Visualizing Parallel Functional Program Runs: Case Studies with the Eden Trace Viewer

Jost Berthold and Rita Loogen

Philipps-Universität Marburg, Fachbereich Mathematik und Informatik
Hans Meerwein Straße, D-35032 Marburg, Germany
E-mail: {*berthold, loogen*}*@informatik.uni-marburg.de*

This paper describes case studies with the Eden Trace Viewer (*EdenTV*), a post-mortem trace analysis and visualisation tool for the parallel functional language Eden. It shows program executions in terms of Eden's abstract units of computation instead of providing a machine-oriented low level view like common tools for parallelism analysis do. We show how typical inefficiencies in parallel functional programs due to delayed evaluation, or unbalanced workload can be detected by analysing the trace visualisations.

1 Introduction

Parallel functional languages like e.g. the parallel Haskell extension Eden[1] offer a highly abstract view of parallelism. While less error-prone, this sometimes hampers program optimisations, because it is very difficult to know or guess what really happens *inside* the parallel machine during program execution. A considerable number of profiling toolkits[2–4] monitor parallel program executions, but are less appropriate for high-level languages. These are implemented on top of a parallel runtime system (PRTS) which implements a parallel virtual machine. Standard profiling tools like `xpvm`[3] monitor the activity of the virtual processing elements (PEs or *machines*, usually mapped one-to-one to the physical ones), the message traffic between these PEs, and the behaviour of the underlying middleware like PVM[a] or MPI[b], instead of the program execution on top of the PRTS.

The Eden trace viewer tool (EdenTV) presented in this paper visualises the execution of parallel functional Eden programs at such a higher level of abstraction. EdenTV shows the activity of the *Eden* threads and processes, their mapping to the machines, stream communication between Eden processes, garbage collection phases, and the process generation tree, i.e. information specific to the Eden programming model. This supports the programmer's understanding of the parallel behaviour of an Eden program.

The Eden PRTS is instrumented with special trace generation commands activated by a runtime option. Eden programs themselves remain unchanged. Parts of the well-established Pablo Toolkit[5] and its *Self-Defining Data Format* (SDDF) are used for trace generation. EdenTV loads trace files after program execution and produces timeline diagrams for the computational units of Eden, i.e. threads, processes, and machines.

The aim of this paper is to show benefits of our high-level EdenTV profiling tool. Using a few simple case studies, we explain how the trace visualisations may help detect typical inefficiencies in parallel functional programs which may be due to delayed evaluation or bad load balancing. Section 2 sketches Eden's language concepts and its PRTS. Section 3

[a] `http://www.csm.ornl.gov/pvm/pvm_home.html`
[b] `http://www.mpi-forum.org`

shortly describes EdenTV. The central section 4 discusses typical weaknesses of program executions that can be detected with EdenTV and how to eliminate them. Section 5 points at related work, and the last section (6) concludes and indicates future work.

2 Eden

Eden[1] extends the functional language Haskell[c] with syntactic constructs for *explicitly* defining and creating parallel processes. The programmer has direct control over process granularity, data distribution and communication topology, but does not have to manage synchronisation and data exchange between processes. The latter are performed by the parallel runtime system through implicit communication channels, transparent to the programmer.

Coordination Constructs. The essential two coordination constructs of Eden are process abstraction and instantiation:

```
process :: (Trans a, Trans b) => (a -> b)    -> Process a b
 ( # )   :: (Trans a, Trans b) => Process a b -> a -> b
```

The function `process` embeds functions of type `a->b` into *process abstractions* of type `Process a b` where the context `(Trans a, Trans b)` states that both `a` and `b` must be types belonging to the `Trans` class of transmissible values.[d]

Evaluation of an expression `(process funct) # arg` leads to the creation of a new process for evaluating the application of the function `funct` to the argument `arg`. The argument is evaluated by new concurrent threads in the parent process and sent to the new child process, which, in turn, fully evaluates and sends back the result of the function application. Both are using implicit 1:1 *communication channels* established between child and parent process on process instantiation.

