Parallel Algorithms for Multinode Allocation in IoT Networks

Paulo Zimpel[1], **Luiz Xavier**[1], **Murilo Táparo**[12], **Samuel Ferraz**[1], **Bianca Dantas**[1], **Ricardo Santos**[1]

[1]College of Computing – Federal University of Mato Grosso do Sul (UFMS)
Cidade Universitária S/N – 79070-900 – Campo Grande – MS – Brazil

[2]Faculty of Exact Sciences – Federal University of Grande Dourados (UFGD)
Dourados - MS - Brazil

{paulo.zimpel, xavier_felipe}@ufms.br, murilotaparo@ufgd.edu.br,

{samuel.ferraz, bianca.dantas, ricardo.santos}@ufms.br

Abstract. *The Multinode Allocation Problem in Internet of Things networks is a computationally intensive optimization problem that consists of assigning jobs to subsets of network nodes while satisfying processing capacity, bandwidth, and latency constraints. As the network scale and allocation demands increase, exploration of candidate solutions becomes computationally prohibitive, motivating the use of heuristic and parallel approaches. The Demand-Index Multinode Allocation heuristic (DI-MNA) was originally proposed to reduce the computational cost of the allocation while obtaining competitive solutions. This paper presents parallel approaches of the DI-MNA heuristic targeting CPU and GPU architectures. Experimental results demonstrate the effectiveness of the proposed parallelization strategies in reducing the computational cost of DI-MNA while preserving the quality and feasibility of the resulting allocations.*

Resumo. *O Problema de Alocação de Multiplos Nós em redes da Internet das Coisas é um problema de otimização que consiste em atribuir tarefas a subconjuntos de nós da rede, respeitando suas restrições. À medida que a escala da rede e as demandas de alocação aumentam, a exploração das soluções torna-se inviável, motivando o uso de abordagens heurísticas e paralelas. A heurística Demand-Index Multinode Allocation (DI-MNA) foi originalmente proposta para reduzir o custo computacional do processo de alocação obtendo soluções competitivas sob múltiplas restrições. Este artigo apresenta abordagens paralelas da heurística DI-MNA direcionadas a arquiteturas de CPU e GPU. Os resultados experimentais demonstram a eficácia das estratégias de paralelização propostas na redução do custo temporal da DI-MNA, preservando a qualidade das alocações resultantes.*

1. Introduction

Among the main paradigms in distributed systems, the *Internet of Things* (IoT) framework is commonly regarded as one of the most complex in terms of heterogeneity and coordination [Savaglio et al. 2020]. IoT networks employ large collections of devices

that may vary in computational and communication capabilities. For this reason, assigning processing nodes to different workloads becomes a constrained optimization problem, in which coordination must simultaneously account for the workload itself and the underlying network's capabilities. As a guide, IoT systems commonly rely on *Quality of Service* (QoS) metrics to measure how appropriate a task is in relation to a candidate allocation [Muhamad et al. 2023].

The Multinode Allocation in IoT networks emerges as the means of supplying one or more processing nodes for a job with specific demands. The underlying network can be seen as a set of nodes that vary in processing resources and bandwidth, each one connected to other nodes subject to an associated latency [Zahoor and Mir 2021]. The processing tasks (jobs), on the other hand, have demands such as the minimum amount of resources, bandwidth, and a maximum supported latency. Consequently, a QoS policy is established to evaluate the quality of a candidate allocation. Therefore, the *Multinode Allocation* (MNA) problem comprises allocating a set of jobs on a given network, optimizing for a specific QoS metric.

For this purpose, several conventional solutions can be applied to the MNA problem, such as exhaustive searches or branch and bound algorithms. However, as the input datasets continue to grow, the evaluation of the complete search space for each job may become unfeasible, which can be mitigated by employing heuristics. The *Demand-Index Multinode Allocation* (DI-MNA) heuristic [Táparo et al. 2024] was designed as a greedy approach that employs a priority queue for jobs and a search space pruning mechanism. The workflow consists of first sorting the input jobs given demand-index parameters and subsequently, for each job, calculating relative latencies, applying a filter function, and, finally, searching for the best solution from the filtered nodes. In order to cap the combinatorial exploration, two hyperparameters are set: one establishes the maximum combination size, and the other bounds the filter evaluation extent.

