Scientific Research Publishing

American Journal of Computational Mathematics, 2026, 16(2), 118-142
https://www.scirp.org/journal/ajcm
ISSN Online: 2161-1211
ISSN Print: 2161-1203

Algorithm Z1 Accelerated and Parallel Generation of Integer Partitions in Standard Representation Form

—Improving the Fastest Existing Algorithm ZS1

Antoine Zoghbi

9226-6642 Quebec Inc., Gatineau, Canada
Email: azog0513@gmail.com

How to cite this paper: Zoghbi, A. (2026) Algorithm Z1 Accelerated and Parallel Generation of Integer Partitions in Standard Representation Form. *American Journal of Computational Mathematics*, **16**, 118-142.
https://doi.org/10.4236/ajcm.2026.162007

Received: April 26, 2026
Accepted: June 20, 2026
Published: June 23, 2026

Abstract

Algorithm ZS1, which generates integer partitions in *standard representation* and anti-lexicographic order, was first introduced in Mr. Zoghbi's Master's thesis, "*Algorithms for Generating Integer Partitions*" (University of Ottawa, 1993), and later published in "*Fast Algorithms for Generating Integer Partitions*" by Zoghbi and Stojmenović (International Journal of Computer Mathematics, 1998, Vol. 70, pp. 319-332). The algorithm is widely regarded by specialists as one of the most efficient methods for generating integer partitions. Algorithm Z1 is an optimized refinement of ZS1. Experimental results demonstrate that Z1 achieves up to 36% reduction in execution time compared with ZS1 in single-threaded environments. In addition, multi-threaded implementations of both ZS1 and Z1 attain runtime reductions up to an 89% relative to their sequential counterparts, highlighting the significant impact of compiler optimizations and system architecture on overall performance.

Keywords

Zoghbi, Integer, Partitions, Anti-Lexicographic, Standard, ZS1, Z1, Fast, Algorithm, Sequential, Parallel, Single-Threaded, Multi-Threaded, Combinatorics

1. Introduction

Mr. Zoghbi's Master's thesis, Algorithms for Generating Integer Partitions (University of Ottawa, 1993) [1], together with the subsequent paper Fast Algorithms for Generating Integer Partitions (Zoghbi & Stojmenović, International Journal of Computer Mathematics, 1998, Vol. 70, pp. 319-332) [2], introduced the ZS1

and ZS2 algorithms. These algorithms quickly gained widespread recognition within the combinatorics community due to their simplicity, efficiency, and strong computational performance [3]-[6]. In particular, Algorithm ZS1 has been applied in a broad range of domains, including combinatorics [7]-[14], networking, agriculture [15], music [6], gaming [16], healthcare applications such as DNA analysis [17] [18], civil engineering [19], and other areas.

The network graphs shown in **Figure 1** illustrate representative partition structures of the integer 10, where nodes correspond to parts and edges represent their associated values. For clarity, only a subset of the complete set of partitions is displayed.

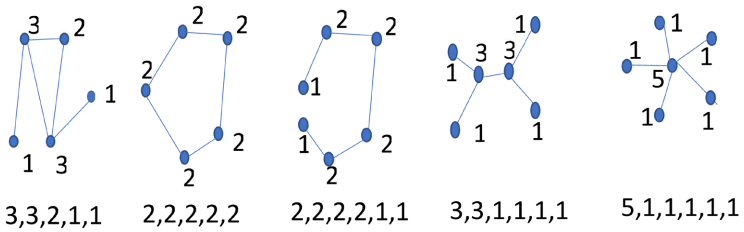

Figure 1. Mapping graphs into partitions.

This paper presents several contributions:

- Algorithm Z1, which efficiently processes partitions whose rightmost part is 3 followed, when applicable, by a sequence of 1's, thereby eliminating iterative processing for a substantial subset of cases.
- A direct concatenation strategy in Algorithm Z1 for subset cases where the rightmost part equals 3, thereby avoiding unnecessary intermediate steps and reducing computational overhead.
- Algorithm Z3, designed to generate fairly balanced and well-distributed subsets of partitions.
- A batch-processing mechanism that automatically initiates and manages parallel threads.

Algorithm Z1 (formerly AZ1), implemented in *standard representation* form and anti-lexicographic order as described in Section 6, achieves performance improvements of up to 36% in single-threaded execution and up to 89% in multi-threaded execution on identical hardware, compared with Algorithm ZS1.

Algorithms ZS1 and Z1 were evaluated on a single system using a multi-threaded execution model in which the partition space was divided into disjoint subsets according to ranges of the largest part.

All reported performance measurements exclude output operations.

2. Partitions with Parts 2 and 3

The number of integer partitions $P(n)$ grows exponentially as n increases. As shown in **Table 1**, the proportion of partitions whose leftmost part is 2 or 3 followed, when applicable, by a sequence of 1 s, also increases steadily with n.

Table 1. Count_2 & 3 ($x \leq 3$) percentage released from iterations.

n	# of partitions P(n)	Count_2 (ZS1)	Count_2% (ZS1)	Count_3 (Z1)	Count_3 ($x \leq 3$) (Z1)	Count_2 & 3 ($x \leq 3$) (Z1)	Count_2 & 3 ($x \leq 3$)% (Z1)
10	42	22	52.38%	8	6	28	66.67%
50	204,226	147,273	72.11%	35,620	16,942	164,215	80.41%
100	190,569,292	150,198,136	78.82%	28,579,511	10,681,693	160,879,829	84.42%
120	1,844,349,560	1,482,074,143	80.36%	263,565,625	92,092,200	1,574,166,343	85.35%
150	40,853,235,313	33,549,419,497	82.12%	5,479,813,969	1,759,520,420	35,308,939,917	86.43%
160	107,438,159,466	88,751,778,802	82.61%	14,137,782,672	4,428,070,930	93,179,849,732	86.73%
170	274,768,617,130	228,204,732,751	83.05%	35,500,619,847	10,860,692,344	239,065,425,095	87.01%

Breakdown of the cases in which the leftmost part equals 3, followed, when applicable, by a sequence of 1 s. In Table 2, x denotes the number of 1 s following part 3 in a given partition.