The type class `Trans` provides overloaded communication functions for *lists*, which are transmitted as streams, element by element, and for *tuples*, which are evaluated componentwise by concurrent threads in the same process. An Eden process can thus contain a variable number of threads during its lifetime.

As communication channels are normally connections between parent and child process, the communication topologies are hierarchical. In order to create other topologies, Eden provides additional language constructs to create channels dynamically, which is another source of concurrent threads in the sender process. Besides, Eden supports many-to-one communication by a nondeterministic merge function.

Parallel Runtime System. Eden is implemented on the basis of the Glasgow Haskell Compiler GHC[e], a mature and efficient Haskell implementation. While the compiler frontend is almost unchanged, the backend is extended with a *parallel* runtime system (PRTS)[6]. The PRTS uses suitable middleware (currently PVM or MPI) to manage parallel execution.
Concurrent threads are the basic unit for the implementation, so the central task for profiling is to keep track of their execution. Threads are scheduled *round-robin* and run through the straightforward state transitions shown in Figure 1.

---

[c] `http://www.haskell.org`
[d] Type classes in Haskell provide a structured way to define overloaded functions. `Trans` provides implicitly used communication functions.
[e] `http://www.haskell.org/ghc`

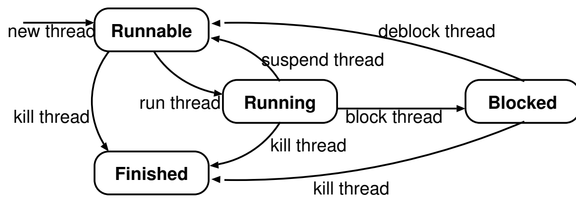


Figure 1. Thread State Transitions

An Eden process, as a purely conceptual unit, consists of a number of concurrent threads which share a common graph heap (as opposed to processes, which communicate via channels). The Eden PRTS does not support the migration of threads or processes to other machines during execution, so every thread is located on exactly one machine during its lifetime.

3 EdenTV — The Eden Trace Viewer

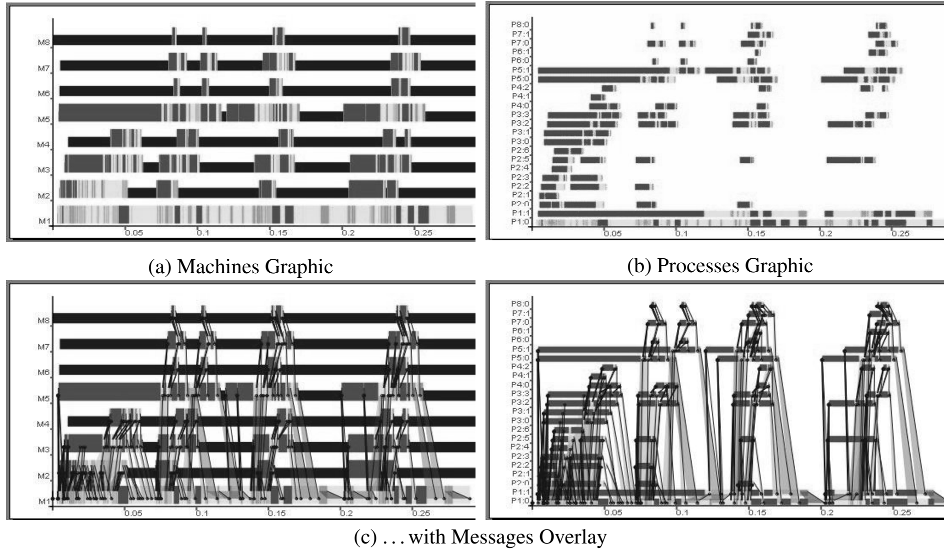
EdenTV enables a *post-mortem analysis* of program executions at the level of the PRTS. The steps of profiling are *trace generation* and *trace representation*, separated as much as possible in EdenTV, so that single parts can easily be maintained and modified for other purposes. Two versions of the trace representation tool are currently available. A Java implementation has been developed by Pablo Roldán Gómez[7]. Björn Struckmeier[8] did a re-implementation in Haskell which provides additional features.

