PONTIFICAL CATHOLIC UNIVERSITY OF MINAS GERAIS
Graduate Program in Informatics

Patrick Rodney de Souza Machado

Evaluating Performance and Scalability of Parallel Unity Job Algorithms

Belo Horizonte
2024

Patrick Rodney de Souza Machado

Evaluating Performance and Scalability of Parallel Unity Job Algorithms

Dissertation presented to the Graduate Program in Informatics at the Pontifical Catholic University of Minas Gerais, as a partial requirement for obtaining the Master's degree in Informatics.

Advisor: Professor Dr. Carlos Augusto Paiva da Silva Martins
Co-Advisor: Professor Dr. Henrique Cota de Freitas

Area of concentration: Computer Science

Belo Horizonte
2024

FICHA CATALOGRÁFICA
Elaborada pela Biblioteca da Pontifícia Universidade Católica de Minas Gerais

M149e

Machado, Patrick Rodney de Souza
Evaluating performance and scalability of parallel unity job algorithms / Patrick Rodney de Souza Machado. Belo Horizonte, 2024.
62 f. : il.

Orientador: Carlos Augusto Paiva da Silva Martins
Coorientador: Henrique Cota de Freitas
Dissertação (Mestrado) – Pontifícia Universidade Católica de Minas Gerais. Programa de Pós-Graduação em Informática

1. Jogos - Desenvolvimento. 2. Processamento paralelo (Computadores). 3. Programação paralela (Computação). 4. Algorítmos computacionais. 5. Computação de alto desempenho. 6. Arquitetura de computador. I. Martins, Carlos Augusto Paiva da Silva. II. Freitas, Henrique Cota de. III. Pontifícia Universidade Católica de Minas Gerais. Programa de Pós-Graduação em Informática. IV. Título.

CDU: 681.3-11

Ficha catalográfica elaborada por Fabiana Marques de Souza e Silva - CRB 6/2086

Patrick Rodney de Souza Machado

Evaluating Performance and Scalability of Parallel Unity Job Algorithms

Dissertation presented to the Graduate Program in Informatics at the Pontifical Catholic University of Minas Gerais, as a partial requirement for obtaining the Master's degree in Informatics.

Area of concentration: Computer Science

Belo Horizonte, December 09, 2024

I would like to express my deep gratitude to Prof. Carlos Augusto Paiva da Silva Martins for his invaluable guidance and support throughout this research. Unfortunately, Prof. Carlos Augusto Martins passed away after the defense of this dissertation, but his academic legacy and contributions will continue to inspire future generations.

ACKNOWLEDGEMENTS

I am deeply grateful for the support provided by CAPES, whose dedication to advancing education in Brazil has empowered countless students, including myself, to achieve their aspirations. Your efforts have been instrumental in making this journey possible.

I extend my heartfelt thanks to Prof. Dr. Carlos Augusto Paiva da Silva Martins for his unwavering belief in my potential and for igniting inspiration throughout this journey. My sincere gratitude also goes to Prof. Dr. Henrique Cota de Freitas for his invaluable guidance and steadfast support, which were crucial in overcoming the challenges along the way.

I am incredibly thankful to the friends who made this journey brighter. To Henrique Augusto, thank you for all the meaningful conversations, laughs, and our shared weird coffee discussions that brought both clarity and humor to challenging moments. To Marta Dias Moreira Noronha and Magna Carla Carvalho Ribeiro, I am grateful for the friendship, shared good times, and unwavering support throughout this experience.

A special thank you goes to my mom, Selma Cristina, for welcoming me and hosting me back when I needed assistance the most, and for always standing by my side. To my little brother, Ruan Miguel, for inspiring me to believe in the possibility of a brighter future for humanity. To Pedro Rigotto and Carol Medeiros, thank you for the great conversations and joyful moments we shared and all the support along the way.

I also want to acknowledge the amazing support and structure provided by the university, which played a vital role in enabling my work. My gratitude extends to all my parents and family members for their love and encouragement, including those who are no longer with us but whose presence is always felt.

Finally, I want to thank myself for refusing to surrender in my darkest moments, for persevering when the odds seemed insurmountable, and for standing firm despite every obstacle.

Perseverança não é apenas encontrar forças para continuar, mas ter a coragem de seguir em frente, mesmo quando elas parecem faltar.

RESUMO

A interseção da tecnologia em evolução e as mudanças sociais no setor de games deu início a uma era de proliferação de dados e aumento das exigências de desempenho. Jogos modernos são esperados para proporcionar experiências imersivas com interatividade fluida, o que coloca considerável pressão sobre os motores de jogo para otimizar o desempenho de forma eficiente. O Unity, um dos motores de desenvolvimento de jogos mais amplamente usado, aborda esses desafios por meio do Unity Jobs System, uma estrutura projetada para permitir que desenvolvedores aproveitem o poder do paralelismo utilizando múltiplos threads da CPU de forma eficaz.

Este estudo investiga a aplicação do Unity Jobs System para paralelizar algoritmos tradicionalmente sequenciais, com o objetivo de explorar seu potencial para melhorias significativas de desempenho. A pesquisa explora aspectos críticos como escalabilidade de threads, configurações de quantidade de agentes e variações no tamanho de batches, além de examinar o papel do Burst Compiler na otimização do desempenho computacional. Ao empregar uma série de benchmarks e cenários de teste, este estudo identifica melhores práticas e recomendações de configuração para desenvolvedores que buscam alcançar o desempenho ideal em contextos diversos de jogos em consonância com a análise de desempenho do Unity Jobs.

Os principais resultados revelam que configurações personalizadas, adaptadas ao caso de uso específico, podem levar a ganhos de desempenho de até 15,06 vezes. Além disso, o Burst Compiler demonstra consistentemente sua capacidade de aprimorar a eficiência de execução em todos os cenários testados, validando sua utilidade como um alicerce para a otimização de desempenho dentro do Unity. Curiosamente, a pesquisa também destaca que as implementações de Unity Jobs de Single-Threaded podem alcançar tempos de execução comparáveis aos de implementações multithreaded em determinadas condições, ressaltando as vantagens de desempenho ao transitar dos paradigmas tradicionais de MonoBehaviour para o Data-Oriented Technology Stack (DOTS).

Esses resultados enfatizam a importância de estratégias de otimização contextuais no desenvolvimento de jogos, onde fatores como características da carga de trabalho e especificações de hardware desempenham papéis fundamentais. Este estudo contribui para o campo mais amplo de engenharia de desempenho em mídias interativas, oferecendo perspectivas práticas para desenvolvedores que buscam aproveitar as ferramentas avançadas do Unity para ampliar os limites do que é possível no universo dos games em performance.

Palavras-chave: Unity Jobs, Jogos, Computação Paralela, Multithreading, Avaliação de Performance.

ABSTRACT

The intersection of evolving technology and societal shifts in gaming has ushered in an era of data proliferation and heightened performance demands. Modern games are expected to deliver immersive experiences with seamless interactivity, which places considerable pressure on game engines to optimize performance efficiently. Unity, one of the most widely adopted game engine, addresses these challenges through the Unity Jobs System, a framework designed to enable developers to harness the power of parallelism by utilizing multiple CPU threads effectively.

This study investigates the application of the Unity Jobs System to parallelize traditionally sequential algorithms, aiming to explore its potential for significant performance enhancements. The research delves into critical aspects such as thread scalability, agent quantity configurations, and batch size variations, while also examining the role of the Burst Compiler in optimizing computational performance. By employing a series of benchmarks and test scenarios, this study identifies best practices and configuration recommendations for developers looking to achieve optimal performance in diverse gaming performance needs contexts.

Key findings reveal that customized configurations tailored to specific use cases can lead to performance gains of up to 15.06 times. Furthermore, the Burst Compiler consistently demonstrates its ability to enhance execution efficiency across all tested scenarios, validating its utility as a cornerstone for performance optimization within Unity. Interestingly, the research also highlights that Unity's single-threaded Jobs implementations can achieve execution times comparable to their multithreaded counterparts under certain conditions, underscoring the performance advantages of transitioning from traditional MonoBehaviour paradigms to the Data-Oriented Technology Stack (DOTS).

These results underscore the importance of context-aware optimization strategies in game development, where factors such as workload characteristics and hardware specifications play pivotal roles. This study contributes to the broader field of performance engineering in interactive media, offering actionable insights for developers seeking to harness Unity's advanced toolsets to push the boundaries of what is possible in gaming performance improvement.

Keywords: Unity Jobs, Game, Parallel Computing, Multithreading, Performance Evaluation.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

DOTS – Data-Oriented Technology Stack
ECS – Entity Component System
CPU – Central Processing Unit
GPU – Graphics Processing Unit
AI – Artificial Intelligence
HPC – High Performance Computing
SISD – Single Instruction, Single Data
SIMD – Single Instruction, Multiple Data
MISD – Multiple Instruction, Single Data
MIMD – Multiple Instruction, Multiple Data
UMA – Uniform Memory Access
NUMA – Non-Uniform Memory Access
NPC – Non-Playable Character

SUMMARY

1 INTRODUCTION

In the current landscape of digital interactive media, performance optimization is not merely a desire but a necessity. Performance bottlenecks in digital games often result from intricate physics simulations, complex polygonal structures, AI's, and inefficient code execution [Koulaxidis e Xinogalos 2022, Singh, Sharma e Sharma 2022], among others. Unity, a prominent game engine, has become a leader in the gaming industry and offers significant benefits to developers [Hussain et al. 2020]. Beyond gaming, Unity is also used in architecture, automotive design, simulation technologies, aerospace engineering, and military simulations [Gang et al. 2016, Pasternak et al. 2018, Shi et al. 2016, Yang e Jie 2011, Kaur et al. 2021, Yang et al. 2016, Gabajová et al. 2021, Bhagat, Liou e Chang 2016].

Despite Unity's versatility and power, developers still face significant challenges in optimizing the performance of their projects. In games especially, user experience can be compromised by freezing, low frame rates or interruptions during execution, often caused by limitations in the sequential processing of complex algorithms. Since games are mostly made for entertainment, such issues in gameplay are undesirable. This challenge is compounded by the need to scale performance efficiently, mainly in systems with multiple CPU cores.

A promising solution to these challenges is the parallelization of algorithms through the Unity Jobs System, enabling parallel processing. This approach has the potential to alleviate performance bottlenecks by distributing the workload across multiple threads and optimizing the use of available architecture resources. By exploring thread scalability, batch variations and compiler alternatives, we can identify configurations that maximize performance. Therefore, investigating the performance optimization capabilities of Unity's Jobs System through parallelism is essential.

1.1 Motivation

The rapid advancement of digital interactive media has led to an ever-increasing demand for more immersive and responsive gaming experiences. This evolution necessitates the continuous improvement of game performance, particularly as games become more complex, involving intricate physics simulations, detailed polygonal structures, and expansive virtual worlds. However, achieving optimal performance is a persistent challenge, as developers must contend with the limitations of sequential processing on modern multi-core architectures. These limitations often result in performance bottlenecks, leading to reduced frame rates, freezing, and interruptions that significantly detract from the user experience.

The motivation for this study stems from the critical need to enhance the efficiency of performance optimization techniques in game development. As Unity has established itself as a leading game engine, widely adopted not only in the gaming industry but also across various fields such as architecture, automotive design, and aerospace engineering, the ability to fully harness the power of its optimization tools becomes paramount. Among these tools, the Unity Jobs System stands out as a promising solution for addressing performance bottlenecks by enabling parallel processing of complex algorithms.

The exploration of parallelism within the Unity environment is motivated by the potential to significantly reduce processing time and improve the overall responsiveness of games. This is especially important in the context of real-time applications, where even minor delays can disrupt the immersive experience. By investigating the Unity Jobs System's capabilities, this study seeks to uncover strategies for effectively distributing computational workloads across multiple threads, thereby maximizing the use of available

CPU resources.

1.2 Problem

The central problem addressed by this study is the need for a deeper understanding of how parallelism and workload management impact performance within the Unity Engine using Unity Jobs tool. Specifically, the research seeks to answer the following questions:

1.2.1 Scalability of Parallelism in Unity Jobs

How effectively does Unity's Job System leverage parallelism to improve performance? This question addresses the core issue of whether multithreading can provide significant speedup and efficiency gains in Unity Engine, a critical consideration for developers aiming to optimize their projects.

Another important subject is to address how comparative performance of Flocking Algorithm Implementations would impact deficiency. What are the performance differences between various implementations of a Flocking algorithm made from scratch within Unity using Jobs? Using Burst Compiler does improve performance on all scenarios? By comparing single-threaded, multi-threaded, and standard implementations, this research explores the practical trade-offs and performance outcomes associated with different parallelism approaches in Unity. How much performance Speedup can be leveraged from the Jobs System? How much Burst Compiler influences on performance optimization? Is using a single-threaded job preferable over the usual sequential implementation without any job?

1.2.2 Impact of Batch Distribution on Workload Management

How does batch distribution affect performance in Unity's Jobs System? This question examines how different batch sizes and workload intensities influence computational efficiency, providing insights into optimal batch distribution practices for improving performance in Unity projects. Is it beneficial to use Unity Jobs' recommended configuration set of 16, 32, or 64 batches? Is it worth carrying out specific tests of workload and scalability variations for custom problems?

These questions guide the exploration and contributions of this study, aiming to provide valuable insights and practical solutions for developers working with Unity's parallelism tools.

Furthermore, the study is motivated by the desire to provide practical insights into the scalability and efficiency of various parallelization strategies. By evaluating different configurations of thread scalability, unit setups, and batch variations, this research aims to offer concrete recommendations for developers seeking to optimize their projects. The goal is to identify configurations that not only enhance performance but also adapt to the specific needs of different gaming scenarios, ensuring that the benefits of parallelization are fully realized.

In summary, this study is driven by the need to overcome the challenges of performance optimization in modern game development. By leveraging the Unity Jobs System, we aim to unlock new levels of efficiency and responsiveness in digital interactive media, ultimately contributing to the advancement of the field and enhancing the experiences of end users.

1.3 Hipothesis

1.3.1 There will be benefits in performance out of parallel scalability strategies

Utilizing the Unity Jobs System with parallel processing will result in significant performance improvements in game algorithms, particularly in configurations with higher batch and thread counts. The optimal configuration will vary depending on the complexity of the algorithm and the specific hardware used.

As the complexity of digital interactive media, especially games, continues to grow, optimizing performance becomes crucial to ensure a smooth user experience. The Unity Jobs System offers a parallel processing framework that allows developers to distribute computational tasks across multiple threads, potentially alleviating performance bottlenecks caused by sequential processing. This hypothesis suggests that by leveraging this system, significant performance gains can be achieved, particularly when using higher batch sizes (e.g., 16, 32, or 64) and thread counts, which are expected to more efficiently utilize multi-core CPU architectures.

However, the hypothesis also posits that the effectiveness of these configurations is not universal; instead, it is highly dependent on the specific nature of the algorithm being optimized and the hardware it is executed on. For example, more complex algorithms like the Flocking algorithm, which requires handling numerous interacting entities, might benefit differently from certain batch and thread configurations compared to simpler algorithms like Find Nearest. Additionally, the same configuration might yield varying results on different hardware setups, such as desktops with different numbers of CPU cores or varying processing power.

Therefore, the hypothesis emphasizes the need for a tailored approach, where developers experiment with different batch sizes and thread counts to identify the most efficient setup for their specific problem and hardware environment.

1.3.2 There can be pros and cons in specific scenarios

While the Unity Jobs System's multithreading offers substantial performance gains (speedup), single-threaded Jobs may still provide comparable improvements for smaller datasets, suggesting that multithreading may not always be necessary for achieving efficient performance in certain scenarios.

This hypothesis explores the potential of single-threaded Jobs within the Unity Jobs System as a viable alternative to full multithreading, particularly in cases where the workload is relatively small or less complex. While multithreading is generally seen as a powerful tool for performance optimization, it introduces complexity in managing threads and synchronizing data, which may not always be justified by the performance gains, especially for smaller tasks.

The hypothesis suggests that for smaller datasets or less complex algorithms, the overhead associated with multithreading might outweigh its benefits. In such scenarios, a single-threaded Job might offer a simpler yet still efficient solution, potentially yielding performance improvements comparable to those achieved with multithreading. This is particularly relevant for developers working on smaller projects or targeting platforms where resource constraints make multithreading less practical.

The hypothesis further implies that for certain cases, the sequential execution of an algorithm without employing Jobs at all might still be competitive, particularly when the added complexity of job management and thread synchronization does not translate into a proportional increase in performance. This leads to the broader conclusion that while multithreading is a powerful optimization tool, it is not always the optimal choice, and developers should consider the specific requirements and constraints of their projects when deciding whether to implement it.

1.4 Objectives

This section outlines the goals of this research, focusing on the application and evaluation of the Unity Jobs System as a tool for optimizing performance in game development. The objectives are designed to explore the capabilities of Unity Jobs, particularly in the context of thread scalability, unit and batch variations, and hardware configurations, with the ultimate aim of providing developers with actionable insights.

1.4.1 General Objective

The primary objective of this research is to investigate the effectiveness of the Unity Jobs System in addressing performance bottlenecks commonly encountered in game development. By systematically evaluating the parallel processing capabilities of Unity Jobs, this study seeks to quantify the performance improvements achievable through its implementation. Additionally, the research aims to offer practical guidance and examples for developers on how to best integrate Unity Jobs into their projects in order to leverage peak performance, ultimately contributing to the enhancement of user experiences in digital interactive media.

Thus, this study examines the performance capabilities of the Unity Jobs System, with a specific focus on thread scalability and charge variations. The goal is to determine the most efficient configuration and quantify the potential performance enhancements achievable through the system's utilization compared to scenarios where it is not employed. The aims on analysis delve into Unity Jobs' thread scalability, agent quantity configurations, and batch variations, plus Burst Compiler usage.

1.4.2 Specific Objectives

The specific objectives are the following:

1. **Evaluate Parallel Strategies**: Assess and compare the performance and scalability impacts of using parallel processing strategies, specifically focusing on the Unity Jobs System to optimize performance in interactive media applications.

