Parallelising a Large Functional Program Or: Keeping LOLITA Busy

Hans-Wolfgang Loidl[1], Richard Morgan[2], Phil Trinder[3], Sanjay Poria[2], Chris Cooper[2], Simon Peyton Jones[1], Roberto Garigliano[2]

[1] Department of Computing Science, University of Glasgow, Glasgow, Scotland, U.K.
[2] Department of Computer Science, University of Durham Durham, England, U.K.
[3] The Computing Department, The Open University, Milton Keynes, England, U.K.

Abstract. In this paper we report on the ongoing parallelisation of LOLITA, a natural language engineering system. Although LOLITA currently exhibits only modest parallelism, we believe that it is the largest parallel functional program ever, comprising more than 47,000 lines of Haskell. LOLITA has the following interesting features common to real world applications of lazy languages:

- the code was not specifically designed for parallelism;
- laziness is essential for efficiency in LOLITA;
- LOLITA interfaces to data structures outside the Haskell heap, using a foreign language interface;
- LOLITA was not written by those most closely involved in the parallelisation.

Our expectations in parallelising the program were to achieve moderate speedups with small changes in the code. To date speedups of up to 2.4 have been achieved for LOLITA running under a realistic simulation of our 4 processor shared-memory target machine. Most notably, the parallelism is achieved with a very small number of changes to, and without requiring an understanding of most of the application. On the Sun SPARCserver target machine wall-clock speedup is currently limited by physical memory availability.

1 Introduction

Despite the advantages of the functional programming model for parallel computation, there are few large non-strict parallel functional programs. We believe that this is due to the scarcity of robust parallel functional language implementations as well as the lacking support for high level languages describing some behavioural aspects of the program. We have recently constructed GUM, a parallel runtime system for Haskell, based on the Glasgow Haskell Compiler [10]. In order to facilitate parallel program design we have developed evaluation strategies, describing the dynamic behaviour of lazy parallel programs. GUM is portable, and available on both shared- and distributed-memory architectures, including the CM5 [1], Sun SPARCServer shared-memory multiprocessor and networks of Suns and Alphas. It is freely available and has users and developers worldwide. GUM is also integrated with the GranSim parameterisable simulator and its visualisation tools [3]. The superset of Haskell used to program parallel machines is called Glasgow Parallel Haskell, GpH.

The LOLITA natural language engineering system [8] has been developed at Durham University over several years. It has not originally been written with a parallel execution of the code in mind. The team's interest in parallelism is partly as a means of reducing runtime, and partly also as a means to increase functionality within an acceptable response-time. The groups at Glasgow and Durham have cooperated to parallelise LOLITA, and we believe that the result is the largest non-strict parallel functional program ever constructed. Such a large program tests the scalability both of our parallel software engineering techniques and of the underlying implementation technology, i.e. the runtime system, simulator and the visualisation tools. It also makes excellent use of our integrated engineering environment for parallel program development and performance evaluation. We focus on this particular aspect in the accompanying paper [5].

The LOLITA system has several features that make it an interesting system to parallelise. First of all its size alone tests both our engineering environment as well as our methodology for developing parallel programs. Secondly, it has not been specifically designed with parallelism in mind. The declarative nature of Haskell, which leaves the evaluation order undefined, allows us to refine this order and to specify parallelism. Finally, laziness is essential for the efficiency of LOLITA, and therefore we have to preserve the degree of laziness specified in the sequential program.

The structure of the remainder of the paper is as follows. Section 2 briefly describes LOLITA with a special focus on those issues that are of special interest for the parallelisation. Section 3 gives a reference to a detailed discussion of GpH, and our new mechanism for introducing and controlling parallelism, evaluation strategies. Section 4 describes the parallelisation to date, and ongoing work. Section 5 discusses the lessons learnt from parallelising LOLITA and addresses runtime-system issues for parallel functional languages.

2 Introducing LOLITA

The LOLITA (Large-scale Object-based Linguistic Interactor Translator and Analyser) system is a state of the art natural language processing system, able to grammatically parse, semantically and pragmatically analyse, reason about, and answer queries on complex texts, such as articles from the financial pages of quality newspapers. In contrast to many existing systems LOLITA was designed as a general, domain-independent knowledge representation and reasoning system. Written substantially in Haskell, with some small components in C, it consists of over 47,000 lines of Haskell source code [2] (excluding comments).

The system has been under development at the Laboratory of Natural Language Engineering, University of Durham since 1986. The project currently involves a team of approximately 20 developers working simultaneously on different components of the system. In June 1993 the LOLITA system was demonstrated to the Royal Society in London. Research in the group follows a pragmatic approach to natural language processing; it is the production of a robust and useful working system that is of primary interest.

