algorithms

MDPI

Article

Strategies for Parallelization of Algorithms for Integer Partition

Iliya Bouyukliev [1,2,*], Dushan Bikov [3] and Maria Pashinska-Gadzheva [1]

[1] Institute of Mathematics and Informatics, Bulgarian Academy of Sciences, 1113 Sofia, Bulgaria; mariqpashinska@math.bas.bg
[2] Centre of Excellence in Informatics and Information and Communication Technologies, 1113 Sofia, Bulgaria
[3] Faculty of Computer Science, Goce Delcev University Stip, P.O. Box 201 Štip , North Macedonia; dusan.bikov@ugd.edu.mk
* Correspondence: iliyab@math.bas.bg

Abstract

In this work we present strategies for parallelization of algorithms for representing integers as a sum of positive integers using OpenMP (Open Multi-Processing). We consider different types of algorithms - a non-recursive algorithm, a recursive algorithm and its modifications to introduce restrictions on the number and size of the summands. Different strategies for their parallelization are presented. In order to evaluate the efficiency of the strategies, we consider both the execution times and the distribution of work among threads in the presented implementations.

Keywords: integer partition problem; parallel programming; OpenMP

1. Introduction

Generating partitions of a positive integer by expressing it as a sum of integers is a fundamental problem with wide-ranging applications in mathematics and computation [1–4]. From a theoretical point of view, there is a large amount of research and hypotheses connected to partitions of integers. Algorithms for generating and counting the distinct partitions of a given positive integer n have been studied for many years and these problems are included in classical books for generation of basic combinatorial objects such as [5,6].

The motivation for our study is rooted within the context of constructive coding theory. The problem of integer partition naturally emerges in the generation of combinatorial configurations that are relevant to the structure and analysis of codes. This work is part of a broader research project dedicated to the parallel implementation of algorithms in constructive coding theory. Within this framework, we have already developed and optimized parallel methods for generating Gray codes, computing the Algebraic Normal Form (ANF) of Boolean functions [7], and performing vectorized operations over finite fields [8]. The generation and analysis of integer partitions arises naturally in this context, particularly when constructing or classifying combinatorial structures. Given the combinatorial explosion inherent in partition-related problems, efficient parallel algorithms are crucial for handling large-scale instances. We have developed two parallel implementations of algorithms that generate distinct partitions of integer n with the parallel programming interface OpenMP (Open Multi-Processing), where each computational unit generates multiple partitions that can be used in further computations. As a case study, we compute the initial terms of the sequence defined by the number of partitions of n such that the sum of the cubes of the parts equals n^2, a problem that requires enumerating and filtering large numbers of partitions under nonlinear constraints.

check for updates

Academic Editor: Stefan Bosse

Received: 30 July 2025
Revised: 11 October 2025
Accepted: 14 October 2025
Published: 16 October 2025

Citation: Bouyukliev, I.; Bikov, D.; Pashinska-Gadzheva, M. Strategies for Parallelization of Algorithms for Integer Partition. *Algorithms* **2025**, *18*, 655. https://doi.org/10.3390/a18100655

There are a few main directions connected to algorithms for integer partition:

- Generating all partitions of n into positive integers, including both distinct and repeated summands. The distinct partitions can be generated in ascending, descending and minimal change order.
- Generating partitions with restrictions such as restricting the number of summands, restricting the minimal or maximal value of the summands and others. Some problems involving integer partitions may also require adaptive constraints and/or combinations of heterogeneous restrictions, depending on the application context.
- The order in which the partitions are generated is one of the important steps in the generation algorithm. For some problems, a minimal change generation ordering (Gray code) [9] is more appropriate. For our purposes, we focus on lexicographic and reverse lexicographic ordering of the partitions. Having an ordering for the target objects, we can also introduce an enumeration for the set of object. *Ranking* and *unranking* functions are used for this purpose. Such functions are given for the basic combinatorial objects (permutation, subsets, etc.) and are also considered for both lexicographic and reverse lexicographic orderings.
- Generating the next partition in the set of ordered partitions for a given current one. Similarly to the ranking and unranking functions, this is also an important algorithm and is typically given for basic combinatorial objects.

Integer partitions can also be viewed as solutions to certain Diophantine equations. In some special cases, they lead to some interesting integer sequences. It is known that the sum of the first n cubes is equal to the square of the sum of the first n integers $1^3 + 2^3 + \cdots + n^3 = (1 + 2 + \cdots + n)^2$. In [10], the authors have considered the sets $\{a_1, \ldots, a_n\}$, where $a_1^3 + \cdots + a_n^3 = (a_1 + \cdots + a_n)^2$. It has been proven that for every natural n there is exactly one such set that contains distinct integer elements. Integer partitions are also connected to some integer sequences, described in OEIS [11]. Such sequences are A000607 and A000537, which give the number of partitions of n into prime parts and the sum of the first n cubes, respectively.

On the other hand, integer partitions are used for the generation of combinatorial objects with specific properties [12]. Many of the studied objects can be represented as binary matrices and partitions can be used in the generation of such matrices. Usually, the constructive part is backtracking, in which the incidence matrix is expanded row by row. The number of 1's in the row is known. And with an integer partition, it is determined in which part of the row how many 1's there should be according to the arrangement of the fixed rows. Generating an integer partition is an element of algorithms for classification. In many cases, for each partition additional calculations are made in order to obtain objects with the desired properties. Therefore, generating partitions of the integer n in parallel can be beneficial for classification problems for large sets of objects. In such algorithms, partition generation is a small part of the overall computations. Thus, it is more important to have a flexible algorithm that allows different restrictions. Efficiency of the partition generation is not the main focus. Other problems, however, require fast and efficient generation of partitions.

In this work, we consider the problem of generating partitions of positive integers and the following connected problems:

- We present two different approaches for generating all distinct partitions - a recursive algorithm and a more efficient non-recursive algorithm. We present some characteristics of both methods.
- Ranking, unranking and successor functions that are used for generating the full set of distinct partitions.

- Developing parallel implementation of the considered algorithms. We present two strategies for parallelization depending on the algorithm. The first strategy uses the *task* construct to parallelize the recursive algorithm for generation. Another strategy is to partition the set of all ordered partitions into subsets and generate all partitions in each subset.
- Evaluating the effectiveness of the presented parallel implementations. For the evaluation we consider the distribution of work between computational units, the flexibility of the algorithms and their scalability with the increase of the computational units. As a case study, we also calculate the number of partitions of n such that the sum of the cubes of the parts is equal to n^2.

Sequential integer partitioning algorithms have been implemented in some of the well-known computer algebra system such as GAP 4.15.0, Magma V2.29, SageMath 10.7, etc. A theoretical development of a parallel algorithm is presented in the article [13], but the existence of an actual implementation is unknown to us. The presented parallel algorithm is conceptual and differs significantly from the algorithms presented here.

The current paper is structured as follows: in Section 2 we give a mathematical background on the problem of integer partitions. Sections 3 presents the parallel programming model, that is used. In Section 4, two sequential algorithms for generating integer distributions and ranking and unranking functions for the two main considered orderings are given. Section 5 presents the parallel implementations of the algorithms. Computational results are presented in Section 6. Finally, some conclusion remarks are given in Section 7.

2. Mathematical Background

A partition of integer n is a collection of integers such that their sum is equal to n. The summands are also called *parts* and in the general case their order can be ignored. We denote the set of all distinct partitions of n with $P(n)$ and their number by $p(n)$. Two popular ways to represent a partition are as a sequence $\langle a_1, \ldots, a_k \rangle$ or by using the multiplicity of the summands. In this work we consider sequence representation. We can order the summands in the sequence. The partition $\langle a_1, \ldots, a_k \rangle$ of n is said to be in increasing order if $a_1 \leq \cdots \leq a_k$ and in decreasing order if $a_1 \geq \cdots \geq a_k$. For example, $\langle 3, 1, 1 \rangle$ is a partition of $n = 5$ written in decreasing order and $\langle 1, 1, 3 \rangle$ is the same partition in increasing order. For the given orderings of the summands in a partition, we also consider orderings for the partitions themselves. Two natural ways to order the set of partitions $P(n)$ are lexicographic order, where the parts are listed in increasing order, and reverse lexicographic order, where the parts are listed in decreasing order. We denote the set of partitions in lexicographic order by $P(n)_{lex}$ and when the set is in reverse lexicographic order, we denote it by $P(n)_{rev}$. Table 1 gives an example of $P(n)_{lex}$ and $P(n)_{rev}$ for $n = 8$. In the table we have two sets of columns, the first of which is the rank. The second column lists the partitions in lexicographic order, and the third column gives them in reverse lexicographic order. Most generation algorithms obtain the set of all partitions in reverse lexicographic order as sequences of integers. Some algorithms for generation of integer partitions are given in [6,14–17].

