HAL
open science

[Rp] Reproducing and replicating the OCamlP3l experiment

Roberto Di Cosmo, Marco Danelutto

To cite this version:

Roberto Di Cosmo, Marco Danelutto. [Rp] Reproducing and replicating the OCamlP3l experiment. The ReScience journal, 2020, 10.5281/zenodo.3763416 . hal-02885664

HAL Id: hal-02885664

https://hal.science/hal-02885664v1

Submitted on 30 Jun 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

RESCIENCE C

Reproduction / parallel programming

[Rp] Reproducing and replicating the OCamlP3l experiment

Di Cosmo, Roberto[1,2] and Danelutto, Marco[3]

[1]Inria, Paris, France – [2]Université de Paris, Paris, France – [3]University of Pisa, Pisa, Italy

Edited by
Konrad Hinsen

Reviewed by
Frédéric Gava

Received
24 March 2020

Published
23 April 2020

DOI
10.5281/zenodo.3763416

This article provides a full report on the effort to reproduce the work described in the article *"Parallel Functional Programming with Skeletons: the OCamlP3L experiment"* [1], written in 1998. It presented `OCamlP3l` [2], a parallel programming system written in the OCaml programming language [3].

The system described in [1] was a breakthrough in many respects: it showed that it was possible to implement *parallel skeletons* [4] a *combinators* in a functional programming language; it showed how this parallel programming style allowed to *write a single source code* that produced executables targeted for sequential execution, hence enabling usual debugging techniques, and executables for parallel execution; and it led to the introduction in OCaml of the ability to *marshal functional closures,* used later on by a wealth of different applications.

The article consists of two main parts, the system description, and the system evaluation, so replicating the results involves the following:

1. recover the source code of the `OCamlP3l` system
2. make it compile and run on a modern OCaml 4.x system
3. recover the tests used in the system evaluation
4. verify we can get speedup in performance similar to the one reported in the article.

When starting this replication effort, we had the following expectations:

1. recover the source code should be *easy*: just look in the paper directory on our machines
2. compile and run might be *difficult*: the code was designed 23 years ago for OCaml 1.07
3. recover tests should be *easy*: just look in the paper directory on our machines
4. verify speedup might be *challenging*: many parameters may have changed in microprocessors and network.

The reality turned out to be surprisingly different. In the following we sum up the steps that we performed to address each of these four challenges, and the final outcome.

Copyright © 2020 R. Di Cosmo and M. Danelutto, released under a Creative Commons Attribution 4.0 International license.
Correspondence should be addressed to Di Cosmo, Roberto (roberto@dicosmo.org)
The authors have declared that no competing interests exist.
Code is available at https://archive.softwareheritage.org/swh:1:rev:2db189928c94d62a3b4757b3eec68f0a4d4113f0;origin=https://gitorious.org/ocamlp3l/ocamlp3l_cvs.git/. – SWH swh:1:rev:2db189928c94d62a3b4757b3eec68f0a4d4113f0;origin=https://gitorious.org/ocamlp3l/ocamlp3l_cvs.git/.
Open peer review is available at https://github.com/ReScience/submissions/issues/22.

1 Recovering the source code

Looking into the original paper directory turned out to be of little help, as there was no trace of the source code or any useful information. So we turned to the paper itself, and found three links to web pages:

- www.di.unipi.it/~marcod/ocamlp3l/ocamlp3l.ml, that today returns `404`; looking at the archived copies on the archive.org allowed to recover some documentation, but not the source code;
- www.di.unipi.it/~susanna/p3l.ml, that is still live, but provides no useful link to the source code
- pauillac.inria.fr/ocaml, that is also live, but the only hope to find the source code was the link to the *anonymous CVS server* which points today to the OCaml GitHub organization, where we found no trace of this 23 years old code.

Searching the web The links from the original paper being now useless, we resorted to searching the web, and found http://ocamlp3l.inria.fr/. We followed the link http://ocamlp3l.inria.fr/eng.htm#download to the download page that offered an ftp link, ftp://ftp.inria.fr/INRIA/caml-light/bazar-ocaml/ocamlp3l/ocamlp3l-2.03.tgz, now dead, and web link, http://ocamlp3l.inria.fr/ocamlp3l-2.03.tgz that was still working. Unfortunately, this is version 2.03 of `OCamlP3l`, way more evolved, and quite different from the version 0.9 used in the original research article, and there was no trace of the version history, so the quest was far from over.

Saving version 2.03 Here we decided to make a pause, and properly deposit this version 2.03, with extended metadata, into Software Heritage [5] via the HAL national open access archive, the result being now availabe as [6].

