CompSci 491 Midterm Exam
Answer key - 2020-10-04

Question 1:

Table 2.4, 3 problem sizes –

1 million:

1e6	"working time"			total time		
np	**1**	**4**	**12**	**1**	**4**	**12**
S	1	84/26 = 3.231	84/44 = 1.909	1	1.201	1.072
E = S / np	1	3..23/4 = 0.808	1.9/12 = 0.159	1	0.808	0.089

100 million:

100e6	"working time"			total time		
np	**1**	**4**	**12**	**1**	**4**	**12**
S	1	4.000	9.050	1	3.786	7.830
E = S / np	1	1.000	0.754	1	1.000	0.653

10 billion:

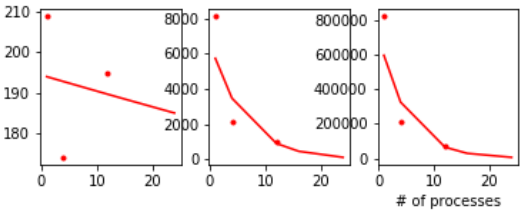
10e9	"working time"			total time		
np	**1**	**4**	**12**	**1**	**4**	**12**
S	1	3.947	11.317	1	3.947	11.294
E = S / np	1	0.987	0.943	1	0.987	0.941

Table 2.5, 3 problem sizes –

Problem size		"working time"			Total time		
	np	**1**	**4**	**12**	**1**	**4**	**12**
1e6	**S**	1	3.231	1.909	1	1.201	1.072
	E = S / np	1	0.808	0.159	1	0.808	0.089
100e6	**S**	1	4.000	9.050	1	3.786	7.830
	E = S / np	1	1.000	0.754	1	1.000	0.653
10e9	**S**	1	3.947	11.317	1	3.947	11.294
	E = S / np	1	0.987	0.941	1	0.987	0.941

Total-running time extrapolations –

np	1	4	12	***16***	***24***
1e6	209	174	205	***201***	***206***
100e6	8120	2145	1037	***466***	***122***
10e9	825,120	209,064	73,059	***28,848***	***5730***



Estimated running times are based on a linear regression through the logarithms of measured running times.
Anomalous running times for 1e6 problem size can be explained by the changeover from internal-bus interconnect for np=4, to Ethernet interconnect for np=12.

Scalability –

For large-enough problem size (10e9) the program appears to be strongly scalable.
At middle-to-large problem sizes (100e6 – 10e9) the program is roughly weakly scalable.

Question 2:

7-D hypercube –

2 ** 7 == 128 processors

7 switch-switch wires, plus 1 switch-processor wire, per switch ==> 8 *128 = 1024 wires
Bisection width = 128 / 2 = 64

Direct interconnect.

Omega interconnect –

p =128
2 * p * log2(p) = 1792
bisection width = p / 2 = 64

Crossbar interconnect –

p = 128
p ** 2 = 16384
bisection width = p = 128

Maximum distance between processors –

diagonal corner of 7-D hypercube = 7
(longest closed path on 7-D hypercube = 2**7 = 128)

Question 3:

Flynn's taxonomy –

Flynn's taxonomy describes ways of associating data with processing instructions.

- SISD – Single Instruction, Single Data – any pure Von Neumann machine, Turing machine, etc.
- SIMD – Single Instruction, Multiple Data – vector processors, GPUs SSE/AVX instructions in Intel/AMD architecture, Neon instructions in ARMv8 architecture, etc.
- MIMD – Multiple Instruction, Multiple Data – multprocessor designs, e.g. clusters
- MISD – Multiple Instruction, Single Data – (redundant designs, viz. Space Shuttle computers)
- SPMD – Single Program, Multiple Data – OpenMPI, OpenMP/Pthreads, clusters

Question 4:

OpenMPI memory –

Designed for distributed-memory systems such as clusters, although it also works on shared-memory multicore cluster systems. Example: message-passing programs only communicate through message buffers, no shared-memory accesses.

OpenMPI can be combined with Pthreads/OpenMP for true shared-memory accesses.

Question 5:

Cache coherence –

Snooping cache coherence depends on sharing/broadcasting cache transactions. Best used on UMA shared-memory systems.