This paper presents two parallelization strategies for the combinatorial search step of the DI-MNA heuristic. The first strategy targets CPU architectures and employs a search procedure that resembles a *Depth-First Search* (DFS) algorithm. The second strategy is directed at GPU architectures and incorporates an unranking mechanism into a lexicographical combination generator. We evaluate and compare both parallelization strategies using large-scale IoT datasets, varying input size and hyperparameter configurations. The results revealed the speedups of both parallel approaches compared to the sequential heuristic.

This work is structured as follows: Section 2.1 introduces the optimization modeling for the MNA problem in IoT networks, Section 2.2 details the DI-MNA heuristic and its computational complexity. Section 3 entails the parallelization approaches tailored for the DI-MNA heuristic, Section 3.1 contains the OpenMP based approach and Section 3.2 contains the CUDA based approach. Finally, Section 4 presents the experimental setup and methodology, Section 5 details the performance results from the parallel approaches and Section 6 synthesizes the main conclusions and future work.

2. The MultiNode Allocation Problem

This section entails the optimization model and the DI-MNA algorithm for the MultiNode Allocation Problem. First, the IoT network and its processing jobs are formally defined,

followed by the DI-MNA allocation heuristic and its filtering algorithm. Finally, it is discussed the computational complexity of the DI-MNA heuristic and its main bottlenecks.

2.1. Optimization Model and Heuristic

An undirected weighted graph $G = (V, E)$ models the IoT network, where each vertex $v \in V$ denotes an IoT device, and each edge $e \in E$ represents a communication link between neighboring nodes. Each node v is characterized by its processing capacity and available bandwidth. The weight associated with each edge corresponds to the communication delay (latency) between connected devices.

Similarly, each job (or task) is modeled as a data structure specifying its minimum processing requirement (R), bandwidth requirement (B), and the maximum tolerable latency for its operation. Furthermore, each job originates at a specific network node, which acts as its entry point to the system. The MNA problem is thus formulated as finding suitable target nodes for a set of jobs such that the latency constraints, as well as the R and B requirements, are strictly satisfied. The *Objective Function* (OF) for optimizing network resource allocation is defined in Equation 1.

$$\min F_R^2(i) + F_B^2(i) - F_L(i) \quad , \forall i = 1, \ldots, n_{\text{elem}} \tag{1}$$

where $n_{\text{elem}} = 2^{n_{\text{nodes}}}$ denotes the total number of node combinations that compose the search space. Furthermore, the terms $F_R(i)$ and $F_B(i)$ quantify the surplus processing and bandwidth capacities available after satisfying the requirements of a job, while $F_L(i)$ measures the remaining latency margin between the maximum delay allowed and the communication latency yielded by the allocation. The computations for the resource, bandwidth, and latency components are given by Equations (2)-(4).

$$F_R(i) = \sum_{k=1}^{n_{\text{nodes}}} f_r(k) - J_r(j) \quad , \forall j = 1, \ldots, nj \tag{2}$$

$$F_B(i) = \sum_{k=1}^{n_{\text{nodes}}} f_b(k) - J_b(j) \quad , \forall j = 1, \ldots, nj \tag{3}$$

$$F_L(i) = J_l(j) - f_l(i) \quad , \forall j = 1, \ldots, nj \tag{4}$$

$$F_R(i) \geq 0, \qquad F_B(i) \geq 0, \qquad F_L(i) \geq 0 \qquad , \forall i = 1, \ldots, n_{\text{elem}} \tag{5}$$

In these formulations, a job j has processing, bandwidth, and latency demands defined by $J_r(j)$, $J_b(j)$, and $J_l(j)$, respectively. For a given network node k, $f_r(k)$ and $f_b(k)$ designate its available processing and bandwidth resources. Conversely, i refers to a candidate allocation, with $f_l(i)$ expressing its accumulated latency. Finally, a candidate allocation is qualified as feasible only if it matches the constraints of Equation 5, which ensures that resource, bandwidth and latency restraints are satisfied by the candidate allocation. Consequently, the proposed model drives the search towards solutions that avoid resource overprovisioning, simultaneously favoring allocations that maintain substantial latency margins.