2. **Investigate Scalability Variations**: Explore the impact of different variations of scalability, including agent quantity, thread configurations, and batch sizes. Assess how these factors affect performance and scalability in various use cases, providing findings on optimization of these parameters in interactive media applications.

3. **Analyze Single-threaded vs. Multi-threaded Jobs**: Investigate the performance benefits of switching from single-threaded job solutions to multi-threaded job solutions, quantifying the speedup achieved by leveraging multiple CPU threads.

4. **Assess the Role of the Burst Compiler**: Examine the performance improvements gained from applying the Burst Compiler to algorithms, comparing execution times and identifying the impact on processing speed when using the compiler versus non-optimized code.

5. **Design Multi-threaded Job Solution**: Develop and test multi-threaded job designs to optimize processing power, ensuring that both single and multi-threaded job configurations are effective for performance scalability in Unity-based interactive media applications.

1.5 Contributions

1.5.1 Evaluation of Parallel Strategies in Unity Jobs System

This study provides an in-depth evaluation of parallel strategies within the Unity Jobs System, with a particular focus on performance scalability. By leveraging multithreading and varying workload characteristics, this research assesses how Unity's Jobs System handles parallel tasks and whether the use of parallelism significantly improves performance in interactive media applications. The findings offer critical insights into the effectiveness of parallel job execution and the impact of thread configurations on Unity's performance.

1.5.2 Exploring Scalability Variations: Agents, Thread Configurations, and Batch Sizes

This study further contributes by exploring various scalability variations, such as agent quantity, thread configurations, and batch sizes, to evaluate their impact on performance. By adjusting these factors, the research uncovers optimal configurations for performance and scalability, providing useful guidelines for developers seeking to optimize resource allocation and task management in Unity's parallelism framework.

1.5.3 Performance Speedup: Single-threaded vs. Multi-threaded Jobs

One of the key contributions of this research is the comparative analysis of performance speedup when switching from single-threaded job implementations to multi-threaded job solutions. By investigating the performance differences between these two approaches, the study highlights the scalability benefits of utilizing multiple threads and offers valuable data on how multi-threading can enhance processing efficiency in Unity's framework.

1.5.4 Impact of Burst Compiler on Performance Optimization

This work examines the role of Unity's Burst Compiler in optimizing performance. By comparing algorithms with and without Burst Compiler optimization, this study quantifies the performance speedup achieved through the use of the compiler. The findings provide a clear understanding of how the Burst Compiler can accelerate computations, especially in high-performance applications requiring fast and efficient execution of complex algorithms.

1.5.5 Design and Implementation of Multi-threaded and Single-threaded Jobs

A significant contribution is the design and implementation of multi-threaded job solutions in Unity. This research not only explores single-threaded job configurations but also develops optimized multi-threaded job architectures to enhance performance scalability. Through the evaluation of both approaches, the study offers practical insights into designing job systems for efficient resource utilization and performance enhancement in Unity-based interactive media projects.

1.6 Organization of this Master's Dissertation

This dissertation is structured to guide the reader through the exploration of performance optimization in digital interactive media using the Unity Jobs System. The organization reflects a logical progression from the motivation behind the study to the detailed examination of relevant literature, followed by the methodology employed, and culminating in the presentation and analysis of results.

Chapter 1: Introduction. This first chapter introduces the topic, outlines the motivation behind the study, and clearly defines the problem being addressed. It presents two hypotheses that drive the research, along with the general and specific objectives that guide the investigation. This chapter sets the stage for the research by explaining the importance of exploring the Unity Jobs System for performance optimization in game development.

Chapter 2: Literature Review. The second chapter provides a comprehensive review of the foundational concepts and classical theories in parallel computing. It covers essential topics such as Flynn's Taxonomy, shared memory architectures, and key metrics in parallel computing. The chapter also delves into advanced concepts and high-performance computing (HPC) strategies, drawing on the work of renowned authors in the field. This literature review establishes the theoretical framework for the study.

Chapter 3: Related Work. This chapter examines recent studies that have explored the parallelization of game algorithms, with a particular focus on the Unity Jobs System. It highlights key works that have utilized Unity's Data-Oriented Technology Stack, also known as DOTS and other performance evaluation methods across different hardware configurations. By analyzing these related works, this chapter positions the current study within the broader research context and identifies gaps that this dissertation aims to fill.

Chapter 4: Methodology. The fourth chapter outlines the methodology used to conduct the research. It describes the data collection process, the shift from frames per second to seconds per frame as a performance metric, and the various software and hardware configurations employed. Detailed descriptions of the algorithms under study, such as the flocking algorithm, are provided, along with their corresponding pseudocode. This chapter provides a clear roadmap of how the research was conducted, ensuring the study's reproducibility.

Chapter 5: Results. The results chapter presents the findings of the research, focusing on key performance metrics such as execution time, speedup, efficiency, and the impact of batch variations. The results are analyzed in the context of different hardware configurations, with a detailed examination of one particular desktop configuration. This chapter critically evaluates the effectiveness of the Unity Jobs System in improving performance and provides insights into the scalability of parallel processing within game development.

Chapter 6: Conclusion. The dissertation concludes by summarizing the key findings, discussing their implications for game development, and suggesting areas for future research. The conclusion reinforces the significance of the study and its contribution to the field of parallel computing in game development.

2 LITERATURE REVIEW

In this chapter, we explore the foundational concepts of parallelism within the field of computer science, drawing on classic and influential works by leading authors. The discussion covers key topics such as Flynn's taxonomy, shared memory architectures, Uniform Memory Access (UMA) and Non-Uniform Shared Memory Access (NUMA), [Pacheco 2011, Hager 2010], as well as the critical metrics of speedup, efficiency, and scalability.

2.1 Flynn's Taxonomy

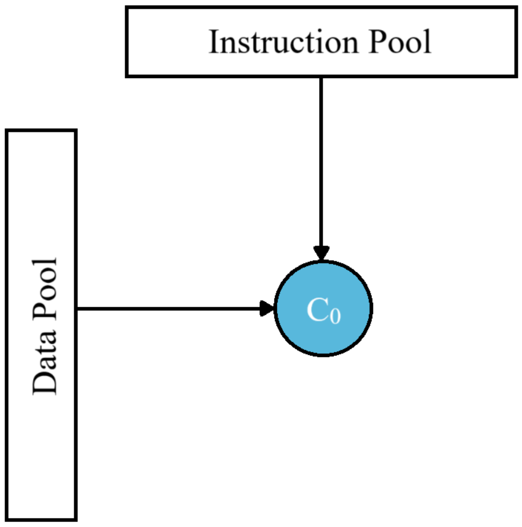
Flynn's taxonomy, introduced by Michael J. Flynn in 1966, on paper [Flynn 1966], is a classification system for computer architectures based on the number of concurrent instruction streams and data streams they can handle, Table 1. Flynn categorized computer architectures into four primary types:

Table 1 – Summary of Relationship Between Pools and Cores in Different Architectures

Architecture	Instruction Pool	Data Pool	Relation to Cores
SISD	Single instruction for the core	Single data processed sequentially	One core processes one task at a time.
SIMD	One instruction for all vector threads	Multiple data processed in parallel	A single core processes multiple data points in parallel using threads.
MISD	Different instructions for each core	Single data processed by all cores	Multiple cores execute different tasks on the same data element (e.g., redundant computations).
MIMD	Independent instructions for each core	Independent data processed by each core	Multiple cores operate independently, making this ideal for general-purpose multi-threaded tasks.

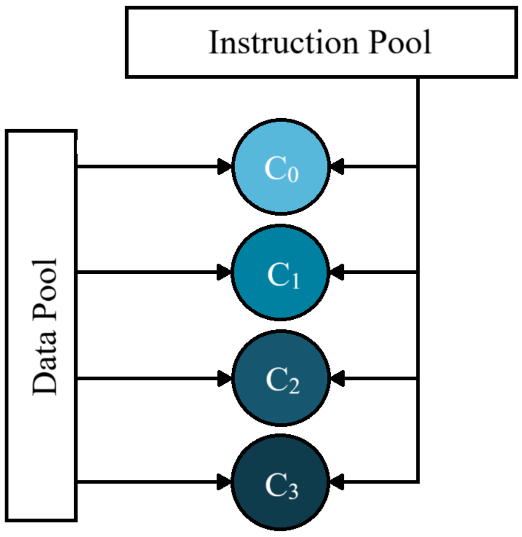
Single Instruction, Single Data (SISD): This represents the traditional sequential computer architecture, where a single processor or core executes one instruction stream at a time, operating on a single data element stored in a unified memory space. In SISD architecture, instructions are processed sequentially, without parallelism or concurrency, making it simpler but typically slower than other architectures that employ multiple processors or instruction streams. SISD systems are characteristic of conventional computing models, such as those found in early microprocessors or simple embedded systems, where processing is performed in a linear manner. Figure 1 illustrates the SISD architecture, highlighting its basic structure with a single control unit, a single data path, and a single memory unit, which together form a straightforward yet foundational computing model.

Figure 1 – Single Instruction Single Data Architecture



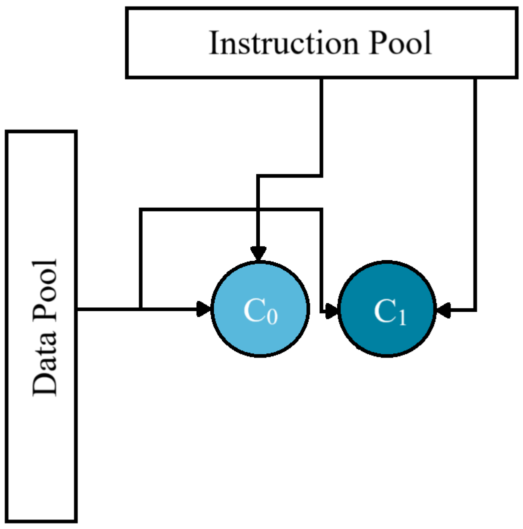
Single Instruction, Multiple Data (SIMD): In SIMD architectures, as shown in Figure 2, multiple processing elements execute the same instruction on different pieces of data simultaneously. This parallel processing model is particularly efficient for tasks involving large datasets where the same operation is applied across multiple data points, such as in vector calculations, image processing, and scientific simulations. SIMD architectures are commonly found in vector processors and Graphics Processing Units (GPUs), which leverage this model to accelerate workloads by dividing data processing across numerous cores. By performing operations in parallel, SIMD significantly enhances performance in applications requiring high throughput and data parallelism, making it ideal for graphics rendering, machine learning, and real-time processing tasks.

Figure 2 – Single Instruction Multiple Data Architecture



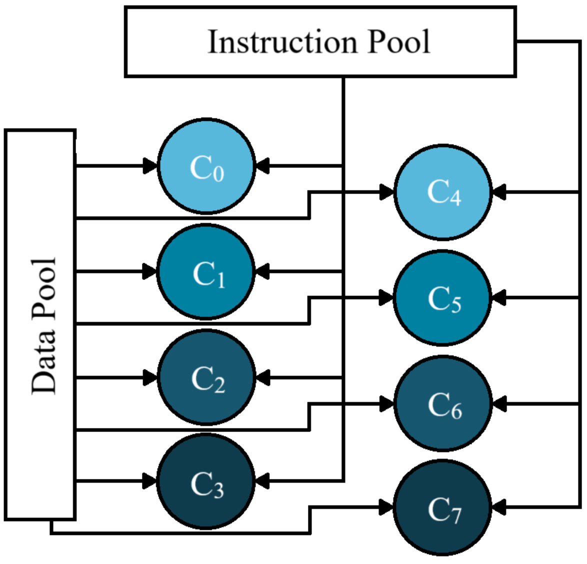
Multiple Instruction, Single Data (MISD): MISD architectures, depicted in Figure 3, involve multiple instruction streams operating concurrently on the same data stream. Although this model is largely theoretical and seldom implemented in practical computing systems, it represents a unique approach to parallelism. In an MISD system, different processing units apply varied operations to the same data, which can be beneficial in specialized applications requiring redundant processing or fault tolerance. For instance, MISD could theoretically be used in fault-tolerant systems, where multiple processors perform different operations on the same data to cross-verify results. However, due to the complexity and limited applicability, MISD is rarely seen outside of niche academic or research contexts.

Figure 3 – Multiple Instructions Single Data Architecture



Multiple Instruction, Multiple Data (MIMD): MIMD architectures, illustrated in Figure 4, consist of multiple cores, each capable of executing its own independent instruction stream on separate data. This form of parallelism is highly versatile, as it allows each processor to perform different tasks simultaneously, making MIMD ideal for a wide range of applications requiring complex, asynchronous processing. MIMD architectures are widely used in modern multi-core and multi-processor systems, as well as in distributed computing environments, where different processors or nodes can execute unique operations in parallel. This flexibility makes MIMD architectures well-suited to handling diverse workloads, such as those found in scientific computing, real-time simulations, cloud computing, and high-performance applications.

Figure 4 – Multiple Instruction Multiple Data Architecture



Flynn's taxonomy remains a fundamental framework for understanding different parallel processing models, and it serves as the foundation for further exploration of shared memory and parallel computing concepts.

In this study, we focus on the MIMD (Multiple Instruction, Multiple Data) architecture as the foundation for our performance optimization approach. MIMD systems are characterized by their ability to execute multiple independent instruction streams simultaneously, each operating on distinct data sets. This allows for significant flexibility and scalability in parallel processing, making it ideal for handling complex tasks such as the Flocking and Find Nearest algorithms discussed in this research. By leveraging the MIMD architecture, we are able to explore various thread management and workload distribution techniques, particularly using the Unity Jobs System, to maximize performance in digital interactive media applications. This architectural model is well-suited for the multi-core and multi-threaded environments prevalent in modern computing, allowing us to achieve higher levels of concurrency and efficiency in our optimizations.

2.2 Shared Memory Architectures

Shared memory is a key concept in parallel computing, where multiple processors access a common memory space. [Adve e Gharachorloo 1996], [Pacheco 2011], [Hager 2010] in their work on computer architecture, describe shared memory architectures as those where processors communicate by reading and writing to a shared memory space through interconnected buses. This model simplifies programming because all processors have a unified view of memory. However, it introduces challenges related to synchronization and memory consistency, as multiple processors may attempt to access the same memory location simultaneously. These processors communicate via shared memory, making memory access a critical design factor.

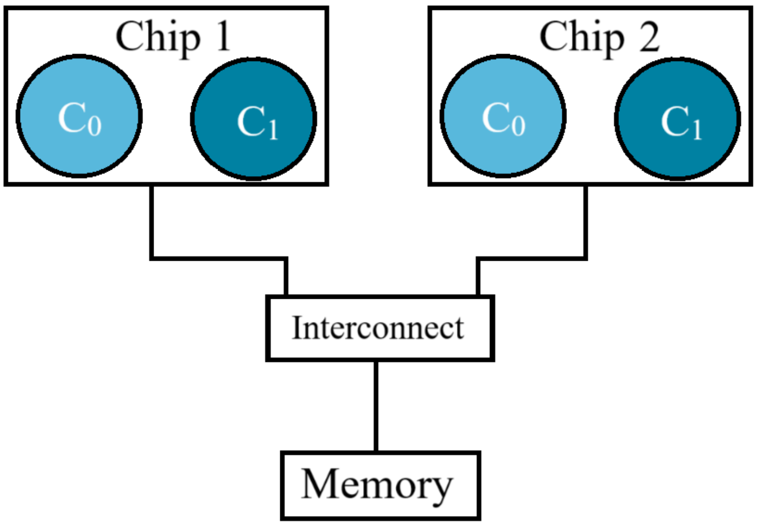
2.2.1 Uniform Memory Access (UMA)

In shared memory systems, Uniform Memory Access (UMA) is a type of memory architecture where the time it takes for any processor to access any memory location is consistent across the entire system. This means that whether a processor is accessing the first or the last memory cell, the time required to do so remains the same. Represented by the Figure 5, where we can observe a example of an UMA architecture of 2 micro-chips each with 2 cores, interconnected unto a shared memory.

For example, imagine a desktop computer with a processor that uses UMA architecture. In this setup, every processor in the system can access any part of the system's memory with the same speed. This uniformity simplifies the system's design because the memory access time is predictable and equal for all processors.

The study of Hennessy and Peterson, [Hennessy e Patterson 2011], experts in computer architecture, point out that while UMA simplifies design by providing equal access time to all processors, it can also lead to issues. As more processors are added, they all compete for access to the same memory. This competition can create contention and bottlenecks because multiple processors might try to access the same memory locations simultaneously, which can slow down the system.

Figure 5 – Uniform Memory Access Architecture



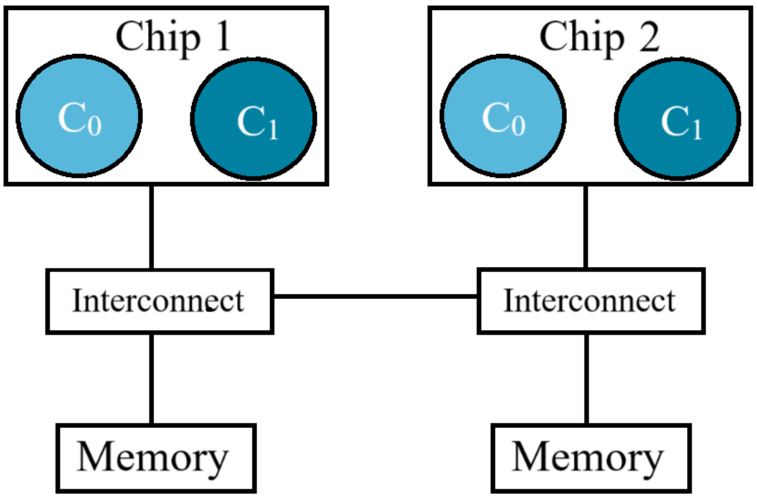
2.2.2 Use of UMA in This Study

In this dissertation, we utilize a Uniform Memory Access (UMA) architecture as part of the system design for performance evaluation. UMA, is particularly relevant to our study because it provides a consistent and predictable memory access time across all processors. This uniformity in memory access simplifies the analysis of parallel processing performance, allowing us to focus on optimizing workload distribution and thread management without having to account for the complexities introduced by varying memory access times. By using UMA, we can establish a clear baseline for understanding how shared memory systems handle concurrent tasks in our multi-threaded environment.