Table 2. Leftmost part 3 Breakdown.

n	Count_3	$x = 0$	$x = 1$	$x = 2$	$x = 3$	$x > 3$	$x > 3\%$	Count_3 ($x \leq 3$)
10	8	2	2	1	1	2	25.00%	6
50	35,620	5237	4510	3872	3323	18,678	52.44%	16,942
100	28,579,511	3,134,927	2,803,342	2,505,329	2,238,095	17,897,818	62.62%	10,681,693
120	263,565,625	26,708,042	24,089,760	21,719,314	19,575,084	171,473,425	65.06%	92,092,200
150	5,479,813,969	503,415,815	458,522,902	417,514,127	380,067,576	3,720,293,549	67.89%	1,759,520,420
160	14,137,782,672	1,262,212,193	1,152,724,706	1,052,460,914	960,673,117	9,709,711,742	68.68%	4,428,070,930
170	35,500,619,847	3,085,272,563	2,824,542,959	2,585,233,667	2,365,643,155	24,639,927,503	69.41%	10,860,692,344

3. Algorithms ZS1 and Z1

3.1. General Overview

For all performance evaluations presented in this document, including both single-threaded experiments in Section 3 and the multi-threaded experiments in Section 5, the following methodology was applied:

- Each benchmark was executed at least three times, and the reported results correspond to the average execution time.
- Execution time was measured in nanoseconds by recording timestamps at the beginning and end of each run.
- All programs were compiled exclusively using the MSYS MinGW64 compiler.
- Benchmarks were performed on an AMD Ryzen 7 5800HS system (8 cores, 16 threads, 16 GB RAM).
- Output operations were disabled during benchmarking by commenting them out using "//" in C language.

- Performance improvements were determined by comparing the measured execution times.
- Algorithms ZS1 and Z1 generate each partition exactly once in *standard representation* form and anti-lexicographic order.
- Algorithms ZS1 and Z1 generate partitions within the bounded range [$X1$, $X2$], where $n \geq X2 \geq X1 \geq 1$ for any given integer n.

3.2. 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 < 2) then {m ← h + t;}
                       else { m ← h + 1; h ← h + 1; a[h] ← t;}
                       }
                       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 users to specify explicit lower [X1] and upper [X2] bounds on the leftmost part of a partition, thereby restricting the generation process to a selected subset of the integer partitions of n. The algorithm processes partition whose rightmost non-unit part equals 2 with exceptional efficiency: in such case, the successor partition is generated using only three elementary operations. This behavior confirms that Algorithm ZS1 satisfies a constant average delay property established in *Fast Algorithms for Generating Integer Partitions*.

As a result, Algorithm ZS1 remains the fastest known algorithm in its class and continues to be widely used. furthermore, as n increases, the proportion of partitions containing at least one part equal to 2 followed, when applicable by a sequence of 1's grows significantly, as illustrated in **Table 1**.

- For $n = 50$, 72.11% of the partitions end with a 2 followed, when applicable by a sequence of 1 s.
- For $n = 170$, this proportion increases to 83.05%.

Consequently, as n becomes larger, this behavior helps explain the strong empirical performance and scalability of Algorithm ZS1.

3.3. Algorithm Z1

Algorithm Z1 is an optimized variant of ZS1, designed to accelerate the generation of integer partitions in *standard representation* form and anti-lexicographic order. It supports explicit lower [X1] and upper [X2] bounds on the leftmost part, enabling restricted-range generation without altering the core structure of the algorithm. Its principal enhancement is a specialized fast path for partitions whose rightmost part is 3 followed, when applicable, by a sequence of 1 s. For the subset of cases 3 ($x \leq 3$), illustrated in Table 1 and Table 2, this optimization eliminates the iterative redistribution loop used in Algorithm ZS1.

Algorithm Z1 handles such partitions by directly redistributing the rightmost 3 and propagating the transformation until all 3 s and any intermediate 2 s are reduced to 1 s. For example, for $n = 10$, the partition (4, 3, 3), triggers case $x = 0$, producing the sequence:

(4,3,2,1), (4,3,1,1,1),

In the final partition, the trailing 3 reappears immediately before the block of 1 s, triggering case $x = 3$. Algorithm Z1 then directly generates:

(4,2,2,2), (4,2,2,1,1), (4,2,1,1,1,1), (4,1,1,1,1,1,1).

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

Each output partition is constructed by concatenating a fixed prefix with suffixes obtained through redistribution of the rightmost 3. For example, when n = 11, the partition (4, 3, 2, 2), has prefix (4, 3) which is then combined with all partitions generated from redistributing (2, 2), such as (2, 1, 1) and (1, 1, 1, 1), yielding:

(4, 3, 2, 1, 1), (4, 3, 1, 1, 1, 1).

This direct concatenation mechanism eliminates repeated state reconstruction and reduces several inner-loop transitions, resulting in an additional efficiency gain of approximately 5%. For $n = 170$, this increases the overall performance improvement to approximately 89%.

By explicitly handling rightmost 3 ($x \leq 3$) structures, algorithm Z1 removes the most computationally expensive iterative steps present in algorithm ZS1, leading to significantly lower average delay and improved empirical performance. As a result, only rightmost 3 ($x > 3$) structures remain within the iterative process.

Algorithm Z1 further accelerates partition generation by optimizing cases in which the rightmost structure consists of a 2 or 3 ($x \leq 3$), followed by any number of 1 s.

For $n = 170$, Algorithm Z1 produces:

- 207,202,977,471 partitions where the rightmost part is 2 before the trailing block of 1 s.
- 31,862,447,624 partitions where the rightmost part is 3 ($x \leq 3$) before the trailing block of 1 s. Table 1 shows that Count_3 ($x \leq 3$) = 10,860,692,344 which are processed in batches to generate 31,862,447,624 partitions.

In comparison, Algorithm ZS1 generates 228,204,732,751 such partitions ending in 2. The difference

$$228,204,732,751 - 207,202,977,471 = 21,001,755,280$$

corresponds to partitions in which a 2 is created exclusively through the processing of rightmost 3 ($x \leq 3$) structures. Algorithm Z1 handles these cases collectively whenever a rightmost 3 ($x \leq 3$) is encountered, thereby improving efficiency. For clarification (21,001,755,280 + 10,860,692,344 = 31,862,447,624).

For $n = 150$, Algorithm Z1 increases the proportion of special-case partitions from 82.12% (ZS1) to 86.43% (Z1).

For $n = 170$, this proportion rises from 83.05% (ZS1) to 87.01% (Z1). Refer to Table 1.

These results show that as n increases, partitions whose rightmost parts equal to 2 or 3 ($x \leq 3$) constitute an increasingly large share of all partitions.

