TLP **11** (4–5): 575–591, 2011. © Cambridge University Press 2011
doi:10.1017/S1471068411000184

Estimating the overlap between dependent computations for automatic parallelization

PAUL BONE* and ZOLTAN SOMOGYI

Department of Computer Science and Software Engineering The University of Melbourne and National ICT Australia (NICTA), Australia
(*e-mail:* {pbone,zs}@csse.unimelb.edu.au)

PETER SCHACHTE

Department of Computer Science and Software Engineering The University of Melbourne, Australia
(*e-mail:* schachte@unimelb.edu.au)

Abstract

Researchers working on the automatic parallelization of programs have long known that too much parallelism can be even worse for performance than too little, because spawning a task to be run on another CPU incurs overheads. Autoparallelizing compilers have therefore long tried to use granularity analysis to ensure that they only spawn off computations whose cost will probably exceed the spawn-off cost by a comfortable margin. However, this is not enough to yield good results, because data dependencies may *also* limit the usefulness of running computations in parallel. If one computation blocks almost immediately and can resume only after another has completed its work, then the cost of parallelization again exceeds the benefit. We present a set of algorithms for recognizing places in a program where it is worthwhile to execute two or more computations in parallel that pay attention to the second of these issues as well as the first. Our system uses profiling information to compute the times at which a procedure call consumes the values of its input arguments and the times at which it produces the values of its output arguments. Given two calls that may be executed in parallel, our system uses the times of production and consumption of the variables they share to determine how much their executions would overlap if they were run in parallel, and therefore whether executing them in parallel is a good idea or not. We have implemented this technique for Mercury in the form of a tool that uses profiling data to generate recommendations about what to parallelize, for the Mercury compiler to apply on the next compilation of the program. We present preliminary results that show that this technique can yield useful parallelization speedups, while requiring nothing more from the programmer than representative input data for the profiling run.

KEYWORDS: automatic parallelism, program analysis, program optimization, Mercury

1 Introduction

When parallelizing Mercury (Somogyi *et al.* 1996) programs, the best parallelization opportunities occur where two goals take a significant and roughly similar time to

* Work supported by an Australian Postgraduate Award and a NICTA top-up scholarship.

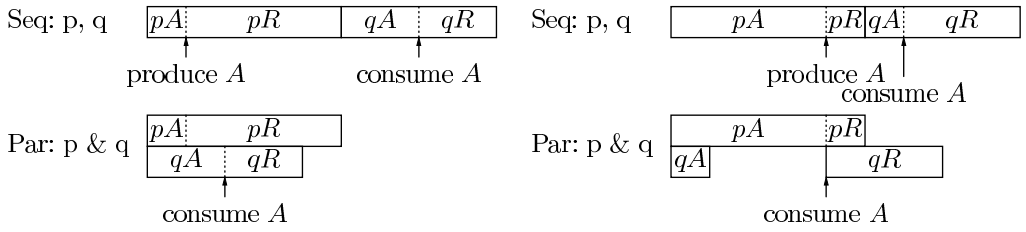

Fig. 1. Ample versus smaller parallel overlap between p and q.

```
map_foldl(_, _, [], Acc, Acc).
map_foldl(M, F, [X | Xs], Acc0, Acc) :-
    M(X, Y),
    F(Y, Acc0, Acc1),
    map_foldl(M, F, Xs, Acc1, Acc).
```

Fig. 2. map_foldl.

execute. Their execution time should be as large as possible so that the relative costs of parallel execution are small, and they should be independent to minimize synchronization costs. Unfortunately, goals expensive enough to be worth executing in parallel are rarely independent. For example, in the Mercury compiler itself, there are 53 conjunctions containing two or more expensive goals, but in only one of those conjunctions are the expensive goals independent. This is why Mercury supports the parallel execution of dependent conjunctions. The Mercury compiler wraps shared variables within a *future* (Wang and Somogyi 2011), to ensure that the *consumer* of the variable is blocked until the *producer* makes the variable available.

Dependent parallel conjunctions may differ in the amount of parallelism they have available. Consider a parallel conjunction with two similarly-sized conjuncts, p and q, that share a single variable A. If p produces A late but q consumes it early, as shown on the right side of Figure 1, there will be little parallelism, because q will be blocked soon after it starts, and will be unblocked only when p is about to finish. Alternatively, if p produces A early and q consumes it late, as shown on the left side of in Figure 1, we would get much more parallelism. The top part of each scenario shows the execution of the sequential form of the conjunction.

Unfortunately, in real Mercury programs, almost all conjunctions are dependent conjunctions, and in most of them, shared variables are produced very late and consumed very early. Parallelizing them would therefore yield slowdowns instead of speedups, because the overheads of parallel execution would far outweigh the benefits. We want to parallelize only conjunctions in which any shared variables are produced early, consumed late, or (preferably) both. The first purpose of this paper is to show how one can find these conjunctions.

The second purpose is to find the best way to parallelize these conjunctions. Consider the map_foldl predicate in Figure 2. The body of the recursive clause has three conjuncts. We could make each conjunct execute in parallel, or we could execute two conjuncts in sequence (either the first and second, or the second and the third), and execute that sequential conjunction in parallel with the remaining conjunct. In this case, there is little point in executing the higher order calls to the map and fold predicates in parallel with one another, because in virtually all cases,

the map predicate will generate `Y` very late and the fold predicate will consume `Y` very early. However, executing the sequential conjunction of the map and fold predicates in parallel with the recursive call *will* be worthwhile if the map predicate is time-consuming, because this implies that a typical recursive call will consume its fourth argument late; the recursive call processing the second element of the list will have significant execution overlap with its parent processing the first element of the list even if (as is typical) the fold predicate generates `Acc1` very late. (This is the kind of computation that Reform Prolog (Bevemyr *et al.* 1993) was designed to parallelize.)

