Programmer-Oriented Visualisation of Parallel Logic Program Execution

Steven Prestwich
Tony Kusalik

ECRC TR 94-07

February 1994

Abstract

There are several visualisation tools for parallel logic programming languages reported in the literature. These systems vary in the level of detail presented to the user, but generally aim to portray the execution tree, interprocessor communications, or both. Although this level of detail is useful to the systems developer, it is hard for the (non-expert) applications programmer to interpret. Considerable mental effort may be needed to relate the picture to the program, and to decide how a poor parallel program should be modified. This report describes a new visualisation tool called ParSee which analyses trace data and generates a simple static picture. The picture is a high-level view of the execution, showing only features intended to be of direct interest to the programmer. It can help explain unexpected speedup or slowdown and guide parallelisation.

Contents

1 Introduction **2**
- 1.1 Background and related systems . . . 3
- 1.2 ParSee overview . . . 3
- 1.3 Terminology . . . 4

2 Trace generation **6**
- 2.1 The prototype . . . 6
- 2.2 Future work . . . 7

3 Trace analysis **8**
- 3.1 The sequential execution metric . . . 8
- 3.2 The incurred overhead metric . . . 9
- 3.3 The potential overhead metric . . . 10
- 3.4 The processor utilisation metric . . . 10
- 3.5 Meta-metrics . . . 10
  - 3.5.1 Usefulness . . . 11
  - 3.5.2 Monotonicity . . . 11
  - 3.5.3 Orthogonality . . . 11
  - 3.5.4 Time locality . . . 11
  - 3.5.5 Relativity . . . 11
  - 3.5.6 Robustness . . . 11
  - 3.5.7 Tractability . . . 12
- 3.6 Future work . . . 12

4 Picture generation **13**
- 4.1 Encoding information by colour . . . 13
- 4.2 User interface features . . . 15
- 4.3 Data handling features . . . 16
- 4.4 Future work . . . 16

5 Concluding remarks **19**

Chapter 1

Introduction

This report describes a new visualisation tool called ParSee. It analyses trace information and presents the user with high level information in the form of a static colour picture. This picture filters out low-level details, and relates properties of the parallel execution (such as granularity) to each use of parallelism in the program source. Thus the effects of each use of parallelism on the execution can be observed, making the picture directly useful for finding the best parallelisation. It can also be used to help explain unexpected speedup or slowdown (for example the programs in [15]) by revealing the underlying causes.

There are several visualisation tools available which are applicable for parallel logic programming languages such as ElipSys, &-Prolog, Muse, Andorra and FGHC. These systems are typically developed for systems programmers, and concentrate on detail relevant to the underlying execution mechanism. While useful for debugging parallel scheduling algorithms, load balancing and other aspects of the implementation, this information is not relevant for most *application* programmers. There is typically too much detail, or the information is at an inappropriate level of abstraction, or both. Important properties such as granularity can be hard to extract from detailed information about interprocessor communication, the shape of the execution tree, which processors are executing which parts of the program, and so on. It may be difficult to relate the picture to the program source code. Finally, animated representations of the computation may take a very long time to view, requiring considerable concentration or a large investment of programmer time (for filtering irrelevant information).

The descriptions of these visualisation tools typically explain how users should interpret the images they see. Indicative situations or characteristic configurations are usually presented and explained. However, the output seen by programmers-in-the-field may not correspond to that of the selected examples. Ambiguous situations can and do occur, as well as complex ones involving amplification, interference, or occlusion caused by interacting effects. Also, automation of this analysis and interpretation is not supported.

Finding a good parallelisation of a program is a difficult task. There are typically many points in a program at which parallelism can be introduced (and possibly variations in how it can be introduced), and it is usually impractical to try every alternative. To find an optimal, or near-optimal parallelisation, the programmer needs to understand how each particular use of parallelism affects the overall behaviour of the program. In other words, a programmer needs to answer the question, "What is the effect on the program behaviour of introducing parallelism at this point?" The program effects might be properties such as granularity, whether the scheduler selects coarse-grained or fine-grained tasks, and so on.

ParSee is currently implemented in and for the OR-parallel logic programming system ElipSys [7]. For simplicity, ParSee is discussed in this report only in the context of OR-parallelism. However, the principles are applicable to AND-parallelism. Further, because the basic approach

does not rely at all upon logic programming concepts such as AND/OR trees, it is applicable to other high-level parallel languages.

1.1 Background and related systems

Execution visualisation tools such as ParTrace [4], VisAndOr [10], WamTrace [13], MUST [18] and VISTA [19] are "... invaluable help when implementing, debugging, and tuning parallel systems" [10]. They provide basic information about the progress of the computation, and the use of resources during the computation. This typically includes some representation of the search tree explored during the computation, the allocation of "work" to "processors", and communications between processors. Various graphical objects and constructs may be used to provide supplementary information. The more recent of these tools optionally use colour to encode information. In some, such as VISTA, colour is fundamental to the visualization paradigm. Key to the use of these tools by users is analysis and careful interpretation of the displayed detail.

ParTrace is the tracing tool for the ElipSys system. While ParTrace has been found useful by the ElipSys developers, application programmers report that it provides too much low-level detail, most of which is not directly useful, and worse, which buries the useful information. Moreover, the kind of information it provides is quite implementation dependent — it relates to internal details of the ElipSys system — and is not generally applicable. What application programmers say would be more useful is information which indicates whether or not coarse-grained parallelism is being achieved and exploited, and which predicates are performance bottlenecks.

VisAndOr, MUST and VISTA also provide a great deal of useful system programming-level detail. The descriptions of these tools include explanations of how such detail is to be interpreted. Consequently, application programmers can be expected to identify, when using the tools, clearly evident situations and problems. Such situations would include only one processor performing the work of a computation or a tight infinitely-recursive branch arising. However, many problems in performance debugging in parallel languages are more subtle. They are complex situations involving several factors interacting in various ways. In that case the underlying conditions or problems may not be obvious from the trace detail, in particular if there is an abundance of that detail. Further, if it is possible to formulate strategies or methodologies for interpretation of the information provided by a tool, it is natural to ask whether such interpretation or analysis can be performed automatically, and the user provided with a higher level of information.

