Analysis of various scalar, vector, and parallel implementations of RandomAccess*

Piotr Luszczek Jack Dongarra

June 17, 2010

1 Introduction

RandomAccess test (previously called GUPS and guppie) measures a sustained rate (measured in GUPS) of updates to random locations in main memory. The random numbers may be generated by a variety of methods. In this paper two random generators are considered: one based on shift registers and one based on linear congruence relation. The quality of the main memory address distribution is considered together with the performance of the generation process. Bothe aspect are measured in various ways to create an objective setting for comparison of the generators. The tests and comparisons include both sequential as well as parallel execution each possibly using code vectorization.

RandomAccess relies on a pseudo Random Number Generator (RNG) to create a sequence of random memory locations that are updated using a simple bit-wise operation. The computational load of the update is very low. Instead, the fetch and store of the data required for the update are time consuming due to the randomness of the sequence of memory locations that span at least half ot the main memory of the tested computer. The RNG used in RandomAccess needs to preserve this balance: generating the random numbers needs to be computationally cheap and take less time than a cache miss (and possibly a Translation Look-aside Buffer – TLB – miss) that can only be resolved by the main memory and not by any of the cache levels.

2 Current RandomAccess RNG

The current RandomAccess RNG is based on the 64-bit Galois Linear Feedback Shift Register (LFSR) generator [7, 3] and uses the following primitive GF(2) polynomial[1]:

$$p(x) = x^2 + x + 1. \tag{1}$$

Due to the choice of the polynomial this generator is not maximal – there exist polynomials that result in generators with longer periods. The period of the generator is 1317624576693539401 ($= 73 \times 337 \times 889 \times 92737 \times 649657$). The period value has a repreated bit pattern that is visible in its hexadecimal representation: `0x1249249249249249`. Such bit patterns are typical in the sequences coming from LFSR generators (and such patterns are statistically expected) but might be a cause of biased address streams when used in a RandomAccess implementation. Given the polynomial from Equation (1) represented by number 7, the random numbers X_n in the sequence are generated by the following recurrence relation:

$$X_{n+1} = 2\,X_n \pmod{2^{64}} \oplus (\mathrm{MSB}(X_n) \times 7) \tag{2}$$

where $\oplus$ is bit-wise exclusive-or operator and MSB stands for the Most Significant Bit. Figure 1 shows a scalar implementation of RandomAccess in C language using Galois LFSR and standard C operators as well as 64-bit integral types (unsigned: `uint64_t` and signed one: `int64_t`). Since it is possible to arbitrarily look ahead in the random stream of Galois LFSR, the generator may be vectorized as shown in Figure 2.

3 Implementation of RandomAccess with LCG

Linear Congruential Generator (LCG) may be a valid alternative to the LFSR RNG. The LCG random sequence is generated with integer arithmetic by properly choosing three integral values a, c, and m. The random numbers X_n in the sequence are generated by the following recurrence relation:

$$X_{n+1} = (aX_n + c) \pmod{m}, \qquad n \geq 0. \tag{3}$$

By choosing $a = 6364136223846793005$ ($=3 \times 5 \times 415949 \times 1020018675983$), $c = 1$, and $m = 2^{64}$ we may obtain a sequence with a period 2^{64} [4]. The same RNG is used in the current implementation of the High Performance LINPACK (HPL) benchmark [1]. A scalar implementation in C based on this choice of LCG constants is shown in Figure 3. There are other choices of parameters a and c that lead to good quality RNGs [5, 6].

A vector implementation may be obtained by using the following look-ahead property of the LCG:

$$X_{n+k} = \left(a^k X_n + \frac{a^k - 1}{a - 1} c\right) \pmod{m},\ k \geq 0, n \geq 0. \tag{4}$$

A vectorized implementation is shown in Figure 4.

