Antoine Zoghbi

Algorithm Z1
Accelerated and parallel Generation
of Integer Partitions in Standard form

Improving the Fastest Existing Algorithm ZS1

Abstract

Algorithm ZS1, for generating integer partitions in standard anti-lexicographic order, was originally presented in Mr. Zoghbi's Master thesis "Algorithms for Generating Integer Partitions" (University of Ottawa, 1993) and later published in "Fast Algorithms for Generating Integer Partitions" (Zoghbi & Stojmenović, International Journal of Computing Mathematics, 1998, Vol. 70, pp. 319–332). It is widely regarded by specialists as representing the state of the art in this area. Algorithm Z1 is an optimized version of ZS1. Performance evaluations demonstrate that Z1 achieves an execution-time reduction of approximately 25% relative to ZS1. Multi-threaded execution of ZS1 and Z1 achieved up to an 85% reduction in runtime compared to sequential execution, highlighting the strong dependence of performance gains on compiler behavior and system architecture.

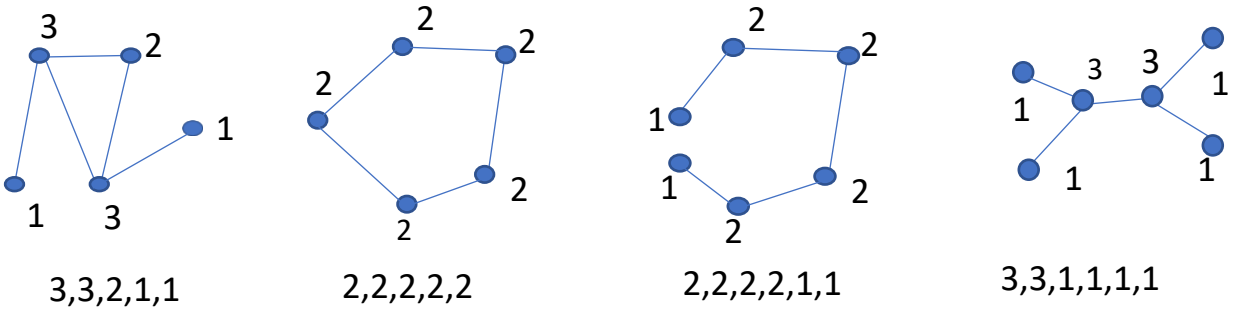
Table of Contents

Contents

1. Introduction

Mr. Zoghbi's Master thesis, "Algorithms for Generating Integer Partitions" (University of Ottawa, 1993), was followed by the paper "Fast Algorithms for Generating Integer Partitions" (Zoghbi & Stojmenović, International Journal of Computer Mathematics, 1998, Vol. 70, pp. 319–332). In this work, the algorithms ZS1 and ZS2 were introduced and were widely regarded by researchers as representing the state of the art at the time, due to their simplicity and high performance. In particular, algorithm ZS1 has been applied across a broad range of domains, including Networking, Agriculture, Music, Healthcare (e.g., DNA analysis), Civil Engineering, Space, and Aerospace. Additional references and related work are provided in Section 12.

The network graphs below show how the integer 10 can be decomposed into partitions, with nodes representing parts and number of edges indicating their values. For clarity, only a subset of all partitions of 10 is illustrated in the graphs.



The present paper offers further analysis of algorithms ZS1 and ZS2 and introduces two new algorithms, Z1 and Z2, corresponding respectively to the anti-lexicographic and lexicographic generation of integer partitions in standard form. Algorithm ZS1 is demonstrably more efficient than algorithm ZS2. Algorithm Z1 is an accelerated version of ZS1, while algorithm Z2 is an accelerated version of ZS2.

The evaluation of algorithms ZS1 and Z1 employed a single-system, multi-threaded execution model in which the integer partition space was decomposed into disjoint subsets according to ranges of the largest part. Parallel execution across these subsets produced runtime reductions of up to 75% relative to sequential execution.

All performance measurements reported in this paper were obtained with output operations excluded.

2. Partitions with parts 2 and 3

The number of integer partitions for a given n P(n), grows exponentially as n increases. The table below shows that, as n increases, the number of partitions whose leftmost part is 2 or 3, followed (if applicable) by parts equal to 1, also increases correspondingly.

Integer	# of partitions	Count_2	% Count_2	% Count_2&3
10	42	22	52.38%	66.67%
50	204,226	147,273	72.11%	80.41%
100	190,569,292	150,198,136	78.82%	84.42%
120	1,844,349,560	1,482,074,143	80.36%	85.35%
150	40,853,235,313	33,549,419,497	82.12%	86.43%
170	274,768,617,130	228,204,732,751	83.05%	87.01%

3. Algorithm ZS1 / Z1

3.1 Algorithm ZS1

```
Input integer -> n;
Input lowest value of leftmost part -> x1;
Input highest value of leftmost part -> x2;
If (x1 = 1) then {x1 ← 2};
For I ←0 to n do a[i] ←1;
A[1] ←x2+1; h←1; m←n – x2; x←0;
A[1] ←n; h←1; m←1; x←0;
While(a[1]>=x1)  do {
                        If (a[h] = 2) then {
                                        m←m+1, a[h] ←1; h←h-1;}
                        else {
                             t ← m – h + 1; a[h] ← a[h] – 1; r ←a[h];
                             while (t >= r) do { h ← h+1; a[h] ← r; t ← t – r;}
                             if  (t > 0) then {
                                             m ← h + 1;
                                             if (t > 1) then {h ← h + 1; a[h] ← t;}
                                             }
                            else { m ←h }
                            if (a[1] >= x1) then {for  i ← 1 to m do output a[i];}
                           }
if (a[1] == 1) then {for  i ← 1 to n do output a[i];}
```

Algorithm ZS1 was extended to allow the user to specify explicit lower and upper partition boundaries, thereby restricting the generation process to a designated subset of integer partitions. The algorithm handles partitions containing a part equal to 2 in a highly dynamic and efficient manner. In this case, a new partition is generated using only three instructions, which supports the conclusion that algorithm ZS1 exhibits a constant average delay property. Refer to "Fast Algorithms for Generating Integer Partitions".

