Goal’s consumed vars
ProducedVars := SharedVars ∩ Goal’s produced vars
if ConsumedVars and ProducedVars are both ∅:
  NewGoal := Goal
else:
  NewGoal := Goal
  for each ConsumedVar in ConsumedVars:
    NewGoal := insert_wait_into_goal(NewGoal, ConsumedVar, FutureMap)
  for each ProducedVar in ProducedVars:
    NewGoal := insert_signal_into_goal(NewGoal, ProducedVar, FutureMap)
  if Rename:
    map each var in ConsumedVars to a fresh clone var
    apply this substitution to NewGoal
```

Figure 3 Synchronizing one parallel conjunct

is dependent and needs synchronization using futures. We create one future for each shared variable; `FutureMap` maps each shared variable to the variable that holds its future. The goals that create and initialize the futures (`AllocGoals`) will all run before execution enters the parallel conjunction (`ParConjuncts`) in the transformed code.

Figure 3 shows the algorithm that transforms each parallel conjunct to ensure that the conjunct waits for each shared variable it consumes before its first use, and that it signals the availability of each shared variable it produces just after it is bound. First, the algorithm finds out which shared variables it consumes and which it produces. If there are no variables in either category, then this conjunct is independent of the others and needs no synchronization. Otherwise, we loop over the consumed and produced variables, each iteration modifying the conjunct by adding the code required to synchronize accesses to that variable.

The reason why `add_sync_par_conj` tells `add_sync_par_conjunct` to rename the variables consumed by each conjunct is to ensure that no variable is produced more than once on any one execution path (the mode system’s conjunction invariant). The calls to `wait_future` bind their second argument. Since each shared variable is bound once in its producing conjunct, binding it again in one or more of its consuming conjuncts would violate this invariant. This way, each consuming conjunct gets its own copy of the shared variable, and

```
insert_wait_into_goal(Goal, ConsumedVar, FutureMap)
  returns <NewGoal, WaitAllPaths>:
if Goal does not consume ConsumedVar:
  NewGoal := Goal
  WaitAllPaths := false
else:
  switch on the type of Goal:
    the switch cases are shown in the following figures
    each case sets NewGoal and WaitAllPaths
  if not WaitAllPaths:
    create a clone of ConsumedVar, call it CloneVar
    apply the substitution {ConsumedVar → CloneVar} to NewGoal
```

Figure 4 Inserting waits into goals, top level

```
    case unify, plain call, higher order call:
      look up ConsumedVar in FutureMap; call the result FutureVar
      WaitGoal := wait_future(FutureVar, ConsumedVar)
      NewGoal := seq_conj([WaitGoal, Goal])
      WaitAllPaths := true
    case sequential conjunction:
      let Goal be seq_conj([Conjunct_1, ..., Conjunct_n])
      NewConjuncts := []
      WaitAllPaths := false
      for i = 1 to n:
        if Conjunct_i consumes ConsumedVar:
          <NewConjunct, ConjunctWaitAllPaths> :=
            insert_wait_into_goal(Conjunct_i, ConsumedVar, FutureMap)
          if ConjunctWaitAllPaths:
            NewConjuncts := NewConjuncts ++ [NewConjunct] ++
              [Conjunct_i+1, ...  Conjunct_n]
            WaitAllPaths := true
            break out of the loop
          else:
            NewConjuncts := NewConjuncts ++ [NewConjunct]
        else:
          NewConjuncts := NewConjuncts ++ [Conjunct_i]
      NewGoal := seq_conj(NewConjuncts)
```

Figure 5 Inserting waits into goals, part 1

the code *after* the parallel conjunction gets the original version from the producing conjunct.

Figures 4, 5, 6 and 7 each show part of the algorithm for inserting the call to `wait_future` on `ConsumedVar` into a goal.

The `insert_wait_into_goal` function returns `NewGoal`, an updated version of `Goal` in which every occurrence of `ConsumedVar` is preceded by code that picks up the value of `ConsumedVar` by waiting on its future. It also says whether all execution paths that lead to the success of `NewGoal` have such a wait operation on them. If they all do, then the code following the original `Goal` will be able to access `ConsumedVar` without waiting, and thus does not need to be transformed. If some or all do not, then the code after the original `Goal` will need to wait for `ConsumedVar`, and thus does need to be transformed. However, we do not want both `Goal` and the code following it to wait for and thus bind `ConsumedVar`, as this would violate the conjunction invariant. That is why, if `WaitAllPaths = false`, we replace all occurrences of `ConsumedVar` in `NewGoal` with a fresh variable. This renaming leaves the meaning of `NewGoal` unchanged. Therefore when `insert_wait_into_goal` returns `WaitAllPaths = false`, `NewGoal` will not bind `ConsumedVar`; it won't refer to it at all.