Directory-based cache coherence involves a (usually) distributed directory of cache transactions, which can be consulted by all involved cores. Scales better than snooping for larger, NUMA shared-memory systems.

Cache coherence isn't really an issue for distributed-memory systems, as the processor caches do not map to the same memory addresses.

Question 7:

`significant_digits()`: root only.
`random_coordinate()`, `initstate_r()`: used by all computing processes.

Sub-linear accuracy improvement: increase in terms by 10x gives < 10x accuracy improvement.

Linear nperformance improvement: # of terms divided evenly among # of processors.

Hailstone w/ OpenMPI

```c
/* the hailstone sequence / Collatz conjecture
|  2020-09-26
*/
#include <stdio.h>
#include <stdlib.h>      // random()/srandom(), atol()
#include <time.h>        // time()
#include <limits.h>      // UINT_MAX
#include <math.h>        // sqrt()
#include <mpi.h>

#define ROOT 0
#define MSG_TAG 0

long unsigned next_hailstone(long unsigned n)
{
    return (n % 2) ? 3 * n + 1 : n / 2;
}

long unsigned sequence(long unsigned n)
{
    long unsigned length = 0;
    while (n != 1) {
        length++;
        n = next_hailstone(n);
        //printf("next n: %lu\n", n);
    }
    return length;
}

int main(int argc, char **argv)
{
    int myrank = 0, worldsize = 1;

    srandom(time(NULL));

    char myhost[MPI_MAX_PROCESSOR_NAME] = "none";
    int myhostlen;

   MPI_Init(&argc, &argv);                        /* Initialize MPI execution env. */
    MPI_Comm_size(MPI_COMM_WORLD, &worldsize);    /* # of processes in MPI_COMM_WORLD */
    MPI_Get_processor_name(myhost, &myhostlen); /* Discover process' hostname */
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);       /* Give each process a unique rank */
    MPI_Status status;                            /* status for MPI_Recv */
    //printf("host %s, process %d\n", myhost, myrank);

    long unsigned starts[worldsize];
    long unsigned lengths[worldsize];
    long unsigned sum, sumsq;


    /* If the process is the root */
    if ( myrank == ROOT ) {
        for (int rank = ROOT; rank < worldsize; rank++) {
            if (argc > rank+1)
                starts[rank] = atol(argv[rank+1]);
            else
                starts[rank] = random();
            //printf("rank %d: start %lu\n", rank, starts[rank]);
        }
        fflush(stdout);

        for (int rank = ROOT+1; rank < worldsize; rank++)
            MPI_Send(starts+rank, 1, MPI_UNSIGNED_LONG, rank, MSG_TAG, MPI_COMM_WORL

        lengths[ROOT] = sequence(starts[ROOT]);
        sum = lengths[ROOT];
        sumsq = sum * sum;

        printf("root/%s reports start=%lu, length=%lu\n",
            myhost, starts[ROOT], lengths[ROOT]);

        for (int count = 1; count < worldsize; count++) {
            long unsigned length;
            MPI_Recv(&length, 1, MPI_UNSIGNED_LONG, MPI_ANY_SOURCE,
                MSG_TAG, MPI_COMM_WORLD, &status);
            int rank = status.MPI_SOURCE;
            lengths[rank] = length;
            sum += length;
            sumsq += length * length;

            printf("%2d:  process %d reports start=%lu, length=%lu\n",
                count, rank, starts[rank], lengths[rank]);
            fflush(stdout);
        }

        double mean = (double)sum / (double)worldsize;
        double stdev = sqrt(worldsize * sumsq - sum * sum) / (double)worldsize;
        printf("\n%d samples: mean %.3lf, s.d. %.3lf\n", worldsize, mean, stdev);

    } else {
        long unsigned start;
        MPI_Recv(&start, 1, MPI_UNSIGNED_LONG, ROOT, MSG_TAG,
            MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        long unsigned length = sequence(start);

        MPI_Send(&length, 1, MPI_UNSIGNED_LONG, ROOT, MSG_TAG, MPI_COMM_WORLD);
    }

    MPI_Finalize();

    return 0;
}
```

