

Hierarchical Sort-Based Parallel Interest-Matching Algorithm for Distributed Simulations

Ma Thị Châu[1] and Lê Thái Thịnh[2(✉)]

[1] HMI Lab, University of Engineering and Technology, Vietnam National University Hanoi, Hanoi, Vietnam
chaumt@vnu.edu.com

[2] University of Engineering and Technology, Vietnam National University Hanoi, Hanoi, Vietnam
23025091@vnu.edu.com

Abstract. Distributed virtual environments (DVEs) form the foundation for many real-time applications, such as online multiplayer games, virtual reality simulations, and collaborative systems. In these environments, a continuous flow of dynamic updates occurs, making it crucial to filter and deliver only the relevant information to each participant. This process, known as interest matching, ensures that each entity receives only the data it needs, thereby reducing bandwidth usage. Traditional methods do not fully exploit the parallel capabilities of modern multiprocessor hardware. The Hierarchical Sort-Based Parallel Interest Matching algorithm improves this process by organizing update and subscription regions into a hierarchical data structure and leveraging parallel processing. This paper identifies and addresses specific algorithmic shortcomings in the original HSP-IM, integrates the enhanced approach within the Unity engine, and conducts experimental benchmark comparisons to establish a foundation for developing simulation applications—particularly distributed VR/AR systems. Our findings demonstrate that advanced interest-matching techniques can be effectively implemented on widely used simulation platforms.

Keywords: Distributed virtual environment · Interest matching · Parallel computation

1 Introduction

DVEs are increasingly utilized across various applications, from online multiplayer games to immersive virtual reality experiences. These virtual reality applications require efficient data exchange mechanisms to facilitate real-time interaction among numerous entities. For example, in applications such as virtual museums or classrooms using VR, users expect a smooth experience while navigating complex virtual worlds populated by hundreds or thousands of objects

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2026
O. Gervasi et al. (Eds.): ICCSA 2025, LNCS 15898, pp. 242–254, 2026.
https://doi.org/10.1007/978-3-031-97657-5_15

and other participants. Typically, a publish-subscribe framework is employed in these systems, where updates are broadcast and selectively delivered based on the interests of recipients. However, without effective filtering, such systems can become inundated with irrelevant messages, leading to increased network bandwidth consumption as the simulation scale expands.

Early approaches were predominantly designed for sequential processing, which meant they couldn't fully exploit the parallelism offered by modern multi-core processors. Recent advancements like HSP-SIM tackle this issue by organizing both update and subscription regions into hierarchical data structures, which minimizes redundant comparisons and leverages concurrency for enhanced performance. Furthermore, instead of recalculating all overlapping regions in every update cycle—even when only a few entities have changed—HSP-DIM employs incremental matching to reprocess only the modified regions, substantially reducing unnecessary computations. From now on, HSP-IM will refer to the core hierarchical sort-based parallel interest matching method, which can be implemented in either its static (HSP-SIM) or dynamic (HSP-DIM) variant.

Although evaluations in controlled environments have demonstrated the scalability and efficiency of the HSP-IM algorithm, its implementation in practical settings introduces additional challenges. In this paper, we extend existing HSP-IM techniques by addressing algorithmic shortcomings and adapting the method into the widely used Unity simulation engine on standard hardware. The primary contributions of this article are as follows:

- **Algorithmic enhancement**: Specific issues in the original HSP-IM algorithm have been identified and resolved, thereby enhancing its overall accuracy. Additionally, a redundant step has been eliminated to improve performance.
- **Unity Integration**: The refined algorithm is adapted for Unity's Job System, demonstrating that advanced interest-matching techniques can be effectively implemented on widely used simulation platforms.
- **Empirical Validation**: Benchmark comparisons were performed against the original HSP-DIM results, and an additional experiment was conducted to compare and analyze the overall performance of HSP-DIM and HSP-SIM. The results indicate that its performance under real-world conditions is very promising.