If the goal given to `insert_wait_into_goal` does not consume `ConsumedVar`, then the given goal does not need modification. If it does consume `ConsumedVar`, then the kind of modification it needs depends on what kind of goal it is, which is why the main body of the `insert_wait_into_goal` is a switch on goal type. The base case handles atomic goals (goals that do not contain other goals): unifications, calls, and higher order calls. For these goals, we insert a call to `wait_future` before the goal to wait for and pick up the value of the consumed variable. For unifications, this is the best we can do. For calls, we can wait for `ConsumedVar` closer to the time when it is actually needed, but we will worry about that in

```
case parallel conjunction:
  let Goal be par_conj([Conjunct_1, ..., Conjunct_n])
  NewConjuncts := []
  WaitAllPaths := false
  for i = 1 to n:
    if Conjunct_i is the first conjunct that consumes ConsumedVar:
      <NewConjunct, WaitAllPaths> :=
        insert_wait_into_goal(Conjunct_i, ConsumedVar, FutureMap)
      NewConjuncts := NewConjuncts ++ [NewConjunct]
    else if Conjunct_i consumes ConsumedVar:
      <NewConjunct, _> :=
        insert_wait_into_goal(Conjunct_i, ConsumedVar, FutureMap)
      create a clone of ConsumedVar, call it CloneVar
      apply the substitution {ConsumedVar → CloneVar} to NewConjunct
      NewConjuncts := NewConjuncts ++ [NewConjunct]
    else:
      NewConjuncts := NewConjuncts ++ [Conjunct_i]
  NewGoal := par_conj(NewConjuncts)
```

Figure 6 Inserting waits into goals, part 2

the next section.

`insert_wait_into_goal` processes the conjuncts of plain sequential conjunctions left to right. When it finds a conjunct that consumes `ConsumedVar`, it calls itself recursively to insert the call to `wait_future` into that conjunct. If all successful execution paths inside this conjunct wait for `ConsumedVar`, then it is guaranteed to be available by the time execution gets to the following conjuncts, so we can leave them unchanged. If only some execution paths inside this conjunct wait for `ConsumedVar`, then `NewConjunct` will wait on a renamed version of `ConsumedVar`, so `ConsumedVar` itself won't be bound when execution gets to the next conjunct. If none of the conjuncts that consume `ConsumedVar` wait for it on all paths, then `ConsumedVar` itself won't be bound by the conjunction itself, which is why we return `WaitAllPaths = false`.

Parallel conjunctions differ from sequential conjunctions in that even if one conjunct waits for `ConsumedVar` on all paths, the later conjuncts cannot assume that it will be available when they need it. Each conjunct that consumes `ConsumedVar` thus needs to wait for it independently. However, as above, having several conjuncts all ask their call to `wait_future` to bind the same variable (`ConsumedVar`) would violate the invariant that no variable have more than one producer in a conjunction. `insert_wait_into_goal` therefore renames the occurrences of `ConsumedVar` in all but one of the conjuncts; the last one may also be renamed by `add_sync_to_par_conjunct` to avoid a similar collision with the binding made by a producer conjunct.

Given a switch on `ConsumedVar`, we obviously need to wait for the value of `ConsumedVar` before the switch. For switches on other variables, we insert the wait into the switch arms that consume `ConsumedVar`.

To see how `insert_wait_into_goal` treats other kinds of goals, and for the code of `insert_signal_into_goal`, see the long version of this paper on the Mercury web site.

```
case switch:
  if the switch is on ConsumedVar:
    look up ConsumedVar in FutureMap; call the result FutureVar
    WaitGoal := wait_future(FutureVar, ConsumedVar)
    NewGoal := seq_conj([WaitGoal, Goal])
    WaitAllPaths := true
  else:
    NewCases := []
    WaitAllPaths := true
    for each Case in Goal:
      if Case's goal consumes ConsumedVar:
        <NewCaseGoal, CaseWaitAllPaths> :=
          insert_wait_into_goal(Case's goal, ConsumedVar, FutureMap)
        NewCase := replace Case's goal with NewCaseGoal
        NewCases := NewCases ++ [NewCase]
        WaitAllPaths := WaitAllPaths and CaseWaitAllPaths
      else:
        NewCases := NewCases ++ [Case]
        WaitAllPaths := false
    NewGoal := replace Goal's cases with NewCases
```

Figure 7 Inserting waits into goals, part 3

4 The specialization transformation

The algorithms in the previous section try to push each `wait_future` operation as late as they can and each `signal_future` operation as early as they can, in order to maximize parallelism by maximizing the time during which the producer and the consumers of the shared variable can all run in parallel. However, the amount of parallelism they can create is limited by procedure boundaries. Overcoming this limitation is the task of the specialization algorithm. The first step in this algorithm is the "making requests" step, which looks for places in the code where creating specialized versions of procedures can increase parallelism:

- We look for plain calls to procedures preceded by one or more calls to `wait_future`, followed by one or more calls to `signal_future`, or both.