There are two fundamental problems connected to partitions - calculating the total number $p(n)$ of distinct partitions and generating the set $P(n)$ of all distinct partitions. These two problems have been the focus of research through the years due to their many applications [4,18–21]. Some asymptotic estimates on $p(n)$ can be found in [22,23]. The values of $p(n)$ also form the integer sequence A000041, given in [11]. Many of the characteristics of $P(n)$ are given there.

Table 1. Partitions for $n = 8$ in lexicographic and reverse lexicographic order.

Rank	$P(n)_{lex}$	$P(n)_{rev}$	Rank	$P(n)_{lex}$	$P(n)_{rev}$
0	11111111	8	11	1223	41111
1	1111112	71	12	125	332
2	111113	62	13	134	3311
3	111122	611	14	17	3221
4	11114	53	15	2222	32111
5	11123	521	16	224	311111
6	1115	5111	17	233	2222
7	11222	44	18	26	22211
8	1124	431	19	35	221111
9	1133	422	20	44	2111111
10	116	4211	21	8	11111111

Another way to represent a partition is through the Ferrer-Young diagram given by an array of dots. For a given partition $\langle a_1, \ldots, a_k \rangle$ written in decreasing order, the Ferrer-Young diagram is an array of dots, where the i-th row contains exactly a_i dots and the row of dots is left-justified. For example, let's consider the partition $\langle 2, 1, 1 \rangle$ for $n = 4$. Then, D is the Ferrer-Young diagram, where:

$$D = \begin{array}{ll} \cdot & \cdot \\ \cdot & \\ \cdot & \end{array}$$

By reflecting the diagram across its main diagonal, we get the conjugate diagram D^*, and the corresponding partition is called a conjugate partition. For example, the conjugate partition of $\langle 2, 1, 1 \rangle$ is $\langle 3, 1 \rangle$ for $n = 4$. Ferrer-Young diagrams have many other applications, some of which are given in [24,25].

Some problems use partitions of an integer n that have some restrictions. One such restriction is having a fixed number of parts. Let us denote the set of all partitions of n having exactly k parts as $P(n,k)$ and the number of such partitions as $p(n,k)$. The value of $p(n)$ can be calculated by using $p(n,k)$ as follows:

$$p(n) = \sum_{i=1}^{n} p(n,i). \tag{1}$$

This equation and algorithms for generating $P(n,k)$ in reverse lexicographic order are shown in [6]. It is proven that the number of partitions having exactly k parts is equal to the number of partitions, where the largest part is exactly k. This can be seen in the Ferrer-Young diagrams. Therefore, we can generate the set of partitions with the largest part k using the conjugate partitions. Algorithm for generating conjugate partitions is given in [6]. For the value of $p(n,k)$ and reverse lexicographic order, we have the following recurrence relation, also given in [6]:

$$p(n,k) = p(n-1,k-1) + p(n-k,k) \tag{2}$$

Let us consider the set $\widehat{P}(n,k)$ that consists of the partitions in $P(n)$ with smallest part at least k and the set $P^*(n,k)$ that consists of the partitions of $P(n)$ with largest part at most k, respectively. Analogously, $\widehat{p}(n,k)$ and $p^*(n,k)$ give the number of partitions in the corresponding sets. For example, for $n = 8$ and $k = 2$, if we look at Table 1, we have $\widehat{p}(8,2) = 7$ and $p^*(8,2) = 5$. For the values of $\widehat{p}(n,k)$ and $p^*(n,k)$ we have recurrence relations given by Lemma 1 and Lemma 2. These recurrence relations are analogous to (2). For $\widehat{p}(n,k)$ and $p^*(n,k)$ we also have $\widehat{p}(n,0) = p^*(n,n) = p(n)$.

Lemma 1. *Let n and k be positive integers. Then*

$$\widehat{p}(n,k) = \begin{cases} 0 & \text{if } k > n \\ 1 & \text{if } k = n \\ \widehat{p}(n-k,k) + \widehat{p}(n,k+1) & \text{otherwise} \end{cases}$$

Proof. Obviously, the summands in any partition of n are $\leq n$ and therefore $\widehat{p}(n,k) = 0$ when $k > n$. The only partition with the smallest summand equal to n is the partition with only one part.

Let $k < n$. We consider the set $\widehat{P}(n,k)$ as a union of the subsets $\widehat{P}(n,k)_k$ with the smallest part equal to k and $\widehat{P}(n,k+1)$. We need to prove that $|\widehat{P}(n,k)_k| = \widehat{p}(n-k,k)$. Let us consider the map $\phi : \widehat{P}(n,k)_k \to \widehat{P}(n-k,k)$ defined by

$$\phi(\langle k, a_2, \ldots, a_l \rangle) = \langle a_2, \ldots, a_l \rangle,$$

where $l+1$ is the number of summands in the current partition and all partitions are ordered lexicographically ($\widehat{P}(n,k) \subseteq P(n)_{lex}$). Then $a_2 + \cdots + a_l = n - k$ and therefore $\langle a_2, \ldots, a_l \rangle$ is a partition of $n-k$. Moreover, $k \leq a_2 \leq \cdots \leq a_l$, hence $\langle a_2, \ldots, a_l \rangle \in \widehat{P}(n-k,k)$. This proves that ϕ is an injection from $\widehat{P}(n,k)_k$ to $\widehat{P}(n-k,k)$. Furthermore, if $\langle a_2, \ldots, a_l \rangle \in \widehat{P}(n-k,k)$, then $\langle a_2, \ldots, a_l \rangle = \phi(\langle k, a_2, \ldots, a_l \rangle)$, and so ϕ is a surjection. It follows that ϕ is a bijection from $\widehat{P}(n,k)_k$ to $\widehat{P}(n-k,k)$ and therefore these two sets have the same cardinality. This proves that in this case $\widehat{p}(n,k) = \widehat{p}(n-k,k) + \widehat{p}(n,k+1)$. □

Lemma 2. *Let $n > 0$ and $k \geq 0$ be integers. Then*

$$p^*(n,k) = \begin{cases} p(n) & \text{if } k \geq n \\ 0 & \text{if } k = 0 \\ p^*(n-k,k) + p^*(n,k-1) & \text{otherwise} \end{cases}$$

Proof. The proof is similar to the proof of the Lemma 1. The difference is only that instead $P(n)_{lex}$, we use $P(n)_{rev}$, where the partitions are in reverse lexicographic order. The number of the partitions with largest part at most n or more is equal to the number of all partitions of n. There are no partitions with a zero summand and therefore $p^*(n,0) = 0$.

Let us consider the case $k < n$. Then $P^*(n,k)$ is a union of the subsets $P^*(n,k)_k$ with the biggest part equal to k and $P^*(n,k-1)$. We now consider the map $\phi^* : P^*(n,k)_k \to P^*(n-k,k)$ defined by

$$\phi^*(\langle k, a_2, \ldots, a_l \rangle) = \langle a_2, \ldots, a_l \rangle \in P^*(n-k,k),$$

where $l+1$ is the number of summands in the current partition. The map ϕ^* is injective (different partitions in $P^*(n,k)_k$ have different images) and surjective (any partition in $P^*(n-k,k)$ is an image of a partition from $P^*(n,k)_k$), hence it is a bijection from $P^*(n,k)_k$ to $P^*(n-k,k)$ and therefore these two sets have the same cardinality. This proves that in this case $p^*(n,k) = p^*(n-k,k) + p^*(n,k-1)$. □

For basic combinatorial objects, there are also ranking, unranking and successor functions in addition to generation. A ranking algorithm gives the position (rank) of an object in the set of all objects with respect to a given order, while unranking algorithms obtain the object corresponding to a given rank. A successor function obtains the next object in a chosen order for a given current object. These algorithms can sometimes be used for more efficient generation. Unranking functions in particular can be used for generating

objects in parallel. Ranking and unranking for the elements in $P(n,k)_{lex}$ and $P(n)_{rev}$ are given in [6]. Successor algorithms for $P(n)_{lex}$ and $P(n)_{rev}$ are given in [6,14].