*ICL Technical Report ICL-UT-10-03

[1] GF(2) stands for Galois Field of order 2. GF(2) polynomial has coeffcients that are either 0 or 1.

```
ran = 1;
for (i=0; i <4 * M; ++i) {
   ran = (ran << 1) ^ (((int64_t) ran <0) ? 7 : 0);
   table[ran & (M-1)] ^= ran;
}
```

Figure 1: Scalar implementation of RandomAccess using Galois LFSR.

```
for (j=0; j<128; j++)
   ran[j] = LFSR_starts ((4*M/128) * j);

for (i=0; i<4*M/128; i++) {
   for (j=0; j<128; j++) {
     ran[j] = (ran[j] << 1) ^ (((int64_t) ran[j] <0) ? 7 : 0);
     table[ran[j] & (M-1)] ^= ran[j];
   }
}
```

Figure 2: Vector implementation of RandomAccess using Galois LFSR.

4 Performance Comparison of Scalar RandomAccess Codes

From the performance stand point the most important issue is the ability to vectorize the loop in the RandomAccess code. Vectorization allows the code to run on the fastest hardware unit of the machine: be it an SSE unit of the Intel x86 architecture or the vector unit of Crax X1E. If the code is not vectorized the random address issue rate might be too slow to reveal the true capability of the memory subsytem and lower the achieved GUPS rate. A common obsticle to loop vectorization is the presence of conditional statements and/or expressions. Speculative execution may be used to deal with this problem at the compiler level but this feature is not widely available (Intel Itanium and ARM processors offer the feature). There is a conditional expression in the code in Figure 1 – it has a form of C's choice operator (?:). The conditional expression may be removed by the use of C's bit operators as shown in Figure 5. Figure 6 shows performance comparison of the GLFSR codes with and without the conditional expression against the LCG scalar code from Figure 3. The codes were run on Intel Core architecutre (Family 6, Model 13, Stepping 6) equipped with 6 MiB L2 cache and with North Bridge as an external memory controller. The data TLB has 256 entries for 4 KiB pages and thus can span 1 MiB of address space at once. The results from Figure 3 indicate that there is a difference in performance for small table sizes – the biggest disparity occurring for table with 1048576 ($= 2^{20}$) entries. A table of that size barely spills the L2 cache which exacerbates the performance difference between various implementations. When the table vastly exceeds the capacity of the L2 cache all three codes deliver almost indistinguishable performance which is most heavily influenced by cache and TLB misses. When these misses do not dominate it is the pipeline stalls that can adversely affect the performance. But on the tested processor, a pipeline stall incurs tens of cycles in latency whereas highest level cache or TLB misses stall the processor for hundreds of cycles.

5 Statistical Properties of the RNGs

Simple RNGs may exihibit bad properties in some uses [10], especially in Monte Carlo simulations [8]. The LCG RNG as presented above was found satisfactory in a number of tests [4]. For testing the LFSR RNG, we limit ourselves to just few simple tests due to poor quality of the RNG. The LFSR RNG fails uniformity tests assessed by Chi-Square statistic. And this applies to both: categorization by low bits as well as categorization by floating-point intervals. We thus proceed to analyze the RNGs for the purpose of hardware benchmarking.

6 Quality of Random Sequences in the Context of Sequential RandomAccess

There are many measures of quality for RNG's [4]. However, in the context of a sequential implementation of RandomAccess it is important to have a relatively balanced distribution

```
ran = 1;
for (i=0; i <4 * M; ++i) {
    ran = 6364136223846793005ULL * ran + 1;
    table[ran & (M-1)] ^= ran;
}
```

Figure 3: Scalar implementation of RandomAccess using LCG.

```
for (j=0; j<128; j++)
    ran[j] = LCG_starts ((4*M/128) * j);

for (i=0; i<4*M/128; i++) {
    for (j=0; j<128; j++) {
      ran[j] = 6364136223846793005ULL * ran[j] + 1;
      table[ran[j] & (M-1)] ^= ran[j];
    }
}
```

Figure 4: Vector implementation of RandomAccess using LCG.