2 Related Work

Interest matching has long posed a critical challenge in distributed virtual environments, directly affecting the efficiency of data distribution among numerous entities. Most existing interest management approaches fall into four categories: aura-based, grid-based, hybrid-based, and visibility-based schemes [1]. Visibility-based techniques incorporate occlusion and detailed geometry for very precise filtering, but their high ray-casting overhead and geometric complexity place them outside the scope of this work. Aura-based methods determine

relevance by modeling the exact area of influence of each entity. Brute-force interest matching in aura-based yields O(m × n) complexity and becomes costly as entities increase. Sort-based [3] further improves aura-based scalability by sorting updates and subscription lists along each axis, reducing pairwise scan. Grid-based [2] schemes partition space into fixed-size cells, restricting potential overlap check operation to nearby buckets - this trades some precision for faster membership/neighborhood lookups. Finally, hybrid methods combine the advantages of these two paradigms to balance precision and computational cost. However, all of which struggled with scalability, often became impractical in modern large-scale simulations.

To overcome these limitations, researchers began exploring parallel solutions that leveraged multi-core processors. For instance, Liu and Theodoropoulos [4] presented a parallel approach based on grid decomposition, while Marzolla et al. [5] introduced a parallel version of the sort-based method that achieved better scalability by distributing the workload among multiple threads. These parallel methods marked a step forward in handling the expanding complexity of interest matching.

A further advancement is the hierarchical sort-based parallel interest matching technique, referred to as HSP-SIM [6], which organizes subscription regions into a full binary-dimensional tree to minimize redundant comparisons via hierarchical partitioning and distribute the matching process across multiple processing cores. Empirical evaluations demonstrated significant improvements over earlier parallel solutions, particularly with very large numbers of entities. More recently, HSP-DIM has emerged as a dynamic enhancement to this model; instead of recalculating all overlaps in every update cycle, HSP-DIM [7] updates only the parts of the data structure that change, making it highly effective in scenarios where updates are frequent yet involve only a fraction of the total entities. Together, HSP-SIM and HSP-DIM represent important progress in efficiently managing data distribution for large-scale simulation environments.

3 Methodology

3.1 Problem Definition

In distributed virtual environments, each participant or entity is associated with spatial regions that govern data communication. An **update region** defines the area over which an entity broadcasts its state changes, while a **subscription region** represents the area for which a participant is interested in receiving updates. Both regions are typically modeled as axis-aligned bounding boxes (AABBs) in dimensional space, which simplify the representation of spatial extents. Each region is characterized by its lower and upper bounds along each spatial dimension. Consequently, two regions are considered to overlap if they share a common sub-interval across all dimensions. Interest matching is the process of identifying which subscription regions intersect with update regions.

3.2 Algorithm Framework

Data Structures. A core component of this method is the Binary Dimensional Tree (BDT) data structure. The BDT is a full binary tree that recursively partitions each dimensional space into smaller subregions. Given a domain $[0, L)$, it is divided into two equal segments at each hierarchical level. At level i, every node, denoted as *IMNode*, represents a subinterval of length $L/2^i$. Each node not only identifies a specific spatial segment but also serves as a container that organizes range data into distinct categories. In particular, each node contains:

- **lowerList:** stores ranges with their lower bound falling within the node
- **upperList:** stores ranges with their upper bound located within the node
- **coverList:** stores ranges that completely span the node's interval
- **insideList:** available in leaf nodes, holds ranges whose both lower and upper bounds lie entirely within the node

These containers are essential for efficiently determining overlapping regions. They enable the algorithm to localize search to those ranges most likely to intersect, thereby reducing unnecessary comparisons by offering these following properties.

Property 1. *if range R belongs to the upperList of $IMNode_{i,j}$, then R either belongs to the lowerList of $IMNode_{i,j-1}$ or R belongs to the lowerList of $IMNode_{i,j-2}$ and the coverList of $IMNode_{i,j-1}$.*

Property 2. *If range R belongs to coverList of $IMNode_{i,j}$, then it also belongs to lowerList of $IMNode_{i,j-1}$.*

In the context of interest matching, let $R_1 = [L_1, U_1)$ and $R_2 = [L_2, U_2)$ be two ranges. These ranges overlap if and only if the lower bound of one must be less than the upper bound of the other, and vice versa:

$$L_1 < U_2 \quad \text{and} \quad L_2 < U_1.$$

This fundamental condition, combined with Property 1 and Property 2, leads to the following theorems:

Theorem 1. *Given that ub is the lower bound of the range U matching with $IMNode_{i,j}$ that it reside in, every range in $upperList_{i,j}$ that is greater than ub and all range in $coverList_{i,j}$ overlaps with U.*

Theorem 2. *Given that ub is the upper bound of the range U matching with $IMNode_{i,j}$ that it reside in, every range in $lowerList_{i,j}$ that is less than ub overlaps with U.*

Theorem 3. *Given that ub is the lower bound of the range U matching with $IMNode_{i,j}$ that it cross, every range in $lowerList_{i,j}$ overlaps with U.*

Theorem 4. *Given that range U matching with $IMNode_{i,j}$ that it cover, every range in $insideList_{i,j}$ overlaps with U.*

Algorithm Workflow. The proposed algorithm follows a structured workflow comprising three primary steps (Algorithm 1). First, the MappingRangeToTree procedure embeds both update and subscription regions into a BDP tree, ensuring that the regions within each node are maintained in sorted order. This structure exploits the hierarchical relationships between nodes to enable efficient management and comparison of regions. Secondly, the dynamic approach includes a RecalculatingOverlapResult step—absent in the static method—that preserves the overlaps for unchanged regions, thereby minimizing redundant computations during the matching process. Finally, the MatchingTreeToTree step computes the overlaps between update and subscription regions in parallel. Unlike the static approach, which process all overlaps, the dynamic method compares only the modified regions. In practice, the matching-update procedure is invoked periodically rather than on every frame, since interest matching is computationally expensive, yet it need to be frequent enough to ensure the DVE's view up-to-date to every user.

Algorithm 1. Matching Update

MatchingUpdate()

1: MappingRangeToTree()
2: RecalculateOverlapResult()
3: MatchingTreeToTree()

Mapping Ranges to BDT. To embed a range into the tree, the HSP-DIM algorithm first determines the appropriate level based on the range's size. It then locates the *IMNodes* with which the range intersects and inserts the range into the corresponding containers. Specifically, if the range's size is smaller than the unit length at the leaf level h, then it is inserted at leaf node. Otherwise, it is inserted at level l where

$$\frac{L}{2^{l+1}} \leq \text{size}(R_i) < \frac{L}{2^l},$$

with $size(R_i) < L$ and $l > 0$. Next, the algorithm computes the indices $lrid$ and $urid$ for the lower and upper bounds LR_i and UR_i, respectively, as follows:

$$lrid = \left\lfloor \frac{LR_i \times 2^l}{L} \right\rfloor, \quad urid = \left\lfloor \frac{UR_i \times 2^l}{L} \right\rfloor.$$

If $urid = lrid$ and $l = h$, then both bounds lie within the same leaf node and the range is added to the *insideList* container. Otherwise, since the range's size is less than twice the unit length at level l, it can intersect at most two nodes; thus, the range is added to the *lowerList* of $IMNode_{l,lrid}$ and the *upperList* of $IMNode_{l,urid}$. Additionally, if $urid - lrid = 2$, it is also added to $IMNode_{l,lrid+1}$'s *coverList*. Finally, the algorithm independently sorts the *insideList*, *lowerList*, and *upperList* in non-decreasing order based on their boundary values, this can be performed concurrently. Once this mapping process is complete, all range information is embedded in the BDP tree, which can then be used to perform efficient interest matching between update regions and subscription regions.

Matching Sorted List Ranges to a BDT Tree. This section provide a detailed explanation of the algorithm for matching a sorted list of ranges to a BDT tree, as described in Algorithm 2. The algorithm's main loop iterates through each element *ub* in the sorted list of bounds, and comparing it with each leaf node of the tree that it fall into, from left most (*lrid*) to right most (*urid*). If *ub* falls within the current node $IMNode_{h,i}$, we apply either Theorem 1 or Theorem 2, depending on whether *ub* represents upper or lower bound (line 10–17). Additionally, the bound lower of R is added into an auxiliary container and remove from when meet its upper bound, *newInList*, which keeps track of ranges that span multiple nodes. In case *ub* not falls within the current node, in other word it is crossing multiple node, the algorithm then applies Theorems 3 and 4 for each item in *newInList* (lines 23–26). Given the hierarchical structure of BDT, where a bound intersect with current leaf nodes, it also overlaps with its ancestor nodes. Due to this, a bubble up loop (Line 9, 23, 27) ensure matching included for all levels.