Back to searching the web More web searches brough up a related webpage for a newer system, http://camlp3l.inria.fr/eng.htm touting a link to a git repository on Gitorious, http://gitorious.org/camlp3l/. Unfortunately, following the link leads to nowhere, as Gitorious has been shutdown in 2015, but luckily Software Heritage has saved the full content of Gitorious, so we could download a a full copy of the git repository, but unfortunately its version history only goes back to 2011, with version 1.03 of CamlP3l, not `OCamlP3l`, and no trace of earlier versions of the system, so we were seemingly back to square one.

Finding it on Software Heritage This long journey gave us an idea: what about searching directly in Software Heritage? This turned out to be the lucky strike: a full copy of https://gitorious.org/ocamlp3l/ocamlp3l_cvs.git had been safely archived by Software Heritage in 2015, and we found in it the whole version history starting from 1997. The journey ended successfully, *we had found the source code*!

2 Compiling and running

We did not know exactly which version of the source code was used in the article, but since the article was published in September 1998, it seemed safe to pick in the version control system a stable version dating from a few months before.

We settled for the version of code source available in the directory whose SWH-ID is swh:1:dir:01d2169c88d0783182b1b7facffa522ba09b5957 contained in the revision dated

June 23rd 1991 with SWH-ID swh:1:rev:2db189928c94d62a3b4757b3eec68f0a4d4113f0.

The code contained in this directory seems to be version 1.0 [7] and is classified as follows by the `sloccount` utility:

SLOC	Directory	SLOC-by-Language (Sorted)
1490	ocamlp3l-vprocess	ml=1490
1451	Source	ml=1451
1162	Examples	ml=1138,perl=13,csh=11
159	ocamlp3l-vthread	ml=159
67	Doc	ml=67
31	Tools	csh=31

To our great surprise, and satisfaction, the code compiled with the modern OCaml 4.05 installed on our machines *unchanged*. The only notable difference is that the modern compiler produces several new warnings that correspond to better static analysis checks introduced over the past *quarter of a century*.

This is a remarkable achievement, not just for our own code, but for OCaml itself.

3 Recovering the test suite and replicating speedup figures

Here too, looking into the original paper directory turned out to be of little help, as there was no trace of the test suite used in the article or any useful information. Web searches were of little interest, as this test suite was used only for the article and not published. A long search through old backups on tape, CR-ROMS and DVDs did not yield anything relevant either. Hence, our *reproducibility* journey ended here.

But we did not want to stop here: having found the original code, we could *replicate* the speed-up results, using *a new test suite*. After all, according to the article we wrote over 22 years ago, the original test suite was just producing a computational load to keep the compute nodes busy enough to take advantage of the parallelism.

As a first step, we adapted code in the Examples directory, from the SimpleFarm/simplefarm.ml [8] and the PerfTuning/pipeline.ml [9] files. The result is a simple parametric test code, shown in Figure 1, that allows to test the speedup one can get from the `farm` parallel skeleton in configurations obtained by varying the number `nproc` of processing nodes, and the time `msecwait` elapsed in each sequential computation.

The second step was to make the `ocamlp3lrun` driver command [10], that was using `rsh` (see these two occurrences) and `rcp` (see this occurrence) back in 1997 , work with the `ssh` and `scp` commands that are mainstream today.
A quick hack that works without even touching the code is to create an executable file `rsh` containing just the two lines:

```
#!
ssh $*
```

and similarly for `rcp`. Running the parallel test on a set of n different machines is then a simple matter of issueing the commands

```
ocamlp3lcc -par test-for-speedup.ml
ocamlp3lrun test-for-speedup <machine1> <machine2> ... <machinen>
```

