Programmer-Oriented Parallel Performance Visualisation

by

Steven Prestwich and **Anthony Kusalik**

TR-96-01

Department of Computer Science
University of Saskatchewan
57 Campus Drive
Saskatoon, Saskatchewan
S7N 5A9 Canada

February 1996

Programmer-Oriented Parallel Performance Visualisation[1]

by

Steven Prestwich[2] and **Anthony Kusalik**
`steven@ecrc.de` `kusalik@cs.usask.ca`

TR-96-01[3]
Department of Computer Science
University of Saskatchewan
57 Campus Drive
Saskatoon, Saskatchewan
S7N 5A9 Canada

Abstract

High-level parallel languages relieve the programmer of low-level details of parallel programming, and several visualisation tools are available for performance evaluation of these languages. However, the abstraction level of the languages does not seem to be reflected by current visualisation tools. Much of the information the latter convey is the same as that for low-level parallel languages; e.g., suspensions and resumptions, interprocessor communication, and processor utilisation. Even higher level forms of display such as program graphs can be very detailed and difficult to interpret. Our thesis is that much of this information is too detailed for application programmers. This paper describes the visualisation tool *ParSee* which provides a high-level view of a parallel computation. This view is a static picture utilising colour to portray several performance metrics, each metric diagnosing the influence of a parallel program annotation on performance. The view is scalable over the number of processors, making it suitable for highly parallel executions. It is also scalable over time, enabling the user to summarise long traces and to focus on periods of interest. ParSee is implemented in and for a parallel constraint logic programming language, but the principles behind the tool are also applicable to other high-level parallel languages.

Keywords: *parallel logic programming, scalable trace visualisation, performance evaluation.*

February 1996

[1]This work was partially funded by the CEC as part of Esprit Project EP6707, ParForce (Parallel Formal Computing Environment) and by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

[2]author's address: ECRC GmbH., Arabellastraße 17, München 81925, Germany

[3]URL is `ftp://ftp.cs.usask.ca/pub/envlop/ParSee/parsee.tr96-01.ps.gz`
or `http://www.cs.usask.ca/ftp/pub/envlop/ParSee/parsee.tr96-01.ps.gz`

Contents

Chapter 1

Introduction

High-level parallel languages, such as parallel constraint logic programming languages (CLP), aim to relieve the programmer of the burden of parallel programming. Typically, parallelism can be specified simply by annotations in the source code, so that complex and error-prone aspects such as communication and synchronisation are handled automatically by the runtime system.

However, this does not make parallel programming trivial, and finding a good parallelisation is still 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. It is usually impractical to try every combination of alternatives. To find an optimal or near-optimal parallelisation, the programmer needs to understand how each annotation 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 task granularity, whether the scheduler selects coarse-grained or fine-grained tasks, and so on.

Trace visualisation is a promising technique for aiding performance evaluation and tuning. Several visualisation tools are available for parallel logic programming languages [4, 6, 8, 11, 15, 16]. However, we believe that much of the information provided by current tools is not relevant for most application programmers. They show too much detail, or the information is at an inappropriate level of abstraction, or both. These systems are typically developed for systems programmers, and concentrate on detail relevant to the underlying execution mechanism. Important properties such as task granularity can be hard to extract from detailed information about interprocessor communication, the shape of the execution tree, or which processors are executing which parts of the program.

The descriptions of these visualisation tools 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. Automation of this analysis and interpretation is not supported. Moreover, animated representations [4, 15] of the computation may take a very long time to view and be difficult to relate to the program source code.

This paper describes a new visualisation tool for parallel logic programs called *ParSee*. It analyses trace information and presents the user with high-level information in the form of a static colour picture, which abstracts away low-level details. The picture 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 performance evaluation and for guiding parallelisation. ParSee is substantially different from tools which display information about a computation based on a computation tree, processor operations, or communication activity.

ParSee is currently implemented in and for the OR-parallel CLP language ElipSys [5]. For simplicity, ParSee is discussed in this paper only in the context of OR-parallelism. However, the principles are applicable to situations involving AND-parallelism or AND/OR-parallelism. Further, because the basic approach does not rely upon any specific logic programming concept or construct, it is applicable to other high-level parallel languages.

1.1 Background and Motivation

Trace visualisation tools such as ParTrace [4], VisAndOr [6], WamTrace [11], MUST [15] and VISTA [16] are "... invaluable help when implementing, debugging, and tuning parallel systems" [6]. 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 visualisation 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 cause 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 are 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, parallel logic programming languages also create more complex situations, involving several factors interacting in various ways. 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, to provide the user with more abstracted information.

ViMust [6] is a synchronised execution of both the 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. 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.

There are many trace visualisation tools outside the area of logic programming, providing an even greater variety of pictures (see for example [10]). One system of particular note is PerfVisS [9], a performance visualiser for parallel Fortran programs on workstation farms. It uses existing scientific visualisation and data exploration technology to produce several views of trace data. A detailed view shows the usage of parts of the program source or assembler code, using either colour intensities or a 3D surface. Various compression techniques are used to reduce the image while drawing attention to performance hot spots. This has commonality with our approach, as it links performance to the program source, but the details are very different. In particular, we provide a different set of performance metrics.