Additionally, the *SortMatch* method employs a variant of the sort-based matching approach to identify intersections between the current bound (ub) and the inside ranges that complement Algorithm 2. Two red-black trees are used as containers for ub and the inside ranges, as they need to be maintained in sorted order while supporting frequent insertions and deletions.

Matching Between Tree Parallel. To match two BDT trees, one approach is to iterate over all sorted ranges from the first tree and compare them with the second tree. However, to fully exploit multiprocessor capabilities, the hierarchical dimensional partitioning of the BDT tree is leveraged to distribute matching workloads in parallel. For each $IMNode_{i,j}$ node, two lists are retrieved: *crossNodeRange* and *insideRange*. *CrossNodeRange* is formed by concatenating $lowerList_{i,j}$ with the corresponding bound from $upperList_{i,j+1}$ and $upperList_{i,j+2}$ which refer to the same range stored in the $lowerList_{i,j}$. Meanwhile, *insideRange* corresponds to $insideList_{i,j}$. Both containers remain sorted because the lists within each node are pre-sorted, and ranges in subsequent nodes are always larger than those in the current node. Once these lists are prepared, a sorted list matching to tree procedure (Algorithm 2) is applied to compare each sorted list with their counterpart trees, thereby efficiently identifying all overlaps concurrently. The key difference between dynamic matching (Algorithm 3) and static matching (Algorithm 4) lies in how input are extracted from each tree. In the dynamic approach, only the modified parts of one tree are compared to the other tree, and vice versa. By contrast, the static approach compares both trees in their entirety within a single pass.

3.3 Algorithm Enhancement

This article has introduced corrections and eliminated redundant steps in the original algorithm. Notably, the proposed method in the original paper did not incorporate Theorem 4 as a rule in Algorithm Matching (Algorithm 2). Without Theorem 4, the algorithm could miss overlaps between ranges that span multiple

Algorithm 2. Matching sorted list ranges in with BDT

Matching(sortList, BDT)

1: $overlapSet \leftarrow \emptyset$ ▷ set of all ranges in `sortedList` that overlap
2: $upset, subset \leftarrow \emptyset$ ▷ two red-black trees store bounds
3: $sub_{i,j}, slb_{i,j}, sib_{i,j}$: an iterator of `upperList`, `lowerList`, `insideList`
4: $ub \leftarrow$ `sortedList.begin()`
5: $right \leftarrow$ `maximun ID of the leaf node intersecting sortedList`
6: $left, i \leftarrow$ `minimun ID of the leaf node intersecting sortedList`
7: **while** $ub \neq$ `sortedList.end()` & $i \neq$ `right` **do**
8: **if** ub `fall into` $IMNode_{h,i}$ **then**
9: **for** $l \leftarrow h,\ k \leftarrow i$ `to` 0 `by` $l \leftarrow l-1,\ k \leftarrow k \div 2$ **do**
10: **if** ub `is lower bound` **then**
11: $R.overlapSet+ = upperList_{l,k}.Where(upper \leq ub)$ ▷ Theorem 1
12: $R.overlapSet+ = coverList_{l,k}$ ▷ Theorem 1
13: **if** $(l = h)$ **then** `newInList.insert(R)`
14: **else if** ub `is upper bound` **then**
15: $R.overlapSet+ = lowerList_{l,k}.Where(lower > ub)$ ▷ Theorem 2
16: **if** $(l = h)$ **then** `newInList.remove(R)`
17: **end if**
18: **end for**
19: SortMatch$(ub, sib_i, upset, subset)$
20: $ub \leftarrow ub.next$
21: **else if then**
22: **for** $R \in$ `newInList` **do**
23: **for** $l \leftarrow h,\ k \leftarrow i$ `to` 0 `by` $l \leftarrow l-1,\ k \leftarrow k/2$ **do**
24: $R.overlapSet+ = lowerList_{l,k}$ ▷ Theorem 3
25: **if** $(l = h$ & R `cover` $IMNode_{h,k})$ **then**
26: $R.overlapSet+ = insideList_{l,k}$ ▷ Theorem 4
27: **if** $(k \mod 2 = 1)$ **then** $k \leftarrow k \div 2$ **else** `breaks;`
28: **end for**
29: **end for**
30: $i \leftarrow i + 1$
31: **end if**
32: **end while**