ViMust [10] is a synchronized execution of both MUST and VisAndOr tracers. This is a useful mechanism as the two tracers give complementary views of the same information. A user thus has greater opportunity to determine the characteristics of the execution. Once again however, it is not clear whether it is any easier for an application programmer to understand the detail shown, or relate it back to the source program.

A well-known performance analysis tool for (sequential) Prolog is Gauge [9]. It provides execution profile information in a visual form. Histograms are shown indicating the frequency of such events as predicate calls, "deep failures", choice-point creation, etc. This information is related (back) to the program source rather than, for example, the search tree or proof tree. The aim of the information is to guide programmers in improving efficiency of their programs. However, Gauge is restricted to sequential computations.

1.2 ParSee overview

ParSee analyzes low-level trace information, and presents the user with high-level information in a static, at-a-glance colour picture. The component stages of the tool are described below, as well as

how it can be used to parallelise and fine-tune programs. As shown in Figure 1.1, ParSee produces a picture in three steps: *trace generation*, *trace analysis* and *picture generation*. This conforms to the usual structure of what may be called *user-oriented program analysis* tools [6]. Outside the realm of visualization of sequential or parallel logic program execution, other systems have been developed using this approach. Opium [5] is such a system, developed for the high-level debugging of logic programs.

Trace generation is described in Chapter 2, trace analysis in Chapter 3 and picture generation in Chapter 4. Chapter 5 draws some conclusions and indicates avenues for future work.

1.3 Terminology

To facilitate the following discussion, some terminology is now defined. A logic program consists of a set of predicate definitions. Each predicate definition consists of a list of (definite) clauses. Each clause has a head and a body. A clause head is a single atom and a clause body is a (possibly empty) conjunction of atoms. An atom is a predicate symbol plus a (possibly empty) list of arguments.

A logic program execution starts from a goal called the **query** and can be represented by an AND/OR tree. Each OR-node is associated with an atom which is a call to a predicate, and each clause matching this atom generates a disjunctive subtree. These subtrees are the **children** of the call. Each matching clause head spawns a child, and so a call will have no children if no clauses match. A predicate may either be sequential (the default) or parallel. In ElipSys, an OR-parallel predicate `p` with arity `n` is denoted as parallel by the declaration

```
:- parallel p/n.
```

At an OR-node corresponding to a sequential predicate the disjunctive subtrees are executed in sequence by backtracking, as in the standard Prolog strategy. At a node belonging to an OR-parallel predicate the disjunctive subtrees are executed as parallel tasks as far as possible given the number of available processors.

At any point of the computation an **active child** is defined as one which has started and not yet finished (and which is not suspended). An **active call** is a predicate call which has an active child. A predicate which has at least one active call instance is termed an **active predicate**.

Given a fixed query and number of processors, the **speedup** of a predicate `p` is the execution time with `p` sequential divided by the execution time with `p` parallel. That is, we use absolute speedup and not relative speedup. [1]

The overhead due to the use of parallelism is termed the **parallelism overhead**, or simply **overhead**. A concept used in this report is that of the **granularity** of parallel tasks (children). **Fine-grained tasks** are defined to be those with duration of the same order of magnitude as the overhead for a parallel call. Other tasks — with significantly longer execution times — are called **coarse-grained**. This definition is left deliberately vague because parallelism overheads are not constant.

[1] Absolute speedup is defined by Sun and Ni [17] as the ratio of the time taken by the best sequential program to the time taken by the parallel program on N processors, and relative speedup as the ratio of the parallel algorithm with 1 processor to the time with N processors.

