Rapid Development of Application-Specific Network Performance Tests

Scott Pakin

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

Abstract. Analyzing the performance of networks and messaging layers is important for diagnosing anomalous performance in parallel applications. However, general-purpose benchmarks rarely provide sufficient insight into any particular application's behavior. What is needed is a facility for rapidly developing customized network performance tests that mimic an application's use of the network but allow for easier experimentation to help determine performance bottlenecks.

In this paper, we contrast four approaches to developing customized network performance tests: straight C, C with a helper library, Python with a helper library, and a domain-specific language. We show that while a special-purpose library can result in significant improvements in functionality without sacrificing language familiarity, the key to facilitating rapid development of network performances tests is to use a domain-specific language designed expressly for that purpose.

1 Introduction

Parallel applications utilize the interconnection network in a variety of ways, including nearest-neighbor communication on a 2-D or 3-D mesh/torus (e.g., in ocean-modeling codes [1]); hierarchical communication (e.g., in molecular-dynamics codes [2]); and, master/slave communication (e.g., in Monte Carlo codes [3]). However, general-purpose network performance tests such as NetPIPE [4], Mpptest [5], and those that appear in the Pallas MPI Benchmarks [6] and SKaMPI [7] suites, measure performance independently of any particular application's usage of the network. For example, it is common to measure network bandwidth as the peak data rate achieved when sending a large number of messages back-to-back between two otherwise idle endpoints, even though few applications utilize such a communication pattern. General-purpose tests are nevertheless important to application developers because they indicate – in a standard format – upper bounds in network performance that developers can use to determine if application performance is being limited by the network.

Special-purpose benchmarks targeted to a particular inquiry are an important complement to general-purpose benchmarks. For example, if an application runs significantly slower than a general-purpose test would indicate, it may be worthwhile to extract the application's particular communication pattern into a separate test program and perform *in vivo* experiments with that (simulating message aggregation, varying message-buffer alignment, reducing communication granularity – all on a real cluster with a real network), the idea being that it is quicker and easier to modify a small test program than a large application. Unfortunately, special-purpose tests receive little attention in practice and in the literature because they can be time-consuming to write and debug; and, because they may be run only a few times before being discarded, few application developers consider the benefits worth the effort.

In this paper, we investigate three approaches intended to facilitate the rapid generation of special-purpose network performance tests and compare these to a baseline test written in standalone C. Section 2 describes the sample performance test which is used as the basis for comparison, lists the metrics used to evaluate the alternatives, and presents the baseline C implementation. Section 3 describes a library for performance testing and examines the improvement over the baseline C code when used with C and with Python. Then, Sect. 4 introduces the coNCePTuaL language, shows how the sample performance test can be rewritten in coNCePTuaL, and highlights the benefits of doing so. Finally, Sect. 5 draws some conclusions about the results of this study.

2 Problem Specification

When selecting a sample problem to use as a running example throughout this paper, the challenge is to choose a communication pattern that is neither too common (such as a latency or bandwidth benchmark) nor too esoteric (such as one so targeted to a single application that the results do not generalize to other patterns). Rather than create an appropriate problem ourselves, we borrow one from a set of exercises associated with a long-existing MPI tutorial [8]. This particular exercise, entitled "Exploring the cost of synchronization delays"[1] reads as follows (unedited):

> In this example, 2 processes are communicating with a third. Process 0 is sending a long message to process 1 and process 2 is sending a relatively short message to process 1 and then to process 0. Arrange the code so that process 1 has already posted an MPI_Irecv for the message from process 2 before receiving the message from process 0, but also ensure that process 1 receives the long message from process 0 before receiving the message from process 2.
>
> This seemingly complex communication pattern mimics a pattern that can occur in an application due to timing variations on each processor. If the message sent by process 2 to process 1 is short but long enough to require a rendezvous protocol, there can be a sigificant delay

[1] The problem statement and sample solution are available on the Web at `http://www-unix.mcs.anl.gov/mpi/tutorial/mpiexmpl/src3/3way/C/main.html`.