The structure of this paper is as follows. Section 2 gives the background needed for the rest of the paper. Section 3 outlines our general approach, which the later sections fill in. Section 4 describes our algorithm for calculating the execution overlap between two or more dependent conjuncts. A conjunction with more than two conjuncts can be parallelized in several different ways; Section 5 shows how we choose the best way. Section 6 evaluates how our system works in practice on some example programs, and Section 7 concludes with comparisons to related work.

2 Background

2.1 Mercury

The abstract syntax of the part of Mercury relevant to this paper is

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
	$\mid$ (*if* G_c *then* G_t *else* G_e) $\mid$ *not* G	if-then-elses and negations
	$\mid$ *some* $[x_1,\ldots,x_n]$ G	quantifications

The atomic constructs of Mercury are unifications (which the compiler breaks down until they contain at most one function symbol each), plain first-order calls, and higher-order calls. The composite constructs include sequential and parallel conjunctions, disjunctions, if-then-elses, negations and existential quantifications. These should all be self-explanatory. A switch is a disjunction in which each disjunct unifies the same bound variable 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 as the argument. If output, the caller must pass a distinct free variable, which the predicate will instantiate to a ground term. It is possible for a predicate to have more than one mode; we call each mode of a predicate a *procedure*. The compiler generates separate code for each procedure of a predicate. The mode checking pass of 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 goals that use this value (the *consumers*). This means that for each variable in each procedure, the compiler knows exactly where that variable gets grounded.

Each procedure and goal has a determinism, which may put upper and lower bounds on the number of its possible solutions (in the absence of infinite loops and exceptions). A determinism may impose an upper bound of one solution, and it may impose a lower bound of one solution. *det* procedures succeed exactly once; *semidet* procedures succeed at most once; *multi* procedures succeed at least once; *nondet* procedures may succeed any number of times.

2.2 Parallelism in mercury

The Mercury runtime system has a construct called a Mercury *engine* that represents a virtual CPU. Each engine is independently schedulable by the OS, usually as a POSIX thread. The number of engines that a parallel Mercury program will allocate on startup is configurable by the user, but it defaults to the actual number of CPUs. Another construct in the Mercury runtime system is a *context*, which represents a computation in progress. An engine may be idle, or it may be executing a context; a context can be running on an engine, or it may be suspended. When a context finishes execution, its storage is put back into a pool of free contexts. Following (Marlow *et al.* 2009), we use *sparks* to represent goals that have been spawned off but whose execution has not yet been started.

The only parallel construct in Mercury is parallel conjunction, which is denoted (G_1 & ... & G_n). All the conjuncts must be deterministic, that is, they must all have exactly one solution. This restriction greatly simplifies the implementation, because it guarantees that there can never be any need to execute (G_2 & ... & G_n) multiple times, just because G_1 has succeeded multiple times. (Any local backtracking inside G_1 will not be visible to the other conjuncts; bindings made by det code are never retracted.) However, this is not a significant limitation. Because the design of Mercury strongly encourages deterministic code, in our experience, about 75% to 85% of all Mercury procedures are det, and most programs spend an even greater fraction of their time in det code. Existing algorithms for executing nondeterministic code in parallel have very significant overheads, generating slowdowns by integer factors. Thus we have given priority to parallelizing deterministic code, which we can do with *much* lower overhead.

The Mercury compiler implements (G_1 & G_2 & ... & G_n) by creating a data structure representing a barrier, and then spawning off (G_2 & ... & G_n) as a spark. Because (G_2 & ... & G_n) is itself a conjunction, it is handled the same way: the context executing it first spawns off (G_3 & ... & G_n), and then executes G_2 itself. Eventually, the spawned-off remainder of the conjunction consists only of the final conjunct, G_n, and the context just executes it. The code of each conjunct synchronizes on the barrier once it has completed its job. When all conjuncts have done so, the original context will continue execution after the parallel conjunction.

Mercury's mode system allows a parallel conjunct to consume variables that are produced by conjuncts to its left, but not to its right. This guarantees the absence of circular dependencies and hence the absence of deadlocks between the conjuncts,

but it does allow a conjunct to depend on data that is yet to be computed by a conjunct running in parallel. We handle these dependencies through a source-to-source transform (Wang and Somogyi 2011). The compiler knows which variables are produced by one parallel conjunct and consumed by another. For each of these shared variables, it creates a data structure called a *future* (Halstead 1984). When the producer has finished computing the value of the variable, it puts the value in the future and signals its availability. When a consumer needs the value of the variable, it waits for this signal, and then retrieves the value from the future.

To minimize waiting, the compiler pushes signal operations as far to the left into the producer conjunct as possible, and it pushes wait operations as far to the right into each of the consumer conjuncts as possible. This means not only pushing them into the body of the predicate called by the conjunct, but also into the bodies of the predicates they call, with the intention being that each signal is put immediately after the primitive goal that produces the value of the variable, and each wait is put immediately before the leftmost primitive goal that consumes the value of the variable. Because the compiler has complete information about which goals produce and consume which variables, the only things that can stop the pushing process are higher order calls and module boundaries: the compiler cannot push a wait or signal operation into code it cannot identify or cannot access.

3 Our general approach

We want to find the conjunctions in the program whose parallelization would be the most profitable. This means finding the conjunctions with conjuncts whose execution cost exceeds the spawning-off cost by the highest margin, and whose interdependencies, if any, allow their executions to overlap the most. Essentially, the greater the margin by which the likely runtime of the parallel version of a conjunction beats the likely runtime of the sequential version, the more beneficial parallelizing that conjunction will be.