1.2 ParSee overview

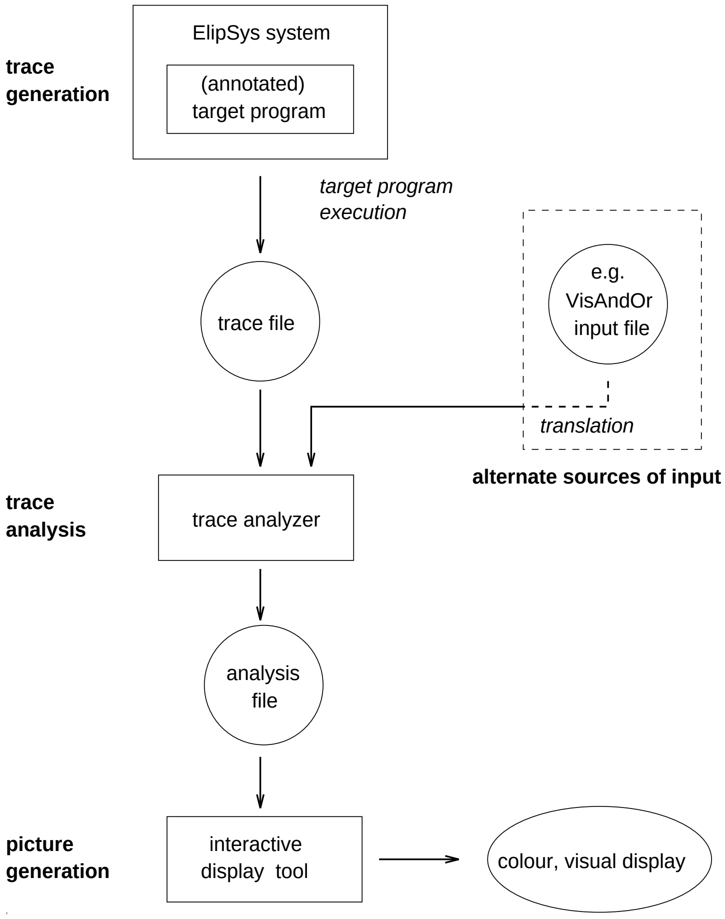


Figure 1.1: *Overview of the ParSee phases*

ParSee generates and analyzes low-level trace information, and portrays the result of the analysis in a static colour picture. It interactively allows the user to view this picture in different ways. As shown in Figure 1.1, ParSee produces a picture in three steps: *trace generation* (described in Chapter 2), *trace analysis* (Chapter 3) and *picture generation* (Chapter 4). Chapter 5 provides a case study describing how this information can then be used to tune the performance of a program. Chapter 6 draws some conclusions and indicates avenues for future work.

1.3 Terminology

To facilitate the following discussion, some terminology is now defined. It is assumed that the reader has some familiarity with the fundamental terminology of logic programming. If not, the paper should still be comprehensible because only the fine details of the performance metrics depend on logic programming concepts.