As a result, ZS1 has been shown to be the fastest known algorithm in this class and is widely regarded as representing the state of the art. As the integer n increases, the proportion of partitions containing a part equal to 2 increases significantly. For example:

- For n=50, 72.11% of the partitions end with a 2 followed by ones (if any).
- For n=170, this proportion increases to 83.05%.

Thus, as n grows larger, an increasingly large fraction of partitions falls into this category, which explains the excellent practical performance and scalability of algorithm ZS1.

3.2 Algorithm Z1

We introduce algorithm Z1 as an enhancement and acceleration of algorithm ZS1. Algorithm Z1 allows the user to specify explicit lower and upper partition boundaries, thereby restricting the generation process to a designated subset of integer partitions. In addition, algorithm Z1 treats, in a direct and highly efficient manner, partitions whose rightmost part is equal to 3 followed by ones (if any), without requiring iterative processing.

Specifically, algorithm Z1 generates partitions sequentially by redistributing the rightmost part of value 3 until all parts of value 3 and 2 are reduced to ones. Each subsequent partition is produced using at most three sequential instructions, resulting in a highly efficient generation mechanism.
For example, consider n=10. The partition (4,3,3) is processed under case x=0 in the Z1 algorithm. The rightmost part equal to 3 immediately generates the following partitions in sequence: (4,3,2,2), (4,3,2,1,1), (4,3,1,1,1). Since in the last partition (4,3,1,1,1) a part equals to 3 again appears at the rightmost position before the block of ones, the partition (4,3,1,1,1) is subsequently processed under case x=3. Algorithm Z1 then directly generates, in sequence (4,2,2,2), (4,2,2,1,1), (4,2,1,1,1,1), (4,1,1,1,1,1,1).

The direct treatment of rightmost parts equal to 3 in algorithm Z1 eliminates the iterative mechanism employed in ZS1, resulting in the observed improvement in execution performance.

Algorithm Z1

```
Input integer -> n;
Input lowest value of leftmost part -> x1;
Input highest value of leftmost part -> x2;
If (x1 = 1) then {x1 ← 2;};
For I ←0 to n do a[i] ←1;
A[1] ←x2+1; h←1; m←n – x2; x←0;
While(a[1]>=x1)  do {
                    While(a[h]=2) do {
                                      m←m+1, a[h] ←1; h←h-1;
                                      for  i ← 1 to m do output a[i];
                                      }
                    if (a[h] = 3) then {
                                      x ← m – h; a[h] ←2;
                                      if (x > 3) then {
                                                        while (x > 0) do {h←h+1; a[h] ← 2; x ← x – 2;}
                                                        if (x=0) then {m ← h + 1;}
                                                     else {m ← h;}
                                                        }
                                     else if ( x = 3) then {
                                                        A[h] ← 2; m ← m – 1; a[m] ← 2;
                                                        for  i ← 1 to m do output a[i];
                                                        a[m] ← 1; m ← m + 1;
                                                       for  i ← 1 to m do output a[i];
                                                       a[h+1] ← 1; m ← m + 1;
                                                       for  i ← 1 to m do output a[i];
                                                       a[h] ← 1; h ← h – 1; m ← m + 1;}
                                     else if ( x = 1) then {
                                                       a[m] ← 2;
                                                      for  i ← 1 to m do output a[i];
                                                      a[m] ← 1; m ← m + 1;
                                                      for  i ← 1 to m do output a[i];
                                                      a[h] ← 1; h ← h – 1; m ← m + 1;}
                                     else if (x =2) then {
                                                      a[h+1] ← 2;
                                                      for  i ← 1 to m do output a[i];
                                                      a[h+1] ← 1; m ← m + 1;
                                                      for  i ← 1 to m do output a[i];
                                                      a[h] ← 1; h ← h – 1; m ← m + 1;}
                                                 else  { /* x = 0 and m=h */
                                                     m ← m + 1;
                                                     for  i ← 1 to m do output a[i];
                                                     a[h] ← 1; h ← h – 1; m ← m + 1;
```

```
                                    }
                            // for all cases x=0, x=1, x=2, x=3
                            for  i ← 1 to m do output a[i];
                            } /* a[h] =3 */
        else if (a[h] > 3) then {
                                t ← m – h + 1; a[h] ← a[h] – 1; r ←a[h]
                            while (t >= r) do {h ← h+1; a[h] ← r; t ← t – r}
                            if (t > 0) then {
                                        m ← h + 1;
                                        if (t > 1) then {h ← h + 1; a[h] ← t}
                                        }
                                    else { m ←h}
                            if (a[1] >= x1) then {for  i ← 1 to m do output a[i];}
                            } /* a[h] > 3 */
    } /* while a[1] >= x1 */
```

Algorithm Z1 improves generation speed by accelerating the processing of partitions whose rightmost structure contains a part equal to 2 or 3 immediately followed by a block of 1's (if any).

For n=170, under algorithm Z1:

- There are 207,202,977,471 partitions that contain a part equal to 2 immediately preceding the block of 1's (if any).
- There are 31,862,447,624 partitions that contain a part equal to 3 immediately preceding the block of 1's (if any).

By comparison, for n=170 under algorithm ZS1, there are 228,204,732,751 partitions that contain a part equal to 2 immediately preceding the block of 1's (if any). In ZS1, a part equal to 2 may be generated from parts equal to 3, 4, or higher. The difference $228,204,732,751-207,202,977,471=21,001,755,280$ corresponds exactly to the partitions in which a part equal to 2 is generated exclusively from a part equal to 3. In algorithm Z1, these partitions are instead handled in batch whenever a part equal to 3 is encountered, as illustrated above by the example partition (4,3,3).

Furthermore, in algorithm Z1, the ordering of the processing cases x=3,1,2,0 is determined by their relative frequencies of occurrence. In particular, the number of occurrences of case x=3 exceeds that of case x=1, and so on. This frequency-based ordering contributes directly to the improved average performance of algorithm Z1.

For n=150, algorithm Z1 increases the coverage of special cases from 82.12% (partitions with rightmost parts equal to 2 only, under ZS1) to 86.43% (partitions with rightmost parts equal to 2 or 3). Similarly, for n=170, this coverage increases from 83.35% to 87.01%. These results indicate that, as n increases, the proportion of partitions whose rightmost structure involves parts of size 2 or 3 grows steadily.