of random numbers to make sure that each generated address has to be resolved by a reference to main memory rather than cache. One issue that was often raised against the RNG in the current RandomAccess implementation was it's bias toward generating number zero much more often than any other value. Indeed, as shown in Table 1 value zero occurs 7611 times. Also, there are other values shown in the table that are generated more often than they should – on average each value should be generated exactly four times. This may be improved by not starting with value 1 but with, say, the millionth value in the sequence (the look-ahead property of the RNG may be used move to an arbitrary location in the sequence). Figure 7 shows how the number of occurances of value zero in the random sequence changes when the squence starts with the first number, the 1000000^{th} number, 2000000^{th} number and so on (for the power of 10 data set) and the first one, the second one, the fourth one and so on (for the power of 2 data set). The random sequence had 2^{30} elements. The best result (the smallest number of occurances of value zero) happens when the sequence is used starting with its 2^{30}-th element – the number of occurances of zero drops to 3441. It is still a large value considering the fact that ideally it should be 4. This problem cannot be fixed by choosing the most significant bits of the random sequence (as opposed to choosing the least significant ones which is done in the implementation in Figure 1). It does not help either to use the last 2^{30} numbers of the sequence. The results always look very similar as those included in Figure 7 – the Galois LFSR generator is biased towards zero at any part of the generated sequence. The LCG does not have the problem with excessive occurences of value zero: zero occurs either four times in the sequences when the least significant bits of the random numbers are used for indexing (as shown in Figure 3) or five times if the most significant bits are used. In fact, when the least significant bits are used, each element of the table is accessed exactly four times – an ideal situation. If the most significant bits are used, some elements of the table are not accessed at all while others are accessed at most 21 times – this is a large improvement over Galois LFSR.

But using the number of occurences of zero in the sequence might not be the best indication of problems or advantages of the resulting address stream. A more telling picture comes from the count of repeated elements in the whole sequence. Such counts are presented in Figure 8. The figure includes three random sequences of length 2^{30}: two LFSR sequences (one starts at first element and the other starts at element 2^{30}) and an LCG sequence that uses highest bits as the address. The X axis has ranges of counts while the Y axis shows the percentage of address that occured the number of times that fits the corresponding range count on X axis. For example, the tallest bar occur for the range $4 \ldots 7$ – each bar barely crosses the 50% mark on Y axis. This means that about 50% (51.5% to be exact) of address were repeated 4, 5, 6, or 7, in the sequence. The number of address that have not been generated by neither of the sequences is represented by the leftmost three bars. These three bars are don't cross the 2% mark on the Y axis: it means that less than 2% address (1.85% to be exact) were not generated from neither of the sequence. The graph goes all the way to the range $4096 \ldots 8191$ because the LFSR sequence that starts at the first element generated address zero exactly 7611 times. One conclusion may be drawn from the figure: aside from the outliers, each of

```
bit63 = 1 << 63;
ran = 1;
for (i=0; i <4 * M; ++i) {
   ran = (ran << 1) ^((uint64_t)-(int64_t)((ran & bit63) << 63) & 7);
   table[ran & (M-1)] ^= ran;
}
```

Figure 5: Scalar implementation of RandomAccess using Galois LFSR.

Value	Occurences
0	7611
7	1223
536870912	1223
1073741821	1088
93001262	1037
583371543	1033
828556680	1033
744010082	1031
186002523	1029
372005041	1027
414278340	1023
1002159032	967
787410670	965
572662305	962
143165578	953
930576247	946
501079516	945

Value	Occurences
286331155	940
71582789	928
3	882
1	877
2	868
14	772
268435456	706
805306368	705
9	639
28	466
27	459
505855053	428
939524096	426
671088640	424
81234350	421
649874809	414
394899217	412

Value	Occurences
402653184	411
452015587	409
789798437	408
825614952	406
134217728	405
162468700	404
1011710106	404
324937407	403
577488087	403
904031174	399
226007794	398
734320523	397
949678388	397
21	379
18	362
194457182	345

Table 1: The top 50 values in the random sequence of lenght 2^{30} generated by Galois LFSR and the number of times they occur.

the three methods generate the same number of repeating addresses. Still, the LCG sequence that uses the least significant bits is the best: each address is generated exactly 4 times as it should.

7 Parallel RandomAccess Considerations

RandomAccess test may be parallelized by splitting the main table between P processes as shown in Figure 9. The random stream of addresses in the main table is generated in a distributed fashion: each process generates a portion of the sequence as shown in Figure 10. The current parallel implementation of RandomAccess uses the variables defined in Table 2. The global RandomAccess table size is always a power of 2 but the number of processes may not be – a proper airthmetic ensures even distribution of data and random stream indices.