To compute this likely benefit, we need information both about the likely cost of calls and the execution overlap allowed by their dependencies. Our system therefore asks programmers to follow this sequence of actions after they have tested and debugged the program.

1. Compile the program with options asking for profiling.
2. Run the program on a representative set of input data. This will generate a profiling data file.
3. Invoke our feedback tool on the profiling data file. This will generate a parallelization advice file.
4. Compile the program for parallel execution, specifying the parallelization advice file. The advice file tells the compiler *which* sequential conjunctions to convert to parallel conjunctions, and exactly *how*. For example, `c1, c2, c3` can be converted into `c1 & (c2, c3)`, into `(c1, c2) & c3`, or into `c1 & c2 & c3`, and as the `map_foldl` example shows, the speedups you get from them can be strikingly different.

It is up to the programmer using our system to select training input for the profiling run in step 2. Obviously, programmers should pick input that is as representative as possible, but the recommended parallelization can be useful even for input data that is quite different from the training input. The main focus of this paper is on step 3; we give the main algorithms used by the feedback tool.

Our feedback tool is an extension of the Mercury deep profiler. One of our modifications gives the deep profiler access to the relevant parts of the compiler's representation of the program. This includes a representation of each procedure body, and for each atomic subgoal (call or unification) within each body, the set of variables bound by that subgoal. Another modification records how many times execution reaches each point in the program. As we will see in Section 4, we need this information to calculate the likely speedup from parallelizing a conjunction.

Our feedback tool looks for parallelization opportunities by doing a depth-first search of the call tree recorded in the profiling data file. It explores the subtree below a node in the call tree only if the overall cost of the call is greater than a configurable threshold, and if the amount of parallelism it has found at and above that node is below another configurable threshold. The first test lets us avoid looking at code that would take more work to spawn off than to execute, whereas the second test lets us avoid creating more parallel work than the target machine can handle.

For each procedure in the call tree, we search its body for conjunctions that contain two or more calls with execution times above a configurable threshold. To parallelize the conjunction, its conjuncts have to be partitioned, each partition being one conjunct in the parallel conjunction. In most cases, this can be done in several different ways. We can use the algorithms of Section 4 to compute the expected parallel execution time of each partition; these algorithms take into account the runtime overheads of parallel execution. We use the algorithms of Section 5 to generate the set of partitions whose performance we want to evaluate. If the best-performing parallelization we find shows a nontrivial speedup over sequential execution, we remember that we want to perform that parallelization on this conjunction. If the depth first search later finds some of the conjuncts to have parallelizable code inside them, we revisit this conjunction, this time using updated data about the cost of those conjuncts. Otherwise, we add a recommendation to perform the selected parallelization to the feedback advice we generate for the compiler.

An important benefit of profile-directed parallelization is that because programmers do not annotate the source program, it can be re-parallelized easily after a change to the program obsoletes some old parallelization opportunities and creates others. Nevertheless, if programmers want to parallelize some conjunctions manually, they can do so: our system will not override the programmer.

4 Calculating the overlap between dependent conjuncts

As we can see from the difference between the two sides of Figure 1, figuring out the overlap in the parallel executions of two dependent conjuncts requires knowing, for each of the variables they share, when that variable is generated by the first conjunct and when it is first consumed by the second conjunct. Our algorithms for

computing these times are considerably simplified by the Mercury mode system and by the fact that we only parallelize deterministic goals.

The profiling data gives us both the total execution time of each conjunct and its number of invocations; the ratio of the two is the expected execution time for each invocation. The algorithm for computing the expected production time of a given shared variable looks at the form of the conjunct:

- If the goal is a unification, the expected production time is zero, because our unit of time is the time between two successive calls.
- If the goal is a first order call, we recurse on the body of the callee.
- If the goal is a higher order call, the expected production time is the cost of the call, because the compiler cannot (yet) insert the signaling of the future into the callee's body.
- If the goal is a conjunction $G_1,\ldots,G_n$, and the variable is generated by G_k, then we add up the total time taken by $G_1,\ldots,G_{k-1}$, and add the sum to the result of invoking the algorithm recursively on G_k.
- If the goal is a switch, we invoke the algorithm recursively on each switch arm, and compute a weighted average of the results, with the weights being the arms' entry counts.
- If the goal is an if-then-else, we need the weighted average of the two possible cases: the variable being generated by the then arm versus the else arm. (It cannot be generated by the condition: variables generated by the condition are visible only from the then-arm.) To find the first number, we invoke the algorithm on the then-arm, and add the result to the time taken by the condition. To find the second, we invoke the algorithm on the else-arm, and add the result to the expected time taken by the condition when it fails. To compute this, we use a version of this algorithm that weights the time taken by each conjunct in any inner conjunction by the probability of its execution, which we know by comparing its execution count with the count of the number of times the condition was entered.
- The goal cannot be a negation, because negated goals cannot bind variables.
- The goal cannot be a disjunction, because disjunctions cannot produce variables visible from det code. (To transition from nondet or multi code to det code, the programmer must quantify away the outputs of the nondet code.)
- If the goal is a quantification, then the inner goal must be det, in which case we invoke the algorithm recursively on it. If the inner goal were not det, then the outer quantification goal could be det only if the inner goal did not bind any variables visible from the outside.

Using the weighted average for switches and if-then-elses is meaningful because the Mercury mode system dictates that if one arm of a switch or if-then-else generates a variable, then they *all* must do so.