2.2. Demand-Index Multinode Allocation Heuristic

The DI-MNA heuristic introduces specific hyperparameters to prune the search space, thereby reducing computational overhead while maintaining solution quality. Hyperparameter α, also known as cut_comb_nodes, limits the filter range of evaluated nodes and β, also referenced as cut_sol, sets the maximum combination size. Furthermore, the heuristic incorporates a Demand Index-based ordering mechanism to prioritize jobs with more critical resource requirements. This deterministic evaluation order intrinsically defines the heuristic; any stochastic modifications or deviations from this sequence violate its formal definition. The heuristic's functionality is described in Algorithm 1 [Táparo et al. 2024], along with the computational cost of each step.

Algorithm 1 The DI-MNA heuristic

Inputs: Resources: R, B, L; Job demands: J_r, J_b, J_l, J_o; Coefficients: c'_0, c'_1, c'_2 Adjacency list: $adjList$;
Hyperparameters: $cut_comb_nodes \leftarrow \alpha$, $cut_sol \leftarrow \beta$;
Outputs: Allocated nodes: v_all_nodes; OF values: v_all_OF

$t_c \leftarrow 1$ % connection time (ms) ▷ $\boldsymbol{O}(1)$
$available \leftarrow$ [True, . . . , True] ▷ $\boldsymbol{O}$(V)
$DI = c'_0 R + c'_1 B - c'_2 L$ ▷ $\boldsymbol{O}(1)$
$J =$ list$(*ordered_list(DI))$ ▷ $\boldsymbol{O}$(J log J)
for all job j in J **do** ▷ $\boldsymbol{O}$(J)
    Compute node latencies from source $J_o[j]$ using Dijkstra ▷ $\boldsymbol{O}$(E log V)
    $filtered_nodes \leftarrow$ Preselect based on latency threshold ▷ $\boldsymbol{O}(V^\beta)$
    Initialize $Min_FX \leftarrow \infty$, best solution $\leftarrow \emptyset$ ▷ $\boldsymbol{O}(1)$
    for all node subsets $comb_nodes \subseteq filtered_nodes$ with size $\leq cut_sol$ **do** ▷ $\boldsymbol{O}(V^\beta)$
        Resource computation (Sum_R, Sum_B, Sum_L) ▷ $\boldsymbol{O}(1)$
        if job requirements satisfied: $Sum_R \geq J_r[j]$, $Sum_B \geq J_b[j]$, $Sum_L \leq J_l[j] - t_c$ **then**
            Compute $OF \leftarrow (Sum_R - J_r[j])^2 + (Sum_B - J_b[j])^2 - (J_l[j] - t_c - Sum_L)$ ▷ $\boldsymbol{O}(1)$
            if $OF < Min_FX$ **then**
                update best solution ▷ $\boldsymbol{O}(1)$
            end if
        end if
    end for
    if best solution found **then**
        Allocate nodes and set them unavailable ▷ $\boldsymbol{O}(1)$
        Store Min_FX and allocated nodes ▷ $\boldsymbol{O}(1)$
    else
        Store 0 and empty allocation ▷ $\boldsymbol{O}(1)$
    end if
end for

The algorithm processes each job through three sequential phases. The first phase computes the cumulative latency cost between the job's entry node and all other nodes

within the network. This step is essential to filter out candidate nodes that violate the job's maximum latency threshold, and it adopts the Dijkstra's algorithm to solve the *Single-Source Shortest Path* (SSSP) problem.

The second phase entails the preselection of candidate nodes based on a multi-criteria evaluation, which encompasses latency bounds, current node resource utilization, and the specific requirements of the job. The detailed pseudocode for this procedure is outlined in Algorithm 2[Táparo et al. 2024].

The third and final phase consists of exploring the pruned search space to identify the optimal node assignment. An exhaustive search is conducted exclusively within this restricted subset of candidates, strictly adhering to the hyperparameter bounds.

