Understanding the NAS "Embarrassingly Parallel" Benchmark

Derrell Lipman
University of Massachusetts, Lowell
dlipman@cs.uml.edu

1 Introduction

Getting started understanding and implementing the NAS [1] benchmarks can be a difficult task for the uninitiated. The simplest of the benchmarks is Embarrassingly Parallel (EP). This paper will provide explanations of key portions of the code required to implement EP, and will present complete, documented code to implement a serial version of EP in C, and parallel versions using both C with MPI [2], and ZPL 2.0 [3].

Since the primary purpose of this paper is to assist the reader in better understanding the NAS EP benchmark, I will not be overly concerned with efficiency or lack thereof of the presented code. I acknowledge that there are more efficient ways to implement portions of this code (some of which will be mentioned), but strongly value readability over efficiency here.

This document and the complete source code, portions of which are described herein, are available at http://www.cs.uml.edu:~dlipman/understanding-nas-ep.

2 Generation of Pseudo-random Numbers

A specific sequence of pseudo-random numbers is required for EP (and other NAS benchmarks). The algorithm for generation of these numbers is provided in section 2.3 of [1]. There are three unique components of the algorithm: modulo-46 multiplication, pseudo-random number generator, and specific seed calculation.

2.1 Modulo-46 Multiplication

The algorithm for modulo-46 multiplication is provided in pseudo-code form in the benchmark document, as follows:

To compute $c = ab \pmod{2^{46}}$, perform the following steps:

$$
\begin{array}{lcl}
a_1 & \leftarrow & \text{int } (2^{-23}a) \\
a_2 & \leftarrow & a - 2^{23}a_1 \\
b_1 & \leftarrow & \text{int } (2^{-23}b) \\
b_2 & \leftarrow & b - 2^{23}b_1 \\
t_1 & \leftarrow & a_1b_2 + a_2b_1 \\
t_2 & \leftarrow & \text{int } (2^{-23}t_1) \\
t_3 & \leftarrow & t_1 - 2^{23}t_2 \\
t_4 & \leftarrow & 2^{23}t_3 + a_2b_2 \\
t_5 & \leftarrow & \text{int } (2^{-46}t_4) \\
c & \leftarrow & t_4 - 2^{46}t_5
\end{array}
$$

For an initial, naive implementation, this can be mapped directly to a C function as follows:

```
double
multiplyModulo46(double a, double b)
{
    double  a1 = floor(pow(2, -23) * a);
    double  a2 = a - pow(2, 23) * a1;
    double  b1 = floor(pow(2, -23) * b);
    double  b2 = b - pow(2, 23) * b1;
    double  t1 = (a1 * b2) + (a2 * b1);
    double  t2 = floor(pow(2, -23) * t1);
    double  t3 = t1 - (pow(2, 23) * t2);
    double  t4 = (pow(2, 23) * t3) + (a2 * b2);
    double  t5 = floor(pow(2, -46) * t4);

    return t4 - (pow(2, 46) * t5);
}
```

Since this function will be called very frequently, we may want to optimize it somewhat. In the naive implementation, we have a number of constants which are frequently re-calculated using the expensive `pow()` function. Those values can be calculated once and reused. A somewhat more efficient implementation might then look like this:

```
static double       POW_2_N23 = 0.0;
static double       POW_2_23  = 0.0;
static double       POW_2_N46 = 0.0;
static double       POW_2_46  = 0.0;

double
multiplyModulo46(double a, double b)
{
    static int          bInitialized = FALSE;
    double              a1;
    double              a2;
    double              b1;
    double              b2;
    double              t1;
    double              t2;
    double              t3;
    double              t4;
    double              t5;

    if (! bInitialized)
    {
        POW_2_N23 = pow(2, -23);
        POW_2_23  = pow(2,  23);
        POW_2_N46 = pow(2, -46);
        POW_2_46  = pow(2,  46);

        bInitialized = TRUE;
    }
```

```
    a1 = floor(POW_2_N23 * a);
    a2 = a - POW_2_23 * a1;
    b1 = floor(POW_2_N23 * b);
    b2 = b - POW_2_23 * b1;
    t1 = (a1 * b2) + (a2 * b1);
    t2 = floor(POW_2_N23 * t1);
    t3 = t1 - (POW_2_23 * t2);
    t4 = (POW_2_23 * t3) + (a2 * b2);
    t5 = floor(POW_2_N46 * t4);

    return t4 - (POW_2_46 * t5);
}
```

