Programmer-Oriented Visualisation of Parallel Logic Program Execution

Anthony J. Kusalik	Steven D. Prestwich
Dept. of Computational Science	ECRC GmbH
University of Saskatchewan	Arabellastr.17
Saskatoon, Saskatchewan	München 81925
S7N 0W0 Canada	Germany
kusalik@skorpio.usask.ca	steven@ecrc.de

1 Introduction

This paper gives an overview of a new visualisation tool called ParSee. It interprets trace information from a parallel logic program execution and presents the user with high level information in the form of a static colour picture. The picture can be used directly by programmers to detect poor granularity and scheduling, and hence to guide parallelisation and "performance debugging".

Although we shall describe the tool in terms of OR-parallelism, where *predicates* have parallel *calls*, the discussion also applies to AND-parallelism where conjunctions of goals may be called in parallel, and indeed to any parallel system where tasks of unpredictable granularity are allocated to a set of processors. The basic principle of the system is to display the effects on an execution of using parallelism at various points in the program. This distinguishes it from tools which present a view based on the computation tree or on processors and communication.

ParSee produces a picture in three phases: *tracing generation* (Section 3) generates trace data from an execution; *trace analysis* (Section 4) transforms the trace data into high level information; and *picture generation* (Section 5) displays the information as a colour picture.

2 Motivation

Execution visualisation tools such as ParTrace [1], VisAndOr [2], WamTrace [4], MUST [5] and VISTA [6] provide low level information about the progress of a computation and its use of resources. They usually provide some representation of the search tree explored during the computation, the allocation of work to processors and communication between processors. Various graphical shapes may be used to provide extra information. The more recent of these tools, such as VisAndOr and VISTA, also make use of colour to encode information.

While useful for debugging parallel scheduling algorithms, load balancing and so on, we believe that this information is not relevant for most application programmers. Feedback from applications programmers on ParTrace indicates that it provides too much low level detail which obscures the (to the programmer) useful information. What programmers say would be useful is an indication of whether or not coarse grained parallelism has been achieved.

Granularity information can be hard to extract from details of communication between processors, which processor is busy with what part of the tree, the shape of the computation tree and so on. We would like to automate a large part of the picture interpretation to provide the user with a higher level of information.

ParSee performs analysis on the trace information and presents the user with a static colour picture. Low level details which do not seem directly useful to the programmer are filtered out during the analysis. For example, it indicates whether there are idle processors, but not which processors are idle. Neither is communication between processors shown, nor where in the computation tree the processors are busy, nor the shape of the tree. Information on each predicate call is collected and used in the analysis of the effects of the relevant predicate, and the computation tree is abstracted away completely. The information, though derived from the execution, is closer to the program source than to the computation tree and the parallel system.

3 Trace generation

The program is executed and data on parallel events is collected in the **trace file**. The information needed from the tracing phase is a set of events. An example of an event is the creation of a parallel call. Each event has its own time stamp because the granularity analysis is based on execution times.

ParSee currently generates its own trace file via program instrumentation, for pure OR-parallel search only. To handle AND-parallel, pruned and coroutined programs, and to achieve compatibility with other existing tools, it is being extended to accept VisAndOr files (MUST trace files are specific to the Muse execution model, but they can be converted to VisAndOr format). With this general purpose format, executions of OR-parallel, restricted AND-parallel, and mixed AND/OR-parallel computation can be supported.

4 Trace analysis

The trace file contains information on the calls of selected parallel predicates. The analysis phase analyses each call in the trace file and collects this into useful information for each predicate. The results are summarised in the **analysis file**.

The aim of the analysis is to detect fine granularity and to estimate how it affects the computation. Granularity is measured using execution times, and not by counting events. However, there are two problems. Firstly, it is not always possible to decide whether a call is coarse or fine grained, because there is some uncertainty about what constitutes fine granularity. The parallelism overhead is hard to estimate exactly, in fact it is not a constant. Programs with grain sizes near the threshold between fineness and coarseness may behave very differently when ported to machines with slightly different granularity or even if the loading on the system changes. Secondly, if both coarse and fine grains are present it is hard to predict what the net effect will be. It may depend strongly on parameters such as the number of processors or the exact scheduling algorithm, giving effects such as nonmonotonic speedup curves.

ParSee handles these problems by presenting both an optimistic and a pessimistic view of grain sizes, which we call coarseness and fineness respectively. If the two views agree then we can be confident that the analysis reflects reality. If they disagree then of course we cannot be sure because the grain sizes are near the boundary between coarseness and fineness. However, the ambiguity itself is useful as an indication of potential sensitivity to changes in the system.