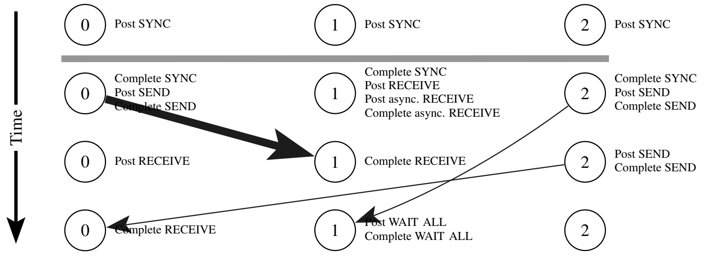

Fig. 1. Communication pattern described by the sample problem

before the short message from process 2 is received by process 1, even though the receive for that message is already available. Explore the possibilities by considering various lengths of messages.

In essence, this is the sort of performance test an application developer would create if his application uses such a communication pattern and he wants to find the source of its performance problems.

The sample solution, written in C and using MPI as the communication library, is 74 lines long and is illustrated in Fig. 1. The core communication and reporting routines are presented below but the reader is directed to the URL shown at the bottom of page 150 for the complete listing.

(44 lines omitted)

```
    MPI_Barrier(MPI_COMM_WORLD);

    if (myrank == 0) {
        MPI_Send( bigdata, BIGSIZE, MPI_DOUBLE, 1, tag, MPI_COMM_WORLD);
        MPI_Recv( litdata, litsize, MPI_DOUBLE, 2, tag, MPI_COMM_WORLD, &status);
    }
    else if (myrank == 1) {
        MPI_Irecv(litdata, litsize, MPI_DOUBLE, 2, tag, MPI_COMM_WORLD, &request);
        MPI_Send (litdata, litsize, MPI_DOUBLE, 2, tag, MPI_COMM_WORLD);
        MPI_Recv( bigdata, BIGSIZE, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD, &status
              );
        MPI_Wait( &request, &status );
    }
    else if (myrank == 2) {
        MPI_Recv( litdata, litsize, MPI_DOUBLE, 1, tag, MPI_COMM_WORLD, &status);
        t1 = MPI_Wtime();
        MPI_Send( litdata, litsize, MPI_DOUBLE, 1, tag, MPI_COMM_WORLD);
        ts1 = MPI_Wtime() − t1;
        t1 = MPI_Wtime();
        MPI_Send( litdata, litsize, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD);
        ts2 = MPI_Wtime() − t1;
    }

    if (myrank == 2) {
        printf("[%d] Litsize = %d, Time for first send = %f, for second = %f\n",
                myrank, litsize, ts1, ts2 );
    }
```

(4 lines omitted)

There are a number of shortcomings of the preceding code:

1. Too little information is output. Without a more detailed record of the experimental setup (e.g., shared libraries and versions being used, compiler flags, environment variables, etc.) the application developer may overlook some performance-affecting detail. Ideally, much more information should be logged but doing so can require daunting coding effort.
2. 48 lines of initialization and finalization code (not shown) is too long. Application developers are unlikely to consider writing a special-purpose performance test if including header files, declaring variables, initializing the messaging layer, parsing the command line, and doing all of the other banal, non-performance-related activities take so much coding time.
3. The level of abstraction is too low. It is difficult to ascertain which lines of code correspond to which parts of the problem statement.

We attempt to address these shortcomings in Sects. 3 and 4.

3 A Library for Performance Testing

Regardless of what particular communication patterns or performance characteristics a benchmark tests, there are a number of mundane operations that the code must perform such as parsing the command line, recording elapsed time, computing performance statistics, and logging results to a file. Because such operations are of common utility, it is practical to implement them in a helper library so they can be reused by numerous performance tests.

For the pupose of this study, we use an existing library, the run-time performance library used by coNCePTuaL [9]. This library exports a rich set of functions intended to simplify performance testing. The reader is referred to the coNCePTuaL User's Guide [10] for details.

