Ministry of Higher Education and Scientific Research
Hassiba Benbouali University of Chlef
Faculty of Exact Sciences and Informatics
Department of Informatics

THESIS

Submitted in fulfilment of the requirements for the degree of

DOCTORAT LMD

Field: Informatics
Speciality: Informatics

By:

MOHAMMED LOTFI BENDIAF

On September 30th, 2025

Subject:

DYNAMIC ADAPTATION APPROACH FOR DISTRIBUTED SYSTEM BEHAVIOR

The jury composed of:

Rachid BECHAR	MCA	University of Chlef	Jury President
Ahmed LOUAZANI	MCA	University of Blida	Examiner
Mourad LOUKAM	MCA	University of Chlef	Examiner
Mohammed Amin TAHRAOUI	MCA	University of Chlef	Director
Ahmed HARBOUCHE	MCA	University of Chlef	Co-Director

Declaration of Authorship

I, Mohammed Lotfi BENDIAF, declare that this thesis titled, "**Dynamic adaptation approach for distributed system behavior**" and the work presented in it are my own. I confirm that:

- This work has not previously been submitted in substance for any degree and is not concurrently being submitted for any other degree.
- This thesis represents my own research efforts, except where explicitly stated otherwise. References to other sources are duly acknowledged through footnotes, and a full bibliography is provided at the end of this document. Specifically, the thesis includes content from the following publications:
    1. L. Bendiaf, A. Harbouche, MA. Tahraoui, 2023. *A Novel independent Tasks Scheduling Method Using the Dynamic Programming in Multiprocessor Systems*, **NATEC**. National Conference On Artificial Intelligence, Smart Technologies And Communications AISTC'23
    2. Bendiaf, L.M., Harbouche, A., Tahraoui, A.M., and Lebbah, F.Z., 2024. *An Innovative Task Scheduling Method Using the Knapsack Algorithm in Heterogeneous Computing Systems*, *Informatica*, **48**(16). DOI: https://doi.org/10.31449/inf.v48i16.5765
    3. Bendiaf L, Harbouche A, Tahraoui MA. Dynamic Adaptation for Independent Task Scheduling Using Dynamic Programming in Multiprocessor Systems. Revue Nature et Technologie. 2025 Mar 30;17(1):09-16.
- I consent to this thesis, if accepted, being made available for photocopying and inter-library loan, and for the title and summary to be accessible to external organizations.
- All ethical procedures established by the university have been followed, and ethical approval has been obtained where required.

Signed: Mohammed Lotfi Bendiaf

Date: October 5, 2025

Abstract

As distributed systems (DS) evolve, managing dynamic workloads and resource availability becomes essential for maintaining optimal performance. This thesis presents a novel Dynamic Adaptation (DA) framework designed to enhance the efficiency and responsiveness of DS, particularly in heterogeneous computing environments.

The central problem addressed in this work concerns how the performance of distributed systems can be effectively enhanced through dynamic adaptation, particularly at the level of a single-node multiprocessor system considered as a fine-grained unit within a distributed architecture. Furthermore, what key strategies and algorithmic approaches are essential to optimize task allocation, ensure load balancing, and minimize execution time in such heterogeneous environments.

A key contribution of this work is the introduction of two innovative algorithms for dynamic task scheduling. The first, DyTAg (Dynamic Task Allocation using Dynamic Programming), focuses on optimizing task allocation in heterogeneous multiprocessor systems with independent tasks. It leverages dynamic programming to minimize makespan and balance workloads effectively, laying the groundwork for more advanced scheduling approaches.

Building on DyTAg, the thesis introduces the Knapsack-based Algorithm Co-Scheduling Task Allocation (KaCoSTA), which integrates dynamic programming with knapsack optimization techniques to address task precedence and resource constraints. KaCoSTA dynamically adapts to system states, task priorities, and processing capabilities to maximize resource utilization (RU) and minimize makespan. Extensive experiments demonstrate its superiority over existing methods, such as Min-Min, Max-Min, and HEFT (Heterogeneous Earliest Finish Time), in both static and dynamic scenarios.

Results reveal significant advancements in system adaptability, load balancing, and overall efficiency under fluctuating workloads. By combining foundational research with practical innovation, this work provides a comprehensive solution for optimizing DS behavior in real-world applications.

Keywords: Distributed systems, dynamic adaptation, optimization, multiprocessor systems, scheduling, knapsack.

الملخص

مع تطوّر الأنظمة الموزعة (DS) ، أصبحت إدارة مهام المعالجة الديناميكية وتوفّر الموارد وحالات النظام أمرًا ضروريًا للحفاظ على الأداء الأمثل. تقدّم هذه الأطروحة إطارًا مبتكرًا للتكيّف الديناميكي (DA) يهدف إلى تحسين كفاءة واستجابة الأنظمة الموزعة، لا سيّما في بيئات الحوسبة غير المتجانسة.

تكمن الإشكالية في هذا العمل في كيفية تعزيز أداء الأنظمة الموزعة بفعالية من خلال التكيّف الديناميكي، خصوصًا على مستوى نظام متعدد المعالجات، الذي يُعد وحدة دقيقة ضمن بنية نظام موزع أشمل. فما هي الاستراتيجيات الرئيسية والخوارزميات الضرورية لتحسين تخصيص المهام، وضمان توازن الأحمال، وتقليل زمن التنفيذ في مثل هذه البيئات غير المتجانسة؟

تتمثّل إحدى المساهمات الأساسية لهذا العمل في اقتراح خوارزميتين مبتكرتين لجدولة المهام الديناميكية. الأولى، DyTAg ، تركّز على تحسين تخصيص المهام في أنظمة متعددة المعالجات غير المتجانسة التي تحتوي على مهام مستقلة. وتعتمد هذه الخوارزمية على البرمجة الديناميكية (Dynamic Programming) لتقليل زمن الإنجاز وتحقيق توازن فعّال في أعباء العمل، مما يمهّد الطريق لتقنيات جدولة أكثر تقدمًا.

انطلاقًا من DyTAg ، تقدّم الأطروحة خوارزمية KaCoSTA ، التي تدمج بين البرمجة الديناميكية وتقنيات تحسين الحقيبة (Knapsack Optimization) لمعالجة أولويات المهام وقيود الموارد. تتكيّف KaCoSTA ديناميكيًا مع حالات النظام، وأولويات المهام، وقدرات المعالجة بهدف تعظيم استغلال الموارد (RU) وتقليل زمن الإنجاز. وقد أظهرت التجارب المكثفة تفوّقها على الأساليب التقليدية مثل Min-Min و Max-Min و HEFT (Heterogeneous Earliest Finish Time) ، سواء في السيناريوهات الثابتة أو الديناميكية.

وتكشف النتائج عن تطورات ملحوظة في قدرة النظام على التكيّف، وتوازن الأحمال، والكفاءة العامة في ظل تغيّر أعباء العمل. ومن خلال الجمع بين البحث النظري والابتكار التطبيقي، يقدّم هذا العمل حلًا متكاملًا لتحسين سلوك الأنظمة الموزعة في التطبيقات الواقعية.

الكلمات المفتاحية : Distributed Systems ، Dynamic Adaptation ، Optimization ، Multiprocessor Systems ، Scheduling ، Knapsack .

Résumé

À mesure que les systèmes distribués évoluent, la gestion des charges de travail dynamiques et de la disponibilité des ressources devient essentielle pour maintenir des performances optimales. Cette thèse présente un nouveau cadre d'adaptation dynamique conçu pour améliorer l'efficacité des systèmes distribués, en particulier dans les environnements de calcul hétérogènes.

La problématique abordée dans ce travail est la suivante : comment peut-on améliorer de manière efficace les performances des systèmes distribués grâce à l'adaptation dynamique, en particulier au niveau d'un système multiprocesseur ? Quelles stratégies sont nécessaires pour optimiser l'allocation des tâches et minimiser le temps d'exécution dans de tels environnements hétérogènes ?

Une contribution majeure de ce travail est l'introduction de deux algorithmes innovants pour l'ordonnancement dynamique des tâches. Le premier, DyTAg, vise à optimiser l'allocation des tâches indépendantes dans les systèmes multiprocesseurs hétérogènes. Il s'appuie sur la programmation dynamique pour minimiser le temps total d'exécution, constituant ainsi une base solide pour des approches d'ordonnancement plus avancées.

En s'appuyant sur DyTAg, la thèse introduit KaCoSTA, qui intègre la programmation dynamique avec des techniques d'optimi-sation du problème du sac à dos pour gérer les contraintes de précédence des tâches et les limitations des ressources. KaCoSTA s'adapte dynamiquement aux états du système, aux priorités des tâches et aux capacités de traitement pour maximiser l'utilisation des ressources (RU) et minimiser le makespan. Des expériences approfondies démontrent sa supériorité par rapport aux méthodes existantes, aussi bien dans des scénarios statiques que dynamiques.

En combinant des fondements théoriques solides avec une innovation pratique, ce travail propose une solution complète pour optimiser le comportement des systèmes distribués dans des applications réelles.

Mots-clés : systèmes distribués, adaptation dynamique, optimisation, systèmes multiprocesseurs, ordonnancement, knapsack.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisors, Harbouche and Tahraoui, for their invaluable guidance, support, and encouragement throughout the duration of my research. Your insightful advice and expertise have been instrumental in shaping the direction of my work, and I am deeply thankful for your patience and dedication.

I am also grateful to the faculty of the Department of Informatics at Hassiba Benbouali University, who have provided a stimulating academic environment and have offered constructive feedback and support during the course of my PhD journey. Special thanks go to the jury members: Rachid BECHAR, Ahmed LOUAZANI, Mourad LOUKAM, Mohammed, Amin TAHRAOUI, Ahmed HARBOUCHE and Fatima Zohra LEBBAH for their critical input and inspiring discussions.

I would like to acknowledge the research teams at Department of Informatics at Hassiba Benbouali University, particularly the LME Laboratory, for the collaborative efforts and for providing access to resources and tools essential for this research. Your contributions have enriched my work and inspired me to explore new ideas.

A sincere thank you to my colleagues and friends who have stood by me through this journey. Your moral support, constructive criticism, and friendship have been essential in keeping me motivated during difficult times.

Finally, I owe everything to my family, especially my parents and my spouse Fatima Zahra, for their unwavering love, patience, and support. Your belief in me has been a source of strength, and without your understanding and sacrifices, this dissertation would not have been possible.

To everyone who has contributed in any way, no matter how small, I extend my heartfelt thanks.

Contents

Glossary of Terms

This appendix provides definitions of key terms and concepts used throughout the thesis.

- **Scheduling**: The process of assigning tasks to computational resources such as processors or nodes in a way that optimizes one or more performance metrics, including makespan and resource utilization.
- **Adaptive Scheduling**: A scheduling approach that dynamically adjusts task assignments based on system changes such as workload variations or resource availability.
- **Approximation Algorithm**: A technique used for solving NP-hard problems, providing near-optimal solutions with provable performance guarantees.
- **Branch-and-Bound (B&B)**: An exact optimization method used to systematically explore solution spaces while pruning suboptimal solutions.
- **Cloud Computing**: A distributed computing paradigm where resources such as storage, processing power, and applications are provided over the internet on demand.
- **Combinatorial Optimization**: The process of finding an optimal object from a finite set of objects, often applied in task scheduling and resource allocation.
- **Concurrency**: The ability of a computing system to execute multiple tasks simultaneously, either through multi-threading or distributed execution.
- **Distributed System (DS)**: A collection of independent computing entities that communicate and coordinate their actions through a network to perform a common task.
- **Dynamic Adaptation**: The process of adjusting system parameters or configurations in real-time to respond to changes in the environment or workload.
- **Dynamic Programming (DP)**: A method for solving complex problems by breaking them into simpler subproblems and solving them recursively.

- **DyTAg (Dynamic Task Allocation Algorithm)**: A scheduling algorithm that dynamically assigns independent tasks to heterogeneous processors using dynamic programming principles.
- **Energy-Aware Scheduling**: A scheduling strategy that considers power consumption as a constraint to optimize performance while reducing energy usage.
- **Heterogeneous Computing System (HCS)**: A computing environment consisting of multiple processors with different computational capabilities, architectures, or functionalities.
- **Heterogeneous Multiprocessor System (HMS)**: A single node with diverse processors (e.g., CPU and GPU) working cooperatively, representing the fine-grained unit within a distributed system.
- **Heuristic Algorithm**: A problem-solving approach that finds a good-enough solution efficiently, often without guaranteeing optimality.
- **KaCoSTA (Knapsack-based Co-Scheduling Algorithm for Task Allocation)**: A task scheduling approach that considers multiple processors simultaneously, optimizing resource utilization using knapsack problem techniques.
- **Knapsack Problem**: A combinatorial optimization problem where items with given weights and values must be selected to fit within a limited-capacity knapsack while maximizing total value.
- **Latency**: The delay experienced in task execution or data transmission within a distributed system.
- **Load Balancing**: A technique used in distributed computing to evenly distribute tasks among available resources to prevent bottlenecks and maximize efficiency.
- **Machine Learning in Scheduling**: The use of artificial intelligence techniques to predict workload patterns and optimize scheduling decisions dynamically.
- **Makespan**: The total time required to complete all scheduled tasks in a computing environment.
- **Metaheuristics**: High-level strategies such as genetic algorithms and simulated annealing used to find near-optimal solutions to complex optimization problems.
- **Multiprocessor Scheduling**: The process of assigning computational tasks to multiple processors to optimize performance, energy consumption, or other constraints.
- **Multiple Knapsack Problem (MKP)**: A generalization of the knapsack problem where multiple knapsacks must be filled optimally with given constraints.

- **Parallel Processing**: The simultaneous execution of multiple computational tasks across different processors to improve performance.
- **Quality of Service (QoS)**: A measure of system performance, including factors such as response time, throughput, and availability.
- **Real-Time Scheduling**: A scheduling strategy that ensures tasks are completed within strict time constraints, critical for applications in robotics, healthcare, and telecommunications.
- **Resource Utilization (RU)**: The efficiency with which available computing resources are used to execute tasks.
- **Scalability**: The ability of a system to handle increasing workloads or expand computational resources while maintaining performance.
- **Scheduling Algorithm**: A method used to allocate computational tasks to processors or resources while optimizing specific performance criteria.
- **Simulated Annealing**: A probabilistic optimization technique inspired by the annealing process in metallurgy, used in scheduling problems.
- **Swarm Intelligence**: A category of bio-inspired algorithms (e.g., artificial bee colony, particle swarm optimization) used in scheduling and optimization problems.
- **Task Scheduling**: The process of assigning tasks to computational resources in a way that optimizes performance metrics such as execution time or energy consumption.
- **Task Dependency Graph**: A representation of task relationships, where nodes represent tasks and edges define execution dependencies.
- **Virtual Machines (VMs)**: Software-based computing environments that simulate physical hardware and are used in cloud computing to optimize resource allocation.
- **Workload Distribution**: The process of distributing computational tasks across available resources to ensure efficient execution.
- **Softure**: combination of SOFTware and futURE.

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The rapid evolution of computing has significantly transformed the design and operation of modern systems. From the early days of large and isolated mainframes to today's interconnected and high-performance computing environments, advancements in hardware and networking have greatly enhanced computational capabilities and efficiency. From 1945 to the mid-1980s, computers were expensive, standalone systems lacking the ability to communicate with each other. This paradigm shifted with two major technological breakthroughs: the development of increasingly powerful microprocessors evolving from basic 8-bit architectures to today's advanced 64-bit multi-core CPUs and the advent of high-speed computer networks. These innovations drastically enhanced computing performance and enabled efficient interconnection between machines across both local and wide-area networks. Concurrently, advances in miniaturization gave rise to compact yet powerful computing devices, most notably smartphones, which integrate sensors, substantial memory, and multi-core processors within portable, network-enabled platforms. Similarly, plug computers and compact devices started offering performance levels approaching those of traditional desktop systems. These developments have extended the reach of distributed computing to the network edge, empowering ubiquitous, high-performance, and interconnected computational environments.

These technological developments have laid the foundation technological basis of modern Distributed Systems (DS), where multiple autonomous computational entities collaborate to execute tasks efficiently. Such systems are inherently dynamic, characterized by fluctuating workloads, variable resource availability, and constantly evolving computational demands. Key milestones include the development of multi-core processors, which support extensive parallelism, and the rise of high-speed networks that enable seamless communication among distributed computing nodes.

Performance is a critical measure of the success and efficiency of computer systems

and algorithms. In computer science, performance refers to how effectively a system or application utilizes resources to achieve its goals, often evaluated through metrics such as speed, resource utilization, scalability, and responsiveness. Performance optimization is essential in all computing domains as it directly impacts user experience, system reliability, and operational costs.

New technologies, such as distributed computing, multiprocessor systems and multicore processors, have emerged to enhance the efficiency of Information Technology (IT) systems in terms of performance, particularly time and quality of service. However, despite hardware development, the conception of modern materials remains insufficient without efficient software techniques.

Despite these advances, efficiently managing DS remains a complex challenge, particularly in environments where workloads and resources change unpredictably. Traditional scheduling and resource allocation techniques, often based on static heuristics, struggle to adapt to real-time variations. This limitation has fueled the need for Dynamic Adaptation (DA) strategies that can adjust task assignments, resource utilization, and execution priorities in response to changing conditions.

DS serve as the backbone of modern computing paradigms, including cloud computing, edge computing, blockchain networks, and high-performance computing clusters. These systems must balance task execution, resource distribution, and system responsiveness while adapting to changing conditions. However, several challenges remain unresolved: distributed environments comprise diverse hardware architectures, requiring adaptable scheduling strategies; task arrival patterns are often unpredictable, making static scheduling approaches ineffective; efficient allocation of computational power, memory, and bandwidth is crucial to maintaining performance; and systems must handle node failures and disruptions without significantly degrading performance.

A major challenge in DS is the optimization of critical performance indicators, including makespan, resource utilization, latency, and throughput. Conventional scheduling algorithms frequently fall short in handling the inherent complexities of heterogeneous environments and dynamic conditions such as varying workloads or node failures. Consequently, they often fail to sustain efficiency and scalability under real-world operational demands.

Dynamic Adaptation in Distributed systems involves the ability to reallocate tasks and adjust system configurations, workload, resource availability, or system state in real time, with the goal of improving overall DS performance. Unlike static systems that follow predetermined configurations or schedules, dynamically adaptive systems modify their behavior, structure, or resource allocation based on ongoing conditions to ensure optimal performance despite external fluctuations. However, successfully addressing these chal-

lenges requires advanced algorithms capable of balancing computational loads, respecting task dependencies, and optimizing resource usage in real time.

This adaptability is enabled by several key mechanisms. Load balancing distributes tasks across multiple nodes to avoid overloading any single resource and to improve overall system efficiency. Fault tolerance ensures uninterrupted service by adjusting operations or re-routing tasks in the event of node failures or network disruptions. Resource scaling dynamically allocates or deallocates computing resources such as virtual machines, processors, or storage—based on real-time workload demands, helping optimize performance and reduce resource wastage. Task scheduling continuously reassigns tasks and adjusts execution priorities according to changing workloads, task dependencies, and processor availability, aiming to minimize execution time and maximize resource utilization. Together, these mechanisms significantly enhance the efficiency, resilience, and scalability of distributed systems, enabling them to sustain high performance even in dynamic and unpredictable computing environments.

This research addresses these challenges by developing innovative optimization techniques that incorporate DA to enhance DS performance. The objective is to design methods that can efficiently manage heterogeneous resources, handle dynamic workloads, and minimize execution time while maximizing resource utilization. By tackling these issues, this work contributes to advancing the efficiency, scalability, and reliability of DS in diverse applications, from heterogeneous computing to real-time data processing.

The objectives of this thesis are to conduct an in-depth study of DS, focusing on the inherent challenges and complexities that arise in their operation. DS, with their geographically dispersed and heterogeneous components, present specific problems such as task scheduling, resource allocation, scalability, and fault tolerance. This work specifically addresses these issues in the context of heterogeneous computing environments, where DA is crucial for maintaining efficiency and responsiveness under fluctuating workloads and changing system states.

The thesis investigates the key performance metrics critical to the optimization of DS, including makespan minimization, resource utilization, latency, throughput, and system scalability. By exploring these metrics, we aim to identify bottlenecks and inefficiencies that can be addressed through novel scheduling and resource management strategies.

In the context of DS architecture, each node often encapsulates a multiprocessor system that must operate efficiently as an autonomous unit while contributing to the overall system performance. Treating such a system as a fine-grained element of the distributed network, this research aims to optimize scheduling and resource allocation at the node level. By improving task coordination within a single heterogeneous multiprocessor system, we enhance the foundational performance upon which broader distributed scheduling

strategies can be built.

A significant focus of this research is on multiprocessor systems, where the diversity in processor capabilities and task requirements adds another layer of complexity to scheduling problems. The study delves into DA mechanisms that allow real-time adjustments to task assignments and resource configurations, ensuring that systems remain efficient and responsive even in dynamic and unpredictable environments.

Traditional scheduling algorithms such as Min-Min, Max-Min, and HEFT (Heterogeneous Earliest Finish Time) have proven effective in certain scenarios but lack real-time adaptability. Their reliance on predefined task priorities and static workload assumptions makes them unsuitable for highly dynamic environments. Existing methods fail to dynamically adjust task assignments, leading to load imbalances, resource under utilization, and increased execution times. This research aims to bridge this gap by introducing adaptive scheduling techniques that can dynamically reallocate tasks, optimize resource distribution, and enhance overall system efficiency in real-time.

To address these challenges, this thesis proposes two novel scheduling algorithms designed for DA for scheduling in heterogeneous distributed environments. The first, Dynamic Task Allocation using Dynamic Programming (DyTAg), optimizes task allocation for independent tasks in multiprocessor systems, minimizing makespan while improving workload balance. The second, Knapsack-based Co-Scheduling Task Allocation (KaCoSTA), integrates knapsack optimization with dynamic programming to effectively handle task precedence constraints, system state fluctuations, and priority-based execution.

These approaches outperform conventional algorithms by enhancing scalability, ensuring efficient task execution across different workloads, improving fault tolerance by adapting to resource failures without significant performance degradation, maximizing resource utilization through dynamic allocation, and reducing execution time by optimizing task sequencing and dependency handling. Comprehensive experimental evaluations compare DyTAg and KaCoSTA against state-of-the-art scheduling methods, demonstrating their superiority in dynamic workload management and heterogeneous task execution.

Through this comprehensive study, the thesis seeks to advance the field of DS by addressing their core challenges and providing innovative, adaptable, and efficient solutions for task scheduling and resource management. These contributions aim to enhance the performance and scalability of DS, ensuring their applicability across diverse real-world scenarios, from cloud and edge computing to scientific simulations and large-scale data processing.

The remaining document is structured as follows.

- Chapter 2 provides an overview of Distributed Systems (DS) architecture, including its

core characteristics, structural strategies, and intended purposes. It also discusses the key challenges associated with distributed environments, such as heterogeneity, fault tolerance, scalability, and coordination, while highlighting the critical role DS play in modern computing infrastructures.

- Chapter 3 introduces the concept of DA, emphasizing its relevance in handling fluctuating workloads and evolving system conditions.
- Chapter 4 presents the DyTAg algorithm, detailing its design rationale, mathematical modelling, and implementation techniques.
- Chapter 5 introduces the KaCoSTA algorithm, which combines knapsack-based optimization with dynamic programming to enable co-scheduling across multiple processors, and evaluates its performance through experimental comparisons with established algorithms.
- Chapter 6 concludes the thesis by summarizing the key results and highlighting the main contributions.
- Chapter 7 outlines perspectives and future research directions aimed at further improving adaptability, scalability, and performance in DS.

By addressing the fundamental limitations of existing scheduling techniques, this thesis contributes to the advancement of DA strategies in distributed computing, enhancing the efficiency, scalability, and reliability of modern computing infrastructures.

Chapter 2

Fundamental Principles of Distributed System

The evolution of computer systems has been both rapid and transformative, fundamentally changing how computational systems are designed, deployed, and interconnected. As the demand for scalability, reliability, and responsiveness has grown, traditional centralized architectures have become insufficient to meet the complex requirements of modern applications. This shift has led to the emergence of Distributed Systems (DS), an architectural paradigm designed to address the limitations of centralized systems by enabling scalable, fault-tolerant, and efficient execution of computational tasks across multiple interconnected nodes.

First, grid computing emerged in the late 1990s as a heterogeneous collaborative distributed system, evolved from homogeneous distributed computing platforms. Grids are shared systems that encompass potentially any computing device connected to a network, from workstations to clusters. Computing grids are infrastructures that enable resource sharing by establishing usage policies and security rules, forming what are known as Virtual Organizations (VOs) [5].