The key requirement for scalability of parallel implementation of RandomAccess is ability to cheaply look ahead in the random stream. Stepping from the begining of the stream into an arbitrarly position in the stream should preferrably be of order $O(1)$. The current parallel implementation of RandomAccess includes a function (called LFSR_starts in Figure 2) that can compute any number in the sequence in $O(\log_2 m)$ time (m being the period of the LFSR RNG). A corresponding function (called LCG_starts in Figure 4) may be implemented for LCG by using the formula from Equation (4). Such function will also have complexity $O(\log_2 m)$ for LCG with period m. Both methods RNG's are thus suitable for parallel implementation since for each of them it holds that $m \leq 2^{64} \Rightarrow \log_2 m \leq 64$.

By distributing the random sequence generated by each of the considered RNG's between the process it is possible to perform a simple test of suitability of the above RNG's for parallel RandomAccess implementation. After distributing the previously mentioned sequences of length 2^{30} evenly across the processes a measure of even distribution (and thus an even parallel load) may be a percentage difference between the minimum and maximum number of updates that each process receives. That difference is at most 0.3% when using up to 16 processes. This seems an acceptable value and does not favor decisively one RNG over another one.

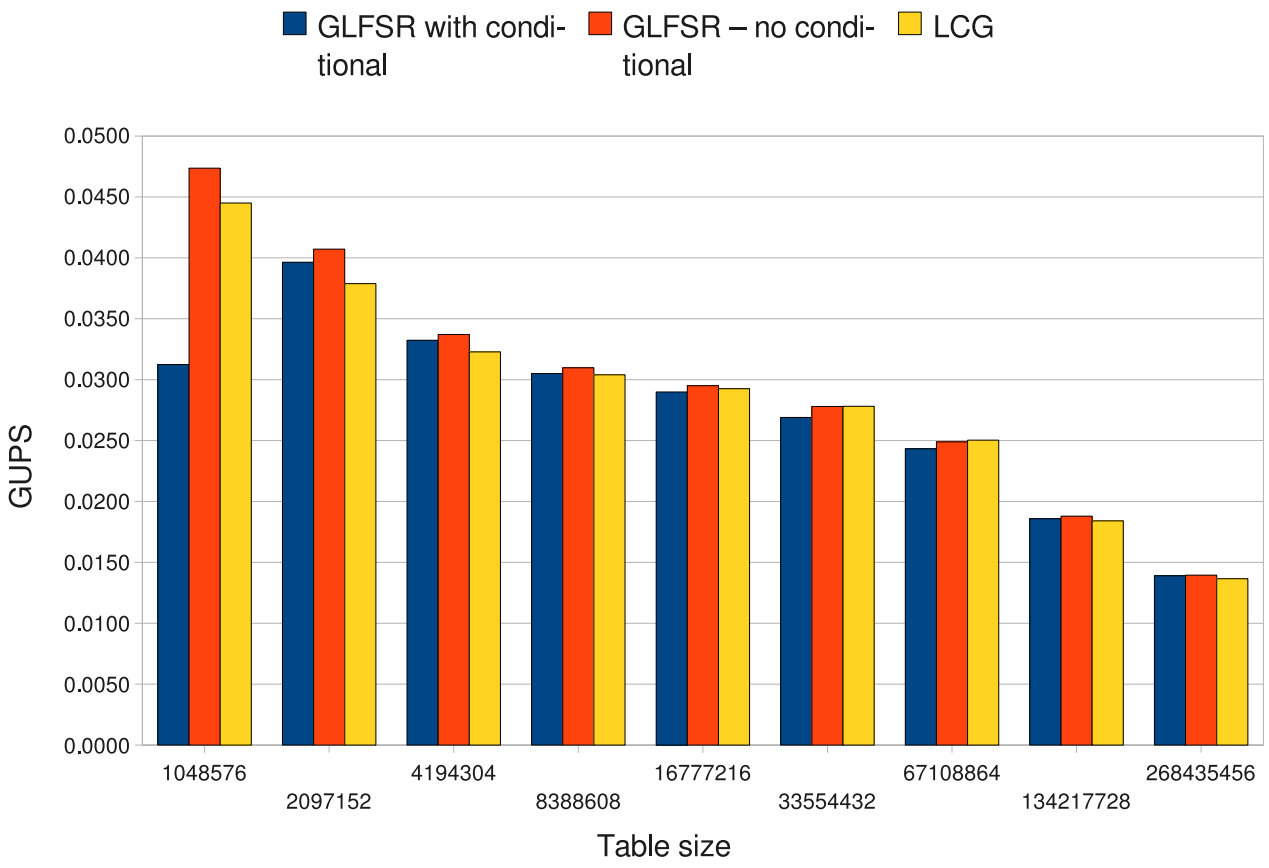

Figure 6: Performance comparison of various scalar implementations of RandomAccess.

8 Verification of Benchmarked RandomAccess Run