For $n = 170$, as shown in Table 3, algorithm ZS1 required 1099.354004 seconds in a single-threaded execution, whereas algorithm Z1 completed the same task in 696.825012 seconds, representing a 36.38% reduction in execution time.

Table 3. Z1 Time saving seconds percentage.

Integer	ZS1 Time Sec	Z1 Time Sec	Z1 Time Reduction (Sec) %	Z1 Time Saving Sec %
50	0.000000	0.000000	0.00%	0.00%
100	0.734000	0.508000	69.21%	30.79%
150	151.744003	104.516998	68.88%	31.12%
170	1099.354004	696.825012	63.38%	36.62%

When evaluated using the MSYS MinGW64 compiler on an Intel™ Core i7-12700H (2.3 GHz), performance tests for $n = 150$ show that algorithm ZS1 required 522.903992 seconds in a sequential run, whereas algorithm Z1 completed the same task in 297.003902 seconds, representing 43.20% reduction in execution time.

These results indicate that the performance advantages of Algorithm Z1 are influenced by both hardware configuration and compiler environment.

Constant Average Delay Property for Algorithm Z1

Algorithm Z1 is organized into three components for generating the partitions of a given n:

- Generation of partitions with leftmost part equal 2 (followed, if applicable, by 1 s)
- Generation of partitions with leftmost part equal 3 (followed, if applicable, by 1 s)
- Generation of partitions with leftmost part greater than 3

For partitions whose leftmost part is 2, the generation procedure is identical to that of the original Algorithm ZS1 and requires no modification.

For partitions with leftmost part greater than 3, consider for example $n = 17$ with the partition

$$(4, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1).$$

The next partition generated is

$$(3, 3, 3, 3, 3, 2).$$

In this case, the loop redistributes the 1 s $t = n - p = 17 - 4 = 13$, using parts of size $p = 4 - 1 = 3$, followed by a minor adjustment. The number of iterations is therefore approximately $\lfloor 13/3 \rfloor$, plus a constant term, resulting in about five iterations. As the size of the redistributed part p increases, the number of required iterations decreases.

For example, when $n = 17$, if the leftmost part is 10 instead of 4, the partition

$$(10, 1, 1, 1, 1, 1, 1, 1)$$

is transformed into

$$(9, 8),$$

requiring only a single iteration. The leftmost part 9, is obtained by decrementing 10, by 1. Each iteration of the while loop performs a constant number of operations, approximately three.

The most computationally expensive iterations occur when a part of size $p = 4$ is redistributed into parts of size 3, In this situation, the number of loop iterations is on the order of

$$W = \lfloor (t + 1)/(p - 1) \rfloor + S).$$

where, $S = 0$ if $(\lfloor (t + 1)/(p - 1) \rfloor \bmod 0) = 0$,

$S = 1$ if $(\lfloor (t + 1)/(p - 1) \rfloor \bmod 0) > 0$

Consequently, the total number of instructions C required to generate the next

partition, including loop execution, termination, and trailing conditional operations, can be expressed as follows:

$U = 3$ (instructions per partition generation),

$Q = 3$ (termination and trailing conditions)

$$C = ((\lfloor (t+1)/(p-1) \rfloor + S) * U + Q$$

or equivalently,

$$C = W * U + Q.$$

For partitions whose leftmost part is 3, we consider the cases 3 ($x \leq 3$), where x denotes the number of trailing 1 s. In these cases, the iterative procedure is replaced by a constant number of instructions (at most ten), including the conditional branching to determine the correct generation path. Each case produces a small batch of partitions via direct concatenation, without arithmetic iteration. When $x = 3$, four partitions are generated; when $x = 2$ or $x = 1$, three partitions are generated; and when $x = 0$, two partitions are generated. This yields an upper bound of $C = 10$ operations per generated partition.

When $x > 3$, the iterative procedure is again required, and the same expression $C = W * U + Q$ applies.

Therefore, the most computational expensive iterations occur when $t = n - 3$ and $p = 3$, giving

$$C = (\lfloor ((n-3)+1)/2 \rfloor + S) * 3 + 3.$$

For example, when $n = 15$, the partition

$$(3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)$$

generates the partition

$$(2, 2, 2, 2, 2, 2, 2, 1), \text{ using}$$

$$C = ((12+1)/2 + 1) * 3 + 3.\ C = 7 * 3 + 3.\ C = 24.$$

Let RP(n, m) denote the number of restricted partitions of n using at most m parts.

Lemma 1 and Theorem 1 are taken from my paper "*Fast Algorithms for Generating Integer Partitions*". [2]

Lemma 1. RP(n, L2) $\geq n^2/12$ for L2 ≥ 3.

Proof. Since RP(n, L2) $\geq$ RP(n, 3) for L2 > 3, it is sufficient to prove RP(n, 3) $\geq n^2/12$. In the *multiplicity representation*, the partitions RP(n, 3) are the following kind: $n = 3c_1 + 2c_2 + 1c_3$ (*i.e.* $y_1 = 3$, $y_2 = 2$, $y_3 = 1$). The number of such partitions is equal to the number of the solutions of the above equation. Clearly $0 \leq c_1 \leq \lfloor n/3 \rfloor$. For each c_1 in the interval we solve the equation $2c_2 + c_3 = n - 3c_1$. This equation has a unique solution c_3 for each c_2, $0 \leq c_2 \leq \lfloor n - 3c_1/2 \rfloor$. Therefore, for fixed c_1, the number of solutions is $(\lfloor n - 3c_1)/2 \rfloor + 1 \geq (n - 3c_1)/2$. Taking values of c_1 into account, the number of solutions is $\geq n/2 + (n-3)/2 + (n-6)/2 + \ldots + (n - 3\lfloor n/3 \rfloor)/2 = (\lfloor n/3 \rfloor + 1)n/2 - \lfloor n/3(n\lfloor/3 \rfloor + 1)3/4 = (\lfloor n/3 \rfloor + 1)(n/2 - \lfloor n/3 \rfloor 3/4)$ $\geq (\lfloor n/3 \rfloor + 1)(n/2 - n/4) \geq n^2/12$.

Theorem 1. Algorithms ZS1 and Z1 generate unrestricted integer partitions in

Standard representation with constant average delay, exclusive of the output.