Algorithm 2 Preselect nodes

Input: Resources: R, B, L; Available nodes: A; Job demands j_r, j_b, j_l;
Hyperparameters: cut_comb_nodes, cut_sol
Output: Set containing nodes that compose valid allocations: $valid_nodes$

$t_c \leftarrow 1$
$Candidates \leftarrow \{i \in A \mid A[i] = True\}$
$num_nodes \leftarrow min(Candidates.size, cut_comb_nodes)$
$Candidates \leftarrow candidates[1 \dots num_nodes]$
$found_combs \leftarrow 0$
$valid_nodes \leftarrow \emptyset$
for all node subsets $comb_nodes \subseteq Candidates$
with size $\leq cut_sol$ **do** Resource computation(Sum_R, Sum_B, Max_L)
    if job requirements satisfied: $Sum_R \geq j_r, Sum_B \geq j_b, Max_l \leq j_l - t_c$ **then**
        for $node \subseteq comb_nodes$ **do**
            $valid_nodes \leftarrow valid_nodes \cup \{node\}$
        end for
        $found_combs \leftarrow found_combs + 1$
    end if
    if $found_combs = num_nodes$ **then**
        break
    end if
end for
return $valid_nodes$

To evaluate the scalability of the proposed approach on large-scale networks, we analyze the algorithm's inherent computational complexity. The worst-case runtime is heavily dominated by the third phase: the combinatorial exploration of candidate node subsets, as enumerating and evaluating allocations requires $O(V^\beta)$ operations per job. Although the hyperparameters are employed to bound this search space and maintain polynomial tractability, the number of feasible combinations in extended topologies (e.g., comprising 1,000 nodes or more) imposes a severe computational overhead. Consequently, the sequential evaluation of this combinatorial phase becomes highly prohibitive, explicitly necessitating the adoption of parallel computing strategies to achieve feasible execution times under massive workloads, which is the focus of this work.

3. Parallel Approaches for DI-MNA

This section presents the parallelization approaches proposed for the DI-MNA algorithm. The implemented algorithms focus on DI-MNA's main bottleneck identified in Section 2.2, the combinatorial exploration. Section 3.1 details the parallelization approach tailored for CPUs, implemented with *OpenMP* directives. Finally, Section 3.2 entails the parallel approach based on GPUs, which employs an unranking procedure and a lexicographical generator.

3.1. An OpenMP Design for DI-MNA

Our first approach to parallelizing the DI-MNA search algorithm is a parallel *Depth-First Search* (DFS) with *OpenMP* (OMP) directives [Dagum and Menon 1998, Chapman et al. 2007]. The DFS algorithm is based on nested loops that yield the candidate solutions to be evaluated. This approach simplifies both the necessary data structures and the parallelization directives employed. The *pseudocode* is presented in Algorithm 3 containing an OMP-like *pragma* directive at line 2.

Algorithm 3 OpenMP Fixed Search

Input: Valid nodes V and job J
Output: Best combination C and objective OF

```
1:  Initialize thread-local (OF_t, C_t) for each thread t
2:  # Omp Parallel For Schedule(Dynamic, 10)
3:  for i ∈ V do
4:      Evaluate feasible combination (i)
5:      for j ∈ V, j > i do
6:          Evaluate feasible combination (i, j)
7:          for k ∈ V, k > j do
8:              Evaluate feasible combination (i, j, k)
9:              Update thread-local best using OF and lexicographic order
10:         end for
11:     end for
12: end for
13: (OF, C) ← best solution
14: return (OF, C)
```

Algorithm 3 starts by initializing local variables at each thread, then proceeds to launch them in batches of the outer loop. Figure 1 shows the distribution of combinations among two threads. Here, one may notice that each thread will generate parallel node combinations from its initial node. Also, since loops enforce a lexicographical order, Figure 1 display the inherent unbalancing of this approach: bigger initial nodes yield fewer combinations. To mitigate the load imbalance, our approach uses dynamic scheduling for batch distribution and a relatively small chunk size of 10. This scheduling strategy will enforce demand-driven batch management, while the small chunk size limits the intrinsic workload, resulting in an active load distribution. In our results 16 threads were used.