With improvements in computer networks, the connection among computing nodes in clusters became faster. At the same time, new applications demanded more and more bandwidth, storing and exchanging massive volumes of data. Multimedia and e-Science are examples of applications that handle large data sets nowadays, putting in evidence the importance of communications to improve performance and support Quality of Service (QoS) offerings in distributed systems.

Cloud computing, on the other hand, offers computing resources often virtualized as services to users, abstracting away technical details related to resource management [6]. As such, clusters and grids can be components of data centers within a cloud infrastruc-

ture. This convergence introduces new optimization goals and variables, particularly in the context of green computing [7] and utility computing [8].

This chapter examines the fundamental principles of distributed systems, tracing their evolution and exploring their defining characteristics, architectural strategies, and the key challenges they address in contemporary computing environments.

2.1 Definition of Distributed Systems

Among these aforementioned technologies, Distributed Systems (DS) (see Definitions 1, 2) have emerged as networks of interconnected computers that can range from just a few nodes to millions (See Figure 2.1). These systems are inherently dynamic, with machines frequently joining or leaving the network, and they operate over wired, wireless, or hybrid communication infrastructures [9]. Their topology and performance continuously adapt to reflect the evolving nature of modern computing environments.

Definition 1. *A DS is a collection of autonomous computing elements that appears to its users as a single coherent system. [10].*

Definition 2. *A DS is a collection of autonomous entities that cooperate to solve complex computational problems. These systems are often geographically distributed, lack a shared global clock, and are typically used for tasks that would be infeasible on a single computing unit [11–13].*

These multiple interconnected nodes that often spread across geographical locations, work together to achieve a common computational goal. These systems rely on decentralized control and coordination to manage resources efficiently. Heterogeneity is inherent in DS, where nodes vary in processing power, memory, and communication capabilities. Key challenges include:

- Balancing workloads across nodes to prevent bottlenecks.
- Minimizing communication overhead between nodes.
- Adapting dynamically to changes in workload or node availability.

Machines that are part of a distributed system may include computers, physical servers, virtual machines, containers, or any other nodes capable of connecting to the network, maintaining local memory, and communicating by exchanging messages.

Distributed systems typically operate in one of two ways:

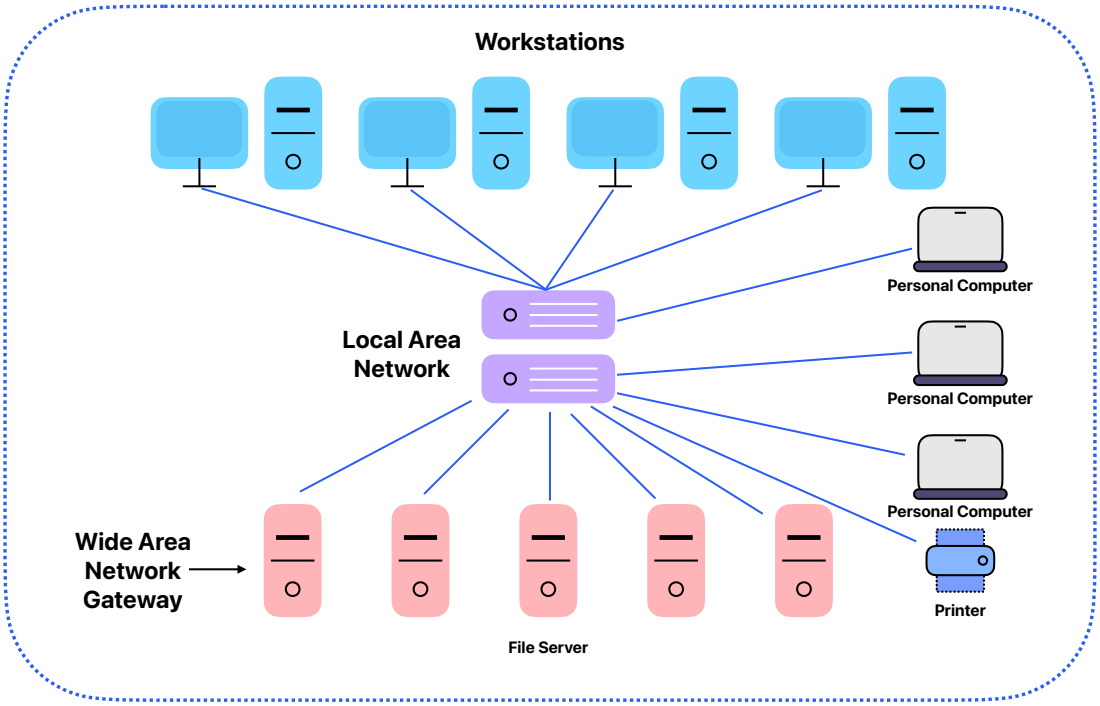

Figure 2.1: DS Architecture

- Multiple machines collaborate to achieve a common objective, with the end user perceiving the output as the result of a single cohesive system.
- Each machine serves its own end user, while the distributed system enables the sharing of communication resources or services among them.

Distributed computing is a specialized form of DS designed for high-performance computing. It can be broadly classified into two categories:

Cluster Computing: A tightly coupled system where nodes are located in close proximity and interconnected via high-speed networks. Clusters are commonly used for parallel processing and high-performance computing applications.

Grid Computing: A loosely coupled system where nodes are geographically distributed and connected over a wide-area network. Grid computing is commonly used for resource sharing across organizations.

2.2 Purposes of Distributed System

A Distributed System (DS) is built on top of a network and tries to hide the existence of multiple autonomous computers. It appears as a single entity providing the user with

whatever services are required (See Definition 2). A network is a medium for interconnecting entities (such as computers and devices) enabling the exchange of messages based on well-known protocols between these entities, which are explicitly addressable (using an IP address, for example). There are various types of DS, such as Clusters [14], Grids [15], P2P (Peer-to-Peer) networks [14, 16], distributed storage systems and so on. A cluster is a group of interconnected computers that appear as a single supercomputer, generally used in high-performance scientific, engineering, and business applications. A grid is a type of DS that enables coordinated sharing and aggregation of distributed, autonomous, and heterogeneous resources based on users' Quality of Service (QoS) requirements. Grids are commonly used to support applications in e-Science and e-Business, which involve geographically distributed communities collaborating to solve large-scale problems. These systems require the sharing of various resources such as computers, data, applications, and scientific instruments.

Peer-to-Peer (P2P) networks are decentralized DS that enable applications such as file-sharing, instant messaging, online multi-user gaming, and content distribution over public networks. Distributed storage systems, such as the Network File System (NFS), provide users with a unified view of data stored across different file systems and computers, which may be located on the same or different networks.

The main features of a DS include [17]:

2.2.1 Functional Separation

Each entity in the system is designed based on the functionality, services provided, capabilities, and purpose it serves.

2.2.2 Inherent Distribution

Information, users, and systems are inherently distributed. Different information is created and maintained by different users and may be stored, analysed, and utilized by different applications that may not be aware of each other's existence.

2.2.3 Reliability

Ensures long-term data preservation and backup (replication) across different locations.

2.2.4 Scalability

Allows the addition of more resources to enhance performance or availability.

2.2.5 Economy

Enables resource sharing among multiple entities, reducing overall costs.

As a consequence of these features, the various entities in a DS can operate concurrently and autonomously. Tasks are carried out independently, and actions are coordinated at well-defined stages through message exchanges. Furthermore, entities within the system are heterogeneous, and failures are independent. Generally, no single process or entity possesses complete knowledge of the entire system state.

2.3 Characteristics of Distributed Systems

DS have become a fundamental paradigm in modern computing, enabling cooperation between multiple autonomous computing entities over a network to perform coordinated tasks. Unlike centralized systems, DS are built to function across multiple machines, often spanning different physical locations, administrative domains, and hardware configurations. Understanding their core characteristics is essential for designing reliable, scalable, and efficient distributed applications. These characteristics define how such systems behave, interact, and meet performance and usability expectations in dynamic and heterogeneous environments, The key characteristics of this environment are outlined below. [18]

2.3.1 Heterogeneity

A DS comprises diverse hardware components, operating systems, and network protocols. This heterogeneity requires interoperability mechanisms to ensure seamless communication. It supports different platforms, including Windows, Linux, and macOS, and relies on communication standards such as TCP/IP and HTTP for data exchange.

2.3.2 Transparency

A well-designed DS hides complexity from users, offering various forms of transparency. Access transparency allows users to interact with the system without needing to know the physical location of resources. Location transparency ensures resources appear local even when they are remote. Replication transparency keeps duplicated data synchronized without user awareness, while concurrency transparency enables multiple users to access shared resources simultaneously. Additionally, failure transparency allows the system to continue functioning despite node failures, and migration transparency ensures processes or data can move between nodes without affecting operations.

2.3.3 Scalability

A DS must efficiently scale as demand increases. It should support size scalability by accommodating an increasing number of nodes, geographical scalability to ensure efficient operation over large distances, and administrative scalability to manage an increasing number of users and policies. Load balancing, caching, and hierarchical system design are essential techniques for maintaining scalability.

2.3.4 Fault Tolerance and Reliability

DS must handle failures without significant disruptions. Fault tolerance mechanisms include redundancy, where critical data is replicated across multiple nodes, failure detection using heartbeat signals, and self-healing techniques such as automatic fail-over and recovery mechanisms.

2.3.5 Concurrency

DS allow multiple processes to execute simultaneously. Proper concurrency management involves synchronization mechanisms like locks and semaphores, deadlock avoidance strategies, and atomic transactions to maintain consistency during simultaneous operations.

2.3.6 Security

Security is crucial in a DS to prevent unauthorized access and cyber threats. Measures such as authentication and authorization help verify user identities, encryption protects data during transmission, and access control mechanisms restrict user privileges based on predefined security policies.

2.3.7 Resource Sharing

DS enable shared access to hardware, software, and data. Hardware resources such as CPUs, GPUs, and storage devices can be distributed across nodes. Software components, including cloud computing services and virtual machines, can be accessed remotely. Distributed databases and file-sharing networks like Hadoop HDFS facilitate data sharing across multiple locations.

2.3.8 Openness and Extensibility

A well-designed DS is open and extendable, supporting standardized interfaces for seamless integration with new technologies. A modular architecture ensures that new components can be added or upgraded without affecting the overall system performance.

2.3.9 High Performance

Performance optimization in DS includes high throughput [11], which ensures efficient processing of large numbers of tasks, low latency to reduce response times, and optimized resource utilization through efficient workload distribution.

Several software companies and research institutions have developed DS frameworks and solutions that address the challenges, driving innovation in modern computing architectures.

2.4 Challenges of Distributed Systems

To investigate the general advantages and limitations of existing development paradigms for DS, several different classes of distributed applications and their main challenges are discussed below. Figure 2.2 illustrates these application classes and their relationship to key criteria such as software engineering, concurrency, distribution, and non-functional aspects. These classes are not meant to be exhaustive but serve to highlight the diversity of distributed computing scenarios and their unique characteristics.

Software engineering has traditionally focused on developing applications for single-computer systems, delivering typical desktop software such as office productivity and entertainment programs. The primary challenges of these applications relate to the functional dimension, specifically how overall application requirements can be decomposed into modular software components while maintaining good software engineering principles such as extensibility, maintainability, and modularity.

Concurrency plays a crucial role in resource-intensive applications that demand significant computational power. Advances in hardware, such as multi-core processors and graphics cards with parallel processing capabilities, have pushed forward the need for concurrency. Applications such as gaming and video manipulation tools rely on multi-threading and parallel execution to achieve high performance. However, concurrency introduces challenges such as maintaining state consistency, avoiding deadlocks and livelocks, and preventing race conditions.

Different classes of naturally distributed applications exist depending on whether data,

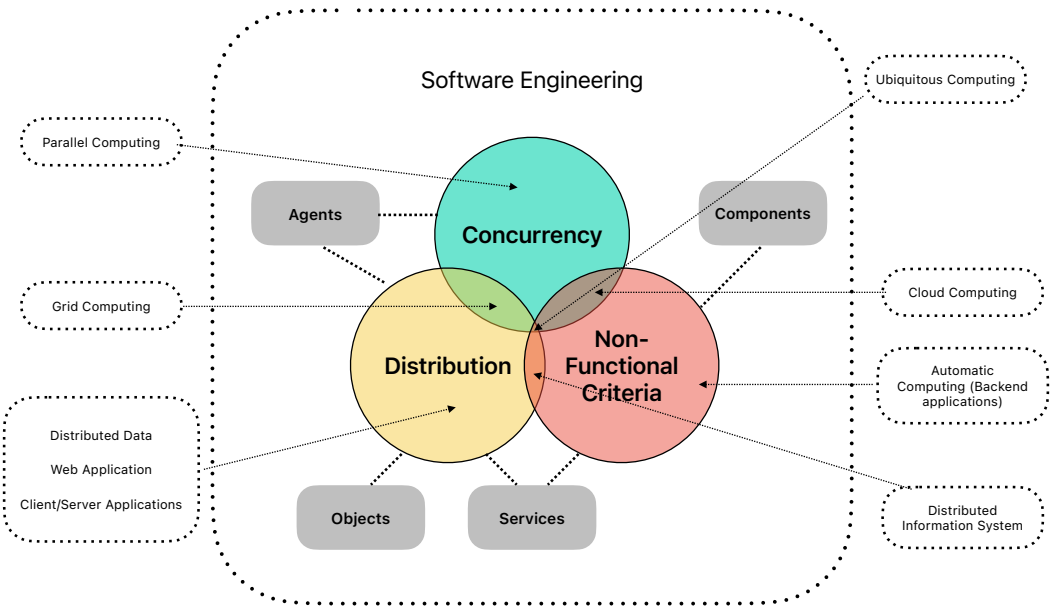

Figure 2.2: Distributed Application in Software Engineering

users, or computation are distributed. Common examples include client-server and peer-to-peer computing applications. Distribution introduces multiple challenges, the most significant being distribution transparency, which aims to hide the complexity of the underlying system. Other critical issues include ensuring openness for future extensions and enabling interoperability, which is often hindered by heterogeneous infrastructure components. Furthermore, modern application scenarios are becoming increasingly dynamic, requiring flexible sets of interacting components that can adapt in real time.

Certain application classes demand strong non-functional characteristics, such as centralized backend applications and autonomic computing systems. The former must ensure secure, robust, and scalable business operations, while the latter focuses on self-properties, including self-configuration and self-healing. Non-functional requirements present particularly complex challenges because they often cross-cut multiple system components. As a result, they cannot be confined to a single module but instead require a system-wide configuration that adheres to non-functional constraints.

Modern distributed applications often involve combined challenges, exhibiting increased complexity by addressing multiple fundamental concerns simultaneously. Coordination scenarios such as disaster management and grid computing applications for scientific calculations must handle both concurrency and distribution. Cloud computing represents a category of applications that share similarities with grid computing but follow a more centralized approach from the user's perspective. In addition, cloud computing places

significant emphasis on non-functional aspects such as service level agreements and accountability. Distributed information systems, including workflow management software, fall into the category of applications that must address both distribution and non-functional aspects. Finally, ubiquitous computing represents one of the most challenging domains, as it involves substantial interconnections with concurrency, distribution, and non-functional requirements, making its implementation particularly difficult.

While challenges such as concurrency and non-functional constraints indirectly involve scheduling decisions, task scheduling itself represents a core challenge in DS. It governs how computational tasks are assigned to processors across nodes to optimize resource usage and performance. In heterogeneous and dynamic environments, scheduling must account for varying execution costs, task dependencies, and real-time system states—making it a critical enabler for efficient and adaptive distributed computing.

2.5 Distributed Systems and Heterogeneous Multiprocessor Systems

In a Distributed Systems (DS) context, each node may itself be a Heterogeneous Multiprocessor Systems (HMS) comprising multiple heterogeneous processors (See Figure 2.4). As such, effective scheduling must operate at both levels: globally across distributed nodes, and locally within each multiprocessor system. The relationship between DS and scheduling in HMS lies in their shared objective: optimizing resource utilization and ensuring efficient task execution across diverse and dynamic computing environments and thus enhance system performance. Due to the unpredictable nature of workloads in these systems, it has become necessary to implement adaptive and intelligent scheduling strategies. Scheduling thus serves as the key mechanism for managing resource allocation, balancing computational loads, and meeting performance objectives by accounting for both system heterogeneity and task-specific constraints.

HMSs represent a subset of DS, where the computational nodes are processors within a shared or interconnected environment, such as data centers, cloud platforms, or high-performance computing clusters. These systems exhibit heterogeneity in processing speeds, power efficiency, and task compatibility, making task scheduling a critical aspect of their operation.

In modern distributed architectures, computing environments are increasingly characterized by hardware diversity and parallel processing capabilities. Two closely related concepts are introduced: Heterogeneous Computing Systems (HCS) and HMS, emphasizing their roles and integration in distributed system design.

2.5.1 Heterogeneous Computing Systems

A Heterogeneous Computing Systems (HCS) [19] refers to a distributed or interconnected computing environment composed of multiple computing nodes that differ in architecture, processing capabilities, and functionalities. These nodes may integrate a variety of hardware components such as general-purpose CPUs, Graphics Processing Units (GPUs), Digital Signal Processors (DSPs), and Field-Programmable Gate Arrays (FPGAs). HCS environments span across different physical systems, and may or may not share memory, depending on the architecture. This heterogeneity enables efficient execution of a wide range of computational tasks but introduces significant challenges in task scheduling, resource management, and system coordination.

2.5.2 Heterogeneous Multiprocessor Systems

In contrast, a Heterogeneous Multiprocessor Systems (HMS) represents a single computing node within an Heterogeneous Computing Systems (HCS), comprising multiple processors or cores of different types. These processors typically share access to a common memory space and operate cooperatively to execute tasks in parallel. Figure 2.3 illustrates a typical multiprocessor architecture. In this context, an HMS serves as a fine-grained unit of computation within the broader HCS in a distributed environment. Optimizing scheduling at the HMS level by effectively utilizing processor diversity is crucial to enhancing the overall performance of the distributed system [20]

By viewing HMS as a granular component of HCS (See Figure 2.4), it becomes evident that performance optimization in distributed systems must address both the macro-level orchestration across nodes and the micro-level scheduling within each node. This layered perspective informs the design of dynamic scheduling strategies that adapt to both global workload distribution and local processor heterogeneity. A summary table (See Table 2.5.2) outlined the differences between HCS and HMS.

2.5.3 Scheduling as the Link

The relationship between Distributed Systems (DS) and Heterogeneous Multiprocessor Systems (HMS) lies in their hierarchical nature: while a DS orchestrates the coordination and communication between multiple nodes, an HMS manages computational workloads at a finer level by efficiently distributing tasks among processors within a single node or cluster.

A critical factor influencing the overall performance of a DS is task scheduling, which determines how workloads are allocated and executed across both heterogeneous processors

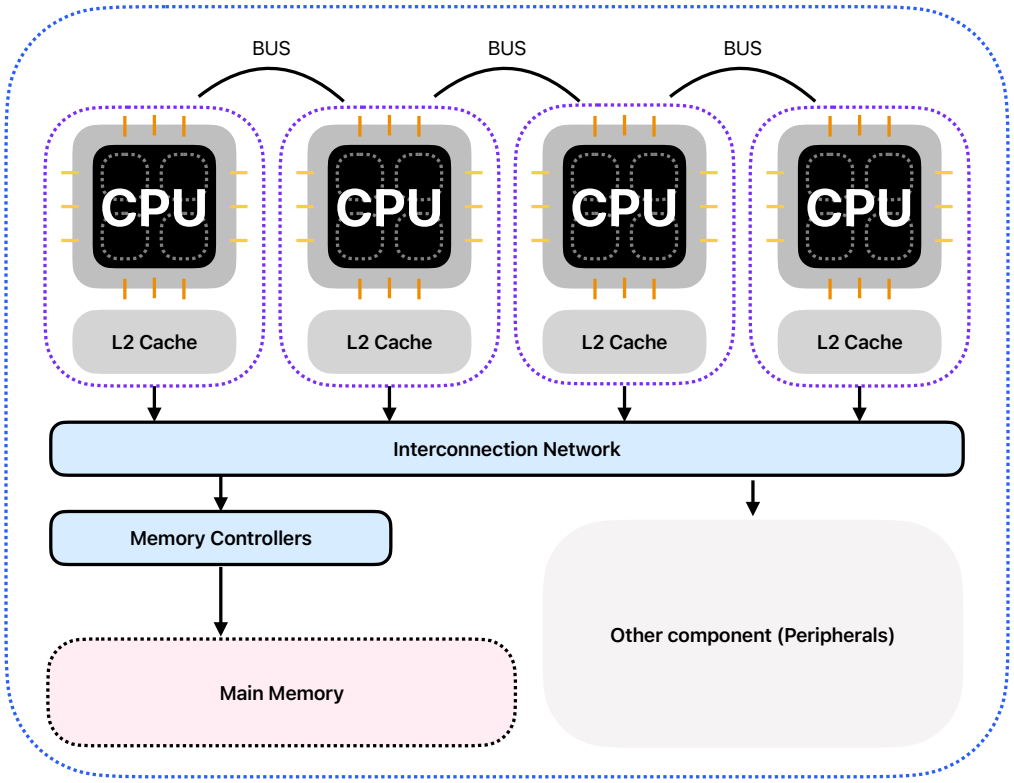

Figure 2.3: Illustration of a Heterogeneous Multiprocessor Architecture

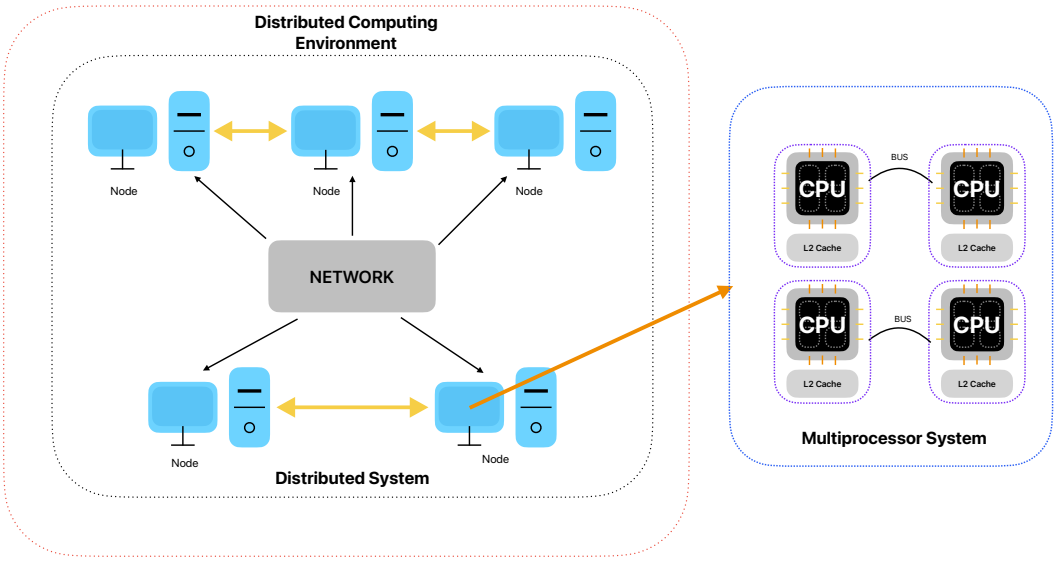

Figure 2.4: Multiprocessor Node Architecture in a Distributed System Context

Feature	HMS	HCS
Scope	Intra-node (within a single system)	Inter-node (across multiple systems)
Components	Multiple processors or cores within one machine (e.g., CPU + GPU)	Multiple heterogeneous devices or nodes (e.g., cloud server + edge device)
Example	A laptop with CPU and GPU, or SoC with CPU + DSP	A distributed system with IoT devices, edge nodes, and cloud servers
Scheduling Target	Local task/thread scheduling among processors	Global job/task scheduling across machines
Relevance to OS	High – requires close coordination at OS level	Moderate to High – involves middleware or orchestration layers
Focus	Task-level optimization, parallelism, and energy efficiency	Resource distribution, system-level scalability, and load balancing

Table 2.1: Comparison between Heterogeneous Computing Systems and Heterogeneous Multiprocessor Systems

in an HMS and multiple nodes in the DS.

Scheduling serves as a fundamental mechanism for enhancing performance in both DSs and HMSs. It ensures that tasks are assigned efficiently to available computational resources, aligning with the following key objectives:

1. Optimizing Makespan: Reducing the total time required to complete all tasks, which is crucial for both systems to maintain performance and responsiveness.

2. Maximizing Resource Utilization (RU): Ensuring that all processing units are effectively utilized, minimizing idle time and energy consumption.