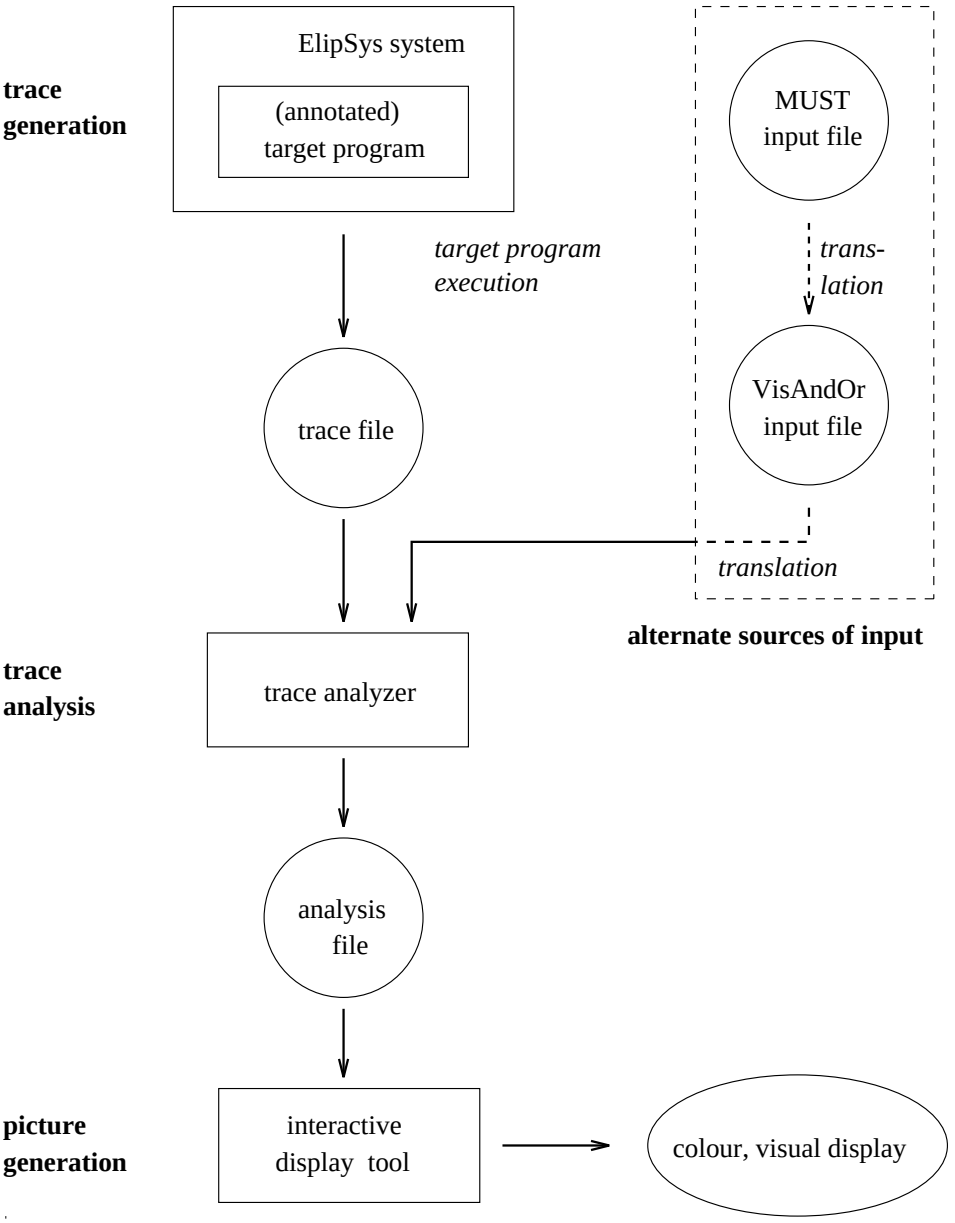


Figure 1.1: *Overview of the ParSee phases*

Chapter 2

Trace generation

The first phase of program analysis is execution of the subject program and collection of data on parallel execution activity. The raw data is collected in a **trace file** which consists of a set of **events**. An example of an event is the initiation of a parallel child computation.

2.1 The prototype

ParSee currently generates its own trace file as follows. The program source is automatically transformed so that each parallel call is annotated with special instructions. The transformed program is compiled and queried as usual. At each parallel call during execution, trace information is generated by the special instructions. This information is written to a set of trace files, one for each processor. Immediately following completion of the program execution, the files are merged to produce a single trace file. The use of multiple files avoids problems with parallel writes to a single file, and is merely an implementation detail. We are only interested in the single merged trace file.

The generated information is a set of events, which may be of two types:

- starting a child of a parallel call,
- ending a child of a parallel call by backtracking past its start.

Each event contains

- an identifier which signifies the parent call involved, with all siblings having a record of the same identifier;
- the name of the predicate corresponding to the call;
- the processor on which the parent call occurred;
- the processor on which the child task began;
- a time stamp (in milliseconds).

This scheme works for pure parallel search (all solutions) only. It cannot yet handle single-solution queries nor computations which use speculation or pruning (for example, via the commit operator). In such cases, a child may have a start event and no end event, and it is not known when tasks are completed.

Automatic instrumentation is easier and less error-prone for the user than manual annotation. It is also more efficient than executing the program by instrumented meta-interpreter because it does

not introduce an interpretation overhead [1] A more efficient scheme still would be to instrument the compiled code, that is annotate the program which the parallel logic program maps to (in parallel Fortran, C or whatever). This allows the instrumented program to exploit most of the same compiler optimisations as the original program, and also makes tracing of the *commit* construct easier. The disadvantage would come when changing to other languages or compilers. The trace generation procedure would have to be completely reimplemented. Our scheme is more portable.

2.2 Future work

Future versions of ParSee will be able to handle tasks aborted by pruning and speculation.

To handle programs with AND-parallelism, pruning, and suspension activity, and to achieve compatibility with other existing tools, ParSee is being extended to accept VisAndOr files [10] (MUST trace files are specific to the Muse execution model, but they can be converted to VisAndOr format). VisAndOr files also contain a set of time-stamped events which describe a computation. For OR-parallel computations these events are very similar to those in ParSee trace files. However, the precise format of the two types of file differ. VisAndOr files may also contain events specific to other types of parallelism, such as the "join" event in the AND-parallel language &-Prolog [11] [2]. With this more widely-used format, executions of OR-parallel, restricted AND-parallel, and mixed AND/OR-parallel computation can be supported by ParSee, and it can operate in conjunction with other tools on the same trace files.

We note that ParSee is already applicable to at least one form of AND-parallelism. In [1] implementations of AND-parallelism by transformation to OR-parallelism are described, and ParSee can of course be applied to the result. We will investigate more recent versions of AND-parallelism in the future.

[1] However, it is acknowledged that the overhead associated with interpretation can be reduced or eliminated via partial evaluation.

[2] The format is, in fact, general enough that files can be used directly for purposes other than visualisation, such as analysis for determination of optimal processor allocation [8]

Chapter 3

Trace analysis

The trace file contains information on the children of all parallel calls. The trace analysis uses this to compute information on each predicate. The results are summarised in an **analysis file**. Low-level details which are considered not directly useful to the programmer are filtered out during the analysis. For example, it is recorded whether there are idle processors, but not which processors are idle. Information collected on each call is summarised for the relevant predicate, and thus the computation tree is abstracted away completely.

For purposes of analysis, the computation is divided into small time slots of identical duration, and trace metrics are calculated for each slot. The value of a metric for a given time slot is the mean value over that period of a function (of time) called a **trace abstraction**. A trace abstraction is defined over the entire trace.

The metrics for a particular time slot and predicate are grouped together to form an "entry" in the analysis file; i.e. the analysis file contains one entry per predicate per time slot. Thus entries summarise the events during a time slot which are relevant to each predicate. Currently, each entry contains three numerical quantities between 0 and 1, which are the values of:

σ: the sequential execution metric;

o_i: the incurred overhead metric.

o_p: the potential overhead metric;

These metrics are used to set the intensities of the three primary colours in the display phase. There is in addition one predicate-independent entry per time slot which is

v: the processor utilisation metric.

In Section 3.5 we discuss some important "meta-metrics" — that is, properties of the trace metrics themselves.

3.1 The sequential execution metric

The **sequential execution metric** σ indicates a lack of exploited parallelism. This may either be caused by

(i) a lack of any inherent parallelism to exploit, or

(ii) lack of exploitation of inherent parallelism.

Case (i) applies if the parallel predicate has only deterministic calls; that is, no call has more than one child and hence there is no inherent parallelism. Case (ii) applies if there is inherent parallelism but there were insufficient processors to exploit it, so that the children were called in sequence. At present there is no way to tell which case applies, but this will be remedied in the future.