Performance tests were conducted using the C-Free compiler on an AMD Ryzen™ 7 5800HS system (8 cores, 16 logical processors, 16 GB RAM). For n=170, algorithm ZS1 required 928.369 s, whereas algorithm Z1 completed the same task in 639.046 s, corresponding to a 31.16% reduction in execution time.

AMD Ryzen 7 5800HS system

Integer	ZS1 Time Sec	Z1 Time Sec	Z1 Time Reduction (Sec)	Z1 Time Saving Sec %
10	0.000000	0.000000	0.00%	0.00%
50	0.000000	0.000000	0.00%	0.00%
100	0.607000	0.451000	74.30%	25.70%
150	127.610001	92.912003	72.81%	27.19%
170	928.369019	639.046021	68.84%	31.16%

However, when using the C-Free compiler on an Intel™ Core i7-12700H system (2.3 GHz), performance tests conducted for n=150 yielded markedly different results. Algorithm ZS1 required 111.429 s, whereas algorithm Z1 completed the same task in 111.524 s. No statistically significant performance difference was observed between the two algorithms under these conditions.

These results indicate that the performance gains of algorithm Z1 are hardware- and compiler-dependent. In particular, the improvements observed on AMD Ryzen architectures do not necessarily transfer to Intel platforms when using the same compiler, suggesting that factors such as instruction scheduling, cache behavior, and branch prediction play a significant role in determining the practical effectiveness of the proposed optimizations

4. Algorithm ZS2 / Z2

4.1 Algorithm ZS2

Algorithm ZS2 generates unrestricted integer partitions in standard lexicographic order and, to date, is the only known algorithm to operate in this category. The algorithm was subsequently extended to allow the user to specify explicit lower and upper bounds on partitions, thereby restricting the generation process to a user-defined subset of the integer partition space.

Algorithm ZS2

```
Input integer -> n;
Input lowest value of leftmost part -> x1;
Input highest value of leftmost part -> x2;
If (x1 = 1) then {x1 ← 2};
If (x2 = n) then {x2 ← x2-1};
For I ←1 to n+1 do a[i] ←1;
If (x2=2) then {for  i ← 1 to n do output a[i];}
a[0] ←-1; a[1] ←x1; h←1; m←n – x1 + 1;
for  i ← 1 to m do output a[i];
while (a[1] <= x2) do {
        if (m – h > 1) then { h←h+1; a[h] ←2; m←m-1;}
        else {
              j←m-2; k←m-1; s←a[k]; r←a[m];
              while (a[j] = s) do {a[j] ←1; r←r + s; j← j – 1;}
              h← J + 1; a[h] ← s + 1; a[m] ← 1;
             if (k != h) then {a[k] ←1;}
             m←h + r – 1;
             }
        if (a[1] <= x2) then {for  i ← 1 to m do output a[i];}
       else if (a[1] = n) {output n;}
     }
```

4.2 Algorithm Z2

Algorithm Z2 accelerates algorithm ZS2 by replacing an if statement with a while statement. This modification enables the algorithm to process certain partitions more efficiently, thereby improving overall performance. Like ZS2, algorithm Z2 allows the user to define explicit lower and upper partition boundaries, restricting the generation process to a specified subset of integer partitions.

Algorithm Z2

```
Input integer -> n;
Input lowest value of leftmost part -> x1;
Input highest value of leftmost part -> x2;
If (x1 = 1) then {x1 ← 2};
If (x2 = n) then {x2 ← x2-1};
For I ←1 to n+1 do a[i] ←1;
If (x2=2) then {for  i ← 1 to n do output a[i];}
a[0] ←-1; a[1] ←x1; h←1; m←n – x1 + 1;
for  i ← 1 to m do output a[i];
while (a[1] <= x2) do {
        while  (m – h > 1) do {
                  h←h+1;  a[h] ←2; m←m-1;
                  for  i ← 1 to m do output a[i];
                 }
         j←m-2; k←m-1; s←a[k]; r←a[m];
         while (a[j] = s) do {a[j] ←1; r←r + s; j← j – 1;}
         h← J + 1; a[h] ← s + 1; a[m] ← 1;
         if (k != h) then {a[k] ←1;}
         m←h + r – 1;
         if (a[1] <= x2) then {for  i ← 1 to m do output a[i];}
        else if (a[1] = n) {output n;}
    }
```

Performance tests were conducted using the C-Free compiler on an AMD Ryzen™ 7 5800HS system (8 cores, 16 logical processors, 16 GB RAM). The results indicate that, as the integer n increases, algorithm Z2 consistently outperforms algorithm ZS2. For n=150, algorithm ZS2 required 190.085 s, whereas algorithm Z2 completed the same task in 179.453 s, corresponding to a performance improvement of approximately 6%.

AMD Ryzen 7 5800HS system

Integer	ZS2 Time Sec	Z2 Time Sec
10	0.000000	0.000000
50	0.000000	0.000000
100	0.899000	0.876000
150	190.085007	179.453003

However, when tested on an Intel™ Core i7-12700H (2.3 GHz) system, no significant performance difference was observed between algorithm Z2 and algorithm ZS2. For n=150, algorithm ZS2 required 117.834 s, whereas algorithm Z2 completed the same task in 116.962 s.

5. Multi-Threaded Parallel Execution on the Same Hardware System

Using algorithms ZS1 and Z1, performance was evaluated through multi-threaded execution on a single hardware system, in which multiple instances of the same program were run concurrently, each generating a disjoint subset of the integer partition space. This parallel execution strategy yielded execution-time reductions of up to approximately 75%.

All experiments were conducted using the MSYS MinGW64 compiler on an AMD Ryzen™ 7 5800HS system (8 physical cores, 16 logical processors, 16 GB RAM). A balanced distribution of partitions among threads was achieved manually through multiple trial runs, resulting in a near-optimal workload allocation for the parallel computations.

5.1 Algorithms ZS1, 8 Threads, n=170

For n=170, eight-thread parallel execution of ZS1 reduced runtime from 1027.450 s to 222.000 s (21.60% of sequential time), corresponding to a 78.39% speedup, with measured CPU utilization of 55%.