3. Parallel Programming Model

OpenMP, short for 'Open Multi-Processing', is an Application Programming Interface (API) designed to support parallel programming in shared-memory environments using C, C++, and Fortran [26]. It enables developers to write parallel code that executes concurrently across multiple processors, cores, or threads within a single system. OpenMP simplifies the parallelization process by providing a set of compiler directives and library routines, allowing developers to define parallel regions in their code without managing low-level threading details. This abstraction makes OpenMP an accessible and efficient tool for developing high-performance parallel applications.

OpenMP operates on a shared-memory model, where threads can access both private and shared variables. Work-sharing constructs, such as parallel and for, efficiently distribute tasks among threads, optimizing resource utilization. Renowned for its portability, OpenMP is compatible with various compilers across platforms, making it a versatile tool for parallel programming.

Its simplicity and incremental integration make OpenMP particularly accessible, attracting a broad range of developers, especially in scientific and engineering domains. By leveraging parallelism, OpenMP significantly boosts computational efficiency, making it widely used in applications such as numerical simulations, data analytics, and other compute-intensive tasks.

The tasking execution model (Task construct), illustrated in Figure 1, facilitates the parallelization of irregular (unstructured) patterns and recursive function calls by defining independent units of work that are executed by a thread. Several scenarios are possible: a single creator, typically implemented using a *parallel* construct followed by *single*, and multiple creators, leveraging work-sharing and nested tasks to efficiently distribute the workload.

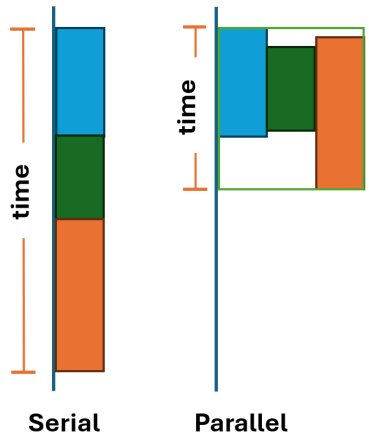

Figure 1. Task execution model.

The creation of threads and tasks in OpenMP is generally performed through compiler directives and functions defined in the *omp.h* header and its associated library. The concept of tasking was introduced in OpenMP 3.0, however, not all compilers support OpenMP 3.0 and its tasking capabilities. Additional information about the OpenMP programming interface can be found in [26].

4. Sequential Implementation

In this section we present two sequential algorithms for generating all partitions of an integer n - a recursive generation and a non-recursive generation using the successor function. We also present ranking and unranking functions. We consider the functions for generating the partitions in lexicographic and reverse lexicographic order. In the presented pseudocode of the algorithms we use the function *usePartition()* to indicate further computations that use the currently generated partition (e.g., generating incedence matrix of a specific combinatorial object).

4.1. Recursive Algorithm

The most natural algorithms for generating integer partitions are recursive. Such algorithms provide a powerful and flexible framework for problem-solving, enabling elegant solutions to complex challenges by decomposing problems into smaller, more manageable sub-problems. Integer partitioning is an excellent example of the use of recursion. The recursive algorithm essentially breaks down the problem of partitioning a positive integer as a sum of natural numbers into smaller sub-problems, providing a systematic way to explore all possible partitions. Recursive algorithms for generation of integer partitions are given in [6,14].

We can present the integer n as sum of $n-k$ and k starting with $k=1$. Afterwards, we can recursively apply the partitioning for both $n-k$ and k. In this case, the parts will be ordered in decreasing order and the partitions in reverse lexicographic order. In general, we have the following steps:

- Base Case: If the input is 0, return an empty set, as there is no partition for it.

$$partitions(0) = \{\}$$

- Recursive Step: For a given natural number n:

$$partitions(n) = \{k\} \cup partitions(n-k); \text{for } k = n, n-1, \ldots, 1$$

  - Start with the largest possible value of k (which is n or the previous value of k).
  - Recursively apply the same partitioning algorithm to the remaining sum $(n-k)$.
  - Continue decreasing the value of k until reaching 1.

Note: We need to be cautious and avoid generating repetitive partitions.

In each recursive iteration, in order to generate only distinct (and ordered) partitions, the starting value of k is either n or the value of k from the previous recursive iteration. Using a variable pos_start, that contains the value of k from the previous recursive call of the function, also ensures avoiding generating duplicate partitions. The recursive steps apply the partition algorithm to the remaining sum $(n-k)$. This process continues, decreasing the value of k down to 1. In Algorithm 1, we present pseudocode for generating all partitions in reverse lexicographic order:

Algorithm 1 PARTNUM(n, pos, $part$, pos_start)

Input:
n – integer to be partitioned in this step
pos – index in $part$ where the new summand will be placed
$part$ – array containing the current partition
pos_start – value of the part added in the previous recursive call
Output:
Generates all partitions of n recursively in non-increasing order.
Calls `usePartition(part)` for each generated partition and updates the global variable `total`.

```
1:
2: start ← pos_start
3: if n < pos_start then
4:     start ← n
5: for k ← start downto 1 do
6:     part[pos] ← k
7:     if n ≠ k then
8:         call PARTNUM(n − k, pos + 1, part, k)
9:     else
10:        total ← total +1
11:        call usePartition(part)
```

Example usage: To generate all partitions of $n = 80$, initialize an array $part$ of length 80 and call:

$$\text{total} \leftarrow 0$$

$$\text{PARTNUM}(80, 1, part, 80)$$

After the call, `total` will contain the number of generated partitions, and each partition will have been processed by `usePartition(part)`.

In the presented pseudocode, the function *usePartition* executes additional computations that use the generated partition. As can be seen, a full partition is generated and can be used only when the end of the recursion is reached. Thus, to generate a new partition with k elements we have k recursive calls. To generate all $p(n)$ partitions we execute at most n recursive calls for each partition. Thus, in the worst case, we execute $np(n)$ recursive calls. To generate new partition, however, we do not need to have the previous partition already generated. Thus, the algorithm can be easily modified to generate partitions with some restrictions. This is one of its main advantages when the partitions are used in the generation process of more complex combinatorial algorithms.

Alternatively, we can start with the smallest possible value for k, which is 1. In this case, we can generate the partitions in lexicographic order. The starting value for k in the recursive call here will be the largest of the values between 1 and the previous value for k.

4.2. Non-Recursive Algorithm

Recursive algorithms are more inefficient compared to non-recursive ones. They provide great flexibility and can be easily modified to introduce some restrictions, for example to generate all partitions with less than k parts. In cases where fast generation of the partitions is more important than having an easily modifiable algorithm, a non-recursive approach to the generation is preferred. An algorithm that takes on average constant time to generate the next partition (CAT algorithm) is presented in [17]. It generates $P(n)_{rev}$ using sequence representation. It is shown that the given algorithm is significantly more effective than algorithms for generation with other representations.

Another approach, based on the algorithm in [17], uses successor function for the generation. Using such functions, we can generate the partitions in both lexicographic

and reverse lexicographic order. A successor function generates the next partition using the current one. Algorithms for calculating the successor of a given partition in $P(n)$ are presented in [14] and are based on the following Lemmas:

Lemma 3. *Let $a = \langle a_1, \ldots, a_k \rangle \in P(n)_{lex} \setminus \langle n \rangle$. If $k = 2$ and $a = \langle \frac{n}{2}, \frac{n}{2} \rangle$ for even n or $a = \langle \frac{n-1}{2}, \frac{n+1}{2} \rangle$ for odd n, the successor of a is $\langle n \rangle$. If $a_k = a_{k-1} < \lfloor \frac{n}{2} \rfloor$ or $a_k = a_{k-1} + 1 < \frac{n}{2}$, $k \geq 2$, then the successor of a in $P(n)_{lex}$ is $\langle a_1, \ldots, a_{k-2}, a_{k-1} + a_k \rangle$. In the other cases, the successor of a is*

$$S(\langle a_1, \ldots, a_k \rangle) = \langle a_1, \ldots, a_{k-2}, \underbrace{m, \ldots, m}_{\mu}, x \rangle,$$

where $m = a_{k-1} + 1$, $x = a_{k-1} + a_k - \mu m$, $\mu = \lfloor \frac{a_{k-1}+a_k}{m} \rfloor - 1$.

Proof. Obviously, $k = 1$ holds only for the last partition $\langle n \rangle \in P(n)_{lex}$, which has no successor, and therefore we exclude it. It is clear that for even n the only possible successor of $\langle \frac{n}{2}, \frac{n}{2} \rangle$ in lexicographic order is $\langle n \rangle$, the same is true for odd n and $a = \langle \frac{n-1}{2}, \frac{n+1}{2} \rangle$.