2.2 Pseudo-random Number Generator

Generation of each random number is accomplished in conjunction with generating the seed for the subsequent random number. The formula uses a constant, *a,* which has the value 5^{13}. Each next seed is generated by multiplying the prior seed by the constant *a.* That prior seed is also used to generate the current random number by multiplying that prior seed by 2^{-46}. The initial seed is a constant that is specified in the description of each benchmark requiring these pseudo-random numbers. For the EP benchmark, the constant value is 271,828.183, but since the pseudo-random number generator is used by multiple benchmarks, I will refer to that constant herein as `INITIAL_SEED`.

Assuming that the global variable `random_seed` has been initialized, either to the constant `INITIAL_SEED` or via prior calls to this `pseudorandom()` function, a pseudo-random number can be generated as follows. This function generates the next seed before returning the current random number.

```
double
pseudorandom(void)
{
    double   oldS;

    /* Save the old seed as it's used to calculate the return value */
    oldS = random_seed;

    /* Calculate the next seed */
    random_seed = multiplyModulo46(a, random_seed);

    /* Calculate the return value based on the old seed */
    return POW_2_N46 * oldS;
}
```

Note that if the seed has been initialized to the initial seed constant, the current (first) pseudo-random number to be generated will be random number 0, not random number 1. The EP requirements are to begin with random number 1.

2.3 Specific Seed Calculation

For parallel operation, it is desirable for each process to generate its own keys. The random number function used by these benchmarks allows easily retrieving the seed for the k'th pseudo-random number. The pseudo-code and additional required information from the benchmark text is equivalent to:

```
Let m be the smallest integer such that 2^m > k
set b = INITIAL_SEED
set t = a
for i = 1 to m do
      j     ←    k/2
      b     ←    bt (mod 2^46) if 2j <> k
      t     ←    t^2 (mod 2^46)
      k     ←    j
end
```

The “smallest integer such that $2^m > k$” really means floor($\log_2 k$). Therefore, retrieving the seed for the k'th pseudo-random number can be accomplished by the C function shown here:

```
double
getSeedFor(double k)
{
    int             i;
    int             j;
    double          b;
    double          t;
    int             m;

    m = floor(log2(k)) + 1;
    b = INITIAL_SEED;
    t = a;
    for (i = 1; i <= m; i++)
    {
        j = k / 2;
        if (2 * j != k)
        {
            b = multiplyModulo46(b, t);
        }
        t = multiplyModulo46(t, t);
        k = j;
    }

    return b;
}
```

This function is called in the C with MPI and the ZPL implementations, but not in the serial implementation.

3 Embarrassingly Parallel

With the difficult concept of how to generate random numbers out of the way, we can move on to the requirements of the EP benchmark itself. We will be generating $n = 2^{28}$ (for Class A) random number pairs. As previously discussed, the initial seed value, `INITIAL_SEED`, is 271,828,183. Each pair of random numbers contains an *x* value and a *y* value, each of which is in the range of (0, 1). A pair of numbers may or may not be acceptable to use. It is acceptable if the sum of x^2 and y^2 (referred to hereafter as *t)*, is less than or equal to 1, i.e., accept the pair *j* if $(t_j = x_j^2 + y_j^2) \leqslant 1$; otherwise discard the pair. Because some pairs will be found to be not acceptable, we will have fewer than $n = 2^{28}$ pairs used in the operations to be described next.

Once we have an acceptable pair, we obtain a temporary multiplier for the *x* and *y* values with the following formula using log base e:

```
temp = sqrt((-2 * log(t)) / t);
```

The *x* and *y* values are multiplied by this temporary value, and running sums of the *x* values and *y* values are kept for later result validation. These running sums will be referred to as `sumX` and `sumY`.