Proof. Consider $x_i \geq 3$ in the current partition. It received its value after a backtracking search (starting from the last part) was performed to find an index $j \leq i$, called the turning point, that should change its value by 1 (increase/decrease for anti-lexicographic order) and to update values x_i for $j \leq i$. The time to perform both backtracking searches is $O(r_j)$ where $r_j = n - x_1 - x_2 - \ldots - x_j$ is the remainder to distribute after first j parts are fixed. We decided to change the cost of the backtrack search evenly to all "swept" parts, such that each of them receives constant O(1) time. Part x_i will be changed only after a similar backtracking step "swept" over i-th part or recognized i-th part as the turning point (note that i-th part is the turning point at least one of the two backtracking steps). There are $RP(r_i, x_i)$ such partitions which all keep x_i intact. For $x_i \geq 3$ the number of such partitions, according to Lemma 1, is $\geq r_i^2/12$. Therefore, the average number of operations that are performed by such part i during the "run" of $RP(r_i, x_i)$, including the change of its value, is $O(1)/RP(r_i, x_i) \leq O(1)/r_i^2 = O(1/r_i^2) < q_i/r_i^2$ where q_i is a constant. Thus the average number of operations for all parts of size ≥ 3 is $\leq q_1/r_1^2 + q_2/r_2^2 + \ldots + q_3/r_3^2 \leq q(1/r_1^2 + \ldots + 1/r_3^2) < q(1/n^2 + 1/(n-1)^2 + \ldots + 1/1^2) < 2q$ (the last inequality can be obtained easily by applying integral operation on the last sum), which is a constant. The case in which $x_i \leq 2$ was not considered. In this case, however, both algorithms ZS1 and Z1 perform a constant number of steps per partition on all such parts. Therefore, the algorithms ZS1 and Z1 have overall constant time average delay.

4. Generating Balanced Distribution Partitions

4.1. Algorithm Z3, 8 Threads, n = 150

Algorithm Z3, a refinement of algorithm Z1, as illustrated in Figure 2, generates a balanced distribution of partition subsets by:

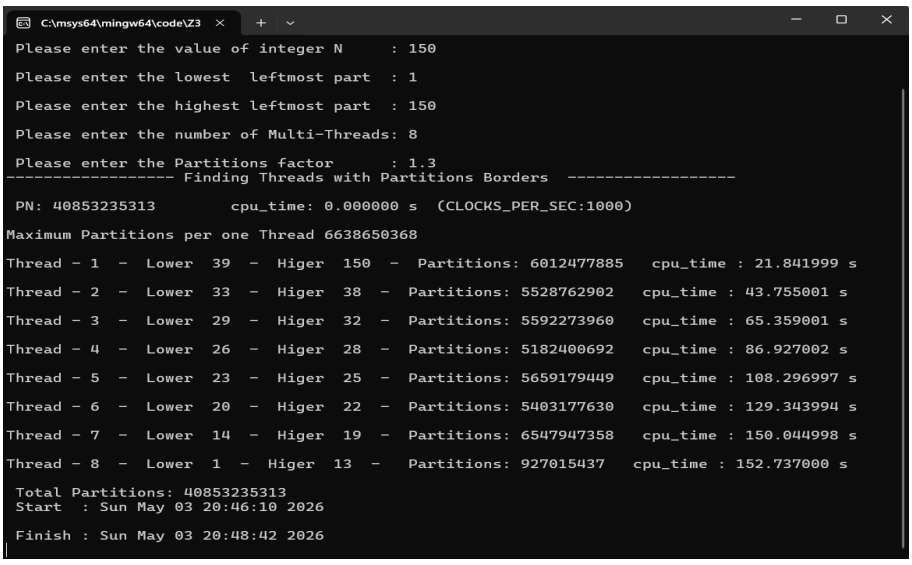

Figure 2. Example of algorithm Z3 input/output.

- Dividing the full set of partitions into disjoint subsets.
- Ensuring that all partitions sharing the same leftmost part are grouped within the same subset.
- Organizing these subsets to enable efficient parallel or distributed execution.

The *Partitions Factor* parameter improves load balancing by enabling a more uniform distribution of partitions across subsets. Each subset is assigned based on the total number of partitions and the number of available threads. Since all partitions sharing the same leftmost part must remain grouped together, any subset boundary that would otherwise divide such a block instead assigns the entire block to the next subset. As a result, the subsets do not contain identical numbers of partitions.

To facilitate efficient parallel execution, Algorithms Z1 and ZS1 were also adapted to accept input parameters directly from a batch file.

int main(int arg1, char *argv[]) {

n = atoi (argv[3]); x_1 = atoi (argv[4]); x_2 = atoi (argv[5]);

as an example in the batch file Z1a190.16.bat below: n = 190, x_1 = 28, x_2 = 29 etc….

Example of batch file: Batch file name: Z1a190.16.bat

```
echo Starting launching All scripts in parallel
start C:/msys64/mingw64/code/Z1a 3 Z1a 190 28 29
start C:/msys64/mingw64/code/Z1a 3 Z1a 190 26 27
start C:/msys64/mingw64/code/Z1a 3 Z1a 190 31 32
start C:/msys64/mingw64/code/Z1a 3 Z1a 190 23 24
start C:/msys64/mingw64/code/Z1a 3 Z1a 190 52 190
start C:/msys64/mingw64/code/Z1a 3 Z1a 190 33 34
start C:/msys64/mingw64/code/Z1a 3 Z1a 190 39 41
start C:/msys64/mingw64/code/Z1a 3 Z1a 190 42 45
start C:/msys64/mingw64/code/Z1a 3 Z1a 190 46 51
start C:/msys64/mingw64/code/Z1a 3 Z1a 190 18 20
start C:/msys64/mingw64/code/Z1a 3 Z1a 190 21 22
start C:/msys64/mingw64/code/Z1a 3 Z1a 190 35 36
start C:/msys64/mingw64/code/Z1a 3 Z1a 190 37 38
start C:/msys64/mingw64/code/Z1a 3 Z1a 190 25 25
start C:/msys64/mingw64/code/Z1a 3 Z1a 190 30 30
start C:/msys64/mingw64/code/Z1a 3 Z1a 190 1    17
echo Finish launching All scripts in parallel
```

Tasks are prioritized according to partition count and estimated execution time, with larger workloads scheduled first in order to reduce startup imbalance and ensure a more even distribution of computational load across threads.

