Serialisation Analysis of Concurrent Logic Programs

Andy King and Paul Soper

Department of Electronics and Computer Science,
University of Southampton, Southampton, S09 5NH, UK.

Abstract

Serialisation analysis reduces the time a concurrent logic program spends communicating and scheduling. It fits granularity analysis and schedule analysis into a single unified framework for partitioning a program into grains for parallel evaluation and ordering grains for sequential evaluation. Serialisation analysis is simple, avoids the complexity of thresholding, and unlike previous proposals for controlling granularity, is based on threads. The threads avoid the creation of suspensions and therefore reduce scheduler activity. Threads also act as indivisible units of work, and therefore inhibit the parallel evaluation of fine-grained processes. Furthermore, serialisation analysis incurs no extra run-time overhead.

1 Introduction

The concurrent logic languages apply a process model of Horn clauses which is, in principle, well suited to multi-processor implementation. The execution of a concurrent logic program and query divide naturally into abstract processes for parallel execution. If this division is carried too far, however, the gains due to parallel execution can be outweighed by the overheads associated with process creation and communication. Serialisation analysis combines into a unified framework the two main techniques for reducing these overheads.

Partitioning and scheduling are two of the main obstacles to achieving an efficient mapping of a program to a multi-processor. Granularity analysis is a way of partitioning a program into grains (King and Soper, 1990; Tick, 1990; Debray *et al.*, 1990) and schedule analysis is a way of dividing a program into threads of totally ordered atoms (King and Soper, 1991a). Granularity analysis decreases the time spent communicating and schedule analysis decreases the time spent scheduling. Each analysis breaks a program into fragments satisfying distinct constraints. For granularity analysis, the set of constraints depend on the time complexities of processes, whereas for schedule analysis, the set of constraints relate to the data-dependencies between processes.

Granularity analysis aims to coalesce processes (or goals in the case of logic programs) into larger grains, to be executed on a single processor, when it would be less efficient to evaluate them in parallel. Tick (1990) describes a simple heuristic algorithm for estimating granularity. This analysis is crude and does not satisfactorily model recursive predicates. Recursive predicates present difficulties because the quantity of computation is data dependent and therefore is difficult to determine at compile-time. Debray *et al.* (1990) explain how to deal with recursive predicates by deriving complexity functions for predicates at compile-time. The complexity functions are the solutions of difference equations on argument sizes, formulated as functions of the size of the data. Once the size of the data is known at run-time, the complexities can be simply calculated. Specifically, size of the data is checked against a threshold to determine whether or not the goal should be evaluated in parallel. King and Soper (1990) independently proposed the use of complexity functions to control granularity, the complexity functions again derived as the solutions of difference equations on argument sizes. Instead of checking a complexity function to a threshold at run-time, King and Soper argued that coalescing decisions should be made at compile-time, pointing out that in some circumstances the overhead of thresholding can actually give a net slow-down. The idea was to coalesce processes together if the complexity of their communication dominated the complexity of the computation on all sizes of the possible data.

Schedule analysis is concerned with deducing at compile-time a partial schedule of processes, or equivalently the body atoms of a clause, which is consistent with the program behaviour. Program termination characteristics are affected if an atom which binds a shared variable is ordered after an atom that matches on that variable. In order to avoid this, an ordering of the atoms is determined which does not contradict any data-dependence of the program. The data-dependencies between processes can be inferred by producer and consumer analysis (King and Soper, 1991; Foster and Winsborough, 1991) or alternatively by the mode algorithm proposed by Ueda and Morita (1990). Producer and consumer analysis does not infer the instantiation states of an atom, as described by Debray (1989), but rather deduces which processes are responsible for the construction of which parts of a term (produce) and which processes match on which parts of a term (consume). In general processes cannot be totally ordered and thus the analysis leads to a division into threads of totally ordered processes. In this way the work required of the run-time scheduler is reduced to ordering threads.

Serialisation analysis makes explicit the relationship between the granularities and the data-dependencies among a set of processes. Originally King and Soper (1991a) suggested a two step serialisation: first granularity analysis is used to partition a program into grains; second schedule analysis is used to generate threads for each grain. This procedure has the right basic ingredients, but applies them in the wrong order. Serialisation analysis, as presented in this paper, is a one step procedure: schedule analysis is used to generate threads which simultaneously satisfy the granularity constraints. The basic idea is to divide a program into threads which are allocated to queues for scheduling. Each queue is considered to be accessible to all processors of a multi-processor. Two sets of constraints are used to generate threads. First, a thread must not contradict any data-dependence in a program. Second, since a thread may be evaluated in parallel with other threads, atoms of a program which are sufficiently coarse-grained processes must be allocated to different threads. A further important aspect of serialisation analysis is that it avoids the introduction of spurious coroutining activity between threads. The threads generated by serialisation analysis satisfy all these sets of constraints and are therefore taken to be the basic grains of parallelism.