The team's interest in parallelism is partly as a means of reducing runtime, and partly also as a means to increase functionality within an acceptable response-time. The overall structure of the program bears some resemblance to that of a compiler, being formed from the following large stages:

- Morphology (combining symbols into tokens; similar to lexical analysis);
- Syntactic Parsing (similar to parsing in a compiler);
- Normalisation (to bring sentences into some kind of normal form);
- Semantic Analysis (compositional analysis of meaning);
- Pragmatic Analysis (using contextual information from previous sentences).

These stages form the core of LOLITA. Depending on how LOLITA is to be used, a final additional stage may perform a discourse analysis, the generation of templates or text (e.g. in a translation system), or it may perform inference on the text to answer queries. This modular structure of LOLITA is shown in Figure 1.

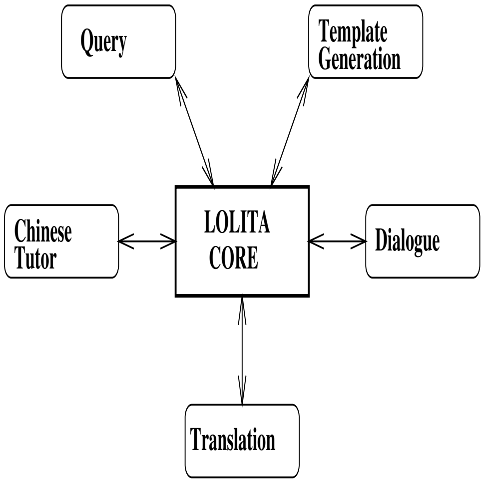

Fig. 1. Applications built around the LOLITA core

Central to LOLITA's flexibility is the semantic network, called SemNet, a graph based knowledge representation used in the core of LOLITA. In SemNet concepts and relationships are represented by nodes and arcs respectively, with knowledge being elicited by graph traversal. SemNet has been designed for an efficient form of inference, namely inheritance [6], and additionally supports other forms of reasoning such as analogy [7]. The task of the analysis stages is to transform the possibly ambiguous input into a piece of SemNet. Application-dependent backend stages can then extract pieces of the SemNet and present it in the required form. Currently, SemNet comprises approximately 100,000 nodes or 12Mb.

LOLITA is a large system, and a detailed description of all its features is beyond the scope of this paper. The interested reader is referred to [8]. However, the following features are important because they have a direct impact on our introduction of parallelism.

- LOLITA has already been carefully tuned, using time and heap profiling, for efficient sequential execution. We are parallelising an efficient program.
- Preserving laziness is crucial at several points in LOLITA. For example, in both parsing and semantic analysis the program selects the "best" parse- and semantic-trees from a forest. The cost of evaluating a single additional tree is prohibitively expensive. It is important that the parallel program is no more strict than the sequential program at these crucial points.
- The parser is substantially written in a "foreign language", namely C. As a result we must control parallelism in both languages, and the consequences of this are discussed in Section 5.
- LOLITA represents the knowledge used in the semantic analysis, SemNet, as a large (12Mb) C data structure. SemNet is stored outside the Haskell heap, and read using C function calls. The issues raised by making this structure available on each node of a multiprocessor are discussed in Section 4.6 and Section 5.

3 GPH and Evaluation Strategies

The GPH parallel language and the associated programming environment and methodology are described in Section 2 of a companion paper in these proceedings, namely *Engineering Large Parallel Functional Programs* [5]. The reader is referred to this section now as an understanding of these concepts is required to appreciate how parallelism is introduced in LOLITA. A complete description and discussion of evaluation strategies can be found in [9].

4 Parallelising LOLITA

4.1 Sequential Profiling

As a preparation for parallelising such a large program we performed sequential profiling of the code. This did not reveal a particular hotspot in the program although the syntactic parsing stage is the biggest component in the top level

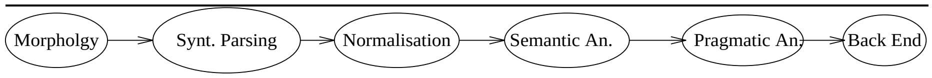

Fig. 2. Overall pipeline structure of LOLITA

structure with about 20% of the execution time. This stage makes heavy use of C-functions, called from within Haskell, to optimise the time consuming parsing process. These foreign language calls complicate the parallelisation of the parsing stage. The Haskell part of the parsing, however, can be parallelised without major recoding.