Manual thread initiation introduced a ~17 s Startup overhead. Automated simultaneous launch would reduce effective runtime to 216.801 s (21.10% of sequential time), corresponding to an estimated 78.90% time saving.

AMD Ryzen 7 5800HS system CPU usage 55%							
					Parallel execution (single hardware)		
Integer 170	Partitions from	Partitions To	Number of Partitions	Time Sec	Time Sec	Time Start	Time End
ZS1	1	170	274,768,617,130	1027.449951			
ZS1	44	170	33,124,212,423	126.389999	202.574005	**13:34:50**	13:38:12
ZS1	37	43	35,185,709,823	131.731995	216.800995	13:34:52	13:38:29
ZS1	33	36	31,674,295,131	117.536003	196.304993	13:34:55	13:38:11
ZS1	30	32	29,918,886,548	111.320000	182.731995	13:34:56	13:37:59
ZS1	27	29	34,171,394,328	124.651001	202.742996	13:34:59	13:38:21
ZS1	24	26	35,646,266,112	129.867004	209.755997	13:35:02	**13:38:32**
ZS1	21	23	32,477,629,429	115.942001	195.884003	13:35:05	13:38:20
ZS1	1	20	42,570,223,336	149.074005	177.417999	13:35:07	13:38:04
TOTAL			274,768,617,130	1006.512008	**222.000000**	**13:34:50**	**13:38:32**

5.2 Algorithm Z1, 8 Threads, n=170

Under the same experimental conditions, algorithm Z1 was evaluated using eight parallel threads on the same hardware platform, with each thread executing the same program and generating a distinct subset of the integer partition space. The parallel execution completed in 181.000 s, compared with 730.684 s for the sequential run. This corresponds to 24.77% of the sequential execution time, yielding a 75.23% reduction in runtime. During parallel execution, average CPU utilization was measured at 55%.

AMD Ryzen 7 5800HS system CPU usage 55%							
					Parallel execution (single hardware)		
Integer 170	Partitions from	Partitions To	Number of Partitions	Time Sec	Time Sec	Time Start	Time End
Z1	1	170	274,768,617,130	730.684021			
Z1	44	170	33,124,212,423	91.837997	166.490005	**22:36:07**	22:38:53
Z1	37	43	35,185,709,823	94.559998	178.018997	22:36:08	22:39:06
Z1	33	36	31,674,295,131	84.371002	160.839005	22:36:09	22:38:50
Z1	30	32	29,918,886,548	76.619003	152.675995	22:36:10	22:38:43
Z1	27	29	34,171,394,328	87.563004	166.270004	22:36:11	22:38:57
Z1	24	26	35,646,266,112	89.935997	175.740997	22:36:12	22:39:07
Z1	21	23	32,477,629,429	81.930000	160.934998	22:36:13	22:38:54
Z1	1	20	42,570,223,336	100.502998	174.016006	22:36:14	**22:39:08**
TOTAL			274,768,617,130	707.319999	**181.000000**	**22:36:07**	**23:39:08**

For parallel execution using eight threads on the same hardware platform, the runtime of algorithm Z1 was reduced to 81.53% of that of algorithm ZS1 (181.000 s versus 222.000 s), corresponding to an additional time saving of 18.53%.

Overall, this trial reduced the execution time to 17.61% of the original sequential ZS1 runtime (181.000 s versus 1027.450 s), representing an 82.39% total reduction. The average CPU utilization during parallel execution was measured at 55%.

5.3 Algorithm ZS1, 17 Threads, n=150

For n=150, algorithm ZS1 was executed concurrently on 17 parallel threads on the same hardware platform, with each thread generating a disjoint subset of the integer partition space. The observed execution time was 36.000 s, compared with 153.507 s for the sequential execution. This corresponds to 23.45% of the sequential runtime, representing a 76.55% reduction in execution time. During the parallel execution, average CPU utilization was measured at 80%.

Because the parallel threads were launched manually, a Startup delay of approximately 18 s (from 09:39:05 to 09:39:23) was incurred when initiating the 17 concurrent runs. If all threads could be started simultaneously and automatically on the same hardware system, the effective execution time would be reduced to 20.702 s, corresponding to 13.48% of the sequential runtime and an estimated 86.52% reduction in execution time.

AMD Ryzen 7 5800HS system							
					Parallel execution (single hardware)		
Integer 150	Partitions from	Partitions To	# Partitions	Time Seconds	Time Sec	Time Start	Time End
ZS1	1	150	40,853,235,313	**153.507004**			
ZS1	47	150	2,359,749,941	9.146000	13.259000	**9:39:05**	9:39:18
ZS1	41	46	2,426,263,189	9.235000	15.105000	9:39:06	9:39:22
ZS1	37	40	2,731,413,157	10.288000	16.854000	9:39:08	9:39:25
ZS1	34	36	2,878,200,649	10.845000	18.813999	9:39:09	9:39:28
ZS1	32	33	2,391,007,106	9.054000	15.025000	9:39:10	9:39:25
ZS1	30	31	2,796,991,289	10.524000	18.344999	9:39:11	9:39:29
ZS1	28	29	3,195,009,613	11.744000	20.702000	9:39:12	9:39:33
ZS1	27	27	1,731,734,354	6.359000	11.525000	9:39:13	9:39:25
ZS1	26	26	1,805,546,156	6.691000	11.937000	9:39:14	9:39:26
ZS1	25	25	1,861,842,426	6.831000	12.526000	9:39:15	9:39:28
ZS1	24	24	1,895,596,593	7.023000	12.777000	9:39:16	9:39:29
ZS1	23	23	1,901,740,436	7.015000	12.964000	9:39:17	9:39:30
ZS1	22	22	1,875,641,078	6.849000	12.534000	9:39:18	9:39:31
ZS1	20	21	3,527,536,556	12.772000	21.014000	9:39:20	**9:39:41**
ZS1	18	19	2,983,095,141	10.752000	17.851000	9:39:21	9:39:39
ZS1	16	17	2,205,850,323	7.782000	13.877000	9:39:22	9:39:36
ZS1	1	15	2,286,017,336	7.920000	9.718000	9:39:23	9:39:33
TOTAL			40,853,235,343	150.830000	**36.000000**	**9:39:05**	**9:39:41**

5.4 Algorithm Z1, 17 Threads, n=150

Under the same experimental conditions, for n=150, algorithm Z1 was executed concurrently on 17 parallel threads on the same hardware platform, with each thread generating a disjoint subset of the integer partition space. The observed execution time was 28.000 s, compared with 112.378 s for the sequential run. This corresponds to 24.91% of the sequential runtime, representing a 75.09% reduction in execution time. The average CPU utilization during parallel execution was measured at 80%.