4.2. Hardware Capacity

Algorithm Z3 computes $P(n)$ using a single array. For example, for n = 200, it produces $P(200)$ = 3,972,999,029,388. However, due to hardware and compiler constraints, generating all partitions for this value in a single sequential run is

impractical. Empirical tests on an AMD Ryzen 7 5800HS system (8 cores, 16 threads, 16 GB RAM) indicate that the largest value that can be reliably handled on a single system is approximately $n = 170$, although this limit ultimately depends on the available hardware resources.

For $n = 190$, a complete single-run execution of Algorithm ZS1 resulted in system instability. While sequential execution of individual subsets completed successfully in 6139 seconds, parallel execution of all subsets was not feasible.

Similarly, for Algorithm Z1 at $n = 190$, a complete single-run execution also proved unstable. However, sequential execution of the subsets completed successfully in 4174 seconds, and parallel execution of all subsets was achievable representing a 32% reduction in total execution time compared with ZS1 (4174 seconds versus 6139 seconds).

For $n = 200$, generating all partitions in a single run using Algorithm ZS1, was not possible. Although sequential execution of the individual subsets remained feasible, full parallel execution again resulted in instability.

For Algorithm Z1 at $n = 200$, single-run generation was likewise not achievable; however, sequential execution of the subsets remained stable and provided approximately a 34% performance improvement over Algorithm ZS1. For example, the subset 60 - 200 required 596 seconds under Algorithm ZS1, compared with 393 seconds under algorithm Z1. Full parallel execution of all subsets within a single batch was not stable; consequently, parallel processing was curried out across three batches, although occasional instability was still observed.

5. Empirical Analysis

5.1. Single-Threaded and Multi-Threaded Parallel Execution on the Same Hardware System

Performance of Algorithms ZS1 and Z1 was evaluated under multi-threaded execution model, where multiple instances of each program were executed concurrently, with each instance responsible for generating a distinct subset of the integer partition space.

Partition subsets were assigned to threads through iterative tuning to obtain an approximately optimal balance of computational workload across threads.

5.1.1. Algorithm ZS1, 16 Threads, $n = 170$

For $n = 170$, as illustrated in Table 4, execution using sixteen-thread reduced the runtime from 1099.354 seconds in sequential mode to 157.539 seconds, corresponding to 14.33% of the original runtime and representing a reduction of 85.67%. CPU utilization reached 100% at the start of execution and subsequently stabilized at approximately 97%.

5.1.2. Algorithm Z1, 16 Threads, $n = 170$

Under identical experimental conditions, as illustrated in Table 5, Algorithm Z1 was executed using sixteen parallel threads on the same hardware platform, with each thread assigned a distinct subset of the integer partition space. The parallel

Table 4. Algorithm ZS1, 16 Threads, $n = 170$.

$n = 170$	Partitions from	Partitions to	$P(n)$	Single run	Muti-threaded	Muti-threaded	Muti-threaded
				Time Sec	Time Sec	Time Start	Time End
ZS1	1	170	274,768,617,130	1099.354004			
ZS1	49	170	19,193,773,961	74.348999	140.298004	8:27:10	8:29:31
ZS1	43	48	17,659,663,004	67.250000	127.806999	8:27:10	8:29:18
ZS1	39	42	19,035,618,336	71.633003	138.615000	8:27:10	8:29:29
ZS1	36	38	19,456,938,801	72.300003	140.289993	8:27:10	8:29:30
ZS1	34	35	15,826,760,953	58.849998	116.635002	8:27:15	8:29:11
ZS1	32	33	18,218,831,388	67.400002	133.645004	8:27:10	8:29:24
ZS1	30	31	20,511,517,482	75.767998	147.029007	8:27:10	8:29:36
ZS1	28	29	22,438,541,109	82.203003	**157.539993**	**8:27:09**	**8:29:47**
ZS1	27	27	11,732,853,219	42.695999	88.816000	8:27:15	8:28:43
ZS1	26	26	11,913,686,128	43.589001	90.138000	8:27:15	8:28:45
ZS1	25	25	11,941,084,517	43.752998	90.412003	8:27:15	9:28:45
ZS1	24	24	11,791,495,467	42.509998	89.194000	8:27:15	8:28:44
ZS1	22	23	22,340,653,336	81.047997	156.395996	8:27:10	8:29:46
ZS1	20	21	19,325,817,959	69.013000	138.604004	8:27:10	8:29:28
ZS1	17	19	20,544,577,090	71.592003	144.625000	8:27:10	8:29:34
ZS1	1	16	12,836,804,380	43.292999	92.642998	8:27:15	8:28:48
TOTAL			274,768,617,130	1007.247001	**157.539993**	**8:27:09**	**8:29:47**

Table 5. Algorithm Z1, 16 Threads, $n = 170$.

$n = 170$	Partitions from	Partitions to	$P(n)$	Single run	Muti-threaded	Muti-threaded	Muti-threaded
				Time Sec	Time Sec	Time Start	Time End
Z1	1	170	274,768,617,130	696.825012			
Z1	49	170	19,193,773,961	51.344002	113.330002	8:15:05	8:16:58
Z1	43	48	17,659,663,004	46.152000	101.664001	8:15:05	8:16:49
Z1	39	42	19,035,618,336	49.728001	111.119003	8:15:10	8:17:01
Z1	36	38	19,456,938,801	50.187000	113.459999	8:15:05	8:16:58
Z1	34	35	15,826,760,953	40.425999	94.254997	8:15:05	8:16:39
Z1	32	33	18,218,831,388	46.551998	105.808998	8:15:05	8:16:51
Z1	30	31	20,511,517,482	53.355000	119.105003	8:15:05	8:17:04
Z1	28	29	22,438,541,109	56.541000	**124.427002**	**8:15:05**	**8:17:09**
Z1	27	27	11,732,853,219	29.278999	71.709000	8:15:05	8:16:17
Z1	26	26	11,913,686,128	29.958000	72.875000	8:15:06	8:16:19
Z1	25	25	11,941,084,517	29.827999	72.261002	8:15:06	8:16:18
Z1	24	24	11,791,495,467	29.166000	69.338000	8:15:06	8:16:15
Z1	22	23	22,340,653,336	54.713001	119.942000	8:15:10	8:17:09
Z1	20	21	19,325,817,959	47.192000	112.999000	8:15:10	8:17:03
Z1	17	19	20,544,577,090	48.462000	115.941002	8:15:11	8:17:07
Z1	1	16	12,836,804,380	29.027000	76.518000	8:15:11	8:16:27
TOTAL			274,768,617,130	691.909999	**124.427002**	**8:15:05**	**8:17:09**

execution completed in 124.427 seconds, compared with 696.825 seconds for the sequential execution. This corresponds to 17.86% of the sequential runtime (124.427 seconds, versus 696.825 seconds) and represents an overall reduction of 82.14% in execution time. Similarly, CPU utilization reached 100% at the start of execution and subsequently stabilized at approximately 97%.