Inside the nested loops, each thread will compute the objective function value of the candidate allocation and insert the smallest values as its local result. Both the computation of the objective function and the feasibility check of candidate allocations

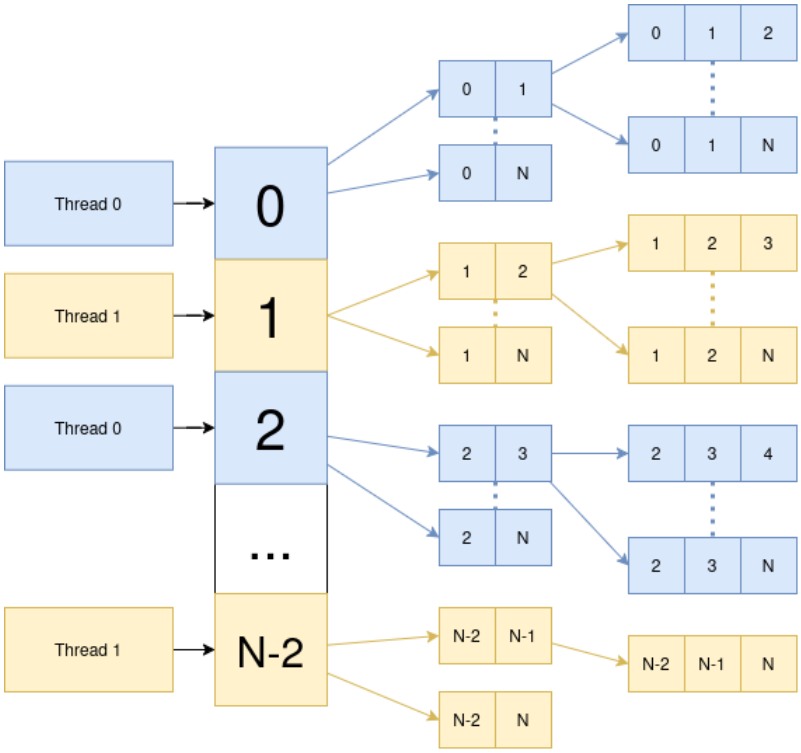

Figure 1. Visualization of the OMP parallel approach.

are performed similarly to Section 2.2. Once a combination has the same OF evaluation as the local best, it must compute which combination is first in lexicographical order to update the local combination. This constraint must be kept to yield the same results as in a sequential solution, even if the parallelism shuffles the evaluation order.

3.2. A CUDA Design for DI-MNA

The second DI-MNA parallelization strategy targets Graphical Processing Units (GPUs) architectures. We employ NVIDIA's *Compute Unified Architecture* (CUDA) [Cook 2012, Garland et al. 2008] API to design a parallel algorithm for GPUs that is based on two steps: combinatorial *unranking* and lexicographical generation. Broadly, these steps are capable of providing a partitioning system for the combinatorial generation, allowing each thread to explore its own section of search space.

The combinatorial *unranking* step at Algorithm 4 [Genitrini and Pépin 2021] is used as an initialization procedure for every thread. In this context, each thread calculates the first combination of its partitioned search space. At first, when the threads are launched, the input i represents its first combination index, n is the number of nodes in the network and k is the β hyperparameter. At lines 1 and 2, the procedure uses the binomial coefficient to calculate the number of combinations contained in each combination size until k, so that it can find the corresponding size of i. Lines 3-9 represent the unranking itself which also uses the binomial coefficient to calculate whether the ith combination contains the pth node. This is done by selecting p and testing whether its indexing range contains i at line 5. This is repeated until size s is reached. In conclusion, the unranking mechanism is used as a means to index the search space in order to properly distribute loads across many GPU threads.

Algorithm 4 Combinatorial Unranking

Input: Index i, n, maximum combination size k
Output: Combination C

1: Calculate the combination size s containing index i with maximum size k
2: Convert i to local size indexing s
3: $x \leftarrow 0$
4: **for** $p \leftarrow 0$ to $s - 1$ **do**
5: Skip combinations that precede the target index
6: Select the corresponding node as $C[p]$
7: $x \leftarrow C[p] + 1$
8: **end for**
9: **return** C