If $a_k = a_{k-1}$ or $a_k = a_{k-1} + 1$, $k \geq 2$, but not the above case, then $a_{k-1} + 1 > a_k - 1$ and therefore it is not possible to have an successor with k or more summands.

In the last case $a_{k-1} + 1 \leq a_k - 1$ and therefore if the successor is $b = \langle b_1, \ldots, b_s \rangle$, then $b_i = a_i$ for $1 \leq i \leq k-2$, $b_{k-1} = a_{k-1} + 1$, $s > k$ and $b_k + \cdots + b_s = a_k - 1$. The first partition of $a_k - 1$ in lexicographic order such that $b_k \geq a_{k-1} + 1 = m$, is $\langle \underbrace{m, \ldots, m}_{\mu}, x \rangle$ for $\mu = \lfloor \frac{a_{k-1}+a_k}{m} \rfloor - 1$. To verify the last partition, we calculate the sum

$$a_1 + \cdots + a_{k-2} + \mu m + x = a_1 + \cdots + a_{k-2} + \mu m + a_{k-1} + a_k - \mu m = n.$$

Furthermore, to check whether the parts in b are given in ascending order, we calculate $x = a_{k-1} + a_k - \mu m = a_{k-1} + a_k - (\lfloor \frac{a_{k-1}+a_k}{m} \rfloor - 1)m = m + ((a_{k-1} + a_k) \bmod m) \geq m$. □

Example 1. *Let $n = 12$ and $a = \langle 1, 1, 3, 3, 4 \rangle$. Then $k = 5$ and $a_4 + 1 = a_5$. The successor of a in $P(n)_{lex}$ is $a' = \langle 1, 1, 3, 7 \rangle$. Let us continue with the successors in $P(n)_{lex}$. The successor of a' is $a'' = \langle 1, 1, 4, 6 \rangle$. Now $\mu = \lfloor \frac{10}{4} \rfloor - 1 = 1$ and $x = 10 - 4 = 6$.*

Lemma 4. *Let $a = \langle a_1, \ldots, a_k \rangle \in P(n)_{rev} \setminus \langle \underbrace{1, \ldots, 1}_{n} \rangle$. Then the successor of a in $P(n)_{rev}$ is*

$$S(\langle a_1, \ldots, a_k \rangle) = \langle a_1, \ldots, a_{q-1}, \underbrace{m, \ldots, m}_{\mu}, x \rangle,$$

where q is the position of the right-most part with value $\neq 1$, $m = a_q - 1$, $x = n' - \mu m$, $n' = a_q + k - q$, and $\mu = \lfloor \frac{n'}{m} \rfloor$.

Proof. The last partition in this ordering is $\langle \underbrace{1, \ldots, 1}_{n} \rangle$ and therefore we exclude it. Take $a = \langle a_1, \ldots, a_k \rangle \in P(n)_{rev} \setminus \langle \underbrace{1, \ldots, 1}_{n} \rangle$. If a_q is the right-most part with value $\neq 1$, the first $q - 1$ parts in a and its successor $b = \langle b_1, \ldots, b_s \rangle$ are the same, and after that b contains as many parts $m = a_q - 1$, as possible. Since

$$n = \sum_{i=1}^{q} a_i + k - q = \sum_{i=1}^{q-1} a_i + \mu m + x,$$

we have $n' = a_q + k - q = \mu m + x$. Moreover, $0 \leq x < m$ and so $\mu m \leq n' < (\mu + 1)m$, which gives us that $\mu = \lfloor \frac{n'}{m} \rfloor$. Note that the part x can be equal to 0. □

Example 2. *Let $n = 12$ and $a = \langle 4, 3, 3, 1, 1 \rangle$. Then $k = 5$, $q = 3$, $a_q = 3$ and $n' = 5$. The successor of a in $P(n)_{rev}$ is $\langle 4, 3, 2, 2, 1 \rangle$. We see here that $m = 2$, $\mu = 2 = \lfloor \frac{5}{2} \rfloor$ and $x = 1$.*

Here, we will consider only the algorithm for generating $P(n)_{lex}$. As can be seen in Lemma 3, to generate the next partition we only consider the last two parts of the current partition. In the algorithm itself, we look at the values of a_k and a_{k-1}. We add 1 to the value of a_{k-1} and to retain the sum n we subtract 1 from a_k. The pseudocode in Algorithm 2 gives the successor function that generates the next partition in the lexicographically ordered set. This function can be used to generate the full set $P(n)_{lex}$.

Algorithm 2 SUCCESSOR$(n, part, k)$

Input:
n – integer being partitioned
$part$ – array containing the current integer partition of n
k – index of the last summand in the array ($0 \leq k < n$)
Output:
Modifies $part$ to the next partition in lexicographic order and calls `usePartition(part)`.

```
1:
2:  if k ≥ n or k < 0 then
3:      return                                   ▷ Invalid index
4:  y ← part[k] − 1
5:  k ← k − 1
6:  x ← part[k] + 1
7:  while x ≤ y do
8:      part[k] ← x
9:      y ← y − x
10:     k ← k + 1
11:     if k ≥ n then
12:         return                               ▷ Invalid index
13: part[k] ← y + x
14: call usePartition(part)
```

4.3. Ranking and Unranking Algorithms

Other functions, that are typically discussed with algorithms for generation of basic combinatorial objects, are ranking and unranking functions. Such functions can be used for partitioning the set $P(n)$ into close to equal subsets - a key part in the parallelization of algorithms. Here, we present two sets of ranking and unranking functions for $P(n)_{lex}$ and $P(n)_{rev}$.

Using the recurrence relations, given in Lemma 1 and Lemma 2, we can calculate the rank of a given partition $\langle a_1, \ldots, a_m \rangle$ by using $\widehat{p}(n, k+1)$ and $p^*(n, k-1)$ for $P(n)_{lex}$ and $P(n)_{rev}$, respectively. A formula for computing the rank of partition in $P(n)_{lex}$ is given in (3). Analogously, a formula for computing the rank of partition in $P(n)_{rev}$ is given in (4). We write the partition with a leading zero for both orderings. This is easily avoided in the algorithms themselves by changing the execution order of some computations.

$$rank_{lex}(\langle 0, a_1, \ldots, a_m \rangle) = p(n) - 1 - \sum_{i=0}^{m-1} \widehat{p}(n - \sum_{j=0}^{i} a_j, a_{i+1} + 1) \tag{3}$$

$$rank_{rev}(\langle 0, a_1, \ldots, a_m \rangle) = p(n) - 1 - \sum_{i=0}^{m-1} p^*(n - \sum_{j=0}^{i} a_j, a_{i+1} - 1). \tag{4}$$

In practice, we use the number of parttitions succeeding the given partition. The pseudocode in Algorithm 3 give the ranking function for the lexicographic order $P(n)_{lex}$, where a 2-dimensional array N contains the values for $\widehat{P}(n,k)$:

Algorithm 3 RANK_LEX$(part, n, k)$

Input:
$part$ – array containing the current integer partition of n
n – integer being partitioned
k – number of summands in the current partition
$N[i][j]$ – precomputed table: number of partitions of i whose smallest part is at least j
Output:
Returns the lexicographic rank of the given partition.

```
1:
2: rank ← N[n][0] − 1                          ▷ Initialize to largest possible rank
3: for i ← 0 to k − 1 do
4:     x ← part[i]
5:     rank ← rank − N[n][x + 1]
6:     n ← n − x
7: return rank
```

Analogously, we can use $\widehat{p}(n-k,k)$ that gives the number of partitions of n with fixed first part k in the unranking function for $P(n)_{lex}$. If the given rank is less than this value, then k is the first part of the corresponding partition. Otherwise, we continue for the next possible value of k and change the value of the rank correspondingly. The pseudocode in Algorithm 4 give the unranking function for the lexicographic order, where a 2-dimensional array N contains the values for $\widehat{p}(n,k)$:

Algorithm 4 UNRANK_LEX$(n, rank, part)$

Input:
n – integer to be partitioned
$rank$ – integer rank ($0 \leq rank < N[n][0]$)
$N[i][j]$ – precomputed table: number of partitions of i whose first part is at least j
Output:
The array $part$ contains the integer partition corresponding to the given rank.
The function returns the index of the last summand (i.e., length -1).

```
1:
2: if rank < 0 or rank ≥ N[n][0] then
3:     return 0                                          ▷ Invalid rank
4: m ← 1
5: len ← 0
6: while n > 0 do
7:     for x ← m to n do
8:         count ← N[n − x][x]          ▷ Number of partitions of n with first part x
9:         if rank < count then
10:            part[len] ← x
11:            len ← len + 1
12:            n ← n − x
13:            m ← x
14:            break
15:        else
16:            rank ← rank − count
17: return len − 1
```

The unranking function for the reverse lexicographic order is analogous. We use a 2-dimensional array that contains the values of $p^*(n,k)$ and we traverse k in decreasing

order starting with n. When we add k to the current partition, the next possible value, however, is either k or the new value of n (which is $n - k$), depending on which is the smaller of the two. To parallelize an algorithm for generating all partitions, we can use the unranking function to set the first partition for subsets that will be generated in parallel.