Serialisation analysis is superior to the previous proposals for controlling granularity in four important respects. First, it avoids the complexity of thresholding (Debray *et al.*, 1990) and possesses much of the simplicity of the granularity analysis suggested by Tick (1990). Second, unlike the granularity analysis of Tick, serialisation analysis produces threads. The threads avoid extra suspensions and therefore scheduler activity is reduced. Third, the threads act as indivisible units of work, and therefore inhibit the parallel evaluation of fine-grained processes. In contrast, the analysis of Tick permits fine-grained processes to be evaluated on remote processors. Fourth, serialisation analysis incurs no extra run-time overhead.

The paper is organised as follows. Section 2 illustrates serialisation analysis with a simple example. The formal description of serialisation analysis is presented in two parts. Section 3 defines a classification scheme from which granularity constraints are derived. Section 4 reviews schedule analysis and defines a procedure for generating threads which satisfy schedule and granularity constraints. It includes a proof that the procedure cannot affect termination properties of the program. Section 5 outlines implementation details. Sections 6 and 7 present future work and the concluding discussion.

2 Serialisation by example

Example 1 *Consider the recursive clause of the `fib/2` predicate listed in figure 1. Granularity analysis will classify the comparison and arithmetic atoms as too fine-grained to warrant parallel evaluation, so the only remaining atoms which might possibly be worth evaluating in parallel are the two recursive `fib/2` atoms. Since only one of the `fib/2` atoms should be evaluated on a remote processor, the atoms of the clause might be divided into two threads, each thread containing one of the `fib/2` atoms.*

Serialisation analysis also endeavours to construct threads which reduce the number of cyclic data-dependencies between threads. Such data-dependencies engender coroutining. By avoiding

```
fib(N, F) <-
   N =< 1 : F = 1.
fib(N, F) <-
   N > 1 : N1 is N - 1, N2 is N - 2,
   fib(N1, F1), fib(N2, F2), F is F1 + F2.
```

Figure 1: The `fib/2` predicate.

cyclic data-dependencies, fewer suspensions are likely to be created, and thus scheduling is further improved.

Example 2 *Returning to the recursive clause of the* `fib/2` *predicate listed in figure 1, two threads might be generated. One thread might include the atoms* `N1 is N - 1`, `N2 is N - 2`, `fib(N1, F1)` *and* `F is F1 + F2`*, placed in that order, leaving the* `fib(N2, F2)` *atom for the other thread. Although the partition is valid, a cyclic data-dependency is introduced between the threads, since the evaluation of* `fib(N2, F2)` *has to suspend until* `N2` *is calculated and the evaluation of* `F is F1 + F2` *has to suspend until* `F2` *is computed. The cyclic data-dependency does not originate in the clause but is induced by the partition.*

In the frequently occurring case of a clause which does not possess any cyclic data-dependencies, threads are formed and are themselves ordered so that the data-dependencies between the threads flow left-to-right. Threads are added to the end of the queue in reverse order. By interpreting the queue as a stack, depth-first scheduling can be performed on the local processor. Since the threads are allocated to the queue in reverse order, threads are deallocated from the stack in the right order, and thus when evaluated sequentially, the threads are ordered and atoms ordered within each thread, so as avoid the creation of suspensions. Depth-first scheduling is also likely to improve cache locality and reduce the need for garbage collection. Furthermore, since threads are composed of fine-grained processes, and often a single coarse-grained process, each thread is also likely to warrant parallel evaluation. Thus a remote processor can steal a thread for parallel evaluation, with the thread itself acting as the granularity control, without incurring any extra scheduling overhead. (In practice, however, a variant of the tail recursive optimisation can be employed to hold the first thread ready for evaluation and only place the following threads, in the case of the recursive `fib/2` clause just the second, on the stack. This further reduces references to the queue and the scheduler.)

Example 3 *Continuing with the* `fib/2` *predicate listed in figure 1, to alleviate cyclic data-dependencies between the threads, allocate the* `N1 is N - 1`, `N2 is N - 2` *and* `fib(N1, F1)` *atoms, in that order, to the first thread and the* `fib(N2, F2)` *and* `F is F1 + F2` *atoms, in that order, to the second thread. The data-dependencies flow left-to-right between threads and are not contradicted within each thread. Thus by placing the threads in the queue in reverse order, the second thread first and the first thread second, when the queue is interpreted as a stack for depth-first scheduling, the first thread is selected first for evaluation and the second thread is selected second. Hence scheduler activity is reduced. Note too that both threads are suitable for parallel evaluation.*

3 Classifying atoms for threads

Processes are classified as constant, linear or non-linear according to how the cumulative difference between the computation and communication grows over the lifespan of the process. The following treatment is based on the formulation of granularity analysis set out by King and Soper (1990). Processes in the constant class correspond to predicates which are defined as builtins, or are defined in terms of constant atoms. Processes which are linear correspond to a sub-class of the linearly recursive predicates. A linearly recursive predicate is a recursive predicate such that each of its clauses is either non-recursive or linearly recursive. A linearly recursive clause is one composed of atoms which correspond to non-recursive predicates and, in addition, includes a single atom which coincides with the predicate itself. The linear sub-class corresponds to those linearly recursive predicates for which it is known that communication dominates computation. A non-linear process is defined as being a process which is neither constant nor linear. This classification gives a simple prescription for controlling granularity: two atoms which are non-linear should be allocated to different threads whereas any other combination of two atoms, for instance a linear atom and a non-linear atom, can potentially be allocated to the same thread.