The source code of an optimized implementation of RandomAccess usually cannot be examined and thus a reliable method is needed to verify the correctness of the computation. This is done by peforming two RandomAccess runs: an optimized one and a reference one. Each one performs Exclusive-OR (XOR) operation on the main table entries and thus after two runs the table should return to its initial state. It is relatively easy to set the initial state of that table such that it is fast to verify a RandomAccess test. The current implementation sets the value of the i^{th} element of the table to be i. Either RNG is suitable for this verification method both in terms of accuracy of verification as well as its performance.

9 Conclusions and Future Work

The conclusion based on the tests so far is that the current RNG used in the RandomAccess implementation might have some disadvantages when compared with the LCG. On average however, it seems to be equally good. Additional tests that might shed more light on the issue are specific tests for cache line hits (for common values of cache line lengths such as 64 bytes or 128 bytes) and data TLB hits. The influence of intruction TLB seems irrelevant due to a compact nature of code but the influence of using huge pages might be worthy of note. Also, measure cache misses with toolkits such as PAPI could give a better understanding of hardware behavior for various RNG's.

To better test the parallel behavior the presented RNG's, larger sequences need to be used and larger process counts need to be simulated. Analysis of the process communication matrix for the parallel runs might also reveal potential problems and make one of the RNG's to be a preferred choice. The communication patterns should be examined for both the reference parallel implementation as well as the recent parallel implemention based on the organization of the processes into a virtual hypercube [9] as well as souftware routing approaches [2].