We now need to select the larger of the *x* and *y* values. We refer to this larger one as `maxXY` and use it to determine which one of 10 counters is to incremented. The counter are used to track how many values are in the various ranges [0, 1), [1, 2), [2, 3), …, [9, 10). The counters are called `results`, and and are stored as a 10-element array. By truncating `maxXY` to an integer, a single element of the array can be incremented:

```
++results[maxXY];
```

Once all $n = 2^{28}$ pairs have been tested for acceptability and summed and counted as just described, the values of the two sums and 10 counters are displayed. This benchmark requires that the time for the entire process described above, including tabulation of the results, be included in the running time. Therefore the time taken to run the entire algorithm is calculated after displaying the results.

4 Parallelizing EP

So far, all functions we've discussed apply to a serial implementation of EP. Converting from a serial implementation to a multi-processing one is fairly easy. The benchmark is called “embarrassingly parallel” because there is very little communication required between the processes. The only required communication is accumulating the counts and sums generated by each process, to ensure that the totals match the benchmark requirements.

To allow each process to work entirely on its own, we make use of the `getSeedFor()` function. Each process knows how many processes there are, and knows its own rank (process number), so it can divide the total number *n* by the number of processes to discover how many pairs of random numbers it should generate; and it can use that calculated number multiplied by the process' rank to determine the index of the first pair of random numbers it should use.

4.1 Changes for MPI

The initial seed for any process can be determined like so:

```
/* Find out process rank */
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

/* Find out number of processes */
MPI_Comm_size(MPI_COMM_WORLD, &p);

/*
 * Get the seed for this process. Each iteration takes two random
 * numbers, so we need to multiply the per-process count by 2. Also
 * add 1, to account for the first random number being generated after
 * calculating the first seed from the initial seed value.
 */
random_seed = getSeedFor(my_rank * (n / p) * 2 + 1);
```

We can then run the EP algorithm as before, independently in each process. After calculating counts and sums in each process, every process other than process 0 sends its counts and sums to process 0 which accumulates them. (Process 0 can then also display the counts, sums, and running time.)

```
/* Process 0 receives sums and counts from all other processes. */
if (my_rank == 0)
{
    /* From each other process... */
    for (i = 1; i < p; i++)
    {
        /* ... retrieve his results */
        MPI_Recv(hisResults, 10, MPI_INT, i, 0,
                 MPI_COMM_WORLD, &status);

        /* Accumulate his results with our own */
        for (j = 0; j < 10; j++)
        {
            results[j] += hisResults[j];
        }

        /*
         * Also get his sumX and sumY, and accumulate them with
         * our own
         */
        MPI_Recv(&hisSumX, 1, MPI_DOUBLE, i, 0,
                 MPI_COMM_WORLD, &status);
        sumX += hisSumX;

        MPI_Recv(&hisSumY, 1, MPI_DOUBLE, i, 0,
                 MPI_COMM_WORLD, &status);
        sumY += hisSumY;
    }
}
else
```

```
    {
        /* Send my results to process 0 */
        MPI_Send(results, 10, MPI_INT, 0, 0, MPI_COMM_WORLD);
        MPI_Send(&sumX, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
        MPI_Send(&sumY, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    }
```

4.2 A ZPL implementation of EP

It is fairly straight forward to implement the EP benchmark in ZPL 2.0, by following a similar structure to what we developed for C with MPI. By making use of *free* procedures and variables, we allow each process to operate independently, with its own data, and reduce the results to a parallel variable at the end in order to tabulate the data.

First, the constants are declared globally for single-time initialization:

```
INITIAL_SEED : double = 271828183;
POW_2_N23    : double = pow(2, -23);
POW_2_23     : double = pow(2,  23);
POW_2_N46    : double = pow(2, -46);
POW_2_46     : double = pow(2,  46);
```

We determine the number of processes and the rank.

```
p      : integer = numLocales(); -- number of processes
myRank : integer = localeID();   -- which process am I?
```

Results are manipulated in a user-defined record type that includes an indexed array to hold the ten counts, and an indexed array to hold the two sums.