4.2 Top Level Pipeline

Without a dominating hotspot in the sequential execution of the program a pipeline approach is a promising way to achieve enough parallelism for a 4 processor shared-memory machine like the Sun SPARCServer we are aiming at. The structure of a pipeline parallel version is shown in Figure 2. Each stage discussed in Section 2 is executed by a separate thread, which are linked to form a pipeline. In contrast to classical pipelines, which require a large input set to achieve good parallelism, the non-strict semantics of Haskell makes it possible that several stages of the pipeline work on the same unit of input data in parallel. In particular it is possible for later stages to process parts of the generated data structure while earlier stages are still generating other parts of this data structure.

```
infixl 6 $|, $||       -- strategic function application
infixl 9 .|, .||       -- strategic function composition

($|), ($||) :: (a -> b) -> Strategy a -> a -> b
(.|), (.||) :: (b -> c) -> Strategy b -> (a -> b) -> (a -> c)

($|) f s x  = f x `demanding` s x
($||) f s x = f x `sparking`  s x

(.|) f s g = \ x -> let  gx = g x
                    in   f gx `demanding` s gx
(.||) f s g = \ x -> let  gx = g x
                     in   f gx `sparking` s gx
```

Fig. 3. Definition of strategic function application

The key step in constructing the pipeline is to define strategies on the complex data structures (e.g. parse trees) passed between stages. In order to avoid naming the data structures we use a *strategic function application* operator, which comes in a sequential `$|` and in a parallel `$||` variant. An expression of

```
wholeTextAnalysis opts inp global =
  result
  where
    -- (1) Morphology
    (g2, sgml) = prepareSGML inp global
    sentences  = selectEntitiesToAnalyse global sgml

    -- (2) Parsing
    rawParseForest = map (heuristic_parse global) sentences `using` parList rnf

    -- (3)-(5) Analysis
    anlys = stateMap_TimeOut (parse2prag opts) rawParseForest global2

    -- (6) Back End
    result = back_end anlys opts

-- Pick the parse tree with the best score from the results of
-- the semantic and pragmatic analysis.  This is done speculatively!

parse2prag opts parse_forest global =
 pickBestAnalysis global  $|| evalScores  $
 take (getParsesToAnalyse global)         $
 map analyse parse_forest
 where
   analyse pt =   mergePragSentences opts $ evalAnalysis
   evalAnalysis = stateMap_TimeOut analyseSemPrag pt global
   evalScores =   parList (parPair rwhnf (parTriple rnf rwhnf rwhnf))

-- Pipeline the semantic and pragmatic analyses
analyseSemPrag parse global =
 prag_transform            $|| rnf   $
 pragm                     $|| rnf   $
 sem_transform             $|| rnf   $
 sem (g,[])                $|| rnf   $
 addTextrefs global        $|  rwhnf $
 subtrTrace global parse

back_end inp opts =
 mkWholeTextAnalysis     $|  parTriple rwhnf (parList rwhnf) rwhnf $
 optQueryResponse opts   $|| rnf $
 traceSemWhole           $|| rnf $
 addTitleTextrefs        $|| rnf $
 unifyBySurfaceString    $|| rnf $
 storeCategoriseInf      $|| rnf $
 unifySameEvents opts    $|  parPair rwhnf (parList (parPair rwhnf rwhnf)) $
 unpackTrees             $|  parPair rwhnf (parList rwhnf)  $
 inp
```

Fig. 4. The top level function of LOLITA

the form `f $|| s $ x` applies the strategy `s` to the input value `x` and in parallel applies the function `f` to `x`. Thus, the strategy is mainly used to specify the evaluation degree whereas the `$||` operator specifies the pipeline parallelism. The `$` operator in Haskell denotes (explicit) function application, i.e. `f $ x = f x`, and the `.` operator denotes function composition, i.e. `(f . g) x = f (g x)`. The definitions of the new combinators are given in Figure 3.

These new combinators are particularly useful for defining data-oriented parallelism because they can be used to define parallelism on the intermediate data structures, via the strategy parameter. This approach simplifies the top-down parallelisation of very large systems, since it is possible to define the parts of the data structure that should be evaluated in parallel without considering the algorithms that produce the data structures. Note that the data-oriented parallelism

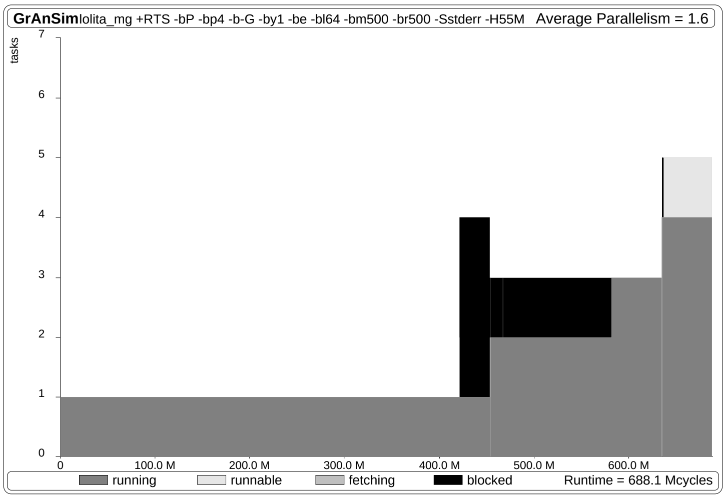

Fig. 5. Activity profile of LOLITA with pipeline parallelism

on the data structures can be defined independently from the control-oriented parallelism of the pipeline itself.