The justification for the classification is that the constant and linear classes tend to represent those frequently occurring fine-grained processes which can be coalesced without employing thresholding. Predicates which correspond to the constant class have a fixed time complexity and are therefore susceptible to simple granularity analysis. The constant class is important because, if the statistics presented for Prolog by Touati and Despain (1987) are also applicable to concurrent logic programs, half the atoms of a clause, on average, correspond to builtins, and therefore fall into the constant class. In addition, linearly recursive predicates represent the simplest and perhaps the most prevalent form of recursion. The linear sub-class thus corresponds to another important and commonly occurring type of predicate which gives rise to fine-grained processes.

The linear sub-class coincides with processes (for those linearly recursive predicates) which can be shown to be fine-grained by comparing the complexity of the computation to the complexity of the communication. If the predicate traverses part of a data-structure at each level of recursion, the complexity of the communication will grow in proportion to the complexity of the computation and thus only prefactors need be used to decide whether or not the process is linear. Specifically, to keep with a conservative form of granularity analysis, communication is under-estimated by calculating the minimum rate of growth of communication for the recursive clauses of a predicate. The rate of growth is computed by comparing the size of the head arguments to the size of the arguments of the recursive atom. The size of the arguments is, in turn, found by inferring types. Types are convenient way of characterising the arguments of the head atom and the arguments of the recursive atom but, however, associate an argument with a set of terms and thus only give a range of possible sizes. Nevertheless, the size of a term can be inferred from its type, measured in a way which reflects the size of the machine representation and therefore gauges the communication overhead.

A norm $||.||_{\mathrm{space}}$ is introduced to count the number of machine words that are typically used in the structure-sharing representation of a term. If a n-ary function occupies $n+1$ machine words, excluding the space required to store its arguments, and each occurrence of a constant or variable requires one word, then the $||t||_{\mathrm{space}}$ norm can be defined to be the sum of the arities of the functions in t totalled with the number of sub-terms in t.

Example 4 *$||\texttt{V}||_{space} = 0 + 1 = 1$, $||\texttt{f(a, V)}||_{space} = (1 + 2) + (0 + 1) + (0 + 1) = 5$ and $||\texttt{[a, b]}||_{space} = (1 + 2) + (0 + 1) + \{(1 + 2) + (0 + 1) + (0 + 1)\} = 9$.*

To make coalescing decisions, the communication and the computation have to be compared, and therefore put into a common unit. Thus a (latency) norm $||.||_{\mathrm{lat}}$ is introduced and defined to be a constant multiple of $||.||_{\mathrm{space}}$, the constant determined by the implementation, and chosen to quantify the communication overhead in terms of the units of computation.

To classify atoms, clauses are catogorised and then predicates classified according to their component clauses. An atom adopts the classification of its predicate. Clauses are classified in definition 1 and predicates, and thus atoms, are classified in definition 2.

Definition 1 *Let c denote a clause with head atom h, guard atoms $g_1, \ldots, g_i$ and body atoms $b_1, \ldots, b_j$ and suppose that $t_1^{\#}, \ldots, t_k^{\#}$ denote the types of the head arguments of c. In addition, let T_{loc} denote the time required to create a process and initiate communication, T_{com} denote an over-estimate for the amount of computation, and T_{lat} denote an under-estimate for the amount of communication.*

- *If c is non-recursive, c is constant if $T_{com}(c) - T_{lat}(c) \leq T_{loc}$ and c is non-linear otherwise where $T_{com}(c) = T_{com}(h) + T_{com}(g_1) + \ldots + T_{com}(g_i) + T_{com}(b_1) + \ldots + T_{com}(b_j)$ and $T_{lat}(c) = \min(\{||t||_{lat} \mid t \in t_1^{\#}\}) + \ldots + \min(\{||t||_{lat} \mid t \in t_k^{\#}\})$.*

```
sum(1, Sum) <- Sum = 1.
sum(N, Sum) <- N > 1 : N1 is N - 1, sum(N1, Sum1), Sum is Sum1 + N.
```

Figure 2: The `sum/2` predicate.