2.2.3 Non-Uniform Memory Access (NUMA)

In contrast to UMA, Non-Uniform Memory Access (NUMA) Figure 6, is a memory architecture that can be explained by: The time it takes for a processor to access memory depends on the location of the memory in relation to the processor. In NUMA systems, memory is divided into different regions, and each processor has faster access to its local memory compared to the memory associated with other processors. This architecture is commonly used in larger multi-processor or multi-core systems to reduce memory access contention and bottlenecks. While NUMA can offer improved scalability and performance in systems with many processors, it also introduces complexity in memory management, as developers must be mindful of which memory regions each processor accesses in order to avoid performance penalties.

Figure 6 – Non-Uniform Memory Access Architecture



2.3 Key Metrics in Parallel Computing

When evaluating parallel computing systems, several critical metrics are used to measure performance, including speedup, efficiency, and scalability. Peter Pacheco [Pacheco 2011] in his book on parallel computing, provides a detailed explanation of these metrics:

- **Speedup:** Speedup is the ratio of the time taken to solve a problem on a single processor to the time taken on multiple processors. It quantifies the performance gain achieved by parallelizing a task. Ideally, the speedup should be proportional to the number of processors, but in practice, it is often limited by factors such as communication overhead and load imbalance. Overall concepts as Speedup from Amdahl [Hill e Marty 2008] are refreshed on recent studies of Hill e Marty 2008, Pacheco 2011, Hager 2010.

- **Efficiency:** Efficiency is a measure of how effectively the processors are utilized in a parallel system. It is defined as the ratio of speedup to the number of processors. High efficiency indicates that the processors are being used effectively, with minimal idle time or overhead.

- **Scalability:** The study of [Bondi 2000, Pacheco 2011, Hager 2010] refers to the ability of a parallel system to maintain efficiency as the number of processors increases.

A scalable system can handle increasing workloads by adding more processors without a significant drop in efficiency. Scalability is a critical consideration in high-performance computing (HPC) environments, where systems are often required to scale to thousands or even millions of processors.

2.3.1 Strong Scalability

Strong scalability refers to the ability of a parallel system to handle a fixed workload more efficiently as the number of processors increases. In other words, the total problem size remains constant, but the system's performance is expected to improve as more processors are added. Originally purposed by Amdahl, the Strong scalability [Pacheco 2011, Hager 2010], is an ideal outcome for many parallel computing systems because it demonstrates that the system can make efficient use of additional processors to reduce the overall time required to solve a problem. However, in practice, achieving perfect strong scalability is often difficult due to factors such as communication overhead, synchronization delays, and load imbalance among processors. This form of scalability is particularly useful in applications where the problem size cannot easily be increased, and the goal is to solve the task as quickly as possible using more computational resources.

2.3.2 Weak Scalability

Weak scalability, on the other hand, focuses on a system's ability to maintain efficiency as both the problem size and the number of processors increase proportionally. Explored in depth on the recent study [Pacheco 2011, Hager 2010, Michiels et al. 2013], weak scalability shows itself as a variation of Strong Scalability. In this case, the workload grows with the number of processors, ensuring that each processor works on a similar amount of data as more processors are added. A system that exhibits good weak scalability can handle larger and more complex problems without a significant drop in performance. This type of scalability is essential in environments where the problem size naturally increases over time, such as large-scale simulations, data processing, or high-performance computing tasks. Weak scalability is particularly relevant when the goal is to maintain performance while expanding the scope or size of the problem being solved.

3 RELATED WORK

The related work section provides a comprehensive overview of recent studies that have investigated the parallelization of game algorithms within the Unity ecosystem. These studies have explored various approaches to performance optimization, particularly focusing on parallel computing techniques such as the Unity Jobs System, the Burst Compiler, and, in some cases, the Entity Component System (ECS). By analyzing these contributions, we aim to identify their relevance to our work, draw comparisons where applicable, and highlight the gaps our research seeks to fill.

Our study is primarily concerned with optimizing performance through job parallelization using the Unity Jobs System and Burst Compiler, excluding the use of ECS to streamline complexity. The Table 2 offers a side-by-side comparison of the key characteristics of the studies, illustrating the different methodologies, technologies employed, and performance metrics examined.

Table 2 – Comparison of Related Studies

Study	**Jobs**	**Burst**	**ECS**	**Thread Scal.**	**Batch Var.**	**Agent Var.**	**Hardware Var.**
[Wang, Lin e Zhao 2020]	Yes	Yes	Yes	No	Yes	No	No
[Eyniyev 2022]	Yes	No	No	No	No	Yes	Yes
[Borufka 2020]	Yes	Yes	Yes	No	Yes	No	Yes
[Näykki 2021]	Yes	Yes	Yes	Yes	No	No	Yes
Our Study	**Yes**	**Yes**	**No**	**Yes**	**Yes**	**Yes**	**No**

3.1 Parallelization in Unity Using the DOTS

In recent years, the parallelization of game algorithms has attracted considerable attention, with several studies exploring methods to improve performance in gaming environments. One prominent study effectively employed Unity Jobs as a parallelization tool to enhance the scalability of agents or entities within a shooter game, utilizing the Data-Oriented Technology Stack (DOTS) [Wang, Lin e Zhao 2020]. Their comprehensive approach integrated all three key components of the DOTS stack: the Unity Jobs System, Entity Component System (ECS), and the Burst Compiler, providing valuable insights into the performance gains achievable through full DOTS adoption.

While the referenced study presents an in-depth exploration of the entire DOTS ecosystem, our research strategically narrows its focus to two specific components: the Unity Jobs System and the Burst Compiler. This decision is grounded in a pragmatic evaluation of the complexities and trade-offs involved in implementing the entire DOTS stack. By concentrating on Unity Jobs and Burst, we aim to streamline the investigation, eliminating the overhead introduced by ECS, and instead channeling our efforts into understanding how these two elements can maximize performance optimization in parallel game algorithms. Our study differs from the referenced work in that we apply Unity Jobs and Burst to the Flocking and Find Nearest algorithms, a fundamental components in AI and behavior simulation. In contrast to their shooter game analysis, our approach emphasizes data parallelism within this algorithm, highlighting how Unity Jobs can efficiently distribute the workload among multiple cores. Additionally, we explore the practical benefits of Burst's low-level optimizations, such as code vectorization and cache optimization, which further improve execution speed without requiring a shift to the full ECS architecture.

Moreover, while the prior work offered a view of DOTS, encompassing a broad spectrum of parallelization tools, our study's more focused scope allows us to provide deeper insights into the Unity Jobs System's specific parallelization mechanics. This focus permits us to optimize code more directly, using fewer abstractions and ultimately producing a more accessible and adaptable solution for game developers looking to enhance performance without committing to the entire DOTS stack.

Through this targeted exploration, we aim not only to demonstrate the effectiveness of Unity Jobs and Burst in real-world gaming scenarios but also to present a streamlined pathway for developers seeking to incorporate parallel computing into their projects without adopting the complete DOTS framework. Our findings highlight a practical and scalable approach to parallelization that balances simplicity and efficiency, providing a strong foundation for further advancements in game development.

3.2 Unity Jobs Performance and Scalability Evaluation

A prior study conducted performance evaluations of the Jobs, testing scalability across various hardware configurations by adjusting algorithm problem sizes [Eyniyev 2022]. This work provided a foundational perspective on how Unity's parallelization framework performs under different computational loads, highlighting its adaptability across diverse hardware.

Our study builds upon these findings by conducting a more granular set of tests, specifically focusing on diverse thread configurations and examining how different levels of parallelism impact performance in Unity. By incorporating a broader range of thread variations, we offer a comprehensive benchmark that evaluates Unity Jobs' handling of computational loads in real-time scenarios. This approach allows us to identify specific factors that influence the performance scalability of Unity Jobs, providing insights into the interplay between hardware capabilities and Unity's parallel processing tools.

While the referenced study primarily examined problem size and hardware variability, our work extends the analysis to cover a wider array of parallelism levels, thereby yielding a richer understanding of Unity's performance optimization potential. Through this more detailed approach, our study contributes to an enhanced benchmark for Unity's Jobs System, offering a valuable resource for developers aiming to optimize performance setups within the Unity ecosystem.

3.3 Flocking Algorithm Performance in DOTS

A significant study in the field conducted performance testing on the Flocking algorithm within Unity's Data-Oriented Technology Stack (DOTS), focusing on batch variation scalability as a key metric [Borufka 2020]. The study employed the Burst Compiler alongside Unity Jobs and the Entity Component System (ECS), examining how varying batch sizes impacted the algorithm's performance. Borufka's research provided important insights into the scalability and efficiency improvements achievable through batch processing within DOTS, demonstrating the effectiveness of this approach in optimizing parallel algorithms like Flocking.

Our study builds upon this foundation by shifting the focus to thread scalability, agent set variations, and batch performance within the Unity Jobs System, plus Burst, while deliberately excluding the Entity Component System (ECS). This strategic choice allows us to concentrate on understanding how job parallelization alone can be leveraged to maximize performance gains, without the added complexity introduced by ECS. In contrast to Borufka's work, which looked at the DOTS environment, our research narrows on the specific contributions of Unity Jobs and the Burst Compiler to performance speedups. By isolating these elements, we aim to provide a more detailed and focused analysis of the parallel capabilities of Unity Jobs, particularly when applied to complex algorithms like Flocking.

Furthermore, our research expands on the previous study by incorporating a broader range of performance metrics. While the referenced work emphasized batch variation scalability, our study explores additional factors such as thread scalability and variations in unit set sizes. This more comprehensive approach allows us to assess how different levels of thread parallelism impact the overall performance, offering a deeper understanding of the relationship between job distribution and computational efficiency. Moreover, we employ more precise time measurement techniques in our analysis, such as the Stopwatch library and Unity Profiler, which enable us to gain a more granular view of thread performance and bottlenecks.

By focusing exclusively on Unity Jobs plus Burst Compiler and leaving out ECS, we aim to provide practical insights into the direct performance benefits of job parallelization within Unity's ecosystem. This approach not only simplifies the analysis but also offers game developers a more accessible pathway for optimizing parallel computations without the need for full DOTS integration. Our findings contribute to the broader understanding of data-oriented parallelism in game development, highlighting how developers can achieve significant speedups through targeted job-based parallelization techniques.

3.4 Performance Evaluation Using DOTS

A recent study [Näykki 2021] benchmarks the performance of Unity's Data-Oriented Technology Stack (DOTS) across various scenarios, including pathfinding and animation, in comparison to conventional object-oriented approaches. The study sheds light on DOTS's efficiency, particularly in pathfinding tasks, with significant performance gains attributed to its job-based parallelism and Burst compiler optimizations.

Both Alarik Näykki's work and our research focus on thread scalability, analyzing how DOTS and Unity Jobs perform under diverse thread configurations. While both studies implement jobs and utilize the Burst compiler to enhance performance, our research diverges by excluding the Entity Component System (ECS), which Näykki's study incorporates. This allows our study to examine the performance of DOTS components and Unity Jobs without the added structure of ECS, potentially offering valuable insights for developers choosing not to integrate ECS.

The study [Näykki 2021] provides a critical reference for understanding the capabilities and limitations of DOTS, contributing data that can guide developers in assessing its production readiness. The findings suggest DOTS's promising potential and establish a basis for future research as Unity further refines its data-oriented approach.

4 METHODOLOGY

Our approach begins with the implementation of the algorithms using a standard sequential method in Unity. This serves as a baseline for comparing the impact of parallelization. Specifically, we first implement the Flocking and Find Nearest algorithms, using a conventional sequential approach without job parallelization. Subsequently, the algorithm is reimplemented using both a multithreaded job system and a single-threaded job system, also enabling the Burst Compiler on job-enabled versions of both algorithms in order to make tests either with and without it.

To evaluate performance, we conduct tests using the Stopwatch library to measure the time, in milliseconds, required to process 100 frames. To focus on the parallelism of our performance measurements, we exclude the initial "warm-up"phase, which consists of the instantiation and initialization steps of the algorithms. This warm-up period represents the setup phase where agents are spawned and the algorithm is prepared for execution, and it does not reflect the steady-state performance. Specifically, the instantiation process involves sequential operations that vary based on the number of agents instantiated. In the Flocking algorithm, this takes relatively less time, as only the agents (boids) are spawned. However, in the Find Nearest algorithm, each Seeker requires the instantiation of a corresponding Target, which increases the initialization overhead. By excluding this initial phase, we focus solely on the performance during the loop execution of the algorithms, providing a clearer representation of their runtime performance enhancement under typical conditions. This approach avoids including sequential ignition and set-up results and ensures that the measurements reflect the true execution time of the core loop of the algorithm. After completing the multithreaded job tests for both the Flocking and Find Nearest algorithms, we perform the same tests for each agent set variation, applying the single-threaded job and sequential no-job variations and Burst Compiler variations. The parallel job-enabled experiment is subsequently repeated for different batch configurations and agent set variations and Burst Compiler derivations. This Batch variation repetition is not applicable to the single-threaded jobs due to the absence of the batch concept in sequential pipelines, and so are made the tests with it both on and off. Each configuration is executed 12 times, with each execution processing 100 frames. Every test executed were executed each considering both algorithms all threads, agents, batch variations, and the sequential executions for both algorithms and their respective single-threaded job-enabled ones, and Burst Compiler on and off variation for job-enabled algorithms.

To further assess the performance enhancements offered by Unity Jobs, we conducted a comprehensive investigation using a modified Unity template that employs the Find Nearest algorithm, and also implementing a Flocking algorithm. Both algorithms consists of three algorithm versions, each representing a progressive iteration:

- **Version 1:** A traditional no-job, sequential script, which represents a standard Unity usage scenario.

- **Version 2:** A single-thread job system, transitioning towards a job-enabled and single-thread paradigm.

- **Version 3:** A multithreading implementation of the algorithms using Unity's multithreading Jobs.

Version 2 and 3 have their execution also made with Burst Compiler on and off for comparison. For better understanding of implementation employing particularities that will be present for each algorithm let us take a look at Table 3 that will displace the characteristics for each algorithm variation.

Table 3 – Comparison of Algorithm Configurations

Algorithm Variant	**Thread Var.**	**Agents Var.**	**Batches Var.**	**Burst Var.**
Multithreaded Job	Yes	Yes	Yes	Yes
Single-Threaded Job (SJ)	N/A	Yes	N/A	Yes
Sequential (S)	N/A	Yes	N/A	N/A

The Burst Compiler, an essential tool for achieving optimal performance in computationally intensive applications, enhances the efficiency of Unity Jobs by leveraging advanced optimization techniques. It enables automatic vectorization, allowing the CPU to process multiple data points in a single instruction, which significantly improves data throughput. Additionally, it optimizes memory access patterns to reduce cache misses and improve data retrieval efficiency. Furthermore, by eliminating unnecessary function calls and removing unused code through inlining and dead code elimination, Burst ensures faster execution times and smaller code sizes. Considering the impact of it, tests will leverage the impact it have on performance. This systematic exploration allows us to evaluate the efficacy of Unity Jobs in various scenarios, ranging from traditional single-threaded scripts to multithreading implementations.

4.1 Data Collection and Measurement

We have ten algorithms variations, all adjusted for equivalency. Job-modified versions, initially, use a BatchSize of 1, then we modify batches, Burst compiler options, we also make use of Stopwatch class from System.Diagnostics of Microsoft for measuring time in order to have a precise time measurement. Every test is conducted in distinct scenes. For scalability tests, we varied the load by changing the number of agents (120, 240, 480, 720, 960, 1200, and 1440) and threads (8, 6, 4, 2, and Sequential No Jobs, then, Single Threaded Job). Each configuration was tested with 12 executions of 100 frames. Batch configurations were 1, 2, 4, 8, 16, 32, 64, and 128 for multithreading jobs. We tested higher agent configurations (1200 and 1440 agents) to evaluate performance scaling beyond the architecture's maximum charge proportion of 960 agents. This comprehensive testing approach provides insights into the algorithms' performance under various conditions. Tests will be conducted in order to leverage out of execution time of each configuration the Speedup and the Efficiency, which Speedup consist on how much of gain the Parallel execution will have against the Sequential version, as we can see by the formula:

$$\text{Speedup} = \frac{\text{Sequential Time}}{\text{Parallel Time}}$$

In order to have fair comparisons, graphs will also show what would be the Speedup if compared to the Single-threaded job version of the Sequential algorithm other than the normal Sequential Mono-Behaviour default algorithm without any Job kind.

The same is also true for Efficiency, graphs will be displace both comparisons of Efficiency for Sequential no-job, and parallel vs Single-Threaded Job-enabled version. Efficiency is the concept of how much of efficiency the Speedup over multi-core, so let it be Speedup divided by the number of cores, as shown on the formula below.

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Number of Cores}}$$

4.2 Hardware, Software, Packages and Operating System Versions

Tests were conducted on the following computer architecture: an AMD Bulldozer FX-8120 3.1 GHz eight-core processor with 32 GB of DRAM. For this experiment, we utilized Unity version 2022.2.0f1. The following packages were employed: Entities 1.0.0-pre.15, Burst 1.8.2, Mathematics 1.2.6, Collections 2.1.0-pre.6, Performance Testing API 2.8.1 (preview), Properties and Properties UI 2.1.0, Scriptable Build Pipeline 1.21.0, Serialization 3.0.0-pre1, and Unity Profiling Core API 1.0.2. The operating system was Ubuntu 22.04.