3. Dynamic Adaptation: Adjusting to changes in workload, system conditions, or available resources, particularly important in distributed environments where sudden increases in task demand (known as workload spikes) or unexpected system failures.

4. Managing Heterogeneity: Leveraging the diverse capabilities of processors or nodes to execute tasks optimally, taking into account processing speed, task priority, and execution dependencies.

Poor scheduling can lead to load imbalance, resource underutilization, increased execu-

tion time, and higher energy consumption, ultimately degrading the efficiency of the entire DS. Thus, optimizing scheduling at the HMS level enhances throughput, responsiveness, and scalability, positively impacting the overall performance of the DS.

2.6 Scheduling Paradigm

The concepts of task scheduling and task allocation algorithms are essential in computer systems to determine the execution order of tasks and the nodes on which each task will run [21, 22]. Task scheduling defines the sequence in which tasks are executed, whereas task allocation determines which processor will handle each task [21, 23]. In some cases, scheduling algorithms perform both allocation and sequencing [23, 24]. A task, as the smallest schedulable unit, consists of either threads or processes [25]. Scheduling in distributed computing systems is crucial for optimizing hardware utilization and achieving efficient parallel execution.

In 1999, Kwok and Ahmad [26] highlighted that heterogeneous platforms posed a significant challenge for the development of scheduling algorithms. Then they presented static scheduling algorithms for multiprocessors, which are also applicable to distributed systems, and their classification.

The rise in popularity of these platforms in grid and cloud environments has since driven the emergence of novel scheduling strategies [1, 27–30]. While the core principles of scheduling remain relevant, new optimization objectives have proliferated, substantially enriching the field's literature. As Smith pointed out [31], scheduling is still far from being a completely solved problem. He emphasized the importance of handling uncertainty, supporting controlled changes to solutions, refining constraints, and operating effectively in contexts with multiple self-interested agents.

Finding an optimal scheduling solution in a heterogeneous distributed systems with a finite number of nodes is an NP-hard problem [32]. A problem classified as NP-hard means that its solution can only be found in exponential time, making it infeasible to solve optimally for large-scale systems [33]. The NP-hard complexity class indicates that the problem is at least as difficult as any problem in NP [34]. Since NP problems cannot be solved in polynomial time, approximate algorithms such as heuristic and genetic algorithms are commonly used to provide near-optimal solutions within a feasible time frame [32]. Heuristic algorithms rely on educated guesses or trial-and-error methods to reach a satisfactory solution [35].

Task scheduling can be categorized into static and dynamic scheduling. In static scheduling, computation is performed at compile-time, meaning task execution is predetermined before runtime. In contrast, dynamic scheduling occurs at runtime, allowing task

assignments to adapt based on system conditions and workload changes [23].

These distributed computing paradigms aim to improve performance, scalability, and fault tolerance by efficiently distributing workloads across multiple computational resources [11].

2.6.1 Scheduling Mechanism

In general, job (defined as a set of tasks (See Definition 3)) scheduling consists of two interdependent mechanisms:

1. **Space-sharing** the allocation of processes to available workstations.
2. **Time-sharing** the scheduling of these processes over time. In many systems, additional complementary strategies are employed to further optimize performance.

Definition 3. *A task is a distinct and measurable unit of work requiring specific skills or competencies to complete. In complex systems, tasks are often grouped into jobs, with each job comprising multiple tasks aligned to specialized roles or functions. Task differentiation forms the basis for role specialization and efficient work allocation in both human and automated systems* [1].

Upon job submission, the system performs job placement, selecting the most appropriate set of workstations for execution based on resource requirements, such as memory, CPU time, and deadlines. This selection relies on a resource information table, which may be centralized or distributed, and maintains real-time data such as processor load and available memory [36].

The job is subsequently decomposed into smaller processes, which are dispatched to the designated workstations. At the local level, each workstation applies its own scheduling policies to assign time slices to the processes, aiming to balance performance metrics like response time and fairness.

Moreover, synchronization among distributed processes may be necessary—for instance, if one process must wait for data from another. To mitigate inter-process waiting time, coordinated scheduling is implemented, ensuring efficient communication across the system [37].

To further enhance performance, process migration can be employed. As system load fluctuates, processes from heavily loaded nodes can be migrated to underutilized workstations, thus improving load balancing and reducing job completion time.

[1] https://joint-research-centre.ec.europa.eu/projects-and-activities/employment/job-tasks-and-work-organisation_en

While space-sharing improves response time but may reduce throughput, time-sharing increases throughput at the cost of longer response times. A balanced scheduling mechanism often combines both approaches, enhanced by coordinated co-scheduling and dynamic process migration.

2.6.2 Scheduler

A scheduler is a system component or algorithm responsible for managing the execution of tasks by determining their allocation to computational resources over time. In multiprocessor and distributed systems, the scheduler plays a central role in deciding when, where, and in what order tasks should execute to optimize key performance metrics such as makespan, resource utilization, throughput, and latency. Schedulers may operate under different paradigms—such as static vs. dynamic or centralized vs. decentralized—and often incorporate constraints like task dependencies, deadlines, and processor heterogeneity to make efficient scheduling decisions. The work of a scheduler consists of:

1. **Select**: Select a job or a task
2. **Resource Assignment (Binding)**: Allocate the selected task(s) to appropriate computing resources (e.g., processor, core, node) considering availability, heterogeneity, and performance constraints.
3. **Ordering (Sequencing)**: Determine the execution order of tasks, especially when multiple tasks share the same resource. This may involve queueing policies such as FIFO, priority-based, or deadline-aware ordering.
4. **Dispatching**: Initiate the actual execution of the task on the assigned resource.
5. **Monitoring and Rescheduling (if dynamic)**: Optionally, observe the execution and reassign or reorder tasks in response to workload changes, failures, or updated conditions (mainly in dynamic scheduling systems).

2.6.3 Scheduling Purpose

Scheduling lies at the heart of efficient resource management in computing systems. Its primary purpose is to determine when, where, and how tasks should be executed in order to achieve specific operational goals. These goals often include minimizing execution time (makespan), maximizing resource utilization, ensuring fairness among competing tasks, and meeting deadlines or quality of service (QoS) requirements. In distributed and heterogeneous systems, where resources and workloads can vary widely, effective scheduling

becomes even more critical. This section outlines the core purposes of scheduling, highlighting its role in optimizing system performance, adapting to dynamic conditions, and supporting scalability and reliability across diverse computing environments. The scheduler should align with these following properties:

- General Purpose: A scheduling approach should make minimal assumptions about the nature of the applications it handles. It should support a wide range of job types including interactive, distributed, parallel, and non-interactive batch jobs, while delivering acceptable performance across them. Achieving this generality can be challenging due to potentially conflicting requirements. For instance, real-time jobs require low latency and may benefit from space-sharing strategies, while batch jobs prioritize throughput and may favour time-sharing. A well-designed scheduler must find a balance or trade-off that accommodates these differing needs [37, 38].

- Efficiency: Refers to two aspects to the ability of the scheduler to enhance overall system performance (e.g., minimizing makespan or maximizing throughput) and ensuring the overhead incurred by the scheduling process itself should be minimal, which means that the performance gains are not offset by scheduling costs.

- Fairness: In a multi-user or distributed environment, fairness ensures that no single user monopolizes system resources during periods of high demand. Each user or task should receive an equitable share of computational resources. While fairness is well-addressed in single-node systems, it becomes more complex in distributed environments, requiring mechanisms to monitor and balance resource allocation across nodes.

- Dynamic Responsiveness: A dynamic scheduler should adapt in real time to changes in system load, resource availability, or job characteristics and at the same time, efficiently redistribute tasks or reassign priorities to exploit available resources and maintain performance under fluctuating conditions.

- Transparency: Implies that the behavior and output of a task should not depend on the specific node (local or remote) where it is executed. Users should not be required to manage or even be aware of the underlying execution location. Moreover, applications should not need significant modification to run in the distributed environment. Ideally, remote and local executions should be functionally indistinguishable to the user, except for possible performance benefits.

2.6.4 Static vs. Dynamic Scheduling

In the context of heterogeneous computing systems and distributed architectures, scheduling strategies are typically categorized as either static or dynamic, depending on when and

how scheduling decisions are made.

1. **Static Scheduling** refers to approaches where task allocation and execution order are determined *before* execution begins. This type of scheduling assumes complete knowledge of the application, its tasks, execution costs, dependencies, and available resources. As a result, static scheduling can compute an optimized schedule offline with low runtime overhead. However, it lacks flexibility in dynamic environments, where resource availability or workload may change at runtime. Static methods are well-suited for predictable and repeatable workloads, such as embedded systems or applications with fixed execution patterns [2, 26, 39, 40]
2. **Dynamic Scheduling**, on the other hand, makes scheduling decisions *at runtime*, based on current system states, task arrivals, and resource availability. It is especially valuable in systems where workloads are unpredictable or when operating under variable resource conditions, such as in cloud or edge computing. Dynamic schedulers continuously monitor the system and can adapt to failures, workload fluctuations, or node heterogeneity. While they introduce additional runtime overhead due to monitoring and decision-making, they provide better adaptability and responsiveness in volatile environments [41–43]

The comparison between the two categories is outline in the following Table 2.2.

In practice, hybrid approaches that combine static and dynamic elements are increasingly adopted, leveraging the predictability of static planning with the flexibility of dynamic adjustment.

2.6.5 Scheduling in Homogeneous vs. Heterogeneous Environments

Task scheduling strategies are significantly influenced by the nature of the underlying computing environment. A distinction is typically made between homogeneous and heterogeneous environments, each posing different challenges and requiring tailored scheduling approaches.

In a homogeneous environment, all computing nodes possess identical or nearly identical processing capabilities, memory capacities, and architectural structures. This uniformity simplifies the scheduling process since task execution times are consistent across processors, and decisions can be based purely on workload balancing and queue lengths. As a result, classic scheduling algorithms such as First-Come-First-Served (FCFS) [44, 45], Round-Robin, and simple Min-Min [46] strategies are often effective and sufficient in such settings.

In contrast, heterogeneous environments consist of processors with varying computational powers, memory hierarchies, and possibly different hardware architectures (e.g., a

Aspect	Static Scheduling	Dynamic Scheduling
Decision Timing	Before execution (compile-time or offline)	During execution (runtime)
Flexibility	Inflexible; does not adapt to runtime changes	Highly adaptable to system and workload variations
System Knowledge	Requires complete knowledge of tasks and resources in advance	Operates with partial or evolving knowledge
Overhead	Low runtime overhead	Higher overhead due to runtime monitoring and decision-making
Suitability	Ideal for predictable, repeatable, and static workloads	Suitable for unpredictable, real-time, and dynamic environments
Fault Tolerance	Limited; lacks mechanisms to respond to failures during execution	Strong; can reassign tasks in case of failures or node unavailability
Examples of Use	Embedded systems, scientific computing with known workloads	Cloud computing, distributed systems, real-time applications

Table 2.2: Comparison between Static and Dynamic Scheduling [2]

mix of CPUs, GPUs, and FPGAs). This diversity introduces complexity into the scheduling process, as task execution times can vary greatly depending on the processor to which they are assigned. Consequently, scheduling in heterogeneous systems requires more sophisticated algorithms that consider processor-task affinities, task execution cost matrices, communication overheads, and sometimes Quality of Service (QoS) constraints. Algorithms like HEFT, CPOP [2], and QoS-aware Min-Min [47] are commonly adopted in such environments.

The primary challenge in heterogeneous scheduling lies in optimizing both makespan and resource utilization while accounting for the non-uniform performance of computing units. This often requires predictive models, historical performance data, or real-time monitoring to inform scheduling decisions.

To summarize, while homogeneous systems offer simplicity and predictability in scheduling, heterogeneous environments demand adaptive and cost-aware strategies to achieve optimal performance and fairness across diverse computing resources.

2.7 Task Allocation Strategies

Efficient task allocation is fundamental to achieving high performance in distributed and multiprocessor systems. Depending on the architecture and scheduling objectives, various strategies are employed to assign tasks to processors. This section presents four principal task allocation strategies—Local Scheduling, Global Scheduling, Co-Scheduling Paradigm, and Task Clustering & Partitioning, each suited to different operational contexts and system requirements [48]

2.7.1 Local Scheduling

Local scheduling refers to the strategy where each processor or node independently manages its own task queue. Once tasks are assigned to a processor, the local scheduler decides the execution order based on internal policies such as First-Come-First-Served (FCFS), Round-Robin, or Shortest Job First (SJF) [45] This approach reduces overhead and decentralizes decision-making, making it suitable for systems with minimal inter-task communication. However, it may lead to load imbalance, especially in heterogeneous systems where execution capacities differ significantly across processors.

2.7.2 Global Scheduling

In global scheduling, a centralized or distributed scheduler maintains a global view of the system's task pool and processor states. Tasks are assigned dynamically across all available processors, aiming to optimize overall system metrics such as makespan, throughput, or resource utilization [49]

This approach is more suitable for heterogeneous systems or environments with frequent workload fluctuations, as it allows tasks to be allocated to the most appropriate processors. However, it incurs higher coordination overhead and may face scalability issues in large systems if not designed efficiently [50]

2.7.3 Co-Scheduling Paradigm

Co-scheduling involves the simultaneous scheduling of related or interdependent tasks across multiple processors. This paradigm ensures that parallel components of a job are executed in synchrony, minimizing waiting times due to inter-task communication and improving performance consistency. It is particularly important in multi-threaded or parallel applications, where components frequently exchange data. Co-scheduling enhances execution efficiency but requires sophisticated synchronization mechanisms and careful planning

to avoid contention and idle time [51]

2.7.4 Task Clustering and Partitioning

Task clustering and partitioning are pre-scheduling techniques aimed at optimizing task allocation by grouping tasks based on certain criteria such as data locality, communication frequency, or dependency structure [52]

- Task clustering merges closely related tasks into groups to reduce inter-cluster communication.
- Task partitioning divides the task graph into subgraphs that can be processed independently or in parallel.

These techniques are essential in large-scale or complex DAG-based workloads, allowing for improved parallelism, cache efficiency, and execution predictability. By strategically organizing task execution and minimizing inter-task delays, they form the basis for scalable and efficient scheduling frameworks. These allocation strategies not only address the computational challenges within a single system but also serve as foundational elements for broader scheduling paradigms.

While distributed systems (DS) typically rely on global scheduling strategies due to their geographically distributed nature and hardware heterogeneity, which require coordinated task allocation across multiple nodes. In contrast, multiprocessor systems may adopt either local or global scheduling, depending on the architecture, particularly whether processors share a common scheduler or manage tasks independently.

In the following section, we delve into how these strategies are applied and adapted within both DSs and Multiprocessor Systems, highlighting the distinctions and overlaps in their scheduling requirements and methodologies.

2.8 Scheduling in Distributed Systems and Multiprocessor Systems

Early scheduling approaches in distributed systems, such as Min-Min and Max-Min heuristics, aim to minimize makespan by assigning tasks based on minimum and maximum completion times, respectively. While Min-Min favours shorter tasks to achieve quick completion, Max-Min prioritizes longer tasks to balance the overall load. Despite their simplicity and effectiveness in static environments, both methods often suffer from load imbalance

and limited adaptability to real-time changes. Enhanced variants like QoS-guided Min-Min incorporate Quality of Service (QoS) constraints to improve task prioritization and responsiveness. However, these improvements typically come at the expense of increased algorithmic complexity and reduced scalability in large, dynamic systems.

Recent methods, such as HEFT (Heterogeneous Earliest Finish Time) and CPOP (Critical Path on a Processor)[1, 53], employ task ranking techniques to improve scheduling efficiency but are restricted by high implementation complexity and static assumptions, making them less adaptable to the variability of Heterogeneous Computing Systems (HCS). Evolutionary algorithms, like Genetic Algorithms (GA) and Ant Colony Optimization (ACO) [54], have also been applied for task scheduling in cloud systems, providing adaptive solutions with some success in load balancing and makespan reduction. However, these methods can suffer from high computational costs and may require fine-tuning to adapt to specific HCS requirements.

Because of the strong relationship between performance in IT systems and scheduling, several scientific works have been proposed in this field. Indeed, algorithms [55, 56] performed in multiple steps to solve the problem of matching application needs with resource availability without neglecting Quality of Service (QoS). In HCS, one of the goals of tasks' scheduling is to achieve high system throughput while considering available computing resources and QoS. Thus, to meet minimal makespan requirements, the authors in [47] have proposed a technique to assign tasks to processors according to the minimum execution cost of each task computed on a different processor. However, in the majority of cases, the results obtained show an increase in the makespan measure, which is poor in terms of QoS. Therefore, several researchers, such as Ezzatti et al. [46, 47, 57], have proposed an improved implementation of Min-Min heuristic by taking into account QoS constraints. The researchers in [58] consider that a Genetic Algorithm (GA) should be performed naturally in parallel systems with multiple processing nodes. Thus, they proposed an appropriate allocation by applying genetic operators crossover and mutation. On the other hand, the authors in [59] have adapted a distributed algorithm for cloud systems and proposed a workflow scheduling algorithm that considers dynamic priority for the tasks. This approach undergoes a process of Min-Max normalization [60]. In addition, Jasim et al. [61] present an intelligent algorithm for scheduling tasks in cloud data centres, based on the Cuckoo intelligent methodology. The authors analyse in detail the different optimization methods such as genetic algorithms, greedy algorithms, Ant-lions optimiser and ant colony optimization. The proposed use of an algorithm based on the Cuckoo method is expected to improve the scheduling time and the optimization of resources in dynamic environments, which would contribute to the efficiency of cloud services. While we focus on the optimization aspect of the scheduling problem, Stützle et al. [62] introduced Max–Min Ant System (MMAS), which is an Ant Colony based optimization algorithm that considers

ants as simple agents that progressively construct candidate solutions to treat an NP-hard static combinatorial optimization problem.

Task scheduling in Heterogeneous Multiprocessor Systems (HMS) is a challenging NP-hard problem [34], with the primary objective of optimizing makespan (See Definition 4) and maximizing resource utilization (RU). Various approaches have been developed to address this challenge, leveraging different algorithmic techniques and heuristics.

Task scheduling in HMS involves allocating computational tasks to different processing units while optimizing performance metrics such as makespan and execution efficiency. The efficiency of scheduling depends on various factors:

- Processor computing capacities
- Task execution costs
- System resource utilization

Definition 4. *Makespan The makespan C_{max}, also known as the total schedule length or the completion time, is defined as the maximum finish time (FT) of the last task in the schedule across all processors. It represents the time required to complete all tasks [63], as shown in Equation 2.8.1:*

$$C_{max} = \max(FT(Exit\_Task)) \tag{2.8.1}$$

where:

FT: is the finish time of a task,

C_{max}: is the overall makespan or total schedule length,

Exit_Task: is the last scheduled task.

2.9 Scheduling Techniques

Efficient task scheduling plays a crucial role in distributed computing and HMS, ensuring optimal resource utilization, reduced execution time, and improved system throughput. Various scheduling approaches have been proposed in the literature, ranging from heuristic-based methods to metaheuristic and AI-driven algorithms. This section reviews some of the most prominent scheduling techniques used in these systems.

2.9.1 Heuristic-Based Scheduling Approaches

Heuristic methods are widely used in scheduling due to their computational efficiency in large-scale distributed environments. These methods provide approximate solutions within

a feasible time frame.

- Min-Min & Max-Min Algorithms: The Min-Min algorithm selects the task with the minimum completion time and assigns it to the processor that completes it the earliest [46, 47, 57]. Conversely, Max-Min prioritizes tasks with the maximum completion time, ensuring that long tasks do not delay execution excessively. These approaches work well in homogeneous systems but face challenges in heterogeneous environments due to load imbalance [62].

- QoS-guided Min-Min: An enhanced version of the classic Min-Min algorithm, the QoS-guided Min-Min approach incorporates Quality of Service (QoS) constraints to prioritize high-importance tasks [47]. It balances between task urgency and execution efficiency, thereby improving scheduling in environments where service guarantees and deadlines are critical. However, this enhancement introduces additional complexity, especially in systems with diverse QoS requirements [64].

- Heterogeneous Earliest Finish Time (HEFT): The HEFT algorithm assigns tasks based on their earliest possible finish time, making it one of the most effective static scheduling approaches for directed acyclic graphs (DAGs) in HMS [23]. HEFT prioritizes tasks using bottom-level ranking and schedules them on processors with the earliest completion time.

- Critical Path On a Processor (CPOP): The CPOP algorithm schedules tasks based on their critical path length, assigning all tasks in the critical path to a single processor if possible. While effective in reducing inter-processor communication, it may lead to load imbalance [33].

- Round-Robin Scheduling: A simple yet effective technique, Round-Robin allocates tasks sequentially to available processors, ensuring fairness. However, it does not account for task execution times, making it inefficient for heterogeneous systems [32].

2.9.2 Metaheuristic and Evolutionary Algorithms

Due to the NP-hard nature of task scheduling in HMS and DS, metaheuristic approaches such as genetic algorithms (GA), particle swarm optimization (PSO), and simulated annealing (SA) have been widely explored.

- Genetic Algorithm (GA): GA-based scheduling evolves task assignments through selection, crossover, and mutation, optimizing scheduling efficiency [30, 58, 65].

- Particle Swarm Optimization (PSO): Inspired by swarm intelligence, PSO improves task scheduling by adjusting processor assignments based on the collective movement of par-

ticles [66]. PSO-based scheduling is efficient in minimizing makespan but requires careful parameter tuning.

- Simulated Annealing (SA): This approach explores different scheduling configurations by probabilistically accepting worse solutions to escape local optima. SA is particularly useful for minimizing energy consumption in cloud-based distributed computing [67].

2.9.3 AI and Machine Learning-Based Scheduling

Recent advancements have integrated AI-driven approaches into scheduling to dynamically adapt task assignments in complex DS.

- Reinforcement Learning (RL) Scheduling: Deep reinforcement learning (DRL) is increasingly used to optimize scheduling decisions in real-time, learning from system states and workload changes [68].
- Neural Network-Based Scheduling: AI-driven approaches use neural networks to predict optimal task-to-processor mappings, significantly improving scheduling efficiency in cloud and IoT environments [11].

2.9.4 Hybrid Scheduling Techniques

Hybrid scheduling approaches combine multiple techniques to achieve higher efficiency and adaptability. As mentioned in [32], Hybrid HEFT-GA Models combines HEFT's static scheduling efficiency with GA's adaptability for dynamic workload scenarios.

The field of task scheduling in DS and heterogeneous multiprocessors has evolved significantly, transitioning from rule-based heuristics to intelligent AI-driven techniques. While traditional approaches like HEFT and CPOP remain widely used, emerging methods such as AI-based reinforcement learning and hybrid scheduling demonstrate promising performance in dynamic environments. Future research will likely focus on energy-efficient, adaptive, and real-time scheduling algorithms for large-scale cloud computing and high-performance DS.

2.10 Conclusion

The literature review has explored the evolution of task scheduling and resource management in both HMPs and HCSs. It highlights the significant challenges posed by the diversity of computing resources, dynamic workloads, and the need for efficient task allocation to optimize performance metrics such as makespan and resource utilization (RU).

Traditional scheduling approaches, including Min-Min, Max-Min, and HEFT, have provided foundational methods for task allocation but often struggle in dynamic and heterogeneous environments due to scalability issues, static assumptions, and limited adaptability. Recent advancements, such as QoS-guided scheduling and evolutionary algorithms, have addressed some of these limitations but introduce complexities in implementation and require fine-tuning for specific scenarios.

This chapter lays the foundation for developing advanced scheduling strategies by analysing the limitations of existing algorithms and highlighting areas where innovation is needed. The observations underscore the necessity for a more adaptable and efficient framework capable of addressing the complex demands of HCS, particularly in environments characterized by dynamic workloads and resource variability.

The review highlights the necessity of dynamic adaptation as a unifying concept for managing heterogeneity and variability in computing systems. By addressing gaps in existing methods and integrating real-time feedback with optimization techniques, research in this domain continues to push the boundaries of task scheduling efficiency and system performance.

Chapter 3

Dynamic Adaptation

Dynamic Adaptation (DA) is becoming increasingly important in software development, especially in areas like automotive systems, web services, and computer networks. In these fields, systems often need to adjust to changing conditions in their environment or work across different types of hardware platforms. This growing need encourages the use of technologies that allow software to adapt more flexibly.

However, current research shows that many challenges remain. For example, adapting systems within strict time limits without causing conflicts between features is still an open problem. While there are many adaptation approaches available, most of them are fixed or limited in flexibility. Many systems rely on simple adjustments like changing parameters or configurations, which can restrict how much they can adapt later on.

A good example is the future of automotive systems, where cars might adapt themselves automatically based on their configuration, making things easier for users by removing the need for manual setup or customization [3].