The code of the top level function `wholeTextAnalysis` in Figure 4 clearly shows how the algorithm is separated from the dynamic behaviour in each stage by using the `$||` operator. In a first parallel version we achieved the same separation with an explicit pipeline strategy. However, this required to name every intermediate value in the pipeline which did not reflect the structure of the code in quite the same way. This experience was our main motivation for developing the strategic function application operator.

Note that this code uses a `parList` strategy in the definition of the variable `rawParseForest` in the parsing stage to describe data parallelism over the whole input by processing sentences in the input text in parallel. At the moment we cannot use this source of parallelism because the C code in this stage is not re-entrant. Changing the C code to exploit this form of parallelism is ongoing work.

The semantic and pragmatic analysis stages are wrapped into a timeout function in order to guarantee a worst case response time of the system. This indicates that these stages can be very computationally intensive. Therefore, both analyses are kept rather simple in the sequential system. By providing the strategy `evalScores`, in `parse2prag`, speculative parallelism is defined, which allows the system to perform a more sophisticated analysis by examining several possible parse trees. The goal of this strategy is therefore to improve the quality of the result. Section 4.4 discusses this issue in more detail. In general, it would be very desirable to improve the quality of semantic and pragmatic analysis

```
mergeStrategy :: (NFData a, NFData b) =>
                 (ParseForest,FeatureForests) -> Span -> MergeStrategy a b

mergeStrategy (pf,ff) span
  | totalSpan == 0             = MStrat serialMerge
  | percentSpanned >= minSpan  = MStrat parallelMerge
  | otherwise                  = MStrat serialMerge
   where
     percentSpanned = (span * 100) `div` totalSpan
     totalSpan = forestSpan pf
     minSpan = getParsingParPercent (forestGlobal pf)

     parallelMerge :: (NFData a, NFData b) =>
                      [(a,b)]  -> [(a,b)] -> Strategy [(a,b)]
     parallelMerge as bs _
      = fstPairFstList bs `par`
        fstPairFstList as `seq`
        ()

     fstPairFstList :: (NFData a, NFData b) => Strategy [(a,b)]
     fstPairFstList = seqListN 1 (seqPair rwhnf r0)

     serialMerge :: (NFData a, NFData b) =>
                    [(a,b)]  -> [(a,b)] -> Strategy [(a,b)]
     serialMerge as bs
      = r0
```

Fig. 6. A granularity control strategy used in the parsing stage

in the system. Parallelism inside these stages could be used to maintain good performance despite the increased complexity of the system.

On first glance the activity profile in Figure 5 seems to exhibit very poor parallelism. However, part of the sequential front end of this algorithm is the C part of the parsing, which cannot be parallelised in this fashion because of its strictness. Furthermore, the consistent parallelism towards the end of the compilation yields a fairly good utilisation for a four processor machine. Considering the minimal changes in the code to achieve parallelism this is a reasonable first step in the parallelisation process.

4.3 Parallel Parsing

One major source of parallelism in the time consuming syntactic parsing phase is the merging of possible parse trees in order to build a parse tree for a whole sentence. One complication in the parsing of natural languages is their ambiguity. Because of this ambiguity the parsing stage produces not just one but a list of possible parse trees. Internally, however, the result is represented as a single tree, which at some points contains alternatives ("or-nodes") representing different possible parses of the subtrees. A lazy function is used to convert this single tree into a list of possible parse trees. In each or-node the parser, which returns a list of parse trees, must merge the lists of parse trees produced by the recursive calls. In merging these lists the possible parse trees have to be sorted based on some simple syntactic criteria representing the likelihood of a parse, and the laziness