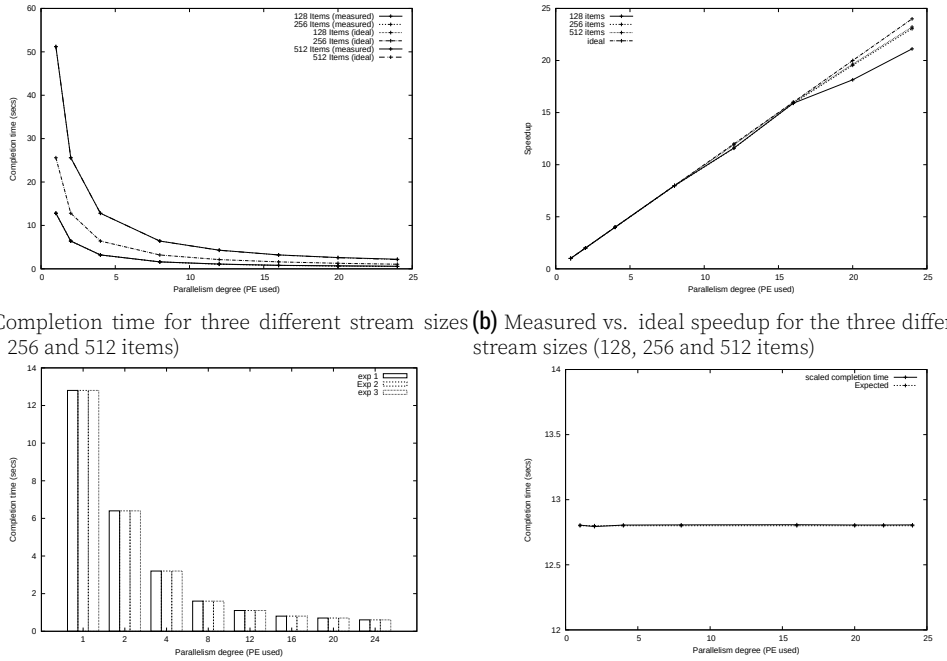
```
(* compute a function over a stream of floats using a farm  *)
(* very simple code to test the speed-up of a farm skeleton *)

let msecwait = 100;; (* time spent in sequential computation, microseconds *)
let nproc = 1;;      (* number of nodes allocated in the farm skeleton *)

(* active wait for n microseconds *)
let microwait n =
  let t = Unix.gettimeofday() in
  let now = ref (Unix.gettimeofday()) in
  while(!now < (t +. (0.001 *. (float n)))) do
    now := Unix.gettimeofday()
  done;;

let farm_worker comptime =
  (function x ->
  (microwait comptime ; x *. x));;

let nothing _ = ();;

let generate_input_stream =
  let x = ref 0.0 in
  (function () ->
    begin
      x := !x +. 1.0;
      if(!x < 129.0) then !x else raise End_of_file
    end);;

(* the timing stuff *)
let now = ref (Unix.gettimeofday());;
let stop_init _ =
  now := (Unix.gettimeofday());;
let stop_end _ =
  print_string "Elapsed time on stop node is ";
  print_float ((Unix.gettimeofday()) -. !now );
  print_newline();;

let do_nothing _ = ();;

let progr ()=
  startstop
    (generate_input_stream, nothing)
    (do_nothing,stop_init,stop_end)
    (farm(seq(farm_worker msecwait), nproc))
in pardo progr;;
```

test–for–speedup.ml

Figure 1. Test code for evaluating speedup of a farm skeleton, by varying the `nproc` and `msecwait` parameters. The exact source of the test suite has SWH-ID swh:1:cnt:8e7f96cb82d50ea73c2d8e4bf2c832b0ada49a7e

The third step was to run a parameter sweep experiment on a cluster available at the University of Pisa, and collect the data that was used to produce the new figures that we show in Figure 2.

The cluster is configured with 32 nodes each equipped with dual socket Intel(R) Xeon(R) CPU E5-2640 v4 2.40GHz. At the time of this experiment 5 nodes where busy or in maintainance and therefore our replication experiments were run with parallelism degrees $n_w \in [1 - 24]$. It is worth pointing out that the cluster nodes, differently from the ones used in the original experiments, sport 20 cores with 2-way hyperthreading. Hence, in order to replicate the very same experiments dating back to late '90s, we used only one process per node, as if the node had a single processor available.
Figure 2a and Figure 2b show the completion times and the relative speedups measured



(a) Completion time for three different stream sizes (128, 256 and 512 items)

(b) Measured vs. ideal speedup for the three different stream sizes (128, 256 and 512 items)

(c) Stability: Completion times measured in three different consecutive experiments

(d) Scaled speedup: stream size proportional to the parallelism degree

Figure 2. Replication results. Syntetic benchmark using a farm pattern.

in three different experiments, processing streams of data items of different lengths. The completion times are very close to the ideal ones, but looking at the speedup figures we can see that the larger the load the better speedup is achieved. Indeed, the interarrival time of tasks on the stream is negligible with respect to the time spent processing the single task and therefore longer streams help giving more work to each one of the "workers" in the parallel farm. This, in turn, results in a minor impact of the overheads associated to the set up and orchestration of the nodes that take part in the computation. Figure 2c shows the result of three different runs of the same experiments, executed at three different times of the same day on the cluster. We observe the same completion times, confirming the stability results already achieved at the time OcamlP3L was developed. Finally, Figure 2d reports the scaled speedup results. For each parallelism degree n_w, we used an input stream whose length was $k \times n_w$. The measured completion times are almost constants and close to the ideal one, which is the sequential time taken to compute a k item stream.