4.4. Generating Partitions with Restrictions

There are cases where we need to generate the partitions of integer n with some restrictions. The most popular restrictions in the literature are generating the set of partitions of n into less than k parts and generating the partitions n having parts with values less than k. In practice, partitions in the set containing parts with values less than k are conjugated to partitions in the set, that contains partitions with less than k parts. Thus, if we generate one set, we can easily generate the other as their conjugate partitions. When working with the recursive algorithm, we can easily modify the algorithm to generate partitions containing parts less than k by setting the value of *pos_start* to k. Other limitations can also be easily added to the recursive algorithm. Such is not the case for the non-recursive algorithm since it generates the next partition based on the current one.

5. Parallel Implementation

In this section we present different approaches to parallelize known algorithms for generating integer partitions. The most significant stage of developing parallel implementation is the way the problem is divided into smaller problems that will be solved in parallel. Inefficient division can result in slower execution times. Another crucial moment in the implementation is choosing appropriate data types and minimizing data sharing. In OpenMP, data sharing is emulated by writing in global memory and synchronization. In the current work, we present two parallel strategies for the generation of integer partitions based on two different algorithms. We use OpenMP for the implementations, while simultaneously commenting on some of its advantages and challenges when developing parallel programs and algorithms.

5.1. Recursive Algorithm

Let us consider the parallel algorithm with a recursive approach for breaking down a number n into a sum of positive integers. For the implementation, we use the *task* construct. A task is only created at the first level (position) of the integer partition. The partitions are written in a structure in order to be shared in the recursive step between tasks. To evaluate the efficiency of the decomposition of the problem and to calculate the total number of partitions, we also use arrays for counting the number of tasks and the number of computed distinct partitions per task. The main steps in the parallelization of the algorithm are the following:

1. Generate a set ω consisting of partitions of n with two elements.
2. In parallel, for each element of ω, generate all partitions of the second element with the function *partnum*.

Visualization of one possible outcome of the algorithm for $n = 5$, executed in single-threaded mode and with four threads simultaneously (with the master thread as 0), is shown in Figure 2. The integer $n = 5$ can be partitioned in seven distinct ways. Based on everything discussed so far, it is evident that the first partition for $n = 5$ is 5. When the code runs and reaches the line 5, since $n = 5$ and $k = 5$ do not meet the conditions, the execution proceeds to *'else'* . Further execution of the code recursively applies the same partitioning algorithm to $(n - k)$. This process continues, decreasing the value of k until it reaches 1.

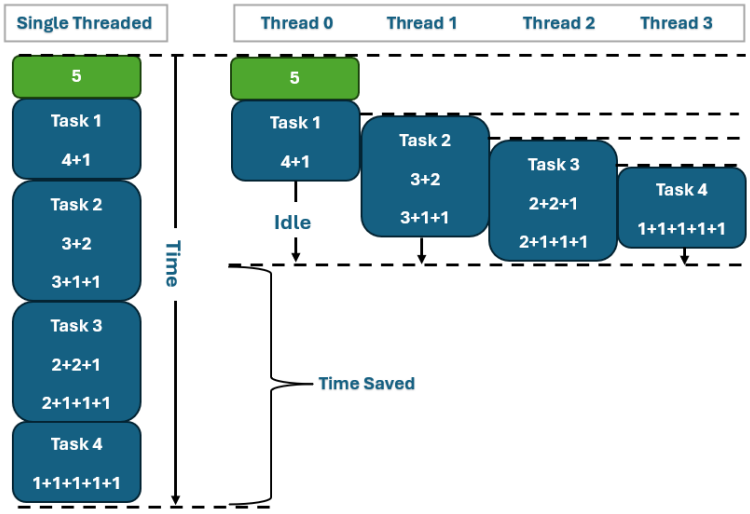

Figure 2. Example of task workload and parallelization for $n = 5$.

The arrays defined in this way lead to poor scaling. Since a one-dimensional array is used, independent data elements may share the same cache line, causing each data update to trigger a continuous shift of the cache line between threads. This is called *false Sharing*, illustrated in Figure 3.

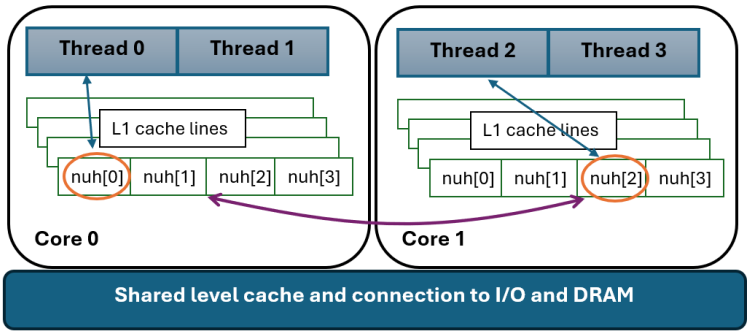

Figure 3. Example of false sharing.

We can use padded arrays as a solution to this issue, ensuring that the elements occupy distinct cache lines. This is achieved by defining two-dimensional arrays to count the number of tasks and the number of computed distinct partitions per task. Using padded arrays requires a deep understanding of the cache size and architecture. If the code is run on a machine with a different cache line size, performance can degrade significantly. This means that padded arrays are not a portable solution. The solution for addressing both false sharing and portability is to use the OpenMP *threadprivate* directive. This directive allows named common blocks and variables to be private to a thread while remaining global within that thread. In other words, threadprivate preserves global scope within each thread.

The directive *#pragma omp threadprivate (list)* must appear after the declaration of the listed variables or common blocks. Each thread receives its own copy of the variable or common block, ensuring that data written by one thread is not visible to other threads. The list is a comma-separated collection of file-scope, namespace-scope, or static block-scope variables that do not have incomplete types.

In this version, the necessary modifications are made after the inclusion of the libraries, as shown in Algorithm 5:

Algorithm 5 PARTNUM_PARALLEL($n, pos, pos\_start, part$)

Input:
n – integer to be partitioned
pos – index in *part* where the new summand will be placed
pos_start – maximum allowed value for current summand
part – array representing the current partition
Output:
Generates all partitions of n recursively in non-increasing order
Uses parallel tasks for the first level of recursion
Calls `usePartition(part)` for each generated partition

```
start ← min(n, pos_start)
for k ← start downto 1 do
    part[pos] ← k
    if n ≠ k then
        if pos = 1 then                          ▷ First level: create parallel tasks
            parallel task:
                Create a local copy part_local of part
                PARTNUM_PARALLEL(n − k, pos + 1, k, part_local)
        else
            PARTNUM_PARALLEL(n − k, pos + 1, k, part)
    else
        Increment thread-local counter NumPartT
        call usePartition(part)
```

Example usage: To generate all partitions of n, initialize the counters `total` and thread-local `NumPartT`(Globally defined by *#pragma omp threadprivate (NumPartT)*). Only the master thread is active in the parallel region and it calls:

$$\text{PARTNUM\_PARALLEL}(n, 1, n, part)$$

After the parallel execution, accumulate the counts from all threads:

$$\text{total} \leftarrow \sum \text{NumPartT across all threads}$$

Each partition has been processed by `usePartition(part)`.

The thread-specific copy of the variable NumPartT is used to count the number of computed distinct partitions per task. The counting of computed distinct partitions occurs in the partnum function, within the final else condition. The final step is to sum the count of computed distinct partitions that occur per thread. This summation is performed in the main function using the *#pragma omp critical* directive for synchronization, ensuring protected access to prevent race conditions. The use of the critical directive defines this version of the algorithm, referred to hereafter as *Recursion Critical*.