In Algorithm 5 [Ruskey 2003], it is detailed how, given a combination, threads will compute its following candidate solution lexicographically. The inputs are composed of an input combination C, the number of nodes in the network n and the β hyperparameter. At lines 1-7, the algorithm calculates the so-called pivot e, which represents the last element in the combination that can be increased. The pivot is inferred from traversing the combination from end to start, testing whether the current node could be increased by one and still preserve the lexicographical order. Once the pivot is found, lines 8-12 show how the elements at the pivot position forward are updated whenever a valid pivot exists, which represents the case in which the combination size stays the same. Finally, lines 13-16 show how a combination with a new size is generated when the absence of a pivot indicates that the current size is exhausted. Lines 16-18 represent a final state where the thread containing the last partition ends its execution once no more combinations exist. In this scope, threads now can iterate through generated combinations in order to traverse their partitioned search space.

Within this framework, Algorithm 6 shows the complete procedure that is executed by the assigned GPU threads. At lines 1-2 we calculate the total indexing range and use the inherited global thread index to calculate its assigned search space. Then, at line 3, the first combination inside its search space is computed through the unranking mechanism. Lines 4-7 show the main block where threads compute the *OF* from the present combination, save its partial solution and compute the next combination. Finally, lines 9-10 represent the reduction steps where we gather the global best solution reduced from all threads. Ultimately, this workflow uses the combinatorial unranking and the lexicographical generation to compute the search space as an index vector reduction. This structure benefits the GPU architecture at the expense of two complex algorithms for the indexing abstraction. This tradeoff can be better observed once experiments show performance results for different workloads. At this implementation, the CUDA kernel is launched with 256 threads per block, the total amount of blocks guarantees at least 32 blocks per *Streaming Multiprocessors* (*SMs*). Thus keeping high occupancy.

4. Experimental Setup

The experiments designed in this work investigated the proposed parallel strategies in a large-scale *IoT* dataset, comparing the runtime of the single-threaded (sequential), OpenMP, and CUDA at the combinatorial exploration step. Thus, different platforms

Algorithm 5 Lexicographical Combination Generator

Input: Combination C of size s, n, maximum combination size k
Output: Next combination C

$e \leftarrow -1$
for $i \leftarrow s-1$ **down to** 0 **do**
    if $C[i] < n - s + i$ **then**
        $e \leftarrow i$
        break
    end if
end for
if $e \neq -1$ **then**
    $C[e] \leftarrow C[e] + 1$
    for $j \leftarrow e+1$ **to** $s-1$ **do**
        $C[j] \leftarrow C[j-1] + 1$
    end for
else if $s < k$ **then**
    $s \leftarrow s + 1$
    $C \leftarrow (0, 1, \ldots, s-1)$
else
    return END
end if
return C

Algorithm 6 GPU Combinatorial Search

Input: Combinatorial space (n, k), node attributes, job requirements, $thread_index$
Output: Best solution (OF, C)

Compute total combinatorial range R
Determine assigned index range $[start, end)$ from $thread_index$
Unrank the first assigned index to obtain C
while assigned indices remain **do**
    Calculate $OF(C)$
    Update local best solution
    $C \leftarrow$ next combination in lexicographical order
end while
Reduce local solutions within the block
Update best solution
return (OF, C)

for OpenMP and CUDA versions only affect the speedup of the parallelized areas. The objective of results focus on understanding the capabilities and restraints of each proposed version.

The hardware setup was based on two configurations. The OpenMP configuration consists of a 13th Generation Intel i5 CPU clocked at 4.6GHz, with 16GB of memory. The CUDA configuration consists of a dedicated NVIDIA GeForce RTX 4080 GPU backed by an AMD Ryzen 5 5600GT CPU and 128GB of memory. The sequential (single-thread) DI-MNA heuristic was also evaluated using the same configuration of the OpenMP approach.