To sum up, we could replicate quite faithfully the quality of the results achieved more than 20 years ago. It is worth pointing out that this is a nontrivial achievement, as the architectures used for these experiments today and in the past are completely different, both in terms of computation power (processors) and in terms of communication bandwidth and latency (network interface cards).
In our opinion, this is clearly due to two distinct and synergic factors:

- the clean "functional" design and implementation of OcamlP3l, that resisted to language and system development, and
- the algorithmic skeleton[1] principles which are the base of the overall implementation of OcamlP3L that naturally implement "portability" across different architectures, independently of the fact the architectures use different hardware.

[1] aka "parallel design patterns"

4 Conclusion

We have reported on our experience in reproducing work we have done ourselves on the `OCamlP3l` experiment over 22 years ago [1]. Contrary to our expectations, the most difficult part has been to *recover the source code*. For its presevation, we had relied on institutional repositories first, and freely available collaborative development platforms later, neither of which passed the test of time.

We are delighted to report that leveraging the Software Heritage archive [5] we have been able to recover the full history of development of the system, and rebuild it as it likely was at the time the original article had been published. Despite the fact that we did not find the exact test suite used 22 years ago to test the scalability of the system, we have been able to *replicate the results* on modern hardware.

As a byproduct of this work, we have also safely archived in Software Heritage, and described in HAL, the stable final release 2.3 of `OCamlP3l` [6].

Based on this experience, we strongly suggest to systematically archive and reference research source code following the Software Heritage guidelines [11].

References

1. M. Danelutto, R. Di Cosmo, X. Leroy, and S. Pelagatti. "Parallel Functional Programming with Skeletons: the OCamlP3L experiment." In: **ACM Workshop on ML and its applications**. ACM. Baltimore, United States, Sept. 1998.
2. [SW] R. Di Cosmo, M. Danelutto, X. Leroy, and S. Pelagatti, **The OCamlP3l library**, 1998. LIC: GPL. URL: https://archive.softwareheritage.org/gitorious.org/ocamlp3l/ocamlp3l_cvs.git.
3. X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. **The OCaml system release 4.09: Documentation and user's manual**. Intern report. Inria, Sept. 2019, pp. 1–789.
4. M. Cole. **Algorithmic skeletons: structured management of parallel computation**. MIT Press, 1989.
5. J.-F. Abramatic, R. Di Cosmo, and S. Zacchiroli. "Building the Universal Archive of Source Code." In: **Commun. ACM** 61.10 (Sept. 2018), pp. 29–31.
6. [SW] R. Di Cosmo, P. Weis, F. Clement, and Z. Li, **OcamlP3l release 2.03**, 2007. LIC: LGPL-2. HAL: ⟨hal-02487579v1⟩, SWHID: ⟨swh:1:dir:85642a2e0333bbd6340c0a84ae6bad48cba11940;origin=https://hal.archives-ouvertes.fr/hal-02487579⟩.
7. [SW REL.] R. Di Cosmo, M. Danelutto, X. Leroy, and S. Pelagatti, **The OCamlP3l library** version 1.0, 1998. LIC: GPL. SWHID: ⟨swh:1:rev:2db189928c94d62a3b4757b3eec68f0a4d4113f0;origin=https://gitorious.org/ocamlp3l/ocamlp3l_cvs.git/⟩.
8. [SW EXC.] M. Danelutto, "Simple Farm test program," from **The OCamlP3l library** 1998. LIC: GPL. SWHID: ⟨swh:1:cnt:4d99d2d18326621ccdd70f5ea66c2e2ac236ad8b;origin=https://gitorious.org/ocamlp3l/ocamlp3l_cvs.git/;anchor=swh:1:rev:2db189928c94d62a3b4757b3eec68f0a4d4113f0;path=/Examples/SimpleFarm/simplefarm.ml⟩.
9. [SW EXC.] M. Danelutto, "Pipeline test program," from **The OCamlP3l library** 1998. LIC: GPL. SWHID: ⟨swh:1:cnt:8415f9451cf1ecaef70daab45c0ea2e5200f7d38;origin=https://gitorious.org/ocamlp3l/ocamlp3l_cvs.git;anchor=swh:1:rev:2db189928c94d62a3b4757b3eec68f0a4d4113f0;path=/Examples/PerfTuning/pipeline.ml⟩.
10. [SW EXC.] R. Di Cosmo, "Driver command," from **The OCamlP3l library** 1998. LIC: GPL. SWHID: ⟨swh:1:cnt:c428f4deb1cdff8500fff5c449b99454a816c163;origin=https://gitorious.org/ocamlp3l/ocamlp3l_cvs.git;anchor=swh:1:rev:2db189928c94d62a3b4757b3eec68f0a4d4113f0;path=/Tools/ocamlp3lrun⟩.
11. R. Di Cosmo. "How to use Software Heritage for archiving and referencing your source code: guidelines and walkthrough." HAL preprint hal-02263344 . Apr. 2019.