The algorithm we use for computing the time at which a shared variable is first consumed by the second conjunct is similar to this one, the main differences being that negated goals, conditions and disjunctions are allowed to consume variables,

and some arms of a switch or if-then-else may consume a variable even if other arms do not. Suppose the first appearance of the variable (call it X) in a conjunction $G_1,\ldots,G_n$ is in G_k, and G_k is a switch. If X is consumed by some switch arms and not others, then on some execution paths, the first consumption of the variable may be in G_k (a), on some others it may be in $G_{k_1},\ldots,G_n$ (b), and on some others it may not be consumed at all (c). For case (a), we compute the average time of first consumption by the consuming arms, and then compute the weighted average of these times, with the weights being the probability of entry into each arm, as before. For case (b), we compute the probability of entry into arms which do *not* consume the variable, and multiply the sum of those probabilities by the weighted average of those arms' execution time *plus* the expected consumption time of the variable in G_{k+1}, …, G_n. For case (c) we pretend X is consumed at the very end of the goal, and then handle it in the same way as (b). This is because for our overlap calculations, a goal that does not consume a variable is equivalent to a goal that consumes it at the end of its execution.

Suppose a candidate parallel conjunction has two conjuncts p and q, and their execution times in the original, sequential conjunction p,q, are $SeqTime_p$ and $SeqTime_q$. Suppose SV_i are the variables shared between them, and for each SV_i, the time at which p produces it is $ProdTime_{pi}$, and the time at which q consumes it is $ConsTime_{qi}$.

If we denote the execution times of the conjuncts in the parallel conjunction p & q as $ParTime_p$ and $ParTime_q$, then the expected speedup from parallelizing the original sequential conjunction is $Speedup = SeqTime/ParTime$, where $SeqTime = SeqTime_p + SeqTime_q$, and $ParTime = SpawnOverhead + max(ParTime_p, ParTime_q)$. The profile gives us $SeqTime_p$ and $SeqTime_q$, and if we ignore overheads for now (we will come back to them later), then $ParTime_p$ will always be equal to $SeqTime_p$. The main task of computing the speedup therefore consists of computing $ParTime_q$; as we saw in Figure 1, this will differ from $SeqTime_q$ whenever q needs to wait for p to produce a shared variable.

Figure 3 shows a simplified version of the algorithm we use to compute the expected execution time of a conjunction when its conjuncts are executed in parallel, assuming an unlimited number of CPUs. The inputs of the algorithm are `Conjs`, the conjuncts themselves, and `ProdConsList`, which gives, for each conjunct, the list of its input and output variables, together with the times at which, in a sequential execution, they are respectively first consumed or produced. The times are relative to the start of the execution of the relevant conjunct.

The main task of the algorithm is to divide the execution times of all the conjuncts into chunks and keep track of when those chunks can execute. The execution time of `Conj_i` has one chunk (`Duration_ij`) for each of `Conj_i`'s shared variables that ends at the time at which that variable is produced or first consumed, and there is one chunk (`DurationRest_i`) at the end, during which the call may produce nonshared variables. Figure 1 shows that the production of A divides p into two chunks, pA and pR, whereas the consumption of A divides q into qA and qR.

The algorithm processes the chunks in order, and keeps track of the sequential and parallel execution times of the chunks so far. When a chunk of `Conj_i` ends with

```
find_par_time(Conjs) returns TotalParTime:
  N := length(Conjs)
  ProdTimeMap := empty
  TotalParTime := 0
  for i in 1 to N:
    CurSeqTime := 0
    CurParTime := 0
    sort ProdConsList_i on Time_ij
    forall (Var_ij, Time_ij) in ProdConsList_i:
      Duration_ij := Time_ij - CurSeqTime
      CurSeqTime := CurSeqTime + Duration_ij
      if Conj_i produces Var_ij:
        CurParTime := CurParTime + Duration_ij
        ProdTimeMap[Var_ij] := CurParTime
      else Conj_i must consume Var_ij:
        ParWantTime := CurParTime + Duration_ij
        CurParTime := max(ParWantTime, ProdTimeMap[Var])
    DurationRest_i := SeqTime_i - CurSeqTime
    CurParTime := CurParTime + DurationRest_i
    TotalParTime := max(TotalParTime, CurParTime)
```

Fig. 3. Dependent parallel conjunction algorithm.

the production of a variable, we record when that variable is produced, and the next chunk can start executing immediately. When a chunk ends with the consumption of a variable, then in the *sequential* version of `Conj_i` the next chunk can also execute immediately, because the values of all the input variables will be available when it starts, but in the *parallel* version, the variable may not have been produced yet. If it has, then `Conj_i` does not need to wait for it; the left side of Figure 1 shows this case. However, it is also possible that it has not. In that case, `Conj_i` will suspend on the variable, and will resume only when its producer signals that it is available; the right side of Figure 1 shows this case. Note that `Var_ij` will always be in `ProdTimeMap` when we look for it, because the Mercury mode system reorders conjunctions to put the producer of each variable before all its consumers.

The version of this algorithm we have actually implemented is a bit longer than the one in Figure 3, because it also accounts for several forms of overhead:

- Creating a spark and adding it to a work queue has a cost. Every conjunct but the last conjunct incurs this cost to create the spark for the rest of the conjunction.
- It takes some time to take a spark off a spark queue, create or reuse a context for it, and start its execution. Every parallel conjunct that is not the first incurs this delay before it starts running.
- The signal and wait operations have a cost.
- It takes some time to wake up a context when its wait operation succeeds.
- It takes time for each conjunct to synchronize on the barrier when it has finished its job.