The functions to multiply modulo 46, retrieve a random number, get the seed for a particular index of pseudo-random number, and get the next pair of random numbers are structurally the same as their equivalent C versions. One minor different, though, is that although ZPL claims to have a log2() function, it appears not to, so we implement one based on the natural logarithm.

```
free procedure
multiplyModulo46(free in a : double; free in b : double) : double;

free var
  a1 : double = floor(POW_2_N23 * a);
  a2 : double = a - POW_2_23 * a1;
  b1 : double = floor(POW_2_N23 * b);
  b2 : double = b - POW_2_23 * b1;
  t1 : double = (a1 * b2) + (a2 * b1);
  t2 : double = floor(POW_2_N23 * t1);
  t3 : double = t1 - (POW_2_23 * t2);
  t4 : double = (2^23 * t3) + (a2 * b2);
  t5 : double = floor(POW_2_N46 * t4);

begin
  return t4 - (POW_2_46 * t5);
end;
```

```
free procedure
log2(free in a : double) : integer;

free var
  i : integer;

begin
  i := floor(log(a) / log(2));
  return i;
end;


free procedure
getSeedFor(free in k : double) : double;

free var
  i : integer;
  j : integer;
  m : integer;
  b : double;
  t : double;

begin
  m := log2(k) + 1;
  b := INITIAL_SEED;
  t := MULT;

  for i := 1 to m
  do
    j := k / 2;
    if (2 * j != k) then
      b := multiplyModulo46(b, t);
    end;
    t := multiplyModulo46(t, t);
    k := j;
  end;

  return b;
end;


free procedure
randomPair() : Pair;

free var
  pair : Pair;

begin
  pair.x := 2 * pseudorandom() - 1;
  pair.y := 2 * pseudorandom() - 1;

  return pair;
end;
```

The heavy lifting of the EP benchmark is accomplished in the `ep_worker()` function, which again has a structure similar to its C counterpart, `ep()`:

```
procedure ep_worker() : Results;

free var
  i        : integer;
  maxXY    : integer;
  pair     : Pair;
  t        : double;
  temp     : double;

  -- Local results: array of counts, and array of sums (of X and Y values).
  myResults: Results;

var
  -- Final results will be accumulated from all processes, here.
  results  : Results;

begin
  -- Initialize local results
  for i := 0 to 9
  do
    myResults.counts[i] := 0;
  end;
  myResults.sums[0] := 0.0;
  myResults.sums[1] := 0.0;

  -- For each random pair that this process is supposed to generate...
  for i := 1 to n / p
  do
    -- Generate a pair
    pair := randomPair();

    -- Is this pair acceptable?
    t := pair.x^2 + pair.y^2;
    if (t > 1.0) then
      -- Nope. Reject it.
      continue;
    end;

    -- Munge the pair as required by NAS EP
    temp := sqrt(-2 * log(t) / t);
    pair.x *= temp;
    pair.y *= temp;

    -- The sum of X values is placed in [0]; Y values in [1]
    myResults.sums[0] += pair.x;
    myResults.sums[1] += pair.y;

    -- We're told to use the maximum of absolute value of the two values
    -- in the pair. Find that maximum, truncated to an integer.
    maxXY := maximum(fabs(pair.x), fabs(pair.y));
```

```
    -- Increment the appropriate count depending on the selected value
    myResults.counts[maxXY] += 1;
  end;

  -- Now that we have counts and sums in each process (in free variables),
  -- reduce those counts and sums into a parallel variable containing an
  -- index array so we have a global set of counts and sums.
  results.counts[] := +<< myResults.counts[];
  results.sums[] := +<< myResults.sums[];

  -- Give 'em our results!
  return results;
end;
```

The last few lines require some additional explanation. The entire loop up through the point where the *myResults.counts* array, indexed by *maxXY,* is incremented, operates on variables which are entirely local to a process. The variables are all *free*. During that loop, each process is calculating its own pairs of random numbers, obtaining the counts and sums just as the C versions did.