Each DI-MNA approach was evaluated with the same set of IoT network sizes and the α and β hyperparameters. The datasets comprised networks of sizes $1,000$, $10,000$, and $100,000$, where the number of jobs corresponded to 70% of its size. The α hyperparameter had the following values: $5,000$, $10,000$, $50,000$, and $100,000$, while $\beta \in \{2, 3\}$. Each run was repeated 10 times for mean and standard deviation analysis. Configurations that exceeded an imposed 48-hour total execution time limit were not evaluated. For the sequential version, only $\alpha = 5,000$ met this duration limit, therefore the speedup analysis was made only at this configuration. The Objective Function (OF) gain analysis was not added, as all strategies always yield the same allocation results. Additionally, the scalability analysis of the OpenMP algorithm was tested 10 times on a network with 10.000 nodes, hyperparameters α set to 5.000 and β set to 3. At the scalability experiments, thread number varied within $\{2, 4, 8, 16\}$.

5. Results

The results for all experiments are presented in Table 1. N corresponds to the network size, α and β are the input hyperparameters. The runtime of the combinatorial exploration step mean and standard deviation of the 10 runs are presented for each parallel algorithm, including the sequential DI-MNA heuristic. Cells marked with an asterisk (*) denote runs that have no results due to the imposed time limit. The *speedups* achieved by each parallel strategy in comparison to the sequential runtime are shown in Table 2. Finally, the scalability analysis of the OMP version is shown at Table 3.

The DI-MNA OpenMP approach provides a substantial and consistent speedup over the sequential baseline. Regarding $\beta = 2$, the parallel approach achieves a speedup ranging from $5\times$ to $6\times$. This advantage becomes critical for $\beta = 3$, a significantly more demanding combinatorial scenario. In this case, the OpenMP version yields speedups from $6.5\times$ to $6.8\times$. These results highlight the effectiveness of the parallelization strategy in mitigating the severe computational cost of the evaluations.

The CUDA DI-MNA strategy also provides substantial speedup across different configurations. For $\beta = 2$, the CUDA implementation has less significant speedup, tends to $2\times$ performance improvement. In contrast, for $\beta = 3$, the CUDA approach had a significant performance boost, seemingly directly correlated to the dataset size. This configuration yielded the best speedups, ranging from $24\times$ at the $1,000$ nodes dataset to $114.11\times$ at the $10,000$ nodes dataset. These results suggest the CUDA algorithm is better applicable to larger workloads.

In comparison, the OpenMP parallel approach surpasses the CUDA alternative in performance when the β hyperparameter is set to two and, consequentially, the workload

N	α	β	Sequential		OpenMP		CUDA	
			Mean	SD	Mean	SD	Mean	SD
1k	5k	2	1.83E-01	7.61E-04	3.59E-02	2.12E-04	3.56E-01	1.80E-04
		3	2.47E+01	1.40E-02	3.58E+00	3.16E-03	1.01E+00	4.86E-04
	10k	2	*	*	3.60E-02	1.27E-04	3.57E-01	1.79E-03
		3	*	*	3.58E+00	3.20E-03	1.01E+00	2.54E-03
	50k	2	*	*	4.06E-02	1.30E-03	3.68E-01	1.55E-04
		3	*	*	3.58E+00	2.64E-03	1.03E+00	5.63E-03
10k	5k	2	7.31E+01	3.24E-03	1.19E+01	3.21E-02	3.45E+01	2.71E-02
		3	8.87E+04	4.84E+01	1.35E+04	1.68E+00	7.77E+02	2.83E-01
	10k	2	*	*	2.49E+01	4.25E-03	5.08E+01	1.89E-02
		3	*	*	4.30E+04	8.11E+01	4.34E+03	9.22E-01
	50k	2	*	*	3.37E+01	3.77E-03	5.69E+01	2.32E-02
		3	*	*	6.56E+04	2.86E+02	1.06E+04	2.22E+03
100k	5k	2	6.35E+02	1.44E+00	1.07E+02	1.07E-02	3.03E+02	2.69E-01
		3	*	*	*	*	6.26E+03	7.59E+02

Table 1. Experiment results for all datasets and hyperparameters configurations.

Network Size	β	OpenMP	CUDA
		Speedup	
1k	2	5.10	0.52
	3	6.89	24.49
10k	2	6.13	2.12
	3	6.57	114.11
100k	2	5.93	2.10

Table 2. Speedup comparison between *OMP* and *CUDA* strategies on varying datasets with $\alpha = 5,000$ and $\beta = 2;3$.