Trace Generation. To profile the execution of an Eden program, we collect information about the behaviour of machines, processes, threads and messages by writing selected *events* into a trace file. These trace events, shown in Figure 2, indicate the creation or a state transition of a computational unit. Trace events are emitted from the PRTS, using the Pablo *Trace Capture Library*[5] and its portable and machine-readable "Self-Defining Data Format". Eden programs need not be changed to obtain the traces.

Start Machine	End Machine
New Process	Kill Process
New Thread	Kill Thread
Run Thread	Suspend Thread
Block Thread	Deblock Thread
Send Message	Receive Message

Figure 2. Trace Events

Trace Representation. In the timeline diagrams generated by EdenTV, machines, processes, and threads are represented by horizontal bars, with time on the x axis. EdenTV offers separate diagrams for *machines*, *processes*, and *threads*. The machines diagrams correspond to the view of profiling tools observing the parallel machine execution. Figure 3 shows examples of the machines and processes diagrams for a parallel divide-and-conquer program with limited recursion-depth. The trace has been generated on 8 Linux workstations connected via fast Ethernet. The diagram lines have segments in different colours, which indicate the activities of the respective logical unit in a period during the execution. As explained in Figure 3(d), thread states can be directly concluded from the emitted events. Machine and process states are assigned following a fundamental equation for the thread count inside one process or machine. The example diagrams in Figure 3 show that the program has been executed on 8 machines (virtual PEs). While there is some continuous activity on machine 1 (the bottom line) where the main program is started, machines 6 to 8 (the upper three lines) are idle most of the time. The corresponding processes graphic (see Figure 3(a)) reveals that several Eden processes have been allocated on each machine. The diagrams show that the workload on the parallel machines was low — there were only



(a) Machines Graphic

(b) Processes Graphic

(c) . . . with Messages Overlay

Thread Count:	$0 \leq$ Runnable Threads + Blocked Threads $\leq$ Total Threads				
Condition	**Machine**	**Process**	**Thread**	Black/White	Colour
Total Threads $= 0$	Idle	n/a	n/a	**smaller bar**	**Blue**
Blocked + Runnable $<$ Total	Running	Running	Running	**dark grey**	**Green**
Runnable Threads > 0	System Time	Runnable	Runnable	**light grey**	**Yellow**
Runnable Threads $= 0$	Blocked	Blocked	Blocked	black	Red

(d) Explanation of Colour Codes

Figure 3. Examples of EdenTV Diagrams and Colour Codes Table

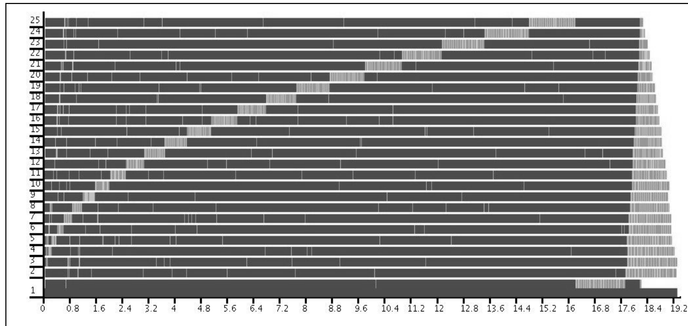
small periods where threads were running. Messages between processes or machines can optionally be shown by arrows which start from the sending unit line and point at the receiving unit line (see Figure 3(c)). The diagrams can be *zoomed* in order to get a closer view on the activities at critical points during the execution.

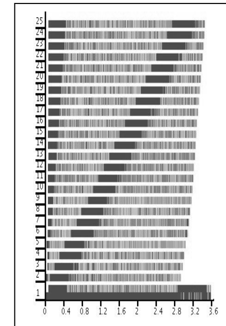
Additional Features. EdenTV provides additional information about the program run, e.g. a summary of the messages sent and received by processes and machines (on the right for the trace in Figure 3), stream communication is indicated by shading the area between the first and the last message of a stream (see Figure 3(c)), garbage collection phases and memory consumption can be shown in the activity diagrams, and the process generation tree can be drawn.