- For each of these `wait_future` operations, we check whether the waited-for variable is an input argument of the call and the callee does non-trivial work before it needs the variable. `PushedInputs` is the set of variables for which the answer is "yes".
- For each of these `signal_future` operations, we check whether the signalled variable is an output argument of the call and the callee does non-trivial work after it produces the variable. `PushedOutputs` is the set of variables for which the answer is "yes".
- If `PushedInputs` ∪ `PushedOutputs` is not empty, and we have access to the code of the callee, we replace the call, all the `wait_futures` on `PushedInputs` and all the `signal_futures` on `PushedOutputs` with a call to a specialized version of the called procedure, one in which all the replaced waits and signals will be done in the callee. We also request that this specialized version be created. This request identifies the variables in `PushedInputs` and `PushedOutputs` by their position in the call's argument list.

The specialized version of the callee will have each formal parameter corresponding to a variable in `PushedInputs` or `PushedOutputs` replaced by a newly-created variable that will hold its future.

The second step executes each request by creating the specialized version of the request's callee. To generate the body of the new procedure, it invokes `add_sync_par_conjunct` from figure 3 with the callee's original code as `Goal`. As `SharedVars`, it passes the formal parameters corresponding to `PushedInputs` ∪ `PushedOutputs`, and `FutureMap` will map each variable in `SharedVars` to the future that replaced it in the list of formal parameters. `add_sync_par_conjunct` will push the synchronization code for each variable in `SharedVars` as deeply into the code of the callee as possible. Since the goal returned by `add_sync_par_conjunct` will be the body of the specialized procedure and thus will not be conjoined to anything, it is ok for this goal to bind the consumed variables, which is why we pass "no" as the value of `Rename`.

We invoke the specialization algorithm after the algorithms of section 3 have inserted synchronization operations into all the procedures in the module being compiled that contain dependent parallel conjunctions. We invoke the "making requests" step on all of these procedures, looking for opportunities for specialization, and modifying the code at each such opportunity as if the specialized version of the callee already existed. In doing so, we collect a queue of specialization requests, none of which have yet been acted upon. The rest of the specialization algorithm is a fixpoint operation. While there are specialization requests that have not yet been acted upon, the algorithm selects one such request, executes it (creating the requested specialized procedure), and then invokes the "making requests" step of the algorithm again on the newly created procedure to look for more specialization requests. Some of these may have been already acted upon, some of them may already be in the queue, and some may be new; we add the new ones to the queue. The algorithm must terminate because the number of procedures and of possible specialization requests are both finite.

5 Performance evaluation

We have room to report on only one benchmark, but it should suffice to demonstrate the low overheads of our system. This benchmark is a raytracer: its top level loops over the rows of the picture, with each iteration computing the RGB values of the pixels in a chunk of consecutive rows. We have two versions of this loop. The independent version (i1) computes the pixels for these rows, (i2) recurses to compute the pixels for the remaining rows, then (i3) add both sets of pixels to the list of pixels so far. The parallelized conjunction is `(i1 & i2), i3`, and there is no data flow from i1 to i2. The dependent version (d1) computes the pixels for these rows, (d2) adds these pixels to the list of pixels so far, and then (d3) recurses to compute the pixels for the remaining rows, based on the updated list of the pixels so far. The parallelized conjunction is `(d1, d2) & d3`, and there *is* data flow from d1 to d2 and from d2 to d3. We tested both forms of the loop with varying chunk sizes, which divided the rows into 1, 8, 16 and 32 chunks. The table shows speedups compared to the best sequential version on our test machine, which was a Dell Optiplex 755 PC with a 2.4 GHz Intel Core 2 Quad Q6600 CPU. Each test was run ten times.