- *If c is linearly recursive, b_l is the recursive atom, and $t_1^{\#'}, \ldots, t_k^{\#'}$ the types of the arguments of b_l, c is linear if $T_{com}(c) \leq T_{lat}(c)$ and c is non-linear otherwise where $T_{com}(c) = T_{com}(h) + T_{com}(g_1) + \ldots + T_{com}(g_i) + T_{com}(b_1) + \ldots + \ldots + T_{com}(b_{l-1}) + T_{com}(b_{l+1}) + T_{com}(b_j)$ and $T_{lat}(c) = \min(\{\|t\|_{lat} - \|t'\|_{lat} \mid t \in t_1^{\#}, t' \in t_1^{\#'}\} \cap \mathbb{N}_0) + \ldots + \min(\{\|t\|_{lat} - \|t'\|_{lat} \mid t \in t_k^{\#}, t' \in t_k^{\#'}\} \cap \mathbb{N}_0)$.*
- *If c is neither non-recursive nor linearly recursive, c is non-linear.*

Definition 2 *Let p denote a predicate with clauses $c_1, \ldots, c_m$.*

- *If $c_1, \ldots, c_m$ constant, p is constant.*
- *If $c_1, \ldots, c_m$ are either constant or linear, and there exists a c_i which is linear, p is linear.*
- *If there exists a c_i which is non-linear, p is non-linear.*

The minima used in definition 1 reflects the desire to maintain a conservative form of granularity analysis which under-estimates the communication overhead. Type approximations might introduce negative values for $\|t\|_{\text{lat}} - \|t'\|_{\text{lat}}$ and hence an intersection with the set of non-negative numbers $\mathbb{N}_0$ is used to keep the communication growth rates non-negative.

Example 5 *The second clause of the `sum/2` predicate listed in figure 2 is linearly recursive and thus a candidate for being linear. If type analysis were equipped with a domain which includes integers, $t_1^{\#}, t_2^{\#}$ and $t_1^{\#'}, t_2^{\#'}$ might be inferred to be integers. Therefore $\min(\{\|t\|_{lat} - \|t'\|_{lat} \mid t \in t_1^{\#}, t' \in t_1^{\#'}\}) = \min(\{\|t\|_{lat} - \|t'\|_{lat} \mid t \in t_2^{\#}, t' \in t_2^{\#'}\}) = \min(\{0\}) = 0$ indicating that the amount of computation can grow in a way which is unconstrained by the communication. Thus the second clause of `sum/2` is non-linear, and indeed, `sum/2` can warrant parallel evaluation.*

The relationship between $t_i^{\#}$ and $t_i^{\#'}$ has to be clarified for recursive types. A recursive type characterises a set of recursively defined terms so that lists of various lengths might, for instance, be assigned the same type. In particular $\|t\|_{\text{lat}} - \|t'\|_{\text{lat}}$ can collapse to zero even though the arguments which correspond to t and t' might actually be different sizes. As a consequence of this inaccuracy, linear atoms might be categorised as non-linear atoms. However, the type analysis which underpins producer and consumer analysis and therefore schedule analysis, can be simply adapted to calculate more realistic values for $\|t_i\|_{\text{lat}} - \|t'_i\|_{\text{lat}}$. This modification is detailed in section 5.

4 Coalescing atoms into threads

Schedule analysis was motivated by the observation that schedule activity can be reduced by increasing the proportion of atoms which are totally ordered. A thread defines a total ordering on a set of atoms, and therefore, by maximising the length of the threads, scheduling is improved. The length of the threads is, in turn, maximised by minimising the number of threads. In addition, by reducing the number of threads, the number of data-dependencies which exist between the threads is also likely to be reduced. The number of data-dependencies is significant because it can predict scheduler activity. For cyclic data-dependencies in particular, the scheduler might have to alternate the evaluation of the threads. Thus, the fewer the number of cyclic data-dependencies, the fewer the number of times each thread is likely to be suspended and resumed.

In practice, since coroutining tends to be used infrequently, a high proportion of clauses do not possess cyclic data-dependencies between their atoms. Hence, schedule analysis typically generates a small number of threads for a clause, usually just one, without incurring any cyclic data-dependencies between the threads. Once the granularity of the atoms of a clause is considered, the number of threads tends to increase, and thus there is more scope for introducing cyclic data-dependencies. Cyclic data-dependencies which originate from the clause, for instance, due to coroutining, cannot be removed. Extra cyclic data-dependencies can be introduced, however, by the way the threads are generated. It is these additional cyclic data-dependencies which, whenever possible, should be avoided. If cyclic data-dependencies can be removed completely, so that the data-dependencies flow left-to-right between the threads, depth-first scheduling can be performed in a way which avoids the creation of any suspensions. Example 2 illustrates a valid though sub-optimal partition which introduces an extra cyclic data-dependency and consequently degrades scheduling.

To explain serialisation analysis, some notation from schedule analysis has to be recalled (King and Soper, 1991). Schedule analysis concerns the division of the atoms ($q \in$) Q_p of a clause p into into threads, using a relation δ_p^+ on Q_p, which over-estimates the data-dependencies between the atoms of p. Thus, if a data-dependence exists from q to q' in p, $\langle q, q' \rangle \in \delta_p^+$. Cyclic data-dependencies between two atoms q and q', identify data-dependencies which can only be resolved at run-time with a scheduler by allocating q and q' to different threads. Pairs of atoms which have to be separated in this way, are indicated by a relation σ_p on Q_p, derived from δ_p. Specifically, if $\langle q, q' \rangle \in \sigma_p$, q and q' are to be allocated to different threads. Colouring Q_p with σ_p leads to a partitioning of Q_p into $\{Q_p^1, \ldots, Q_p^t\}$, each Q_p^i defining the constituent atoms of the ith thread. A total ordering o_p^i is assigned for each thread, chosen so as not to contradict any data-dependency in δ_p. The cumulative effect of these total orderings is given by $\tau_p = o_p^1 \cup \ldots \cup o_p^t$. A safety result asserts a condition for which the work required of the run-time scheduler can be safely reduced from ordering the atoms to ordering threads. The condition is stated as theorem 1.