Under sixteen-thread parallel execution on the same hardware platform, Algorithm Z1 achieved 78.98% of the runtime of Algorithm ZS1 (124.427002 seconds versus 157.539993 seconds), corresponding to a performance improvement of 21.02%.

6. Summary

The performance evaluation of the proposed algorithms is presented in Table 6, which reports sequential and multi-threaded execution times, percentage reductions in runtime, estimated time savings, and average CPU utilization across multiple values of n and varying thread counts. Across all experiments, Algorithm Z1 consistently outperforms Algorithm ZS1 under both sequential and parallel execution settings.

Table 6. Summary measurement results Algorithm ZS1 vs Algorithm Z1.

Algorithm	n	Parallel Runs	Single-threaded	Single-threaded	Muti-threaded	Muti-threaded	Muti-threaded
			Single execution Time Sec/Sec per Partition	Sequential Runs Time Sec/Sec per Partition	Time Sec/Sec per Partition	Time Saving	CPU Usage
ZS1	**150**	**4**	154.584002	154.434006	62.945999	59.28%	28%
			3.783886E−09	3.780215E−09	1.540784E−09		
Z1	**150**	**4**	108.193001	106.355001	47.875999	55.75%	28%
			2.648334E−09	2.603343E−09	1.171902E−09	69.03%	
ZS1	**150**	**17**	151.744003	150.232000	23.315001	84.64%	97%
			3.714369E−09	3.677359E−09	5.707015E−10		
Z1	**150**	**17**	105.403000	103.927000	18.730000	82.23%	97%
			2.580040E−09	2.543911E−09	4.584704E−10	87.66%	
ZS1	**170**	**16**	1099.354004	1007.247001	157.539993	85.67%	97%
			4.001017E−09	3.665801E−09	5.733551E−10		
Z1	**170**	**16**	696.825012	691.909999	124.427002	82.14%	97%
			2.536043E−09	2.518155E−09	4.528428E−10	88.68%	
ZS1	**190**	**16**	0.000000	6250.526956	0.000000	0.00%	97%
			0.000000E+00	3.747931E−09	0.000000E+00		
Z1	**190**	**16**	4174.296875	4127.428999	763.338939	81.71%	97%
			2.502985E−09	2.474882E−09	4.577121E−10	87.79%	

For the highest integer tested, $n = 170$ on an AMD Ryzen 7 5800HS system (8 cores, 16 threads, 16 GB RAM), Algorithm Z1 reduced the sequential execution time by 36.62% compared with the original sequential implementation of Algorithm ZS1 (696.825 seconds versus 1099.354 seconds). Similarly, the parallel im-

plementation of Z1 achieved a 21.02% reduction in execution time relative to the parallel implementation of ZS1 (124.427002 seconds versus 157.539993 seconds). Furthermore, comparing the parallel execution time of Z1 with the original sequential baseline of ZS1 yields an overall reduction of 88.68% (124.427002 seconds versus 1099.354 seconds). For comparison, at $n = 150$, the 17-thread implementation achieved a time saving of 87.66%. These results confirm the superior efficiency and scalability of Algorithm Z1 over Algorithm ZS1.

For clarification:

82.14% = (100 − (124.427002/696.825012)) %;

88.68% = (100 − (124.427002/1099.354004)) %

7. Conclusions

Using MSYS MinGW64 on an AMD Ryzen 7 5800HS system (8 cores, 16 threads, 16 GB RAM), Algorithm Z1 consistently outperforms algorithm ZS1 under both sequential and parallel execution conditions. Although both algorithms maintain the constant average delay property, their practical performance is influenced by system architecture and compiler behavior. The performance advantage of Algorithm Z1 becomes particularly significant for values of $n > 100$.

Empirical results demonstrate that parallel execution on a single hardware system yields substantial runtime reduction. Using four threads produces an approximate 69% decrease in execution time, while sixteen threads achieve reductions of nearly 89%. Notably, subdividing the generation process improves performance even in the absence of parallel execution.

In practice, executing a single subset is more efficient than launching multiple subsets concurrently, since parallel initialization introduces overhead that increases with the number of simultaneously active subsets. A key advantage of Algorithms ZS1 and Z1 is their support for selective partition generation through parameter adjustment alone, without requiring structural modifications. This distinguishes them from alternative standard-form generation methods, particularly those outside the anti-lexicographic and lexicographic frameworks, which typically rely on additional conditional checks that significantly increase computational cost.

Another important observation is that generating the full partition set $P(n)$ through multiple sequential executions of complete subsets is more efficient than performing a single-threaded monolithic run. For example, when $n = 170$, a full execution of Algorithm Z1 required 696.825 seconds, whereas sequential execution of sixteen subdivided runs required a total of 691.909 seconds. This consistently observed behavior suggests that subdivision reduces system load and improves performance even in the absence of parallelism.

The ability to execute subsets sequentially or in parallel and subsequently combine their outputs makes it feasible to generate partitions for significantly larger values of n than would be practical in a single run. For instance, this approach enables full computation of partitions for $n = 190$ using Algorithm ZS1.

For sufficiently large n, a trade-off emerges between executing subsets sequen-

tially or in parallel on a single system and distributing them across multiple machines, each handling a distinct subset. The case $n = 200$ highlights these considerations and illustrates the resource constraints inherent in large-scale partition generation.

For large value of n, the most efficient strategy is to distribute subsets across multiple machines using multi-threaded execution, ensuring that the number of subsets assigned does not exceed the available threads per machine and that total CPU utilization on each machine remains below approximately 97%.

All performance measurements reported in this study were obtained with output operations disabled.

8. Conclusions: Algorithm Z1 vs Existing Parallel Algorithms

Theoretical analysis indicates that Algorithm Z1, as illustrated in Table 7, operates as a high-efficiency sequential baseline, characterized by strong per-core computational throughput and minimal control-flow overhead. These properties allow it to remain competitive with parallel partition generation methods, particularly in scenarios where synchronization, communication, and thread-management overhead contribute significantly to total execution time.