	1i	8i	16i	32i	1d	8d	16d	32d
sequential	1.00	1.00	1.00	1.00	0.99	0.99	0.99	0.99
parallel, 1 CPU	0.91	1.08	1.08	1.10	0.90	1.02	1.03	1.04
parallel, 2 CPUs	N/A	1.46	1.73	1.74	N/A	2.05	2.14	2.13
parallel, 3 CPUs	N/A	2.16	2.19	2.21	N/A	2.36	2.73	2.72
parallel, 4 CPUs	N/A	2.41	2.49	2.55	N/A	2.83	3.28	3.32

The parallel version of the system needs to use a real machine register to point to threadspecific data, such as each engine's abstract machine registers. On x86s, this leaves only one real register for the Mercury abstract machine, which is why switching to the parallel version of the system can yield a 10% slowdown on one CPU. However, inevitable differences between program versions (such as in the placement of code and data) happen to allow similar speedups as well.

The independent versions on the left do not need any of the algorithms in this paper. The dependent versions on the right do, and thus incur extra synchronization costs. However, these costs must be very small, since they are *massively* outweighed by the better cache utilization resulting from appending each chunk's pixels to the pixels so far immediately after computing those pixels. (In all versions, the append is done in constant time using a cord, not a list, and all versions do the same appends and compute the same cord.)

Different rows take different times to render. Dividing the input into more chunks evens out such fluctuations. The data shows this to be worthwhile (up to a point) even though it also increases synchronization overhead. Overall, the speedup of 3.32 over the sequential version and $3.32/1.04 = 3.19$ over the 1-CPU parallel version shows that we have succeeded in keeping synchronization costs to a very low level.

6 Conclusion

We don't have space for a detailed comparison with all existing parallel logic programming systems [9], so we list only the main differences between our work and previous systems.

- Unlike Concurrent Prolog, Parlog, GHC and their descendants, our system incurs synchronization overhead on accesses only to a very small proportion of variables; the fraction can be as low as tens out of billions.
- Our system cannot deadlock: every shared variable has a producer, which (unless it throws an exception) cannot exit without binding the variable.
- Our system does not require programmers to divide clauses into guard and body, much less into ask guard, tell guard and body.
- Unlike e.g. Janus [6], our system allows a variable to have more than one consumer.
- Unlike most other systems that allow multiple consumers, ours requires synchronization for consumers only up to the first one that waits for the variable on all paths.
- Unlike the DASWAM and similar parallel Prolog systems [7], our system does not allow the producer of a variable to be decided at runtime, so it does not have to keep track of which goal is the leftmost goal referring to a variable.
- Unlike CIAO's `<&` operator [1], our futures are significantly simpler to implement, and support finer grained parallelism.
- Unlike Moded FGHC [10], our system supports separate compilation.

One important further advantage of our system is that it completely eliminates the need for a system for controlling the order of parallel code's interactions with the outside world, such as the one in [5]. This is because Mercury uses dummy variables called I/O states to represent states of the outside world, and models operations that interact with the outside world as predicates that consume and destroy the current I/O state and produce a new one. If two or more goals in a conjunction update I/O states, the algorithm in section 3 will ensure that the parallel version of the conjunction executes the same actions in the same order as the sequential version.

The system we have described is part of recent versions of Mercury, which are available for free download from the Mercury project's web page.

We would like to thank Tom Conway for implementing independent AND-parallelism.

References

1 Daniel Cabeza and Manuel V. Hermenegildo. Implementing distributed concurrent constraint execution in the CIAO system. In *Proceedings of the AGP'96 Joint Conference on Declarative Programming*, pages 67–78, 1996.

2 Thomas C. Conway. *Towards parallel Mercury*. PhD thesis, University of Melbourne, 2002.

3 Anderson Faustino da Silva and Vítor Santos Costa. The design of the YAP compiler: an optimizing compiler for logic programming languages. *J. UCS*, 12(7):764–787, 2006.

4 Robert H Halstead. Implementation of Multilisp: Lisp on a multiprocessor. In *Proceedings of the 1984 ACM Symposium on List and Functional Programming*, pages 9–17, Austin, Texas, 1984.

5 K. Muthukumar and M. Hermenegildo. Complete and efficient methods for supporting side effects in independent/restricted and-parallelism. In *Proceedings of ICLP 6*, pages 80–101, Lisbon, Portugal, June 1989.

6 Vijay Saraswat. Janus: a step towards distributed constraint programming. In *Proceedings of the Second North American Conference on Logic Programming*, pages 431–446, Austin, Texas, October 1990.

7 Kish Shen. Overview of DASWAM: exploitation of dependent AND-parallelism. *Journal of Logic Programming*, 29(1-3):245–293, 1996.

8 Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 26(1-3):17–64, October-December 1996.

9 Evan Tick. The deevolution of concurrent logic programming languages. *Journal of Logic Programming*, 23(2):89–123, May 1995.

10 Kazunori Ueda and Masao Morita. Moded flat GHC and its message-oriented implementation technique. *New Generation Computing*, 13(1):3–43, 1994.