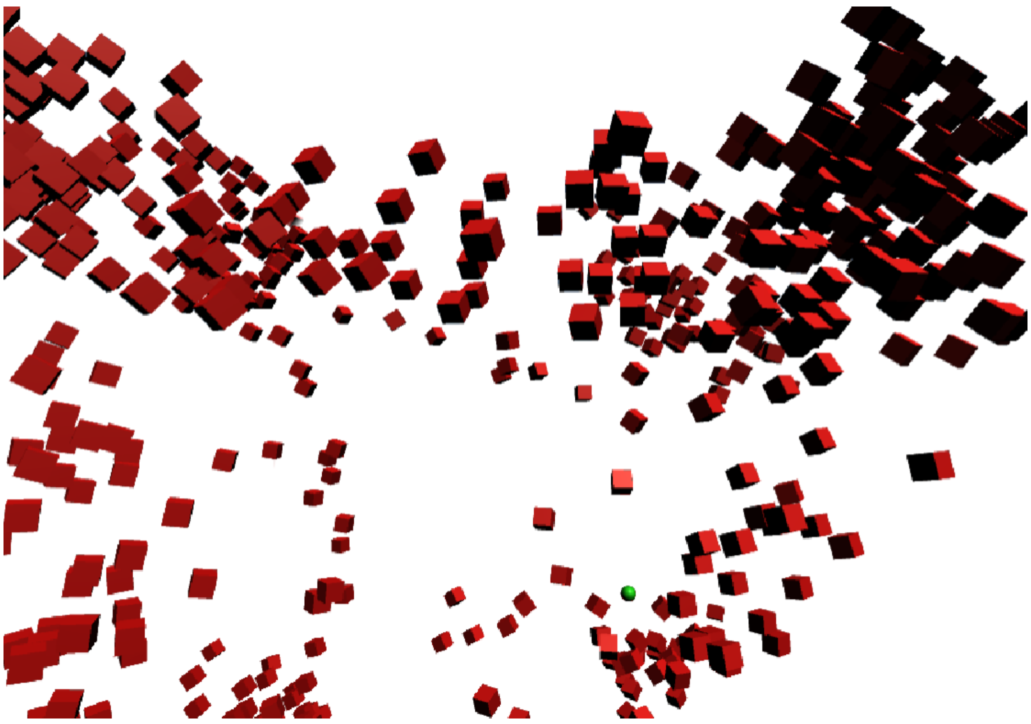
4.3 Algorithms

Video Game algorithms traditionally impose substantial CPU overhead due to their intricate logic processes. Consequently, we selected these specific algorithms as a case study, simulating the demands of real and complex game algorithms scenarios. The choice of both algorithms are based on their common use in games. Craig Reynolds' Flocking algorithm is popular for particle simulations, dynamic movement of flocks, enemy swarm behavior, and fluid simulations. As Find Nearest algorithm is also on Unity's template, it is ideal for performance benchmarking, this algorithm iterates through data to calculate minimum distances—frequently used in games to compute object proximity. The aim is to assess the potential optimization achievable through the Unity Jobs System in mitigating the computational burden associated with these scripts processes and observe their performance improvement. It is important mention that the Single-Thread and Burst Compiler variants implementation will be very similar to Multithreading Job-enabled versions, changing a few lines. The algorithms where chosen based on most used cases scenarios in games, as Flocking is widely used for flocks, NPC groups, or animals hordes, Find Nearest will also be easily compared to any euclidean distances validation needed in any game interaction or measurement either for checking if something is in the right distance, or to check how far is an object to another, both then representing the mosst likely to be used and applied to a game. Not to mention that, both algorithms share a similarity, having a quadratic order loop, where, each agent will iterate through all other agents for each agent, making then at a certain level comparable. Find Nearest also is integral part of Unity Jobs System template, making it a most likely to be a case for performance standards comparison with other algorithms such as Flocking. All algorithms and their variations can be found in the footnote*.

* https://github.com/Patrick-Machado/Unity-Jobs-Research-Algorithms

4.3.1 Flocking

Figure 7 – Flocking Execution in Unity Engine



Source: Author.

On Figure 7 we can visualize how the Flocking algorithm behave in engine. Pioneered by Craig Reynolds [Reynolds 1987], Flocking is a AI algorithm designed to replicate the behavior of organic collectives such as swarms, flocks, or shoals, extending its versatility to simulate diverse entities like dust particles or bubbles. Widely employed in games. This algorithm proves instrumental in generating effects like particle motion or orchestrating swarms of enemy forces. Beyond gaming, the Flocking algorithm has found diverse other applications such physics simulations, and also in film, notably utilized to simulate the procedural animation of wildebeest herds in Disney's "The Lion King" [Wang et al. 2014]. Besides its similarities with Find Nearest, Flocking relies on Unity Physics for some collision and physic related calculus, those calculus are executed on the main thread, but may also run on multiple threads when needed, this kind of algorithm that uses physics provides a realistic comparison of a real in-game possible scenario. Our implementation includes both Sequential and Job-enabled versions, i.e., multithreading and single threading, plus Burst Compiler variants. Although Flocking exhibits dynamic real-time behavior, setting up a controlled test window is crucial for accurate performance measurement. Since boids move freely based on the positions of their neighboring boids, simply running a 100-frame test without direction is insufficient—they would have no clear destination. To address this, our Flocking algorithm directs the boids to follow a target that moves in a predefined looping pattern over 100 frames. This ensures that the boids dynamically organize themselves in space while responding to the target's movement. The boids are initially spawned within a limited radius, and as they interact and navigate, collisions may occur. These collisions trigger physics calculations, adding a higher demand on computational resources. This setup simulates a more realistic scenario where boids not only move freely but must also adapt to dynamic spatial conditions and physical interactions. We can observe the pseudo-code of Flocking Job-enabled on Algorithm 1, and on the sequential no-job version on Algorithm 2. The paralel fraction in Algorithm 1 pseudo-code is the "Schedule(boids.Length, BatchSize);"where the Unity Scheduler will schedule FlockJob divided on BatchSizes declared, where for each unit agent set size there will be scheduled a job of Flock Job (better understood on the pseudo-code "JobStruct (FlockJob)"segment). The data is updated on the first While For loop, from agents to their structs, this struct data then is parallelized on Schedule, then on LateUpdate, the data is back brought to agents sequentially. This AI script is not only more complex due to physics but in terms of implementation, Flocking will have FlockJob as a job to control boids and a Boid job struct to hold data of each boid. In addition, this script also has to duplicate NativeArrays because the data cannot be read and written at the same time. Therefore, it needs to swap arrays to avoid mixing readable and writable arrays, and only then update the boid prefabs visually with the newly processed positions but sequentially (via MonoBehavior). Even though this logic is more complex compared to Find Nearest, which only calculates and stores a Euclidean distance, Flocking has its heavier processing time due to physics calculations.

Algorithm 1: Flocking Job-enabled Pseudocode

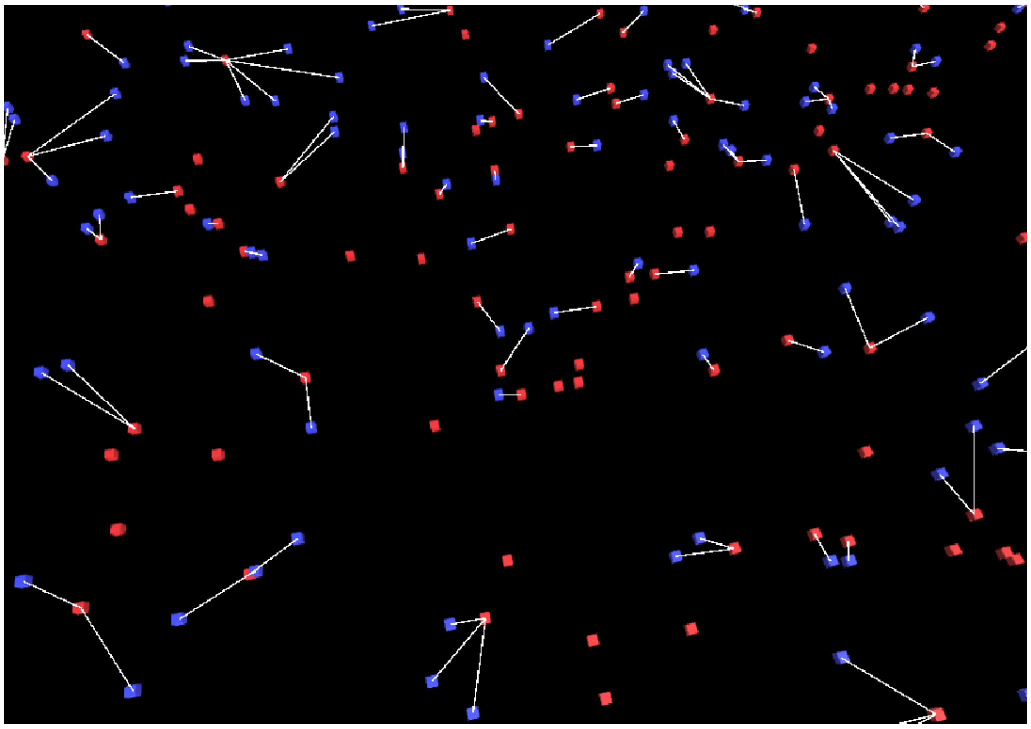
```
Awake()
    for Boids do
        instantiateBoidPrefab();
    end
    for Boids do
        initializeBoidJobStruct();
    end
while currentFrame < maxFrames do
    if frameCount ≥ maxFrames then
        pauseExecution();
    end
    var flockJob = new FlockJobMultiThread;
    for Boids do
        updateBoidPosition();
        updateBoidVelocity();
        updateJobStruct();
    end
    Schedule(boids.Length, BatchSize);
    frameCount++;
    LateUpdate
        flockJobHandle.Complete();
        swapArrays();
        for Boids do
            updateBoidPrefabs();
        end
    end
end
JobStruct (FlockJob)
    for Boids do
        for Boids do
            calculateAlignment();
            calculateSeparation();
            calculateCohesion();
        end
    end
end
JobStruct (Boid)
    position, direction, acceleration;
end
```

Algorithm 2: Flocking Sequential Pseudocode

```
Awake()
    for Boids do
        instantiateBoidPrefab(); updateBoidPositionRotation();
    end
while $currentFrame < maxFrames$ do
    if $frameCount \geq maxFrames$ then
        pauseExecution();
    end
    for Boids do
        for Boids do
            calculateCohesion();
            calculateSeparation();
            calculateAlignment();
            updateBoidPosition();
            updateBoidVelocity();
        end
    end
    frameCount++;
end
```

4.3.2 Find Nearest

Figure 8 – Find Nearest Execution in Unity Engine



Source: Author.

On Figure 8 we can observe how Find Nearest behave in engine. The Find Nearest algorithm, is an integral part of the Unity Jobs System template by default. This algorithm is designed to determine the nearest path between two types of agents: Targets and Seekers. In this study it were modified and implemented to attends the experiments needs. For each Seeker, the algorithm iterates through the Targets, identifying the closest distance and drawing a line between them. The number of Targets corresponds to the number of Seekers in the system and the data, which means that for each Seeker, there is going to be one Target. In this study will refer to each set of Seeker and Target as one Agent. We can observe the pseudo-code of Find Nearest Job-enabled on Algorithm 3, and on the sequential no-job version on Algorithm 4. The parallel fraction in Algorithm 3 pseudo-code is the "Schedule(SeekerPositions.Length, BatchSize);"where the Unity Scheduler will schedule FindNearestJob job struct divided on BatchSizes declared, where for each unit agent on set size there will be scheduled a job of FindNearestJob (better understood on the pseudo-code "JobStruct (FindNearestJob)"segment). The data is updated on the two first While, For loops, from agents to their structs, where each job calculates and storage the lower distance and the reference to its nearest neighbor, this struct data then is parallelized on schedule, then the algorithm draw a line on them, the data is back brought to agents sequentially to update their position and lines drawn.

Algorithm 3: Find Nearest Job-enabled Pseudocode

```
Awake()
    for targets do
        instantiateTargetPrefab();
    end
    for seeker do
        instantiateSeekerPrefab();
    end
    initTargets&SeekersNativeArrays();
while current Frame < maxFrames do
    if frameCount ≥ maxFrames then
        pauseExecution();
    end
    for targetPositions do
        updateTargetPosition();
    end
    for seekerPositions do
        updateSeekerPosition();
    end
    FindNearestJob findJob = new FindNearestJob;
    Schedule(SeekerPositions.Length, BatchSize);
    findHandle.Complete();
    for seekerPositions do
        drawLine();
    end
    frameCount++;
    for targetsPrefabs do
        updateTargetsPosition();
    end
    for seekersPrefabs do
        updateSeekersPosition();
    end
end
JobStruct (FindNearestJob)
    for seekers do
        for targets do
            calculateDistance();
        end
    end
end
```

Algorithm 4: Find Nearest Sequential Pseudocode

```
Awake()
    for $targets$ do
        instantiateTargetPrefab();
    end
    for $seeker$ do
        instantiateSeekerPrefab();
    end
while $current\ Frame < maxFrames$ do
    if $frameCount \geq maxFrames$ then
        pauseExecution();
    end
    for $seekers$ do
        for $targets$ do
            calculateDistance();
        end
        drawLine();
    end
    for $targetPrefab$ do
        updateTargetsPosition();
    end
    for $seekerPrefab$ do
        updateSeekersPosition();
    end
    frameCount++;
end
```

5 RESULTS

The tests yielded promising results, demonstrating higher performance with the multithreading Job-based versions. The tests with Burst Compiler enabled proved to be better than not using it, as we can observe further ahead on the Execution Time section. The visual representation of the results for both the Flocking and Find Nearest algorithms, with Burst Compiler enabled across various configurations and scalability variations, is provided in Tables 5 and 7, the graphics in this study will always compare the speedup of multithreaded job-enabled against normal sequential algorithm with no-job-enabled and also multithreaded job-enabled versus single-threaded job-enabled version, regardless if it is with or not Burst Compiler, and will be covering both weak and strong scalability scenarios.

5.1 Execution Time

Tests were made on every algorithm both with Burst Compiler on and off, as we can observe at results of Execution Time isolated Batch 1 of algorithms without Burst Compiler and its deviation. For Flocking we can observe on Tables 4, and for Find Nearest, Table 6, where we can see its Execution Time and deviation.

Tables 5 and 7 present the Flocking job-enabled and Burst-enabled execution time results with its Execution Time and following deviation. Smaller agent configurations take less advantage of parallelism, though noticeable improvements begin to appear at 480 and 720 agents and beyond. As the agent count scales up, significant improvements are observed compared to the No Jobs Sequential (S) version. The reason is that when compared to the Sequential Job-disabled (S), the Speedup will increase substantially because of the technology shift; when speaking of Job-enabled version, the technology of the data structure is way lighter, not to mention the fact that the parallel cost will not surpass the sequential fraction of the frame in this particular case of study, resulting in the Single-Threaded Job-enabled (SJ) with a horizontal linear behavior on speedups in contrast with the super linear of the Multithreaded Job-enabled.

If comparing the results of Flocking 1-Batch Burst Compiler off, Table 4 to Flocking 1 Batch Burst Compiler-enabled Table 5, we can clearly see a sample of performance difference, that is a phenomenon that happens on every other Batch set if compared to the versions with Burst Compiler-enabled, even on Find Nearest as the samples of Tables 4 and 7 also shows. This improvement is even noticeable on Single-Threaded Job-enabled versions.

When comparing the single-thread Job version to the Sequential version, performance improvements are only evident in larger agent configurations. In smaller sets, such as 120 agents with Burst on, there is no noticeable performance speedup. However, on larger sets, the improvements are substantial. It is important to keep in mind that the Sequential experiments do not employ any Burst Compiler, and on Tables 5 and 7, Sequential traditional mono behavior (S), will displace the same data of its Burst-disabled version for better visualization purposes only, but they do not employ any Burst Compiler technology.

5.2 Speedup

Speedup is a critical metric for evaluating performance, defined as the ratio of sequential execution time to parallel execution time Pacheco 2011, Hager 2010, Hill e Marty 2008.

As previously mentioned, experiments were conducted with and without the Burst Compiler enabled. Tests were performed across multiple batch sizes (1, 2, 4, 8, 16, 32,

Table 4 – Flocking 1-batch, Burst-disabled - Execution Time (ms)

Threads	S		SJ		2t		4t		6t		8t	
Agents	Exec.	Dev.	Exec.	Dev.	Exec.	Dev.	Exec.	Dev.	Exec.	Dev.	Exec.	Dev.
120	143.35	8.01	315.50	3.45	133.64	0.44	132.77	0.32	132.40	0.40	132.42	0.36
240	218.36	3.52	368.84	2.41	146.76	0.88	143.28	0.67	142.81	0.62	142.35	0.44
480	397.82	2.23	418.42	4.16	184.53	1.10	173.52	1.07	169.58	1.20	166.82	1.11
720	696.36	3.66	484.28	10.06	225.16	1.29	206.38	0.80	199.64	1.15	195.87	0.82
960	1107.32	4.80	557.22	15.38	268.84	2.46	238.38	2.04	225.89	1.35	219.32	0.84
1200	1625.08	6.12	653.63	23.67	309.86	1.36	269.72	3.23	260.31	4.45	248.85	3.90
1440	2249.92	8.47	770.03	38.27	364.51	1.79	301.59	2.04	280.95	1.65	271.37	1.43

Table 5 – Flocking 1-batch, Burst-enabled - Execution Time (ms)

Threads	S		SJ		2t		4t		6t		8t	
Agents	Exec.	Dev.	Exec.	Dev.	Exec.	Dev.	Exec.	Dev.	Exec.	Dev.	Exec.	Dev.
120	143.35	8.01	183.41	9.11	109.32	1.45	108.81	2.53	104.06	1.11	103.64	1.14
240	218.36	3.52	192.42	13.34	126.21	3.85	125.93	1.83	115.30	2.18	115.44	1.58
480	397.82	2.23	215.02	10.79	158.72	3.26	151.03	5.98	146.88	4.61	154.31	7.32
720	696.36	3.66	251.51	10.71	175.53	4.98	174.95	6.77	174.11	4.51	170.81	4.38
960	1107.32	4.80	276.14	10.99	197.94	3.97	194.91	3.42	195.12	4.90	203.53	2.06
1200	1625.08	6.12	301.47	12.88	242.33	4.53	230.93	3.83	228.30	3.33	225.94	3.81
1440	2249.92	8.47	343.08	13.30	259.08	3.56	250.10	2.34	248.62	4.02	248.94	5.71

64, 128), agent set sizes (120, 240, 480, 720, 960, 1200, 1440), and thread configurations (sequential, 2, 4, 6, and 8 threads), in addition, also were made Single-Threaded Job-enabled tests.

