Reproducible Network Benchmarks with coNCePTuaL

Scott Pakin

Los Alamos National Laboratory, Los Alamos, NM 87545, USA,
pakin@lanl.gov
http://www.c3.lanl.gov/~pakin

Abstract. A cornerstone of scientific progress is the ability to reproduce experimental results. However, in the context of network benchmarking, system complexity impedes a researcher's attempts to record all of the information needed to exactly reconstruct a network-benchmarking experiment. Without this information, results may be misinterpreted and are unlikely to be reproducible.

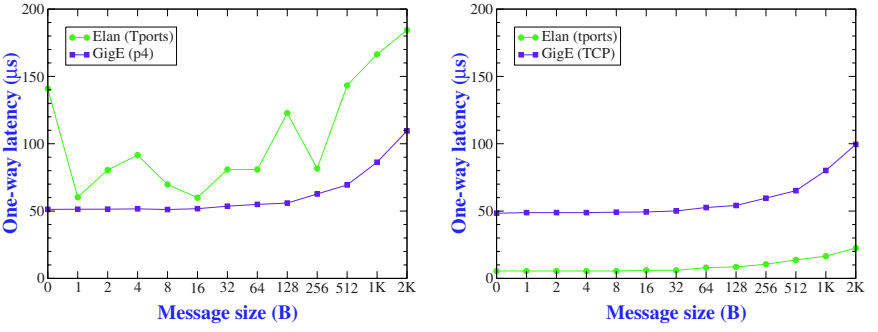
This paper presents a tool called coNCePTuaL which simplifies most aspects of recording and presenting network performance data. coNCePTuaL includes two core components: (1) a compiler for a high-level, domain-specific programming language that makes it possible to specify arbitrary communication patterns tersely but precisely and (2) a complementary run-time library that obviates the need for writing (and debugging!) all of the mundane but necessary routines needed for benchmarking, such as those that calibrate timers, compute statistics, or output log files. The result is that coNCePTuaL makes it easy to present network-performance data in a form that promotes reproducibility.

1 Introduction

Network and messaging-layer performance measurements are used for a variety of purposes, such as explaining or predicting system and application performance, procuring large-scale systems, and monitoring improvements made during system deployment or messaging-layer development. Unfortunately, following a truly scientific approach to measuring network performance is not easy. In the absence of clear but precise experimental descriptions, the consumers of network performance data may draw incorrect conclusions, leading to dire consequences.

Consider a standard ping-pong latency test, which reports the time needed to send a message of a given size from one node to another by calculating half of the measured round-trip time. Fig. 1(a) shows the result of running a latency test atop two messaging layers and networks: p4 [1] (based on TCP) over Gigabit Ethernet [2, Sect. 3] and Tports over Quadrics Elan 3 [3]. The latency test is implemented using MPI [4] and both the p4 and Tports layers are integrated as MPICH channel devices [5]. The latency program was compiled with GCC 2.96 using the `-O3` and `-g` flags. All experiments were performed across the same – otherwise idle – pair of nodes, each containing two 1 GHz Itanium 2 processors

M. Danelutto, D. Laforenza, M. Vanneschi (Eds.): Euro-Par 2004, LNCS 3149, pp. 64–71, 2004.



(a) First variation (b) Second variation

Fig. 1. Two variations of a latency test.

(one unused). Each data point represents the arithmetic mean of 11 executions of the latency test. As Fig. 1(a) shows, the p4/GigE version of the latency test exhibits lower latency than the Tports/Elan 3 version on all message sizes from 0 bytes to 2 kilobytes. Furthermore, the p4/GigE latency increases smoothly while the Tports/Elan 3 latency varies erratically.

Figure 1(b) also shows the result of running an MPI-based latency test. This test was run atop the same two messaging layers and networks as the previous test. The same compiler was used and the experiments were performed across the same nodes of the same cluster. As before, each data point represents the arithmetic mean of 11 sequential executions of the latency test and, as before, nothing else was running on the system. However, Fig. 1(b) delivers the opposite message from that delivered by Fig. 1(a): Fig. 1(b) shows that Tports/Elan 3 is significantly *faster* than p4/GigE. Also, the Tports/Elan 3 curve shown in Fig. 1(b) is smooth, unlike the erratic curve presented in Fig. 1(a).