Table 7. Cross-model comparative summary.

Criterion	Algorithm Z1	Distributed Parallel	Multi-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

Framework Conclusion

Algorithm Z1 serves as a highly efficient sequential baseline, characterized by strong runtime performance and minimal execution overhead. Although some parallel algorithms may offer higher throughput or improved scalability, these gains are often accompanied by additional coordination, communication, and synchronization costs. Consequently, the practical advantages of parallelism depend heavily on the structure and characteristics of the workload. In this context, Algorithm Z1 remains a competitive approach, particularly in environments where overhead limitations reduce the effectiveness of parallel execution.

9. Open Discussion and Challenges

Develop an algorithm for generating all integer partitions of large n on a single hardware system. For $n = 510$, $P(510) = 3{,}991{,}268{,}667{,}606{,}861{,}086{,}720$ which necessitates an efficient enumeration strategy. The proposed method employs a structured subdivision scheme that decomposes the solution space into subsets

organized vertically and horizontally according to the Algorithm Z1 framework. Each complete partition is formed by concatenating component partitions drawn from these subsets. The independence of the subsets enables parallel execution and also supports distribution across multiple hardware systems, thereby improving scalability and computational efficiency.

For $n = 170$, executing 16 subsets on a 16-thread processor reduced computation time by approximately 89% with average CPU utilization below 98%. Similar experiments on 32- and 64-thread systems, using 32 and 64 subsets respectively, are expected to achieve comparable or greater speedup while maintaining CPU utilization under 98%.

A comparative evaluation framework should also be developed to assess Algorithm Z1 against existing parallel algorithms. The evaluation should consider performance, scalability, runtime efficiency, and execution overhead under two deployment models:

- Distributed or multi-system parallelism
- Single-system, multi-threaded parallelism.

10. Standard Form Partitions

For $n = 10$, the Standard form of the Partitions is:

Antilexicografic order ZS1 / Z1										Lexicographic order 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. C Code: ZS1, Z1 and Z3

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 in-
teger %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 < 2) {m = h + t;}
                        else {m = h + 1; ++h; a[h] = t;}
                        }
                  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 <math.h>
#include <time.h>

int m,n,i,h,t,r,x,x1,x2;
long long int v,v0,v1,v2,v3,v23,v4,w,y,t0,t1,t2;
clock_t clock1, clock2;
time_t tstart, tfinish;
float cpu_time_used;
float vp=0.00;
int main(void) {
      printf("\n Algorithm Z1 - Restricted \n \n \n ");
      printf("\n please enter the value of integer N : ");
      scanf ("%d", &n);
      printf("\n please enter leftmost part from: ");
      scanf ("%d", &x1);
      printf("\n please enter leftmost part to    : ");
      scanf ("%d", &x2);
      if (x1 == 1) {x1 = 2;}
      printf("\n \n \n Generating Standard Antilexicographic Partitions for in-
teger %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; w=0;
      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"); ++w;}
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;}
                   } // x>3
             else if (x == 0) { //m = h
                     ++m;
                     for (i = 1; i <= m; ++i) (printf("%d ", a[i]));
printf("\n");
                     a[h] = 1; --h; ++m; ++w;}
                else if (x == 1) {
                     for (i = 1; i <= h; ++i) (printf("%d ", a[i]));
                     printf(" 2"); printf(" \n");
                     for (i = 1; i <= h; ++i) (printf("%d ", a[i]));
                     printf(" 1 1"); printf("\n");
                     a[h] = 1; --h, m += 2;    w += 2;}
           else if (x == 2) {
                     for (i = 1; i <= h; ++i) (printf("%d ", a[i]));
                     printf(" 2 1"); printf(" \n");
                     for (i = 1; i <= h; ++i) (printf("%d ", a[i]));
                     printf(" 1 1 1"); printf(" \n");
                     a[h] = 1; --h; m += 2; w += 2;}
  else     {   // x=3
                     for (i = 1; i <= h; ++i) (printf("%d ", a[i]));
                     printf(" 2 2"); printf("\n");
                     for (i = 1; i <= h; ++i) (printf("%d ", a[i]));
                     printf(" 2 1 1"); printf(" \n");
                     for (i = 1; i <= h; ++i) (printf("%d ", a[i]));
                     printf(" 1 1 1 1"); printf("\n");
                     a[h] = 1; --h; m += 2; w += 3;}