The results demonstrated consistent performance improvements when the Burst Compiler was enabled. Across all configurations and both algorithms, enabling the Burst Compiler led to higher speedup values. For the Flocking algorithm, the Burst-disabled version achieved an average speedup of 8×, whereas the Burst-enabled version achieved an average speedup of 9×. Similarly, for the Find Nearest algorithm, the Burst-disabled version averaged a speedup of 10×, compared to 12× with the Burst Compiler enabled. These results indicate that leveraging the Burst Compiler significantly enhances performance across all tested scenarios.

Figures 9 and 11 will displace the Burst-disabled Speedups and Figures 10 and 12 illustrate the Burst-enabled Speedup observed for all configurations of the Flocking and Find Nearest algorithms of 1-batch, respectively. Notably, the results highlight the performance benefits of enabling the Burst Compiler. We also can observe a distinct behavior on Speedup relative to Single-threaded job enabled on Figures 9 and 11, where it has a bigger Speedup to when Burst is enabled (Figures 10 and 12), this is because the Single-threaded Job-enabled execution time, raises significantly as it scales on Burst-disabled execution time, that can be observed on Tables 4 and 6, Burst implies optimization even when Single-threaded solutions.

We also conducted strong scalability tests (varying threads for a fixed agent and batch set sizes) and weak scalability tests (varying the number of agents and batches for and varying the number of threads). Even when compared to the best-performing configuration without the Burst Compiler (Figures 23 and 24), the Burst-enabled versions consistently outperformed both the sequential no-job execution (S) and the

Table 6 – Find Nearest 1-batch, Burst-disabled - Execution Time (ms)

Threads	S		SJ		2t		4t		6t		8t	
Agents	Exec.	Dev.	Exec.	Dev.	Exec.	Dev.	Exec.	Dev.	Exec.	Dev.	Exec.	Dev.
120	92.07	1.42	248.03	0.62	85.69	0.32	85.31	0.30	85.20	0.26	84.90	0.38
240	126.49	1.15	202.46	0.81	88.97	0.53	87.57	0.39	87.01	0.43	86.58	0.18
480	227.48	0.80	255.95	1.01	102.57	0.72	95.76	0.66	93.91	0.32	92.42	4.46
720	387.84	1.26	284.34	0.64	119.46	0.62	104.28	0.66	99.56	0.71	96.47	0.45
960	608.99	1.81	323.75	1.34	138.24	0.68	116.97	0.84	108.19	0.59	103.43	0.41
1200	891.56	1.28	369.22	1.21	158.47	0.89	128.40	0.69	117.42	0.65	110.82	0.42
1440	1239.56	2.30	426.80	0.42	181.96	0.89	142.02	0.85	128.30	1.74	123.46	2.82

Table 7 – Find Nearest 1-batch, Burst-enabled - Execution Time (ms)

Threads	S		SJ		2t		4t		6t		8t	
Agents	Exec.	Dev.	Exec.	Dev.	Exec.	Dev.	Exec.	Dev.	Exec.	Dev.	Exec.	Dev.
120	92.07	1.42	108.08	0.56	75.50	0.89	75.65	0.68	75.28	1.40	75.54	0.76
240	126.49	1.15	109.89	0.58	78.43	1.31	78.26	1.01	77.56	1.18	78.23	1.11
480	227.48	0.80	114.74	0.45	84.56	0.96	83.88	1.06	83.62	1.35	82.99	1.00
720	387.84	1.26	119.49	0.41	90.53	1.04	89.19	1.04	88.91	1.30	88.87	1.31
960	608.99	1.81	126.34	0.65	98.65	1.54	96.08	4.72	94.97	4.82	94.17	1.12
1200	891.56	1.28	133.67	0.69	105.73	1.20	102.79	0.98	101.73	1.42	100.17	0.92
1440	1239.56	2.30	141.08	1.50	113.76	1.22	109.89	1.11	108.25	1.22	100.74	8.09

Single-Threaded Job-enabled execution (SJ).

In conclusion, enabling the Burst Compiler is highly beneficial, as it maximizes the potential of Unity Jobs. The subsequent analysis will focus on results obtained with the Burst Compiler enabled to fully leverage its advantages.

Figure 9 – Flocking 1-batch Job, Burst-disabled - Speedup

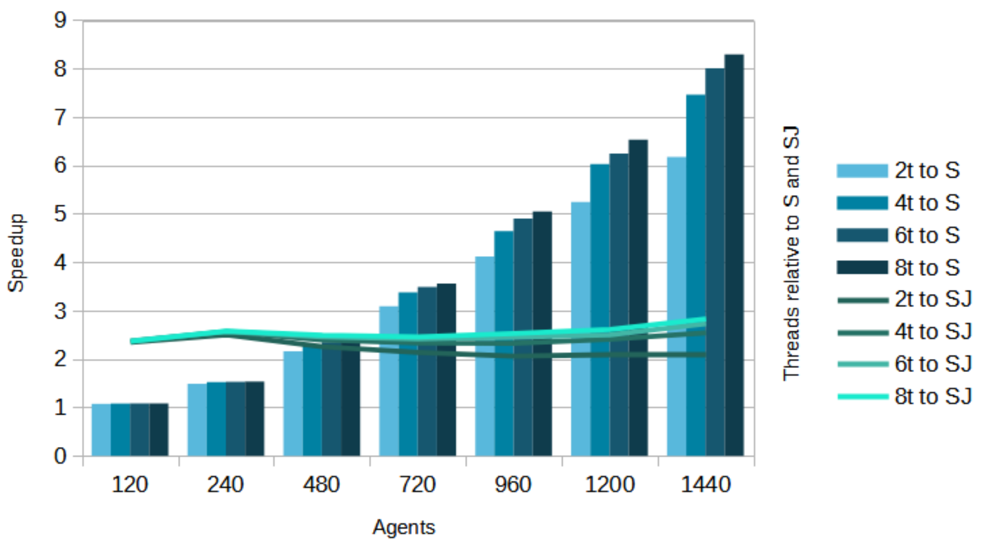


Figure 10 – Flocking 1-batch Job, Burst-enabled - Speedup

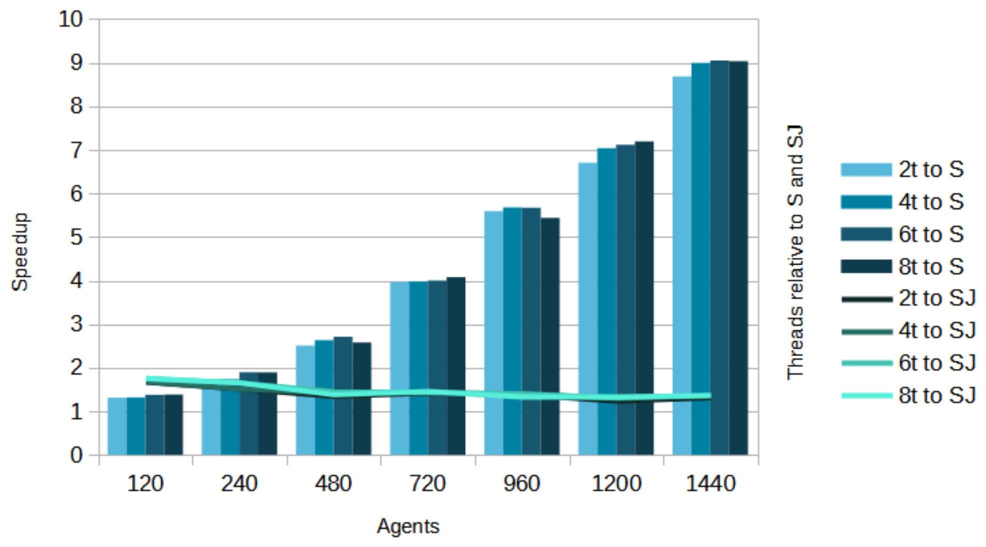


Figure 11 – Find Nearest 1-batch Job, Burst-disabled - Speedup

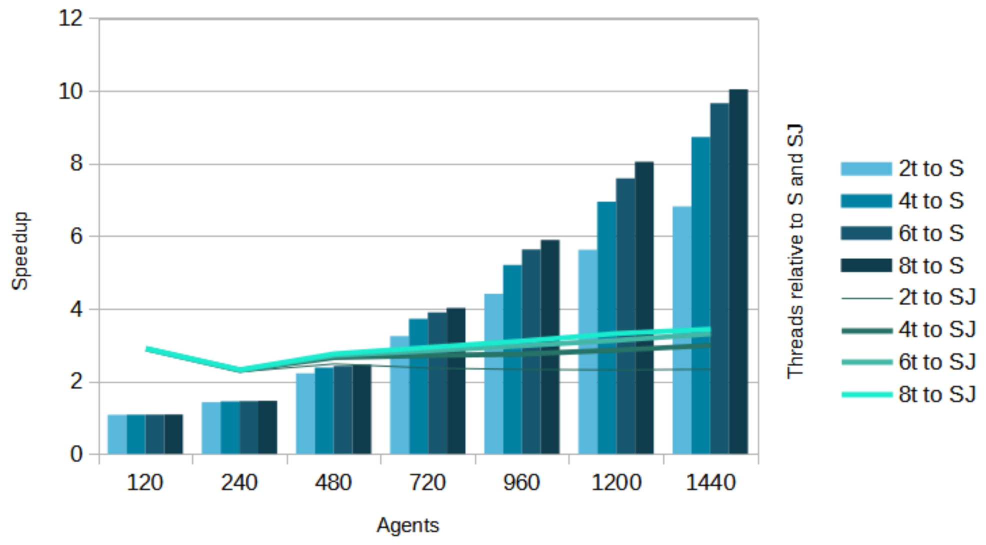
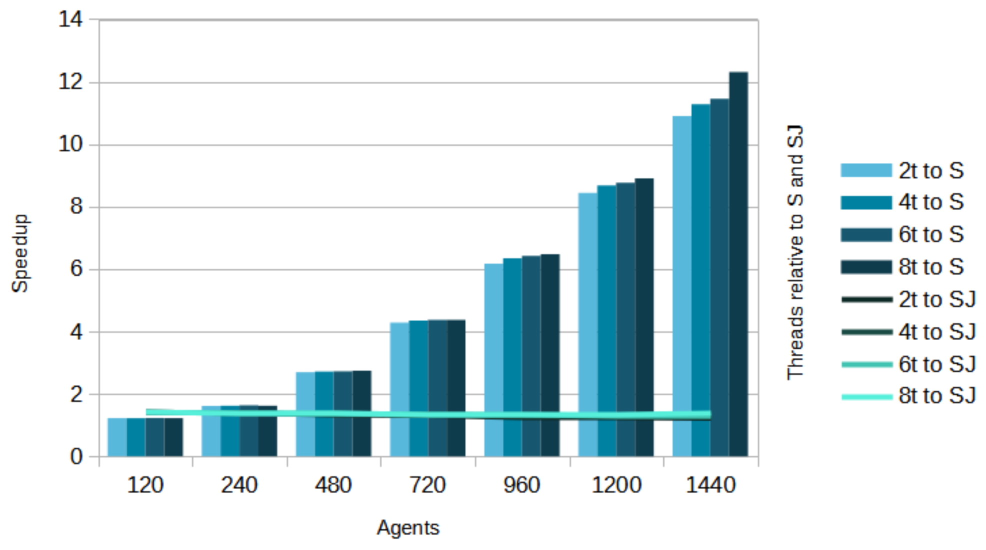


Figure 12 – Find Nearest 1-batch Job, Burst-enabled - Speedup



According to the Flocking results, performance improvements become increasingly evident with higher agent configurations. This behavior can be explained by the nature of the speedup formula, where the sequential execution time is divided by the parallel execution time. In scenarios with a larger number of agents, the sequential execution time increases significantly due to the overhead of managing numerous MonoBehaviour instances of boids and the additional physics computations required.

Conversely, with fewer agents, the sequential execution time is disproportionately high relative to the parallel execution time. This is because parallel execution distributes the workload across threads and uses a more optimized data structure compared to traditional MonoBehaviour-based sequential algorithms that neither utilize the Unity Jobs System nor the Burst Compiler, making the Sequential (S), be way less efficient.

Both algorithm exhibits a distinction on performance improvement trend. This is due to its less complex operations on Find Nearest and a heavy cost in physics processing on Flocking side, allowing it both to achieve steady performance Speedup improvements mainly with the largest datasets.

5.3 Efficiency

Figure 13 and 14 illustrate that efficiency is highest with lower thread configurations and gradually decreases as the number of threads increases.

Figure 13 – Flocking 1-batch Job, Burst-enabled - Efficiency

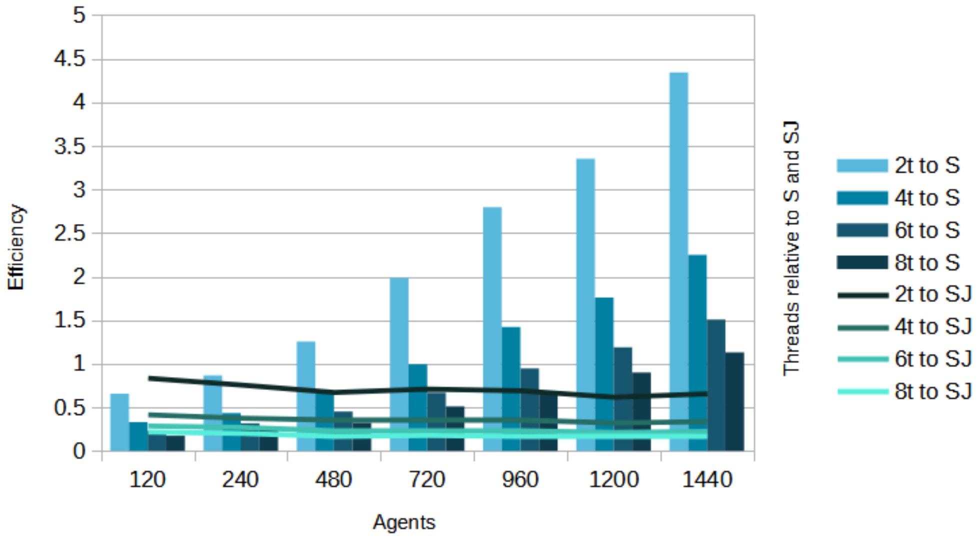
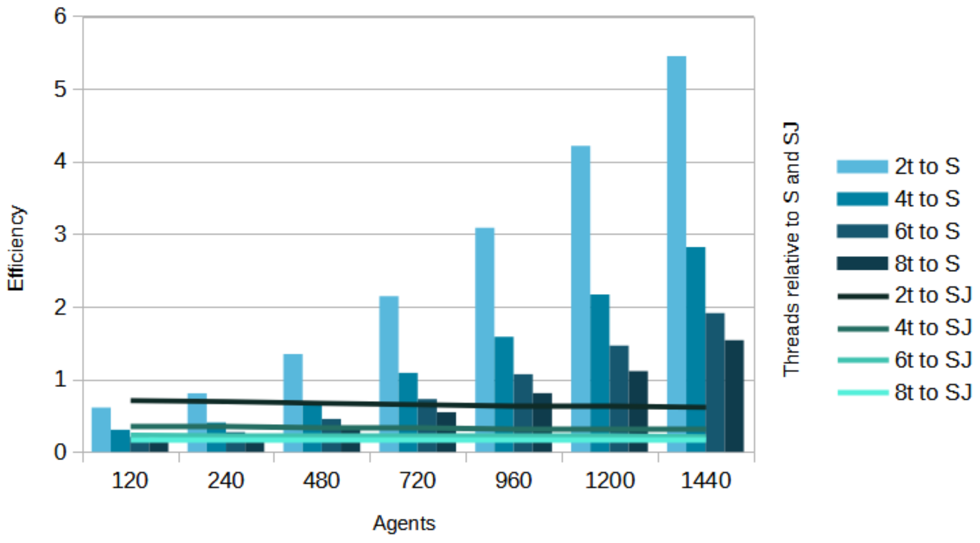


Figure 14 – Find Nearest 1-batch Job, Burst-enabled - Efficiency



In the configuration of 120 agents with 2 threads, the speedup was 1.31, thus the efficiency was 0.66, as shown in Figure 13. As the thread count increases, the efficiency per thread drops, which can be attributed to the increased overhead associated with managing a larger number of threads. This trend is observed across different configurations, with the efficiency for Find Nearest similarly calculated and depicted in Figure 14.

5.4 Batch variations

The initial tests were conducted using only a batch size of 1 before diversify to other batch sets. We will now examine how it behaves with different Batch variations. To explore this, we ran the same tests with different batch sizes, specifically 2, 4, 8, 16, 32, 64, and 128, across various agent configurations (120, 240, 480, 720, 960, 1200, and 1440) and thread variations (2, 4, 6, 8). The concept of batch size does not apply to sequential executions, including Unity Jobs single-threaded implementations. On sequential job-based, the tasks are managed by the scheduler, but without any control of batch size allowed, unlike in multithreading implementations, where the tasks are also controlled by the scheduler but batch size can be explicitly defined. It's important to note that in multithreaded job-enabled versions, such as it is on Single-Threaded Job-enabled the Unity scheduler is responsible for distributing jobs among threads, automatically determining the best way to leverage parallelism by spreading tasks across available worker threads. This dynamic distribution can result in some threads occasionally remaining idle during certain frames, depending on how the scheduler allocates jobs. Control over these idle threads is limited, but increasing the complexity of job tasks or adjusting the batch size can help optimize thread utilization. Batch sizes can be explicitly set in multithreaded job-enabled versions but are not applicable in single-threaded scenarios, including the job-enabled.