In this chapter, we introduce the DA paradigm and examine its relevance in modern computing environments. We outline key concepts, classification criteria, and representative approaches from the literature that illustrate how DA enhances system flexibility, resilience, and responsiveness in dynamic operational contexts.

3.1 Adaptation and Adaptability

First of all, flexibility and adaptability are synonyms in the standard glossary of software engineering terminology. **Flexibility = adaptability** and both are defined as: "The ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed." [70] Second, adaptation is defined as

the ability of software systems to withstand changes in their environment. The following quote provides a hint into the nature of adaptation in software systems, mainly at the level of abstraction at which of a system is enabled: "a software system will be adaptable provided its software architecture is itself adaptable in the first place". Third, there is a further distinction that needs to be considered, whether we refer to adaptability or adaptiveness. Addressing this distinction is a matter of recognizing the differences between static and DA.

3.2 Definition of Dynamic Adaptation

We call system with DA if the system is able to adapt to an environment other than from which it was designed and to perform the planned tasks correctly without external intervention and interact with the elements of its environment. (See Definitions 5, 6)

Definition 5. *Dynamic Adaptation refers to the system's ability to adjust task allocation, resource usage, and operational parameters in real-time based on workload characteristics, processor performance, and system constraints (e.g., power, thermal limits).*

Definition 6. *Dynamic Adaptation refers to the capability of a system to adjust its behavior, configuration, or resource allocation in response to changes in its operating environment, workload, or internal state. This concept is applicable across various domains, such as computing, robotics, biological systems, and organizational processes, emphasizing flexibility, resilience, and efficiency in achieving goals. [71]*

3.2.1 Key Characteristics of Dynamic Adaptation

The research [72] outlined key characteristics of dynamic adaptation (DA) summarized as follows:

- Real-Time Adjustments: The system responds to changes as they occur, ensuring continuity and optimal performance without requiring manual intervention.
- Feedback-Driven: DA relies on continuous monitoring of system performance and external conditions, using feedback to guide adjustments.
- Context-Aware: Adaptation is tailored to the specific context, considering factors like available resources, task priorities, and environmental constraints.
- Self-Management: A dynamically adaptive system often exhibits self-optimization, self-configuration, and sometimes self-healing capabilities, minimizing external dependencies.

3.2.2 Examples of Dynamic Adaptation

- **Computing Systems**: A cloud platform scaling resources up or down based on user demand; task scheduling algorithms reallocating workloads to balance processor usage and prevent bottlenecks.
- **Robotics**: A robot adapting its navigation path based on real-time obstacle detection.
- **Biological Systems**: The human body maintaining homeostasis, such as adjusting heart rate in response to physical activity.

3.2.3 Software adaptation

Defined as the ability of a software system to reconfigure itself in response to changing conditions. This capability is often referred to as software adaptability, or more specifically, software adaptiveness, emphasizing the system's capacity for self-reconfiguration without manual intervention by software engineers [3].

3.2.4 Difference between software adaptability and software adaptiveness

Software adaptability focuses on enabling the evolution and reuse of software components across future contexts. In contrast, software adaptiveness refers to the system's ability to dynamically alter its behavior at runtime to prevent performance degradation and resource conflicts [73].

3.2.5 The notion of dynamic adaptive system

Refers to a system's ability to adjust dynamically in response to evolving conditions such as changing user requirements, system intrusions or faults, variations in the operational environment, and fluctuations in resource availability.

3.3 Dynamic Adaptation Techniques

Adaptation plays a crucial role in enhancing the performance [11] and resilience of distributed systems, particularly in environments characterized by dynamic workloads and heterogeneous resources [70]. As systems become increasingly complex and responsive to changing conditions, it is essential to implement effective adaptation techniques that can adjust system behavior in real time. These techniques vary based on their mechanisms,

scope, and level of automation. Primary adaptation techniques are classified into three main categories, each offering distinct strategies for maintaining optimal system performance under variable conditions [74]

- **Dynamic Component Linking/Unlinking**: Enables the addition of new functionalities or the replacement of existing ones by modifying or substituting the current communication channels between components.
- **Generic Interceptors**: Do not alter the behavior of components directly. Instead, they intercept and manipulate messages exchanged between components to inject additional behavior or monitoring capabilities.
- **Reconfiguration Techniques**: Involve adjusting internal settings or global system parameters in response to environmental changes, with the goal of maintaining performance, reliability, or functionality.

To systematically classify various approaches within the domain of dynamic adaptation (DA), it is necessary to first explore the fundamental dimensions along which adaptive systems can be evaluated. Adaptation denotes a system's capacity to respond to environmental changes through mechanisms such as structural adjustments, architectural reconfigurations, parameter tuning, or combinations thereof. A primary classification axis is the degree of anticipation associated with such changes—that is, whether the system is designed to respond to unforeseen, unanticipated events or to pre-identified, expected variations. Unanticipated adaptation typically presents greater conceptual and implementation challenges due to its inherent unpredictability.

For anticipated adaptation, further categorization may be established based on characteristics such as the scope of the adaptation (ranging from localized adjustments to full architectural changes) and the adaptation mechanism employed (compositional vs. parametric). Importantly, existing literature suggests that compositional and parametric approaches should not be viewed as mutually exclusive paradigms. Rather, they represent orthogonal dimensions that can be combined within hybrid strategies, enabling more flexible and context-sensitive adaptive behavior [3].

Before examining DA approaches, we present the classification criteria used to evaluate current research efforts in the field of dynamic adaptation. These criteria, inspired by the foundational work of Fox et al. [3], provide a structured framework for characterizing the adaptability features of software systems. The criteria are as follows:

3.3.1 Unanticipated Adaptation

This criterion assesses the extent to which a system is capable of adapting to unforeseen changes in its environment. A high level of unanticipated adaptation reflects the system's ability to handle unknown triggers and requirements at runtime without prior specification. Frameworks that support this capability are generally regarded as more flexible and generic, capable of operating effectively under dynamic and unpredictable conditions.

3.3.2 Scope of Adaptation

The scope criterion refers to the breadth of system components affected during the adaptation process. It is categorized as follows:

- Low: Adaptation is confined to a single or localized component.
- Medium: Adaptation involves a subset of system components.
- High: Adaptation spans the entire system architecture.

A broader scope generally reflects greater complexity and a more integrated adaptation strategy.

3.3.3 Parametric Adaptation

Parametric adaptation involves modifying predefined parameters within software entities such as components, services, or methods to achieve the desired adaptive behavior. Although this approach allows for runtime tuning, it may suggest a limited level of flexibility, as it depends heavily on a predefined range of configurable options.

3.3.4 Compositional Adaptation

This category evaluates whether the system achieves adaptation by inserting, replacing, or removing functional units—typically components or services—at runtime. Compositional adaptation often relies on binding and unbinding mechanisms and enables a more modular and scalable adaptation strategy. Systems employing this approach tend to support a higher level of structural flexibility.

3.3.5 Dynamicity

Dynamicity refers to the system's ability to undergo structural or behavioral changes during execution without requiring redeployment or shutdown. To ensure consistency, certain

preconditions, such as system quiescence (a state in which no ongoing operations are disrupted), may be necessary. This capability is crucial for real-time systems and environments with high availability requirements.

3.3.6 Static Adaptation

In contrast to dynamic adaptation, static adaptation occurs at design time or requires system redeployment for any configuration changes. Static approaches typically involve parametric configurations with limited runtime flexibility. While less resource-intensive and simpler to implement, static adaptation is generally inadequate for highly dynamic operational environments.

3.3.7 Tools

This classification criterion evaluates whether the examined approach is supported by practical tools, such as development environments or runtime monitoring frameworks. The research teams referenced here in were selected based on a comprehensive review of the literature, with priority given to those producing notable contributions in the domain of dynamic adaptation (DA). Preference was also accorded to teams affiliated with academic research groups, consortia, or institutions demonstrating sustained engagement in this field. The purpose of this survey is not to provide an exhaustive account of all DA approaches, but rather to elucidate the proposed classification framework and highlight key characteristics observed in representative works.

3.4 Approaches for Dynamic Adaptation

3.4.1 Adaptive CORBA (ACT)

ACT is a language-independent template designed for developing object-oriented frameworks and enhancing CORBA applications [74]. It introduces generic interceptors, which are specialized request interceptors registered with the ORB at startup. These interceptors can be either static or dynamic: dynamic interceptors can be registered or unregistered at runtime, whereas static ones remain fixed once the system is running. The approach also leverages weaving, a mechanism that relates dynamic interceptors at runtime. The concept of generic interceptors supports unanticipated adaptation, as they are initially registered without predefined behavior and can be dynamically updated to provide required functionalities. For these reasons, ACT is classified in Table 3.1 as exhibiting a high degree of unanticipated adaptation. Its scope of adaptation is considered moderate since only

dynamic interceptors can be modified and it shows a medium level of parametrization due to the reliance on proxies and redirection. Additionally, it demonstrates a high degree of compositionality.

3.4.2 Dynamic Adaptive System Infrastructure (DAiSI)

DAiSI is a framework designed to enable dynamic adaptation at three primary levels: component service usage, component service implementation, and configuration adaptation [75]. The first adaptation type supports the runtime selection of components and services based on quality attributes. The second allows for behavioral modifications in component implementations. The third aims at non-localized reconfiguration of component interrelations, enabling activation or deactivation of services.

DAiSI is based on a component model tailored for DA and relies on a formal foundation. It combines both parametrization and compositional mechanisms. However, its ability to handle unanticipated changes remains limited. Adaptation is primarily driven by a configuration component manager (referred to as a browser), and no explicit mechanism is provided for reacting to unforeseen changes in the environment. Despite this limitation, the framework offers a moderate level of tool support. These characteristics are summarized in Table 3.1.

3.4.3 DynamicTAO and 2K

DynamicTAO is an extension of the TAO ORB (Object Request Broker), part of the ACE ORB framework. It introduces support for runtime reconfiguration by allowing dynamic linking and unlinking of ORB components [73]. One of its key features is the ability to remotely reconfigure and replace ORB components without requiring a system restart—an essential capability for runtime dynamic adaptation.

DynamicTAO also supports the deployment of new component implementations during execution. Given its design, the scope of adaptation is considered high, as the architecture theoretically permits any ORB component to be adaptable at runtime.

3.4.4 iPOJO Components

iPOJO is a runtime component framework that operates over OSGi, a platform that promotes modular service-oriented applications [73, 74]. The core mechanism in iPOJO is the injection of Plain Old Java Objects (POJOs) at runtime. Adaptation is achieved primarily through dependency management and service provisioning, while the core business logic resides within the POJOs.

Redirection of dependencies is managed via handlers, configured using metadata defined in XML descriptors. Although the notion of "service" is used, it closely resembles the concept of "features." Adaptation in iPOJO is constrained by its reliance on the underlying runtime environment, resulting in a moderate scope of adaptation.

3.4.5 Mobility and Adaptation enAbling Middleware (MADAM)

MADAM offers a component model extended with capabilities for adaptation [75]. It facilitates system variation through the recursive application of pre-defined realization plans, which are designer-specified compositions of components. The model includes an adaptation manager that handles context, configuration, and adaptation at runtime.

MADAM is categorized as mid-level compositional and highly parametric. Since the adaptation paths are pre-specified by designers, the framework exhibits a low level of unanticipated adaptation. This classification is reflected in Table 3.1.

3.4.6 Model-Based Development of Dynamically Adaptive Software (MBD DA)

The MBD DA approach focuses on the formal modeling of adaptive program behavior, with a particular emphasis on reliability and system consistency [73]. Adaptations are represented through state-machine models, and adaptive behaviors are defined as sets of transitions known as adaptation sets.

This methodology ensures safe adaptations by analyzing dependencies among components and identifying valid execution sequences. The system is capable of component insertion, removal, and replacement in response to external changes. The approach has been demonstrated using a wireless multicast video application.

MBD DA supports both static and dynamic analysis and has been classified accordingly. It provides a moderate level of tool support, though its primary contribution lies in offering a formal framework for the verification of adaptive programs rather than a practical adaptation middleware.

3.4.7 A Component System for Pervasive Computing (PCOM)

PCOM is a distributed application framework that enables adaptation through signaling mechanisms and predefined strategies [70]. In PCOM, components interact to fulfill their functional dependencies, resembling service-based interaction models. Applications are structured as hierarchical trees of components, with dependencies forming the edges of this tree.

However, the nature of dependency relationships—whether limited to the tree hierarchy or allowing more general interconnections—is not fully specified, which imposes limitations on the flexibility of the model. Due to the reliance on predefined strategies, PCOM is considered to offer only a medium level of unanticipated adaptation.

Although PCOM supports runtime adaptation, its compositional capabilities are constrained. As a result, it is classified as more parametric than compositional, but highly dynamic, as reflected in Table 3.1.

3.5 Comparison of Dynamic Adaptation approaches

All the reviewed approaches are inherently tied to specific component models, service abstractions, or middleware frameworks. A preliminary analysis indicates that the development of a truly generic and universally applicable adaptation model remains an open challenge in the field.

As discussed in the previous section, most approaches focus on achieving adaptation dynamically, at runtime resulting in runtime-adaptive systems. However, the underlying mechanisms and degree of dynamicity vary. For instance, some frameworks rely on reconfiguration mechanisms (e.g., DAiSI), while others implement adaptation by redefining dependencies as a dependency tree (e.g., PCOM). In contrast, systems like DynamicTAO support dynamic linking and unlinking of components, along with facilities to upload new code implementations.

Another important distinction between research efforts is the degree of anticipation of change. Most approaches are designed with a fixed set of change sources and corresponding strategies to manage them. In this regard, ACT stands out as one of the most adaptive frameworks [76–78] in terms of unanticipated adaptation, particularly within component-based architectures.

Finally, the scope of adaptation appears to be closely related to the level of compositionality and parametrization supported by the framework. In some cases, the extent of changes is limited by the underlying architecture. For example, MADAM represents a highly parametric approach but does not support runtime adaptation. Moreover, not all frameworks are built upon a formal foundation. Formal methods, as used in DAiSI and MADAM, provide clearer definitions and add precision to the adaptation process.

We identified that most frameworks for dynamic adaptation are based on component models. While in the case of service-oriented approaches, most work in the literature seem to have a broad definition of services that is somewhat same as other definitions such as features. Even more, the distinction between components and services is in most cases

Concept approach	ACT (COBRA)	DAiSI	Dynamic TAO	iPOJO	MADAM	MBD DA	PCOM
Unanticipated	***	*	*	*	*	**	**
Scope	**	**	***	**	***	**	**
Parametric	**	***	***	*	***	*	***
Compositional	**	***	***	*	***	*	***
Dynamicity	***	***	***	***	*	**	***
Static	*	*	*	*	**	**	*
Tool	**	**	*	*	*	**	**

Table 3.1: Approaches Classification [3]

Low level = "*"	Medium level = "**"	High level = "***"

not clear, which renders their underpinnings and composition mechanisms unclear. In addition, there is a lack of adaptation mechanisms at the level of services or components logic, behavior itself.

3.6 Conclusion

A major challenge in distributed systems is optimizing critical performance metrics, including makespan, resource utilization, latency, and throughput. The lack of adaptability in traditional approaches can lead to inefficient resource allocation, making real-time scheduling essential for improving computational efficiency.

Dynamic Adaptation (DA) emerged as a crucial concept in both heterogeneous multiprocessor systems (HMSs) and heterogeneous computing systems (HCSs), as it addresses the challenges arising from their inherent diversity and dynamic operational contexts. Although its implementation and scope may differ between HMSs and broader HCSs, DA consistently plays a vital role in optimizing task scheduling by enabling systems to respond effectively to workload fluctuations, resource availability, and execution uncertainties. In particular, scheduling within multiprocessor systems requires balancing multiple objectives such as minimizing makespan, maximizing resource utilization, and maintaining adaptability under real-time conditions. Traditional scheduling algorithms like Min-Min and Max-Min, despite their effectiveness in static scenarios, often fall short in dynamic environments due to their rigid task allocation strategies, thereby highlighting the need for more adaptive, context-aware scheduling approaches.

In response to these limitations, recent research has explored optimization techniques inspired by the Knapsack problem and dynamic programming. To address the challenges of task scheduling, this work first introduces a novel algorithm designed to efficiently al-

locate independent tasks using dynamic programming principles, ensuring adaptability to varying workloads and system conditions. Building on this foundation, the research further develops a Knapsack-based co-scheduling algorithm a new approach that combines the strengths of dynamic programming and knapsack optimization. Specifically designed for heterogeneous multiprocessor systems (HMS), this algorithm dynamically balances resource utilization and minimizes makespan by systematically partitioning tasks based on priority levels and estimated makespan thresholds. It effectively adapts to the diversity of processing capabilities across processors and addresses both task precedence and resource constraints.

Overall, while traditional methods provide a strong foundation, our proposed approaches offer a more adaptive and robust solution for complex HCS environments by integrating real-time scheduling strategies with resource-aware optimization.

Chapter 4

Dynamic Task Allocation using Dynamic Programming (DyTAg)

Modern multiprocessor systems, particularly in heterogeneous computing environments, are designed to handle increasingly complex and dynamic workloads. As these systems scale in size and capability, efficient task scheduling becomes essential to fully exploit available computational resources. One of the most pressing challenges in such environments is the dynamic allocation of independent tasks where no inter-task dependencies exist in a way that minimizes execution time (makespan) and ensures balanced resource utilization.

In increasingly dynamic and heterogeneous distributed systems, maintaining high performance is an ongoing challenge. A key part of this challenge lies in the efficient management of computational resources particularly within multiprocessor systems, which serve as the fundamental or granular units of distributed architectures. As workloads fluctuate and system conditions evolve in real time, static scheduling strategies quickly become inadequate. This raises a central question: how can dynamic adaptation (DA) be effectively applied to enhance the performance of DS, especially through the optimization of task scheduling within heterogeneous multiprocessor nodes?

Dynamic Adaptation (DA) strategies are vital in addressing the variability of system states, resource availability, and workload fluctuations. They enable the scheduler to respond in real-time to changes within the system, leading to more robust and efficient execution patterns.

The complexity of task scheduling in heterogeneous computing systems requires the development of efficient and adaptive methodologies to optimize resource allocation and execution time. This chapter presents the methodological framework adopted in this research to address the challenges of dynamic task scheduling in distributed and multiprocessor environments. The proposed approaches aim to enhance system performance by

leveraging advanced scheduling techniques, optimization models, and DA strategies.

The first part of this chapter provides a formal definition of the scheduling problem, outlining key constraints and objectives in heterogeneous multiprocessor environments. Furthermore, this chapter details the mathematical models underlying the proposed approach, including the dynamic programming formulation and computational complexity analysis. Given the inherent NP-hard nature of task scheduling in such environments, the study explores heuristic techniques to derive feasible and near optimal solutions.

Then we introduced the *DyTAg* algorithm, a novel dynamic scheduling technique designed to optimize task allocation in heterogeneous computing systems. The performance evaluation criteria and comparative analysis methodologies are also discussed, providing insights into the effectiveness of *DyTAg* in real-world computing environments.

The evaluation focuses on scenarios involving independent tasks, where *DyTAg*'s performance is analysed in terms of makespan reduction and load balancing. Its effectiveness is compared against standard algorithms such as Min-Min and QoS-guided Min-Min, providing insights into its relative strengths and adaptability in different contexts.

By structuring the methodology around adaptive scheduling principles, combinatorial optimization techniques, and performance-driven evaluation metrics, this chapter lays the foundation for the experimental validation and comparative assessment presented in the subsequent chapters.

4.1 Scheduling Problem Definition

In this section, we first introduce the scheduling problem within the context of heterogeneous multiprocessor systems (HMS). This is followed by the formulation of a scheduling model that incorporates key evaluation criteria such as processor computing capabilities, total execution time, and overall system resource utilization (RU).

Heterogeneous multiprocessor systems consist of processors with varying speeds and capabilities, making the scheduling process more complex compared to homogeneous systems. Each task's execution time depends on the processor to which it is assigned, and an optimal scheduling algorithm must balance the workload across processors to minimize idle time and maximize resource utilization.

Definition 7. *Scheduling System*
The scheduling system in HMS is defined by the quintuple $S = (T, C, P, R, M)$*, where:*

$T = \{t_1, \ldots, t_n\}$ *is the set of available non-preemptive tasks,*

$P = \{p_1, \ldots, p_m\}$ *is the set of heterogeneous processors,*

C is the task execution cost matrix, where $C[i][j]$ represents the processing time of task t_i on processor p_j,

$R = \{r_1, \ldots, r_n\}$ is the set of task ranks (or weights),

M denotes the makespan (see Definition 4).

The researchers are often attracted by applying algorithms and models coming from local search area, to raise the scheduling challenges. Thus, local optimization techniques and DP methods [80] have been proposed and succeeded to provide efficient schedulings.

Definition 8. *Dynamic programming*
Dynamic programming (DP) is a method used for solving complex problems by breaking them down into simpler subproblems. It is particularly effective for optimization problems where the solution can be constructed from the solutions of overlapping subproblems. In the context of task scheduling, DP helps in determining the best assignment of tasks to processors by systematically evaluating all possible task allocations and selecting the one that minimizes makespan and maximizes RU [80].

4.2 Dynamic Programming Model

In this study, we employ a Dynamic Programming (DP) approach to address the task scheduling problem in a heterogeneous multiprocessor system. Our primary focus is on systems with independent tasks, meaning there are no precedence constraints between tasks. This simplification allows us to concentrate on optimizing the allocation of tasks to processors to minimize the overall makespan, while accounting for the heterogeneity of the system.

Due to the NP-hard nature of discrete optimization problems, researchers are often drawn to applying algorithms and models from local search techniques to address scheduling challenges. As a result, local optimization methods and dynamic programming approaches [81] have been proposed and have proven effective in delivering efficient scheduling solutions.

Indeed, dynamic programming (DP) has emerged as a well-suited solution, as it systematically explores all possible task assignments while maintaining computational efficiency. By decomposing the scheduling problem into smaller sub-problems, DP enables the incremental computation of an optimal allocation, ensuring that each intermediate step contributes toward the global optimum.

The aim of this work is to highlight the efficiency of our approach, which introduces dynamic programming for solving scheduling problems.

4.2.1 Definitions and Notation

- **Tasks**: $T = \{t_1, t_2, \ldots, t_n\}$ represents a set of n tasks, where each task t_i has a specific workload.
- **Processors**: $P = \{p_1, p_2, \ldots, p_m\}$ represents a set of m heterogeneous processors, each with a distinct processing speed.
- **Processing Times**: For each task t_i and processor p_j, let $c_{i,j}$ denote the processing time required to complete task t_i on processor p_j. This processing time depends on both the task's workload and the processor's speed. It is defined as:

$$c_{i,j} = \frac{w_i}{s_j} \tag{4.2.1}$$

  where:

  w_i is the workload of task t_i,

  s_j is the speed of processor p_j.

- **Decision Variable**: Let $x_{i,j}$ be a binary decision variable such that:

$$x_{i,j} = \begin{cases} 1 & \text{if task } t_i \text{ is assigned to processor } p_j, \\ 0 & \text{otherwise.} \end{cases}$$

4.2.2 Dynamic Programming Mathematical Model

Let T be a set of tasks to schedule, and T_k be the subset of tasks assigned to processor P_m. T_k represents the best subset of jobs that satisfy the recursive relation given in Equation 4.2.2.

1. **Recursive Relation** The recursive relation captures the trade-off between assigning tasks to different processors to balance the workload. For each task t_i and processor p_j, the recursive relationship is expressed as:

$$DP(i, j) = \min\left(\max\left(DP(i-1, j), c_{i,j}\right)\right) \tag{4.2.2}$$

   where:

   $DP(i-1, j)$: The makespan when assigning $i-1$ tasks among j processors.

   $c_{i,j}$: The processing time required for task t_i on processor p_j.

2. **Boundary Conditions**

- **Base Case**: If there are no tasks ($n = 0$), then:

$$DP(0, j) = 0 \quad \forall j$$

- **Single Processor Case**: If there is only one processor ($j = 1$):

$$DP(i, j) = \sum_{k=1}^{i} c_{k,1}$$

which is the cumulative time to execute all tasks sequentially on p_1.

3. **Objective Function** The goal is to minimize the maximum completion time across all processors, i.e., the makespan $C_{\max}$:

$$C_{\max} = \min_{j} (DP(i, j)) \tag{4.2.3}$$

4.3 Proposed Approach: DyTAg

To address the aforementioned limitations, we propose the Dynamic Task Allocation using Dynamic Programming (DyTAg) approach [82], an adaptive scheduling method that leverages dynamic programming (DP) to optimize task allocation in heterogeneous multiprocessor systems. DyTAg enhances existing methods by incorporating real-time task reassignment mechanisms, thereby improving load balancing and minimizing execution time across diverse processing units.