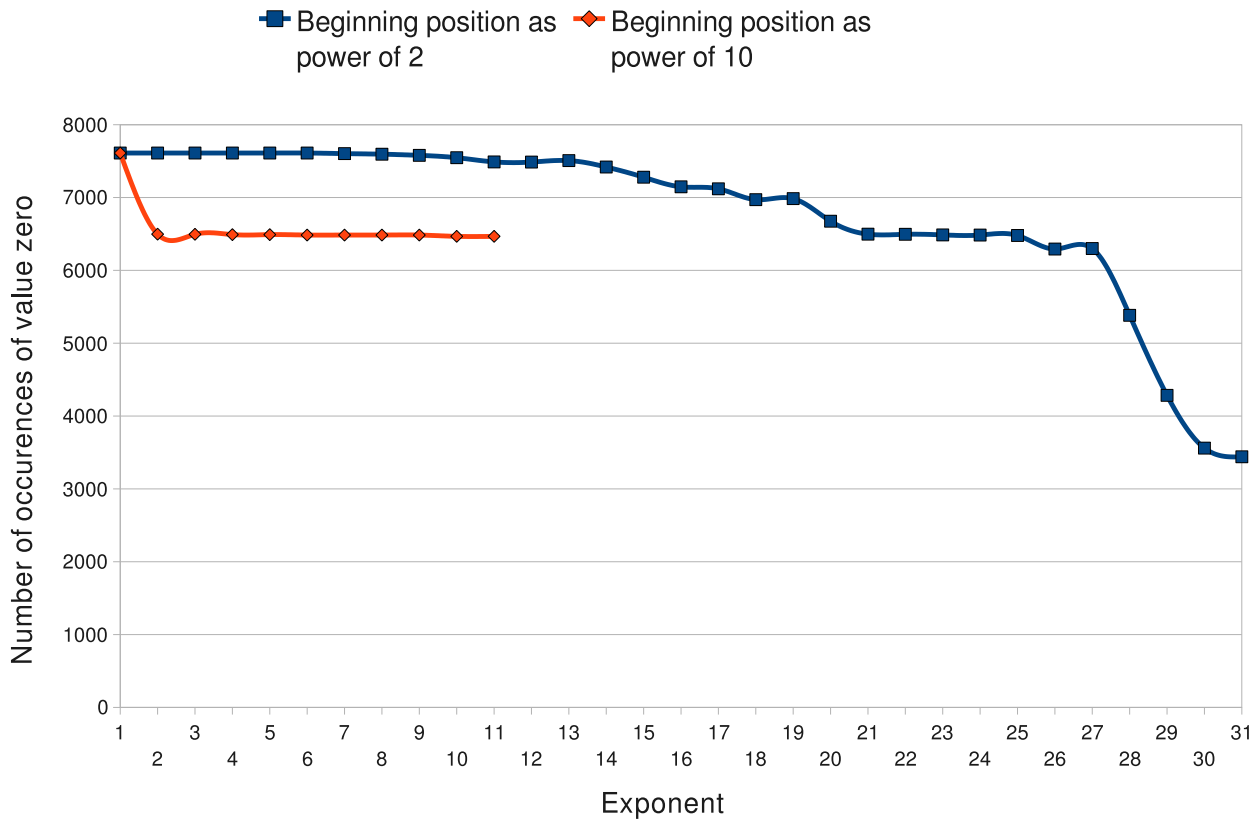

Figure 7: The number of occurences of zero in the random sequence of length 2^{30} that begin i^{th} position where i is of the form either 10^j or 2^k.

References

[1] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK benchmark: Past, present, and future. *Concurrency and Computation: Practice and Experience*, 15:1–18, 2003.

[2] Rahul Garg and Yogish Sabharwal. Software routing and aggregation of messages to optimize the p erformance of the HPCC Randomaccess benchmark. In *Proceedings of SC06*, Tampla, FL, November 11-17 2006.

[3] James E. Gentle. *Random Number Generation and Monte Carlo Methods*. Springer-Verlag New York, Inc., 2nd edition, 2003. ISBN 0-387-00178-6.

[4] Donald E. Knuth. *The Art of Computer Programming*, volume 2. Addisona-Wesley Professional, 2nd edition, October 1998.

[5] Pierre L'Ecuyer. Uniform random number generators: A review. In S. Andradóttir, K. J. Healy, D. H. Withers, and B. L. Nelson, editors, *Proceedings of the 1997 Winter Simulation Conference*, 1997.

[6] Pierre L'Ecuyer. Tables of linear congruential generators of different sizes and good lattice structure. *Mathematics of Computation*, 68(225):249–260, January 1999.

[7] T. G. Lewis and W. H. Payne. Generalized feedback shift register pseudorandom number algorithm. *Journal of the Association for Computing Machinery*, 20(3):456–468, July 1973.

[8] Giovanni Ossola and Alan D. Sokal. Systematic errors due to linear congruential random-number generators with the swendsen-wang algorithm: A warning. *Phys. Rev. E*, 70(2):027701, Aug 2004.

[9] Steve J. Plimpton, R. Brightwell, Courtney Vaughan, K . Underwood, and M. Davis. A simple synchronous distributed-memory algorithm for the HPCC RandomAccess benchmark. In *Proceedings of Cluster 2006 – IEEE International Conference on C luster Computing*, September 2006.

[10] J. P. R. Toothill, W. D. Robinson, and A. G. Adams. The runs up-and-down performance of tausworthe pseudorandom number generators. *Journal of the Association for Computing Machinery*, 18(3):381–399, July 1971.