The Unity Jobs System's scheduler itself is not fully open source; it operates internally within the engine's core, and only certain aspects of it are exposed through the Unity C# Job System APIs. These APIs allow developers to define, manage, and schedule jobs, but the detailed mechanics of how jobs are queued, prioritized, and executed across threads remain proprietary and hidden from the public codebase.

To determine the optimal configurations, each batch and agent configuration was tested under identical conditions: 12 executions per configuration, each processing 100 frames, with execution time measured in milliseconds per frame, just as in the initial batch size of 1 tests. In most scenarios, the 8-thread configuration performed better across the different thread variations, with similar results observed for the 1440-agent configuration. Tables 8 and 9 presents the results of these tests, which help identify the best-fitting batch configuration for the 8-thread setup and distinguish between agent and batch variations for both algorithms.

Table 8 – Flocking 8-thread Job and batch variations, Burst-enabled - Execution Time (ms)

Batches	1b		2b		4b		8b		16b		32b		64b		128b	
Agents	Exec.	Dev.	Exec.	Dev.	Exec.	Dev.	Exec.	Dev.	Exec.	Dev.	Exec.	Dev.	Exec.	Dev.	Exec.	Dev.
120	103.64	1.14	152.60	0.77	111.75	0.53	140.82	0.29	138.64	4.16	130.69	0.46	161.17	7.10	165.42	3.80
240	115.44	1.58	176.22	2.34	120.66	0.61	142.95	5.20	140.49	3.24	144.48	6.27	172.33	6.17	167.94	6.36
480	154.31	7.32	184.24	0.62	142.42	0.78	164.07	3.36	165.62	5.10	171.10	2.95	191.26	6.65	194.93	5.36
720	170.81	4.38	211.01	6.82	171.31	6.67	191.40	2.20	192.01	1.82	192.94	1.91	204.32	0.45	206.50	4.48
960	203.53	2.06	234.42	5.74	191.42	1.20	213.06	0.71	213.39	1.17	210.57	0.95	225.73	6.42	222.76	1.01
1200	225.94	3.81	253.54	7.55	211.13	1.19	232.66	1.65	235.16	1.36	231.73	1.48	247.34	3.73	249.47	6.54
1440	248.94	5.71	265.60	5.74	232.91	3.23	257.54	3.49	256.77	3.90	256.68	4.74	268.85	6.93	271.49	6.88

As we can see in Table 8, 1 batch and 4 batches configurations have the better performance, being 4 the highest speedup from Figure 15. In addition, 4 batches configuration reached its peak with 9.66 of speedup.

On Find Nearest, Table 9, the peak was also reached by the 4 batches on 1440 agents configuration, achieving the speedup of 15.06.

From the Flocking results in Table 8, we can extract the speedup information shown in Figure 15. We observed a distinct behavior where the 4 batches configuration have a stand-up over the others on 1440 agents configuration. Thus, for this algorithm, the best configuration of batches is 4. On the Find Nearest side, though, we observe in Figure 16 that the configuration of 4 batches, 1440 agents, is also the best fit for this problem. It is

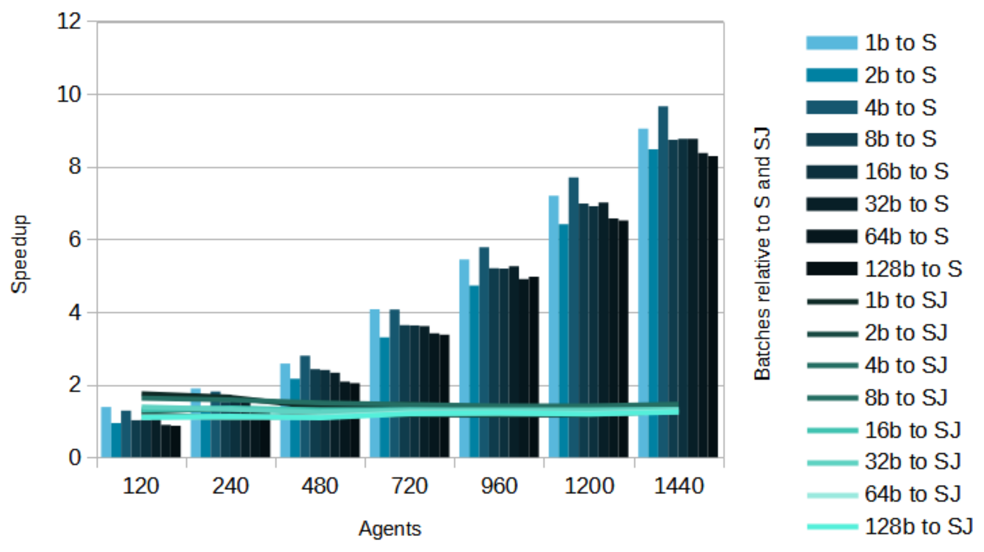
Table 9 – Find Nearest 8-thread Job and batch variations, Burst-enabled - Execution Time (ms)

Batches	1b		2b		4b		8b		16b		32b		64b		128b	
Agents	Exec.	Dev.	Exec.	Dev.	Exec.	Dev.	Exec.	Dev.	Exec.	Dev.	Exec.	Dev.	Exec.	Dev.	Exec.	Dev.
120	75.54	0.76	101.70	4.67	62.44	1.29	89.43	3.35	83.22	0.36	83.65	0.15	105.82	4.41	108.63	0.53
240	78.23	1.11	109.69	0.47	64.29	0.22	90.13	2.57	84.91	0.33	85.53	0.68	110.32	0.44	106.29	4.51
480	82.99	1.00	103.09	1.77	66.88	0.20	88.81	2.58	90.08	2.23	93.84	0.34	104.80	3.82	109.44	5.29
720	88.87	1.31	107.51	3.81	67.09	0.47	96.77	3.15	93.33	1.43	93.20	0.47	108.23	4.07	111.66	5.21
960	94.17	1.12	117.42	3.99	73.28	0.76	99.82	3.11	98.32	1.11	98.23	1.06	118.41	4.01	118.50	4.80
1200	100.17	0.92	117.66	5.67	76.96	0.65	103.37	2.77	102.26	1.00	102.87	0.70	117.85	5.52	120.65	5.24
1440	100.74	8.09	124.40	5.99	82.32	1.16	107.44	3.28	108.18	2.01	108.52	2.13	125.33	5.58	125.19	11.14

also noticeable the efficiency improvement on Figures 19, 20, 21 and 22.

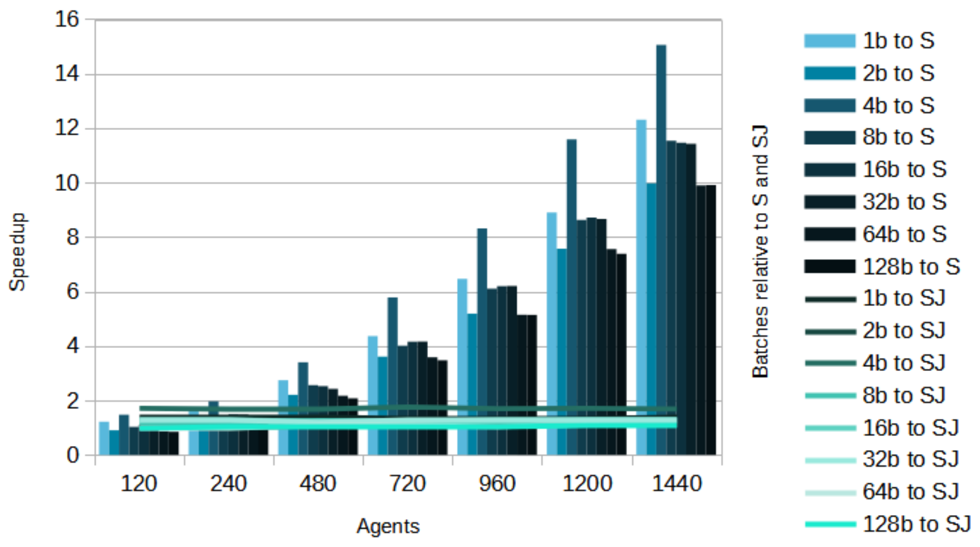
The unity documentation* underscores the importance of tailoring the batch size to specific needs. The documentation suggests that when there is a significant amount of work in each iteration, a batch size of 1 may be the best fit. Another common additional recommendation is 16 batches, and up to 128, the ideal batch sizes. They also encourage using the power of 2 to find the specific batch size, as every problem has an unique requirement.

Figure 15 – Flocking 8-thread Job, Burst-enabled - Speedup



* https://docs.unity3d.com/ScriptReference/Unity.Jobs.IJobParallelFor.html

Figure 16 – Find Nearest 8-thread Job, Burst-enabled - Speedup



Let us delve into the isolation of agents for each algorithm. In Figure 17, we can observe closer the behavior of 4 batches if compared to other batches' configurations on the Flocking algorithm on a set of 1440 agents. The same can be seem better in Figure 18 but this time with the batch for Find Nearest compared to other batch configurations.

For better visualization purposes, let us look at the data from the best scenario for each algorithm as we can observe in Table 10 for Flocking, and Table 11 for Find Nearest best performance. On Tables 12 and 13 we can observe in percentage how was the actual impact of Burst Compiler over the execution time of Flocking and Find Nearest 4-batch workload sets. Let it be the Percentage Gain the following formula:

$$\text{Percentage Gain} = \frac{\text{Burst-disabled} - \text{Burst-enabled}}{\text{Burst-disabled}} \times 100$$

Figure 17 – Flocking 1440-agent Job, Burst-enabled - Speedup

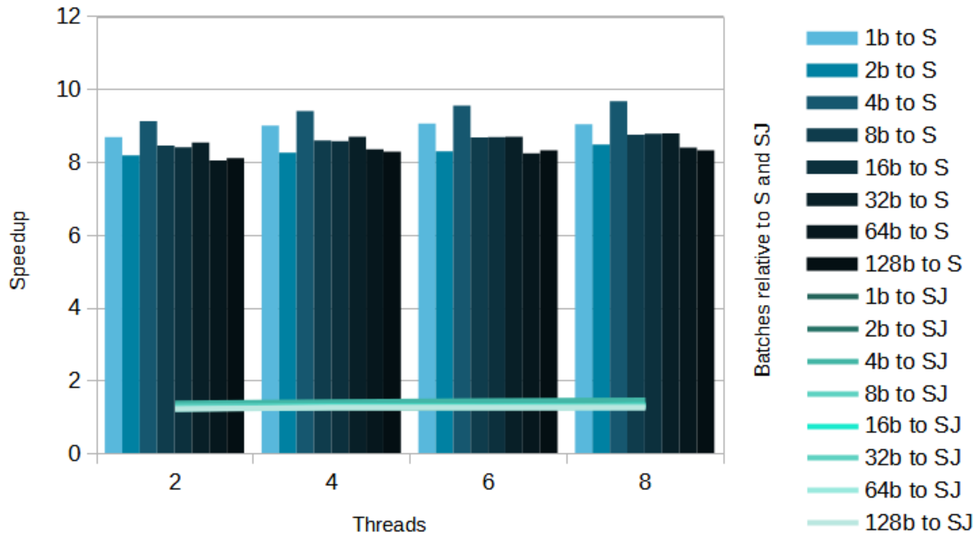


Figure 18 – Find Nearest 1440-agent Job, Burst-enabled - Speedup

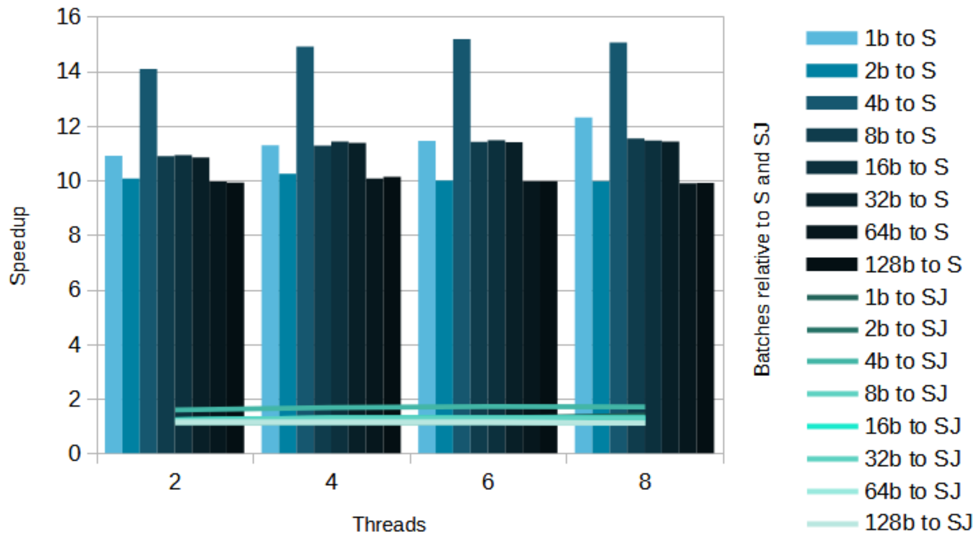


Table 10 – Flocking 4-batch Job, Burst-enabled - Execution Time (ms)

Threads	2t		4t		6t		8t	
Agents	Exec.	Dev.	Exec.	Dev.	Exec.	Dev.	Exec.	Dev.
120	103.69	0.73	103.50	0.34	109.06	3.93	111.75	0.53
240	115.79	4.62	111.45	0.51	115.23	4.26	120.66	0.61
480	146.21	4.01	147.61	5.62	142.77	2.91	142.42	0.78
720	170.60	2.61	167.66	2.82	169.01	5.70	171.31	6.67
960	196.87	1.13	193.67	1.72	191.76	1.90	191.42	1.20
1200	221.34	1.77	213.38	1.38	212.85	1.54	211.13	1.19
1440	246.94	5.41	239.59	5.45	235.95	5.01	232.91	3.23

Table 11 – Find Nearest 4-batch Job, Burst-enabled - Execution Time (ms)

Threads	2t		4t		6t		8t	
Agents	Exec.	Dev.	Exec.	Dev.	Exec.	Dev.	Exec.	Dev.
120	58.97	0.18	62.94	0.29	63.08	0.31	62.44	1.29
240	60.45	0.21	60.31	0.32	62.47	1.96	64.29	0.22
480	66.20	1.71	63.73	0.29	65.79	1.75	66.88	0.20
720	69.10	1.07	68.78	1.29	69.87	0.62	67.09	0.47
960	75.37	0.55	73.59	0.67	72.96	0.77	73.28	0.76
1200	81.21	0.88	78.18	0.45	77.34	0.51	76.96	0.65
1440	88.01	1.99	83.18	1.07	81.67	0.76	82.32	1.16

Figure 19 – Flocking 8-thread Job, Burst-enabled - Efficiency

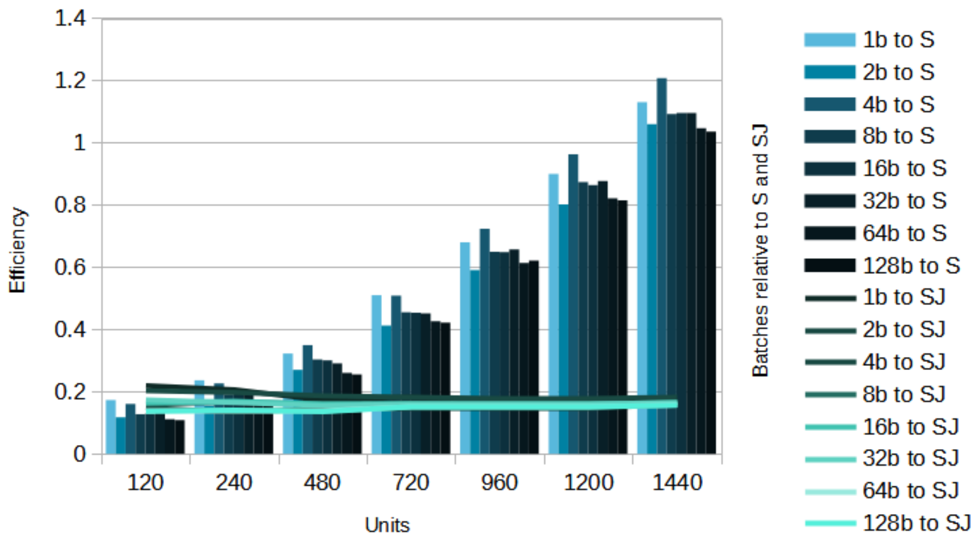


Table 12 – Flocking 4-batch Percentage Gain of Burst-enabled vs Burst-disabled execution time

Batch Size	1 SJ	2t	4t	6t	8t
120	41.9%	27.7%	22.9%	17.6%	21.5%
240	47.8%	24.0%	23.3%	24.4%	20.6%
480	48.6%	20.9%	17.5%	18.0%	15.3%
720	48.1%	24.0%	18.9%	16.3%	13.8%
960	50.5%	26.6%	19.0%	16.1%	14.4%
1200	53.9%	28.3%	22.6%	18.0%	17.2%
1440	55.4%	31.9%	20.9%	16.9%	16.1%

Table 13 – Find Nearest 4-batch Percentage Gain of Burst-enabled vs Burst-disabled execution time

Batch Size	1 SJ	2t	4t	6t	8t
120	56.4%	35.0%	28.7%	25.9%	27.2%
240	45.7%	35.2%	33.4%	28.0%	27.5%
480	55.2%	35.6%	33.0%	30.6%	30.2%
720	58.0%	42.5%	35.4%	30.4%	34.0%
960	60.9%	45.7%	37.0%	33.3%	32.0%
1200	63.8%	48.5%	39.7%	35.3%	32.8%
1440	67.0%	51.9%	41.9%	36.8%	34.5%