Theorem 1 *If $\tau_p \cup \delta_p^+$ has no more non-trivial cycles than δ_p^+, then for any possible query, a scheduler can interleave $o_1, \ldots, o_t$ so as not to contradict the data-dependencies on Q_p.*

In the notation of schedule analysis, serialisation analysis has to separate atoms q and q' such that $\langle q, q' \rangle \in \sigma_p$, allocate non-linear atoms to different threads and, in addition, reduce the number of cyclic data-dependencies between the threads. Non-linear atoms can be simply dealt with by augmenting σ_p with a relation γ_p on Q_p to indicate the pairs of non-linear atoms of p which have to separated. Colouring Q_p with $\sigma_p \cup \gamma_p$ instead of with σ_p, complies with both σ_p and γ_p. Thus, q and q' which are either both non-linear or satisfy $\langle q, q' \rangle \in \sigma_p$, will be allocated to different threads.

Definition 3 *γ_p on Q_p is defined by: $\langle q, q' \rangle \in \gamma_p$ if and only if q is non-linear and q' is non-linear.*

Thus colouring Q_p with $\sigma_p \cup \gamma_p$ leads to a partition $\{Q_p^1, \ldots, Q_p^t\}$ in which $q \in Q_p^i$, $q' \in Q_p^j$ for $i \neq j$ if either $\langle q, q' \rangle \in \sigma_p$ or $\langle q, q' \rangle \in \gamma_p$. Reducing the number of cyclic data-dependencies between the threads is more subtle and involves modifying the way $\sigma_p \cup \gamma_p$ is coloured, specifically, a sequential colouring algorithm (Matula *et al.*, 1972) is adapted. A sequential colouring algorithm colours each q_i in turn, according to how the preceding $q_1, \ldots, q_{i-1}$ are coloured. Thus q_1 is allocated to Q_p^1 and any proceeding q_i is allocated to the Q_p^j with the least j such that $\langle q_i, q \rangle \notin \sigma_p \cup \gamma_p$ for all $q \in Q_p^j$. The crucial point to note is that colouring Q_p in this way can itself introduce extra cyclic data-dependencies, as illustrated in example 6.

Example 6 *Consider again the recursive clause of the `fib/2` predicate listed in figure 1, and suppose that $Q_{\texttt{fib/2}} = \{q_1, \ldots, q_5\}$ where q_1 = `N1 is N - 1`, q_2 = `N2 is N - 2`, q_3 = `fib(N1, F1)`, q_4 = `F is F1 + F2`, q_5 = `fib(N2, F2)`. The relations $\delta^+_{\texttt{fib/2}}$, $\sigma_{\texttt{fib/2}}$ and $\gamma_{\texttt{fib/2}}$ for non-linear q_3 and q_5 are given in figure 3. Colouring Q_p with $\sigma_{\texttt{fib/2}} \cup \gamma_{\texttt{fib/2}}$ by the sequential colouring algorithm leads to the two threads $Q^1_{\texttt{fib/2}} = \{q_1, q_2, q_3, q_4\}$ and $Q^2_{\texttt{fib/2}} = \{q_5\}$. $Q^1_{\texttt{fib/2}}$ and $Q^2_{\texttt{fib/2}}$ are the two threads discussed in example 2 which possess a cyclic data-dependency.*

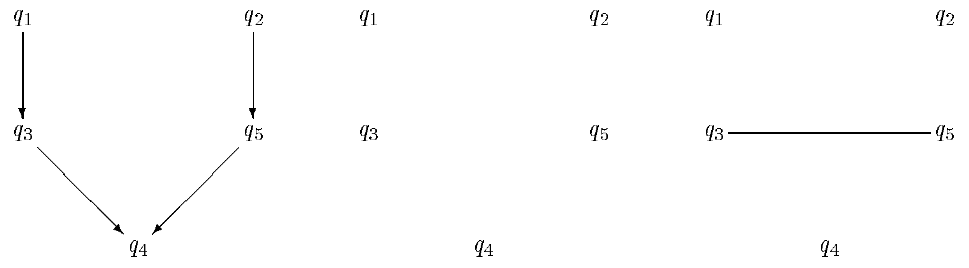

Figure 3: $\delta^+_{\texttt{fib/2}}$, $\sigma_{\texttt{fib/2}}$ and $\gamma_{\texttt{fib/2}}$