Naturally, something is different between the experiment/experimental setup used in Fig. 1(a) from that used in Fig. 1(b) – but what? Although we defer the answer to Sect. 2 the point is that even with all of the experimental setup described above, the performance results are untrustworthy; some critical piece of information is missing. This simple exercise demonstrates the problem with the status quo of network benchmarking: performance data that lacks a complete and precise specification is subject to misinterpretation. This paper proposes a solution in the form of a programming environment called coNCePTuaL which was designed specifically to simplify the implementation of reproducible network-performance tests. The rest of this paper is structured as follows. Sect. 2 motivates and describes coNCePTuaL and showcases some sample output. Sect. 3 places coNCePTuaL in context, discussing the types of tasks for which coNCePTuaL is best suited. Finally, Sect. 4 draws some conclusions about the implications of using coNCePTuaL for network performance testing.

2 coNCePTuaL

Fig. 1 on the preceding page shows how subtle differences in experimental setup can lead to radically different performance results. coNCePTuaL (= "Network Correctness and Performance Testing Language") is a programming environment designed to help eliminate the ambiguities that can limit the usefulness of performance results. It centers around a high-level, domain-specific language created for the express purpose of writing network benchmarks. The design decision to introduce a new language instead of merely creating a performance library stemmed from the desire to make coNCePTuaL programs more readable than a jumble of function calls and control structures. Although a library can certainly encapsulate all of the functionality needed for the scientific acquisition and reporting of data and a textual or pseudocode description of a benchmark can convey the basic idea, coNCePTuaL combines the best features of both approaches:

1. Like pseudocode or prose but unlike the combination of a general-purpose programming language and a library, coNCePTuaL programs are English-like and can largely be read and understood even by someone unfamiliar with the language.
2. Like the combination of a general-purpose programming language and a library but unlike pseudocode or prose, coNCePTuaL programs precisely describe all aspects of a benchmark, most importantly the implementation subtleties that may be omitted from a description yet have a strong impact on performance (as demonstrated by the previous section's description of Fig. 1).

The coNCePTuaL compiler, written in Python with the SPARK compiler framework [6], sports a modular design that enables generated code to target any number of lower-level languages and messaging layers[1]. Hence, the same high-level coNCePTuaL program can be used, for example, to compare the performance of multiple messaging layers, even semantically disparate ones such as MPI and OpenMP. The generated code links with a run-time library that takes care of most of the mundane aspects of proper benchmarking, such as calculating statistics, calibrating timers, parsing command-line options, and logging a wealth of information about the experimental setup to a file.

The coNCePTuaL language provides too much functionality to describe in a short paper such as this; the reader is referred to the coNCePTuaL user's manual [7] (available online) for coverage of the language's syntax and semantics. In lieu of a thorough description of the language, we now present a few trivial code samples with some accompanying explanation to convey a basic feeling for coNCePTuaL. Listing 1 presents the complete coNCePTuaL source code which produced the data for Fig. 1(a) and Listing 2 presents the complete coNCePTuaL source code that produced the data for Fig. 1(b). One thing

[1] Currently, the only backends implemented are C + MPI and C + Unix-domain datagram sockets; more backends are under development.

Listing 1. Source code that produced Fig. 1(a)

```
msgsize is "Message size (bytes)" and comes from "--bytes" or "-b"
with default 0.

All tasks synchronize then
task 0 resets its counters then
task 0 sends a msgsize byte message to task 1 then
task 1 sends a msgsize byte message to task 0 then
task 0 logs elapsed_usecs/2 as "1/2 RTT".
```

Listing 2. Source code that produced Fig. 1(b)

```
msgsize is "Message size (bytes)" and comes from "--bytes" or "-b"
with default 0.

All tasks synchronize then
for 100 repetitions {
  task 0 resets its counters then
  task 0 sends a msgsize byte message to task 1 then
  task 1 sends a msgsize byte message to task 0 then
  task 0 logs the median of elapsed_usecs/2 as "1/2 RTT"
}
```

to note is that the language is very English-like; coNCePTuaL programs are intended to read much like a human would describe a benchmark to another human. Although one can probably grasp the gist of Listings 1–2 without further explanation, the following details may be slightly unintuitive:

- The first statement in Listings 1 and 2 parses the command line, assigning the argument of `--bytes` (or simply `-b`) to a variable called msgsize.
- To simplify the common case, **sends** is synchronous and implies a matching synchronous **receives**. Either or both operations can be asynchronous. coNCePTuaL also supports data verification and arbitrary buffer alignment.
- The coNCePTuaL run-time library automatically maintains the elapsed_usecs ("elapsed time in microseconds") counter and many other counters, as well.