4.3.1 Dynamic Programming for Scheduling

DP provides a structured approach to solving optimization problems by breaking them into smaller subproblems. In DyTAg, DP is used to minimize makespan while optimizing task allocation across multiple processors.

4.3.2 Mathematical Model

To formally represent the scheduling problem in a heterogeneous computing environment, we introduce a mathematical model that captures the key elements involved in task execution and processor assignment. This model provides the foundation for developing and analyzing the proposed scheduling algorithms. Let:

- $T = \{T_1, T_2, ..., T_n\}$ be the set of tasks.
- $P = \{P_1, P_2, ..., P_m\}$ be the set of processors.

- $C_{i,j}$ represent the execution cost of task T_i on processor P_j.

The objective function is:

$$\min \max \sum_{i=1}^{n} C_{i,j} \tag{4.3.1}$$

where the constraint:

$$\sum_{j=1}^{m} x_{i,j} = 1, \quad \forall i \tag{4.3.2}$$

ensures each task is assigned to exactly one processor.

4.3.3 Proposed Model

Based on the scheduling model (see Definition 7), our approach, DyTAg, is a task scheduling algorithm specifically designed for heterogeneous computing systems. By leveraging dynamic programming (DP), DyTAg effectively combines resource optimization with completion time minimization.

To apply DP for minimizing the makespan, we define a DP state that keeps track of the minimum achievable makespan as tasks are assigned one by one. The goal is to allocate tasks to processors such that the maximum load across all processors is minimized. The steps of the method are outlined as follows:

1. **Recurrence Relation**

   - **Base Case**: If there are no tasks, the makespan is zero for all processors:

   $$DP(0, j) = 0 \quad \forall j = 1, 2, \ldots, m$$

   - **Recursive Case**: For each task T_i and each processor P_j, the DP formula updates $DP(i, j)$ by considering the workload of assigning T_i to P_j, and finding the maximum time required by any processor:

   $$DP(i, j) = \min_{m \in \{1,\ldots,m\}} (\max (DP(i-1, k), DP(i, k) + c_{i,j})) \tag{4.3.3}$$

   The inner maximum accounts for the load balancing effect of assigning task T_i to processor P_j.

   The outer minimum ensures that the minimum makespan across all valid assignments is found.

2. **State Definition** Let $DP(i, j)$ represent the minimum possible makespan after assigning the first i tasks across j processors.

3. **Solution** The solution is obtained by calculating the final $DP(i, j)$ values through iterative computation, minimizing the maximum makespan across all processors. (See Equation 4.3.4)

$$C_{\max} = \min_{j} (DP(n, j)) \tag{4.3.4}$$

where:

$DP(n, j)$: Represents the makespan after all n tasks are assigned across m processors.

This first study formed the foundation of our research into dynamic scheduling for heterogeneous computing systems, ultimately leading to the development of an innovative scheduling approach. This research culminated in a peer-reviewed publication[1], where the proposed techniques were validated and their effectiveness demonstrated through detailed experimental analysis. This achievement underscores the relevance and impact of our research in advancing task scheduling strategies for modern distributed systems particularly in multiprocessor system.

4.3.4 DyTAg Algorithm

Traditional scheduling algorithms, such as Min-Min, Max-Min, and HEFT, often struggle with dynamically changing workloads in heterogeneous computing systems. These approaches typically assume static task allocation, leading to inefficiencies in resource utilization and increased makespan when workloads fluctuate.

By integrating dynamic programming principles, DyTAg systematically decomposes the scheduling problem into smaller subproblems, solving them recursively to achieve an optimal solution. Unlike static scheduling techniques, DyTAg continuously adapts to workload variations, making it particularly suitable for cloud computing, scientific simulations, and high-performance DSs.

1. **Algorithm Steps**
   (a) Initialize available processors and tasks.
   (b) Calculate execution costs for all task-processor assignments.
   (c) Apply dynamic programming to determine optimal scheduling.
   (d) Assign tasks to processors based on computed DP results.
   (e) Update scheduling dynamically as new tasks arrive.

[1] https://asjp.cerist.dz/index.php/en/article/265248

2. **Algorithm**

Algorithm 1 DyTAg (Dynamic Task Allocation using Dynamic Programming)

Input: $T = \{t_1, t_2, \ldots, t_n\}$: Set of tasks, $P = \{p_1, p_2, \ldots, p_m\}$: Set of processors, $C[i][j]$: Execution cost matrix where $C[i][j]$ is the cost of task t_i on processor p_j

Output: A: Assignment vector such that $A[i] = j$ indicates task t_i is assigned to processor p_j; $C_{\max}$: The final makespan

Initialize processor load array: $L[j] \leftarrow 0$ for all $j = 1$ to m
Initialize assignment array: $A[i] \leftarrow -1$ for all $i = 1$ to n
for $i = 1$ to n **do** ▷ For each task
    $min\_makespan \leftarrow \infty$
    $best\_processor \leftarrow -1$
    for $j = 1$ to m **do** ▷ Try each processor
        $temp\_makespan \leftarrow \max(L[j] + C[i][j], \max_{p \neq j} L[p])$
        if $temp\_makespan < min\_makespan$ **then**
            $min\_makespan \leftarrow temp\_makespan$
            $best\_processor \leftarrow j$
        end if
    end for
    t_i to $best\_processor$: $A[i] \leftarrow best\_processor$
    $L[best\_processor] \leftarrow L[best\_processor] + C[i][best\_processor]$ ▷ Update load of processor $best\_processor$
end for
$C_{\max} \leftarrow \max_{j=1}^{m} L[j]$
Return $A, C_{\max}$

4.4 Experimental Studies

This section presents the experimental studies conducted to evaluate the performance and effectiveness of the proposed scheduling algorithms in heterogeneous computing environments. All experiments were executed on a machine equipped with macOS Ventura 13.2.1 (64-bit), running on an Apple Silicon M1 processor with an 8-core GPU and 8 GB of RAM. The algorithms were implemented using the Python programming language, ensuring portability and reproducibility of the results. This simulation setup provides a consistent platform for comparing performance metrics such as makespan, resource utilization, and load balancing under various workload scenarios.

4.4.1 Comparison Metrics

The scheduling approaches in this study are evaluated using the following metrics:

- **Execution Time** The execution time (Δtime) is calculated to measure the algorithm's performance. It is defined as the difference between the finish time and the begin time:

$$\Delta\text{time} = \text{FinishTime} - \text{BeginTime} \tag{4.4.1}$$

- **Makespan** The makespan, also referred to as schedule length, is defined in the context of task scheduling (see Definition 4).

- **Processors' Utilization Ratio (PU_{Ratio})**
  To define resource utilization (RU) measure, we introduce processor utilization ratio (PU_{Ratio}) as a metric used to evaluate how evenly computational tasks are distributed among the processors. It quantifies the difference in utilization between the most and least loaded processors, normalized by the total number of processors:

$$PU_{Ratio} = \frac{C_{\max} - \min(\text{Processor\_Load}_j)}{|P|} \tag{4.4.2}$$

  where:

  $C_{\max}$ is the makespan, i.e., the maximum load or finish time among all processors (See Definition 4).

  $\min(\text{Processor\_Load}_j)$ is the minimum workload assigned to any processor $p_j \in P$.

  $|P|$ is the total number of processors in the system.

  A lower PU_{Ratio} value indicates a more balanced workload distribution across processors, which is desirable for improving parallelism and reducing idle time.

4.4.2 Experimental Results

This section presents an illustrative case to evaluate the performance of scheduling in a heterogeneous computing environment.

The scenario involves a heterogeneous environment S shown in Table 4.1, where the same set of tasks $T = \{t_1, t_2, t_3, t_4, t_5\}$ is assigned to three interconnected processors $P = \{p_1, p_2, p_3\}$, each with different processing speeds. As a result, varying execution costs are observed for each task. In this initial setup, it is assumed that all tasks are independent and are assigned equal rank.

Processors	t_1	t_2	t_3	t_4	t_5
p_1	94	55	14	30	108
p_2	74	35	10	22	81
p_3	99	65	23	42	130

Figure 4.1: Tasks execution costs in heterogeneous environment

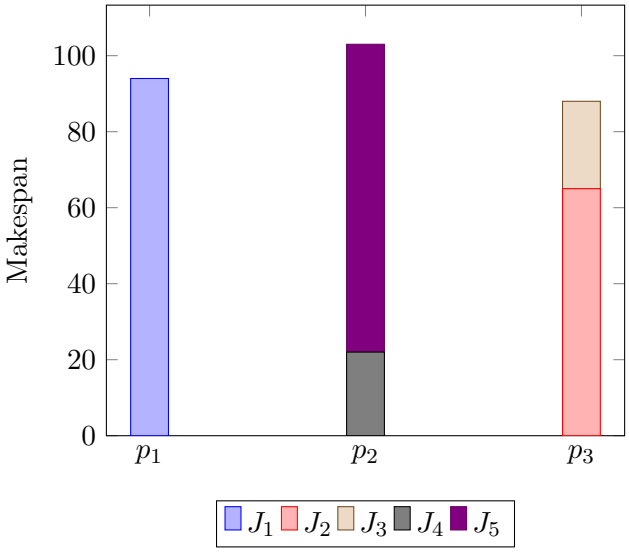

Figure 4.2: QoS guided Min-Min approach

A new version of Min-Min approach was proposed in[47], by He et al. to improve the original algorithm results. Effectively, by applying both algorithms on the same system, traditional Min-Min and QoS Guided Min-Min (see Figure 4.2), the makespan shows an enhancement of 8.85% from 113 to 103.

Another study, called the Static Task Graphs Stratification algorithm, was presented in [83]. This method primarily focuses on load balancing in multi-processor systems by stratifying and partitioning tasks. Initially, independent tasks are assigned to distinct levels (see Figure 4.3). Then, a static task group scheduling algorithm is applied to allocate these tasks across secondary processors.

During execution, tasks whose running times are unpredictable are dynamically allocated to available processors using a dynamic link algorithm.

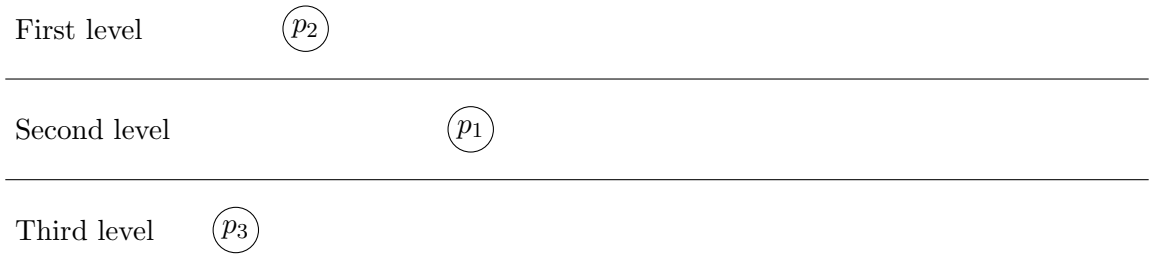

Figure 4.3: Graph stratification approach

Tasks $T_i,\ i = 1, \ldots,$ n are assigned to processors $p_j \in P,\ j = 1, \ldots, m$ in descending order of their execution time. Specifically, the unassigned task T_k that satisfies the condition in Equation 4.4.3 is selected for allocation, and the process continues until all tasks are mapped.

$$\text{TotalTime} + t_k > \text{AVG}_{\text{Time}} \tag{4.4.3}$$

where:

AVG_{Time} : is the average expected execution time across all processors, defined as $\sum_{i=1}^{n} E(T_i)/|P|$,

$P = \{p_1, p_2, \ldots, p_m\}$ is the set of available processors in the system,

TotalTime : is the cumulative execution time already assigned to processors $p \in P$, and p is not equal to some specific processor c which means $p \neq c$,

This formulation supports efficient and fair distribution of task load across heterogeneous processors.

In another experiment, the table 4.1 shows a performance comparison between DyTAg, Min-Min, and QoS Min-Min.

Algorithm	**Execution Time (ms)**	**Makespan**	**Resource Utilization (%)**
Min-Min	140	200	70%
QoS Min-Min	130	180	75%
DyTAg	**115**	**160**	**85%**

Table 4.1: Performance Comparison of Scheduling Algorithms

The proposed approach DyTAg outperforms others, by achieving a makespan of 160 compared to 200 for Min-Min and 180 for QoS Guided Min-Min.

Through this part of experiment section, our objective is to compare our approach to algorithms and techniques that share the same specifications, like priority-based criteria, makespan minimisation and heterogeneous computing systems involving.This process is done by comparing DyTAg results to Min-Min and QoS guided Min-Min. Aforementioned purposes above, we consider the following example described in (Table 4.2) and summarised in the following table:

4.4.3 Performance Analysis

For this first comparison (See Table 4.2), the results (See Figure 4.4) show comparatively better performance of the proposed approach, effectively for the given set of tasks DyTAg provide a makespan of 119 this result is reached by assigning t_3, t_6 to processor p_2 and t_2, t_4 to processor p_3 and the remaining tasks t_1, t_5, t_7 to the last processor p_1 . Min-Min gives a makespan of 140 while QoS Guided Min-Min and Max-Min give both a makespan

of 130, which results in a gain of 15% in this test with DyTAg. It is noted that complexities of DyTAg, Max-Min equal $O(n.m)$ and $O(n^2.m)$ respectively where N is the number of tasks, m number of processors and. This complexity disparity results to a better DyTAg's performance then other algorithms, effectively in this example the complexity of DyTAg = 833 while the complexity of Max-Min = 1029, as a result $\delta Time$ enhanced by 19.04% In the other hand, processors utilisation rate for our approach shows better results, PU_{Ratio} (DyTAg) = 2 while PU_Ratio (Min-Min) = 25, the obtained result shows that DyTAg has a better resources management with fair tasks distribution compared to the other heuristics.

Processors	t_1	t_2	t_3	t_4	t_5	t_6	t_7
p_1	77	56	23	29	9	40	31
p_2	98	90	54	50	22	65	76
p_3	82	70	40	43	17	51	45

Table 4.2: Tasks-set execution times on the processors p_1, p_2, and p_3.

DyTAg experiment aims mainly to focus on the performances disparities between our proposed technique and existing approaches in the field of scheduling, nevertheless also through these tests, we consider the importance of improving algorithm processing capabilities to give result in less time, especially in scheduling for heterogeneous multiprocessor systems (HMS), which every measure matters in the general process. This section consists of comparing our approach to traditional methods like Min-Min and QoS Guided Min-Min. Finally, presenting the improvements results relative to well-known and recent approaches. Every test is given with a real time context, and this is for the attention of giving the closest results to real situation system environment.

These experiments collectively emphasize the importance of improving algorithmic processing capabilities to produce efficient and scalable results, which is particularly crucial for scheduling in HMS. The proposed DyTAg approach demonstrates its effectiveness in handling independent task scheduling through dynamic programming, showcasing improved makespan and workload balancing. However, real-world distributed systems often present more complex scenarios, involving task dependencies, dynamic workloads, and additional constraints. This highlights the necessity of exploring alternative optimization techniques capable of addressing these challenges in more intricate scheduling contexts.

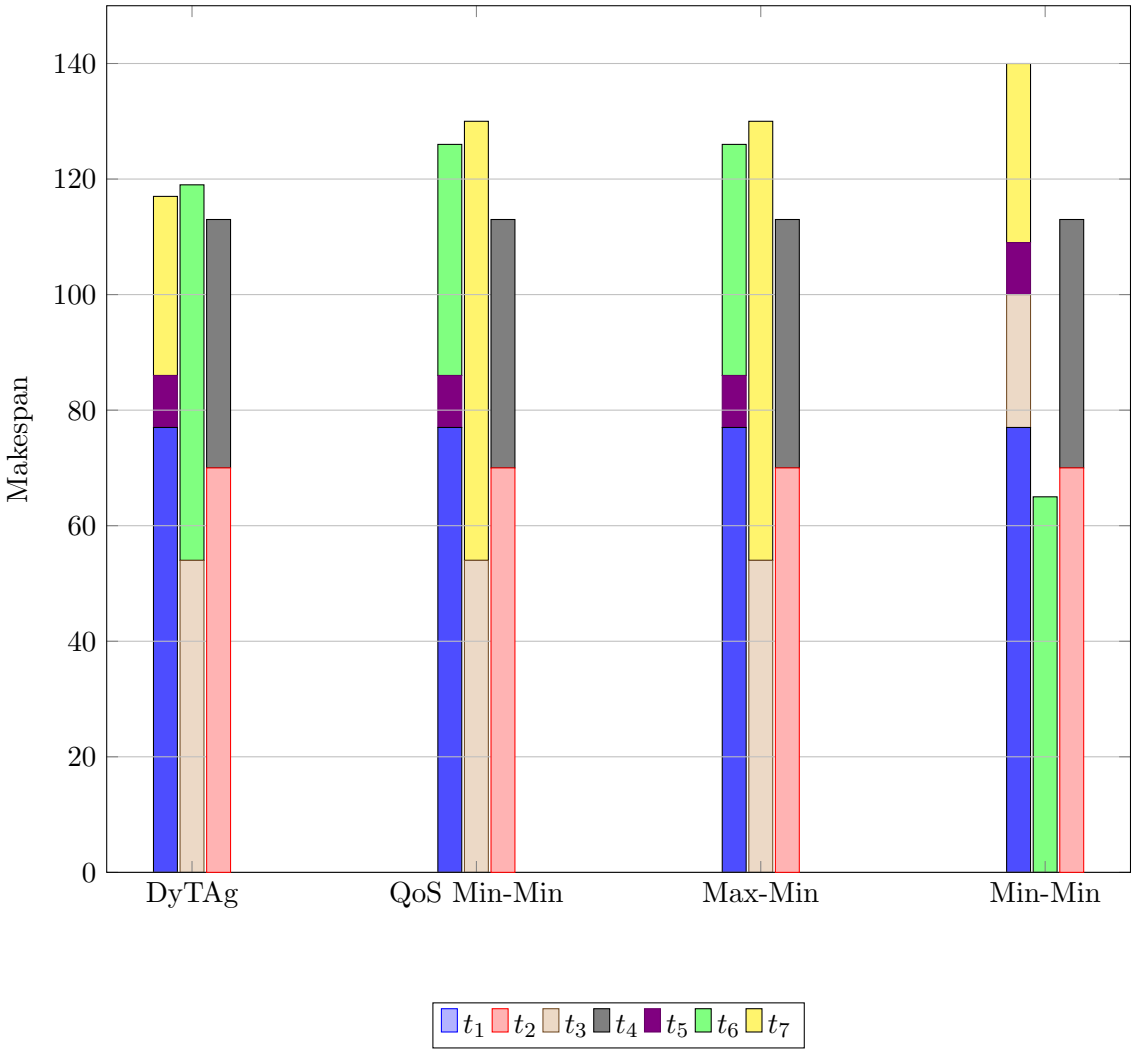

Figure 4.4: Min-Min, Max-Min, QoS guided Min-Min and DyTAg makespan comparison

Algorithm	**Strategy**	**Advantages**	**Limitations**	**Complexity**
Min-Min	Shortest task first	Reduces execution time	Leads to load imbalance in heterogeneous environments	$\mathcal{O}(n^2)$
Max-Min	Longest task first	Balances load among processors	May delay shorter tasks, increasing makespan	$\mathcal{O}(n^2)$
QoS-guided Min-Min	Priority-aware shortest task first	Improves response time for high-priority tasks, balances performance and QoS	Increased complexity due to priority handling; may delay low-priority tasks	$\mathcal{O}(n^2 + n \log n)$
HEFT	DAG-based ranking	Efficient for heterogeneous systems with dependencies	High time complexity for large DAGs	$\mathcal{O}(e+n \log n)$
CPOP	Critical path first	Reduces inter-task communication	Load imbalance for non-critical tasks	$\mathcal{O}(e + n^2)$
Genetic Algorithm (GA)	Evolutionary metaheuristic	Finds near-optimal solutions in complex spaces	Slow convergence and high computational cost	$\mathcal{O}(g \cdot p \cdot n)$
DLS	Dynamic task ranking	Effective for tasks with dependencies	May require task duplication	$\mathcal{O}(n \cdot p)$
Round-Robin	Equal time slicing	Fair and easy to implement	Poor for heterogeneous or priority-based systems	$\mathcal{O}(n)$

Table 4.3: Comparison of Scheduling Algorithms in Heterogeneous and Distributed Environments

Symbol: n: Number of tasks in the system, p: Number of processors, e: Number of edges in the task graph, g: Number of generations in Genetic Algorithm (GA).

4.5 Conclusion

Efficient task scheduling plays a vital role in optimizing performance in heterogeneous computing systems. Different algorithms have been proposed to balance execution time, resource utilization, and system adaptability. Table 4.3 provides a comparison of some widely used scheduling algorithms, highlighting their advantages, disadvantages, and computational complexity.

DyTAg introduces a dynamic programming-based scheduling method that adapts to system changes, reducing makespan and improving processor utilization. Future research will focus on integrating machine learning techniques to further enhance scheduling decisions.

The next chapter introduces an advanced co-scheduling strategy based on knapsack optimization, aiming to extend the capabilities of task allocation to dependent tasks and further enhance performance in dynamic and heterogeneous computing environments.

Chapter 5

Knapsack-based Algorithm Co-Scheduling Task Allocation (KaCoSTA)

Efficient scheduling in heterogeneous multiprocessor systems is a critical concern in distributed computing, where diverse processing capabilities and dynamic workloads present significant challenges. Traditional task scheduling approaches often operate sequentially, assigning tasks to one processor at a time. which can lead to sub-optimal resource utilization and unbalanced workloads. To address these limitations, co-scheduling strategies have emerged as powerful alternatives, aiming to allocate tasks across multiple processors simultaneously for improved performance.

This chapter focuses on the evaluation of the Knapsack-based Co-Scheduling Algorithm for Task Allocation (KaCoSTA), a novel scheduling technique that integrates dynamic programming with knapsack optimization principles [84]. KaCoSTA is specifically designed to enhance global resource utilization and minimize execution time by considering task priorities, resource constraints, and system states during allocation.

5.1 Multiple Knapsack Problem (MKP)

Given m knapsacks with capacities c_i $(i = 1, \ldots, m)$ and n items with profits p_j and weights w_j $(j = 1, \ldots, n)$, the goal in the MKP is to select m disjoint subsets of items such that the total profit of the selected items is maximized, and each subset is assigned to a knapsack whose capacity is no less than the total weight of the assigned items.

Let x_{ij} be a binary variable that takes the value 1 if and only if item j is assigned to

knapsack i. The MKP can be formulated as follows:

$$\text{Maximize} \quad \sum_{i=1}^{m}\sum_{j=1}^{n} p_j x_{ij} \tag{5.1.1}$$

$$\text{Subject to} \quad \sum_{j=1}^{n} w_j x_{ij} \leq c_i, \quad \forall i = 1, \ldots, m \tag{5.1.2}$$

$$\sum_{i=1}^{m} x_{ij} \leq 1, \quad \forall j = 1, \ldots, n \tag{5.1.3}$$

$$x_{ij} \in \{0, 1\}, \quad \forall i = 1, \ldots, m, \quad \forall j = 1, \ldots, n \tag{5.1.4}$$

Equation (5.1.1) represents the objective function, which maximizes the total profit of the selected items. Constraint (5.1.2) ensures that the total weight of items assigned to each knapsack does not exceed its capacity. Constraint (5.1.3) guarantees that each item is assigned to at most one knapsack. Finally, constraint (5.1.4) enforces the binary nature of the decision variables, where each item is either assigned to a knapsack or not.

MKP is classified as strongly NP-hard, which can be demonstrated through a reduction from the well-known 3-partition problem (see, e.g., [85]). Due to its computational complexity, MKP has been extensively studied, leading to the development of exact, approximation, and heuristic-based solution approaches.

5.1.1 Exact Solution Methods

Several exact algorithms have been proposed to tackle MKP. A Branch-and-Bound (B&B) approach was introduced by Fukunaga and Korf [86], utilizing dominance criteria to effectively prune nodes during item-to-knapsack assignments. Later, Fukunaga [87] improved upon this approach by incorporating advanced dominance rules and symmetry-breaking techniques, extending the classic B&B framework originally developed by Martello and Toth [88]. These methods have been tested on problem instances with up to 10 knapsacks and 100 items, demonstrating their feasibility for small- to medium-scale MKP problems.

Sitarz [89] introduced an exact solution technique based on multiple criteria Dynamic Programming (DP). However, computational results indicated that direct solution approaches using commercial solvers outperformed this method in terms of execution time. Hickman and Easton [90] proposed a novel class of valid inequalities derived from merging lower-dimensional constraints within the MKP polyhedral space. They further established conditions under which these inequalities are facet-defining.