5.2. Parallelization Through Subsets

Recursive algorithms are typically less effective than non-recursive ones. Therefore, we also consider parallelization of the non-recursive algorithm for generating the set of all partitions in increasing order shown in [14]. One approach for parallelization uses an unranking function for the set. The main idea is to partition the set $P(n)$ into t equal subsets, where t is the number of threads, and each thread would calculate $per\_th = p(n)/t$ partitions. If *per_th* is not an integer, then each thread except one generates $per\_th = \lfloor p(n)/t \rfloor$ and a single thread will generate $p(n) - (t-1) * per\_th$ partitions. The algorithm has the following main steps:

1. Calculate $p(n)$ and a 2-dimensional array N, where $N[n][k]$ contains the number of partitions of n with all parts greater than or equal to k;
2. Calculate *per_th* and the rank of the first partition ($th\_id * per\_th$, where *th_id* is the id of the current thread) for each thread;
3. Generate the first partition using the unranking function;
4. Generate next partition until *per_th* partitions have been generated.

Let us now comment on each step in more detail. We use $p(n)$ to partition the set of all partitions into subsets. Here, the 2-dimensional array N is used in the unranking function that generates the first partition in each subset. The functions to calculate $p(n)$ and array N are executed once before the parallel region and take less than 5% of the execution time. The rest of the steps are executed in parallel by multiple threads. For the parallelization, we use a *parallel for* construction to divide the computational work among the threads. Each thread executes the computations for a single iteration of the for cycle.

The second step consists of calculating the number of partitions that each thread needs to generate. The presented division method distributes the work almost equally - only one thread generates more partitions. For large enough $p(n)$, the difference in the workload will be negligible. For example, $p(20) = 627$ and if we use 24 threads then each thread will generate 26 partitions and the last thread will generate 3 more partitions.

After we have the initial partitions using the unranking function, we generate the next partition [14] until each thread has generated *per_th* partitions. A global variable is updated to calculate the total number of partitions using the *critical* clause. This update is executed only once per thread and therefore has minimal impact on the execution time. The pseudocode in Algorithm 6 gives the parallel version.

Algorithm 6 SUBSETS(n)

Input:
n – integer to be partitioned
TH – number of parallel threads
$N[n][n]$ – precomputed partition table
Output:
Generates all partitions of n in parallel
Calls `usePartition(a)` for each generated partition

```
1:
2:  compute_partition_table(n)
3:  total ← N[0][n]                                   ▷ total number of partitions
4:  sum ← 0
5:  parallel for each thread i = 0 to TH − 1:
      per_th ← total/TH
      beg_local ← i× per_th
      last ← total −(TH − 1)× per_th
      if i = TH − 1, then per_th ← last
      num_local ← 1
      initialize local array a_local[1..128]
      k ← UNRANK(n, beg_local, a_local)
6:  while num_local < per_th do
7:    y ← a_local[k] − 1
8:    k ← k − 1
9:    x ← a_local[k] + 1
10:   while x ≤ y do
11:     a_local[k] ← x
12:     y ← y − x
13:     k ← k + 1
```

Algorithm 6 *Cont.*

```
14:     a_local[k] ← y + x
15:     num_local ← num_local + 1
16:     call usePartition(a_local)
17: critical section:
      sum ← sum + num_local
```

5.3. Computational Aspects

Let us consider some of the computational aspects of the presented parallel implementations. The integer partition algorithms were parallelized using OpenMP multi-threading in a shared-memory environment. For the recursive algorithm, the computation naturally forms a search tree. We applied task-level parallelism, assigning independent branches to different threads, enabling concurrent exploration of disjoint subproblems. For the iterative, non-recursive algorithm, parallelization was achieved over independent loop iterations, allowing multiple threads to process separate portions of the solution space simultaneously. Overall, parallelism is achieved through concurrent execution of tasks rather than through multi-process execution or traditional data-level parallelism. Due to the nature of the algorithms themselves, no GPU acceleration or SIMD vectorization was employed.

Both versions - using recursion and subsets - demonstrate strong scalability. For the implementation using recursion, we have improved the initial scalability issue, that occurs when counting the total number of partitions, by using the *threadprivate* directive. In practice, when using the partitions in algorithms for constructing combinatorial objects we are not interested in the total number of partitions. Furthermore, in both variants individual threads generate a close to equal number of partitions, ensuring efficient load balancing and optimal resource utilization. In the recursive version this is possible through the use of tasks. The benefits of their use become more evident, since they provide great flexibility and ease of use. The problem is divided into varying numbers of tasks, either larger or smaller. The number of the tasks depends on the level of the recursion on which the parallelization occurs. While increasing the number of tasks leads to a more balanced work distribution, it also introduces overhead from creating additional tasks. Moreover, the sequential portion of the program grows, which reduces scalability. Therefore, a compromise must be made, taking the available hardware resources into account. In the implementation using subsets, the work distribution is generally satisfactory in the presented cases. However, it can be made even more uniform as follows: the remaining m partitions assigned to the last thread can be redistributed among the last m threads. The good scalability in both cases is also seen in the experimental results, given in Section 6.

Another main aspect of parallel algorithms is the communication between computational units. The proposed implementations do not require communication between threads. The *Recursion Critical* algorithm explicitly illustrates how to prevent false sharing, thereby enhancing performance and independence of the computations. In the version with subsets, there is no communication. The proposed implementations also are not affected by some hardware specifications such as cache lines and NUMA nodes.

Lastly, we can consider the use of extended vector registers. Such optimization is not applicable here since additional data is not used in the generation process. Thus, they do not directly enhance the performance of the number partitioning algorithms that are described. However, extended registers can be used in additional computations that can be executed in the generation algorithms for different combinatorial objects.

6. Experimental Results

In this section we present some experimental results that are used to evaluate the efficiency of the presented parallel implementations. All computations are executed on a system Fujitsu Primergy RX 2540 M4 with NVIDIA Tesla V100 32GB, 128 GB RAM, CPU 2x Intel Xeon Gold 5118 2.30GHz 24 core, 2x800GB SSD, 3*12TB HDD and a Red Hat Enterprise operating system version 7.9. For the compilation we use GCC 9.2 compiler with partial support for OpenMP 5.0.

6.1. Efficiency Evaluation

Table 2 presents execution times in seconds for the recursive implementation. The first column gives the value of n for which we calculate the partitions and the rest give the execution times in seconds for $TH = 1, 2, 4, 8, 16, 24, 32, 48$. Here, we use the *Recursion Critical* implementation. As can be seen, the execution times decline between 1.5 and 2 times when we double the number of threads for $TH \leq 16$. For $TH = 32$ we still have faster execution time in most cases. However, for $TH = 48$ the execution time is greater compared to the case with 24 threads. There are two main reasons for this - additional overhead in the recursive implementation and the hardware system itself.

Let us also consider the standard deviation σ given by the equation:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{t}(t_i - ave)^2}{t}},$$

where t is the total number of executions, t_i is the time in seconds for the i-th execution and ave is the average execution time. We have calculated the standard deviation for the experimental results presented in Tables 2 and 3. In most cases the standard deviation is less than 3% of the average execution time. The exceptions are the cases with more than 16 threads and computational time less than 0.06 seconds. The standard deviation in those cases is between 10% and 30% of the average execution time. This is due to the multi-tasking system which is responsible for processing device interrupts, scheduling other programs, etc. Moreover, memory partitioning and allocation has an impact.

Table 2. Execution times for parallel implementation using recursion.

n	TH = 1	TH = 2	TH = 4	TH = 8	TH = 16	TH = 24	TH = 32	TH = 48
80	0,311	0,190	0,095	0,052	0,039	0,043	0,051	0,081
100	4,226	2,569	1,278	0,693	0,442	0,410	0,404	0,508
120	45,093	27,298	13,649	7,256	4.28	3,726	3,765	4,387
140	400,583	241,667	121,289	64,303	36,59	31,049	31,322	36,272

Table 3. Execution times for parallel implementation using subsets.

n	TH = 1	TH = 2	TH = 4	TH = 8	TH = 16	TH = 24	TH = 32	TH = 48
100	0,352	0,186	0,109	0,058	0,031	0,022	0,024	0,025
120	3,550	1,886	1,076	0,574	0,304	0,215	0,202	0,154
140	29,767	15,694	8,937	4,761	2,498	1,693	1,560	1,203

Table 3 presents the execution times for the implementation using subsets. The first column again gives the value of n for which we calculate the partitions and the rest give the execution times in seconds for $TH = 1, 2, 4, 8, 16, 24, 32, 48$. Here we have similar results. When increasing the number of threads up to 24 the decrease in execution times is close to the maximal possible. This indicates to the good scalability of the parallelization of the implementation. When the number of threads exceeds the number of physical cores, we still have a decrease in execution time. The speed-up, however, is less than the maximum

possible for $TH \leq 24$. This again shows the significance of the hardware. The standard deviation σ for this implementation is again less than 3% in most cases. This shows that there is minimal variation in the computational times with repeated executions.