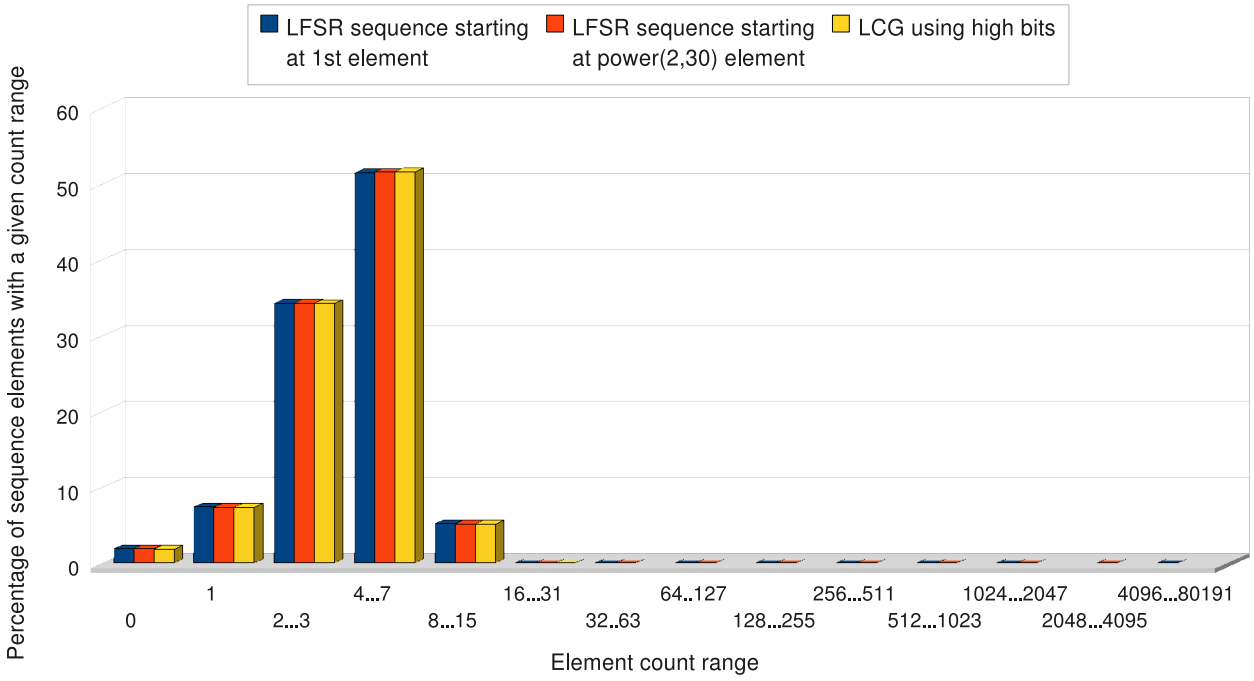

Figure 8: Counts of repeated elements for three different random sequences.

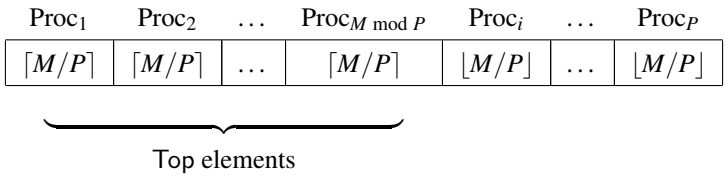

Figure 9: Parallel distribution of the main table data.

Process		First index	Last index
1	$\mapsto$	1	$4\lceil M/P\rceil$
2	$\mapsto$	$4\lceil M/P\rceil+1$	$8\lceil M/P\rceil$
	$\vdots$		
$M \bmod P$	$\mapsto$	$4\left((M \bmod P)-1\right)\lceil M/P\rceil$	$4(M \bmod P)\lceil M/P\rceil$
$(M \bmod P)+1$	$\mapsto$	$4(M \bmod P)\lceil M/P\rceil+1$	$4\left((M \bmod P)\lceil M/P\rceil+\lfloor M/P\rfloor\right)+1$
	$\vdots$		
P	$\mapsto$	$4(M-\lfloor M/P\rfloor)$	$4M$

Figure 10: Distribution of generated table indices among the processes.

Description	Math formula	Name used in code
Global size of the table	M	TableSize
Total number of processes	P	NumProcs
Size of the small local tables	$\lfloor M/P \rfloor$	MinLocalTableSize
Size of the large local tables	$\lceil M/P \rceil$	MinLocalTableSize+1
No. of processes with large local tables	$M \bmod P$	Remainder
Number of elements in large local tables	$(M \bmod P) \times \lceil M/P \rceil$	Top

Table 2: Important source code variables and their definitions.