There are three measures for each predicate, which vary in time:

- The **coarseness measure** is an indication of how certain it is that there are coarse grained calls which may possibly counteract the bad effects of any fine grained calls. This is an optimistic view because we base the measure on the coarsest call, and we also use a small estimate of the parallelism overhead.

- The **fineness measure** measures the rate of creation of fine grained calls. This is a pessimistic view because not all fine grained calls will have a significant effect on execution time. The presence of a single coarse grained call may completely outweigh their effects. We use a large estimate of the parallelism overhead.

- The programmer does not need to know what the scheduler does in detail, but it might be useful to know when poor scheduling decisions are being made if this suggests how the program might be modified. If small grained calls are selected when there are large grained calls waiting then we interpret this as a poor scheduling decision, because it will make the parallelism overhead higher by causing more communication between processors. This is summarised by the **poor scheduling** measure.

There is also one predicate-independent entry which varies in time:

- The **processor utilisation** measure indicates whether there are too few or too many processors as time varies. Too many processors indicates a lack of parallelism, and too few means we could profitably use more processors.

5 Picture generation

The analysis file is displayed via an interactive tool. A sample picture is shown in Figure 1 (the predicate names are not shown). It consists of a set of vertical strips, each strip representing the effect of a parallel predicate on the execution as the computation proceeds. The vertical axis represents "real time" and the user can see features relating to each predicate at any given time. The lighter the shade the better the use of parallelism, and the hue of darker colours indicates what is going wrong. Predicates which are not being used have no colour at that point, and are shown by the background colour. There is an additional histogram strip on the left, based on the processor utilisation measure, to indicate too few (right) and too many (left) processors.

The strips use a subtractive colour scheme to display the predicate information. Cyan, magenta and yellow are primary colours, and all other colours can be obtained by combinations of these. White is the default and black is the combination of all three primary colours. Each primary colour corresponds to a measure defined in Section 4. The intensity of cyan increases as coarseness decreases, that of magenta as fineness increases and that of yellow as poor scheduling increases. The idea is that more colour indicates worse parallelism. An advantage of this scheme is that where there are no problems there is no detail. Only where problems are detected does detail appear and so the eye is drawn immediately to problem areas.

We can classify the hues according to what sort of parallelism they denote:

Good (*white*). This is the ideal colour, denoting coarse granularity.

Bad (*blue, green, black*). These colours mean that there are fine grained calls and no coarse grained calls active to offset them.

Unpredictable (*red, magenta, yellow, cyan*). Red, magenta and yellow mean that there are fine and coarse grained calls coexisting, and it is not known which will have the greater effect. Cyan means that the granularity is on the borderline between coarse or fine.

Because it may be hard to decide what intermediate hues (such as grey, brown and purple) consist of, ParSee allows the user to mask out any primary colour and also provides a 3-dimensional spectrum. There are several other interactive features for changing the view, which are described in [3].