nodes and the inside ranges within those nodes—an issue arising when ranges vary in size and reside at different tree levels.

A further enhancement addresses performance by removing the "Merge Overlap Result" step from the original paper and shifting its workload to a lookup operation. In the original design, merging final results—by intersecting overlaps across all dimensions—was output-sensitive hence computationally expensive. Offloading this process to a hash-based lookup container now provides constant-time access for each participant in the DVE.

Algorithm 3. Dynamic matching between BDT parallel

MatchingTreeToTree()

for all $IMNode_{l,k} \in UpdateBDT$ `in parallel` **do**
  if $(l = h)$ **then** MATCHING($[Modified]insideRanges_{l,k}, SubcribtionBDT$)
  $crossNodeRanges \leftarrow lowerRanges_{l,k}$
  for $b \in upperRange_{l,k+1} \cup upperRange_{l,k+2}$ **do**
    if $Range(b) \in lowerRange_{l,k}$ **then**
      $crossNodeRanges.PushBack(b)$
    end if
  end for
  MATCHING($[Modified]crossNodeRanges_{l,k}, SubcribtionBDT$)
end for
for all $IMNode_{l,k} \in SubcribtionBDT$ `in parallel` **do**
  if $(l = h)$ **then** MATCHING($[Modified]insideRanges_{l,k}, UpdateBDT$)
  $crossNodeRanges \leftarrow lowerRanges_{l,k}$
  for $b \in upperRange_{l,k+1} \cup upperRange_{l,k+2}$ **do**
    if $Range(b) \in lowerRange_{l,k}$ **then**
      $crossNodeRanges.PushBack(b)$
    end if
  end for
  MATCHING($[Modified]crossNodeRanges_{l,k}, UpdateBDT$)
end for

4 Experimental Setup and Analysis

4.1 Implementation Details

This section describes the implementation of the refined HSP-IM algorithms, developed in Unity (version 6000.1.16f) using C#. The algorithms were integrated using Unity's Job System to manage parallel processing and were tuned for optimal performance. The development environment consisted of a standard computer equipped with an Intel Core i7 processor supporting 11 logical cores and 16 GB of RAM, running a 64-bit version of Windows 10. Each data point represents the mean of 50 independent repetitions.

4.2 Benchmark Testing Environment

To evaluate the algorithm's performance, a benchmarking framework similar to that described in [7] is employed. The benchmark involves update and subscription ranges of equal size, randomly distributed within a multi-dimensional space, and the matching algorithm determines all intersections among them. The number of ranges, N, determines the dataset size, comprises $n = N/2$ subscription regions and $m = N/2$ update regions. The overlap ratio, α, represents the proportion of the total area occupied by the regions in the space as a measure of spatial density. Additionally, the modify ratio, f, represents the fraction of ranges that change with each update. This experiment compares our empirical implementation with the original results.

Algorithm 4. Static matching between BDT parallel

MatchingTreeToTree()

for all $IMNode_{l,k} \in UpdateBDT$ `in parallel` **do**
  if $(l = h)$ **then** MATCHING($insideRanges_{l,k}, SubcribtionBDT$)
  $crossNodeRanges \leftarrow lowerRanges_{l,k}$
  for $b \in upperRange_{l,k+1} \cup upperRange_{l,k+2}$ **do**
    if $Range(b) \in lowerRange_{l,k}$ **then**
      $crossNodeRanges.PushBack(b)$
    end if
  end for
  MATCHING($crossNodeRanges_{l,k}, SubcribtionBDT$)
end for