If not yet obvious, the explanation of the performance discrepancy shown in Fig. 1 is that the code shown in Listing 1 measures only a single ping-pong while the code shown in Listing 2 reports the median of 100 ping-pong iterations. Unlike p4, which uses TCP and therefore goes through the operating system for every message, Tports is a user-level messaging layer that communicates directly with the Elan. However, the Elan, which can transfer data from arbitrary addresses in an application's virtual-memory space, must pin (i.e., prevent the paging of) message buffers before beginning a DMA operation. Because pinning requires both operating-system intervention and a sequence of costly I/O-bus

crossings, a large startup overhead is incurred the first time a message buffer is utilized. The code shown in Listing 1 does not amortize that overhead while the code shown in Listing 2 does. Neither latency test is unreasonable; although codes like Listing 2 are more common in the literature, codes like Listing 1 are used when they more accurately represent an application's usage pattern. For example, two of the three execution modes of Worley's COMMTEST benchmark, which is used to help tune climate and shallow-water modeling applications [8], specify that each message be sent only once.

Although Listings 1–2 state simply "**task** 0 **logs** ⟨*something*⟩", the coNCePTuaL run-time library takes this as a cue to write a set of highly detailed log files, one per task. Each log file contains information about the execution environment, a list of all environment variables and their values, the complete program source code, the program-specific measurement data, and a trailer describing the resources used during the program's execution.

A sample log file is shown below. This particular file – selected arbitrarily from those used in the preparation of this paper – corresponds to one of the values averaged to make the upper-right data point of Fig. 1(a), although the file represents a later run than was used in the figure.

```
1  ###############################################################################
2  # ==================
3  # coNCePTuaL log file
4  # ==================
5  # coNCePTuaL version: 0.5.1
6  # coNCePTuaL backend: c_mpi (C + MPI)
7  # Executable name: /home/pakin/src/coNCePTuaL/latency1-elan
8  # Working directory: /home/pakin/src/coNCePTuaL
9  # Command line: ./latency1-elan --bytes=2048 --logfile=latency1-elan-2048-run4-%d.log
10 # Number of tasks: 2
11 # Processor (0-1): 0
12 # Host name: a11
13 # Operating system: Linux 2.4.21-3.5qsnet #2 SMP Thu Aug 7 10:51:04 MDT 2003
14 # CPU vendor: GenuineIntel
15 # CPU architecture: ia64
16 # CPU model: 1
17 # CPU count: 2
18 # CPU frequency: 1300000000 Hz (1.3 GHz)
19 # Cycle-counter frequency: 1300000000 Hz (1.3 GHz)
20 # OS page size: 16384 bytes
21 # Physical memory: 2047901696 bytes (1.9 GB)
22 # Library compiler+linker: /usr/bin/gcc
23 # Library compiler options: -Wall -W -g -O3
24 # Library linker options: -lpapi -lm -lpopt
25 # Library compiler mode: LP64
26 # Dynamic libraries used: /usr/local/lib/libpapi.so /lib/libm-2.2.4.so /usr/lib/libpopt.so.0.0.0 /usr/
27 # Average microsecond timer overhead [inline assembly code]: <1 microsecond
28 # Microsecond timer increment: 1.00887 +/- 0.256529 microseconds (ideal: 1 +/- 0)
29 # Process CPU-time increment [getrusage()]: 976.57 +/- 0.49757 microseconds (ideal: 1 +/- 0)
30 # WARNING: Process timer exhibits poor granularity (not a serious problem).
31 # Log file template: latency1-elan-2048-run4-%d.log
32 # Message size (bytes): 2048
33 # Number of minutes after which to kill the job (-1=never): -1
34 # List of signals that should not be trapped: 14
35 # Compilation command line: /usr/lib/mpi/mpi_gnu/bin/mpicc -I/tmp/ncptl/include  -I/usr/local/include
36 # Log creator: Scott Pakin
37 # Log creation time: Thu May 27 18:45:43 2004
38 #
39 # Environment variables
40 # ---------------------
41 # HOME: /home/pakin
42 # PATH: /home/pakin/bin:/usr/local/bin:/usr/bin:/usr/sbin:/bin:/sbin:.
43 # PWD: /home/pakin/src/coNCePTuaL
44 # RMS_JOBID: 16588
45 # RMS_MACHINE: a
46 # RMS_NNODES: 2
47 # RMS_NODEID: 0
48 # RMS_NPROCS: 2
49 # RMS_PROCID: 0
50 # RMS_RANK: 0
51 # RMS_RESOURCEID: parallel.17079
52 # RMS_STOPONELANINIT: 0
53 # SHELL: /bin/tcsh
```

```
54  # USER: pakin
55  #
56  # coNCePTuaL source code
57  # ----------------------
58  #     msgsize is "Message size (bytes)" and comes from "--bytes" or "-b"
59  #     with default 0.
60  #
61  #     All tasks synchronize then
62  #     task 0 resets its counters then
63  #     task 0 sends a msgsize byte message to task 1 then
64  #     task 1 sends a msgsize byte message to task 0 then
65  #     task 0 logs elapsed_usecs/2 as "1/2 RTT".
66  #
67  ###############################################################################
68  "1/2 RTT"
69  "(all data)"
70  207
71  ###############################################################################
72  # Program exited normally.
73  # Log completion time: Thu May 27 18:45:43 2004
74  # Elapsed time: 0 seconds
75  # Process CPU usage (user+system): 0 seconds
76  # Task IDs assigned to processor 0: 0
77  # Processors assigned to task ID 0: 0
78  ###############################################################################
```