We shall say that a predicate is being executed sequentially at some time if, for each call of the predicate, there is at most one active child. Denote this by a function

$$seq(t) = \begin{cases} 1 & \text{if the predicate is sequential at time } t \\ 0 & \text{if the predicate is parallel at time } t \end{cases}$$

Then the sequential execution metric $\sigma(t_1, t_2)$ is defined as the mean value of $seq(t)$ as t ranges over the time slot (t_1, t_2); that is, the proportion of time during the interval when $seq(t)$ was equal to 1.

3.2 The incurred overhead metric

The programmer does not need to know what the scheduler does in detail, but it is important to know when fine-grained tasks are causing poor use of the parallel system. Fine-grained tasks incur a high proportion of parallelism overheads, and these overheads seriously degrade speedup, sometimes even causing slowdown. They may also lead to strange speedup curves [15]. It is therefore important to know where and how often these overheads occur.

The **incurred overhead metric** o_i is based on the proportion of processing time during a time slot spent on parallelism overhead. The latter is based on an estimate of the typical overhead. If at least half the time was spent on overhead then the parallelism was not worthwhile during that time slot, and $o_i = 1$. If no time was spent on overhead then there is good use of parallelism, and $o_i = 0$.

Parallelism overhead mainly occurs when a task moves or is moved from one processor to another. Each such event can easily be detected by comparing the processor on which a child task begins with the processor on which the parent call occurs. If they are different then an overhead was incurred. A child task which has the same processor as its parent call will be called a **local child**, and one with a different processor will be called a **migrant child**. The migrant children are the ones which incur overhead.

A processor is classed as being busy on a predicate `q` if it is performing work on a `q` child. It is classed as busy on parallelism overhead for `q` if it is involved in setting up a migrant task for `q`. Say that the trace shows a migrant parallel task beginning at time t_i on processor A, spawned from processor B. The parallelism overhead is given as a duration of time, say t_o. It is inferred that processors A and B were busy on parallelism overhead during the time interval $(t_i - t_o, t_i)$ because they must have communicated with each other over that time period to initiate the migrant child task. These assumptions may not hold for all systems, and the calculation may need to be adapted.

Now let $p_b(t)$ denote the total number of processors busy on the predicate at time t, and $p_o(t)$ the number of processors busy with the parallelism overhead for the predicate at time t. Both $p_b(t)$ and $p_o(t)$ can be inferred from a count of the number of start and end events for the predicate prior to t, and which tasks are local and which migrant. [1] $o_i(t_1, t_2)$ is then defined as the mean value of $\min(1, 2\frac{p_o(t)}{p_b(t)})$ as t ranges over the time slot (t_1, t_2).

ParSee is tuned to a specific parallel system by choosing an appropriate value for t_o. This has been done manually so far, but self-calibrating programs can also be designed to compute these estimates. In future, ParSee will be able to calibrate itself on demand when porting it to a new system.

[1] In some systems there may be complications caused by suspended calls, but we shall not discuss this here.

3.3 The potential overhead metric

Whereas o_i estimates the incurred overhead due to fine-grained calls, o_p estimates the *potential* for further overhead if the predicate is allocated more processors. This might occur if

- more processors are specified at runtime, or
- if another parallel predicate is sequentialised, thus possibly freeing processors to be allocated to this predicate.

Hence high values of o_p mean that adding processors may be dangerous, incurring more overheads and giving a poor speedup curve. Conversely, low values of o_p mean that adding more processors will not degrade performance.

o_i uses a count of migrant children to estimate incurred overhead. For o_p, on the other hand, local children which are *potentially* migrant are used to estimate potential overhead. To do this some assumptions must be made concerning the scheduling algorithm of the language implementation. It is assumed that at least one child is always local no matter how many processors are available. This "always local" child is assumed to be chronologically the first. Then all local children except this one are potentially migrant. For other parallel languages these assumptions may not hold, and the definition must be changed accordingly.

Let $p'_o(t)$ be the number of processors busy with such children at time t. Then $o_p(t_1, t_2)$ is defined analogously to o_i, except that $p'_o(t)$ is used instead of $p_o(t)$: $o_p(t_1, t_2)$ is the mean value of $\frac{p'_o(t)}{p_b(t)}$ over the time slot (t_1, t_2).

3.4 The processor utilisation metric

It is useful to know if and when there are idle processors. The **processor utilisation metric**, v, indicates this. A nonzero value means that not all processors are working during the time slot and there is potential for further exploitation of parallelism. This metric does not relate to individual predicates, but to the computation as a whole.

$v(t_1, t_2)$ is defined as the mean value of $idle(t)$ as t ranges over the time slot (t_1, t_2), where

$$idle(t) = \frac{n_p - b_p(t)}{n_p - 1}$$

denotes the **idleness** of the system at time t. In this expression $b_p(t)$ is the number of processors busy at time t, and n_p is the number of processors specified at runtime. The value of $b_p(t)$ can be deduced from the trace file by counting all the start and end events prior to t. If a processor has more start events than end events before t, then it is busy; otherwise, it is idle. Since each processor is either idle or not idle and at least one processor is always busy, $1 \leq b_p(t) \leq n_p$, and therefore $0 \leq idle(t) \leq 1$.

3.5 Meta-metrics

Many trace metrics can be defined in an attempt to capture the properties of a parallel execution. A danger is that a poor set of metrics is chosen, or too large a set. It is therefore prudent to define a set of **meta-metrics** [3] to evaluate the metrics. Such a group of meta-metrics is given below and used to evaluate the metrics defined for ParSee. Unfortunately some of these meta-metrics can only be evaluated subjectively, but they are still very useful guidelines when defining new metrics.

3.5.1 Usefulness

Each metric should be useful to a programmer, either by revealing important features of the trace, or by inspiring some program modifications such as sequentialisation or optimising transformations. The usefulness of the metrics will be shown in future reports on the application of ParSee to program development.

3.5.2 Monotonicity

A general principle we aim for in ParSee is that the darker a colour, the worse the corresponding property. This aids picture interpretation. For each metric defined above, the greater its value the stronger (worse) the corresponding property for the predicate in question.