Benchmark Result Validation. In order to compare benchmark results related to the scalability of the algorithm, we measuring execution time required to find intersections as the number of ranges N increased while keeping α constant with $f = 1$. As shown in the table, our measured execution times were approximately 15 times higher than the original authors' results, yet the overall performance trend remained consistent. Notably, our implementation utilized 8 threads compared to the authors' 64 threads, which accounts for a threefold speedup in their experiments. However, even after considering thread count, there remains about a five-fold discrepancy in absolute execution times. This additional difference is likely due to the overhead introduced by hardware architecture and algorithm implementation. Nevertheless, the observed scalability trend clearly demonstrates that the algorithm remains linear manner for large scale data (Table 1).

Table 1. Performance Comparison with the Original HSP-DIM matching

Number of ranges (N)	Original HSP-DIM matching execution time (ms)	Our HSP-DIM matching execution time (ms)
10^2	-	0.26
2×10^2	-	0.33
10^3	-	1.1
2×10^3	-	1.9
10^4	0.6	5.4
2×10^4	0.9	12.2
10^5	4	59
2×10^5	7	120
10^6	40	-
2×10^6	80	-

Overall Stepwise Performance Evaluation. Because HSP-IM comprises multiple sequential steps, evaluating the total runtime is crucial to determining its feasibility. In the experiments described below, each step is measured for different strategies includes *Static* and *Dynamic*, denoted *Dynamic f*, where f represents the fraction of modified ranges (e.g., *Dynamic 0.1*).

First, the mapping step is evaluated (Fig. 1). Overall, it exhibits an almost linear increase in runtime. The *Dynamic 0.1* configuration is only slightly faster than the *Static*, whereas *Dynamic 1* performs significantly worse. This is because the dynamic approach uses binary insertion, which has a worst-case complexity of n^2, and requires binary deletion as well, while the *Static* method performs a bulk sort on all ranges at once. We use C#'s built-in sorting method, which is an introspective sort (introsort) that generally defaults to heapsort for large inputs, providing O(nlogn) complexity. Moreover, even though binary insertion and deletion are quadratic, these operations are partitioned and parallelized, meaning that each sort operates on a smaller data subset concurrently, making the process appear nearly linear. In fact, the *Static* can even achieve slightly sub-linear performance.

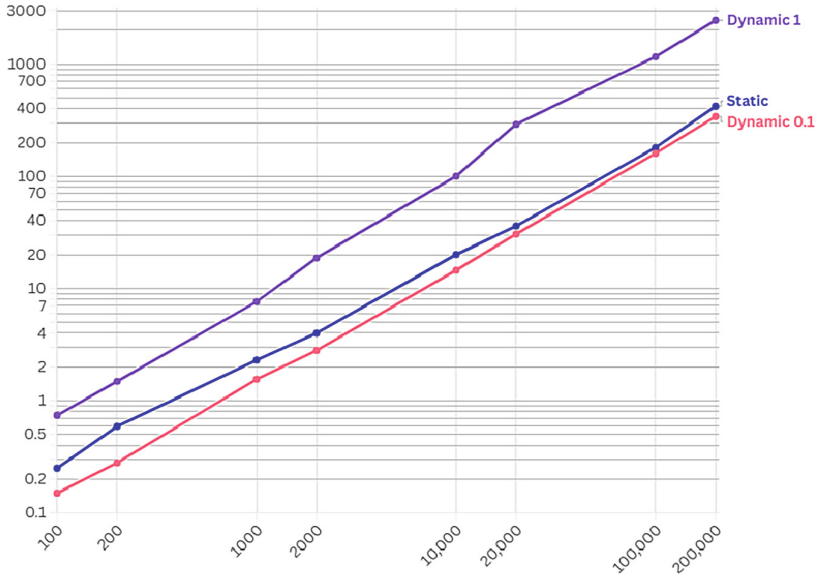

Fig. 1. Mapping execution time

Next, unlike the synthesis benchmark, the matching step only measures the time needed to identify overlaps, this evaluation also accounts for the output-sensitive nature of the recording result step (Fig. 2). Overall, the matching process tends to scale more rapidly as the dataset grows. With an increasing number of ranges, more overlaps occur, causing the performance curve to shift from linear to worst-case quadratic scenario. *Dynamic 1* is slower than *Static*, and nearly

doubles at large data, because *Dynamic 1* is matching redundantly the modified ranges, which are all the ranges, twice. Meanwhile, *Dynamic 0.1* is the most efficient, highlighting the benefit of calculating only the parts that change.