Another exact approach was presented by Dell'Amico et al. [91], who employed a pseudo-polynomial arc-flow model, inspired by Valério de Carvalho [92], to represent item-

to-knapsack assignments as paths in a graph. This model was further enhanced through techniques such as partial B&B, primal decomposition, Benders cuts, and a graph reduction strategy, achieving effective results on instances with up to 500 items and 50 knapsacks, as well as 300 items and 150 knapsacks. Additionally, Detti [93] recently introduced a new upper bound for the MKP, further refining solution quality.

5.1.2 Approximation Algorithms

Unlike the classical 0-1 Knapsack Problem (KP01), MKP does not admit a Fully Polynomial Time Approximation Scheme (FPTAS) unless P = NP. Chekuri and Khanna [94] developed a Polynomial Time Approximation Scheme (PTAS) with a running time of $n^{O(\log(1/\epsilon)/\epsilon^8)}$. Jansen [95] later improved upon this by proposing an Efficient Polynomial Time Approximation Scheme (EPTAS) with a complexity of $2^{O(\log(1/\epsilon)/\epsilon^5)} \cdot \text{poly}(n) + O(m)$.

Wang and Xing [96] introduced an iterative approximation algorithm that sequentially fills knapsacks in nondecreasing order of their capacities. For each knapsack, the algorithm selects the most profitable subset of items using an exact KP01 procedure, with worst-case analysis provided for cases where $m = 2$ and $m = 3$. Khutoretskii et al. [97] designed a 0.5-approximation algorithm with a computational complexity of $O(mn)$, excluding preprocessing steps such as sorting items and knapsacks based on lexicographic ordering.

5.1.3 Heuristic Approaches

Due to the computational difficulty of solving MKP optimally for large-scale instances, heuristic methods have been widely explored. Various heuristic techniques include:

- *Population-based approaches*: Shah-Hosseini [98] applied evolutionary strategies to solve MKP efficiently.
- *Recursive constructive methods*: Lalami et al. [85] proposed a recursive approach that incrementally builds feasible solutions.
- *Swarm intelligence techniques*: Several studies have leveraged bio-inspired algorithms for MKP, such as artificial bee colony optimization [66, 99] and artificial fish swarm optimization [100].

These heuristic and metaheuristic methods offer practical solutions for large-scale MKP instances, balancing computational efficiency with solution quality.

5.2 Knapsack Based Scheduling Model

Dynamic programming methods are widely used for solving discrete optimization problems, particularly because many such problems are classified as NP-hard. Among these, the knapsack problem (see Definition 9) and dynamic programming is one of the most adopted strategies in the literature. In fact, many problems in the discrete optimization domain (like scheduling) share characteristics similar to those of the classical knapsack problem [101].

5.2.1 Problem Definition

In the context of task scheduling for heterogeneous multiprocessor systems, the problem can be effectively modeled using principles from combinatorial optimization. Specifically, this section introduces a knapsack-based scheduling model that formulates the task allocation challenge as a resource constrained optimization problem. Inspired by the classical knapsack problem, this model aims to assign tasks each with associated execution costs and priorities to available processors such that the overall resource utilization is optimized while adhering to system constraints. This formulation enables more structured and adaptive scheduling decisions, particularly in environments with dynamic workloads and diverse processing capabilities.

Definition 9 (Knapsack Algorithm)**.** *The Knapsack Algorithm is a combinatorial optimization technique used to solve problems where a set of items, each with a specific weight and value, must be selected to maximize the total value without exceeding a given weight (or capacity) constraint. It is inspired by the practical problem of selecting the most valuable items that can fit into a knapsack of limited capacity [88]*

Definition 10 (Knapsack Scheduling System)**.** *We define the knapsack-based scheduling system in heterogeneous multiprocessor systems (HMS) as the quintuple* $S = (T, E, K, P, MT)$*, where it slightly differs from the previously defined DyTAg model by introducing task precedence constraints and priority measures, allowing more complex scheduling decisions in dynamic environments:*

$T = \{t_1, \ldots, t_n\}$ *is the set of available non-preemptive tasks,*

$K = \{k_1, \ldots, k_m\}$ *is the set of heterogeneous processors,*

E *is the task execution cost matrix, where* $E[i][j]$ *represents the processing time of task* t_i *on processor* p_j*,*

$P = \{r_1, \ldots, r_n\}$ *is the set of task priorities (or weights),*

MT *denotes the makespan threshold (see Definition 11).*

The system is also represented as a directed acyclic graph (DAG), $G = (T, E_d)$, where $T = \{t_1, \ldots, t_n\}$ denotes the set of tasks, and E_d denotes the set of directed edges representing inter-task data dependencies [102]. If there exists a directed edge $(t_y, t_x) \in E_d$, then task t_x is said to be a child of task t_y and thus, t_x cannot begin execution until t_y has completed. In other terms, t_y is a predecessor of t_x $t_y \in Pred(t_x)$.

Definition 11. *$Makespan_{Threshold}$ (MT), is an estimated value, which is defined as the average of the processing times of all jobs across processors. MT in this work is obtained by processing the following formula 5.2.1.*

$$MT = \frac{1}{|K|^2} \sum_{j=1}^{m} \sum_{i=1}^{n} E[i][j] \tag{5.2.1}$$

where:

$T = \{t_1, t_2, \ldots, t_n\}$ *is the set of tasks,*

$K = \{k_1, k_2, \ldots, k_m\}$ *is the set of processors,*

$E[i][j]$ *: Execution matrix, is the execution cost of task t_i on processor k_j.*

n *: Number of tasks on the queue.*

m *: Number of processors on the system.*

Assume we have n items, their costs $c_i > 0$ and weights $w_i > 0, i = 1, 2, \cdots n$, and a knapsack carrying capacity R. In addition, suppose that $\sum w_i > R$, and $0 < w_i \leq R$.

The purpose is to fill the knapsack with a set of items such that the total value (or priority) of the selected items is maximized without exceeding the capacity constraint. As shown in Eq. (5.2.2), the problem is formulated as a discrete mathematical model using boolean decision variables $x_i,\ i = 1, \ldots, n$, where:

$$\begin{aligned} &\text{Max F} = \textstyle\sum_{i=1}^{n} c_i \cdot x_i, \\ &s.c \begin{cases} \sum_{i=1}^{n} w_i \cdot x_i \leq R \\ x_i \in \{0, 1\} \end{cases} \end{aligned} \tag{5.2.2}$$

By analogy with model 5.2.2, we consider a processor K_m with a positive estimated makespan threshold $Makespan_{Threshold}$ (MT). Let $T = t_1, \ldots, t_n$ be the set of tasks, each associated with a positive processing cost (execution time) $e_1, \ldots, e_n$ and a corresponding set of positive priority values $p_1, \ldots, p_n$.

The purpose of our approach is to select the best subset T_k of tasks (see Definition 12), which compromises between the priorities $p_i/t_i \in T_k$ and the processing times $e_i/t_i \in T_k$. In addition, the sum of the processing times $e_i/t_i \in T_k$ should not exceed $Makespan_{Threshold}$.

5.2.2 Mathematical Formulations

This subsection outlines the knapsack-based algorithm mathematical formulations. These formulations typically involve defining an objective function to maximize total value or priority under various constraints as shown in model (See definition 12). This mathematical foundation provide a structured framework for addressing resource-constrained scheduling and allocation problems in heterogeneous computing contexts.

Definition 12. *Scheduling Mathematical Model*
Let T be a set of tasks to schedule and T_k its tasks' subset which is assigned to the processor k_j. T_k is the best subset $T_k \subset T$ of tasks, that verify the linear program given below:

$$\begin{array}{l} Max\ Z = \sum_{i=1}^{n} P_i \cdot x_i \\ s.c \begin{cases} \sum_{i=1}^{n} E[i][j] \cdot x_i \leq Makespan_{Threshold} \\ x_i \in \{0,1\} \end{cases} \end{array} \tag{5.2.3}$$

where:

P_i: the priority of the task t_i (See Definition 13),

$E[i][j]$: the execution cost of the task t_i on the processor j,

$Makespan_{Threshold}$ (See Definition 11): the estimated makespan threshold of the system,

x_i: the corresponding variable of the task t_i, where:

$$x_i = \begin{cases} 1 \text{ if } t_i \text{ is assigned to the processor} \\ 0 \text{ otherwise} \end{cases}$$

Definition 13. *Task Priority Calculation [103]*
The Upward Technique is applied for task prioritization while satisfying the precedence constraint [81]. The priority of each task is calculated by Eq. 5.2.4

$$P_i = avgEx_i + \max_{j \epsilon succ(t_i)} CC(i,j) + P_j \tag{5.2.4}$$

where:

$avgEx_i$: average execution cost of t_i on all processors

$succ(t_i)$: represent all successors of t_i on the scheduling system

$CC(i,j)$: Communication Cost between t_i and t_j

It is noted that the communication cost is considered zero when two tasks, t_i and t_j, are assigned to the same processor. Otherwise, data must be transferred from the processor executing task t_i to the processor assigned to task t_j, incurring a non-zero communication cost [104].

Based on model 12, our approach is a scheduling method that performs in a heterogeneous computing system. By means of DP, this approach combines between makespan (See Definition 14) minimisation and optimization of resources utilization.

Definition 14. *Makespan, C_{max}*
The total schedule length also defined as completion time, calculated in Eq. 5.2.5, is the maximum Finish Time (FT) of the exit task. [103]

$$makespan, C_{max} = Max_i(FT(exit\_task)) \tag{5.2.5}$$

Basically, dynamic programming (DP) is used for solving complex problems by breaking them down into simpler sub-problems. Particularly, DP is effective for tackling problems from combinatorial optimization field, whose the solution can be constructed from the solutions of overlapping subproblems. In the context of task scheduling, DP helps in determining the best assignment of tasks to processors by evaluating systematically all possible task allocations and selecting which minimizes makespan and maximizes RU.

Our purpose, through DP techniques, is to maximize the total priority weight without exceeding the MT for the execution times of the tasks chosen. We interpreted our problem in a formal mathematical way as defined in Eq. (5.2.3).

The utilization of DP Recurrence, allows the leading of the best capacity $DP(k)$, which is the DP table entry for capacity k. For each task t_i and capacity k, $DP(k)$ is evaluated through Formula 5.2.6

$$DP[k] = \max(DP[k], DP[k - E[i][j]] + P[i][j]) \tag{5.2.6}$$

The boolean matrix $A(n \times m)$ gives the assignments' information of a task to a processor. In other words, a term of A is defined as follows:

$$A[i][j] = \begin{cases} 1 & \text{if task } t_i \text{ is assigned to processor } k_j \\ 0 & \text{otherwise} \end{cases} \tag{5.2.7}$$

Therefore, at each recursion, if $DP[k] > DP[k - E[i][j]]$ then $A[i][j] = 1$ (the matrix A is updated).

As given in Formula 5.2.8, the makespan MP is calculated, by using the last evaluation

of A as the maximum completion time across all processors.

$$MP = \max_{j=1}^{m} \sum_{i=1}^{n} E[i][j] \cdot A[i][j] \tag{5.2.8}$$

After computing A then MP through the system, the resource utilization in the HCS is calculated by formula 5.2.9

$$\text{RU} = 100 - \frac{MP - Min(ProcessorUsage)}{MP} \tag{5.2.9}$$

where:

$Min(ProcessorUsage)$ is the least utilized processor in the scheduling system.

5.2.3 Implementation of the Approach

The method process in this subsection details the practical implementation of the proposed approach. It describes the sequence of steps followed to operationalize the scheduling model, including data preparation, algorithm initialization, and the execution of decision-making procedures. The method process steps are outlined as follows:

1. **Tasks Partitioning Dependency**
   The first step consists of addressing tasks precedence constraint by partitioning the tasks, the independent tasks are grouped to distinct partition levels from DAG G(T, E_d) (Figure 5.1), and the result is defined in Figure 5.2.
   The following steps are applied to each partition

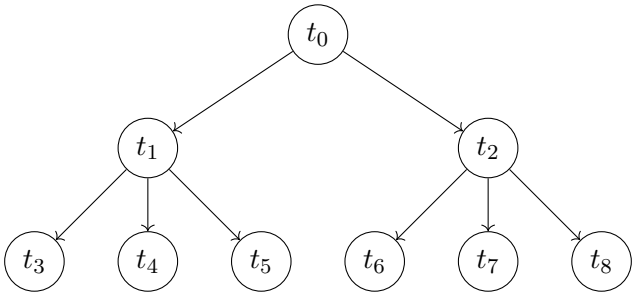

Figure 5.1: Example of DAG (Task Graph)

2. **Estimation of the** $Makespan_{Threshold}$ (See Definition 11)

3. **Modulation and decomposition of the MT**
   $Makespan_{Threshold}$ modulation takes part of the DP process (See Eq 5.2.6), thus MT value is decomposed. For instance, in the case of $MT = 500$, the modulation process of

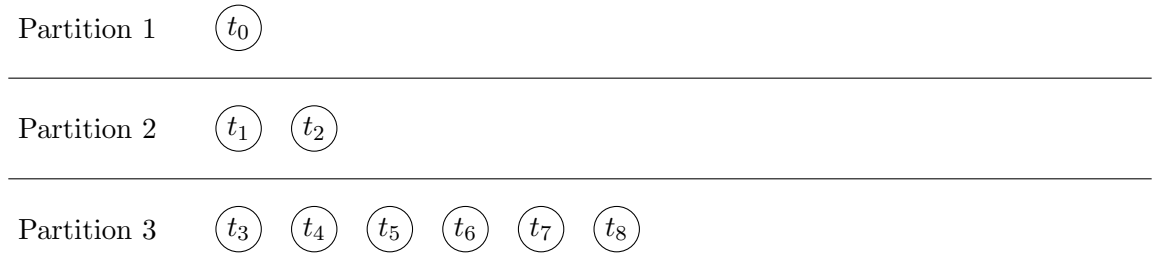

Figure 5.2: Tasks Partitioning Dependency (Step 1)

$Makespan_{Threshold}$ provides the following modulation vector V_l defined as:
$V_l = 1, (1,2), (1,2,3), ...(1, ...500)$, where the MT is broken down into units, each unit called Modulated Makespan Threshold (MMT_l), and l represents the unit number.

4. **Adding tasks process (Pre-scheduling phase)**
   Adding the tasks one by one according to their arrival times in the system, then browse the MTM_l modulation vector and process step 3 to each unit.

5. **Fill the dynamic table**
   Based on DP, completing the table (Tab) by dynamically adding the tasks by processing the following formula:

$$Tab[i, E[i-1][j]] = Max[Tab(i-1, E[i-1][j]), \\ Tab(i-1, E[i][j] - E[i][j]) + P[i]] \quad (5.2.10)$$

6. **Subset and Assignment Vector Construction**
   If $Tab[i, E[i][j]] = Tab[i-1, E[i][j]]$
   $\Rightarrow$ Task not assigned

$$A[i][j] = \begin{cases} 1 \text{ if } t_i \text{ is assigned to the processor } k_j \\ 0 \text{ otherwise} \end{cases}$$

   where:

   A : Assignment Matrix

7. **Processing all the tasks in the system** The process is repeated with none selected tasks for the remain processors, Unassigned tasks will be handled by the same process for the next node until all tasks will be treated, then assigned tasks are executed according to their priority.

5.3 Sequential Task Allocation Strategies

Scheduling in HMS requires efficient algorithms capable of balancing computational loads across multiple processors while minimizing execution time. As a foundational step toward achieving optimal scheduling solutions, the KReSTA (Knapsack-based Recursive Scheduling Task Allocation) and KISTA (Knapsack-based Iterative Scheduling Task Allocation) algorithms were developed to allocate tasks sequentially across processors.

Both KReSTA and KISTA operate by scheduling tasks on one processor at a time, ensuring that each processor receives an optimal task assignment before moving on to the next. KReSTA follows a recursive approach, breaking down the scheduling problem into smaller subproblems and solving them recursively to optimize resource allocation. In contrast, KISTA adopts an iterative approach, iteratively refining task assignments in a step-by-step manner to achieve efficient scheduling.

By employing these methodologies, KReSTA and KISTA serve as the initial steps toward designing scalable scheduling solutions for complex heterogeneous computing environments. Their sequential task allocation strategy provides a structured foundation for more advanced algorithms that handle dynamic and parallel task scheduling across multiple processors.

5.3.1 Knapsack based Recursive algorithm for Scheduling Tasks Allocation (KReSTA)

The first algorithm represents the first version of the approach.

Recursion is one of the popular problem-solving approaches in data structure and algorithms. Even some problem-solving approaches are totally based on recursion for example: decrease and conquer, divide and conquer, DFS traversal of tree and graphs, backtracking, top-down approach of DP and many others. Thus, time complexity analysis of recursion is critical to understand these approaches and improving our code's efficiency.

In the Figure 5.3 bellow, we present the recursive version.

5.3.2 Knapsack-based Iterative Scheduling Task Allocation (KISTA)

The algorithm 3 mentioned bellow is featured by the introduction of iterative approach method to enhance time complexity.

The second algorithm uses iterative approach and requires $\mathcal{O}(N \times MT) \Rightarrow \mathcal{O}(N^2)$ time complexity, which is better than algorithm 2

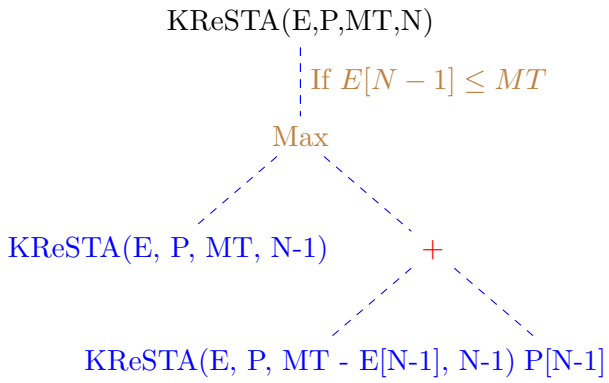

Figure 5.3: Knapsack Recursive Scheduling Tasks Allocation (KReSTA)

First, we will estimate the difference between the two proposed algorithm 2 and 3 to highlight the optimization of the approach which intuitively brings an added value and thus a relevant aspect to the scheduling system overall. Scheduling algorithms performances are estimated with Δtime measure (See Eq (4.4.1)), consequently, according to tasks-set processed, trivial difference can be noticed on Figure 5.4 bellow, Indeed, KReSTA algorithm shows exponential comportment while increasing number of tasks, whereas KISTA increases proportionally, which explains time complexity differences, we can observe for KReSTA with tasks-set : N=960, the algorithm needs 400.13 milliseconds to process, while with tasks-set : N=1920 the algorithm displayed the result in 848.57 milliseconds, hence, an increase of 112.07%, in the other side, KISTA with the same tasks-set : N=240, N=480 shows respectively 281.98 and 566.73 milliseconds which results in 100.98% increase, as a consequence, KISTA increases proportionally with N while KReSTA does not, for instance for N = 1920, KISTA is 41.63% better than KReSTA, as a result traduces a better performance compared to KReSTA in the first experiments as shown in Figure 5.4.

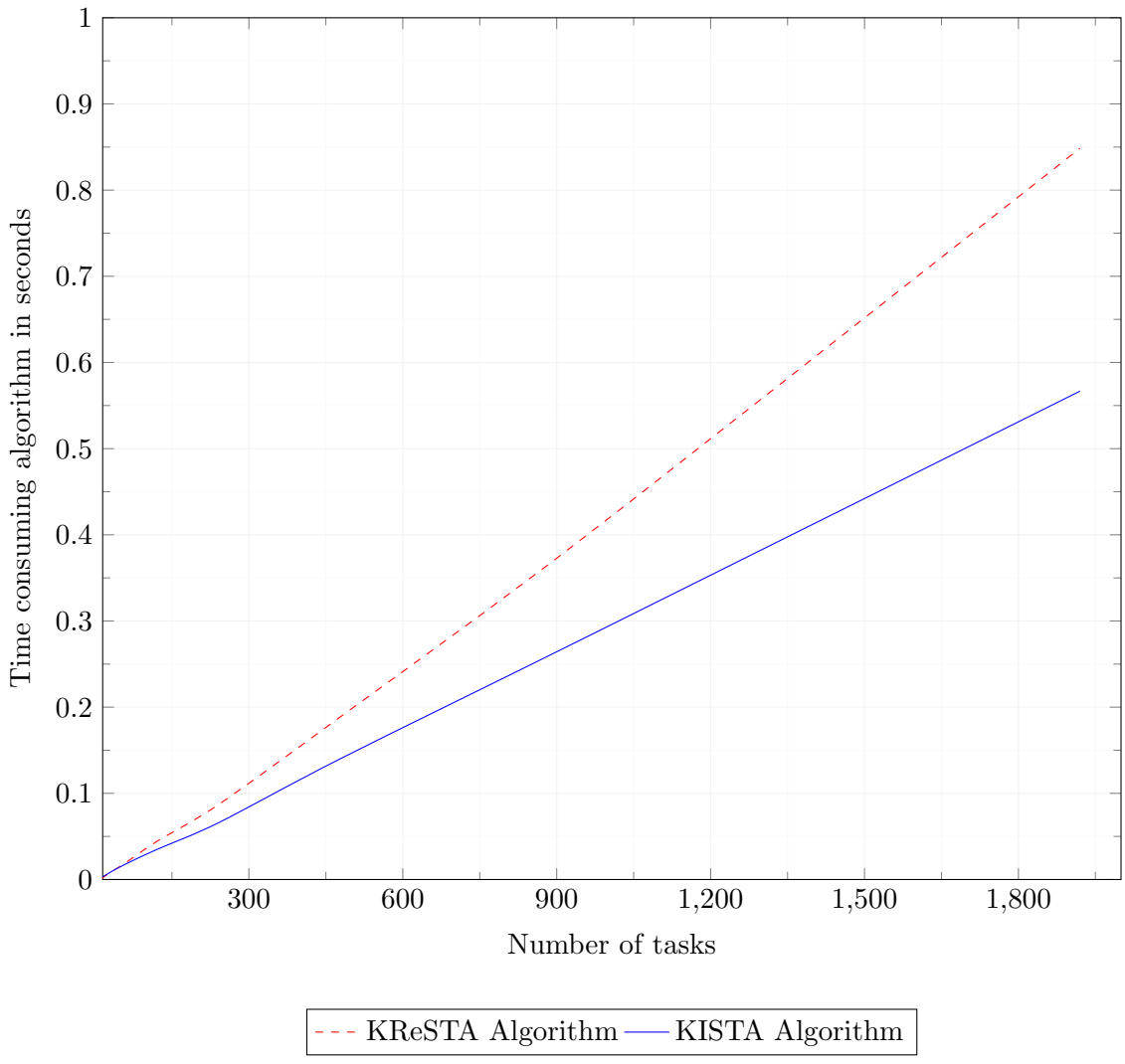

Figure 5.4: Comparison of KISTA and the improved KReSTA algorithm on total elapsed time with the same number of tasks

Algorithm 2 KReSTA: Knapsack-based Recursive Scheduling Algorithm for Task Allocation

Input: $T = \{t_1, \ldots, t_n\}$: Set of tasks, $E[i][j]$: Execution cost of task t_i on processor j, $P[i]$: Priority weight of task t_i, $PList$: List of processors sorted by efficiency, MT: Makespan Threshold

Output: $A[i][j] = 1$ if task t_i is assigned to processor j, else 0

Initialize $A[i][j] \leftarrow 0$ for all i, j
Initialize $used[i] \leftarrow$ false for all i
for all processor $j \in PList$ **do**
    Initialize memoization table $DP[k] \leftarrow -1$ for $k = 0$ to MT
    function DP(k)
        if $k = 0$ **then Return** 0
        end if
        if $DP[k] \neq -1$ **then Return** $DP[k]$
        end if
        $max\_val \leftarrow 0$
        for $i = 1$ to n **do**
            if not $used[i]$ **and** $E[i][j] \leq k$ **then**
                $val \leftarrow \text{DP}(k - E[i][j]) + P[i]$
                $max\_val \leftarrow \max(max\_val, val)$
            end if
        end for
        $DP[k] \leftarrow max\_val$
        Return $DP[k]$
    end function
    DP(MT) ▷ Compute optimal priority sum for processor j
    $k \leftarrow MT$
    while $k > 0$ **do**
        for $i = 1$ to n **do**
            if not $used[i]$ **and** $E[i][j] \leq k$ **then**
                if $DP[k] = DP[k - E[i][j]] + P[i]$ **then**
                    $A[i][j] \leftarrow 1$ ▷ Assign task t_i to processor k_j
                    $used[i] \leftarrow$ true ▷ Mark task t_i as used
                    $k \leftarrow k - E[i][j]$
                    break
                end if
            end if
        end for
        $k \leftarrow k - 1$
    end while
end for
Return A

Algorithm 3 KISTA (Knapsack-based Iterative Scheduling Task Allocation)

Input: $T = \{t_1, t_2, \ldots, t_n\}$: Set of tasks, $E[i][j]$: Execution cost of task t_i on processor j, $P[i]$: Priority weight of task t_i, $PList$: List of processors sorted by efficiency, MT: Makespan threshold

Output: $A[i][j] = 1$ if task t_i is assigned to processor j, else 0

Initialize $A[i][j] \leftarrow 0$ for all i, j
Initialize $used[i] \leftarrow$ false for all i
for all processor $j \in PList$ **do**
    Initialize DP table $tab[0 \ldots n][0 \ldots MT] \leftarrow 0$
    for $i = 1$ to n **do**
        for $mt = 0$ to MT **do**
            if $E[i-1][j] \leq mt$ **and not** $used[i-1]$ **then**
                $tab[i][mt] \leftarrow \max(P[i-1] + tab[i-1][mt - E[i-1][j]],\ tab[i-1][mt])$
            else
                $tab[i][mt] \leftarrow tab[i-1][mt]$
            end if
        end for
    end for
    $Res \leftarrow tab[n][MT]$
    $mt \leftarrow MT$
    for $i = n$ to 1 **step** -1 **do**
        if $Res \leq 0$ **then**
            break
        end if
        if $Res = tab[i-1][mt]$ **then**
            continue
        else
            $A[i-1][j] \leftarrow 1$ ▷ Assign task t_i to processor k_j
            $used[i-1] \leftarrow$ true ▷ Mark task t_i as used
            $Res \leftarrow Res - P[i-1]$
            $mt \leftarrow mt - E[i-1][j]$
        end if
    end for
end for
Return A

5.4 Global Task Allocation Strategy

The final step consists of taking into consideration the different processors in the system, which implies managing all tasks to represent the heterogeneous computing environment's behavior.