3.5.3 Orthogonality

Another general principle is that each metric should be meaningful when considered alone, and need not be considered in relation to any other metric. A user can thus focus on each metric in isolation if necessary. This is particularly important if the number of metrics which can be displayed simultaneously is limited. [2] The defined metrics observe this meta-metric.

3.5.4 Time locality

Each metric $m(t_1, t_2)$ must say something about the execution during the time slot (t_1, t_2). That is, it should not be necessary to view the entire execution to deduce what is happening at any particular time. However, this does not rule out looking at what is happening to other predicates at the same time — for example, to observe that one predicate is being allocated processors at the expense of another predicate.

The values of each of ParSee's metrics do indeed depend upon instantaneous properties of an execution, though they may require analysis of other parts of the trace to deduce these properties. For example, to reconstruct the number of idle processors at a given time, it is necessary (given the form of our trace files) to analyse the trace history prior to that time. However, this is an implementation detail which is invisible to the user.

3.5.5 Relativity

The final picture (which depends upon the metrics) should not depend qualitatively upon the frame of reference chosen for the time slots — that is, their length and their location in time. Recall that all the metrics are defined as the mean of a function over a time slot. In a future report we shall show that all such metrics are relative, and that this is an important meta-metric. [3]

3.5.6 Robustness

A metric must not be sensitive to small inaccuracies in timings. That is, small variations in events times must produce small variations in the metric. If this does not hold, the metric may be unreliable because its values rely critically upon the accurate measurement of the event times. In a future report we shall show that each metric is robust.

[2] In the current ParSee implementation we are limited to the 3 prime colours and therefore 3 metrics.

[3] This meta-metric is called "relativity" because it is reminiscent of the Galilean principle of relativity in physics: that physical properties are independent of the frame of reference.

3.5.7 Tractability

Trace files may be very long, so each metric must not have analysis time which grows excessively with increasing file length. Ideally, each metric's analysis time should be linear in the file length.

The time taken to calculate each of ParSee's metrics is linear in the trace file length. Informally, this can be seen by considering that each metric is computed by a fixed number of passes through the trace file, and all necessary information is stored in arrays and hash tables which have constant retrieval time.

3.6 Future work

The trace analysis can be extended in several ways to analyse other properties, for example:

- OR-parallel calls matching no clause heads
- OR-parallel calls matching one clause head
- wasted speculation (parallel work which leads to no solutions and which is aborted when an alternative task succeeds)
- lack of inherent parallelism (as opposed to lack of exploited parallelism, currently measured by σ)
- unexploited parallelism due to insufficient processors

The analysis can also be extended to handle AND-parallelism in a similar way to OR-parallelism. This will be necessary to provide support for VisAndOr trace files, as discussed in Section 2.2.

Chapter 4

Picture generation

The analysis file produced by the previous phase is displayed via an interactive tool. An example of a ParSee picture is shown in Figure 4.1(a).

The picture consists of a set of vertical strips, each strip representing the effect of a parallel predicate on the execution as the computation proceeds. [1] The vertical axis represents "real time". The lighter the colour at any point in the strip, the better the use of parallelism. The hue of darker colours indicates what is going wrong. There is an additional strip to indicate good and poor processor utilisation.

4.1 Encoding information by colour

ParSee uses the subtractive cyan-magenta-yellow colour scheme to display its information. In this scheme (pure) cyan, magenta and yellow are primary colours, and all other colours can be obtained by mixtures of them. Thus, for example, blue is obtained by mixing cyan and magenta; green from cyan and yellow.

The basic principle of the colour encoding is that "lighter is better", and that the darker or closer to black a colour is, the worse the represented situation is (in terms of useful exploitation of parallelism). The hue indicates *what* the problems are. Primary colours are used to indicate individual metrics, while composite colours correspond to combinations of (potential) problems and efficiencies. Thus blue, a combination of magenta and cyan, indicates two problems occurring simultaneously. Dark blue indicates a worse situation than light blue. These properties follow from the metric properties defined in Section 3.

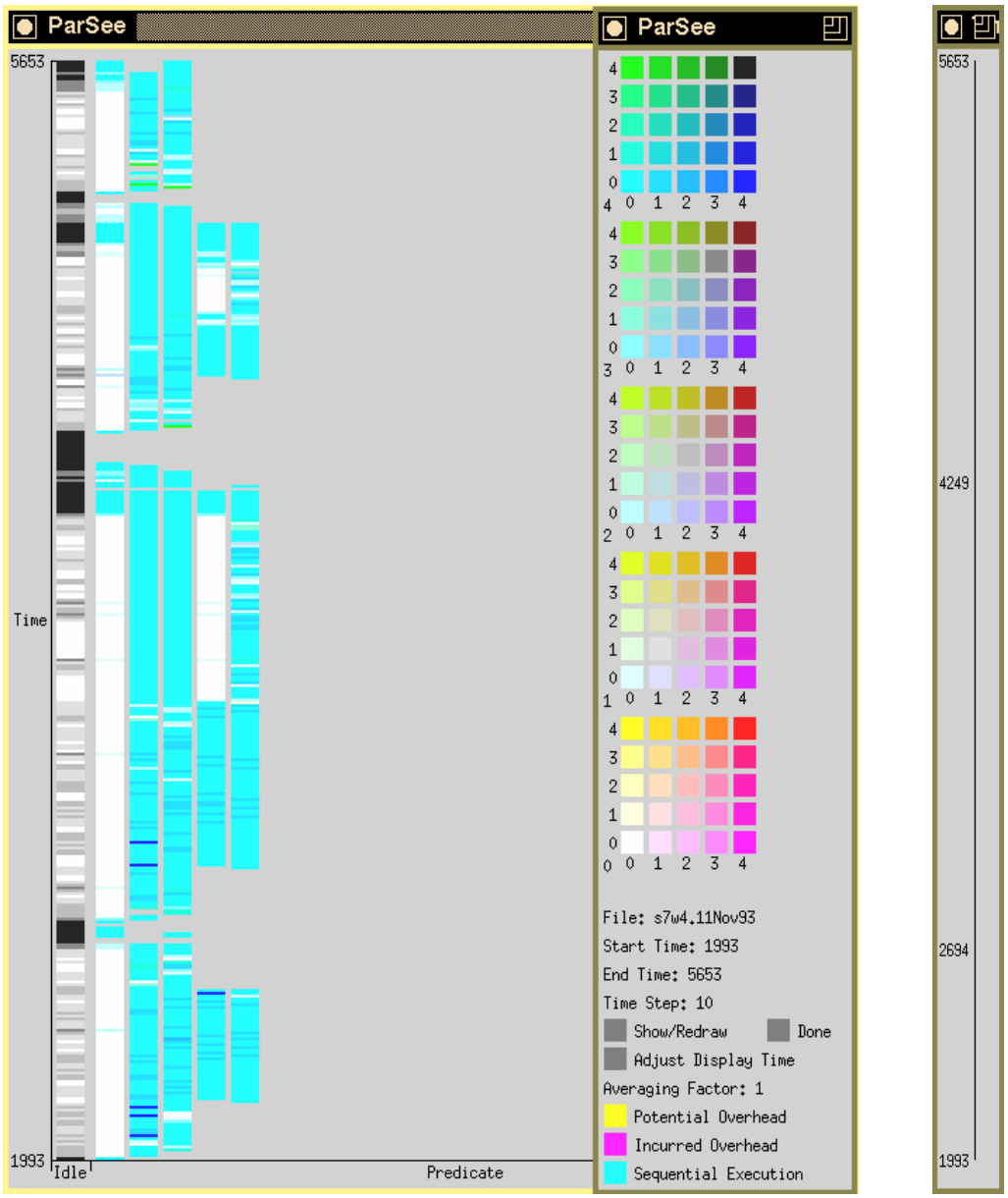
Currently, five levels of each primary colour can be shown, for a total of $5^3 = 125$ possible colours (primary and composite). The limit of five colour levels was initially imposed by the implementation system (i.e. only 128 colours available in the colour map). However, limitations of the human visual system (only being able to accurately discern between a few hundred distinct colours) suggest that extension to more levels for each primary colour is not cost-effective.