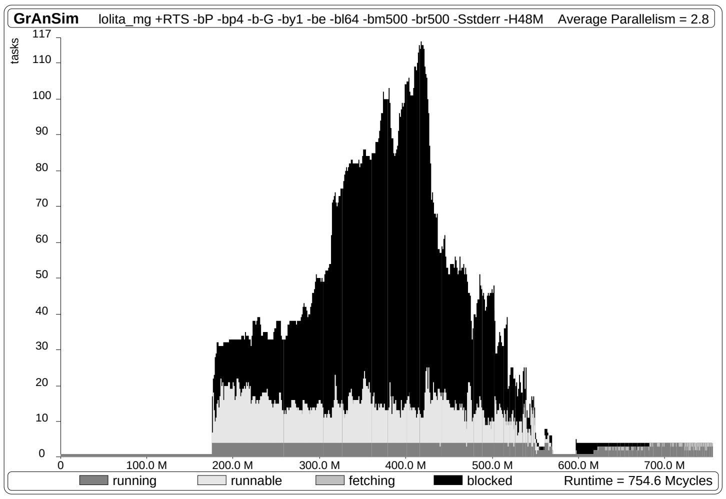

Fig. 7. Activity profile of LOLITA with a span threshold of 50%

of Haskell is crucial. In order to produce one parse tree in an or-node it is only necessary to evaluate the first element in the lists produced by all alternatives.

From a parallelism point of view this behaviour explains why we cannot force the evaluation of parts of the parse forest without risking to introduce a high degree of redundant work. Furthermore, within the parsing process the merging of lists triggers the evaluation of sublists, in particular the evaluation of the quality of possible parses. Although the merging itself is very cheap it triggers work that can be usefully done in parallel.

In order to improve the granularity of the threads produced by the parallel tree traversal in the parsing stage, we apply a thresholding strategy shown in Figure 6 to the "span" in the tree. The span value, which is attached to each node in the tree, specifies the number of leaves in the current subtree. The threshold for generating a parallel process in order to merge all possible subtrees is specified as a percentage of leaves that can be reached from the current node, and this percentage is part of the global system environment. Checking the threshold is very cheap because it only involves the comparison of the `span` argument, as a percentage, with a system parameter assigned to `minSpan`.

The two parallel calls to `fstPairFstList` in `parallelMerge` define parallelism in this stage. Only the first element of the pair is evaluated because it contains the value determining the quality of the resulting parse tree. Thus, the `fstPairFstList` strategy specifies an evaluation degree that is sufficient to se-

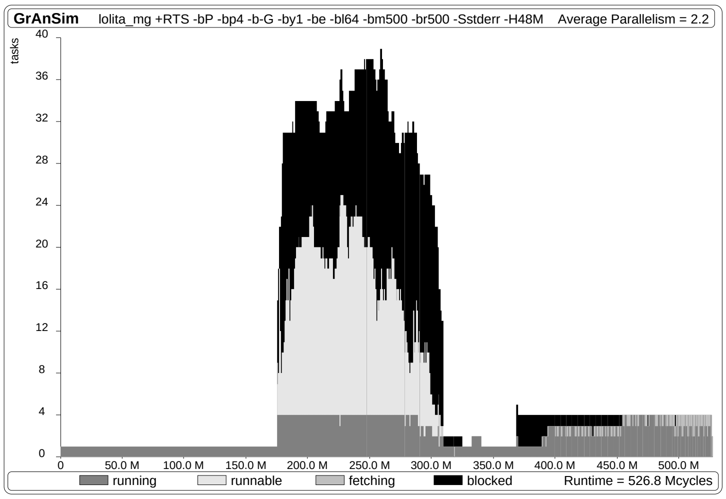

Fig. 8. Activity profile of LOLITA with a span threshold of 90%

lect the tree to return as the result of the syntactic parsing stage but without evaluating the tree itself more than necessary.

One strength of strategies is their reusability for different algorithmic code that has the same dynamic behaviour. We were able to exploit this feature with `mergeStrategy` in Figure 6 by applying the same polymorphic thresholding strategy to two lists of different types within the syntactic parsing stage. This reuse is highlighted by the parameterisation of the `MergeStrategy` datatype over the two possible types in the list.

We have performed a series of measurements under GranSim in order to determine the best value for the span in this strategy. We have used a setup that models the 4 processor shared-memory Sun SPARCServer available at Durham. Time is always measured in machine cycles.

Figure 7 shows the activity profile for a span threshold of 50%. During the syntactic parsing stage we achieve a quite good utilisation of the system. However, almost 100 blocked threads and a high number of runnable threads are generated, too. These impose significant runtime overhead in the system. The histogram of threads sorted by thread size in the granularity profile at the left hand side of Figure 9 reveals that most of the threads are very fine-grained: 3,422 of the 5,122 threads (67%) are shorter than 2,000 cycles. This leads to a bad ratio of computation versus parallelism overhead.