5.4.1 Assumptions

In our approach we suppose that:

- All tasks are non-preemptive, meaning once a task starts execution on a processor, it runs to completion.
- Task execution costs are known and deterministic.
- Processors have different capacities and execution speeds, modeled by the execution costs matrix E.
- MakespanThreshold (MT) is estimated based on average processing costs and may be adjusted dynamically.

5.4.2 Knapsack-based Algorithm Co-Scheduling Task Allocation (KaCoSTA)

While scheduling approaches, such as KReSTA and KISTA, allocate tasks sequentially by considering one processor at a time, achieving optimal performance in heterogeneous computing systems requires a more holistic strategy. To address this limitation, we propose the Knapsack-based Algorithm Co-Scheduling Task Allocation (KaCoSTA) algorithm, which simultaneously considers all available processors during scheduling. Unlike sequential strategies, KaCoSTA applies a global task allocation approach, where task assignments are optimized collectively across multiple processors rather than independently.

KaCoSTA is designed to minimize makespan while maximizing resource utilization by leveraging knapsack optimization principles and dynamic programming techniques. The algorithm efficiently distributes tasks across heterogeneous processors, dynamically adapting to system constraints and workload variations. By evaluating all processors together, KaCoSTA ensures an optimal balance of computational load, reducing idle time and improving execution efficiency in high-performance computing environments.

Algorithm 4 KaCoSTA (Knapsack-based Algorithm for Co-Scheduling Task Allocation)

Input: $T = \{t_1, t_2, \ldots, t_n\}$: Set of tasks, $K = \{k_1, k_2, \ldots, k_m\}$: Set of processors, $E[i][j]$: Execution time of task t_i on processor k_j, $P[i]$: Priority of task t_i, MT: Makespan Threshold

Output: $A[i][j]$: Assignment matrix, where $A[i][j] = 1$ if task t_i is assigned to processor k_j, MP: Final makespan

Sort processors K in ascending order of performance
Initialize $A[i][j] \leftarrow 0$ for all i, j
Initialize $L[j] \leftarrow 0$ for all j ▷ Tracks load on each processor
Initialize $used[i] \leftarrow$ **false** for all $t_i \in T$
for each processor $k_j \in K$ **do**
    Initialize DP table $Memo[k] \leftarrow -1$ for $k = 0$ to MT
    function DP(k)
        if $k = 0$ **then Return** 0
        end if
        if $Memo[k] \neq -1$ **then Return** $Memo[k]$
        end if
        $max\_value \leftarrow 0$
        for each task $t_i \in T$ where $used[i] =$ **false do**
            if $E[i][j] \leq k$ **then**
                $value \leftarrow \text{DP}(k - E[i][j]) + P[i]$
                $max\_value \leftarrow \max(max\_value, value)$
            end if
        end for
        $Memo[k] \leftarrow max\_value$
        Return max_value
    end function
    for each task $t_i \in T$ where $used[i] =$ **false do**
        if $E[i][j] \leq MT$ **then**
            for $k = MT$ **down to** $E[i][j]$ **do**
                if $\text{DP}(k) > \text{DP}(k - E[i][j]) + P[i]$ **then**
                    $A[i][j] \leftarrow 1$ ▷ Assign task t_i to processor k_j
                    $used[i] \leftarrow$ **true** ▷ Mark task t_i as used
                    $L[j] \leftarrow L[j] + E[i][j]$ ▷ Update load of processor k_j
                    break
                end if
            end for
        end if
    end for
end for
$MP \leftarrow \max_j L[j]$
Return A, MP

Algorithm

In this section, the KaCoSTA algorithm (see Algorithm 4) is applied to the entire system within a heterogeneous environment, where processors have different processing capacities. First, the Makespan Threshold (MT) is calculated. Next, the processors are sorted in ascending order based on their performance capacities. The KaCoSTA algorithm is then applied. Any tasks that remain unassigned, indicated by $A[i][j] = 0$ for all $i = 1, \ldots, N$, are queued for processing in the next iteration. This process continues until all tasks are successfully assigned to a processor during the pre-scheduling phase (See Figure 5.5).

Complexity Analysis

The complexity of the KaCoSTA depends on tasks number n and processors number m, The algorithm complexity can be broken down into several components based on its steps.

Proposition 1 (Initialization of Makespan Threshold (MT))**.** *The time complexity for calculating the Makespan Threshold (MT) is* $O(n \cdot m)$.

Proof. Calculating the average execution costs involves iterating over the execution cost matrix E, which has dimensions $n \times m$.

$$\text{MT} = \frac{1}{|K|^2} \sum_{i=1}^{n} \sum_{j=1}^{m} E[i][j]$$

Since we need to sum up all the elements in the matrix, the time complexity is:

$$O(n \cdot m)$$

Thus, the time complexity for calculating MT is $O(n \cdot m)$. □

Proposition 2 (Sorting Processors)**.** *The time complexity for sorting the processors is* $O(m \log m)$.

Proof. Sorting the processors by their processing performances involves using a sorting algorithm such as quicksort or mergesort, both of which have a time complexity of $O(m \log m)$.

$$\text{Time Complexity} = O(m \log m)$$

Thus, the time complexity for sorting the processors is $O(m \log m)$. □

Proposition 3 (Dynamic Programming with Memoization)**.** *The time complexity for solving the knapsack problem using dynamic programming with memoization is* $O(m \cdot MT \cdot n)$.

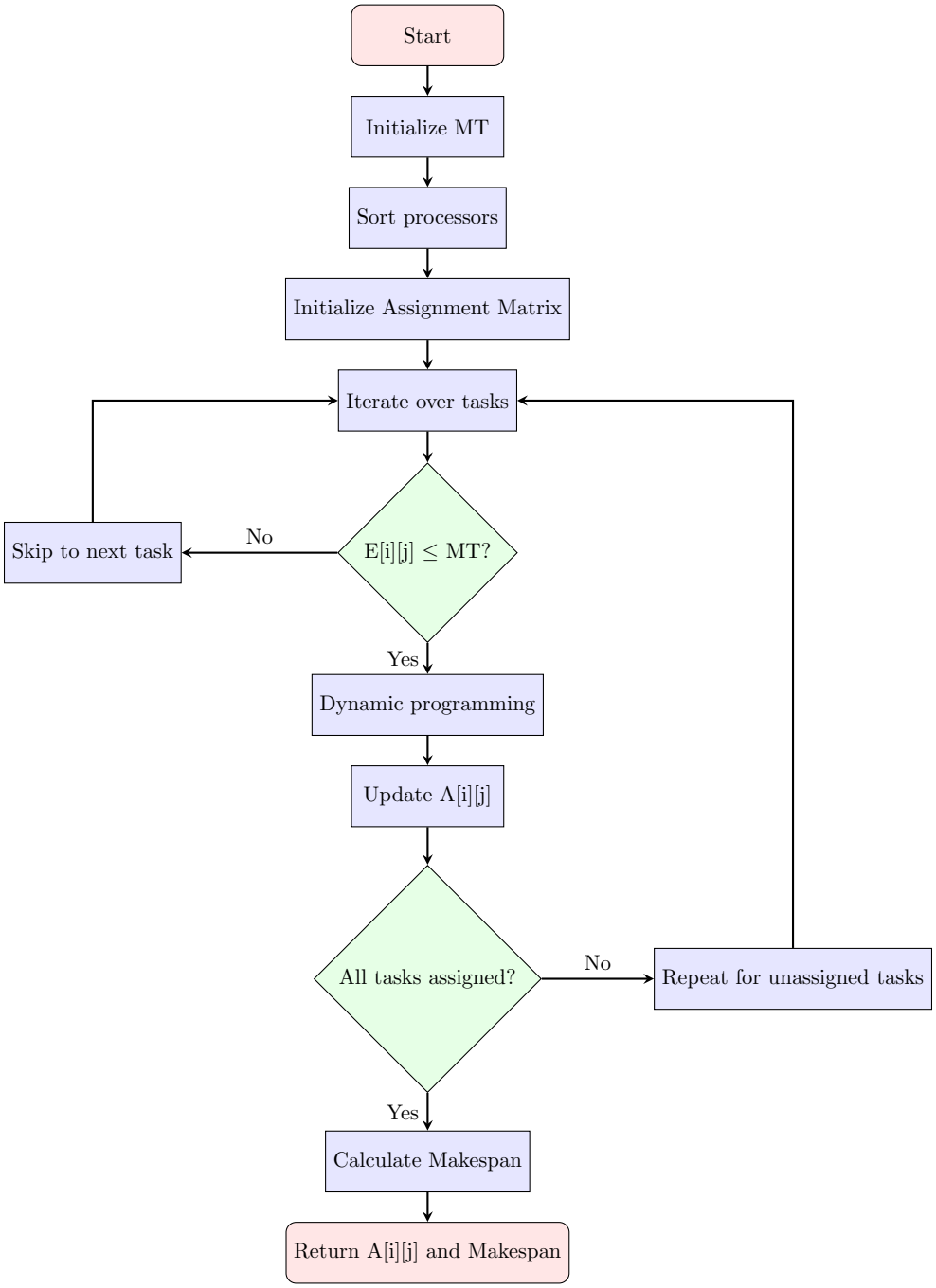

Figure 5.5: KaCoSTA Flowchart

Proof. The dynamic programming approach involves filling a table DP for each processor k_j up to the Makespan Threshold (MT). For each task t_i, we update the table from MT down to the execution cost $E[i][j]$. The nested loops iterate over all processors, all tasks, and all capacities up to MT.

$$TimeComplexity = O(m) \times O(MT) \times O(n) = O(m \cdot MT \cdot n)$$

Thus, the time complexity for solving the knapsack problem using dynamic programming with memoization is $O(m \cdot MT \cdot n)$. □

Proposition 4 (Assignment and Update)**.** *The time complexity for the assignment and update step is* $O(n \cdot m)$*.*

Proof. For each task t_i, we need to update its assignment to a processor p_j and update the available capacities. This involves iterating over all tasks and all processors.

$$\text{Time Complexity} = O(n) \times O(m) = O(n \cdot m)$$

Thus, the time complexity for the assignment and update step is $O(n \cdot m)$. □

Proposition 5 (Overall Complexity)**.** *The total time complexity for the KaCoSTA algorithm is* $O(m \cdot MT \cdot n)$*.*

Proof. Combining the complexities of each step:

1. Initialization of Makespan Threshold (MT): $O(n \cdot m)$
2. Sorting Processors: $O(m \log m)$
3. Dynamic Programming for Knapsack Problem with Memoization: $O(m \cdot MT \cdot n)$
4. Assignment and Update: $O(n \cdot m)$

Summing these complexities:

$$O(n \cdot m) + O(m \log m) + O(m \cdot MT \cdot n) + O(n \cdot m)$$

Since $O(n \cdot m)$ and $O(m \log m)$ are dominated by $O(m \cdot MT \cdot n)$ and the MT is a value calculated beforehand and considered a constant, the overall complexity is simplified to:

$$O(m \cdot MT \cdot n)$$

Thus, the total time complexity for the KaCoSTA algorithm is $O(m \cdot n)$. □

5.4.3 Evaluation

Our second approach, Knapsack-based Co-Scheduling Algorithm for Task Allocation (KaCoSTA), represented a significant advancement in dynamic task scheduling by integrating knapsack optimization principles with dynamic programming to address the complexities of

heterogeneous computing systems. Unlike our first method DyTAg (See Section 4.2 Chapter 4), KaCoSTA explicitly accounts for task precedence and succession constraints, ensuring that dependent tasks are scheduled in the correct order while optimizing resource utilization. This innovative method demonstrated superior performance in terms of makespan reduction and resource efficiency, paving the way for its recognition in the academic community. The results of this work culminated in the publication of a peer-reviewed paper titled "An Innovative Task Scheduling Method Using the Knapsack Algorithm in Heterogeneous Computing Systems"[1] with DOI: https://doi.org/10.31449/inf.v48i16.5765. This publication highlights the practical and theoretical impact of our approach on the field of distributed computing.

The allocation strategy evaluation presented here investigates KaCoSTA's effectiveness in heterogeneous computing environments. Its performance is systematically compared against established algorithms. Through this comparative analysis, the evaluation demonstrates how KaCoSTA's global allocation strategy leads to superior load balancing and reduced makespan in both static and dynamic scheduling scenarios.

KaCoSTA is designed to optimize task allocation by simultaneously considering multiple processors, leveraging knapsack problem principles to achieve an optimal balance between execution time and resource utilization. Unlike sequential scheduling strategies, which allocate tasks processor by processor, KaCoSTA adopts a global allocation approach that improves load balancing and minimizes makespan.

To assess the effectiveness of the proposed scheduling approach, this section presents an extensive evaluation of KaCoSTA. This approach was developed by addressing the co-scheduling problem and proposing a global task allocation strategy that considers task precedence and resource heterogeneity. Additionally, a detailed complexity analysis is conducted, enabling accurate evaluation of the algorithm's performance in terms of scalability, execution efficiency, and adaptability in heterogeneous computing environments.

The following section details the experimental setup, dataset characteristics, evaluation criteria, and comparative performance results. These analyses highlight KaCoSTA's effectiveness in enhancing scheduling efficiency and demonstrate its potential for application in distributed and high-performance computing environments.

To validate the theoretical advantages of KaCoSTA, we conduct a comprehensive evaluation across key performance metrics, including makespan, resource utilization, and scheduling overhead. Both synthetic and benchmark datasets are employed to simulate realistic heterogeneous environments, thereby ensuring the robustness and generalizability of the results. Moreover, KaCoSTA's scalability is assessed by varying the number of tasks and processors, allowing for an in-depth analysis of its adaptability under dynamic computing

[1] https://informatica.si/index.php/informatica/article/view/5765

conditions.

5.5 Experimental Studies

To evaluate the effectiveness and robustness of the proposed scheduling algorithms, this section presents a series of experimental studies conducted in heterogeneous computing environments. The experiments aim to assess key performance indicators such as makespan, resource utilization, and scalability. We begin by comparing our techniques with traditional scheduling methods like Min-Min and QoS-guided Min-Min, which serve as common baselines in the literature. Subsequently, we benchmark our algorithms against more recent and advanced approaches to demonstrate their adaptability and superiority under dynamic and realistic workload conditions. All tests are carried out in a real-time context to ensure the practical relevance and reliability of the results.

To generate random graphs with diverse structural characteristics, A Direct Acrylic Graph (DAG) Generator is used, which supports multiple configurable parameters. Key parameters include the number of nodes in the graph, the graph's width, the density of edges between levels, and the number of edges that allow a task to skip from one level to another. A total of 1000 random DAGs were generated using this tool and were subsequently used as input to evaluate the proposed scheduling algorithm. The following parameters were set and combined in all possible combinations:

- **Number of tasks (n)**: {10, 50, 100, 200, 500}
- **Graph shape (FAT)**: {0.1, 0.5, 1, 5, 10}
- **Computation-to-Communication Ratio (CCR)**: {0.1, 5, 10}
- **Edge density**: 0.5
- **Jump factor**: 5

Each parameter in the DAG generation process plays a specific role in shaping the structure and complexity of the resulting graph. The number of tasks (n) corresponds to the number of nodes in the DAG. The FAT parameter controls the height-to-width ratio of the DAG: smaller values produce narrower (thinner) graphs, while larger values generate wider DAGs with more child nodes per parent [105]. The Computation-to-Communication Ratio (CCR), defined as the ratio of the sum of edge weights to the sum of node weights, is adjusted to simulate varying computational and communication demands. The density parameter determines the number of edges between two adjacent levels of the DAG—higher values lead to denser connections. Finally, the jump factor allows edges to connect nodes

across multiple levels; for example, a jump value of 5 enables a node at level l to have children between levels $l + 1$ and $l + 5$, increasing task dependency flexibility and graph complexity.

Regarding the datasets adopted in our experiments, we utilized a mix of synthetic and benchmark datasets [2] to evaluate the performance of our approach. We chose this kind of datasets to cover a variety of tasks' sizes and complexities, that reflect real-world scenarios in heterogeneous computing systems.

Comparing our proposed techniques with exact methods is essential to validate their effectiveness and optimality. Exact algorithms, though often computationally expensive, provide benchmark solutions that serve as a reference for evaluating the quality of heuristic or metaheuristic approaches. This comparison not only highlights the performance trade-offs but also demonstrates how closely the proposed methods approximate optimal solutions under realistic constraints, thereby reinforcing their practical relevance and applicability.

5.5.1 Exact Methods Comparison

Comparing the results of our algorithm to exact methods and metaheuristics [106] is a critical step in validating its effectiveness, scalability, and practical applicability in task scheduling for heterogeneous computing systems. Exact methods, such as branch-and-bound or integer programming, are designed to guarantee optimal solutions, making them the gold standard for evaluating algorithm performance on small-scale problems. By demonstrating that our algorithm produces results close to these optimal solutions, we establish its accuracy and reliability. However, exact methods are computationally infeasible for larger problem sizes due to their exponential complexity. This is where the strength of our algorithm lies it provides near-optimal results in a fraction of the time, highlighting its efficiency and scalability for real-world applications.

Metaheuristics, such as Genetic Algorithms (GA) [65, 107–109], Particle Swarm Optimization (PSO) [110–112], and Simulated Annealing (SA) [113, 114], are widely adopted for solving NP-hard problems like task scheduling due to their ability to handle complex, dynamic, and large-scale scenarios. These techniques often involve trade-offs between solution quality and computational overhead. By comparing our algorithm to these metaheuristics, we can evaluate its performance in terms of execution time, makespan, and resource utilization under diverse conditions. If our algorithm matches or outperforms metaheuristics in these metrics, it underscores its practical relevance and adaptability to heterogeneous and distributed environments.

[2] Experimented benchmark datasets are taken from http://mistic.heig-vd.ch/taillard/problemes.dir/ordonnancement.dir/ordonnancement.html

Moreover, such comparisons allow us to demonstrate the robustness of our approach across various scenarios, including static and dynamic workloads, while showcasing its advantages in balancing task allocation and optimizing resource usage. Evaluating our algorithm alongside both exact methods and metaheuristics not only validates its theoretical contributions but also positions it as a reliable and competitive solution for real-world scheduling challenges, ensuring confidence in its broader applicability.

In our work, we employ a Mixed-Integer Linear Programming (MILP) [115] model to determine the optimal makespan for the task scheduling problem in heterogeneous computing systems. The MILP formulation serves as a benchmark, providing globally optimal solutions for small to medium-sized problem instances. By explicitly modeling task assignment, processor constraints, and makespan minimization, the MILP captures the essence of the scheduling problem with high precision.

MILP guarantees optimal solutions within its feasible region by systematically exploring all possibilities. This makes it an ideal benchmark for evaluating the performance of heuristic or metaheuristic algorithms. For scheduling problems, MILP can minimize the makespan C_{max}, ensuring that the task allocation is globally optimal.

The results of our proposed approaches, DyTAg and KaCoSTA, are compared against the optimal solutions obtained using the MILP model. This comparison allows us to evaluate the effectiveness and efficiency of our algorithms in approximating optimal solutions while highlighting their scalability and adaptability to larger problem instances where MILP becomes computationally infeasible. The results demonstrate that our methods achieve near-optimal performance, validating their practical applicability and competitive advantage in dynamic and heterogeneous environments.

5.5.2 MILP Model for Task Scheduling

This section presents the MILP formulation used to find the optimal makespan for task scheduling in heterogeneous computing systems.

1. **Decision Variables**

   $x_{i,j} \in \{0, 1\}$: A binary variable, where:

   $$x_{i,j} = \begin{cases} 1 & \text{if task } t_i \text{ is assigned to processor } k_j, \\ 0 & \text{otherwise.} \end{cases}$$

   $C_{\max}$: A continuous variable representing the makespan, i.e., the maximum completion time across all processors.

2. **Objective Function** The objective is to minimize the makespan $C_{\max}$:

$$\text{Minimize: } C_{\max} \tag{5.5.1}$$

3. **Constraints**

Task Assignment: Each task t_i must be assigned to exactly one processor:

$$\sum_{j=1}^{m} x_{i,j} = 1, \quad \forall i \in \{1, \ldots, n\} \tag{5.5.2}$$

Processor Load: The total processing time for tasks assigned to each processor p_j must not exceed the makespan:

$$\sum_{i=1}^{n} e_{i,j} \cdot x_{i,j} \leq C_{\max}, \quad \forall j \in \{1, \ldots, m\} \tag{5.5.3}$$

where $e_{i,j}$ is the processing time of task t_i on processor p_j.

Binary Constraints: The decision variables $x_{i,j}$ are binary:

$$x_{i,j} \in \{0, 1\}, \quad \forall i \in \{1, \ldots, n\},\ j \in \{1, \ldots, m\} \tag{5.5.4}$$

The objective function (5.5.1) minimizes the makespan $C_{\max}$, ensuring the shortest possible schedule length. Constraint (5.5.2) ensures that each task is assigned to exactly one processor. Constraint (5.5.3) limits the total processing time on each processor to the makespan. Finally, constraint (5.5.4) enforces the binary nature of the task assignment variables.

This MILP model provides an exact solution to the task scheduling problem, serving as a benchmark for evaluating heuristic and metaheuristic approaches.

The effectiveness of task scheduling algorithms in Heterogeneous Computing Systems (HCS) hinges on their ability to balance competing objectives, such as minimizing makespan and maximizing resource utilization, across varying system configurations.

The experiments were conducted across diverse datasets, each designed to simulate realistic and challenging task scheduling scenarios. These datasets vary in task sizes, dependencies, and processor heterogeneity, providing a comprehensive benchmark for evaluating the robustness and adaptability of DyTAg and KaCoSTA. Key performance metrics, including makespan and resource utilization, were used to assess the algorithms' efficiency in handling both static and dynamic environments.5.2.1

5.5.3 Synthetic Datasets

To rigorously evaluate the performance of the proposed algorithms under controlled conditions, we employ synthetic datasets that simulate a variety of execution scenarios. These datasets are designed to reflect diverse task characteristics, including heterogeneous execution times and dependency structures, as well as varying processor capabilities within a heterogeneous computing system (HCS). As shown in Table 5.1, the synthetic workloads aim to mimic real-world conditions while enabling reproducible and scalable experimentation.

Dataset	Number of tasks	Type of execution time	Number of heterogeneous processors (NHP)
A	<50	random, uniform	2, 3
B	500	random	10
C	1000	follow a normal distribution	15

Table 5.1: Characteristics of experimented datasets

More precisely, the chosen synthetic datasets allow to understand the impact of task distribution and processor heterogeneity on the efficiency of the implemented algorithms. Moreover, we taken in consideration benchmark datasets that offer real-world scenarios to validate the practical applicability of the proposed algorithm.