Because the parallel threads were launched manually, a startup delay of approximately 17 s was incurred when initiating the 17 concurrent runs. If all threads could be started simultaneously and automatically, the effective execution time would be reduced to 15.284 s, corresponding to 13.60% of the sequential runtime and an estimated 86.40% reduction in execution time.

AMD Ryzen 7 5800HS system							
					Parallel execution (single hardware)		
Integer 150	Partitions from	Partitions To	# Partitions	Time Seconds	Time Sec	Time Start	Time End
Z1	1	150	40,853,235,313	**112.377998**			
Z1	47	150	2,359,749,941	6.652000	10.683000	20:14:26	20:14:37
Z1	41	46	2,426,263,189	6.527000	12.317000	20:14:28	20:14:40
Z1	37	40	2,731,413,157	7.424000	14.577000	20:14:29	20:14:43
Z1	34	36	2,878,200,649	7.797000	14.357000	20:14:29	20:14:44
Z1	32	33	2,391,007,106	6.296000	12.594000	20:14:30	20:14:43
Z1	30	31	2,796,991,289	7.405000	14.414000	20:14:31	20:14:45
Z1	28	29	3,195,009,613	8.279000	15.284000	20:14:32	20:14:47
Z1	27	27	1,731,734,354	4.568000	9.095000	20:14:34	20:14:43
Z1	26	26	1,805,546,156	4.686000	9.129000	20:14:34	20:14:44
Z1	25	25	1,861,842,426	4.746000	9.416000	20:14:35	20:14:45
Z1	24	24	1,895,596,593	4.826000	9.944000	20:14:36	20:14:46
Z1	23	23	1,901,740,436	4.792000	9.663000	20:14:37	20:14:47
Z1	22	22	1,875,641,078	4.823000	9.446000	20:14:38	20:14:47
Z1	20	21	3,527,536,556	8.873000	10.484000	20:14:39	**20:14:49**
Z1	18	19	2,983,095,141	7.433000	11.711000	20:14:22	20:14:34
Z1	16	17	2,205,850,323	5.302000	6.637000	**20:14:21**	20:14:28
Z1	1	15	2,286,017,336	5.273000	6.123000	20:14:25	20:14:32
TOTAL			40,853,235,343	105.702000	**28.000000**	**20:14:21**	**20:14:49**

As an overall result of this trial, the execution time was reduced to 18.24% of the original sequential ZS1 runtime (28.000 s versus 153.507 s), corresponding to an 81.76% reduction in execution time. The average CPU utilization during the parallel execution was 80%.

5.5 ZS1 / Z1 Observations

Empirical results demonstrate that parallel execution on the same hardware system yields substantial reductions in runtime as the number of threads increases, exhibiting near-linear scaling aside from the overhead associated with loading the executables.

Another notable observation is that even when executions are performed sequentially across multiple subdivided runs to generate the complete set of partitions P(n), the total execution time is lower than that required for a single monolithic run. For example, for n=150, algorithm Z1 required 112.378 s for a single complete execution, whereas executing 17 subdivided runs sequentially required a total of 105.702 s. This behavior was consistently observed across all reported experiments, indicating that subdividing the generation process can reduce system overload and improve performance even in the absence of parallelism.

Experimental results also confirm that the performance advantage of algorithm Z1 over ZS1 is maintained under both sequential and parallel execution scenarios.

Finally, by executing the subdivided runs either sequentially or in parallel and aggregating their outputs to form the complete set of partitions P(n), integer partitions can be generated for significantly larger values of n than would be practical with a single sequential execution, owing to prohibitive time or resource requirements.

5.4 Algorithm ZS2 / Z2, 4 & 8 Threads, n=150

For n=150, algorithms ZS2 and Z2 were executed on 4 and 8 parallel threads on the same hardware platform, with each thread generating a distinct subset of the integer partition space. The results indicate no significant performance improvement, as the parallel executions exhibited runtimes nearly identical to those of the corresponding sequential executions

6. Summary

The performance results of the evaluated algorithms are summarized in the table below, which reports sequential and multi-threaded parallel execution times, percentage reductions in runtime, estimated time savings, and average CPU utilization for different values of n and varying numbers of concurrent threads. Algorithm Z1 consistently demonstrates superior performance relative to ZS1 under both sequential and parallel execution scenarios.

AMD Ryzen 7 5800HS system							
					Parallel Execution (single hardware system)		
Algorithm	n	Parallel Runs	One Execution Time Sec	Sequential Runs Time Sec	Time Sec	Time Saving	CPU usage
ZS1	150	4	154.487000	154.441996	63.000000	59.22%	28%
Z1	150	4	117.828003	108.029002	47.000000	60.11%	28%
ZS1	170	8	1027.449951	1006.512008	222.000000	78.39%	55%
Z1	170	8	730.684021	707.319999	181.000000	75.23%	55%
ZS1	150	17	153.507004	150.830000	36.000000	76.55%	80%
Z1	150	17	112.377998	105.702000	28.000000	75.08%	80%

7. Conclusions

- Algorithms Z1 and Z2 have been demonstrated to outperform ZS1 and ZS2, respectively. All four algorithms preserve the constant average delay property, although their performance depends on the underlying system architecture and compiler.
- Performance improvements of algorithms Z1 and Z2 become increasingly significant for n>100.
- Empirical measurements indicate that parallel execution on a single hardware system significantly reduces total runtime as the number of threads increases:
  - 4 threads: ≈ 60% runtime reduction
  - 8 threads: ≈ 75% runtime reduction
  - 17 threads: ≈ 75% runtime reduction

  When executable loading overhead is excluded, the 17-thread configuration achieves up to an 86% reduction in runtime.
- Subdividing the generation process can improve performance even in the absence of parallelism.
- Algorithms ZS1 and Z1 possess a distinctive advantage in that the selection of desired partitions can be achieved solely by adjusting the values of existing conditions, without modifying the algorithms themselves. Within their category, ZS1 and Z1 have proven to be faster than alternative methods.
- In contrast, other standard form approaches (outside the anti-lexicographic and lexicographic categories) typically require the introduction of additional conditional checks to generate partitions sharing the same leftmost part. Such modifications can introduce significant computational overhead, preventing these algorithms from competing effectively with Z1 and ZS1.