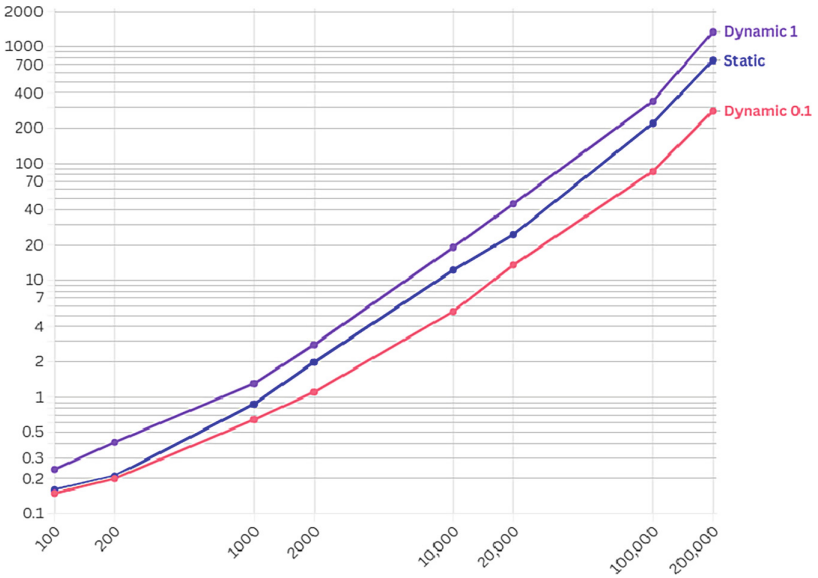

Fig. 2. Matching execution time

Finally, the recalculation of overlap results is examined. Being output-sensitive, it shows a steep slope on a log-log chart (Fig. 3). Regardless of the modification ratio, the workload remains nearly the same. This step involves iterating over the overlap results and removing the entries if they involve modified entities. Its performance suffers for two reasons. First, iterating through a hashset container is inherently costly, also within the Unity Job System design, it must be converted into an array to be iterated which is also computationally expensive. Secondly, Job's max allocation is limited to 2GB. Meanwhile, memory must be preallocated and cannot be resized within a Parallel Job, which forces the hashset to accommodate the maximum possible overlaps. As a result, processing stalls at 10,000 ranges due to the quadratic growth in memory usage. Once that limit is reached, the performance behavior changes; after that point, discrepancies arise only due to the process of recalculating on the actual overlap results. The Static also requires small amount of time to clear old results, which can be considered as "recalculation time".