Machine	Runtime (sec)	Processes	Messages sent	Messages received
1	0.287197	4	6132	6166
2	0.361365	18	1224	1206
		⋮		
8	0.362850	6	408	402
Total	0.371875	66	14784	14784

4 Case Studies: Tuning Eden Programs with EdenTV

Lazy Evaluation vs. Parallelism. When using a lazy computation language, a crucial issue is to start the evaluation of needed subexpressions early enough and to fully evaluate them





Runtime: 19.37 sec. without additional demand control

Runtime: 3.62 sec. with additional demand control

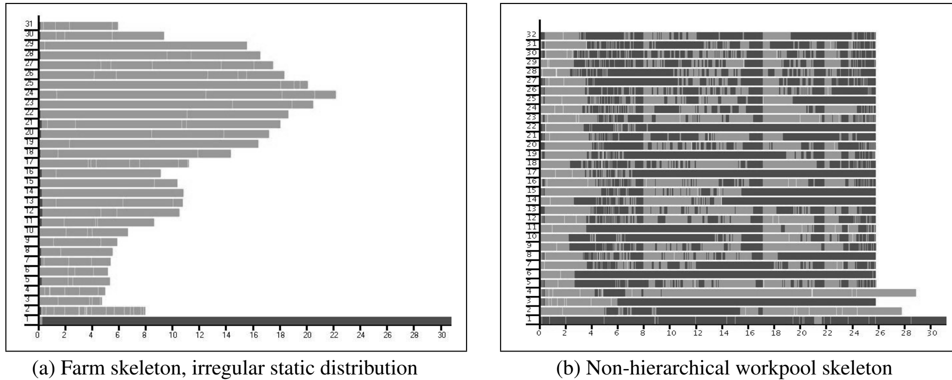
Figure 4. Warshall-Algorithm (500 node graph)

for later use. The basic choice to either evaluate a final result to weak head normal form (WHNF) or completely (to normal form (NF)) sometimes does not offer enough control to optimise an algorithm. Strategies[9] forcing additional evaluations must then be applied to certain sub-results. On the sparse basis of runtime measurements, such an optimisation would be rather cumbersome. The EdenTV, accompanied by code inspection, makes such inefficiencies obvious, as in the following example.

Example: **(Warshall's algorithm)** This algorithm computes shortest paths for all nodes of a graph from the adjacency matrix. We optimise a parallel implementation of this algorithm with a ring of processes. Each process computes the minimum distances from one node to every other node. Initialised with a row of the adjacency matrix, the direct distances are updated whenever a path of shorter distance via another node exists. All distance rows are continuously updated, and traverse the whole ring for one round.

The trace visualisations in Figure 4 show EdenTV's *Processes* view for two versions of the program on a Beowulf cluster, with an input graph of 500 nodes (aggregated on 25 processors). The programs differ by a single line which introduces additional demand for an early update on the local row. The first program version (without demand control) shows bad performance. The trace visualization clearly shows that the first phase of the algorithm is virtually sequential. A period of increasing activity traverses the ring, between long blocked (black) periods. Only the second phase, subsequent updates after sending the local row, runs in parallel on all machines. The cause is that when receiving a row, the local row is updated, but demand for this update only occurs at the time when the local row is sent through the ring. The second version enforces the evaluation of the temporary local row each time another row is received, which dramatically improves runtime. We still see the impact of the data dependence, leading to a short wait phase passing through the ring, but the optimised version shows good speedup and load balance. ◁

Irregularity and Cost of Load Balancing. Parallel skeletons (higher-order functions implementing common patterns of parallel computation[10]) provide an easy parallelization of a program. The higher-order function `map` applies a function to all elements of a list. These computations do not depend on each other, thus a number of worker processes can compute results in parallel. Parallel `map` variants either distribute the list elements statically (which is called a *farm* skeleton), or dynamically, by using a feedback from worker results to



(a) Farm skeleton, irregular static distribution

(b) Non-hierarchical workpool skeleton

Figure 5. *Processes* diagrams for different parallelizations of the Mandelbrot program

worker inputs (called a *workpool*). However, when the tasks expose an irregular complexity, the farm may suffer from an uneven load distribution, and the machine with the most complex tasks will dominate the runtime of the parallel computation. The workpool can adapt to the current load and speed of the different machines, but the dynamic distribution necessarily has more overhead than a static one.