To assess the benefits of parallelization, we evaluated speed-up and efficiency for both the recursive and iterative implementations. Speed-up S(p) is defined as the ratio between sequential and parallel execution times, i.e., $(S(p) = T(1)/T(p))$, $S(p)$ is the speedup with 'p' threads, $T(1)$ is the execution time of the sequential algorithm, and $T(p)$ is the execution time for the parallel algorithm using 'p' threads. Efficiency measures the utilization of resources.The formula for parallel efficiency (E) is $E = S(p)/p$ where S is the achieved speedup with the parallel implementation and p is the number of threads.

Figures 4 and 5 illustrate the observed behavior. For both algorithms, speed-up increases as the number of threads grows, and efficiency remains high up to 24 threads. This corresponds to the number of physical CPU cores available on our server. However, the speed-up and efficiency of the implementation using subsets is overall greater than the implementation using tasks. When the number of threads exceeds 24, performance gains are limited or even reduced, since the system relies on hyper-threading (up to 48 logical threads). Hyper-threading provides only partial overlap of execution units and does not double computational throughput, which explains the decline in efficiency. Overall, the results confirm that the OpenMP implementations achieve good scalability on physical cores and highlight the expected performance saturation when relying on hyper-threading.

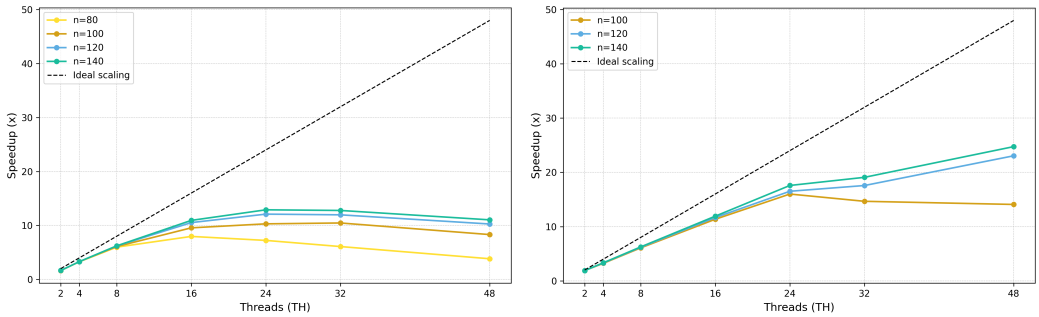

Figure 4. (**a**) Speedup with parallel implementation of the recursive algorithm with *tasks*. (**b**) Speedup with parallel implementation using subsets and *parallel for* construct.

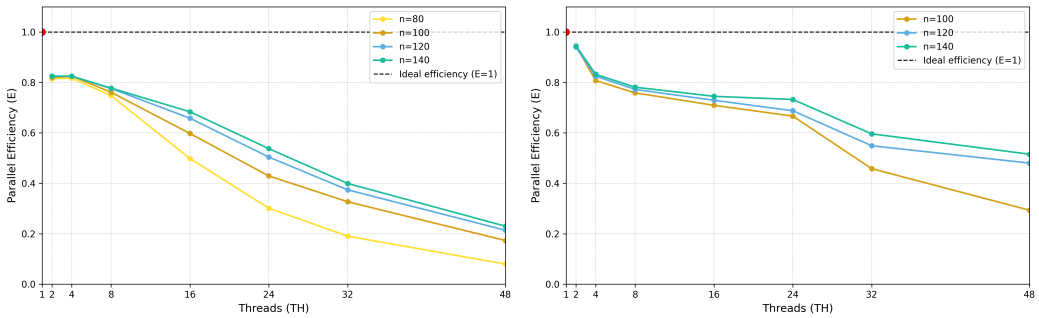

Figure 5. (**a**) Efficiency for the parallel implementation of the recursive algorithm with *tasks*. (**b**) Efficiency for parallel implementation using subsets and *parallel for* construct.

6.2. The Sequence $\langle a_1, \ldots, a_k \rangle$, Where $a_1^3 + \cdots + a_k^3 = (a_1 + \cdots + a_k)^2$

One interesting sequence is $(a_1, \ldots, a_k)$, for which the Diophantine equation $a_1^3 + \cdots + a_k^3 = (a_1 + \cdots + a_k)^2$ holds. Such sequences may include repeated values and negative integers. The research around such sequences includes many different problems. In [10],

sequence where all values are positive, are denoted as positive CS-set. A CSn-set denotes the set when the sequence includes n elements. One CSn-set is the sequence of natural numbers $(1, 2, \ldots, n)$ since $1^3 + \cdots + n^3 = (1 + \cdots + n)^2$. In [10] it is proven that for every natural n there is precisely one positive CSn-set with distinct elements. Other problems connected to CS-sets also include extending such sets, finding zero sum sets, finding good bounds on the number of positive CSn-sets, etc. One approach to extending such sets uses the concept of *bag product*, described in [27]. The bag product of two integer sets consists of all products with one element from each set. The bag product of CS-set is also a SC-set. In [10] it is also proven that a CS-set can be extended by an element if and only if its sum is equal to $z(z-1)/2$ for some integer z. Furthermore, the set is extended by adding exactly z. Bounds connected to the given Diophantine equation are discussed in [28].

As can be seen, the sequence $(a_1, \ldots, a_k)$, for which the Diophantine equation

$$a_1^3 + \cdots + a_k^3 = (a_1 + \cdots + a_k)^2 = n^2 \tag{5}$$

holds is also a partition of $n = a_1 + \cdots + a_k$. So we look at this equation from a different perspective. Instead of counting the number of sets with length n for which the equation holds, we count the number of sets whose sum is equal to n and for which $a_1^3 + \cdots + a_k^3 = n^2$. We denote this number by $P(n)_{cube}$. Using the parallel implementation, we calculate $P(n)_{cube}$ for $n = 1, \ldots, 220$. The computations took less than 12 hours using the parallel implementation with subsets on the same system. The values of $P(n)_{cube}$ also form an integer sequence. The computational results show that for $n = 7$, $n \leq 220$ and $n \equiv 2 \pmod 3$ there are no partitions for which the equation holds. Indeed, according to Fermat's little theorem, $a_1^3 + \cdots + a_k^3 \equiv a_1 + \cdots + a_k = n \pmod 3$ and so $n^2 \equiv n \pmod 3$. This gives us that $n \equiv 0$ or $n \equiv 1 \pmod 3$. In all other cases and $n < 220$, there are partitions of n, for which (5) holds. The sequence $\{1, 0, 1, 1, 0, 1, 0, 0, 2, 1, 0, 2, 1, 0, 2, 5, 0, 3, 4, 0\}$ contains the first 20 members of $P(n)_{cube}$, which gives the number of partitions, that are solutions of (5). The full sequence for $n = 1, \ldots, 220$ can be found in [29].

7. Conclusions

In the current work, we consider algorithms for generating partitions of integer n, modification that generate partitions with restrictions, and ranking and unranking functions for integer partitions in both lexicographic and reverse lexicographic order. We present two parallel implementations for generating partitions . Another concept for parallelization, presented in [13], uses n processors, where each process does not compute the full partition, but just a part of it. In the presented implementations, a thread generates a full partition that can then be used for additional computations. We compare their execution times to the sequential implementations. The experimental results show that a parallelization using the unranking function and subsets of $P(n)$ has better scalability compared to the parallelization of the recursive algorithm. The C/C++ code of the algorithms can be found in [29] and the parallel functions in the Appendix A.

On multi-node computer systems, a distributed memory interface like MPI, or a hybrid MPI+OpenMP setup, is required. The implementation of the non-recursive version will be much more natural. Additionally, the unranking function enables the generation of integer partitions using the CUDA interface for GPU accelerators. This approach is particularly efficient in scenarios where further computations with the generated partitions are not required, offering both speed and convenience. Research in this area remains as future work. We also present a new integer sequence $P(n)_{cube}$ that gives the number of partitions of n for which $n = a_1 + \cdots + a_k$ and $n^2 = a_1^3 + \cdots + a_k^3$.