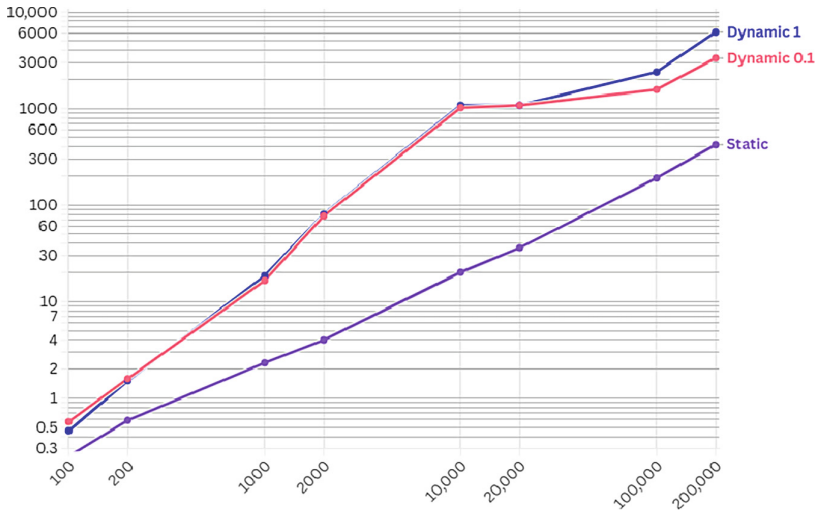

Fig. 3. Recalculate overlap result execution time

4.3 Discussion

In general, the static approach generally outperforms its dynamic counterpart. Although the dynamic method reduces the input size to only the modified data, it introduces an additional output-sensitive step to retain previous results, which undermines its overall efficiency. The mapping process, which operates independently of matching, can be adapted based on the modification ratio: for instance, when the modification ratio exceeds 0.15, the dynamic mapping strategy is recommended, and otherwise the static mapping approach is preferable. Additionally, the current implementation introduces extra steps—data flattening and unflattening, due to data oriented design of the Job System—to convert reference-based, nested data into a blittable, flat structure, and back. These conversions account for up to 25% of the total runtime.

In practice, a threshold of 30 FPS (approximately 33 ms per frame) is typically considered acceptable for real-time simulations. Empirical data suggests that handling up to 2000 ranges is practical to meet this frame budget, as processing 1,000 and 2,000 ranges takes about 6.0 ms and 13.4 ms respectively. Taking into account that some of the available time must be reserved for other tasks, such as other simulation logic and physics.

5 Conclusion

This work refines the original HSP-IM algorithm with minor corrections and optimizations, successfully integrating both HSP-DIM and HSP-SIM into Unity's Job System on standard hardware. The findings indicate that a static matching approach, combined with an adaptive mapping strategy, yields the most practical

performance. Although the reproduced experimental results do not fully match the original due to differences in implementation and hardware architecture, HSP-IM remains consistent with baseline expectations and performs acceptably up to 2,000 ranges in a shared virtual environment. This demonstrates its feasibility for distributed simulation applications like VR, AR developed with widely used engine like Unity, where thousands of users might participate simultaneously. Future efforts will focus on further optimizing algorithmic steps tied to the Job System, designing data structures that integrate more efficiently to reduce overhead, and conduct more diverse scenarios, such as multiple varying range sizes, practical movement patterns.

References

1. Liu, E.S., Theodoropoulos, G.K.: Interest management for distributed virtual environments: a survey. ACM Comput. Surv. **46**(4), Article 51, 42 p (2014). http://dx.doi.org/10.1145/2535417
2. Tan, G., Zhang, Y., Ayani, R.: A hybrid approach to data distribution management. In: Proceedings of 4th IEEE International Workshop on Distributed Simulation and Real-Time Applications (DS-RT), San Francisco, CA, USA, 24–26 August 2000, pp. 55–61 (2000). https://doi.org/10.1109/DISRTA.2000.874064
3. Raczy, C., Tan, G., Yu, J.: A sort-based DDM matching algorithm for HLA. ACM Trans. Model. Comput. Simul. **15**(1), 14–38 (2005). https://doi.org/10.1145/1044322.1044324
4. Liu, E.S., Theodoropoulos, G.: A fast parallel matching algorithm for continuous interest management. In: Proceedings of the 2010 Winter Simulation Conference (WSC), Baltimore, MD, USA, 5–8 December 2010, pp. 1490–1500 (2010). https://doi.org/10.1109/WSC.2010.5679043
5. Marzolla, M., D'Angelo, G., Mandrioli, M.: A parallel data distribution management algorithm. In: Proceedings of 17th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications (DS-RT), Delft, Netherlands, 30 October–1 November 2013, pp. 145–152 (2013). https://doi.org/10.1109/DS-RT.2013.23
6. Tang, W., Yao, Y., Zhu, F., Chen, B., Cai, W.: A parallel hierarchical sort-based interest matching algorithm. In: Proceedings of the 2021 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS 2021), pp. 1–12. ACM, New York (2021). https://doi.org/10.1145/3437959.3459259
7. Tang, W., Yao, Y., Ou, L., Chen, K.: Hierarchical sort-based parallel algorithm for dynamic interest matching. J. Parallel Distrib. Comput. **188**, 104867 (2024). https://doi.org/10.1016/j.jpdc.2024.104867