A logic program execution starts from a goal (the *query*) and overall 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 one branch of 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.
```

We shall refer to this as an *annotation*.

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*. The *granularity* of parallel tasks (children) is an important measure in parallel computations. *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*. It is difficult to define and use a precise threshold to distinguish coarse- and fine-grain tasks because parallelism overheads are not constant.

[1] Absolute speedup is defined by Sun and Ni [14] 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.

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.

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 queries executed 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 succeeding or by backtracking past its start.

Each event contains

- a unique identifier for the child;
- another identifier for the parent call (common to siblings);
- 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).

If there is a global clock available, this can be used for the time stamp. Otherwise the global time can be reconstructed in a post mortem analysis from local processor times.

Because of the ElipSys *commit* operator some children may have a start event but no end event. These can be inserted afterward by an analysis which reconstructs such events. Such an analysis is not described here because in future we intend to insert them at runtime by special low-level primitives.

Chapter 3

Trace analysis

Given a trace file, the next step is to analyse it to abstract out useful properties. The trace file contains information on the children of all parallel calls, and 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. This leaves only performance information linked to annotations.

For purposes of analysis, the trace is divided into small time slots of identical duration, and performance metrics are calculated over each time slot. The value of a metric m for a given time slot (t_1, t_2) is the mean value over that period of a function $f(t)$ of time. We define each metric below in terms of its function $f(t)$.

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

σ: the sequential execution metric;

π: the unexploited parallelism metric;

o_i: the incurred overhead metric.

o_p: the potential overhead metric;

There is in addition one predicate-independent entry per time slot which is

υ: the processor utilisation metric.

More metrics are certainly possible, and various metrics have been devised in the course of the development of ParSee. Only four are included in the current version of ParSee for simplicity. This issue is discussed further in Section 6.1. In Section 3.6 we discuss some important "meta-metrics": characteristics of the trace metrics themselves which can be used to evaluate new ideas for metrics.

3.1 The sequential execution metric

The *sequential execution metric* σ indicates a lack of exploited parallelism or an annotation. 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 annotated 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.

A predicate is considered to be executing 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 $seq(t)$ 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 unexploited parallelism metric

We would like to extrapolate the useful work and overhead metrics to the case of more processors. In general, little can be extrapolated: for example a speculative computation may even give a different result when reexecuted on more processors. However, we can detect cases where the lack of free processors caused tasks to start later than necessary, and this gives a rough guide to the potential for further parallelism.

Say a task has been submitted to the task scheduler at time t, but not scheduled until time t' because there were no free processors. Say also that during this time there was at least one sibling being executed. Then we could say that during the interval (t, t') there was one unit of *unexploited parallelism*; that is, parallel work which would be performed if there were sufficient processors. At a given time there may be several such tasks, which can be summed to give several units of unscheduled work, denoted by $unex(t)$. Then the unexploited parallelism metric $\pi(t_1, t_2)$ is defined as the mean value of $unex(t)$ over the time slot (t_1, t_2).