Each primary colour is used for one of the three per-predicate values. The current allocation is

$$\begin{array}{rcll} \text{cyan} & \longrightarrow & \sigma & \text{(sequential execution metric)} \\ \text{magenta} & \longrightarrow & o_i & \text{(incurred overhead metric)} \\ \text{yellow} & \longrightarrow & o_p & \text{(potential overhead metric)} \end{array}$$

The intensity of the colour is directly proportional to the magnitude of the value. That is, for a value of 0, white is shown. As values approach 1, the associated characteristic is ever stronger, and a correspondingly stronger (darker) shade of the primary colour is shown.

[1] In future versions the predicate strips will be labelled with their predicates.



(a) (b)

Figure 4.1: (a) *Basic user interface layout* (b) *Pop-up window for changing display time*

We can classify the set of prominent colours according to what sort of parallelism they denote:

Good parallelism (white). This is an ideal situation. There are few or no fine-grained processes and some exploited parallelism. Potentially bad parallelism (yellow). Yellow alone shows exploited parallelism, plus some fine-grained calls which do not significantly affect the execution.

Bad parallelism (cyan, green, blue, black). The presence of cyan in all these colours means that there is no exploited parallelism. Thus, if there are fine-grained calls (the case of blue or black), there is no exploited parallelism to offset the incurring overhead.

Unpredictable parallelism (red, magenta). These colours mean that there is exploited parallelism, but with significant incurred overhead. Further, it is not known which will have the greater effect. There may be speedup or slowdown.

In the general case a predicate strip will be a mixture of colours. Often a dominant colour or colour pattern is evident. For example, the intensities of yellow and magenta may be low, indicating a nonzero but low level of fine granularity.

A strip with colours indicating good parallelism at one point, and bad parallelism at another may indicate a situation as unpredictable as a strip with red or magenta everywhere. However, it is useful to show how the parallelism changes with time, instead of simply showing the "average colour" for each strip for the entire computation.

Predicates which are not active (have no parallel calls being processed) at some point in the computation have no colour at the corresponding point in the display; that is, they are shown by the background window colour.

4.2 User interface features

The visual interface of ParSee consists of two windows: the graph window and the control window. The former contains the vertical strips, and a time axis. The time axis is labelled at either end with the start and end times for the plot. Figure 4.1(a) shows the user interface, with the control window placed partially over the graph window.

The processor utilisation strip, labelled by `idle`, is the left-most in the plot. Values of the metric are encoded as a grey-scale intensity, with 0 corresponding to white and 1 to black; greater "processor idleness" corresponds to a darker colour. Five levels in total are possible, just as the number of levels of primary colour.

The control window contains information fields and buttons for controlling the viewing process. As well, a plot of the available colours is shown for reference. The plot shows the individual magenta $\times$ yellow planes (perpendicular to the cyan axis) through the 3-dimensional colour space. The information fields include: the name of the current analysis file, global start and end times for the trace, and the basic time slot length.

The control buttons are:

- Show/Redraw — update the contents of the graph window according to the contents of the analysis file, and according to the current control settings.
- Done — finish with this analysis file; that is, exit the program.
- Adjust Display Time — select the portion of the overall trace time to be shown in the graph window.
- Averaging Factor — average values over multiple time slots (discussed in section 4.4).

Because it may be hard to decide what intermediate hues (such as shades of grey, brown, and purple) consist of, ParSee allows the user to selectively mask out any constituent primary colour and concentrate on only one or two metrics. For this purpose, there are also three coloured buttons, each showing one of the primary colours and labelled by the metric represented by that colour. Toggling one of these buttons (such that the button is white) will cause the corresponding primary colour to be masked out of the display when it is next redrawn. Toggling the button again will cause the colour to again be shown.

4.3 Data handling features

The strips in the graph window are drawn as a sequence of coloured rectangles, each rectangle representing the characteristics of the predicate over a single time slot. The display can show a set maximum number of rectangles and a minimum size for them is enforced.