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

            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));
            printf("\n partitions_W: %lld\n", w);
            scanf ("%d", &i);
            return 0;
}
```

Z3 – C code

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

int c,c1,m,n,i,h,k,t,r,x,x1,x2,x3,y1,y2;
long long int w,w1,w2,w3,w4,v;
float x4,u1,u2,u3,u4;
clock_t clock1, clock2;
time_t tstart, tfinish;
float cpu_time_used;

int main(void) {
      printf("\n Algorithm Z3 - Multi-Thread setting \n");
      printf("\n Please enter the value of integer N          : ");
      scanf ("%d", &n);
      printf("\n Please enter the lowest    leftmost part    : ");
      scanf ("%d", &x1);
      printf("\n Please enter the highest leftmost part    : ");
      scanf ("%d", &x2);
      printf("\n Please enter the number of Multi-Threads: ");
```

```
        scanf ("%d", &x3);
        printf("\n Please enter the Partitions factor          : ");
        scanf ("%f", &x4);

        printf("------------------ Finding Threads with Partitions Borders ---------
---------\n");

        if (x1 == 1) {x1 = 2;}
        int a[n+1];
        for (i=0; i<=n; ++i) a[i]=1;
        a[1] = x2 + 1; h = 1; m = n - x2;

        time(&tstart);
        clock1 = clock();
        long long int p[n+1];
        for (i=0; i<=n; ++i) p[i]=0;

        // calculating P(n)
        p[0] = 1;
        for (k=1; k<=n; ++k) {
              for (i=k; i<=n; ++i) {p[i] += p[i-k];}}

         clock2 = clock();
         w = p[n];
         printf("\n P(n): %lld ", w);
         cpu_time_used = ( (float) (clock2-clock1))/CLOCKS_PER_SEC;
         printf("            cpu_time: %lf s", cpu_time_used);
         printf("    (CLOCKS_PER_SEC:%d) \n\n",CLOCKS_PER_SEC);

         // finding partitions borders of threads

         for (i=0; i<=n; ++i) a[i]=1;
         a[1] = x2 + 1; h = 1; m = n - x2; w1 = 0; c = 0;
         v = w / x3; v = v * x4;

         printf("Maximum Partitions per one Thread %lld \n\n", v);
         c1 = a[1];
         while (a[1] >= x1){
                  if (v > w1) {
                       if (a[1] < c1) {w2 = 0, c1 = a[1];}
             while (a[h] == 2) {++m; a[h] = 1; --h;    ++w1; ++w2;}
             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 == 0) {++m; a[h] = 1; --h; ++m; ++w1; ++ w2;}    //m =
h
                     else if (x == 1) {a[h] = 1; --h, m += 2; w1 += 2; w2 += 2;}
                     else if (x == 2) {a[h] = 1; --h; m += 2; w1 += 2; w2 += 2;}
                        else {a[h] = 1; --h; m += 2; w1 += 3; w2 += 3;} // x = 3
                       ++w1; ++w2;
                       } // 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 < 2) {m = h + t;}
                        else {m = h + 1; ++h; a[h] = t;}
                        ++w1; ++w2;
                        } // (a[h] > 3)
                            } // ( v > w1)
                    else {
                            clock2 = clock();
                            cpu_time_used      =      (      (float)      (clock2-
clock1))/CLOCKS_PER_SEC;
                            ++c; y1 = a[1] + 1; y2 = x2;
                            if (n > x2) {y2 = x2 - 1;}
                            if (y1 > y2) {y1 = y2;}
                            printf("Thread - %d  -  ", c); printf("Lower  %d
-  ", y1);  printf("Higer  %d  -  ", y2);
                            w3 = w1 - w2 - 1; w4 = w4 + w3;
                            printf("Partitions:      %lld                ",      w3);
printf("cpu_time : %lf s \n\n", cpu_time_used);
                            for (i=0; i<=n; ++i) a[i]=1;
                            x2 = y1; y1 = 0; y2 = 0; w1=0; w2=0; a[1] = x2; h = 1;
m = n - x2 + 1;
                            }
                  } // while (a[1] > x1)

            clock2 = clock();
            cpu_time_used = ( (float) (clock2-clock1))/CLOCKS_PER_SEC;
            ++y1; --x2; ++c;
            printf("Thread - %d  -  ", c); printf("Lower  %d  -  ", y1);
printf("Higer  %d  -  ", x2);
            if (c == 1) { --w1; }
```

```
        printf(" Partitions: %lld  ", w1);  printf(" cpu_time : %lf s \n\n",
cpu_time_used);
        w4 = w4 + w1;
        printf(" Total Partitions: %lld  ", w4); printf("\n Start  : %s",
ctime(&tstart));
        time(&tfinish);
        printf("\n Finish : %s", ctime(&tfinish));
        scanf ("%d", &i);
        return 0;
  }
```

Conflicts of Interest

The author declares no conflicts of interest regarding the publication of this paper.

References

[1] Zoghbi, A.C. (1993) Algorithms for Generating Integer Partitions. University of Ottawa. https://ruor.uottawa.ca/handle/10393/6506

[2] Stojmenović, I. and Zoghbi, A. (1998) Fast Algorithms for Genegrating Integer Partitions. *International Journal of Computer Mathematics*, **70**, 319-332. https://doi.org/10.1080/00207169808804755

[3] Knuth, D.E. (2005) The Art of Computer Programming (TAOCP) Volume 4. Addison Wesley.

[4] Opdyke, J.D. (2010) A Unified Approach to Algorithms Generating Unrestricted and Restricted Integer Compositions and Integer Partitions. *Journal of Mathematical Modelling and Algorithms*, **9**, 53-97. https://doi.org/10.1007/s10852-009-9116-2

[5] Landgren, D. (2007) MetaCpan.org. https://metacpan.org/pod/Integer::Partition

[6] Carabott, A. (2009) Final Year Project Dissertation. University of Sussex. https://www.arthurcarabott.com/assets/projects/konnakkol/dissertation.pdf

[7] Nayak, A. and Stojmenovic, I. (2008) Handbook of Applied Algorithms.

[8] Schults, A. (2012) Multiagent Coordination Enabling Autonomous Logistics. https://www.researchgate.net/publication/220634466_Multiagent_Coordination_Enabling_Autonomous_Logistics

[9] Kelleher, J. (2005) Encoding Partitions as Ascending Compositions. Department of Computer Science, University College Cork. https://jeromekelleher.net/downloads/k06.pdf

[10] Beaujean, F. (2017) Create Integer Partitions in Multiplicity Representation. https://gist.github.com/fredRos/1be056502742dba0753828e7852f9986

[11] Julie Documentation. Combinatorics. https://ulthiel.github.io/JuLie.jl/stable/combinatorics/

[12] Stein, W. and Bober, J. (2007) Iterators over the Partitions of an Integer. http://sporadic.stanford.edu/reference/combinat/sage/combinat/partitions.html

[13] University of Waterloo (2017) Introduction. https://cs.uwaterloo.ca/journals/JIS/VOL20/Mertens/mert4.tex

[14] Kowalenko, V. (2021) Developments from Programming the Partitions Method for a Power Series Expansion. https://arxiv.org/pdf/1203.4967

[15] Jann, B. (2008) Multinomial Goodness-of-Fit: Large-Sample Tests with Survey. AgEcon Search. https://ageconsearch.umn.edu/record/122584/files/sjart_st0142.pdf

[16] Wolff, R. (2017) The Integer Nucleolus of Directed Simple Games: A Characterization and an Algorithm. *Games*, **8**, Article 16. https://doi.org/10.3390/g8010016

[17] Blischak, P.D., Latvis, M., Morales-Briones, D.F., *et al.* (2018) Fluidigm2PURC: Automated Processing and Haplotype Inference for Double-Barcoded PCR Amplicons. http://biorxiv.org/cgi/reprint/242677v1

[18] Jiang, Y. (2020) Efficient Algorithms for Calculating Epistatic Genomic. https://pmc.ncbi.nlm.nih.gov/articles/PMC7648578/

[19] Zhang, X.X. (2023) Future Urban Energy System for Buildings: The Pathway towards Flexibility. https://dokumen.pub/future-urban-energy-system-for-buildings-the-pathway-towards-flexibility-resilience-and-optimization-9819912210-9789819912216.html