3.3 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 [12]. It is therefore important to know when these overheads occur and what their causes are.

The *incurred overhead metric* o_i is based on the proportion of processing time spent on parallelism overhead during a time slot, which in turn is based on an estimate of the typical overhead t_o. If at least half the time was spent on overhead then the annotation did not pay off during that time slot.

Parallelism overhead mainly occurs when a task moves or is moved from one processor to another. Each such event can 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. Let $p_b(t)$ denote the total number of processors busy on the predicate at time t. $p_b(t)$ can be inferred from a count of the number of start and end events for the predicate prior to t. [1]

A processor is considered to be busy on parallelism overhead for a predicate `q` if it is involved in setting up a parallel task for `q`. Let $p_o(t)$ denote the number of processors busy with the parallelism overhead for the predicate at time t. $p_o(t)$ can be estimated based on the count of start events where the child task is on a different processor from the parent call. It is assumed that, for a time t_o prior to the start of a parallel task, two processors are busy on overhead: one to spawn the new task and one to perform it. They must communicate with each other before the task can be started.

$o_i(t_1, t_2)$ is then defined as the mean value of $\frac{p_o(t)}{p_b(t)}$ over the time slot (t_1, t_2). If no time was spent on overhead then there is good use of parallelism, and $o_i = 0$. When all processors are busy on overhead $o_i = 1$.

ParSee is tuned to a specific parallel system by choosing an appropriate value for t_o. This has been done manually so far, but in future ParSee will be able to calibrate itself on demand when porting it to a new system.

3.4 The potential overhead metric

Whereas o_i estimates the incurred overhead due to migrant children and 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 (its parallel annotation removed), possibly freeing processors to be allocated to this predicate.

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

Whereas o_i uses a count of migrant children to estimate incurred overhead, o_p uses local children which are *potentially* migrant 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) as the mean value of $\frac{p'_o(t)}{p_b(t)}$ over the time slot (t_1, t_2).

3.5 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.

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

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

3.6 Meta-metrics

Many performance 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] or criteria to evaluate the metrics. Such a group of measures 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. Formalising as many of them as possible and verifying that ParSee metrics meet all of them will be the subject of future study.

3.6.1 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 papers on the application of ParSee to program development.

3.6.2 Monotonicity

A general principle in the ParSee's user interface design (to be discussed in Section 4.1) is that the darker a colour, the worse the corresponding property. This simplifies picture interpretation. For each metric defined above, the greater its value, the stronger (worse) the corresponding property for the predicate in question.

3.6.3 Orthogonality

Another general principle of ParSee is that each metric should be meaningful when considered alone, and need not be considered in relation to any other metric. A user can then 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 have this quality.

3.6.4 Time locality

Each metric $m(t_1, t_2)$ must say something about the execution over 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 period. 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.