The creation of extra cyclic data-dependencies can be avoided by colouring Q_p in an order which, whenever possible, ensures that the data-dependencies occur left-to-right in $\{Q_p^1, \ldots, Q_p^t\}$. Left-to-right data-dependencies are guaranteed by renumbering Q_p, and instead of allocating q_i to Q_p^j with the least j, allocating q_i to Q_p^j with the last j. Since σ_p corresponds to the cyclic data-dependencies in δ_p^+, $\delta_p^+ \setminus \sigma_p$ is acyclic and therefore Q_p can be renumbered so that if $q_i, q_j \in Q_p$ and $\langle q_i, q_j\rangle \in \delta_p^+ \setminus \sigma_p$, $i \leq j$. Since $\delta_p^+ \setminus \sigma_p$ is acyclic, the renumbering is well-defined in the sense that $\delta_p^+ \setminus \sigma_p$ can always be renumbered, although a renumbering is not necessarily unique. Henceforth Q_p will be considered to be renumbered in this way. In addition, colour Q_p as follows. Allocate q_1 to Q_p^1 as before. Suppose that, so far, colouring has generated $\{Q_p^1, \ldots, Q_p^t\}$ and q_i has yet to be coloured. If $\langle q_i, q\rangle \notin \sigma_p \cup \gamma_p$ for all $q \in Q_p^t$ allocate q_i to Q_p^t, otherwise assign a new colour to q_i and thus place it in Q_p^{t+1}. Since Q_p is traversed by following $\delta_p^+ \setminus \sigma_p$, if $\langle q_i, q_j\rangle \in \delta_p^+ \setminus \sigma_p$ then $i \leq j$ and therefore $q_i \in Q_p^k$ and $q_j \in Q_p^l$ for $k \leq l$. Hence, the only cyclic data-dependencies which exist between $\{Q_p^1, \ldots, Q_p^t\}$ are the cyclic data-dependencies which already exist in δ_p^+. Conversely, a non-cyclic data-dependence from one thread to another, can only exist from an earlier thread to a later thread. Example 7 illustrates how colouring Q_p this way avoids the creation any extra cyclic data-dependencies.

Example 7 *Continuing with the the recursive clause of the* `fib/2` *predicate listed in figure 1, define* $Q_{\texttt{fib/2}} = \{q_1, \ldots, q_5\}$ *as in example 6. Renumbering* Q_p *according to* $\delta^+_{\texttt{fib/2}} \setminus \sigma_{\texttt{fib/2}} = \delta^+_{\texttt{fib/2}}$ *might give* $Q_{\texttt{fib/2}} = \{q_1, q_1, q_3, q_5, q_4\}$. *Colouring with the modified method produces* $Q^1_{\texttt{fib/2}} = \{q_2, q_1, q_3\}$ *and* $Q^2_{\texttt{fib/2}} = \{q_5, q_4\}$ *which coincide with the favoured allocation strategy of example 3. In this case the renumbering involves a choice, though for every choice, no extra cyclic data-dependencies incurred.*

Two nice additional properties of the modified colouring method can be observed. First, renumbering Q_p induces a total ordering on each Q_p^i which does not contradict any data-dependence in δ_p^+. Thus each o_p^i is automatically defined. Second, since no extra cyclic data-dependencies are introduced between the threads, the safety condition is always satisfied. More exactly, if $\tau_p = o_p^1 \cup \ldots \cup o_p^t$, $\tau_p \cup \delta_p^+$ has no more non-trivial cycles than δ_p^+. These observations are formally asserted and proved as propositions 1 and 2.

Proposition 1 *If* $\langle q, q'\rangle \in o_p^i$ *and* $q \neq q'$, $\langle q', q\rangle \notin \delta_p^+$.

Proof 1 *If* $\langle q_j, q_k\rangle \in o_p^i$ *and* $q_j \neq q_k$, $j < k$. *Since* $j < k$ *and because of the renumbering,* $\langle q_k, q_j\rangle \notin \delta_p^+ \setminus \sigma_p$. *Thus either* $\langle q_k, q_j\rangle \notin \delta_p^+$ *or* $\langle q_k, q_j\rangle \in \sigma_p$. *Because* $q_j, q_k \in Q_p^i$, $\langle q_k, q_j\rangle \notin \sigma_p$ *and thus* $\langle q_k, q_j\rangle \notin \delta_p^+$ *as required.* $\square$

Proposition 2 $\tau_p \cup \delta_p^+$ *has no more non-trivial cycles than* δ_p^+.

```
procedure serialise(in n, Q_p, δ_p^+, σ_p, γ_p ; out {Q_p^1, ..., Q_p^t}, {o_p^1, ..., o_p^t})
begin
   topological_sort(n, δ_p^+, σ_p, Q_p);
   t := 1;
   Q_p^1 := {};
   for i := 1 to n do
   begin
      if there exists q ∈ Q_p^t such that ⟨q_i, q⟩ ∈ σ_p ∪ γ_p then
      begin
         t := t + 1;
         Q_p^t := {q_i}
      end
      else
         Q_p^t := Q_p^t ∪ {q_i}
   end;
   for i := 1 to t do
      o_p^i := {⟨q_j, q_k⟩ | q_j ∈ Q_p^i, q_k ∈ Q_p^i, j ≤ k}
end
```

Figure 4: The serialisation analysis algorithm.