Author Contributions: Conceptualization, I.B.; methodology, I.B.; software, D.B. and M.P.-G.; validation, D.B. and M.P.-G.; formal analysis, I.B. and D.B.; investigation, D.B. and M.P.-G.; resources, D.B. and M.P.-G.; data curation, D.B. and M.P.-G.; writing—original draft preparation, I.B., D.B. and M.P.-G.; writing—review and editing, I.B., D.B. and M.P.-G.; visualization, D.B. and M.P.-G.; supervision, I.B.; project administration, I.B.; funding acquisition, I.B., D.B. M.P.-G. All authors have read and agreed to the published version of the manuscript.

Funding: The work was partially supported by the Centre of Excellence in Informatics and ICT under the Grant No BG16RFPR002-1.014-0018-C01, financed by the Research, Innovation and Digitalization for Smart Transformation Programme 2021-2027 and co-financed by the European Union. The work of the second author is partially supported by the Bulgarian National Science Fund under Contract No KP-06-H62/2/13.12.2022. The work of the third author has been accomplished with the financial support of the Bulgarian MES by the Grant No. DO1-98/26.06.2025 for NCHDC – part of the Bulgarian National Roadmap on RIs.

Data Availability Statement: The sequence $P(n)_{cube}$ for n up to 220, that contains the number of partitions of n, for which the sum of the cubes of the parts is equal to the square of n, is available at https://github.com/IntegerPartition accessed on 13 October 2025.

Conflicts of Interest: The authors declare no conflicts of interest.

Appendix A. The C/C++ Code for Parallel Algorithms

Parallelization through recursive algorithm using tasks (Algorithm 5):

```
#include <omp.h>
...
struct  partt { int a[256];}part;
unsigned long long int NumPartT=0,total = 0;
#pragma omp threadprivate(NumPartT)
void partnum(int n, int pos,int pos_start, partt &part)
{
   int start = pos_start;
   if (n < pos_start) start = n;
   for (int k = start; k >= 1; k--){
      part.a[pos] = k;
      if (n != k) {
         if(pos==1){
            #pragma omp task
            { //Creates independent parallel tasks
             partt p_local;
             for(int i=0; i<=pos; i++){ p_local.a[i]=part.a[i];}
             partnum(n - k, pos + 1,k,p_local);
             }
            }else {partnum(n - k, pos + 1,k,part);}
        }else {
           NumPartT++;
           usePartition(part.a);
           }
       }
}
void mainRec_partnum(int n){
   total=0; NumPartT = 0;
   #pragma omp parallel
   {
```

```
      NumPartT = 0;
      #pragma omp single
      { partnum(n,1,n,part);}
      #pragma omp critical
      { total+=NumPartT;}
  }
}
```

Parallelization through subsets (Algorithm 6):

```
void subsets(int n) {
    omp_set_num_threads(TH);
    compute_partition_table(n); \\initialize N[n][n]
    unsigned long long int total = N[0][n];
    unsigned long long int sum = 0;

#pragma omp parallel for
    for (int i = 0; i <TH; i++) {
        unsigned long long int num_local = 1;
        int a_local[128];
        unsigned long long int per_th = total / TH;
        unsigned long long int beg_local = i * per_th;
        unsigned long long int last = total - (TH - 1) * per_th;
        unsigned long long int  k = unrank(n, beg_local,a_local);
        if (i == (TH - 1)) { per_th = last; }
        while (num_local < per_th) {
            int y = a_local[k] - 1;
            k--;
            int x = a_local[k] + 1;
            while (x <= y) {
                a_local[k] = x;
                y = y - x;
                k++;
            }
            a_local[k] = y + x; num_local++;
            usePartition(a_local);
        }
#pragma omp critical
        {
            sum += num_local;
        }
    }
}
```

References

1. Bivins, R.L.; Metropolis, N.; Stein, P.R.; Wells, M.B. Characters of the symmetric groups of degree 15 and 16. *Math. Comput.* **1954**, *8*, 212–216.
2. Comet, S. Notations for partitions.*Math. Tables Other Aids Comput.* **1955**, 143–146.
3. Martin, S. *Schur Algebras and Representation Theory*; Cambridge University Press: Cambridge, UK, 1993.

4. Temperley, H.N.V. Statistical mechanics and the partition of numbers I. The transition of liquid helium. *Proc. R. Soc. London. Ser. Math. Phys. Sci.* **1949**, *199*, 361–375.
5. Knuth, D. *Art of Computer Programming, Volume 4, Fascicle 4, The: Generating All Trees–History of Combinatorial Generation*; Addison-Wesley Professional: Boston, MA, USA, 2006.
6. Kreher, D.; Stinson, D. *Combinatorial Algorithms: Generation, Enumeration, and Search*; CRC Press: Boca Raton, FL, USA, 1999.
7. Pashinska-Gadzheva, M.; Bakoev, V.; Bouyukliev, I.; Bikov, D. Optimizations in computing the algebraic normal form transform of Boolean functions. In Proceedings of the 2021 International Conference Automatics and Informatics (ICAI), Varna, Bulgaria, 30 September–2 October 2021; pp. 288–291.
8. Pashinska-Gadzheva, M.; Bouyukliev, I. About Methods of Vector Addition over Finite Fields Using Extended Vector Registers. In *Proceedings of the 14th International Conference on Large-Scale Scientific Computing, Sozopol, Bulgaria, 5–9 June 2023;* Lirkov, I., Margenov, S., Eds.; Springer Nature: Cham, Switzerland, 2023; pp. 427–434.
9. Savage, C. Gray code sequences of partitions. *J. Algorithms* **1989**, *10*, 577–595.
10. Barbeau, E.; Samer S. Sum of cubes is square of sum. *Notes Number Theory Discret. Math.* **2013**, *19*, 1–13.
11. The On-Line Encyclopedia of Integer Sequences (OEIS). Available online: https://oeis.org/ (accessed on 1 July 2025).
12. Bouyukliev, I.; Hristova, M. About an approach for constructing combinatorial objects. *Cybern. Inf. Technol.* **2018**, *18*, 44–53.
13. Akl, G.S. Parallel algorithms for generating integer partitions and compositions. *J. Comb. Math. Comb. Comput.* **1993**, *13*, 107–120.
14. Kelleher, J.; O'Sullivan, B. Generating all partitions: A comparison of two encodings. *arXiv* **2009**, arXiv:0909.2331.
15. McKay, J.K.S. Partitions in natural order. *Commun. ACM* **1970**, *13*, 52.
16. Page, E.; Wilson, L. *An Introduction to Computational Combinatorics*; Cambridge University Press: Cambridge, UK, 1979.
17. Stojmenović, I.; Zoghbi, A. Fast algorithms for genegrating integer partitions. *Int. J. Comput. Math.* **1998**, *70*, 319–332.
18. Actor, A. Infinite products, partition functions, and the Meinardus theorem. *J. Math. Phys.* **1994**, *35*, 5749–5764.
19. Andrews, G.; Kimmo E. *Integer Partitions*; Cambridge University Press: Cambridge, UK, 2004.
20. Santos, J.; Matte, M. A new approach to integer partitions. *Bull. Braz. Math. Soc. New Ser.* **2018**, *49*, 811–847.
21. Tran, M.; Murthy, M.V.N.; Bhaduri, R. On the quantum density of states and partitioning an integer. *Ann. Phys.* **2004**, *311*, 204–219.
22. Merca, M. New upper bounds for the number of partitions into a given number of parts. *J. Number Theory* **2014**, *142*, 298–304.
23. Oruç, A. On number of partitions of an integer into a fixed number of positive integers. *J. Number Theory* **2016**, *159*, 355–369.
24. Chen, S. Arithmetical properties of the number of t-core partitions. *Ramanujan J.* **2009**, *18*, 103–112.
25. Fathima, S.N.; Pore, U. Some infinite families of congruences for t-core partition functions. *Acta Math. Hung.* **2023**, *170*, 351—366.
26. Mattson, T.; Yun, H.; Koniges, A. *The OpenMP Common Core*; The MIT Press Cambridge: London, UK, 2019.
27. Mason, J.J. Generalizing "Sums of cubes equals to squares of sums". *Math. Gaz.* **2001**, *85*, 50–58.
28. Beardon A.F. Bounds on sums of cubes. *Math. Gaz.* **2025**, *109*, 39–46.
29. Sequence $P(n)_{cube}$. Available online: https://github.com/IntegerPartition (accessed on 10 October 2025).