We can account for every one of these overheads by adding the estimated cost of the relevant operation to `CurParTime` at the right point in the algorithm.

In many cases, the conjunction given to the algorithm shown in Figure 3 will contain a recursive call. In such cases, the speedup computed by the algorithm reflects the speedup we can expect to get when the recursive call calls the *original, sequential* version of the predicate. When the recursive call calls the parallelized version, we can expect a similar saving (absolute time, not ratio) on *every* recursive invocation. How this affects the expected speedup of the top level call depends on the structure of the recursion. For the most common recursion structure, singly recursive predicates like `map_foldl`, calculating the expected speedup of the top level call is easy, because we can compute the average depth of recursion from the relative execution counts of the base and recursive cases. For some less common structures, such as doubly recursive predicates like `quicksort`, it is a bit harder, and for irregular structures in which different execution paths contain different numbers of recursive calls, the profiling data gathered by the current version of the Mercury profiler contains insufficient information to allow our system to determine the expected speedup. However, an automated survey of the programs handled by our feedback tool shows that such predicates are rare; our system can compute the expected recursion depth and therefore the expected speedup for virtually all candidates for parallelization.

So far, we have assumed an unlimited number of CPUs, which is of course unrealistic. If the machine has e.g. four CPUs, then the prediction of any speedup higher than four is obviously invalid. Less obviously, even a predicted overall speedup of less than four may depend on more than four conjuncts executing all at once at *some* point. We have not found this to be a problem yet. If and when we do, we intend to extend our algorithm to keep track of the number of active conjuncts in all active time periods. Then if a chunk of a conjunct wants to run in a time period when all CPUs are predicted to be already busy executing previous conjuncts, we assume that the start of that chunk is delayed until a CPU becomes free.

The limited number of CPUs also means that there is a limit to how much parallelism we actually *want*. The spawning off of every conjunct incurs overhead, but these overheads do not buy us anything if all CPUs are already busy. That is why our system supports *throttling*. If a conjunction being parallelized contains a recursive call, then the compiler can be asked to replace the original sequential conjunction not with the parallel form of the conjunction, but with an if-then-else. The condition of this if-then-else will test at runtime whether spawning off a new job is a good idea or not. If it is, we execute the parallelized conjunction, but if it is not, we execute the original sequential conjunction. The condition is obviously a heuristic. If the heuristic allows the list of runnable jobs to become empty, then we will not have any work to give to a CPU that finishes its task and becomes available. On the other hand, if the heuristic allows the list of runnable jobs to become too long, then we incur the overheads of spawning off some jobs unnecessarily. Currently, on machines with N CPUs, we prefer to have a total of M running and runnable jobs where $M > N$, so our heuristic stops spawning attempts iff the queue already has M entries. Our current system by default sets M to be 32 for $N = 4$, though users can easily override this.

```
global NumEvals := 0
find_best_partition(InitPartition, InitTime, LaterConjs)
    returns <FinalTime, FinalPartitionSet>:
  switch on LaterConjs:
  when LaterConjs = []:
    return <InitTime, {InitPartition}>
  when LaterConjs = [Head | Tail]:
    Extend := all_but_last(InitPartition) ++ [last(InitPartition) ++ [Head]]
    AddNew := InitPartition ++ [Head]
    ExtendTime := find_par_time(Extend)
    AddNewTime := find_par_time(AddNew)
    NumEvals := NumEvals + 2
    if ExtendTime < AddNewTime:
      BestExtendSoln := find_best_partition(Extend, ExtendTime, Tail)
      let BestExtendSoln be <BextExTime, BestExPartSet>
      if NumEvals < PreferLinearEvals:
        BestAddNewSoln := find_best_partition(AddNew, AddNewTime, Tail)
        let BestAddNewSoln be <BestANTime, BestANPartSet>
        if BestExTime < BestANTime:
          return BestExtendSoln
        else if BestExTime = BestANTime:
          return <BextExTime, BestExPartSet union BestANPartSet>
        else:
          return BestAddNewSoln
      else:
        return BestExtendSoln
    else:
      <symmetric with the then case>
```

Fig. 4. Search for the best parallelization.

5 Choosing how to parallelize a conjunction

A conjunction with $n > 2$ conjuncts can be converted into several different parallel conjunctions. Converting all the commas into ampersands (e.g. `c1, c2, c3` into `c1 & c2 & c3`) yields the most parallelism. Unfortunately, this will often be *too* much parallelism, because in practice many conjuncts are unifications and arithmetic operations whose execution takes very few instructions. Executing such conjuncts in their own threads costs far more in overheads than they save by running in parallel. Therefore in most cases, we want to create parallel conjunctions with $k < n$ conjuncts, each consisting of a contiguous sequence of one or more of the original sequential conjuncts, effectively partitioning the original conjuncts into groups.

For any conjunction to be worth parallelizing, it should contain two or more expensive goals. Our main algorithm (Figure 4 works on the list of conjuncts from the first expensive goal to the last. This will be the middle of original conjunction, with (possibly empty) lists of cheap goals before it and after it. Our initial search assumes that the set of conjuncts in the parallel conjunction we want to create is exactly the set of conjuncts in the middle. A postprocessing step then removes that assumption.