[2] Since there are only 3 primary colours, in the current ParSee scheme only 3 metrics can be shown at once.

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.

3.6.5 Time scalability

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.

3.6.6 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.

3.6.7 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. Each metric is computed by a small, fixed number of passes through the trace file, and all necessary information is stored in arrays and hash tables which have constant retrieval time.

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.

The picture consists of a set of vertical strips, each strip (except for the two left-most) representing the effect of one parallel predicate on the computation. The predicate name is given below the strip. The vertical axis represents time within the computation. The general paradigm for interpreting the content of the display is that "the lighter the colour at any point in a strip, the better the use of parallelism". The hue of darker colours indicates what is going wrong, or what might potentially be a problem.

The two left-most strips augment the display. The strip labelled "all" is a summary statistic, portraying information accumulated over all (parallel) predicates. The left-most strip indicates utilisation of available processors, with poorer utilisation indicated by darker shades of grey.

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 monotonicity and orthogonality criteria for metrics defined in Section 3.6.

Currently, five levels of each primary colour can be shown, giving a total of $5^3 = 125$ possible colours (primary and composite). The limit of five colour levels was initially imposed by the implementation system (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 provision of more levels for each primary colour would relay little additional information.

The intensity of each colour is directly proportional to the magnitude of the allocated metric. If all metrics are 0 over a time slot, white is shown. As values approach 1 the associated characteristic becomes stronger, and a correspondingly darker shade of the primary colour is shown. In the general case, a predicate strip will be a mixture of colours. For a particular allocation of metrics to primary colours, a dominant colour or colour pattern is often evident. A strip with intermixed bands of dark

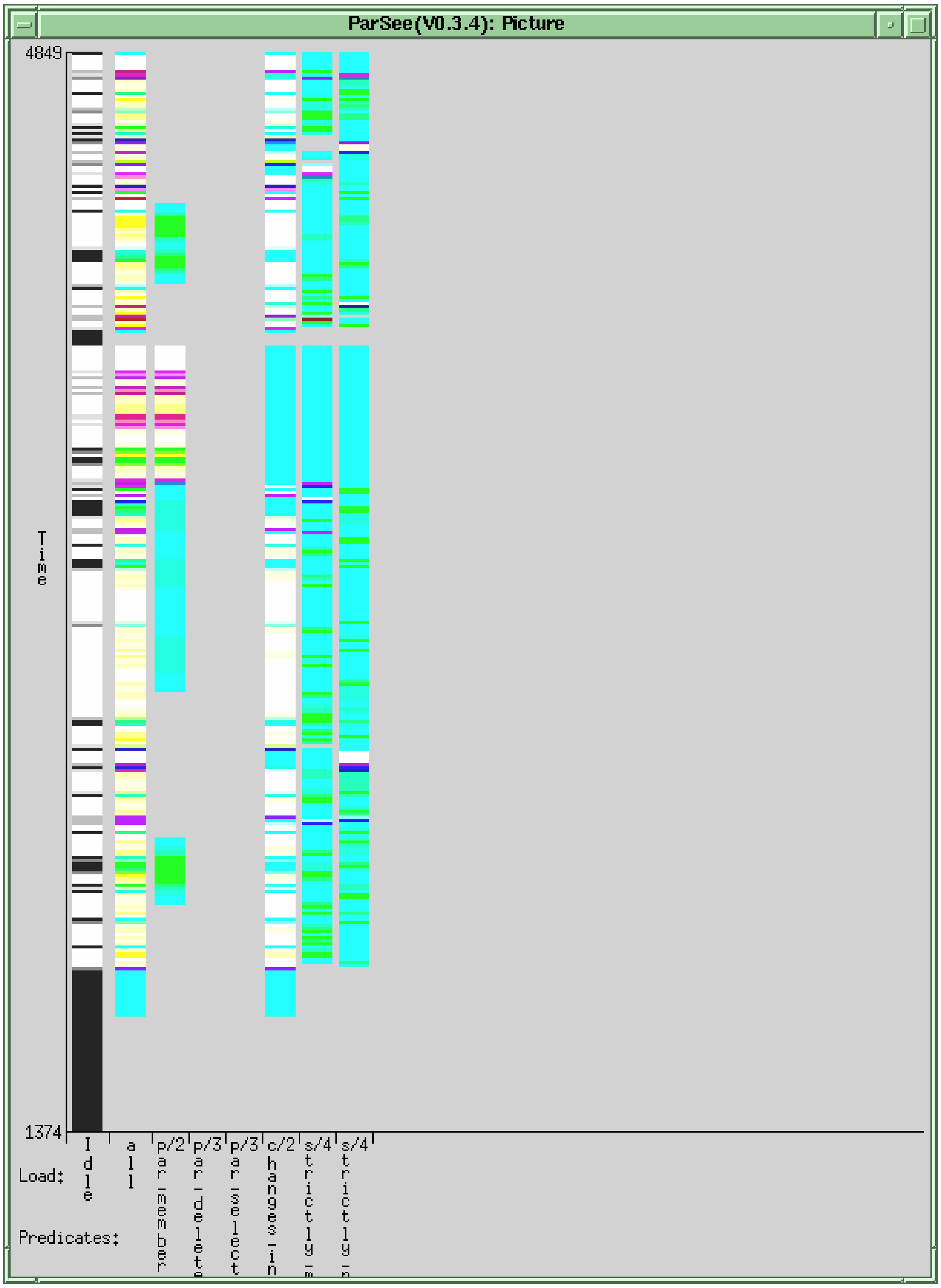


Figure 4.1: *Basic User Display*

and light colours (indicating good parallelism at one point, and bad parallelism at another) may indicate a situation as unpredictable as a strip with dark colour 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 Displaying and manipulating performance metrics

The visual interface of ParSee consists of three windows: the *graph window*, the *control window*, and the *colour-map window*. The graph window (as exemplified in Figure 4.1 earlier) 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. The processor utilisation strip, labelled `Idle`, is the left-most in the plot. Values of the v processor utilisation metric are encoded as a grey-scale intensity, with 0 corresponding to white and 1 to black; greater "processor idleness" corresponding to a darker grey. Five levels in total are possible, as with the number of levels of primary colour.

The colour map window (Figure 4.2(a)) contains a plot of possible, available colours. It is for user assistance in determining the relative degrees of the three primary colours in a particular hue in a strip. The plot shows the individual magenta $\times$ yellow planes (perpendicular to the cyan axis) through the 3-dimensional colour space.

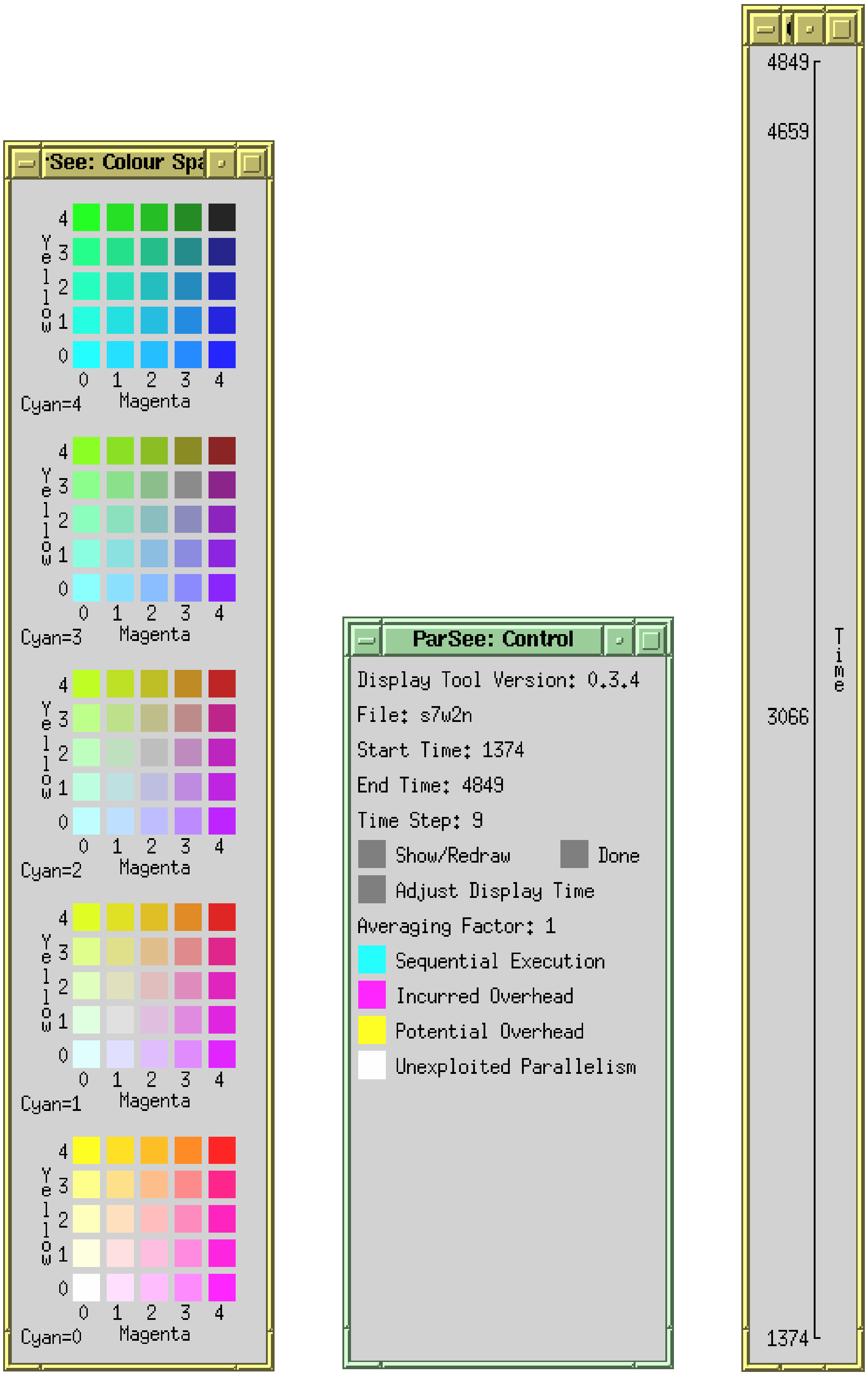
The control window (in Figure 4.2(b)) contains information fields and buttons for controlling the viewing process. The information fields include: the name of the current analysis file, global start and end times for the trace, and the basic time slot length.

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).