Example: **(Mandelbrot)** We compare different work distribution schemes using a program which generates Mandelbrot Set visualizations by applying a uniform, but potentially irregular computation to a set of coordinates. In the traces shown in Figure 5, the pixel rows are computed in parallel, using either a farm or a workpool skeleton. Both programs have run on a Beowulf cluster with 32 processors, computing 5000 rows of 5000 pixels. The trace of the farm skeleton in Figure 5(a) shows that the workers have uneven workload. The blockwise row distribution even reflects the silhouette of the Mandelbrot graphic (bottom-to-top) in the runtimes of the worker processes.

The trace of the workpool skeleton in Figure 5(b) shows how the master process on machine 1 has to serve new tasks to 31 machines, and collect all results. The workers (2 to 32) spent a lot of time in blocked state, waiting for new work assigned by the heavily loaded master node. All workers stop at the same time, i.e. the system is well synchronized, but the single master is a severe bottleneck. ◁

Nesting and Process Placement. In Eden's PRTS, processes are placed either round-robin on available machines or randomly, but the implementation also supports explicit placement on particular machines. This can e.g. be used to optimally place processes in nested skeleton calls[11].

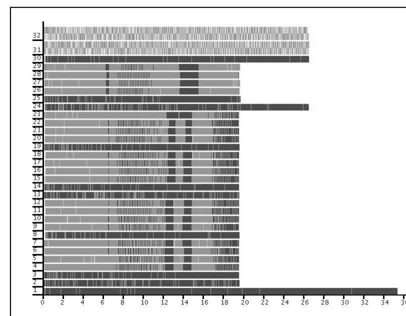


Figure 6. Hierarchical 3-level workpool

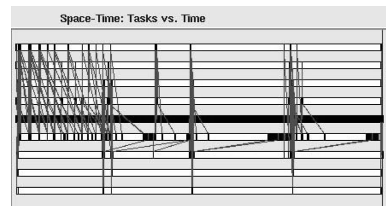
Example: **(Hierarchical Workpool)** The workpool master process can be relieved using a "hierarchy of workpools" with several stages of task distribution. Figure 6 shows the trace of a hierarchical workpool in three levels: a toplevel master, three submasters and two subsubmasters per submaster, each

controlling four workers. Thus, 10 of the 32 PEs are reserved as (sub/subsub)master processes, only 24 worker processes are allocated, and the four worker processes of the final group share two PEs. Nevertheless, the trace shows much better workload distribution and worker activity than in the non-hierarchical system. Only the two machines with two worker processes need more time, which is due to the higher initial workload caused by the identical prefetch for all worker processes.

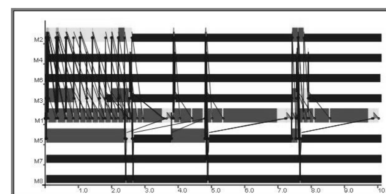
Collecting Results. All Mandelbrot traces suffer from long sequential end phases (omitted in Figure 5) which dominate the runtimes of the various schemes. An inspection of the message traffic reveals that all processes return their results to the root process (bottom line) which merges them at the end of the computation. It can be observed that the end phase is shorter within the hierarchical scheme, because the merging is done level-wise. Nevertheless, it is still too long in comparison with the parallel execution times of the worker processes. ◁

5 Related Work

A rather simple (but insufficient) way to obtain information about a program's parallelism would be to trace the behaviour of the communication subsystem. Tracing PVM-specific and user-defined actions is possible and visualization can be carried out by `xpvm`[3]. Yet, PVM-tracing yields only information about virtual PEs and concrete messages between them. Internal buffering, processes, and threads in the PRTS remain invisible, unless user-defined events are used.