The key point is that a coNCePTuaL log file contains not just performance measurements but also a detailed description of how they were produced, which helps third parties understand the performance results. Such complete log files are also of use to the people who generated them: How often does a researcher struggle to use old data in a new paper or presentation, not remembering if `results.dat` was produced with the environment variable `MPI_BUFFER_MAX` set to `2048` or `1048576`; whether the experiment was run with the older 2.8 GHz processors or the newer 3.2 GHz processors; or, even if the test transmitted messages synchronously or asynchronously? Furthermore, aspects of the execution environment that cannot be determined automatically (e.g., characteristics of the network fabric) can be inserted manually into a log file with a command-line option to the benchmark program. In short, with coNCePTuaL, log files present a complete picture of an experiment, making them far more valuable than measurement data alone.

3 Discussion

The programs presented in Sect. 2 are simple to express in coNCePTuaL but – apart from the creation of such content-rich log files – would be almost as simple to express in any other language. In general, coNCePTuaL's usefulness increases with the complexity of the communication pattern being tested. For example, the 4×4 synchronous-pipe pattern described in a MITRE report [9] requires 248 lines of LSE, a terse but low-level language for describing communication benchmarks. Because coNCePTuaL is a high-level language, the same code (in fact, a more general $M \times N$ synchronous pipe) can be expressed in only 26 lines of coNCePTuaL – far less than the LSE version and not significantly more than what would be needed for a textual description of the communication pattern. It is not merely short code lengths that make coNCePTuaL useful; the increased comprehensibility of a coNCePTuaL program over the equivalent program written in a lower-level language and the increased precision of a coNCePTuaL program over a prose description make coNCePTuaL a useful

tool for any sort of network performance testing. The language even supports a hybrid coding style in which lower-level language code can be inlined and executed from a coNCePTuaL program, thereby ensuring that no functionality is lost by programming in coNCePTuaL instead of a lower-level language.

coNCePTuaL is not intended to be a replacement for existing communication-benchmark suites such as the Pallas MPI Benchmarks [10] or SKaMPI [11]. Rather, its real strengths lie in its ability to rapidly produce customized tests of network and messaging-layer performance:

- A coNCePTuaL mock-up of an application may make "what if" scenarios more easy to evaluate than would rewriting the original application. For instance, a user can evaluate how altering the communication pattern (caused, for example, by a different data decomposition) should affect overall application performance. A coNCePTuaL mock-up of Sweep3D [12] is currently under development.
- System and application performance problems can be diagnosed by generating a simple but representative communication benchmark and successively refining it to hone in on the source of the problem. (This methodology was recently used to nearly double the performance of an application running on ASCI Q [13].)
- Network-performance tests unique to a particular domain or otherwise unfamiliar to a target audience can be presented in a precise, easily understood manner.