In comparison, when increasing the span threshold to 90%, in Figure 8, the number of blocked and runnable threads is reduced significantly (at most 36), and the number of small threads drops drastically, as shown at the right hand side of Figure 9. Now, only 67 of the 165 threads are shorter than 2,000 cycles (40%). Corresponding to this drop in the total number of threads, especially fine-grained threads, the runtime drops from 754.6 Mcycles in the previous version to 526.8 Mcycles in this version.

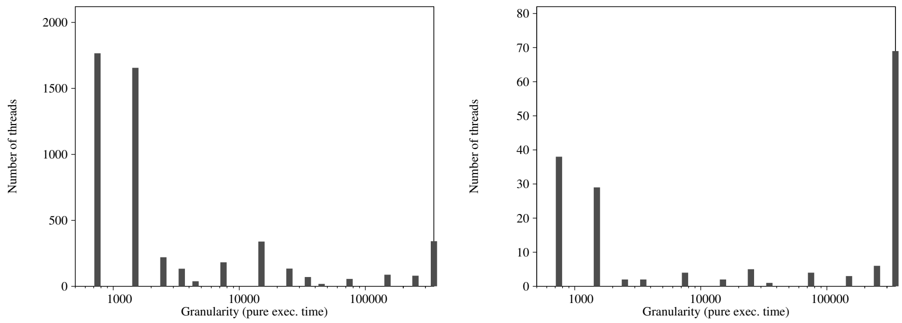

Fig. 9. Granularity profiles of LOLITA with a span thresholds of 50% and 90%

As a result of these measurements and considering the low amount of parallelism that is required to fully utilise the 4 processor shared-memory machine, we use span thresholds around 90% for GUM executions of LOLITA.

4.4 Parallel Semantic Analysis

Another source of parallelism can be used to improve the quality of the analysis by applying the semantic and pragmatic analyses in a data-parallel fashion on different possible parse trees for the same sentence. Because of the complexity of these analyses the sequential system always picks the first parse tree, which may cause the analysis to fail, although it would succeed for a different parse tree. In this case the system cannot produce a result for the current sentence in a sequential setup. Therefore, parallelism in this stage would not reduce runtime of the system, but might improve the quality of the result.

This additional data parallelism is defined by the strategy `evalScores` in the function `parse2prag` (see Figure 4). The parse forest `rawParseForest` contains all possible parses of a sentence. The semantic and pragmatic analyses are then applied to a predefined number (specified in the global system environment `global`) of these parses. The data parallel strategy `evalScores` is applied to the list of these results and demands only the score of each analysis (the first element in the triple) in order to avoid unnecessary computation at this stage.

This score is used in `pickBestAnalysis` to decide which of the parses to choose as the result of the whole text analysis.

So far we have not systematically measured the improvements in the quality of the result by analysing several possible parse trees. However, considering that about 70% of all sentences that are analysed have several possible parse trees, the possibility to analyse several of them without large additional costs is very attractive from a natural language engineering point of view.

4.5 Overall Parallel Structure

Figure 10 summarises the overall parallel structure arising when all of the sources of parallelism described above are used. The syntactic parsing stage is internally parallelised using the granularity control strategy shown in Figure 6. Note that the analyses may add nodes to the semantic net. This creates an additional dependence between different instances of the analysis (indicated as vertical arcs). Lazy evaluation ensures that this does not completely sequentialise the analyses, and it could be turned off for parallel queries to the system.

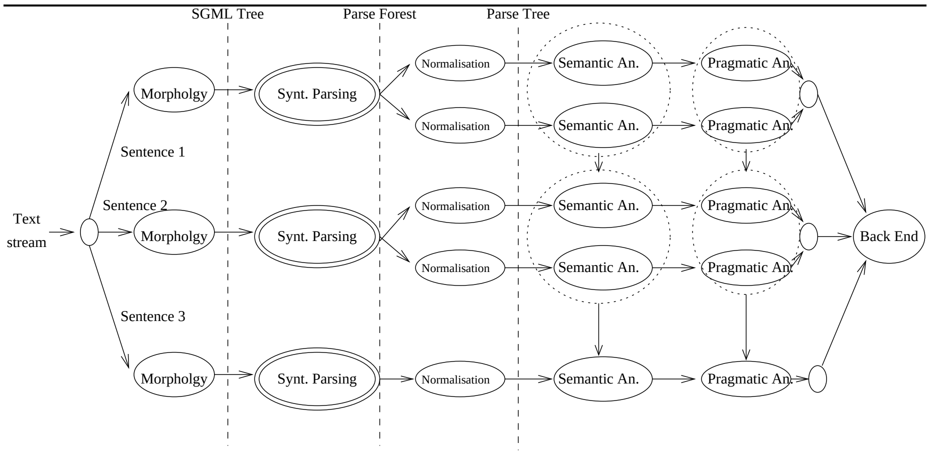

Fig. 10. Detailed structure of LOLITA

It should be emphasised that specifying the strategies that describe this parallel behaviour entailed understanding and modifying only two of about three hundred modules in LOLITA and three of the thirty six functions in that module. So far, the only module we have parallelised is the syntactic parsing stage. In fact, the most tedious part of the code changes was adding instances of `NFData` for intermediate data structures, which are spread over several dozen modules. However, in the meantime this process has been partially automated [11].

