Overview

MPI (Message Passing Interface) is the de facto standard for distributed memory programming. However, the tool support for MPI applications’ development and runtime is far from enough:

1) No practical systematic testing techniques for bug detection.
2) No sufficient runtime support for scaling problems – a class of bugs manifesting at large scale either in terms of problem size or the number of processes.

Figure 1. Overview of our work: (1) automated testing for general bug detection, and (2) runtime support for hard scaling problems that escapes the testing.

Automated Testing with COMPI


Concolic testing automates the testing of a target MPI program by automatically generating inputs. COMPI extends it in following ways:

1) It tackles basic MPI semantics.
2) It controls the testing time.

With up to 3.5 hours, COMPI justifies itself as a practical testing tool:

1) 69-80% branch coverage for complex programs including HPL, IMB, and SUSY-HMC (a physics simulation program).
2) 4 new bugs in SUSY-HMC.
3) 4-81% more branch coverage than standard concolic testing.
4) 46-67% more coverage than random testing.

void main() {
    int x, y, n, r; // input: (x, y, n, r)
    COMPI_int(x);
    COMPI_int(y);
    COMPI_int(n);
    COMPI_int(r);
    MPI_Init();
    MPI_Comm_size(MPI_COMM_WORLD, &n);
    MPI_Comm_rank(MPI_COMM_WORLD, &r);
    B1: if (x == 100) func1();
    B2: else func2();
    B3: if (n < 10) func3();
    B4: else func4();
    B5: if (r == 0) func5();
    B6: else {
        B7: if (y != 123) func6();
        B8: else ABORT_on_Bug();
    }
}

3 decision points, or 6 branches
Inputs: (x, y, rank, nprocs)

1) Execute the program
2) Negate a constraint & Solve

Input: (100, 200, 4, 0)
1
x = 100, n < 10, r = 0
(62.5%): B1, B3, B5, B6, B7
2 w/ x ≠ 100
Input: (101, 200, 4, 3)
1
x ≠ 100, n < 10, r = 0
(75%): B1, B2, B3, B5, B6, B7
2 w/ n ≥ 10
Input: (101, 246, 12, 0)
1
x ≠ 100, y ≠ 123, n ≥ 10, r = 0
(87.5%): B1, B2, B3, B4, B5, B6, B7
2 w/ r ≠ 0

Figure 1. An MPI program.
Figure 2. COMPI drives the testing.

Avoiding Scaling Problems for MPI Collectives


Scaling problems frequently occur with the use of MPI collectives. It is challenging to fix them. We provide an avoidance framework as an immediate remedy. Its core idea consists of two parts:

1) Find the trigger point of a scaling problem via testing.
2) Bypass the bug via partitioning the communication.

Figure 3. The process of root filling its receiving buffer recvbuf in one MPI_Gatherv communication partitioned by two strategies, supposing the bug occurs when the root receives more than 4 elements at a time.

Hang Detection at Large Scale


On supercomputers, the execution of one batch job costs (#cores) * allocated time.

Hangs, once occurring, cause a great wastage of computing resources as it stalls the program execution until the allocated time expires, e.g., thousands of cores within the allocated time can be wasted in large scale runs. Hence, it is urgent to devise a tool to detect hangs at runtime and terminates hanging job to avoid the wastage.

Key Insight: a program hang occurs when a majority of processes stay inside MPI communication for a long time.

Based on the insight, ParaStack detect hangs based on a statistical model maintained at runtime with following highlights:

1) It avoids the complexity of timeout setting.
2) It detects hangs with high accuracy, in a timely manner, with negligible overhead, and with low false positive rate, and thus enables great saving for hanging jobs.
3) It is adapted to work with two most popular batch job schedulers Torque and Slurm.

What is majority? 90% or 100%? How long? 1 min or 1 hour?

Guess one!
Statistics decides!

Figure 4. ParaStack v.s. Timeout.