Pi-darts w/ OpenMPI

```c
/*
* Area-based approximation of pi
*/
#include <stdio.h>
#include <stdlib.h> // strtoul(), random_r()
#include <time.h>   // time()
#include <math.h>   // sqrt(),  # define M_PI 3.14159265358979323846
#include <mpi.h>

#define ROOT 0

// long double only suppports 35 digits
#define PI35 3.14159265358979323846264338327950288L
//      -.---|----|----|----|----|----|----|

/*
| Use a fancy pseudo-random number generator that is "thread-safe"
| meaning it can be called by multiple processes and they all get
| independent values.
*/
#define PRNG_BUFSIZE 8  // used in randome-number generation

// Generate a random value between -1 and +1:
double random_coordinate(struct random_data *rand_state)
{
    int rn;
    random_r(rand_state, &rn);
    return (-1.0 + 2.0 * (double)rn/(double)RAND_MAX);
}
//---------------------------------------------------------------------

// Calculate the number of pi's digits that are "correct" (and add 3 extra).
int significant_digits(long double pi)
{
    long double diff = fabsl(PI35 - pi);
    long double logdiff = -log10(diff);
    return logdiff + 2;
}
//---------------------------------------------------------------------


int main(int argc, char **argv)
{
    int worldsize, myrank;
    int namelen;
    char hostname[MPI_MAX_PROCESSOR_NAME];
    double timing[4];

    timing[0] = MPI_Wtime();

    MPI_Init(&argc, &argv); // args not needed, but supplied anyway.
    MPI_Get_processor_name(hostname, &namelen);
    MPI_Comm_size(MPI_COMM_WORLD, &worldsize);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    long unsigned global_ndarts;
    long unsigned ndarts;
    if (ROOT == myrank) {
        global_ndarts = (long unsigned)( (argc >= 2) ? atof(argv[1]) : 1e6 ) ;
        ndarts = 1 + (float)global_ndarts / (float)worldsize;  // round up...
        global_ndarts = ndarts * worldsize;
    }

    timing[1] = MPI_Wtime();

    MPI_Bcast(&ndarts, 1, MPI_UNSIGNED_LONG, ROOT, MPI_COMM_WORLD);
    //printf("%d/%s gets %lu\n", myrank, hostname, ndarts);

    /*
  | Each process creates a separate random-number generator:
  */
    char rand_state_buf[PRNG_BUFSIZE];
    struct random_data rand_state;
    srandom(time(NULL) + myrank);   // different seed for each process
    initstate_r(random(), rand_state_buf, PRNG_BUFSIZE, &rand_state);

    // Do the work:
    long unsigned incircle = 0;
    for (long unsigned i = 0; i < ndarts; i++) {
        double x = random_coordinate(&rand_state);
        double y = random_coordinate(&rand_state);
        if (1.0 >= sqrt(x*x + y*y))
            incircle++;
    }
    //printf("%d/%s local darts: %lu, in circle: %lu\n",
    //   myrank,hostname, ndarts, incircle);
    //fflush(stdout);

    long unsigned global_incircle = 0;  // only used by root
    MPI_Reduce(
        &incircle, &global_incircle, 1, MPI_UNSIGNED_LONG,
        MPI_SUM, ROOT, MPI_COMM_WORLD);

    timing[2] = MPI_Wtime();

    // Only root does the final calculation and output:
    if (ROOT == myrank) {
        printf("total darts: %lu, in circle: %lu\n", global_ndarts, global_incircle);

        long double pi = 4.0L *
            (long double)global_incircle / (long double)global_ndarts;

        int signif = significant_digits(pi);
        printf("%d significant digits\n", signif);
        printf("approx.: %.*Lf\n    C    : %.20Lf\n manual: %.20Lf\n",
            signif, pi, (long double)M_PI, (long double)PI35);
    }

    MPI_Finalize();

    timing[3] = MPI_Wtime();
    printf("%d/%s    setup time: %lf ms\n",
        myrank, hostname, 1e3 * (timing[1] - timing[0]));
    printf("%d/%s working time: %lf ms\n",
        myrank, hostname, 1e3 * (timing[2] - timing[1]));
    printf("%d/%s cleanup time: %lf ms\n",
        myrank, hostname, 1e3 * (timing[3] - timing[2]));
    printf("%d/%s    Total time: %lf ms\n",
        myrank, hostname, 1e3 * (timing[3] - timing[0]));

    return 0;
}
//---------------------------------------------------------------------
```