Proof 2 *Suppose, for the sake of a contradiction, that there exists a non-trivial cycle in* $\tau_p \cup \delta_p^+$ *which is not in* δ_p^+*. Since the cycle cannot be confined to one thread, it must straddle at least two threads,* Q_p^m *and* Q_p^n *with* $m < n$*. Specifically, there exists* $\langle q_i, q_j \rangle \in \tau_p \cup \delta_p^+$ *such that* $q_i \in Q_p^n$ *and* $q_j \in Q_p^m$*. Since* $\langle q_i, q_j \rangle \notin \tau_p$*,* $\langle q_i, q_j \rangle \in \delta_p^+$*, and because* $\langle q_i, q_j \rangle \notin \sigma_p$*, it follows that* $i < j$*. Hence* $m \geq n$*, which is a contradiction.* □

Propositions 1 and 2 are useful because they enable serialisation analysis to be simplified. First, Q_p has to be reordered just once to define the total orderings $o_p^1, \ldots, o_p^t$, whereas previously $o_p^1, \ldots, o_p^t$ had to be formed separately. Second, and more significantly, the safety condition does not have to be checked, which before, was the most complicated part of schedule analysis. Thus, better scheduling is achieved by a simpler compilation technique! Figure 4 presents the central part of the serialisation analysis algorithm.

5 Notes on implementation

Figure 4 describes the stages of serialisation analysis which are relevant to an individual clause p, assuming that the relations δ_p^+, σ_p and γ_p on Q_p are already derived. The relation γ_p can be computed by traversing the reduced call graph for the program in a bottom-up way, classifying atoms, clauses and predicates in the manner defined in section 3. The reduced call graph simplifies classification since, for a non-recursive clause, the atoms of the clause are guaranteed to be already categorised before the classification is extended to the clause. Furthermore, the reduced call graph groups together the predicates participating in recursion so that linearly recursive predicates can be straightforwardly identified.

The stages of serialisation analysis presented in figure 4 divide into renumbering Q_p, which amounts to topologically sorting Q_p according to $\delta_p^+ \setminus \sigma_p$; colouring Q_p with $\sigma_p \cup \gamma_p$; and finally inferring $o_p^1, \ldots, o_p^t$ from the reordered Q_p. By representing the sets $Q_p^1, \ldots, Q_p^t$ as ordered sets, for instance, as lists, the final stage of serialisation analysis can be eliminated. The idea is to construct each o_p^i on-the-fly as Q_p^i is generated and represent o_p^i by the total ordering associated with the ordered set Q_p^i. This is simply accomplished by replacing the union of the set Q_p^t with $\{q_i\}$ with an append which inserts q_i at the end of the ordered set Q_p^t.

```
procedure topological_sort(in n, δp+, σp; inout Qp)
begin
   j := 1;
   repeat
   begin
      s := 0;
      for i := 1 to n − j do
         if ⟨qj, qi⟩ ∈ δp+ \ σp then
         begin
            t := qi;
            qi := qj;
            qj := t;
            s := 1
         end;
      j := j + 1
   end
   until s = 0 or j = n
end
```

Figure 5: The topological sort algorithm.

An algorithm for topologically sorting Q_p with respect to $\delta_p^+ \setminus \sigma_p$ is presented in figure 5. The algorithm is a variant of bubble-sort. Bubble-sort is an attractive way of sorting Q_p because of its simplicity and also because n, the number of atoms, is typically small and therefore the worse case quadratic complexity of bubble-sort is acceptable. Moreover, concurrent logic programs are often written in a Prolog style with data-dependencies flowing left-to-right among the atoms of a clause. Thus, in many cases Q_p will be already ordered according to $\delta_p^+ \setminus \sigma_p$. Bubble-sort can exploit this, so that sorting can frequently be performed in just n comparisons.

Section 3 mentioned that $\|t\|_{\mathrm{lat}} - \|t'\|_{\mathrm{lat}}$ for $t \in t_i^{\#}$ and $t' \in t_i^{\#'}$ can collapse to zero when $t_i^{\#}$ and $t_i^{\#'}$ are recursive types, since $t_i^{\#}$ and $t_i^{\#'}$ are often the same type. The type analysis upon which producer and consumer analysis is founded, however, can be amended to compute more accurate values for $\|t\|_{\mathrm{lat}} - \|t'\|_{\mathrm{lat}}$. A recursive type can assume a number of different forms, for instance, a list type might occur as either a null list type or as a type for a list pair, and so on. Thus, to guarantee termination, these various type combinations have to be recognised and replaced with the recursive type. Once the recursive type is introduced $\|t\|_{\mathrm{lat}} - \|t'\|_{\mathrm{lat}}$ can reduce to zero, but if $\|t\|_{\mathrm{lat}} - \|t'\|_{\mathrm{lat}}$ is computed beforehand in terms of the the raw types, more accurate communication growth rates can often be derived. Thus accuracy can be preserved simply by computing the communication growth rate during type analysis rather than after type analysis.