The lower half of the control window is devoted to controlling the display of the various metrics contained in the analysis file. The name of each metric is shown, preceded by a button. If the button is coloured, that metric will be displayed. The colour of the button is the (primary) colour which will be used. A white button indicates that a particular metric will not be displayed. Since there are three primary colours, at most three buttons can be coloured (i.e. three metrics chosen) at any time. By toggled on the buttons, a user can enable or disable display of metrics, and control which colours are assigned to which metrics. The metrics shown in Figure 4.1 are those indicated in Figure 4.2b. Figure 4.3 shows the result of selecting an alternate set of metrics from the same trace file.

It may not be obvious to a user — even given the colour map window — which colours combine to form intermediate hues such as shades of grey, brown, and purple. The control window therefore allows the user to selectively mask out any constituent primary colour and concentrate on only one or two metrics. This is possible since the user controls the number of (primary) colours shown in the graph window. Figure 4.4, for example, shows the result of concentrating on only two of the colours (metrics) shown in Figure 4.3.



(a) (b) (c)

Figure 4.2: (a) *Assistance-oriented Colour Map*, (b) *Window for Controlling ParSee Operations*, (c) *Pop-up Window for Changing Display Time*

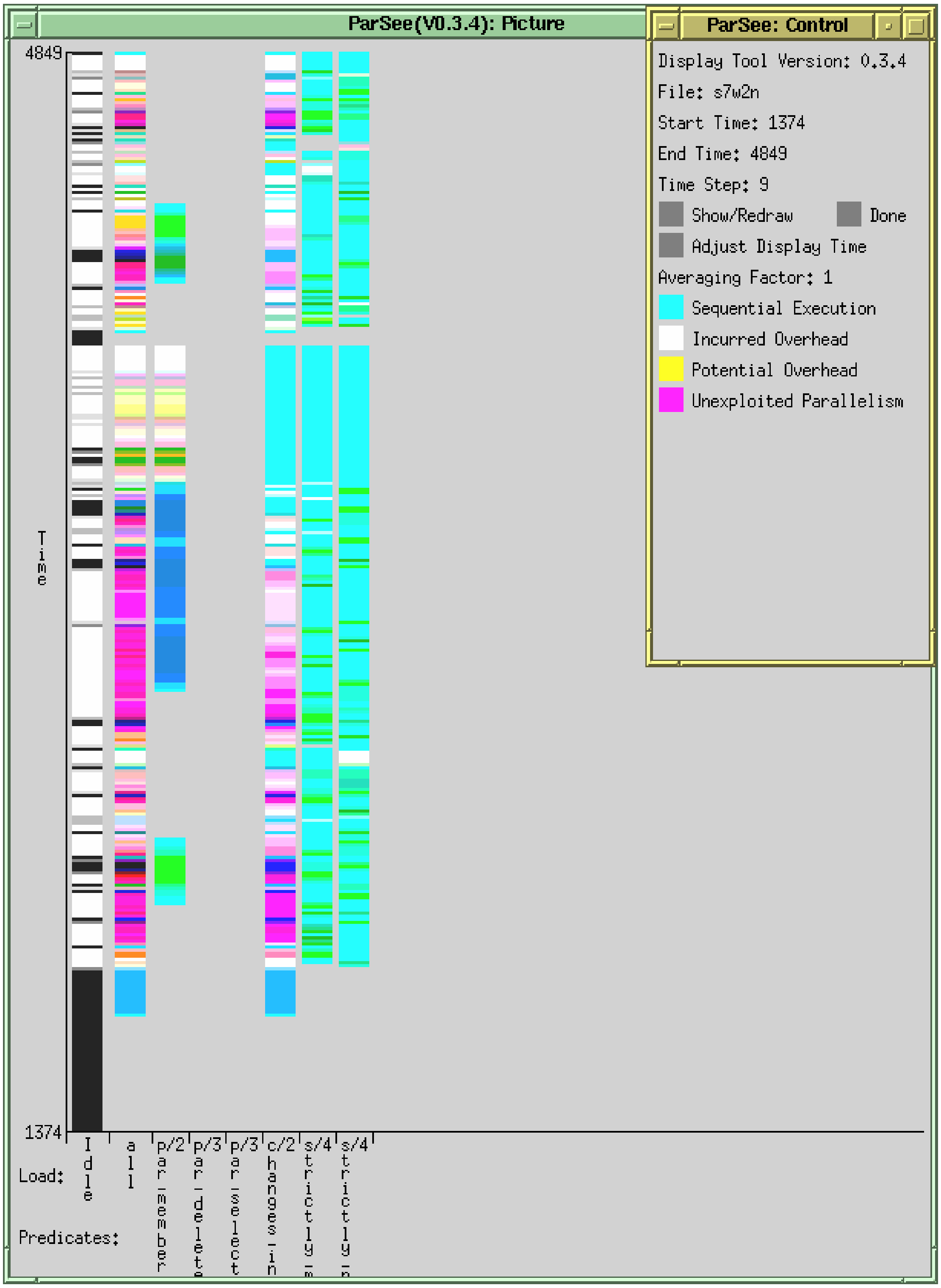


Figure 4.3: *Display of an Alternate Combination of Metrics*

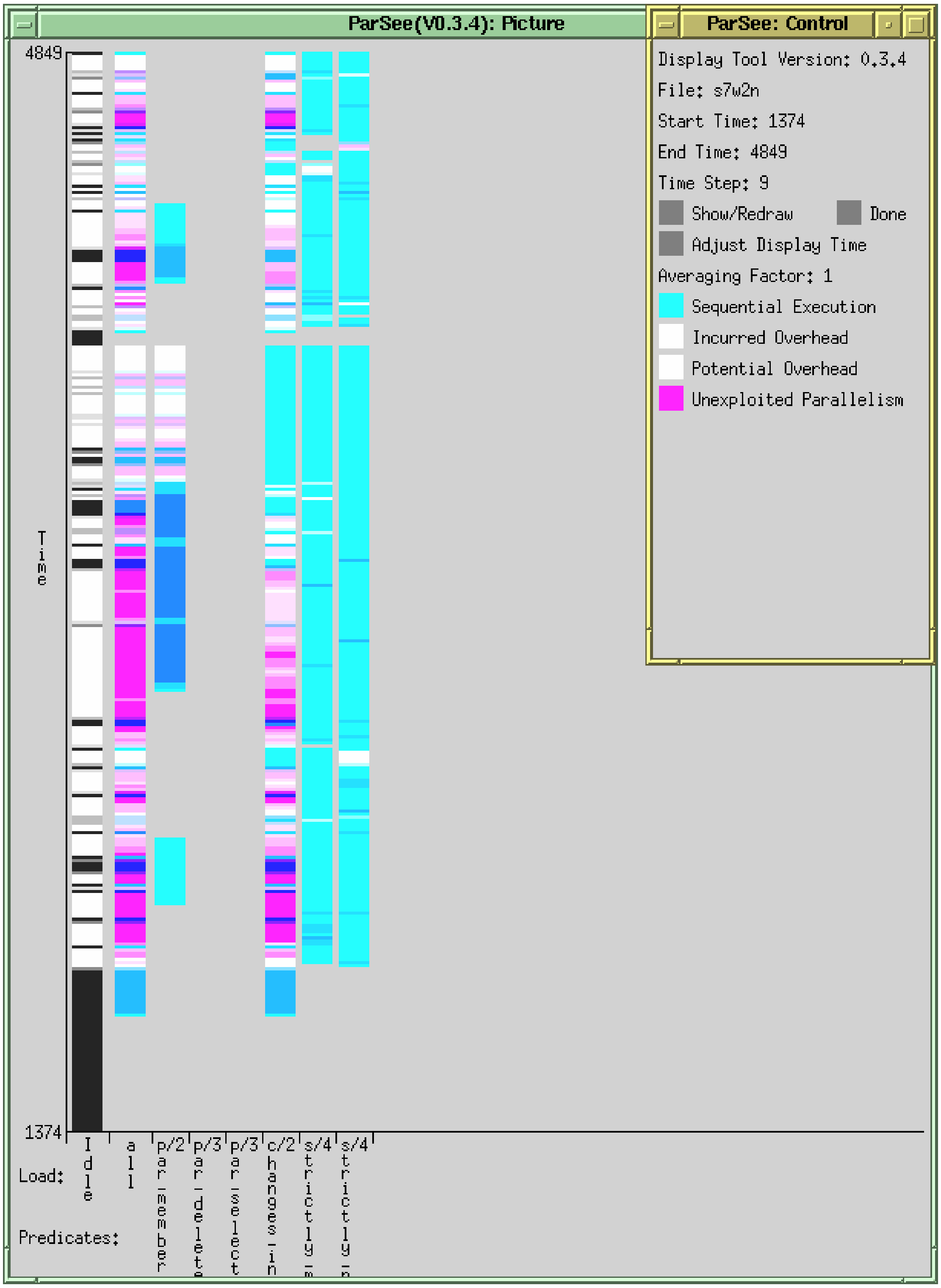


Figure 4.4: *Display of Only Two Metrics*

ParSee is an extendible tool, and the control window has room to display and control additional metrics as they are added in future. Subsequent depictions of the display tool in this paper will include the control window, overlapping the graph window (as in Figures 4.3 and 4.4).

4.3 Exploiting time scalability

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 (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. Within this window, the user can set the display start and end times using mouse clicks along the time axis.

Figure 4.2(c) shows an example of the pop-up window for adjusting the display time (the analysis file being used is the same as for Figures 4.1, 4.3, and 4.4). In Figure 4.2(c), new display start and end times of 3066 and 4659, respectively, have been specified. Once confirmed, the display period for the graph window is set to these times. The resulting display is shown in Figure 4.5.

4.4 Other controls

The "Averaging Factor" field is also user-modifiable. A value greater than 1 directs the display tool to merge information from multiple time slots into larger slots, and an averaging effect is achieved. The field value is the number of time slots to merge into each new one. However, the averaging is only perceived in the display; the information in the analysis file is unaffected.

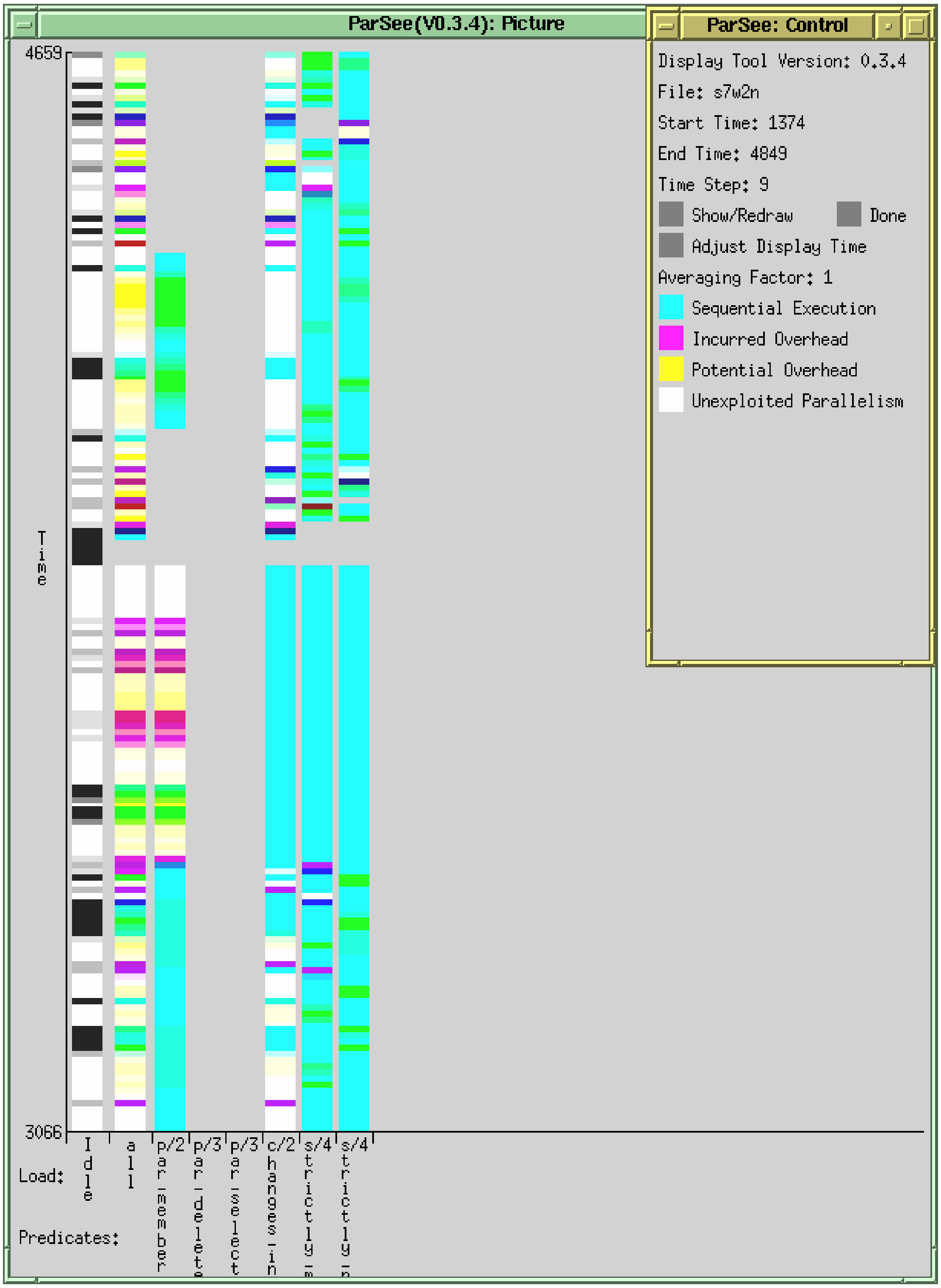


Figure 4.5: *Zooming-in on a Particular Time Period*

Chapter 5

A case study

The pictures shown above were all depictions of a single trace file generated by a protein topology prediction program [1], which is one of a set of parallel constraint logic program applications developed for an Esprit project. [1] The program is written in the ElipSys OR-parallel constraint logic programming language [5] and executed on two Sequent Symmetry processors.

First consider Figure 4.1 together with its control windows in Figure 4.2(b). The colour allocation is

cyan	⟶	sequential execution
yellow	⟶	potential overhead
magenta	⟶	incurred overhead

The overall processor utilisation (the monochrome strip on the left) shows that this program behaves well. The processors are busy most of the time. We can diagnose the (in this case small) faults as follows.

5.1 Sequential execution