None of the performance library's functions are specific to any particular messaging layer. Consequently, the MPI calls in the code shown on page 151 remain intact while the library improves the following constructs:

Parsing the Command Line. Rather than explicitly scanning `argv[]` as in the original code, using `ncptl_parse_command_line()` provides error checking, support for short and long option names, and a `--help` option which describes each of the supported options.

Touching the Message Buffers. The original code writes dummy values to the message buffers as part of initialization. On CPUs with write-no-allocate cache policies, this will not yield the desired result of preloading the buffers into the cache. `ncptl_touch_data()`, in contrast, both reads and writes each word of the buffer. This is a prime example of the usefulness of a helper library: Once the library code is written, debugged, and made portable, all programs that link to the library automatically benefit.

Reading the Timer. The `ncptl_time()` function reads the highest-resolution timer available. Also, the log-file header includes the timer overhead, mean increment, and increment standard deviation, all of which are measured dynamically during initialization time [9]. Hence, unlike `MPI_Wtime()` – which, to begin with, is specific to MPI – the application developer knows exactly how reliable the platform's timing measurements can be.

Outputting Results. Replacing the original code's lone `printf()` with a set of `ncptl_log_`*something*`()` calls provides a number of benefits, the most important of which is that it helps make the performance test *reproducible*. A prior publication expands upon this issue and presents a sample log file [9] but the key idea is that by logging not only measurement data but also the entire experimental setup – system architecture, software used, timer accuracy, environment variables, etc. – log files become self-documenting. This is important to application developers because they can use a special-purpose performance test to diagnose a problem. Then, when they believe they have fixed the problem, they can re-run the performance test in exactly the same way that it was run previously and accurately compare the results.

The following shows how the original code's `printf()` can be replaced by library functions to improve program reproducibility:

(66 lines omitted)

```
    if (myrank == 2) {
        /  Log the results to a file.  /
        NCPTL_LOG_FILE_STATE *logfile = ncptl_log_open("syncdelays−conclib−%p.log",
              myrank);
        ncptl_log_write_header (logfile, argv[0], "N/A", "N/A", myrank, numprocs, conc_args,
              1, NULL);
        ncptl_log_write (logfile, 0, "Litsize", NCPTL_FUNC_NO_AGGREGATE, (double)
              litsize);
        ncptl_log_write (logfile, 1, "Time for first send", NCPTL_FUNC_NO_AGGREGATE, (
              double)ts1);
        ncptl_log_write (logfile, 2, "Time for second send", NCPTL_FUNC_NO_AGGREGATE
              , (double)ts2);
        ncptl_log_commit_data (logfile);
        ncptl_log_write_footer (logfile);
        ncptl_log_close (logfile);
    }
```

(4 lines omitted)

Using a performance-testing-centric library addresses shortcoming 1 on page 152. However, it does not address the remaining two shortcomings. This raises the question: Can special-purpose performance tests be developed more rapidly using a high-level language instead of C? The intention is to reduce code turnaround time and to relieve the application developer of the tedium of declaring variables, allocating and deallocating memory, and performing other low-level operations. To answer the preceding question, we re-coded the solution to the sample problem in Python – specifically, ScientificPython [11] for its MPI support – while continuing to use the performance library, which has a Python interface. The core communication and reporting routines are presented below:

(28 lines omitted)

```
comm_world.barrier()

if myrank == 0:
    comm_world.send(bigdata, 1, tag)
    comm_world.receive(litdata, 2, tag)
elif myrank == 1:
    request = comm_world.nonblockingReceive(litdata, 2, tag)
    comm_world.send(litdata, 2, tag)
    comm_world.receive(bigdata, 0, tag)
    request.wait()
elif myrank == 2:
    comm_world.receive(litdata, 1, tag)
    t1 = ncptl_time()
    comm_world.send(litdata, 1, tag)
    ts1 = ncptl_time() - t1
    t1 = ncptl_time()
    comm_world.send(litdata, 0, tag)
    ts2 = ncptl_time() - t1

if myrank == 2:
    # Log the results to a file.
    logfile = ncptl_log_open("syncdelays-pyconclib-%p.log", myrank)
    ncptl_log_write_header(logfile, sys.argv[0], "N/A", "N/A", myrank,
                           numprocs, conc_args, 1, [])
    ncptl_log_write(logfile, 0, "Litsize", NCPTL_FUNC_NO_AGGREGATE, litsize)
    ncptl_log_write(logfile, 1, "Time for first send", NCPTL_FUNC_NO_AGGREGATE, ts1)
    ncptl_log_write(logfile, 2, "Time for second send", NCPTL_FUNC_NO_AGGREGATE, ts2
        )
    ncptl_log_commit_data(logfile)
    ncptl_log_write_footer(logfile)
    ncptl_log_close(logfile)
```

Line for line, the Python version is fairly similar to the C version; an application developer unfamiliar with Python should have no trouble understanding the code. However, we can say that the Python version remedies shortcoming 2 on page 152 by being shorter (by ∼21%) and arguably less error-prone than the C version. Nevertheless, the level of abstraction is unchanged from the C version; the connection between the problem statement and the code remains unclear. In Sect. 4 we determine if a domain-specific language can improve the situation.

4 A Domain-Specific Language for Performance Testing

Application developers – and programmers in general – are often loath to use domain-specific languages. There is an inherent cost to learning a new language which can be hard to justify in absense of *a priori* understanding of the domain-specific language's practical benefits. It is therefore important for this paper to evaluate how well a domain-specific language can be suited to the rapid development of application-specific network performance tests.

coNCePTuaL – whose unusual capitalization stands for "Network Correctness and Performance Testing Language" – is a domain-specific language designed to facilitate the rapid development of special-purpose network performance tests [9]. The coNCePTuaL language is English-like and uses SPMD semantics to express parallelism. coNCePTuaL programs implicitly use the performance library described in Sect. 3 through various language idioms. The following is the complete, commented coNCePTuaL solution to the problem specified in Sect. 2:

```
1  Require language version ”0.5.2b”.
2
3  bigsize is ”Size of big message (doubles)” and comes from ”−−bigsize” or ”−b” with
       default 10000.
4  litsize is ”Size of small message (doubles)” and comes from ”−−litsize” or ”−n” with
       default 1.
5
6  Assert that ”this program must be run with at least 3 processes” with num_tasks>=3.
7
8  All tasks touch all message buffers then
9  all tasks synchronize then
10
11 # ”Arrange the code so that process 1 has already posted an MPI_Irecv
12 # for the message from process 2 before receiving the message from
13 # process 0,”
14 task 1 asynchronously receives a litsize doubleword message from task 2 then
15
16 # ”Process 0 is sending a long message to process 1”
17 task 0 sends a bigsize doubleword message to task 1 then
18
19 # ”also ensure that process 1 receives the long message from process 0
20 # before receiving the message from process 2.”
21 task 1 awaits completion then
22
23 # ”process 2 is sending a relatively short message to process 1 and
24 # then to process 0.”
25 task 2 resets its counters then
26 task 2 sends a litsize doubleword message to unsuspecting task 1 then
27 task 2 logs litsize as ”Litsize” and elapsed_usecs as ”Time for first send” then
28 task 2 resets its counters then
29 task 2 sends a litsize doubleword message to task 0 then
30 task 2 logs elapsed_usecs as ”Time for second send”.
```

Two things are immediately apparent about the preceding code listing. First, its length is half that of even the corresponding Python code. Second, coNCePTuaL statements closely correspond to the natural-language statements in the problem specification. For example, posting a send implicitly posts the matching receive (suppressable with the **unsuspecting** keyword, as in line 26), as this is how network performance tests typically are described textually. By raising the abstraction level of performance tests to match that of a natural-language problem specification, coNCePTuaL satisfactorily addresses shortcoming 3 on page 152, the final shortcoming of the original C solution. There is no performance penalty to using coNCePTuaL, as indicated by the following message overheads output by the various code versions: C: (5µs, 2µs); C + library: (4µs, 1µs); Python: (10µs, 5µs); coNCePTuaL: (4µs, 2µs). (coNCePTuaL reports slightly lower overheads than C because it directly reads the CPU cycle counter; when configured to use `gettimeofday()` it reports the same overheads as C.)

coNCePTuaL is not limited to MPI. Any of a variety of code generators is selectable at compile time. In fact, Fig. 1 was produced automatically from the preceding listing merely by specifying an “illustration” code generator.