If it proves necessary to expose more parallelism we could parallelise other sub-algorithms, which also contain significant sources of parallelism. For example in the inference process, which tries to extract information from the semantic net, standard graph algorithms such as computing the shortest distance between nodes are used. Parallel versions of these graph algorithms could increase the performance further.

4.6 Sun SPARCserver Implementation

We are currently tuning the performance of LOLITA on the Sun SPARCServer. A realistic simulation showed an average parallelism between 2.5 and 3.1, using just the pipeline parallelism and parallel parsing. The actual speedup, however, does not exceed 2.4. Our measurements with varying span values indicate that this is partly caused by fine-grained parallelism in the parsing stage.

Since LOLITA was originally written without any consideration for parallel execution and contains a sequential front end of about 10–15% (the C part of the syntactic parsing stage), we are pleased with this amount of parallelism. In particular the gain for a set of rather small changes is quite remarkable.

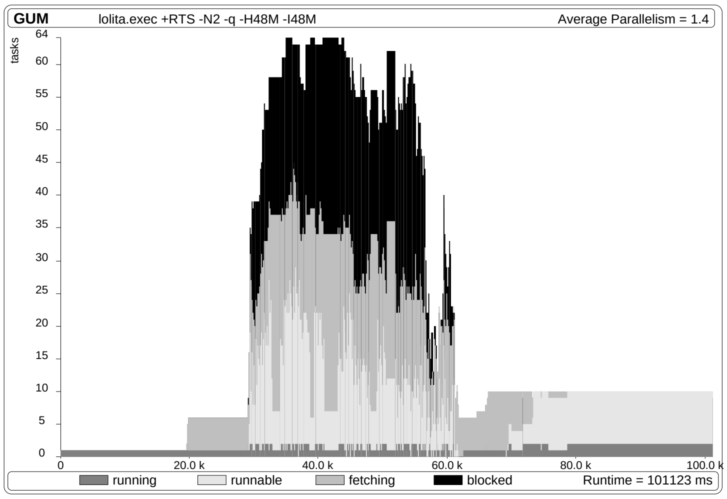

Fig. 11. Activity profile of LOLITA run under GUM with 2 processors

However, the wall-clock speedups obtained to date do not quite match the simulation results. As shown in Figure 11 a two processors execution on small inputs achieves an average parallelism of 1.4. We use a high span value to bound the amount of parallelism in the parsing phase. This also bounds the total heap residency in the system, which proves to be very important. With more processors the available physical memory is insufficient and heavy swapping causes a drastic degradation in performance. The reason for this is that GUM, which is designed to support distributed-memory architectures uniformly, loads a copy of the entire code, and a separate local heap, onto each processor. LOLITA is a very large program, incorporating large static data segments (totalling 16Mb), and requires 100Mb of virtual memory in total in its sequential incarnation. Clearly a great deal of this data could be shared, an opportunity we are exploring. We are also making the C parsing functions re-entrant which will allow the analysis to be performed in a data-parallel fashion over a set of input sentences.

The multiprocessor GUM activity profiles can only be loosely compared with the GranSim profiles, e.g. Figure 8. However the shapes of the curves are similar, except that the GUM activity profile shows a larger degree of fetching. This is probably caused by the rather expensive but generic PVM communication routines used by GUM. In contrast, GranSim measures mainly the hardware costs for performing communication. Together with the fine granularity of the generated threads this increased overhead leads to a significantly smaller utilisation in the parsing stage. However, the later pipeline stages in the computation are still an effective source of parallelism.

5 Discussion

At the current stage of parallelising LOLITA we use top level pipeline parallelism as well as divide-and-conquer parallelism in the most expensive stage of the pipeline. Additional speculative parallelism can be used in the analysis stage in order to improve the quality of the result. We are currently working on changes in the C code to provide global data parallelism over the whole program (the changes in the Haskell code are in place already).

The current parallel version of LOLITA achieves a parallelism between 2.5 and 3.1 under GranSim emulating a 4-processor shared-memory machine. The corresponding speedup, however, does not exceed 2.4. This is partly due to overhead caused by very fine-grained parallelism and partly due to strategies that perform speculative computations (although we avoided speculation on potentially expensive components). The thresholding strategy in the parsing stage proved to be very effective in increasing the average granularity of the generated threads.