(a) `xpvm` Space-Time Graphic



(b) EdenTV Machines Graphic

Figure 7. `xpvm` vs EdenTV

As a comparison, we show `xpvm` execution traces of our first trace examples. The space-time graphic of `xpvm` shown in Figure 7(a) corresponds to the EdenTV machine view of the same run in Figure 7(b), if the machines are ordered according to the `xpvm` order. The monitoring by `xpvm` extremely slows down the whole execution. The runtime increases from 0.37 seconds to 10.11 seconds. A reason for this unacceptable tracing overhead might be that PVM uses its own communication system for gathering trace information.

Comparable to the tracing included in `xpvm`, many efforts have been made in the past to create standard tools for trace analysis and representation (e.g. Pablo Analysis GUI[5], the ParaGraph Suite[12], or the Vampir system[4], to mention only a few essential projects). These tools have interesting features EdenTV does not yet include, like stream-based online trace analysis, execution replay, and a wide range of standard diagrams. The aim of EdenTV, however, is a specific visualization of logical Eden units, which needed a more customized solution. The EdenTV diagrams have been inspired by the per-processor view of the Granularity Simulator GranSim[13], a profiler for Glasgow parallel Haskell (GpH)[9], which, however, does not trace any kind of communication due to the different language concept.

6 Conclusions

The Eden Trace Viewer is a combination of an instrumented runtime system to generate trace files, and an interactive GUI to represent them in interactive diagrams. We have shown case studies exposing typical traps of parallelism like missing demand or imbalance in computation or communication, which can be identified using EdenTV. Load balancing issues and communication structures can be analysed and controlled. Essential runtime improvements could be achieved for our example programs.

Acknowledgements. The authors thank Pablo Roldán Gómez and Björn Struckmeier for their contributions to the Eden Trace Viewer project.

References

1. R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel Functional Programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.
2. I. Foster. *Designing and Building Parallel Programs, Chapter 9*. Addison-Wesley, 1995. `http://www.mcs.anl.gov/dbpp/`.
3. James Arthur Kohl and G. A. Geist. The PVM 3.4 tracing facility and XPVM 1.1. In *Proceedings of HICSS-29*, pages 290–299. IEEE Computer Society Press, 1996.
4. W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1), January 1996.
5. Daniel A. Reed and Robert D. Olson and Ruth A. Aydt et al. Scalable performance environments for parallel systems. In *Sixth Distributed Memory Computing Conference Proceedings (6th DMCC'91)*, pages 562–569, Portland, OR, 1991. IEEE.
6. Jost Berthold and Rita Loogen. Parallel Coordination Made Explicit in a Functional Setting. In *IFL'06, Selected Papers, LNCS 4449*. Springer, 2007.
7. Pablo Roldán Gómez. Eden Trace Viewer: Ein Werkzeug zur Visualisierung paralleler funktionaler Programme. Master's thesis, Philipps-Universität Marburg, Germany, 2004. In German.
8. Björn Struckmeier. Implementierung eines Werkzeugs zur Visualisierung und Analyse paralleler Programmläufe in Haskell. Master's thesis, Philipps-Universität Marburg, Germany, 2006. In German.
9. P.W. Trinder, K. Hammond, H-W. Loidl, and S.L. Peyton Jones. Algorithm + Strategy = Parallelism. *J. of Functional Programming*, 8(1), January 1998.
10. F. A. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2002.
11. Jost Berthold, Mischa Dieterle, Rita Loogen, and Steffen Priebe. Hierarchical master-worker skeletons. Technical report, Philipps-Universität Marburg, April 2007.
12. Michael T. Heath and Jennifer A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5), 1991.
13. H. W. Loidl. GranSim User's Guide. Technical report, University of Glasgow. Department of Computer Science, 1996.