Figure 20 – Find Nearest 8-thread Job, Burst-enabled - Efficiency

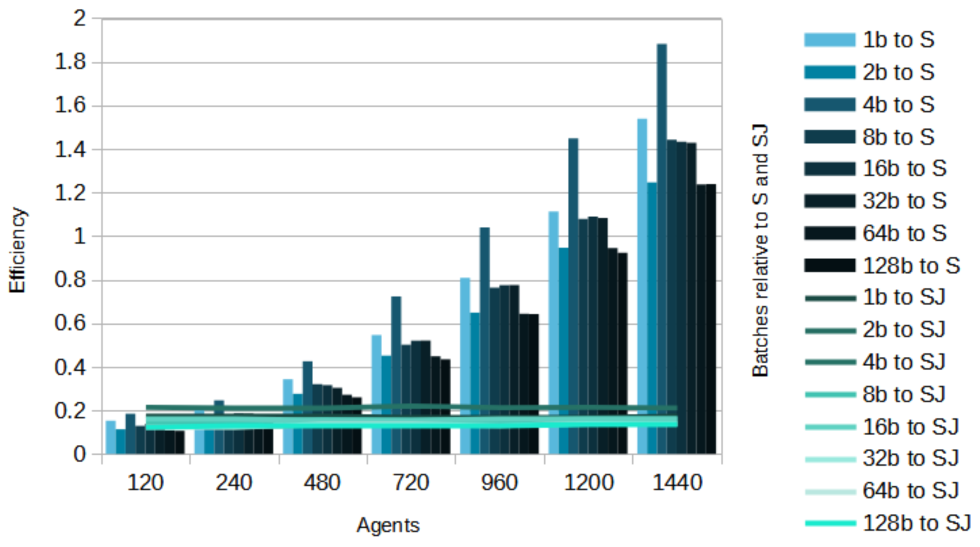


Figure 22 – Find Nearest 1440-agent Job, Burst-enabled - Efficiency

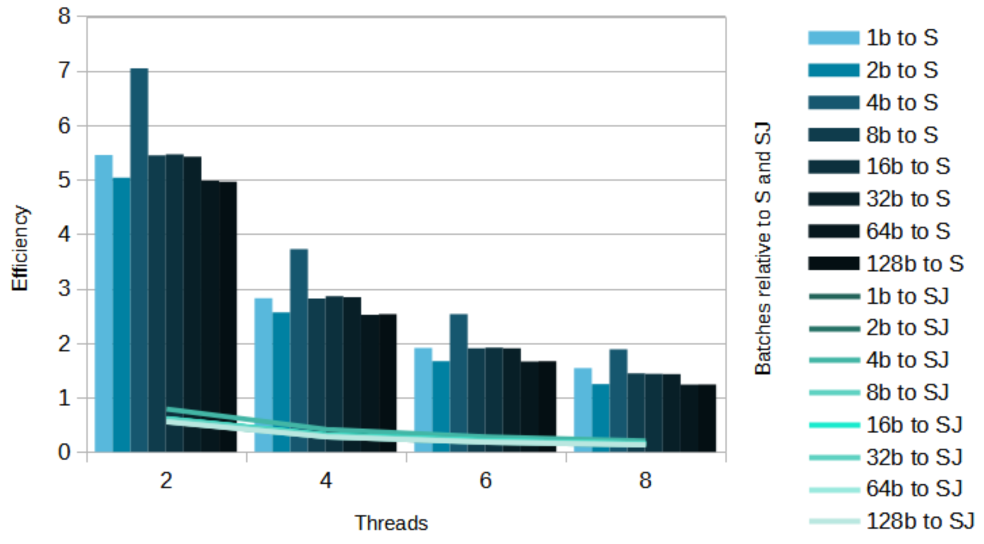
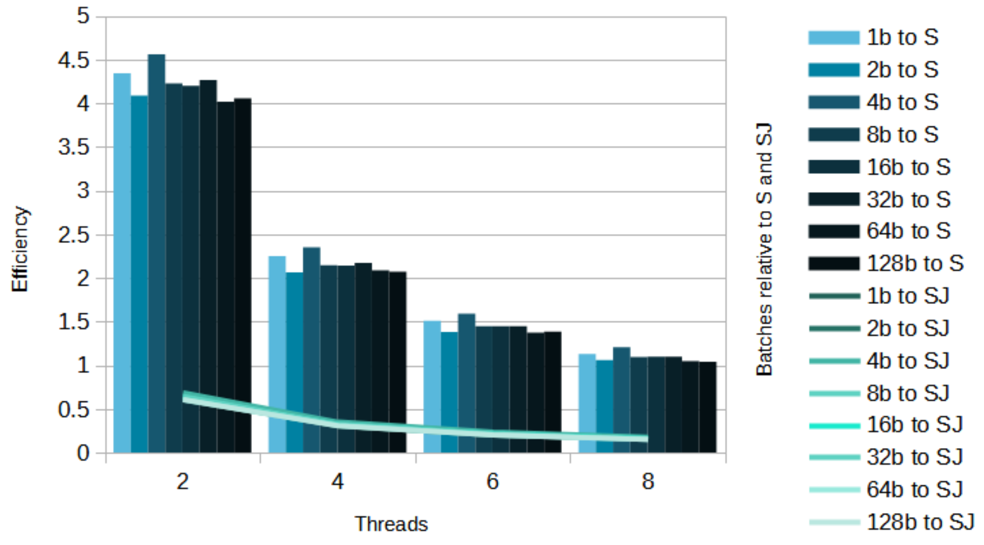


Figure 21 – Flocking 1440-agent Job, Burst-enabled - Efficiency



5.5 Final Remarks

Given the results, it is evident that the parallel job-enabled version offers impressive speedups when compared to the standard MonoBehaviour sequential algorithms. However, even the single-threaded job-enabled versions show significant performance improvements. This raises the question: what factors, aside from parallelism, contribute to such gains?

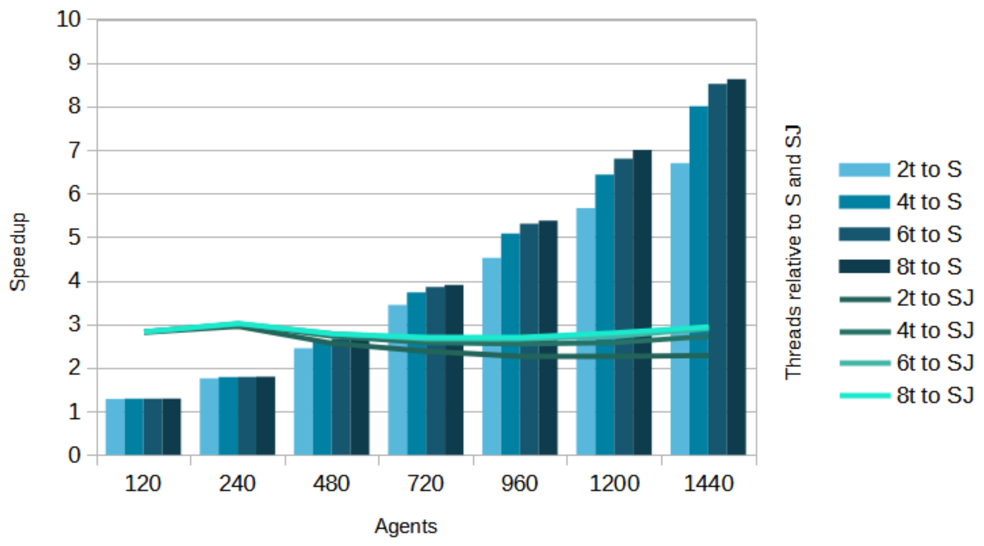
The Unity Jobs System combined with the Burst Compiler optimizes performance in several critical ways beyond parallelism. First, the Jobs System allows for more efficient task scheduling and execution by offloading work from the main thread. Even in a single-threaded job, this removes the overhead associated with Unity's traditional

MonoBehaviour-based architecture, which involves frequent context switching and event-driven updates. The jobified system enables more fine-grained control over when and how tasks are executed, leading to reduced overhead and better performance.

The Burst Compiler further boosts performance by applying low-level optimizations that directly impact how the code is executed on the CPU. Burst compiles C# code into highly optimized machine code, which benefits even single-threaded jobs.

Even without parallel execution, these optimizations significantly reduce the processing time for each frame, making jobified algorithms inherently faster and more efficient than their MonoBehaviour counterparts. But how much? To leverage this answer, the same tests where conducted but now without Burst Compiler, and the results showed an average 8x of speedup for Flocking tests of multithreaded job-disabled compared to normal sequential, with its best case happening on batch 32 with 8.63 of speedup, and 2.95 of speedup if compared the multithreaded job-enabled version against the single-threaded job-enabled version, Figure 23. Find Nearest though, had an average 10x of speedup, if compared the multithreaded job-enabled version against the normal sequential, with its best case being 11.91 happening also on batch 32, and 4.10 if comparing the multithreaded job-enabled version against the single-threaded job-enabled version, Figure 24.

Figure 23 – Flocking 32-batch Job, Burst-disabled - Speedup



Looking at Figure 25, and Figure 26, we can observe that the algorithm tasks represented in the images by the blue bar got allocated completely on the main-thread using the unity MonoBehaviour class, reaching a time for that specific frame to be executed of 2031.72 ms on Flocking and 1172.27 ms on Find Nearest to be processed.

When contrasting the sequential execution screenshots from the Unity Profiler tool (Figures 25 and 26) with the parallel, job-enabled execution profiles (Figures 27 and 28), significant performance improvements are evident. For the Flocking algorithm, Figure 27 reveals a substantial amount of physics calculations, represented by orange bars, which benefit greatly from parallelism in multithreaded scenarios. In contrast, in the sequential version, collision detection is processed on the main thread, particularly when not utilizing the DOTS framework.

Figure 24 – Find Nearest 32-batch Job, Burst-disabled - Speedup

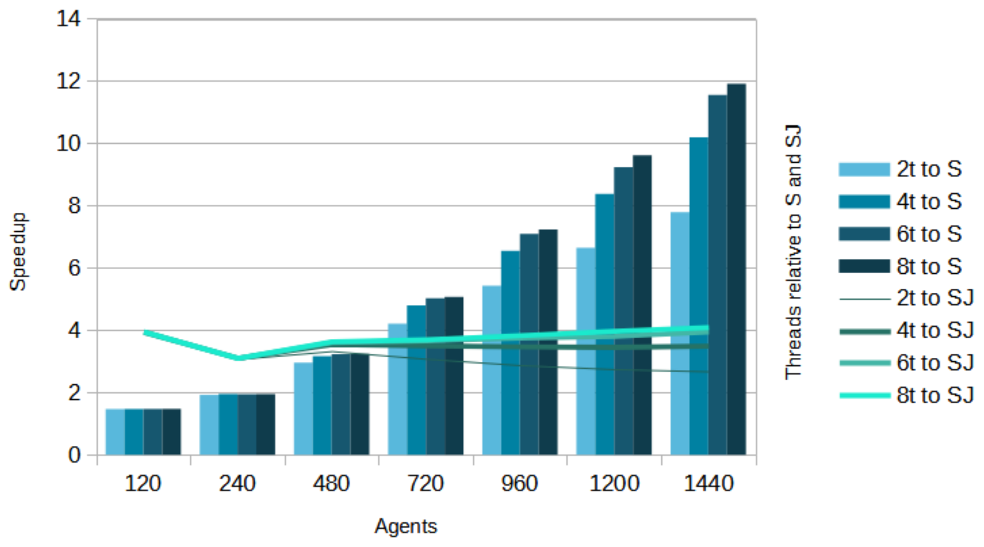


Figure 25 – Flocking 1440-agent Sequential execution Profiler

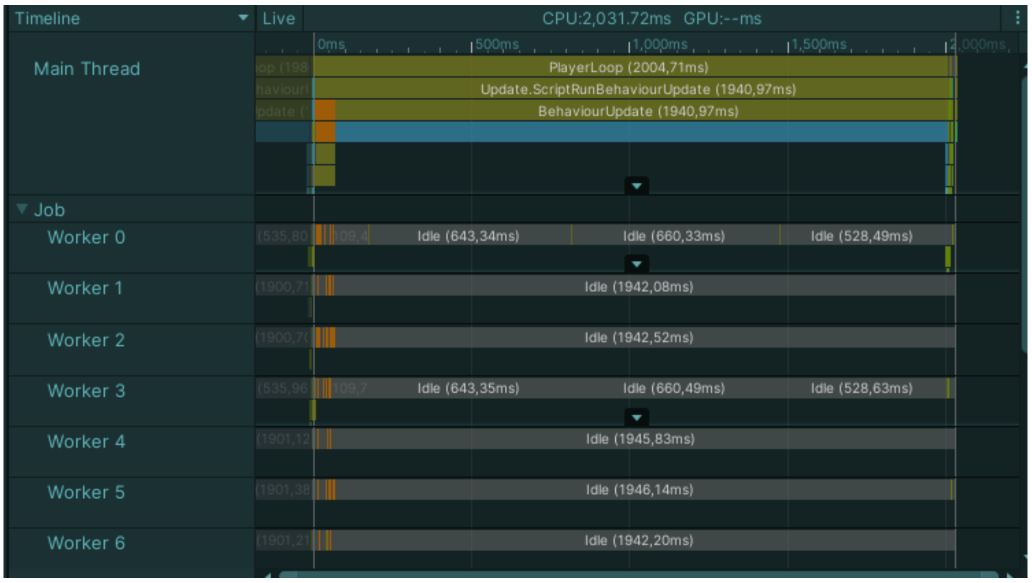
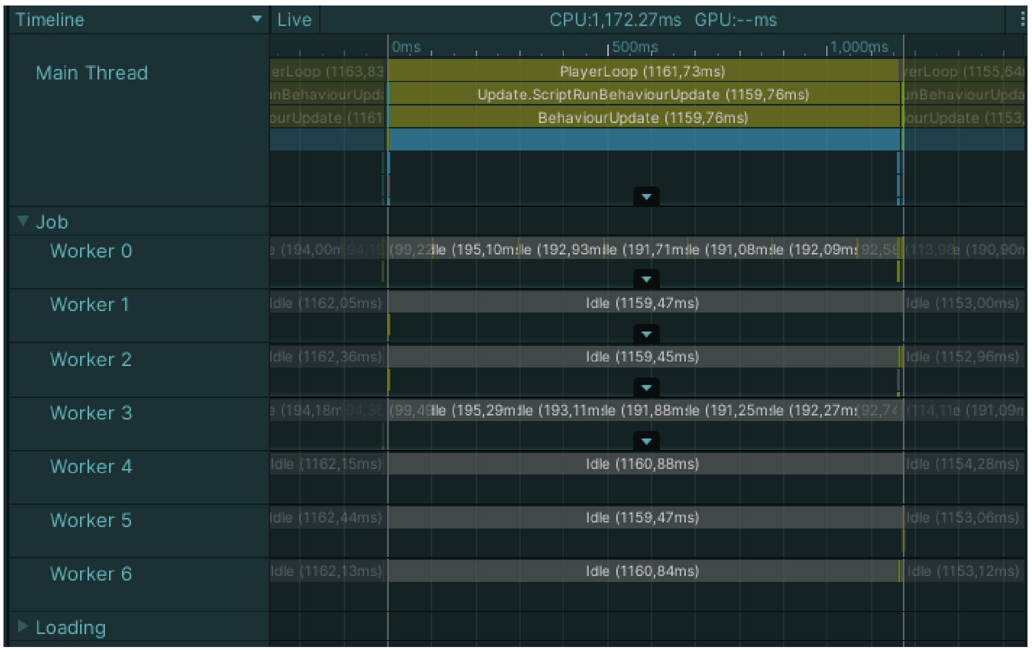


Figure 26 – Find Nearest 1440-agent Sequential execution Profiler

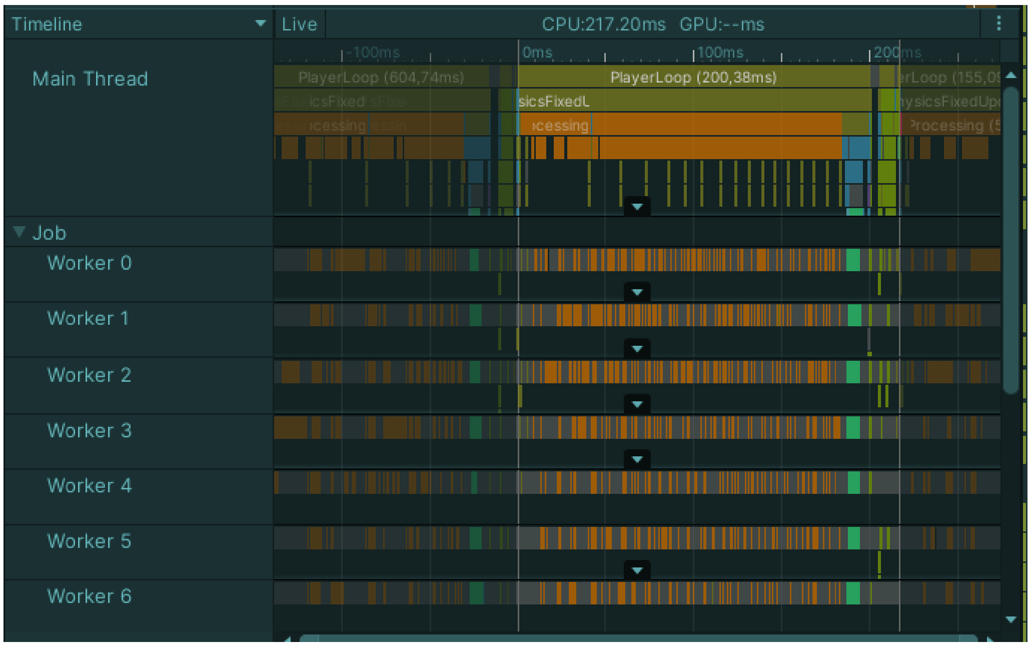


A closer inspection shows that, in both algorithms, the parallel job utilization across threads is consistently lower than the workload processed by the main thread in the sequential implementation. This explains why, as the number of agents increases, the speedup improves: a significant portion of the agents' processing remains sequential, with the main thread handling most of the frame computation. Parallel processing only handles a subset of the workload, using fewer resources compared to the main thread. This behavior accounts for the relatively narrow performance differences observed, as the sequential portion dominates the computation cost. Nevertheless, there is sufficient scalability since the main thread's job workload when compared to job workloads (represented by the green bars in the figures) increases at a slower rate, leaving room for speedup gains.