Serialisation analysis has potential for parallel evaluation since different sub-graphs of the reduced call graph can be classified simultaneously and each clause of the program be renumbered and coloured separately.

6 Future work

Back-communication can be expensive on a multi-processor, for if those processes which participate in the back-communication are allocated to different threads on different processors, bindings might have to be alternately copied from one processor to another. Serialisation analysis might be improved by ensuring that back-communicating threads are kept together on a single processor.

It might be possible to extend serialisation analysis to deduce process placement tactics. As a refinement to the classification, the non-linear class could be divided according to the type of recursion that is employed in a non-linear predicate. By recognising frequently occurring forms of recursion, for instance, the sort employed in divide-and-conquer problems, recursive non-linear processes could be allocated to processors configured in a suitable topology. In the case of the `quicksort/2`, for example, the recursion would induce a two part division into non-linear atoms which could be allocated to a multi-processor configured as a ring of processors. At each level of recursion, one of the non-linear processes would be distributed to the next processor in the ring to effect a simple form of load balancing.

Schedule analysis has already been incorporated into a compiler (King and Soper, 1991) and been found to be an effective way of ordering atoms. An initial examination has suggested that serialisation analysis can be straightforwardly inserted into the compiler. Thus, future work will involve implementing serialisation analysis to quantify the benefits which the analysis can bring to a parallel implementation.

7 Conclusions

Serialisation analysis unifies granularity analysis and schedule analysis into a coherent analysis for partitioning a program into grains for parallel evaluation and ordering grains for sequential evaluation. The division into grains reduces the time spent communicating and ordering of the grains decreases the time spent scheduling.

A useful form of granularity analysis is defined by isolating two classes of fine-grained predicates which are frequently occurring and simple to identify. Basically, non-recursive processes are regarded as constant, and linearly recursive processes are regarded linear if the communication can be shown to dominate the computation over the lifespan of the process. Processes which are neither constant nor linear in this sense are allocated to different grains. This adapts to concurrent logic programs an extension of a simple and effective granularity control technique which has been applied successfully to functional programs (Hudak and Goldberg, 1985).

The schedule analysis is applied by considering data-dependencies between atoms, so that fine-grained processes can be ordered and coalesced into more coarse-grained units. Serialisation analysis combines granularity and schedule analysis in such a way as to satisfy the constraints imposed by each analysis. In particular it guarantees a reduction in the number of suspensions for depth-first scheduling, without incurring any extra scheduling overhead. In addition, serialisation analysis automatically satisfies a satefy result, removing the need for a complicated safety check, and therefore leads to an algorithm which is straightforward to implement.

References

D. W. Matula, G. Marble & J. D. Isaacson (1972). *Graph theory and computing*, chapter Graph colouring algorithms, pp. 109–122. Academic Press, London. Edited by R.C. Read.

Debray, S. K. (1989). "Static Inference of Modes and Data Dependencies in Logic Programs", *ACM Transactions on Programming Languages and Systems*, 11 (3): 418–450.

Debray, S. K., N. Lin, & M. Hermenegildo (1990). "Task Granularity Analysis in Logic Programs", *in Proceedings of the Conference on Programming Languages Design and Implementation*, White Plains, New York. ACM.

Foster, I. & W. Winsborough (1991). "Copy Avoidance through Compile-Time Analysis and Local Reuse", *in Proceedings of the 1991 International Logic Programming Symposium*. MIT Press.

Hudak, P. & B. Goldberg (1985). "Serial Combinators: Optimal Grains for Parallelism", *in Second Conference of Functional Programming Languages and Computer Architectures*, pp. 382–399, Nancy, France. Springer-Verlag.

King, A. & P. Soper (1990). "Granularity Analysis of Concurrent Logic Programs", *in The Fifth International Symposium on Computer and Information Sciences*, Nevsehir, Cappadocia, Turkey.

KING, A. & P. SOPER (1991)a. "Reducing Scheduling Overheads for Concurrent Logic Programs", *in International Workshop on Processing Declarative Knowledge*, Kaiserslautern, Germany.

KING, A. & P. SOPER (1991)b. "A Semantic Approach to Producer and Consumer Analysis", *in International Conference on Logic Programming Workshop on Concurrent Logic Programming*, Paris, France.

TICK, E. (1990). "Compile-Time Granularity Analysis for Parallel Logic Programming Languages", *New Generation Computing*, 7: 325–337.

TOUATI, H. & A. DESPAIN (1987). "An Empirical Study of the Warren Abstract Machine", *in Proceedings of the 1987 Symposium on Logic Programming*, pp. 114–124, San Francisco, California.

UEDA, K. & M. MORITA (1990). *A New Implementation Technique for flat GHC*, pp. 3–17. MIT Press, Jerusalem.