The all-predicate strip shows a few small periods of cyan denoting lack of exploited parallelism. At any given time, we can say which predicates contribute to the parallelism; for example in the early and late parts of the trace we can see that all predicates except `changes_in_winding` were executing sequentially, while in the middle only `par_member` exhibits parallelism. In fact the predicates `strictly_monotonic` and `strictly_nonmonotonic` exhibit no significant parallelism anywhere, except for very brief periods. (In fact, experimentally removing the parallel annotations did make little difference.) However, we do not know whether the sequential periods indicate (i) a lack of inherent parallelism, or (ii) inherent parallelism which was not exploited by the task scheduler. It would be useful to know which is the case: if (i), the program will probably perform poorly with more processors; if (ii), there is parallelism which can be exploited on a more parallel machine. We return to these points below.

5.2 Incurred overhead

The all-predicates strip also shows significant though brief bursts of overhead, and we can attribute these to particular predicates. For example, in the late middle period a burst of overhead was caused by `par_member`. This indicates that it might be worthwhile refining this predicate to avoid incurring overhead, for example by adding cheap runtime granularity tests. In fact, this was tried

[1] Esprit Project EP6708, APPLAUSE (Application and Assessment of Parallel Programming Using Logic).

and did indeed eliminate the magenta at that point, though it did not greatly improve the overall speedup.

5.3 Potential overhead

There appears to be significant potential overhead, which may manifest itself if more processors are available. This can be attributed to the predicates `par_member` (late middle period), `strictly_monotonic` and `strictly_nonmonotonic` (early and late periods). We do not know whether, given more processors, there will be sufficient inherent parallelism to counteract this potential overhead.

To summarise the above: Figure 4.1 shows good processor utilisation, with two predicates providing most of the parallelism (though during different periods) and a danger of increased overhead given more processors. However, some questions are left unanswered. To answer these we need more information, and we turn to Figure 4.3 which reallocates one colour:

$$\text{magenta} \longrightarrow \text{unexploited parallelism}$$

5.4 Unexploited parallelism

We can use this metric to answer the questions above. First, note that there is no unexploited parallelism for the predicates `strictly_monotonic` and `strictly_nonmonotonic`, so it appears doubtful whether these predicates should have been annotated "parallel". Secondly, there is a great deal of unexploited parallelism for `par_member` in the early middle period, which should more than counteract the potential overhead if (at least some) more processors are available — though this is hard to quantify precisely.

5.5 Discussion

This example shows that the performance metrics are directly useful to programmers, conveying in a simple manner information which can be used to evaluate performance and suggest improvements. Selecting the appropriate metrics needs a little thought, but is not hard with practice. If 1, 2 or 3 metrics are found to be dominant (shown by stronger colours) then it is helpful to mask out the others before thinking about the picture. Sometimes a portion of a trace may consist of many different hues, but these can be resolved either by masking out selected metrics, by consulting the colour map, or often by zooming in so that different characteristics are not merged.

It is useful to know whether a predicate exhibits different characteristics at different periods. If this is the case, it may indicate distinct usages of the same predicate which should be treated differently. For example, one usage may benefit from runtime granularity tests, or should not be parallel at all.

A note on the application program. This trace was generated by input which caused fine task granularity. Given different data, the predicates `strictly_monotonic` and `strictly_nonmonotonic` exhibited significant parallelism after all. Of course ParSee cannot extrapolate to different input data, and in general we need to view several trace files. The extrapolations ParSee did make were correct: a good speedup curve was obtained when more processors were used, and on the traces we observed the unexploited parallelism vanishing as we added processors.

Chapter 6

Concluding remarks