For the Flocking algorithm, however, physics calculations can dominate processing time, particularly when boids collide, demanding greater computational resources. In parallel execution, these calculations are distributed across threads, mitigating the impact on performance. Without parallelism, these computations overload the main thread, as observed in the Single-Threaded Job-enabled scenario (Figure 29), which records a frame execution time of 238.97 ms. Although this overwhelmed execution is faster than the sequential version without job-enabled processing, it highlights the limits of single-threaded traditional MonoBehavior approaches.

In contrast, the Find Nearest algorithm does not involve physics processing, which eliminates the risk of main-thread overload due to physics calculations. As shown in the Single-Threaded Job-enabled profile for Find Nearest (Figure 30), even in this configuration, the job execution fraction (green bar) remains significantly lower than the total frame execution time. This demonstrates that, even at the highest agent set size, there is a clear performance difference compared to the fully sequential implementations, with noticeable improvements due to the job-enabled setup and the stack change. The difference is not very big when comparing batches due to the batch variation rely on data oriented jobified workload, which represented by the green bars, can be seen in Figures 27 and 28

Figure 27 – Flocking 1440-agent 128-batch Overwhelmed Multithreaded Job-enabled execution Profiler



the parallelized workload is not bigger than the sequential fraction, differently to when speaking of agents that have a huge part of their prefab made of MonoBehavior script, this leads to batches only having impact over the parallel fraction making the sequential dictates the column trends on graphs.

Figure 28 – Find Nearest 1440-agent 128-batch Multithreaded Job-enabled execution Profiler

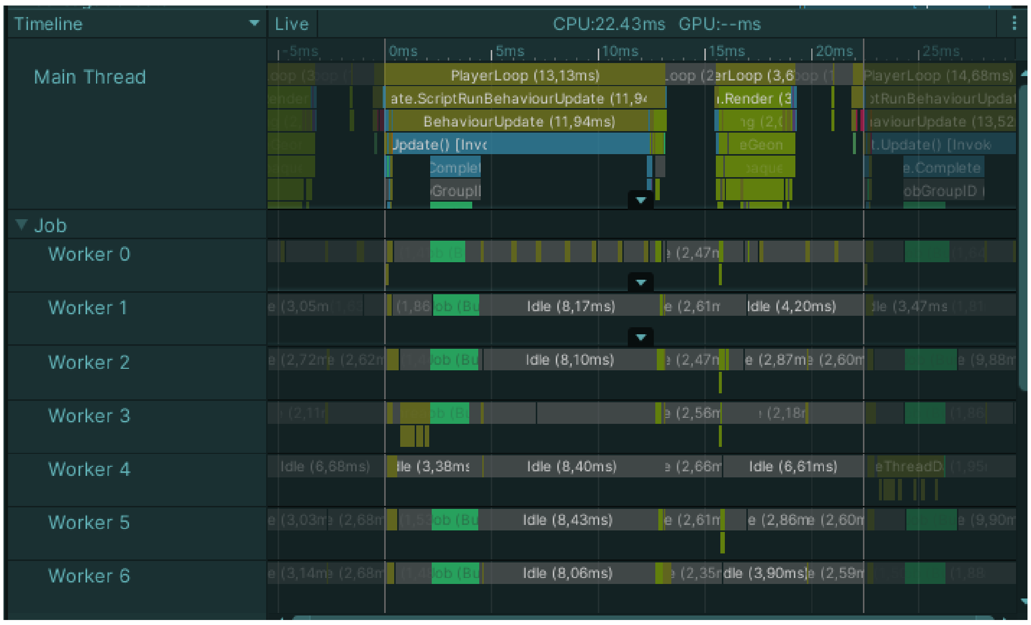


Figure 29 – Flocking 1440-agent Overwhelmed SingleThreadedJob execution Profiler

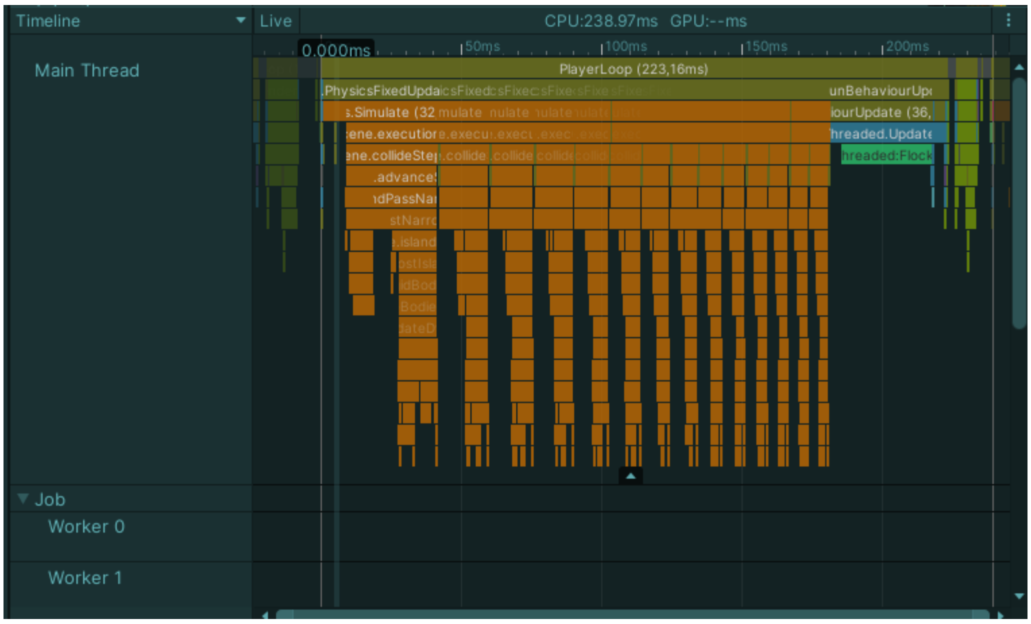
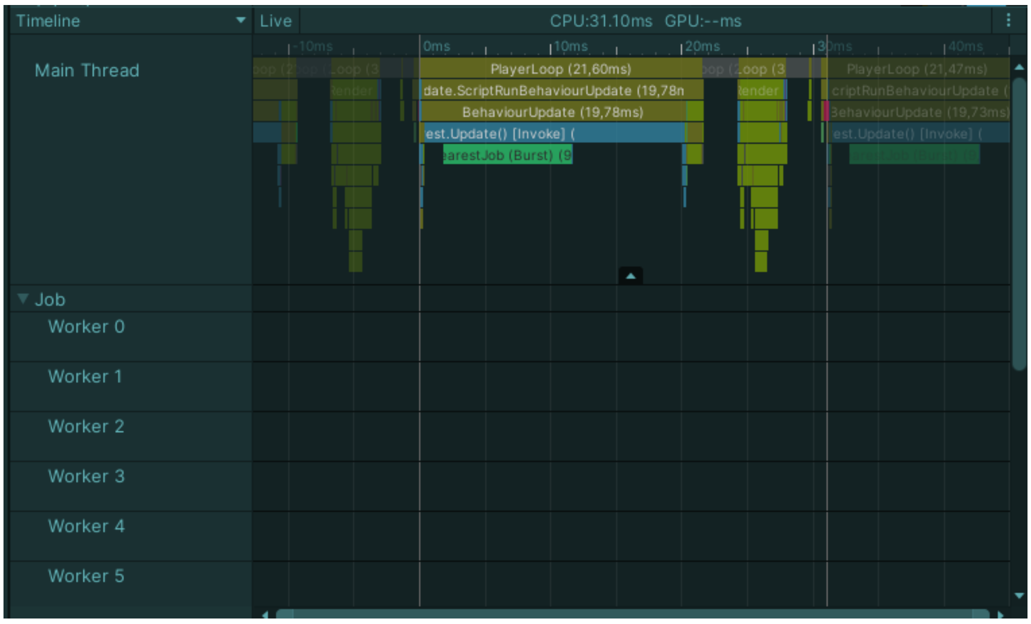


Figure 30 – Find Nearest 1440-agent SingleThreadedJob execution Profiler



6 CONCLUSION

The Unity Jobs System has proven to be a powerful tool for optimization. Using recommended settings, such as batch configurations of 16, 32, or 64, is versatile across different problem complexities and can still provide significant performance improvements, with potential speedups exceeding 15×. However, for high-end needs, it is advisable to conduct specific performance tests, as certain thread configurations may offer better performance for particular algorithms and workloads. For example, as shown in Figure 15, the 8-thread Flocking results indicate that the optimal batch configuration (4 batches, Burst-enabled) achieved a peak speedup of 9.66, while the recommended configurations of 16, 32, and 64 batches yielded speedups of 8.76, 8.77, and 8.37, respectively. The difference between the 4-batch configuration and the best speedup among the three recommended configurations is 0.89. This suggests that the average developer can leverage the significant advantages of Unity Jobs without needing to run extensive tests to find the best configuration for their project.

The same principle applies to the thread scalability of Unity Jobs' multithreading. Specific problems and workloads may require tailored configurations to optimize each algorithm's performance. For instance, in the Find Nearest test with 4 batches and 1440 agents, Burst-enabled, the 6-thread configuration outperformed the 8-thread configuration, achieving a speedup of 15.18 compared to 15.06—a difference of 0.12. This means that even if a developer uses the 8-thread configuration, they will still see an improvement, albeit with a slight loss of 0.12 in potential performance, which is still acceptable in most cases where 8 threads perform better. A similar scenario is observed with the single-threaded Job, where developers can achieve comparable performance improvements to multithreading without needing to delve deeply into the complexities of multithreading. As shown in Table 7, for 1440 agents, the single-threaded Job (SJ), Burst-enabled, averaged 141.08 milliseconds per frame, which is 40.74 milliseconds slower than the best case—an 8-thread configuration averaging 100.74 milliseconds per frame.

The tests of multithreaded job-enabled against the normal sequential no-job also demonstrated that using the Burst Compiler significantly improved performance in both algorithms. For the Flocking algorithm, the 8-thread, 4-batch configuration with Burst enabled achieved a speedup of 9.66, compared to 8.63 in the 8-thread, 32-batch configuration without Burst, resulting in a speedup improvement of 1.03. Similarly, in the Find Nearest algorithm, the 8-thread, 4-batch configuration with Burst enabled reached a speedup of 15.06, compared to 11.91 in the 8-thread, 32-batch configuration without Burst, leading to a notable speedup increase of 3.15. These results highlight the effectiveness of the Burst Compiler in optimizing performance.

Peak performance is crucial for high-performance gaming needs, as the requirements of large projects differ significantly from those of smaller ones. Achieving high-end performance on consoles or desktops is essential. This study also highlights the importance of conducting the necessary tests to determine the best implementation for a project's specific needs, demonstrating the computational value in processing when custom settings are tested against recommended ones. The experiment leads to the conclusion that the use of Burst Compiler is a must on every scenario. Moreover, the study showed that using single-threaded Jobs may not offer improvements over sequential, no-Job-based algorithms when working with small datasets.

6.1 Future Studies

For future research, it would be valuable to explore the integration of the Entity Component System (ECS) into the performance optimization pipeline, particularly in combination with Unity Jobs. ECS's data-oriented design allows for more efficient memory management and parallelism, which could result in significant performance improvements in scenarios involving high object counts or complex interactions. The synergy between ECS, Jobs, and the Burst Compiler could be thoroughly examined to uncover its full potential for optimizing interactive media and games.

Another promising area for investigation involves cross-platform performance comparisons. Evaluating how various CPUs, GPUs, and hybrid processors handle parallel workloads could provide deeper insights into platform-specific optimizations and limitations. This line of research would inform tailored development strategies and highlight the scalability of parallelism approaches across different hardware configurations.

Alternative parallelism implementations also merit further exploration. Task-based parallelism frameworks or other parallel processing paradigms may offer diverse perspectives on performance scalability, code maintainability, and developer ease of use. Comparing these approaches with Unity Jobs could provide valuable insights into their relative advantages and trade-offs.

Additionally, the performance potential of heterogeneous hardware architectures, such as systems combining CPUs with GPUs or AI accelerators, represents an intriguing avenue for future study. Research could focus on load-balancing techniques, data transfer overheads, and efficient utilization of diverse processing units. Understanding how to optimize workloads across such architectures may unlock new possibilities for performance in computationally intensive scenarios.

It would also be valuable to investigate the impact of combining multiple approaches, such as using ECS alongside heterogeneous architectures, Jobs System, and Burst Compiler. This comprehensive study could push the boundaries of what is achievable in performance optimization for interactive media and game development.

Moreover, while the current results provide an average execution time across 100 frames per test unit, analyzing collision events in greater detail could yield additional insights. Future studies could focus on quantifying the number of collisions occurring during execution and measuring their precise impact on processing time. By tracking the frequency and computational cost of these events, researchers could better understand how collisions contribute to performance bottlenecks.

Finally, conducting a memory usage analysis would add another layer of understanding to performance optimization. Monitoring memory usage, cache hits and misses, and their impact on execution time at a lower level could reveal critical inefficiencies. Such an analysis would help refine memory management strategies and further enhance the performance of interactive applications.

REFERENCES

ADVE, S.; GHARACHORLOO, K. Shared memory consistency models: a tutorial. *Computer*, v. 29, n. 12, p. 66–76, 1996. 23

BHAGAT, K. K.; LIOU, W.-K.; CHANG, C.-Y. A cost-effective interactive 3d virtual reality system applied to military live firing training. *Virtual Reality*, Springer, v. 20, p. 127–140, 2016. 14

BONDI, A. B. Characteristics of scalability and their impact on performance. In: *Proceedings of the 2nd International Workshop on Software and Performance*. New York, NY, USA: Association for Computing Machinery, 2000. (WOSP '00), p. 195–203. ISBN 158113195X. Disponível em: <https://doi.org/10.1145/350391.350432>. 25

BORUFKA, R. Performance testing suite for unity dots. Univerzita Karlova, Matematicko-fyzikální fakulta, 2020. 27, 28

EYNIYEV, R. Optimizing parallelism in unity with job system tool. *Azerbaijan Journal of High Performance Computing*, Azerbaijan Journal of High Performance Computing, v. 5, n. 2, p. 183–192, 2022. 27, 28

FLYNN, M. Very high-speed computing systems. *Proceedings of the IEEE*, v. 54, n. 12, p. 1901–1909, 1966. 20

GABAJOVÁ, G. et al. Designing virtual workplace using unity 3d game engine. *Acta Tecnología*, v. 7, n. 1, p. 35–39, 2021. 14

GANG, S.-M. et al. A study on the construction of the unity 3d engine based on the webgis system for the hydrological and water hazard information display. *Procedia engineering*, Elsevier, v. 154, p. 138–145, 2016. 14

HAGER, G. *Introduction to High Performance Computing for Scientists and Engineers*. [S.l.]: CRC Press, 2010. 20, 23, 25, 26, 40

HENNESSY, J. L.; PATTERSON, D. A. *Computer architecture: a quantitative approach*. [S.l.]: Elsevier, 2011. 24

HILL, M. D.; MARTY, M. R. Amdahl's law in the multicore era. *Computer*, IEEE, v. 41, n. 7, p. 33–38, 2008. 25, 40

HUSSAIN, A. et al. Unity game development engine: A technical survey. *Univ. Sindh J. Inf. Commun. Technol*, v. 4, n. 2, p. 73–81, 2020. 14

KAUR, P. et al. A survey on simulators for testing self-driving cars. In: IEEE. *2021 Fourth International Conference on Connected and Autonomous Driving (MetroCAD)*. [S.l.], 2021. p. 62–70. 14

KOULAXIDIS, G.; XINOGALOS, S. Improving mobile game performance with basic optimization techniques in unity. *Modelling*, MDPI, v. 3, n. 2, p. 201–223, 2022. 14

MICHIELS, B. et al. Weak scalability analysis of the distributed-memory parallel mlfma. *IEEE Transactions on Antennas and Propagation*, IEEE, v. 61, n. 11, p. 5567–5574, 2013. 26

NÄYKKI, A. Unity dots in production: Dots pathfinding implementation in vr & ar. 2021. 27, 29

PACHECO, P. *An introduction to parallel programming.* [S.l.]: Elsevier, 2011. 20, 23, 25, 26, 40

PASTERNAK, M. et al. Simgen: A tool for generating simulations and visualizations of embedded systems on the unity game engine. In: *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings.* [S.l.: s.n.], 2018. p. 42–46. 14

REYNOLDS, C. W. Flocks, herds and schools: A distributed behavioral model. In: *Proceedings of the 14th annual conference on Computer graphics and interactive techniques.* [S.l.: s.n.], 1987. p. 25–34. 33

SHI, Y. et al. A multiuser shared virtual environment for facility management. *Procedia Engineering*, Elsevier, v. 145, p. 120–127, 2016. 14

SINGH, N. P.; SHARMA, B.; SHARMA, A. Performance analysis and optimization techniques in unity 3d. In: IEEE. *2022 3rd International Conference on Smart Electronics and Communication (ICOSEC).* [S.l.], 2022. p. 245–252. 14

WANG, J. et al. An improved fast flocking algorithm with obstacle avoidance for multiagent dynamic systems. *Journal of Applied Mathematics*, Wiley Online Library, v. 2014, n. 1, p. 659805, 2014. 33

WANG, X.; LIN, W.; ZHAO, J. Dots in unity3d game. In: IEEE. *2020 International Conference on Virtual Reality and Visualization (ICVRV).* [S.l.], 2020. p. 385–388. 27

YANG, C.-W. et al. Unity 3d production and environmental perception vehicle simulation platform. In: IEEE. *2016 International Conference on Advanced Materials for Science and Engineering (ICAMSE).* [S.l.], 2016. p. 452–455. 14

YANG, K.; JIE, J. The designing of training simulation system based on unity 3d. In: IEEE. *2011 Fourth international conference on intelligent computation technology and automation.* [S.l.], 2011. v. 1, p. 976–978. 14