Network Size	α	β	Speedup			
			2	**4**	**8**	**16**
10k	**5.000**	**3**	1.84	3.42	5.60	6.57

Table 3. OpenMP version speedup by number of threads.

is reduced. In contrast, the CUDA DI-MNA parallel approach yields significantly better execution times once β is set to 3. Therefore, the substantial increase of the search space attributed to the β hyperparameter is directly correlated to the CUDA strategy performance while the OpenMP approach remains generally constant. This behavior in the CUDA implementations can be a direct consequence of the tradeoff mentioned at the end of Section 3.2. For smaller workloads, the additional complexity gained for the indexing abstraction impairs performance although it provides a solid foundation for better performances on larger workloads.

Finally, we evaluate the scalability of the OpenMP CPU implementation in Table 3 . Initially, the speedup increases substantially from 2 threads to 8 threads, ranging from $2\times$ to $5.6\times$. Once thread number reaches 8, the speedup is less pronounced as it reaches $6.57\times$ at 16 threads. In conclusion, 16 threads is the best configuration for the OpenMP approach.

6. Conclusions

This work presented two parallelization strategies for the combinatorial search step of the DI-MNA heuristic. The first strategy was based on the OpenMP library and employed a search procedure that resembles a Depth-First Search (DFS) algorithm. The second strategy was targeted at GPU architectures, using the CUDA API, and incorporated an unranking mechanism into a lexicographical combination generator. To the best of our knowledge, these approaches are the first parallel implementations of heuristics for the MNA problem in IoT networks.

We evaluate and compare both parallelization strategies adopting a large-scale IoT dataset, varying input size and hyperparameter configurations. The results show that the OpenMP approach provides a relatively constant speedup and progressively yields better performances with growing datasets and hyperparameters, with a maximum speedup of $6.89\times$. In contrast, the CUDA approach is more sensitive to the hyperparameters and dataset configurations and was able to achieve up to $114.11\times$ speedup. Overall, the CUDA strategy has shown to be better suited for large workloads whereas the OpenMP implementation can achieve better speedups for smaller and more constrained workloads.

Some opportunities for future work are further exploring the proposed strategies on different workloads and environments and better detailing their scalability and performance gaps. Also, new parallelization approaches can be implemented and compared to the current results. Finally, future research should also better characterize how the DI-MNA hyperparameters affects the workload as well as its inherent filtering algorithm.

References

[Chapman et al. 2007] Chapman, B., Jost, G., and Van Der Pas, R. (2007). *Using OpenMP: portable shared memory parallel programming*. MIT press.

[Cook 2012] Cook, S. (2012). *Cuda Programming*. Elsevier.

[Dagum and Menon 1998] Dagum, L. and Menon, R. (1998). OpenMP: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55.

[Garland et al. 2008] Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Zhang, Y., and Volkov, V. (2008). Parallel computing experiences with CUDA. *IEEE micro*, 28(4):13–27.

[Genitrini and Pépin 2021] Genitrini, A. and Pépin, M. (2021). Lexicographic unranking of combinations revisited. *Algorithms*, 14(3).

[Muhamad et al. 2023] Muhamad, W. N. W., Dimyati, K., Javed, M. A., Sarnin, S. S., and Ametefe, D. S. (2023). Survey of resources allocation techniques with a quality of service (QoS) aware in a fog computing environment. *Computers, Materials and Continua*, 76(1):1291–1308.

[Ruskey 2003] Ruskey, F. (2003). *Combinatorial Generation*. University of Victoria.

[Savaglio et al. 2020] Savaglio, C., Ganzha, M., Paprzycki, M., Bădică, C., Ivanović, M., and Fortino, G. (2020). Agent-based internet of things: State-of-the-art and research challenges. *Future Generation Computer Systems*, 102:1038–1053.

[Táparo et al. 2024] Táparo, M., Dantas, B., and Santos, R. (2024). Multi-node allocation heuristics for iot applications. In *2024 XIV Brazilian Symposium on Computing Systems Engineering (SBESC)*, pages 1–6. IEEE.

[Zahoor and Mir 2021] Zahoor, S. and Mir, R. N. (2021). Resource management in pervasive internet of things: A survey. *Journal of King Saud University - Computer and Information Sciences*, 33(8):921–935.