5 Conclusions

Special-purpose network performance tests enable an application developer to experiment with communication alternatives faster and with less hassle that would be required to restructure the original application. Such tests are rarely used in practice, however, because of the their development overhead. This paper followed the evolution of a sample special-purpose performance test from its natural-language problem specification through C, C + helper library, Python + helper library, and domain-specific-language solutions. Each alternative improved upon the previous one: richer output; shorter, easier-to-develop programs; and, higher levels of abstraction. However, only the domain-specific language, coNCePTuaL, fully encapsulated all three of those benefits.

This paper elucidated the tradeoffs that application developers face when considering using special-purpose network performance tests as a tool for performance optimization. Developers can reduce development time with little effort merely by calling appropriate library functions. Developers who couple a library with a high-level language can further reduce development time, making special-purpose network performance tests more attractive to write. The biggest payoff comes from using a domain-specific language, which provides a natural mapping from problem specification to working code while still retaining the benefits of short code lengths, ease of development, and reproducibility of results.

The coNCePTuaL compiler and performance library are freely available from `http://conceptual.sourceforge.net/`.

References

1. Bleck, R.: An oceanic general circulation model framed in hybrid isopycnic-Cartesian coordinates. Ocean Modelling **4** (2002) 55–88
2. Blelloch, G., Narlikar, G.: A practical comparison of N-body algorithms. In: Parallel Algorithms. Volume 30 of DIMACS: Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, DIMACS (1997)
3. Basney, J., Raman, R., Livny, M.: High throughput Monte Carlo. In: Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing, San Antonio, Texas (1999)
4. Snell, Q.O., Mikler, A.R., Gustafson, J.L.: NetPIPE: A network protocol independent performance evaluator. In: Proceedings of the 1996 ISMM International Conference on Intelligent Information Management Systems, Washington, DC, ACTA Press (1996)
5. Gropp, W., Lusk, E.: Reproducible measurements of MPI performance characteristics. In: Proceedings of the 6th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'99). Volume 1697 of Lecture Notes in Computer Science., Barcelona, Spain, Springer-Verlag (1999) 11–18
6. Pallas, GmbH: Pallas MPI Benchmarks—PMB, Part MPI-1. (2000)

7. Reussner, R., Sanders, P., Prechelt, L., Müller, M.: SKaMPI: A detailed, accurate MPI benchmark. In: 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) 52–59
8. Gropp, W.: Tutorial on MPI: The Message-Passing Interface. Argonne National Laboratory, Argonne, Illinois. (1995) Available from `ftp://info.mcs.anl.gov/pub/mpi/tutorial.ps`.
9. Pakin, S.: Reproducible network benchmarks with coNCePTuaL. In: Proceedings of the 10th International Euro-Par Conference. Volume 3149 of Lecture Notes in Computer Science., Pisa, Italy, Springer (2004) 64–71 ISBN 3-540-22924-8. Available from `http://www.c3.lanl.gov/~pakin/papers/europar2004.pdf`.
10. Pakin, S.: coNCePTuaL user's guide. Technical Report LA-UR 03-7356, Los Alamos National Laboratory, Los Alamos, New Mexico (2004) Available from `http://www.c3.lanl.gov/~pakin/software/conceptual/conceptual.pdf`.
11. Hinsen, K.: ScientificPython User's Guide. Centre National de la Recherche Scientifique d'Orléans, Orléans, France. (2002)