If the middle sequence has n conjuncts, then there are $n - 1$ AND operations between them, each of which can be either sequential or parallel. There are then 2^{n-1} combinations, all but one of which are parallelizations. That is a large space to search for the *best* parallelization, and it would be larger still if we allowed code reordering, that is, parallel conjuncts consisting of a *non*contiguous sequence of the original

conjuncts. We explore this space with a search algorithm, `find_best_partition`, which we invoke with the empty list as `InitPartition`, zero as `InitTime`, and the list of middle conjuncts as `LaterConj`. `InitPartition` expresses a partition of an initial sequence of the middle goals into parallel conjuncts whose estimated execution time is `InitTime`, and considers whether it is better to add the next middle goal to the last existing parallel conjunct (`Extend`), or to put it into a new parallel conjunct (`AddNew`). It explores extensions of the better of the resulting partitions first. If the search is still under the limit on the number of evaluations, it explores the worse partition as well, which is an exponential search. When it hits the limit, it switches to a linear search; we explore the more promising partition first to make this search more effective. (This limit ensures that the algorithm runs in reasonable time.) The algorithm returns a set of equal best parallelizations so far, "best" being measured by a version of the algorithm in Figure 3 that computes the estimated parallel execution time *including* overheads.

There are some simple ways to improve this algorithm.

- Most invocations of `find_par_time` specify a partition that is an extension of a partition processed in the recent past. In such cases, `find_part_time` should do its task incrementally, not from scratch.
- If the expected execution time for the candidate partition currently being considered is already greater than the fastest existing complete partition, we can stop exploring that branch; it cannot lead to a better solution.
- Sometimes consecutive conjuncts do things that are obviously a bad idea to do in parallel, such as building a ground term. The algorithm should treat these as a single conjunct.

At the completion of the search, we select one of the equal best parallelizations, and postprocess it to adjust both edges. Suppose the best parallel form of the middle goals is P_1 & … & P_p, where each P_i is a sequential conjunction. We compare the execution time of P_1 & … & P_p with that of P_1, (P_2 & … & P_p). If the former is slower, which can happen if P_1 produces its outputs at its very end and the other P_i consume those outputs at their start, then we conceptually move P_1 out of the parallel conjunction (from the "middle" part of the conjunction to the "before" part). We keep doing this for P_2, P_3 etc until either we find a goal worth keeping in the parallel conjunction, or we run out of conjuncts. We also do the same thing at the other end of the middle part. This process can shrink the middle part.

In cases where we do not shrink an edge, we can consider expanding that edge. Normally, we want to keep cheap goals out of parallel conjunctions, because more conjuncts tends to mean more shared variables and thus more synchronization overhead, but sometimes this consideration is overruled by others. Suppose the goals before the conjuncts in P_1 & … & P_p in the original conjunction were B_1, …, B_b and the goals after it A_1, …, A_a, and consider A_1 after P_p. If P_p finishes before the other parallel conjuncts, then executing A_1 just after P_p in P_p's context may be effectively free: the last context could still arrive at the barrier at the same time, but this way, A_1 would have been done by then. Now consider B_b before P_1. If P_1 finishes before the other parallel conjuncts, *and* if none of the other conjuncts

wait for variables produced by P_1, then executing B_b in the same context as P_1 can be similarly free.

We loop from $i = b$ down towards $i = 1$, and check whether including $B_i, \ldots, B_b$ at the start of P_1 is improvement. If not, we stop; if it is, we keep going. We do the same from the other end. The stopping points of the loops of the contraction and expansion phases dictate our preferred parallel form of the conjunction, which (if we shrunk the middle at the left edge and expanded it at the right) will look something like $B_1, \ldots, B_b, P_1, \ldots P_k, (P_{k+1}$ & ... & P_{p-1} & $(P_p, A_1, \ldots, A_j)), A_{j+1}, \ldots, A_a$. If this preferred parallelization is better than the original sequential version of the conjunction by at least 1 then we include a recommendation for its conversion to this form in the feedback file we create for the compiler.

6 Performance results

We tested our system on three benchmark programs: matrix multiplication, a mandelbrot image generator and a raytracer. Matrixmult has abundant independent AND-parallelism. Mandelbrot uses the actual `map_foldl` predicate from Figure 2 to iterate over rows of pixels. Raytracer does not use `map_foldl`, but does use a similar code structure to perform a similar task. This is not an accident: *many* predicates use this kind of code structure, partly because programmers in declarative languages often use accumulators to make their loops tail recursive.

We ran all three programs with one set of input parameters to collect profiling data, and with a *different* set of input parameters to produce the timing results in the following table. All tests were run on a Dell Optiplex 755 PC with a 2.4 GHz Intel Core 2 Quad Q6600 CPU running Linux 2.6.31. Each test was run 10 times; we discarded the highest and lowest times, and averaged the rest.

Each group of three rows reports the results for one benchmark. The first column shows the benchmark name, the runtime of the program when compiled for sequential execution, and its runtime when compiled for parallel execution but without enabling auto-parallelization. This shows the overhead of support for parallel execution when it does not buy any benefits. We auto-parallelized each program three different ways: executing expensive goals in parallel only when they are independent ("indep"); even if they are dependent, regardless of overlap ("naive"); and even if they are dependent, but only if they have good overlap ("overlap"). The last four columns give the runtime in seconds of each of these versions of the program on one to four CPUs, with speedups compared to the sequential version.

The parallel version of the Mercury system needs to use a real machine register to point to thread-specific data, such as each engine's abstract machine registers. On ×86s, this leaves only one real register for the Mercury abstract machine, so compiling for parallelism but not using it yields a slowdown ranging from 5% on mandelbrot to 25% on matrixmult. (We observe such slowdowns for other programs as well.) On one CPU, autoparallelization gets only this slowdown, plus the (small) additional overheads of all the parallel conjunctions that cannot get any parallelism.