8. Conclusions: Algorithm Z1 vs Existing Parallel Algorithms

Theoretical analysis indicates that algorithm Z1 is a highly efficient sequential baseline, with superior per-core performance and low implementation complexity, making it competitive even against parallel approaches in overhead-dominated workloads.

8.1 Cross-Model Comparative Summary

Criterion	Algorithm Z1	Distributed Parallel	Mutli-Threaded Parallel
Per-Core Efficiency	Very High	Low-Moderate	Moderate
Absolute Throughput	Low-Moderate	Very High	High
Scalability	Low	Very High	Moderate
Execution Overhead	Minimal	High	Moderate
Implementation Complexity	Low	High	Moderate

8.2 Framework Conclusion

Algorithm Z1 serves as a high-efficiency sequential baseline, excelling in runtime efficiency and minimizing execution overhead. While existing parallel algorithms can outperform Z1 in large-scale throughput and scalability, they incur substantial costs associated with coordination, communication, and synchronization. These observations highlight that performance gains from parallelism are highly dependent on workload characteristics, and that algorithm Z1 remains competitive in scenarios where overhead dominates computation.

9. Open Discussion and challenges

- Develop a systematic and optimal method for parallel partitioning that assigns all partitions sharing the same leftmost (largest) part to a single executable. This approach ensures maximally balanced workloads across threads while minimizing inter-run overhead.

- Design a systematic mechanism for Multi-Threaded launch of the same executable on the same hardware system with one time synchronized initialization, thereby minimizing Startup latency and overall execution overhead.

- Conduct comparative assessment framework (Performance, Scalability, Runtime Efficiency, Execution Overhead) between algorithm Z1 and existing parallel algorithms, under two deployment models:
    - Distributed / multi-system parallelism
    - Single-system, multi-threaded parallelism

10. Standard Form Partitions

For n = 10, the Standard form of the Partitions are:

Antilexicografic order										Lexicographic order									
ZS1 / Z1										ZS2 / Z2									
10										1	1	1	1	1	1	1	1	1	1
9	1									2	1	1	1	1	1	1	1	1	
8	2									2	2	1	1	1	1	1	1		
8	1	1								2	2	2	1	1	1	1			
7	3									2	2	2	2	1	1				
7	2	1								2	2	2	2	2					
7	1	1	1							3	1	1	1	1	1	1	1		
6	4									3	2	1	1	1	1	1			
6	3	1								3	2	2	1	1	1				
6	2	2								3	2	2	2	2					
6	2	1	1							3	3	1	1	1	1				
6	1	1	1	1						3	3	2	1	1					
5	5									3	3	2	2						
5	4	1								3	3	3	1						
5	3	2								4	1	1	1	1	1	1			
5	3	1	1							4	2	1	1	1	1				
5	2	2	1							4	2	2	1	1					
5	2	1	1	1						4	2	2	2						
5	1	1	1	1	1					4	3	1	1	1					
4	4	2								4	3	2	1						
4	4	1	1							4	3	3							
4	3	3								4	4	1	1						
4	3	2	1							4	4	2							
4	3	1	1	1						5	1	1	1	1	1				
4	2	2	2							5	2	1	1	1					
4	2	2	1	1						5	2	2	1						
4	2	1	1	1	1					5	3	1	1						
4	1	1	1	1	1	1				5	3	2							
3	3	3	1							5	4	1							
3	3	2	2							5	5								
3	3	2	1	1						6	1	1	1	1					
3	3	1	1	1	1					6	2	1	1						
3	2	2	2	1						6	2	2							
3	2	2	1	1	1					6	3	1							
3	2	1	1	1	1	1				6	4								
3	1	1	1	1	1	1	1			7	1	1	1						
2	2	2	2	2						7	2	1							
2	2	2	2	1	1					7	3								
2	2	2	1	1	1	1				8	1	1							
2	2	1	1	1	1	1	1			8	2								
2	1	1	1	1	1	1	1	1		9	1								
1	1	1	1	1	1	1	1	1	1	10									

11. Algorithms – C codes

ZS1 - C code

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int m,n,i,h,t,r,x1,x2,y1,y2,z1,z2,z3;
long long int v,w;
clock_t clock1, clock2;
time_t tstart, tfinish;
float cpu_time_used;

int main(void) {
   printf("\n Algorithm ZS1 - Restricted \n \n \n ");
   printf("\n please enter the value of integer N : ");
   scanf ("%d", &n);
   printf("\n please enter the lowest  value of the leftmost part: ");
   scanf ("%d", &x1);
   printf("\n please enter the highest value of the leftmost part: ");
   scanf ("%d", &x2);
   if (x1 == 1) {x1 = 2;}
   printf("\n \n \n Generating Standard Antilexicographic Partitions for integer %d \n \n", n);

   int a[n+1];
   for (i=0; i<=n; ++i) a[i]=1;
   a[1] = x2 + 1; h = 1; m = n - x2;

   time(&tstart);
   printf("\n Start  : %s", ctime(&tstart));
   clock1 = clock();

   while (a[1] >= x1){
                        if (a[h] == 2) { m += 1; a[h] = 1; --h; }
                     else {
                          t = m-h+1; a[h] -=1; r = a[h];
                          while (t >= r) { h +=1; a[h] = r; t -=r;}
                          if (t > 0) {
                                     m = h + 1;
                                     if (t > 1 ) { h +=1; a[h] = t;}
                                     }
                       else { m=h;}
                         }

                     if (a[1] >= x1) {for(i = 1; i <= m; ++i) (printf(" %d", a[i])); printf(" \n");}
                       } // while (a[1] >= x1)