A tool for high-level visualisation of parallel constraint logic program computations, called ParSee, 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. ParSee is therefore substantially different from tools which display detailed trace information based on computation trees, processor loading, or interprocessor communication activity. As well as being useful alone for non-expert programmers, ParSee is valuable for previewing a trace file to pinpoint problem areas which can then be examined in detail by experts using other tools. We are currently working on integrating ParSee with the VisAndOr [6] tool.

Scalability has played an important part in the design of ParSee.

- Scalability in the sense of [2]: the display can handle any number of processors and any size of computation tree because neither per-processor information nor the tree are shown. This makes it applicable to highly parallel executions.
- Computations of any length can be displayed since the information is auto-scaled to fit on the time axis, and capabilities for zooming in and out are available. It is very important for a user to be able to summarise long traces and to examine in more detail interesting features.
- 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.
- One coloured strip is used to diagnose the effect of each parallel annotation on performance. Many parallel predicates can be displayed because the strips can be narrow.

An implementation of ParSee on and for ElipSys [5] has been tested on several applications, and programmers report that it conveys useful information.

6.1 Future work

All three phases of ParSee can be considerably enhanced.

- 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 [6]. 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 [7]. 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.

- The trace analysis stage can in principle be extended in several ways to analyse other execution 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 the π performance metric).

  Another possible 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.

  We also intend to prove formal properties of the performance metrics such as robustness (i.e. the "meta-metrics") in future studies.

- 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. One prototype currently uses 3D rendering techniques to portray the performance metrics as a surface [13].

The use of ParSee as a key component in a methodology for performance debugging, will be the subject of future papers. Finally, it would also be interesting to implement tools for other high-level parallel languages — not necessarily logic programming ones — following the principles in ParSee.

Bibliography

[1] 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.

[2] A. L. Couch. Categories and Context in Scalable Execution Visualization. *Journal of Parallel and Distributed Computing* **18**(2), pp. 195–204, 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] 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.

[6] 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.

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

[8] V. Herrarte and E. Lusk. Studying Parallel Program Behavior with Upshot. Technical Report, Argonne National Labs, Argonne, Il., USA, 1992.

[9] G. J. Klinker, I.-Y. Chen. PerfVisS: A Performance Visualizer for High Performance Fortran Programs on Workstation Farms. Technical Report CRL 94/4, Cambridge Research Lab, Digital Equipment Corporation, June 1994.

[10] E. Kraemer, J. T. Stasko. The Visualization of Parallel Systems: An Overview. *Journal of Parallel and Distributed Computing* **18**(2), pp. 105–117, 1993.

[11] 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.

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

[13] S. D. Prestwich. Refined Processor Utilisation Graphs for Parallel Logic Programs Technical Report ECRC-94-35, ECRC, 1994.

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

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

[16] E. Tick, D.-Y. Park. Kaleidoscope Visualization of Fine-Grain Parallel Programs. CIS-TR-91-18, Department of Computer and Information Science, University of Oregon, Eugene, Oregon, October 1991.