The parallelism in the main predicate of matrixmult is independent, Overlap parallelizes the program the same way as indep, so it gets the same speedup. The numbers look different for three CPUs, but all the runs for both versions actually

Program	Par	1 CPU	2 CPUs	3 CPUs	4 CPUs
matrixmult	indep	14.6 (0.75)	7.5 (1.47)	7.0 (1.66)	5.2 (2.12)
seq 11.0	naive	14.6 (0.75)	7.6 (1.45)	5.2 (2.12)	5.2 (2.12)
par 14.6	overlap	14.6 (0.75)	7.5 (1.47)	6.2 (1.83)	5.2 (2.12)
mandelbrot	indep	35.2 (0.95)	35.1 (0.95)	35.2 (0.95)	35.3 (0.95)
seq 33.4	naive	35.4 (0.94)	18.0 (1.86)	12.1 (2.76)	9.1 (3.67)
par 35.2	overlap	35.6 (0.94)	17.9 (1.87)	12.1 (2.76)	9.1 (3.67)
raytracer	indep	26.2 (0.87)	26.3 (0.86)	26.1 (0.87)	26.2 (0.87)
seq 22.7	naive	25.3 (0.90)	16.0 (1.42)	11.2 (2.03)	9.4 (2.42)
par 26.5	overlap	25.1 (0.90)	16.0 (1.42)	11.2 (2.03)	9.4 (2.42)

took either 5.2 or 7.5 s, depending (we think) on which way the OS arranged the engines across the two CPU die of the Q6600; the indep version just happened to get the 7.5 s arrangement fewer times. For naive, all the runs just happened to take 5.2 s, even though naive creates a worse parallelization than either indep or overlap: during the expansion phase we described in Section 5, it includes an extra goal in the first of the parallel conjuncts; this makes the conjunction dependent, which adds some overhead. Naive also executes the code that does the matrix multiplication in parallel with the goals that create its inputs, which also adds overhead without speedup. These overheads are too small to affect the results.

In mandelbrot and raytracer, all the parallelism is dependent, which is why indep gets no speedup for them. For mandelbrot, naive and overlap get speedups that are as good as one can reasonably expect: $35.2/9.1 = 3.87$ on four CPUs over the one CPU case. For matrixmult and raytracer, the speedups they get, 2.12 and 2.42 on four CPUs, also turn out to be pretty good when one takes a closer look.

For matrixmult, the bottleneck is almost certainly CPU-memory bandwidth. Each step in this program does only one multiply and one add (both integer) before creating a new cell on the heap and filling it in. On current CPUs, the arithmetic takes much less time than the memory writes, and because the new cells are never accessed again, caches do not help, which makes it easy to saturate the memory bus, even when using only three CPUs.

The raytracer is very memory-allocation-intensive, because it does lots of FP arithmetic, and the Mercury backend we are using always boxes floating point numbers, so each floating point operation requires the creation of a new cell on the heap. Because of this, memory bandwidth may also be an issue for it, but its bigger problem is GC; whereas GC takes only about 5% of the runtime when run on one CPU, it takes almost 40% of the runtime when run on four CPUs, even though we used four marker threads. (For fairness, we used four marker threads regardless of how many CPUs the Mercury code used.) Given this fact, the best speedup we can hope for is $(4 \times 0.6 + 0.4)/(0.6 + 0.4) = 2.8$, and we do come pretty close to that.

GC becomes more expensive with more CPUs not only because of increased contention, but also because the GC has more work to do: with more contexts

being spawned, there are more stacks for it to scan. We have tested versions of the raytracer in which each spawned-off goal computed the pixels for several rows, not just one, and these versions yield speedups of about 3.3 on four CPUs. These versions spawn many fewer contexts, thus putting much less load on the GC. This shows that program transformations that cause more work to be done in each context are likely to be a promising area for future work.

Most small programs like these benchmarks have only one loop that dominates their runtime. In all three of these benchmarks, and in many others, the naive and overlap methods will parallelize the same loops, and usually the same way; they tend to differ only in how they parallelize code that executes much less often (typically only once) whose effect is lost in the noise. The raw timings show a great deal of variability: we have seen two consecutive runs of the same program on the same data differ in their runtime by as much as 15%. Some of this variability remains even after filtering and averaging.

To see the difference between naive and overlap, we need to look at larger programs. Our standard large test program is the Mercury compiler, which contains 53 conjunctions with two or more expensive goals. Of these, 52 are dependent, and only 31 have an overlap that leads to a predicted local speedup of more than 1%, our default threshold. Our algorithms can thus prevent the unproductive parallelization of $53 - 31 = 22$ of these conjunctions. Unfortunately, programs that are large and complex enough to show a performance effect from this saving also tend to have large components that cannot be profitably parallelized with existing techniques, which means that (due to Amdahl's law) our autoparallelization system cannot yield overall speedups for them yet.

On the bright side, our feedback tool generates feedback files in less than a second from the profiles of small programs like these benchmarks, and in only a minute or two even from much larger profiles. The extra time taken by the Mercury compiler when it follows the recommendations in feedback files is so small that it is not noticeable.

7 Related work and conclusion

Mercury's strong mode and determinism systems greatly simplify the parallel execution of logic programs. The information gathered by semantic analysis in Mercury makes it easy to solve most of the problems faced by the designers of parallel versions of Prolog and Prolog-like languages. These include testing the independence of goals in systems that support only independent AND-parallelism and discovering producer–consumer relationships in systems that also support dependent AND-parallelism, such as Gras and Hermenegildo (2009). They also make it possible to *avoid* having to solve some tough problems, the main example being how to execute nondeterministic conjuncts in parallel without excessive overhead.