Once each process has completed counting and summing, it is necessary to combine the local results from each process into a single set of results that can be compared to the required values of the benchmark. To accomplish this, we use the *reduce sum* operator. What is of note here is that *myResults* is a free variable, whereas *results* is a parallel variable. The reduce operation then takes the sum of all process' *myResults.counts[0]* and places that sum into *results.counts[0]*. Similarly, the sum of all of the *myResults.counts[1]* is placed into *results.counts[1]*, and so forth for each of the 10 indexes. An equivalent operation is performed to reduce the two *sums* values from each of the processes into single sums of all of the local sums.

5 Performance Comparisons

For comparison, EP benchmark Class A times were determined, using each of the three implementations. All tests were run on a mixed cluster of 16 machines, mostly AMD Phenom X4, 2.3 GHz, with 2 GB of RAM and four cores each. The operating system was Debian Linux, Linux kernel version 2.6.26.

Implementation	Number of Processes	Benchmark Time (seconds)		
		Trial 1	Trial 2	Trial 3
C – serial	1	101.4	102.4	103.0
C with MPI	1	102.2	100.5	102.0
ZPL	1	100.9	103.4	100.8
C with MPI	2	51.1	51.4	51.3
ZPL	2	50.8	50.4	50.5
C with MPI	4	36.5	35.5	35.3
ZPL	4	31.5	35.4	33.4
C with MPI	8	17.8	17.3	17.7
ZPL	8	18.0	16.6	17.3
C with MPI	16	8.7	9.6	8.9
ZPL	16	8.6	7.9	8.3
C with MPI	32	6.1	5.8	5.7
ZPL	32	6.0	6.0	5.7
C with MPI	64	4.2	3.9	4.1
ZPL	64	4.1	4.1	4.0
C with MPI	128	3.4	3.1	3.3
ZPL	128	3.5	3.4	3.2
C with MPI	256	2.5	2.3	2.7
ZPL	256	3.1	3.1	3.0

Tests of the serial version of EP written in C, the MPI version written in C, and the implementation in ZPL 2.0 were run. Three trials of each were run to attempt to account of anomalies in the timing. Start-up time is not measured for the same reason; only the actual time of running the benchmark, as determined with gettimeofday() in the C versions, and ResetTimer() / CheckTimer() in ZPL, is displayed.

Tests were run using a single implementation for each of the process speed, as follows:

```
% ./ep-serial
% for i in 1 2 4 8 16 32 64 128 256; do
>   mpirun -np $i ep-mpi
> done
% for i in 1 2 4 8 16 32 64 128 256; do
>   ./ep-zpl -p$i
> done
```

The results show that as the number of processes increases, the benchmark time decreases, not quite at the linear rate one might expect. Also, we see that the times for the C with MPI version are nearly identical, in all cases from one process up to 128 processes (2 processes per processor), to the ZPL version running in the same number of processes.

6 Conclusion

Understanding the NAS Benchmark document can be a bit daunting. This paper has attempted to provide an alternative explanation for the Embarrassingly Parallel benchmark that I hope makes understanding easier. Additionally, it presents three implementations of the EP benchmark: a serial version written in C, a multi-process version written in C using MPI, and a multi-process version written in ZPL 2.0.

In order to gain some understanding of the relative efficiency of ZPL compared to C with MPI, three trials of the EP benchmark were run using varying numbers of processes. The number of processes were powers of two ranging from 1 to 256. These trials show that the time to run the EP benchmark is nearly identical, at each number of processes, between the ZPL version and the C with MPI version.

References

[1] National Aeronautics and Space Administration, Parallel Benchmarks
http://www.nas.nasa.gov/News/Techreports/1994/PDF/RNR-94-007.pdf

[2] MPI, the Message Passing Interface standard
http://www.mcs.anl.gov/research/projects/mpi

[3] ZPL, a parallel programming language
http://www.cs.washington.edu/research/zpl/home/index.html