    if (a[1] == 1) {for(i = 1; i <= n; ++i) (printf(" %d", a[i])); printf(" \n");}
    clock2 = clock();
    cpu_time_used = ( (float) (clock2-clock1))/CLOCKS_PER_SEC;
    printf("\n cpu_time_used : %lf seconds \n", cpu_time_used);
    printf("  ( CLOCKS_PER_SEC:%d) \n",CLOCKS_PER_SEC);
    printf("\n Start  : %s", ctime(&tstart));
    time(&tfinish);
    printf("\n Finish : %s", ctime(&tfinish));
    scanf ("%d", &i);
    return 0;
}
```

Z1 – C code

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int m,n,i,h,t,r,x,x1,x2;
float cpu_time_used;
clock_t clock1, clock2;
time_t tstart, tfinish;

int main(void) {
  printf("Algorithm Z1  \n \n \n ");
  printf("\n please enter the value of integer N : ");
  scanf ("%d", &n);
  printf("\n please enter the lowest  value of the leftmost part: ");
  scanf ("%d", &x1);
  printf("\n please enter the highest value of the leftmost part: ");
  scanf ("%d", &x2);
   if (x1 == 1) {x1 = 2;}
   printf("\n \n \n Generating Standard Anti-lexicographic Partitions for integer %d \n \n", n);
   int a[n+1];
   for (i=0; i<=n; ++i) a[i]=1;
   a[1] = x2 + 1; h = 1; m = n - x2;
   time(&tstart);
   printf("\n Start  : %s", ctime(&tstart));
   clock1 = clock ();
   while (a[1] >= x1){
                          while (a[h] == 2) {
                                        ++m; a[h] = 1; --h;
                                        for (i = 1; i <= m; ++i) (printf("%d ", a[i])); printf("\n");}
                          if (a[h] == 3) {
                                       x = m - h; a[h] = 2;
                                       if (x > 3)   {
                                                    while (x > 0) {++h; a[h] = 2; x -=2;}
                                                    if (x == 0) {m = h + 1;}
                                                 else {m = h;}
                                                    }
                                  else if (x == 3) {
                                                    a[h+1] = 2; --m; a[m] = 2;
                                                    for (i = 1; i <= m; ++i) (printf("%d ", a[i])); printf("\n");
                                                    a[m] = 1; ++m;
                                                    for (i = 1; i <= m; ++i) (printf("%d ", a[i])); printf("\n");
                                                    a[h+1] = 1; ++m;
                                                    for (i = 1; i <= m; ++i) (printf("%d ", a[i])); printf("\n");
                                                    a[h] = 1; --h; ++m;}
                                  else if (x == 1) {
                                                    a[m] = 2;
                                                    for (i = 1; i <= m; ++i) (printf("%d ", a[i])); printf("\n");
                                                    a[m] = 1; ++m;
                                                    for (i = 1; i <= m; ++i) (printf("%d ", a[i])); printf("\n");
                                                    a[h] = 1; --h; ++m;}
                                  else if (x == 2) {
                                                   a[h+1] = 2;
                                                   for (i = 1; i <= m; ++i) (printf("%d ", a[i])); printf("\n");
                                                   a[h+1] = 1; ++m;
                                                   for (i = 1; i <= m; ++i) (printf("%d ", a[i])); printf("\n");
                                                   a[h] = 1; --h; ++m;}
                                       else { //x = 0  m = h
                                              ++m;
```

```
                                    for (i = 1; i <= m; ++i) (printf("%d ", a[i])); printf("\n");
                                    a[h] = 1; --h; ++m;}

                                // for all cases x=0, x=1, x=2, x=3
                                 for (i = 1; i <= m; ++i) (printf("%d ", a[i])); printf("\n");
                                 } // a[h] == 3)
                        else if (a[h] > 3) {
                                 t = m-h+1; a[h] -=1; r = a[h];
                                while (t >= r) {++h; a[h] = r; t -=r;}
                                if (t > 0) {m = h + 1;
                                            if (t > 1) {++h; a[h] = t;}
                                           }
                                    else {m = h;}
                                if(a[1]>=x1) { or (i = 1; i <= m; ++i) (printf("%d ", a[i])); printf("\n");}
                               } // (a[h] > 3)
                    } // while a[1] >=x1

    clock2 = clock();
    cpu_time_used = ( (float) (clock2-clock1))/CLOCKS_PER_SEC;
    printf("\n cpu_time_used : %lf seconds \n", cpu_time_used);
    printf("  ( CLOCKS_PER_SEC:%lld) \n",CLOCKS_PER_SEC);
    printf("\n Start  : %s", ctime(&tstart));
    time(&tfinish);
    printf("\n Finish : %s", ctime(&tfinish));
    scanf ("%d", &i);
    return 0;
}
```

ZS2 – C code

```
#include <stdio.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int m,n,i,j,h,k,t,r,s,x1,x2;
long long int v,w,y;
clock_t clock1, clock2;
time_t tstart, tfinish;
float cpu_time_used;

int main(void) {
   printf("\n Algorithm ZS2 \n \n \n ");
   printf("\n Generating Standard Lexicographic Partitions for integer %d \n \n", n);
   printf("\n please enter the value of integer N : ");
   scanf ("%d", &n);
   printf("\n please enter the lowest  value of the leftmost part: ");
   scanf ("%d", &x1);
   printf("\n please enter the highest value of the leftmost part: ");
   scanf ("%d", &x2);
   if (x1 == 1) {x1 = 2;}
   if (x2 == n) {--x2;}
   int a[n+1];
   for (i=1; i<=n; ++i) a[i]=1;
   if ( x1 == 2) {for(i = 1; i <= n; ++i) (printf(" %d", a[i]));}
   printf("\n");
   a[0] = -1; a[1] = x1;
   h = 1; m = n - x1 + 1;
   for(i = 1; i <= m; ++i) (printf(" %d", a[i]));
   printf("\n");
   time(&tstart);
   clock1 = clock();
   while (a[1] <= x2){
                        if (m - h > 1) {++h; a[h] = 2; --m;}
                     else {
                          j = m - 2; k = m - 1; s = a[k]; r = a[m];
                          while (a[j] == s) {a[j]=1; r += s; --j;}
                          h = j + 1; a[h] = s + 1; a[m] = 1;
                          if (k != h){a[k] = 1;}
                          m = h + r - 1;
                          }
                       if (a[1] <= x2) { for(i = 1; i <= m; ++i) (printf(" %d", a[i])); printf(" \n");}
                 else if (a[1] == n) { printf(" %d", n); printf(" \n");}
                     } // while (a[1] <= x2)

    clock2 = clock();
    cpu_time_used = ( (float) (clock2-clock1))/CLOCKS_PER_SEC;
    printf("\n cpu_time_used : %lf seconds \n", cpu_time_used);
    printf("  ( CLOCKS_PER_SEC:%d) \n",CLOCKS_PER_SEC);
    printf("\n Start  : %s", ctime(&tstart));
    time(&tfinish);
    printf("\n Finish : %s", ctime(&tfinish));
    scanf ("%d", &i);
    return 0;
}
```