Most research in parallel logic programming so far has focused on trying to solve these problems of getting parallel execution to *work* well, with only a small fraction trying to find when parallel execution would actually be *worthwhile*. Almost all previous work on automatic parallelization has focused on granularity control: parallelizing only computations that are expensive enough to make parallel execution

worthwhile (Lopez *et al.* 1996; Harris and Singh 2007), and properly accounting for the overheads of parallelism itself (Shen *et al.* 1999). Most of the rest has tried to find opportunities to exploit independent AND-parallelism during the execution of otherwise-dependent conjunctions (Muthukumar *et al.* 1999; Casas *et al.* 2007).

Our experience with our feedback tool shows that for Mercury programs, this is far from enough. For most programs, it finds enough conjunctions with two or more expensive conjuncts, but almost all are dependent, and, as we mention in Section 6, many of these have too little overlap to be worth parallelizing.

We know of only three attempts to estimate the overlap between parallel computations. One was in the context of speculative execution in imperative programs. Given two successive blocks of instructions, von Praun *et al.* (2007) decides whether the second block should be executed speculatively based on the difference between the addresses of two instructions, one that writes a value to a register and one that reads from that register. This works if instructions take a bounded time to execute, but in the presence of call instructions this heuristic will not be at all accurate.

Another attempt was a previous auto-parallelization project for Mercury (Tannier 2007). This used the number of shared variables between conjuncts as a measure of the dependency between goals, and as a predictor of the likely overlap. Although two conjuncts are indeed less likely to have useful parallel overlap if they have more shared variables, we have found this heuristic too inaccurate to be useful.

The most closely related work to ours generated parallelism annotations for the ACE and/or-parallel system (Pontelli *et al.* 1997). This system used, much as we do, estimates of the costs of calls and of the times at which variables are produced and consumed. However, it produced its estimates through static analysis of the program. This can work for small programs, where the call trees of the relevant calls can be quite small and regular. In large programs, the call trees of the expensive calls are almost certain to be both tall and wide, with a huge gulf between best-case and worst-case behavior. Using profiling data is the only way for an automatic parallelization system to find out what the *typical* behavior of such calls is.

Our system's predictions of the likely speedup from parallelizing a conjunction are also fallible, because they currently ignore several relevant issues, including cache effects and the effects of bottlenecks such as CPU-memory buses and stop-the-world garbage collection. However, our system seems to be a sound basis for such further refinements. In the future, we plan to support parallelization as a specialization: applying a specific parallelization only when a predicate is called from a specific parent, grandparent or other ancestor. We also plan to modify our feedback tool to accept several profiling data files, with a priority scheme to resolve any conflicts.

We thank the rest of the Mercury team, and Tom Conway and Peter Wang in particular, for creating the infrastructure we build upon, and the anonymous referees for their suggestions.

References

Bevemyr, J., Lindgren, T., and Millroth, H. 1993. Reform Prolog: The language and its implementation. In *Proceedings of the Tenth International Conference on Logic Programming*. Budapest, Hungary, 283–298.

CASAS, A., CARRO, M., AND HERMENEGILDO, M. V. 2007. Annotation algorithms for unrestricted independent AND-parallelism in logic programs. In *Proceedings of the 17th International Symposium on Logic-based Program Synthesis and Transformation*. Lyngby, Denmark, 138–153.

GRAS, D. C. AND HERMENEGILDO, M. V. 2009. Non-strict independence-based program parallelization using sharing and freeness information. *Theoretical Computer Science 410*(46), 4704–4723.

HALSTEAD, R. H. 1984. Implementation of MultiLisp: Lisp on a multiprocessor. In *Proceedings of the 1984 ACM Symposium on List and Functional Programming*. Austin, Texas, 9–17.

HARRIS, T. AND SINGH, S. 2007. Feedback directed implicit parallelism. *SIGPLAN Notices 42*(9), 251–264.

LOPEZ, P., HERMENEGILDO, M. AND DEBRAY, S. 1996. A methodology for granularity-based control of parallelism in logic programs. *Journal of Symbolic Computation 22*(4), 715–734.

MARLOW, S., JONES, S. P. AND SINGH, S. 2009. Runtime support for multicore Haskell. *SIGPLAN Notices 44*(9), 65–78.

MUTHUKUMAR, K., BUENO, F., DE LA BANDA, M. J. G. AND HERMENEGILDO, M. V. 1999. Automatic compile-time parallelization of logic programs for restricted, goal level, independent AND-parallelism. *Journal of Logic Programming 38*(2), 165–218.

PONTELLI, E., GUPTA, G., PULVIRENTI, F. AND FERRO, A. 1997. Automatic compile-time parallelization of prolog programs for dependent and-parallelism. In *Proceedings of the 14th International Conference on Logic Programming*. Leuven, Belgium, 108–122.

SHEN, K., COSTA, V. S. AND KING, A. 1999. Distance: A new metric for controlling granularity for parallel execution. *Journal of Functional and Logic Programming 1999*(1).

SOMOGYI, Z., HENDERSON, F. AND CONWAY, T. 1996. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming 26*(1–3), 17–64.

TANNIER, J. 2007. Parallel Mercury. M.S. thesis, Institut d'informatique, Facultés Universitaires Notre-Dame de la Paix, 21, rue Grandgagnage, B-5000 Namur, Belgium.

VON PRAUN, C., CEZE, L. AND CAŞCAVAL, C. 2007. Implicit parallelism with ordered transactions. In *Proceedings of the 12th Symposium on Principles and Practice of Parallel Programming*. San Jose, California, 79–89.

WANG, P. AND SOMOGYI, Z. 2011. Minimizing the overheads of dependent AND-parallelism. In *Proceedings of the 27th International Conference on Logic Programming*. Lexington, Kentucky.