The start and end times for data to be shown by ParSee (i.e. the "display period") are under user control. Initially, they are set to the global data start and end times specified in the analysis file. Parsee automatically scales the size of the rectangles in the colour stripes so that the entire display period is shown in the graph window.

ParSee allows a user to "zoom in" on the activity over a certain subset of the total display time. Toggling the "Adjust Display Time" button causes a pop-up window with a time axis to appear. The global start and end times (from the analysis file) are always shown as the end-points of the time line. Within this window, toggling the left mouse button sets the display start time to a value directly proportional to the vertical mouse position. Toggling the right mouse button analogously sets the display end time. As user feedback, the time value corresponding to the mouse click is shown. Values are checked for consistency by ParSee. Thus, for example, setting display start time later than the currently set end time is not permitted. If a second start or end time is specified, it overrides the former value. The middle mouse button confirms the selected display start and end times, and causes the pop-up window to be dismissed. If either display start or end time was not set by mouse input, it defaults to the global start or end time, respectively.

Figure 4.1(b) shows an example of the pop-up window for adjusting the display time (the analysis file being used is the same as for Figure 4.1(a)). In the figure, new display start and end times of 2694 and 4249, respectively, have been specified. If the user were to click the middle mouse button in the window at this point, the display period would be set to these times. The resultant graph window for this time period is shown in Figure 4.2.

When the user "zooms in" on a particular time period, it may occur that the analysis strips begin slightly after the display start time, and/or end slightly before the display end time (this is evident in Figure 4.2). This occurs when the start (or end) time does not fall precisely at a time slot boundary. ParSee does not display information for partial time slots. Display start or end times may not coincide with time slot boundaries because of (inevitable) round-off and truncation errors incurred when mapping the global display time interval to the coordinate system of the display window.

4.4 Future work

It can occur that there are more data points to be displayed than allowed by the resolution of the graph window. Due to arithmetic rounding when drawing strips, this may result in two (or more) colour rectangles (representing analysis values) overlapping. They will be drawn in the same location, and only the last-drawn one will be visible. Visual information can thus be lost.

Future versions of ParSee will eliminate this problem (by use of a scrolling window for the strip

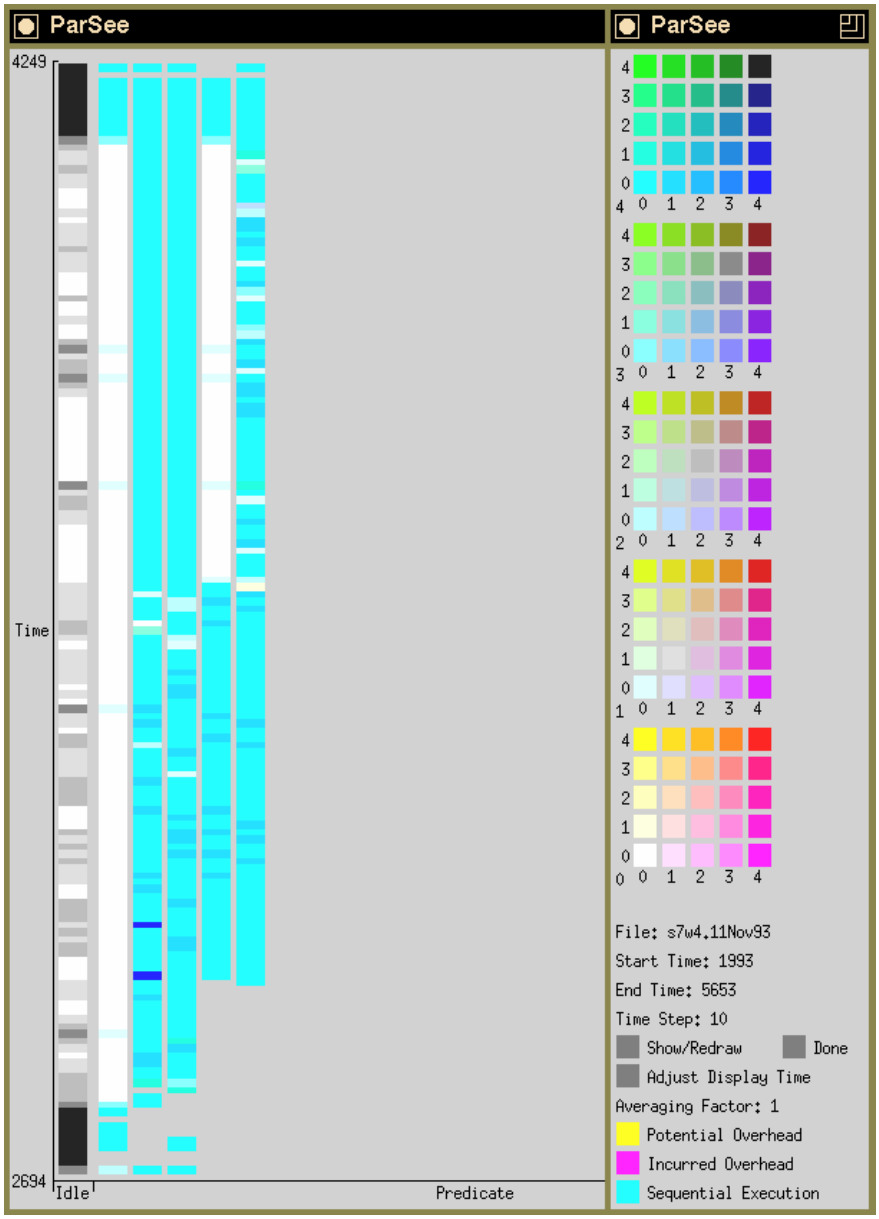


Figure 4.2: *Zooming-in on a particular time period*

plots, for example). To remedy the problem in the current implementation, the "Averaging Factor" field is provided. This is a user-modifiable field. An averaging factor of 2, for example, causes the information from every two adjacent time slots to be merged and (arithmetically) averaged to yield time slots twice as long. The number of rectangles to be plotted is halved. This may then reduce the number of rectangles to be plotted to less than the maximum, solving the problem of lost visual data points.

The current ParSee display interface is only a prototype for determining useful and relevant functionality. Future versions will use more advanced and polished graphics and user interface capabilities.