Z2 – C code

```
#include <stdio.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int m,n,i,j,h,k,t,r,s,x1,x2;
long long int v,w,y;
clock_t clock1, clock2;
time_t tstart, tfinish;
float cpu_time_used;

int main(void) {
   printf("\n Algorithm Z2 \n \n \n ");
   printf("\n Generating Standard Lexicographic Partitions for integer %d \n \n", n);
   printf("\n please enter the value of integer N : ");
   scanf ("%d", &n);
   printf("\n please enter the lowest  value of the leftmost part: ");
   scanf ("%d", &x1);
   printf("\n please enter the highest value of the leftmost part: ");
   scanf ("%d", &x2);
   if (x1 == 1) {x1 = 2;}
   if (x2 == n) {--x2;}
   int a[n+1];
   for (i=1; i<=n; ++i) a[i]=1;
   if ( x1 == 2) { for(i = 1; i <= n; ++i) (printf(" %d", a[i]));  printf("\n");}
   a[0] = -1; a[1] = x1;
   h = 1; m = n - x1 + 1;
   for(i = 1; i <= m; ++i) (printf(" %d", a[i]));
   printf("\n");
   time(&tstart);
   clock1 = clock();
   while (a[1] <= x2){
                        while (m - h > 1) { ++h; a[h] = 2; --m;
                                            for (i = 1; i <= m; ++i) (printf(" %d", a[i])); printf(" \n");}
                        j = m - 2; k = m – 1; s = a[k]; r = a[m];
                        while (a[j] == s) {a[j]=1; r += s; --j;}
                        h = j + 1; a[h] = s + 1; a[m] = 1;
                        if (k != h){a[k] = 1;}
                        m = h + r - 1;
                        if (a[1] <= x2) {for(i = 1; i <= m; ++i) (printf(" %d", a[i])); printf(" \n");}
                  else if (a[1] == n) { printf(" %d", n); printf(" \n");}
                       } //  (a[1] <= x2)

    clock2 = clock();
    cpu_time_used = ( (float) (clock2-clock1))/CLOCKS_PER_SEC;
    printf("\n cpu_time_used : %lf seconds \n", cpu_time_used);
    printf("  ( CLOCKS_PER_SEC:%d) \n",CLOCKS_PER_SEC);
    printf("\n Start  : %s", ctime(&tstart));
    time(&tfinish);
    printf("\n Finish : %s", ctime(&tfinish));
    scanf ("%d", &i);
    return 0;
}
```

12. References:

Using google search “zoghbi integer partitions”, here are some relevant websites and online references that mainly mention or use Algorithm ZS1 (by Zoghbi & Stojmenović):

1. Book – The Art Of Computer Programming (TAOCP) Volume 4 Fascicle 3 – Donald E. Knuth
2. Book – Handbook of Applied Algorithms – Amina Nayak and Ivan Stojmenovic
3. Book – Multiagent Coordination Enabling Autonomous Logistics – Arne Schults
4. Book – Future Urban Energy System for Buildings: The Pathway Towards Flexibility – Xingxing Zhang
5. Zoghbi Stojmenovic – Fast algorithms for generating integer partitions - International Journal for Computing Math 1998, vol 70, pp 319-332
6. JA Unified Approach to Algorithms Generating Unrestricted and Restricted Integer Compositions and integer Partitions – 2010 Journal of Mathematical Modeling and Algorithms – JD Opdyke
7. Encoding Partitions As Ascending Compositions. Department of Computer Science, University College Cork – December 2005 – Jerome Kelleher
8. Georgia Institute of Technology – Layout-Aware Mixture Preparation Of Biochemical Fluids
9. National Institute of Health (NIH) – Efficient Algorithms for Calculating Epistatic Geomic…
10. Boston College – Julie Documentation
11. Sporadic.stanford.edu – William Stein and Jonathan Bober
12. University of Toronto – Space & aerospace – Efficient Homotopy Continuation algorithms with …
13. University of Toronto – Efficient Numerical Differentiation of Implicitly-Defined Curves for Sparce Systems – David A. Brown and David W. Zingg
14. SCISPACE – NASA
15. Uncit – H2F: a hierarchical Hadoop framework to process Big Data…
16. MDPI – The integer Nucleolus of Direct Simple Games
17. University of Auckland – ResearchSpace – Cristian S. Calude
18. Hal-Inria- On the cross-sectional distribution of portfolio returns
19. MetaCpan – David Landgren
20. www.arthurcarabott.com – music
21. Application in Plant Science (agEcon) – sjart_st0142.pdf
22. CHATGPT artificial Intelligence
23. A NONPARAMETRIC INDEPENDENCE TEST USING RANDOM PERMUTATIONS - By Jes´us E. Garc´ıa†,‡ and Ver´onica A. Gonz´alez-L´opez Universidade Estadual de Campinas (ZS2)
24. Developments from Programming the Partitions Method for a Power Series Expension – Victor Kowalenko
25. Fluidigm2PURC: automated processing and haplotype inference for double-barcoded PCR amplicons1 - Paul D. Blischak2,9, Maribeth Latvis3, Diego F. Morales-Briones4, Jens C. Johnson5,Verónica S. Di Stilio5, Andrea D. Wolfe2, and David C. Tank6,7,8

Additional websites and resources have become available since the preparation of this paper.Ex

Email: azog0513@gmail.com