We have yet to obtain significant wall-clock speedups on our target machine. This, however, is not due to a lack of parallelism but due to the very high memory consumption of the application, which exceeds the available main memory in the current setting. We also hope to obtain further parallelism by making the C parsing code re-entrant, allowing us to exploit data parallelism by analysing sentences in parallel.

Parallelising LOLITA has taught us a great deal about large-scale parallel functional programming. The most intriguing aspect is that the parallelism is achieved using a very small number of changes to the Haskell parts of the application. We have been able to use a top-down approach of the parallelisation to an extent, which would be very difficult in a strict language. All of the parallelism has been specified by evaluation strategies acting on the data structures passed between modules. As a result, the parallelism has been introduced without changing, and indeed without understanding most of the program. This abstraction is crucial when working on an application of this size. For example, introducing top-level parallelism entailed changing just one out of around three hundred modules. Considering this small amount of code changes in a large application, the fact that it was not designed to be executed in parallel, and the presence of inherently sequential foreign language calls, we are pleased with the achieved degree of parallelism.

Our experiences with LOLITA and several other medium-scale programs are being distilled into some guidelines for engineering large parallel functional programs [5]. Another concrete outcome of parallelising LOLITA has been the introduction of strategic function application to support pipeline parallelism. This extension to our strategy module has in the meantime proven useful for several other parallel programs like the parallelising compiler Naira [4].

The parallelisation of LOLITA pointed out the importance of several runtime-system issues. The choice of one fixed heap size for each processor proved to be too inflexible and we had to slightly change the runtime-system. In a shared-memory setting we would like to have heaps that can grow dynamically if necessary. In general either program annotations or runtime-system options for tuning the data-locality of the program would be useful. Uniform access to an external data-structure from every processor is important because the semantic net is constructed outside the Haskell heap. For a distributed memory usage of such data-structures the communication mechanism in GUM would have to treat them specially. In the current version, the presence of such a conceptually persistent data structure required recoding of the C routines for reading and sharing this data structure. In fact, recoding just this part of the C code required more changes than the actual parallelisation of the Haskell program itself. General support of persistent data in a distributed setting would simplify the program and its parallelisation considerably.

References

1. K. Davis. MPP Parallel Haskell. In *IFL'96 — Intl. Workshop on the Implementation of Functional Languages*, pp. 49–54, Bad Godesberg, Germany, Sep. 1996. Draft Proceedings.
2. A. Gill and P. Wadler. Real World Applications of Functional Programs. <URL:`http://www.dcs.gla.ac.uk/fp/realworld.html`>
3. K. Hammond, H-W. Loidl, and A. Partridge. Visualising Granularity in Parallel Programs: A Graphical Winnowing System for Haskell. In *HPFC'95 — High Performance Functional Computing*, pp. 208–221, Denver, CO, Apr. 10–12, 1995. <URL:`http://www.dcs.st-and.ac.uk/~kh/papers/hpfc95/hpfc95.html`>
4. S. Junaidu. NAIRA: A Parallel2 Haskell Compiler. In *IFL'97 — Intl. Workshop on the Implementation of Functional Languages*, Univ. of St. Andrews, Scotland, Sep. 10–12, 1997.

5. H-W. Loidl and P.W. Trinder. Engineering Large Parallel Functional Programs. In *IFL'97 — Intl. Workshop on the Implementation of Functional Languages*, Univ. of St. Andrews, Scotland, Sep. 10–12, 1997.
6. D. Long and R. Garigliano. Inheritance hierarchies. Technical Report 4/88, Dept. of Computer Science, Univ. of Durham, 1988.
7. D. Long and R. Garigliano. *Reasoning by Analogy and Causality: A model and application.* Artificial Intelligence. Ellis Horwood, 1994.
8. R.G. Morgan, M.H. Smith, and S. Short. Translation by Meaning and Style in Lolita. In *Intl. BCS Conf. — Machine Translation Ten Years On*, Cranfield University, Nov. 1994.
9. P.W. Trinder, K. Hammond, H-W. Loidl, and S.L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1), Jan. 1998.
10. P.W. Trinder, K. Hammond, J.S. Mattson Jr., A.S Partridge, and S.L. Peyton Jones. GUM: a Portable Parallel implementation of Haskell. In *PLDI'96 — Programming Languages Design and Implementation*, pp. 79–88, Philadelphia, PA, May 1996.
11. N. Winstanley. A Type-Sensitive Preprocessor for Haskell. In *Glasgow Workshop on Functional Programming*, Ullapool, Scotland, Sep. 15–17, 1997.