Another extension is to allow the programmer to instrument programs to denote execution-level events which are carried directly forward to the picture generation phase without transformation. Examples of such events might include the calling of a certain predicate, or entering a particular phase of the program execution. In this way, display features can be linked to execution events.

Chapter 5

Concluding remarks

A high-level tool called ParSee for visualising parallel constraint logic program computations has been described. The tool is aimed at the application programmer rather than at the system developer, and abstracts away information which cannot be directly used when optimising a program. As well as being useful alone, ParSee is useful for previewing a trace file to pinpoint problem areas which can then be examined in detail by other tools (e.g. ParTrace [4]).

Scalability played an important part in the design of ParSee. The display can handle any number of processors and any size of computation tree because neither per-processor information nor the tree are shown. Computations of any length can be displayed since the information is auto-scaled to fit on the time axis, and averaging and zooming capabilities are available. Many parallel predicates can be displayed because we can use a very narrow strip for each predicate. Large trace files do not cause prohibitively long analysis times because ParSee's analysis takes time linear in trace file length. Scaling up to large computations may cause excessively large trace and analysis files, but this applies to any such tool.

An implementation of ParSee on and for ElipSys [7] has been tested on several examples (including a protein topology prediction program [2]) with promising results. A summary of this report will soon be published in [16], and a short paper based on earlier work appeared in [12].

Future work

As described previously, all three phases of ParSee can be considerably enhanced:

- Trace generation will be extended to handle other forms of parallelism (in particular speculative parallelism and AND-parallelism) and to conform to the format of trace files for other tools.
- Trace analysis will be extended to handle other properties of parallel computations.
- Picture generation will need to handle these extra features, and the user interface can be improved.

The general direction for extensions to ParSee will be to provide a complementary tool for execution analysis which works in a coordinated fashion with other tools (as in ViMust [10]).

It would also be interesting to implement versions of ParSee for other high-level parallel languages.

The use of ParSee and details of the trace generation will be the subject of future reports.

Bibliography

[1] M. Carlsson, K. Danhof, R. Overbeek. A Simplified Approach to the Implementation of AND-parallelism in an OR-Parallel Environment. *Proceedings of the Fifth International Conference on Logic Programming*, MIT Press, 1988, pp. 1565–1577.

[2] D. A. Clark, C. J. Rawlings, J. Shirazi, A. Veron, M. Reeve. Protein Topology Prediction through Parallel Constraint Logic Programming. *Proceedings of the First International Conference on Intelligent Systems for Molecular Biology*, eds. L. Hunter, D. Searls, J. Shavlik, AAAI/MIT Press, Menlo Park CA, 1993.

[3] S. D. Conte, H. E. Dunsmore, V. Y. Shen. Software Engineering Metrics and Models. Benjamin/Cummings Publishing Company Inc. 1986.

[4] M. Dorochevsky, J. Xu. ElipSys Parallel Execution Tracer. Internal Report no. 91-71, ECRC, February 1991.

[5] M. Ducassé. A General Trace Query Mechanism Based On Prolog. *Proceedings of the International Symposium on Programming Language Implementation and Logic Programming*, Lecture Notes in Computer Science **631**, Springer-Verlag, August 1992, pp. 400–414.

[6] M. Ducassé, J. Noyé. Logic Programming Environments: Dynamic Program Analysis and Debugging. *Journal of Logic Programming* 1994 (to appear).

[7] A. Véron, K. Schuerman, M. Reeve, L. Li. Why and How in the ElipSys OR-Parallel CLP System. *Proceedings of PARLE'93*, Springer-Verlag, 1993, pp. 291–302.

[8] M. J. Fernández, M. Carro, and M. Hermenegildo. Ideal Resource Allocation (IDRA): A Technique for Computing Accurate Ideal Speedups in Parallel Logic Languages. Technical report FIM26.3/AI/92, Computer Science Faculty, Technical University of Madrid, September, 1992.

[9] M. Gorlick, C. Kesselman. Gauge: A Workbench for the Performance Analysis of Logic Programs. *Proceedings of the 5th International Conference on Logic Programming*, eds. R. A. Kowalski and K. A. Bowen, MIT Press, Seattle, Washington, August 15–19, 1988, pp. 548–560.

[10] L. Gomez, M. Carro, M. Hermenegildo. Implementation of an Event Driven Scheme for Visualizing Parallel Execution of Logic Programs. *Proceedings of the Workshop on Parallelism at IJCSLP'92*, Washington DC, November 1992.

[11] M. Hermenegildo and K. Greene. The &-Prolog System: Exploiting Independent And-Parallelism. *New Generation Computing* **9**(3,4), 1991, pp. 233–257.

[12] A. J. Kusalik, S. D. Prestwich. Programmer-Oriented Visualisation of Parallel Logic Program Execution. *Proceedings of the 5th Workshop on Logic Programming Environments, ILPS'93*, Vancouver, British Columbia, October 29–30 1993, pp. 68–74.

[13] E. Lusk, T. Disz. A Graphical Tool for Observing the Behavior of Parallel Logic Programs. *Proceedings of the 1987 Symposium on Logic Programming*, August 31 – September 4, 1987, San Francisco, pp. 46–53.

[14] A. D. Malony. Event-Based Performance Perturbation: A Case Study. *3rd ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP'91)*, SIGPLAN NOTICES **26**(7), July 1991 pp. 201–212.

[15] S. D. Prestwich. Parallel Speedup Anomalies and Program Development. Technical Report ECRC-93-12 ECRC, 1993.

[16] S. D. Prestwich, A. J. Kusalik. High-Level Visualisation of Parallel Logic Program Execution. *Parallel Architectures and Languages Europe PARLE'94* (to appear).

[17] X.-H. Sun, L. M. Ni. Another View on Parallel Speedup. *Proceedings of Supercomputing '90*, IEEE, New York, November 1990, pp. 324–333.

[18] C. Svensson, J. Sundberg. MuseTrace — A Graphical Tracer for OR-Parallel Prolog. Technical Report T-90/9003, Swedish Institute of Computer Science (SICS), 1990.

[19] E. Tick. Visualizing Parallel Logic Programs with VISTA. *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT 1992, pp. 934–942.