The coNCePTuaL source code will soon be available from http://www.c3.lanl.gov/~pakin/software/. Making the software open-source enables researchers to scrutinize the code so that coNCePTuaL can be used as a trustworthy replacement for C as a network-benchmarking language.

4 Conclusions

In the domain of network benchmarks, recorded performance data cannot blindly be trusted. As demonstrated in Sect. 1, subtle variations in experimental setup – even for a benchmark as trivial as a latency test – can lead to grossly varying performance curves, even leading to different conclusions being drawn about relative performance. The problem is that the complexity of current computer systems makes it difficult (not to mention tedious) to store a sufficiently thorough depiction of an experiment that was run and the experimental conditions under which it ran. As a consequence, performance tests can rarely be reproduced or validated in a scientific manner. Even unpublished performance data used locally suffers from lack of reproducibility; a researcher may unearth old measurements but have no record of what benchmark produced them or what parameters were utilized in the process.

This paper proposes the coNCePTuaL programming environment as a solution to the problem of irreproducible network performance results. coNCePTuaL tries to codify the best practices in network and messaging-layer performance testing into a high-level domain-specific language and accompanying

run-time library. coNCePTuaL was designed specifically to support and facilitate all aspects of network and messaging-layer performance testing, from expressing complex communication patterns tersely yet unambiguously through storing in self-contained log files everything needed to reproduce an experiment. Using coNCePTuaL, a researcher can easily present in a paper or report a benchmark's actual source code – not pseudocode, which may inadvertently omit critical details. Although it will always be possible to misrepresent network performance, coNCePTuaL makes it much easier to be meticulous.

References

1. Butler, R.M., Lusk, E.L.: Monitors, messages, and clusters: The p4 parallel programming system. Parallel Computing **20** (1994) 547–564
2. LAN/MAN Standards Committee: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications. IEEE Standard 802.3, IEEE Computer Society, Technical Committee on Computer Communications, New York, New York (2002)
3. Petrini, F., Feng, W., Hoisie, A., Coll, S., Frachtenberg, E.: The Quadrics network: High-performance clustering technology. IEEE Micro **22** (2002) 46–57
4. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard. (1995)
5. Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A high-performance, portable implementation of the MPI message passing interface standard. Parallel Computing **22** (1996) 789–828
6. Aycock, J.: Compiling little languages in Python. In: Proceedings of the Seventh International Python Conference, Houston, Texas (1998) 69–77
7. Pakin, S.: coNCePTuaL user's guide. Los Alamos Unclassified Report 03-7356, Los Alamos National Laboratory, Los Alamos, New Mexico (2003) Available from http://www.c3.lanl.gov/~pakin/papers/conceptual.pdf.
8. Drake, J.B., Hammond, S., James, R., Worley, P.H.: Performance tuning and evaluation of a parallel community climate model. In: Proceedings of SC'99, Portland, Oregon (1999)
9. Monk, L., Games, R., Ramsdell, J., Kanevsky, A., Brown, C., Lee, P.: Real-time communications scheduling: Final report. Technical Report MTR 97B0000069, The MITRE Corporation, Bedford, Massachusetts (1997)
10. Pallas, GmbH: Pallas MPI Benchmarks—PMB, Part MPI-1. (2000)
11. Reussner, R., Sanders, P., Prechelt, L., Müller, M.: SKaMPI: A detailed, accurate MPI benchmark. In Alexandrov, V., Dongarra, J., eds.: Recent Advances in Parallel Virtual Machine and Message Passing Interface: Proceedings of the 5th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'98). Volume 1497 of Lecture Notes in Computer Science., Liverpool, United Kingdom, Springer-Verlag (1998) 52–62
12. Hoisie, A., Lubeck, O., Wasserman, H.: Performance and scalability analysis of teraflop-scale parallel architectures using multidimensional wavefront applications. The International Journal of High Performance Computing Applications **14** (2000)
13. Petrini, F., Kerbyson, D.J., Pakin, S.: The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In: Proceedings of SC2003, Phoenix, Arizona (2003)