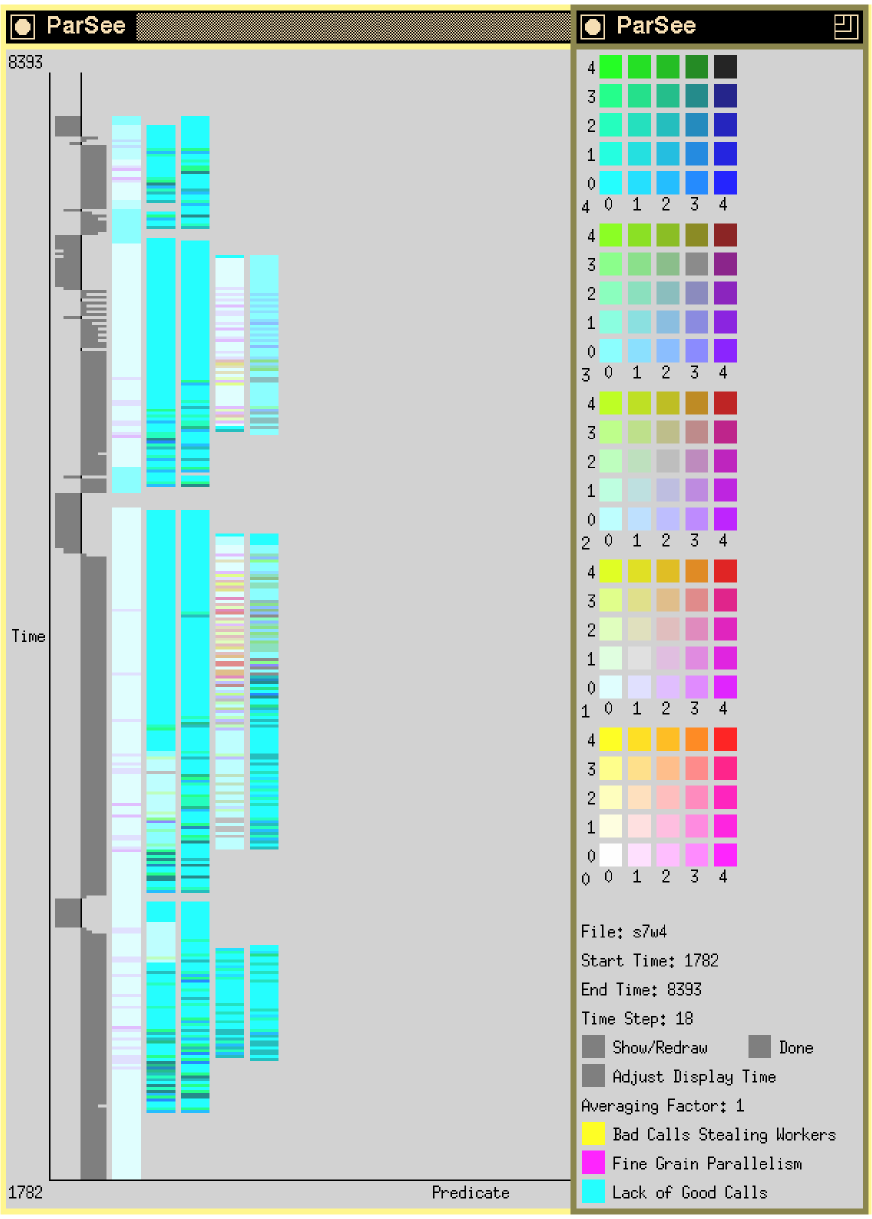

Figure 1: *A sample ParSee picture.*

6 Picture interpretation

Although the picture represents high level information, the user still needs to interpret the picture. Due to space limitations we do not go into the details of how to use ParSee in parallelisation and performance debugging. This is described in [3]. Some examples of the use of ParSee pictures are:

- The user can see immediately where bad use of parallelism occurs.
- Bad predicates can be serialised.
- Unpredictable predicates, or those which are sometimes good and sometimes bad, can often be replaced by specialised versions to serialise fine grained calls.
- Unpredictable predicates indicate a risk of unexpected behaviour and should be handled with care, especially if changing system parameters such as the number of processors or porting to other systems.
- The presence of yellow (or red, green or black) indicates the possibility of improvement by changing the scheduling priorities of predicates, for example by changing clause order if the scheduler always allocates parallel tasks in textual order. If the language allows some user control over scheduling (we do not know of such a system) then the occurrence of yellow can guide this control.

7 Discussion

A prototype implementation of ParSee has been tested on several fairly small examples and gives promising results. We have not yet tested it on large examples, but scalability played an important part in the design of the display. It can handle any number of processors and any size of computation tree because these are not displayed. Computations of any length can be displayed by scrolling along the time axis, or by averaging colours over larger time intervals and providing zooming. Many parallel predicates can be displayed because we can use a very narrow strip for each predicate. Scaling up to large computations may cause excessively large trace and analysis files, but this applies to any such tool. As well as being used alone, ParSee could be useful for previewing the trace file to pinpoint problem areas which can then be examined in detail by other tools.

We intend to extend ParSee to make it more self-sufficient. It currently displays three types of analysis because there are only three primary colours, but it could generate extra information and allow the user to assign the primary colours to the three most interesting properties. For example, it could analyse where unnecessary speculative work is being done, or detect the non-selection of coarse grained calls by the scheduler as well as the selection of fine grained calls. Finally, in the current scheme magenta covers a variety of cases, and knowing exactly what sort of fine granularity is occurring could guide performance debugging better. We can distinguish the following cases:

- OR-parallel calls matching no clause heads
- OR-parallel calls matching one clause head
- OR-parallel calls matching more than one clause whose continuations fail quickly
- OR-parallel calls matching more clauses all but one of whose continuations fail quickly
- AND-parallel conjunctions where all the goals fail quickly
- AND-parallel conjunctions where all but one of the goals fail quickly

All these cases can be classed as fine grained calls because they would be executed faster without parallelism.

References

[1] M. Dorochevsky, J. Xu. ElipSys Parallel Execution Tracer. Internal Report 91-71, ECRC 1991.

[2] 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.

[3] A. J. Kusalik, S. D. Prestwich. Programmer-Oriented Visualisation of Parallel Logic Program Execution. ElipSys Note 93-11, ECRC 1993.

[4] 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.

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

[6] 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.