1. Experiment 1

In this comparison, we aim to highlight the differences between various approaches by comparing KaCoSTA's results with those of Min-Min, Max-Min, and other methods in the literature. To achieve the aforementioned objectives, we consider the same example seen in Chapter 4 with independent tasks on three heterogeneous processors, as described and summarized in the following Table 5.2:

Processors	t_1	t_2	t_3	t_4	t_5	t_6	t_7
k_1	77	56	23	29	9	40	31
k_2	98	90	54	50	22	65	76
k_3	82	70	40	43	17	51	45

Table 5.2: Tasks-set execution times on the processors k_1, k_2, and k_3.

First, the estimated makespan threshold, denoted as $Makespan_{\text{Threshold}}$, is calculated using Equation 5.2.1.

$Makespan_{Threshold} \Rightarrow \frac{1}{3^2}(1068) = \frac{1068}{9} \approx 119$ Then according to the makespan threshold result, a schedule is given by means of DP using knapsack algorithm presented by our approach, the schedule result's makespan (Figure 5.6) does not exceed the $makespan_{Threshold}$. Finally the results are compared with Min-Min [116] and QoS guided Min-Min [47] in the following figure 5.6

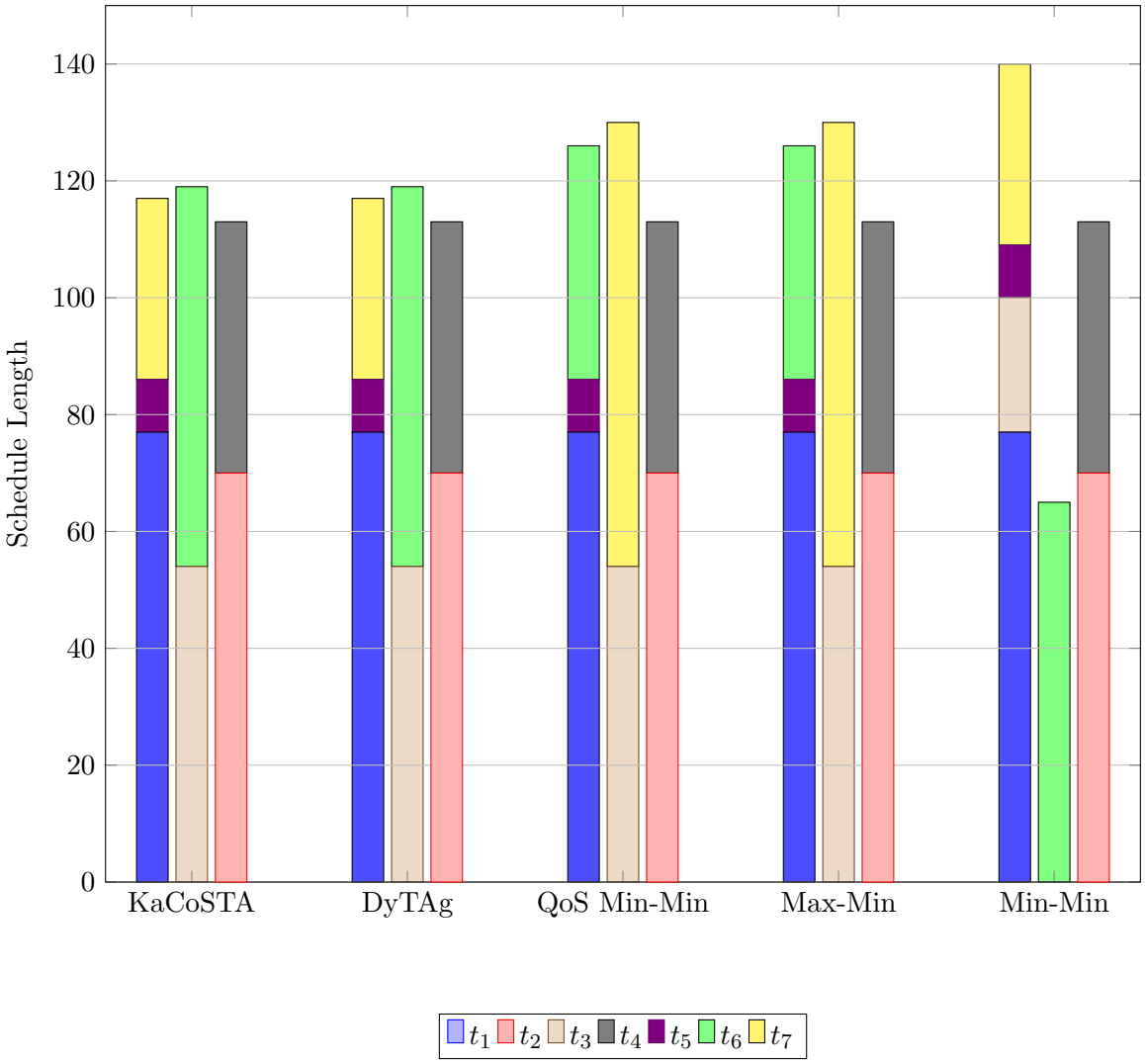

Figure 5.6: Min-Min, Max-Min, QoS guided Min-Min, DyTAg and KaCoSTA makespan comparison

2. Experiment 2

To ensure a fair and consistent evaluation, this experiment adopts the same task set and configuration values as presented in the benchmark study by [4]. The task execution times and associated data, summarized in Table 5.3, provide a standardized basis for comparison.

This enables a direct performance assessment between our proposed scheduling approach and existing algorithms under identical heterogeneous computing conditions. By replicating the original dataset and scenario, we aim to validate the robustness and efficiency of our method in a well-established experimental context.

Processors	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}
k_1	4	15	4	13	10	7	8	4	12	6	9
k_2	6	22.5	6	19.5	15	10.5	12	6	18	9	13.5

Table 5.3: Tasks-set execution times on the processors k_1 and k_2.

In this part of the test, we assess the comparison between PHTS, PETS, Sorted Nodes in Levelled DAG Division (SNLDD), CPOP, HEFT, MILP and our proposed approach, KaCoSTA.

The results of the experiment, as shown in Figure 5.7, demonstrate a clear advantage. Specifically, in terms of scheduling length, our approach outperforms the other compared methods. Notably, our heuristic, KaCoSTA, achieved a makespan of $\max(59, 59) \Rightarrow$ makespan $= 59$ (see results in Figure 5.8). This outcome is particularly significant as it matches the results produced by the MILP (Mixed Integer Linear Programming) exact method, indicating that KaCoSTA not only delivers near-optimal solutions efficiently but also approaches the quality of optimal scheduling techniques with considerably lower computational overhead.

Effectively, KaCoSTA achieves a makespan of 59, while the next best approach provides a makespan of no less than 61. This indicates that our proposed approach shows a 3.39% improvement over PHTS. Additionally, the utilization rates for processors k_1 and k_2 are both 59.

3. Experiment 3

In this experiment, we consider a task set and processor configuration originally presented by [79], which serves as a widely used benchmark for evaluating task scheduling algorithms. The execution costs of the tasks on three heterogeneous processors are detailed in Table 5.4, while the inter-task dependencies are illustrated through a Directed Acyclic Graph (DAG) in Figure 5.9. This setup provides a representative test case for assessing the effectiveness of our proposed approach in handling precedence constraints and heterogeneous execution environments.

The obtained results presented in the following Figure 5.10, shown how the three approaches process, and how are tasks assigned to the given processors. The heterogeneous system example design is described by Ilavarasan et al. [79]

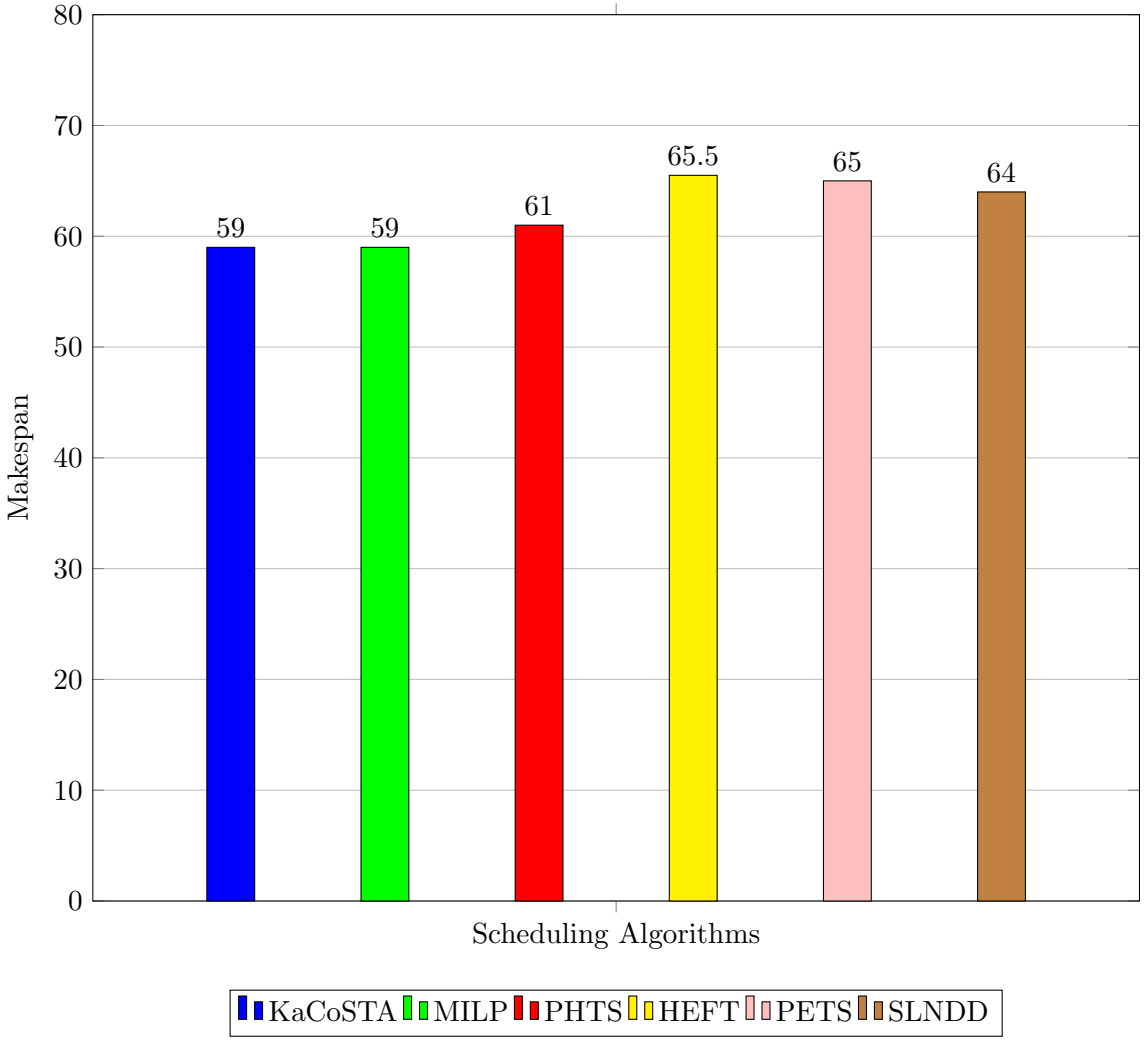

Figure 5.7: Makespan metric comparisons of all the four algorithms with KaCoSTA

As illustrated in Figure 5.7, there is a clear difference in scheduling length (makespan) between our approach and the other presented algorithms. Specifically, KaCoSTA achieves makespans of 35, 49, and 39 on Processor 1, Processor 2, and Processor 3, respectively. In contrast, algorithms like PETS and CPOP result in significantly higher makespans, ranging from 76 to 85 across all processors. This reflects an improvement of approximately 35.52% over PETS. Furthermore, when compared to the MILP exact method, which produces optimal makespan of 43, 45, and 41 on the same processors, KaCoSTA delivers comparable performance, approaching the optimal solution while significantly reducing computational cost. This highlights KaCoSTA's effectiveness as a practical and scalable heuristic for complex heterogeneous scheduling scenarios.

This disparity can be attributed to the fact that PETS and CPOP do not account for independent task groups and employ a task prioritization phase. This phase deter-

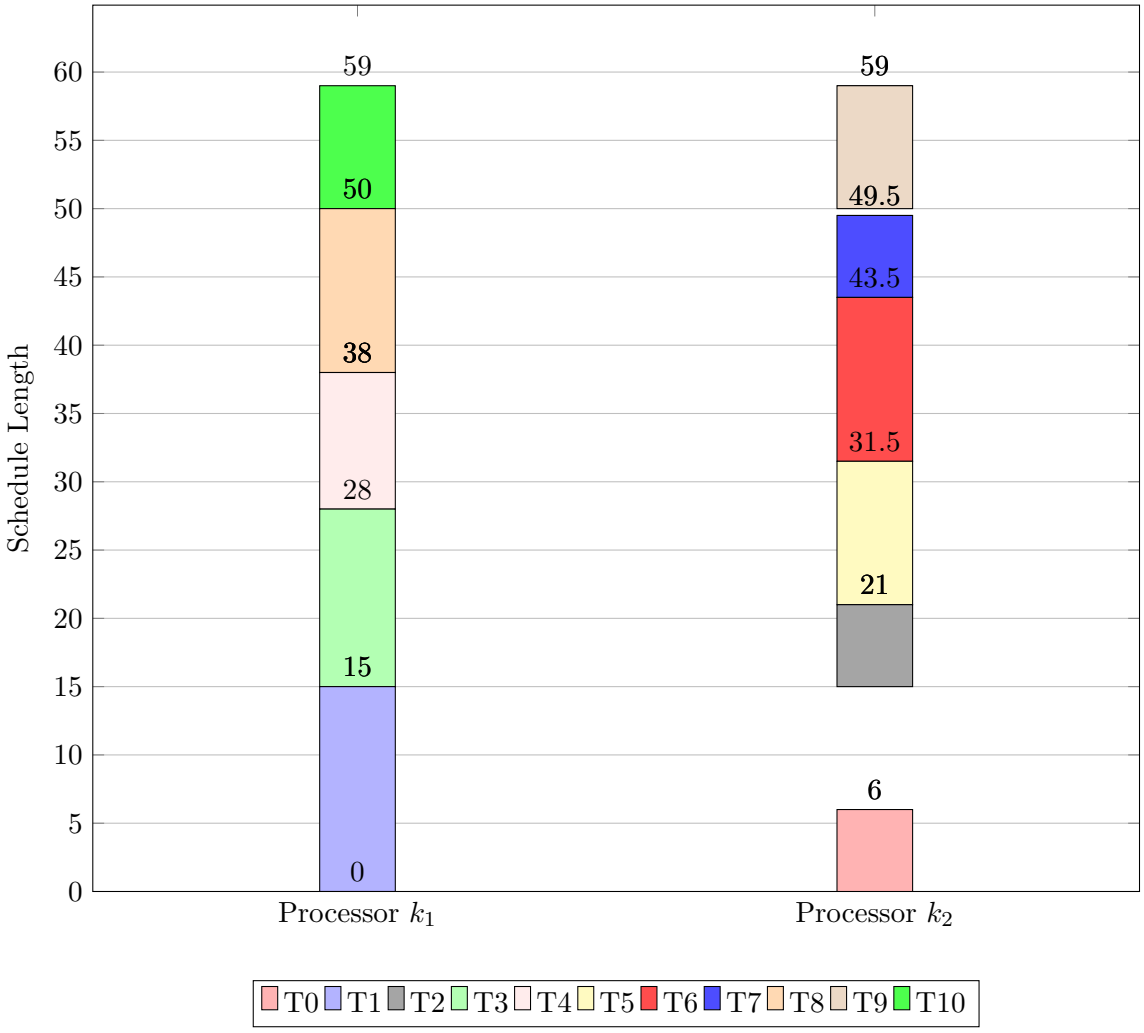

Figure 5.8: Schedule result generated by KaCoSTA

mines the priority of each task based on rank metrics, and is calculated through parental prioritization, as described in [117].

5.5.4 Benchmark Datasets

Because, we adopt a dynamic programming technique based on knapsack method (See Definition 9), which optimally balances the load across processors, KaCoSTA consistently achieves lower makespan compared to existing algorithms across all datasets [3]. For example, in Dataset A, KaCoSTA achieves a makespan of 105, significantly lower than Min-Min's 140 and PHTS's 110. Thus KaCoSTA is the most efficient algorithm in task allocation and execution time minimization.

[3]Experimented benchmark datasets are taken from http://mistic.heig-vd.ch/taillard/problemes.dir/ordonnancement.dir/ordonnancement.html

Processors	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
k_1	14	13	11	13	12	13	7	5	18	21
k_2	16	19	13	8	13	16	15	11	12	7
k_3	9	18	19	17	10	9	11	14	20	16

Table 5.4: Tasks-set execution times on the processors k_1, k_2, and k_3.

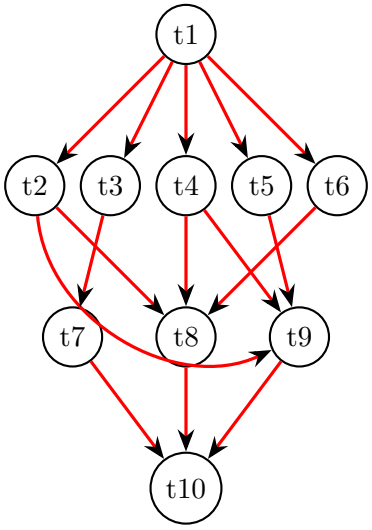

Figure 5.9: Tasks Dependencies DAG [1]

Regarding RU, KaCoSTA shows the highest level of efficiency, with an average of 85% in Dataset A, compared to 65% for Min-Min and 80% for PHTS. This result indicates that KaCoSTA effectively leverages available processing power, reducing idle times and enhancing overall system performance.

In addition, KaCoSTA performs particularly well in scenarios with high task heterogeneity and varying processor capabilities. In Dataset B, with 500 tasks and 10 processors, KaCoSTA's makespan is 600, compared to 700 for Min-Min and 620 for PHTS. This highlights KaCoSTA's adaptability and efficiency in complex environments.

Moreover, lower variance and narrower confidence intervals for KaCoSTA (e.g., 6.5 variance and [103, 107] Confidence Interval (CI) in Dataset A) indicate more consistent performance and reliability compared to other algorithms. This robustness is crucial for applications requiring predictable and stable task scheduling outcomes.

Thus, KaCoSTA demonstrates clear advantages over existing algorithms in terms of makespan reduction, RU, and consistent performance across various datasets.

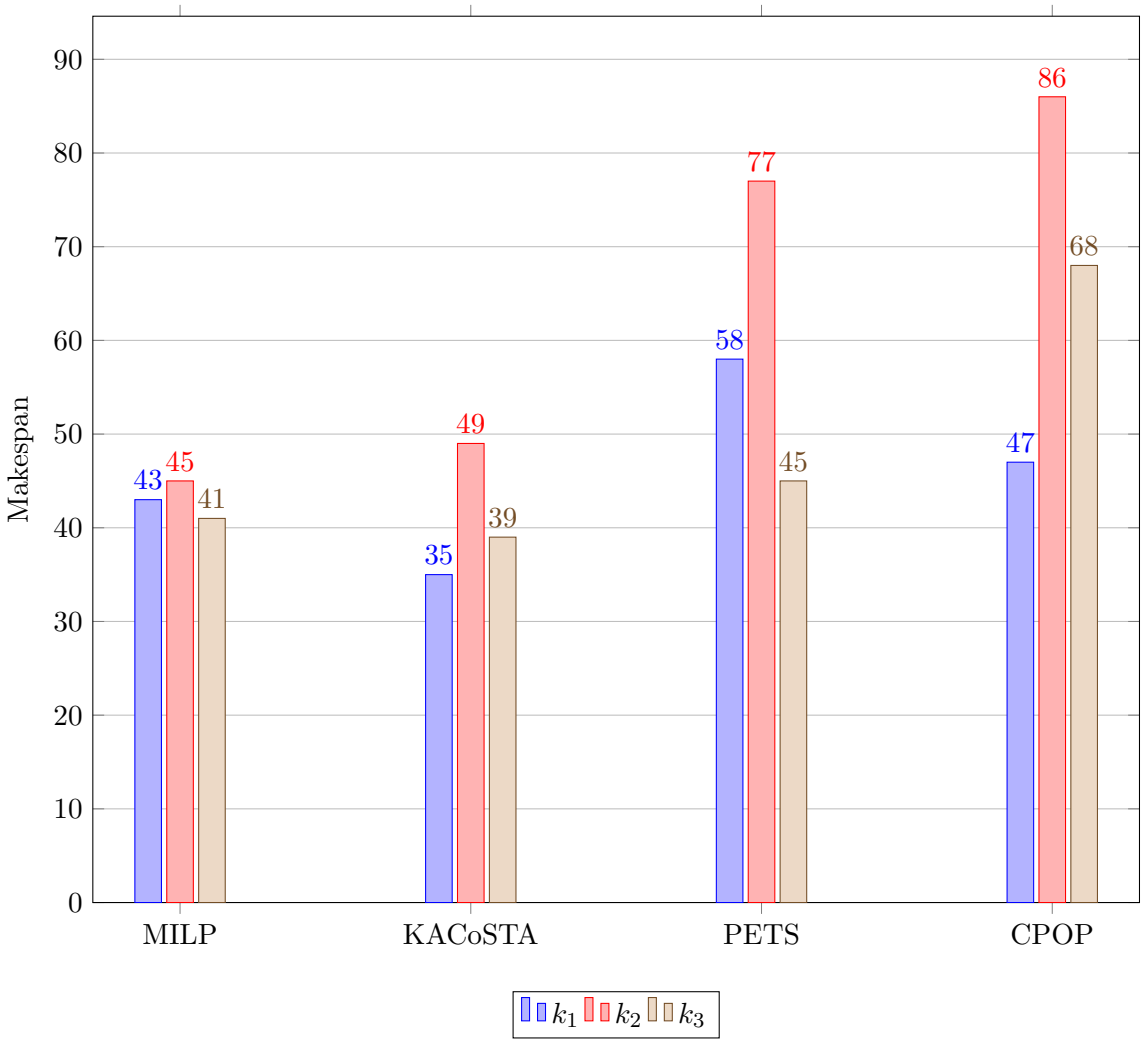

Figure 5.10: Comparison of elapsed time on each processor k_1, k_2 and k_3 for MILP, KaCoSTA, PETS and CPOP

5.5.5 Performance Analysis

This section evaluates the performance of the proposed KaCoSTA algorithm in environments characterized by high task heterogeneity and diverse processor capabilities. The goal is to assess its effectiveness in managing complex scheduling scenarios where task dependencies, execution priorities, and system dynamics play a critical role. By comparing KaCoSTA with traditional and widely-used scheduling methods, we aim to highlight its advantages in optimizing resource utilization, reducing makespan, and maintaining adaptability in heterogeneous computing environments.

For this comparison, the results demonstrate comparatively better performance of the proposed approach. Specifically, for the given set of tasks (Table 5.3), KaCoSTA provides a makespan of 119, which does not exceed the estimated $Makespan_{Threshold}$. This result is

Dataset	Algorithm	Makespan	V	95% CI	RU	V	95% CI U
Dataset A	Min-Min	140	15.2	[135, 145]	65%	2.5	[63%, 67%]
	Max-Min	130	12.8	[126, 134]	70%	3.1	[68%, 72%]
	QoS Min-Min	125	10.4	[121, 129]	72%	2.8	[70%, 74%]
	HEFT	115	8.7	[112, 118]	78%	1.9	[77%, 79%]
	PHTS	110	7.5	[107, 113]	80%	1.7	[79%, 81%]
	PETS	120	9.6	[117, 123]	75%	2.4	[73%, 77%]
	KaCoSTA	**105**	**6.5**	**[103, 107]**	**85%**	**1.2**	**[84%, 86%]**
Dataset B	Min-Min	700	45.6	[688, 712]	60%	3.9	[58%, 62%]
	Max-Min	680	40.8	[669, 691]	65%	4.1	[63%, 67%]
	QoS Min-Min	670	38.2	[659, 681]	67%	3.7	[65%, 69%]
	HEFT	640	35.4	[630, 650]	72%	2.9	[71%, 73%]
	PHTS	620	30.6	[611, 629]	75%	2.2	[74%, 76%]
	PETS	650	36.8	[640, 660]	70%	3.4	[68%, 72%]
	KaCoSTA	**600**	**25.2**	**[593, 607]**	**80%**	**1.8**	**[79%, 81%]**
Dataset C	Min-Min	1500	120.8	[1475, 1525]	55%	5.6	[53%, 57%]
	Max-Min	1450	110.4	[1427, 1473]	60%	4.8	[58%, 62%]
	QoS Min-Min	1400	105.2	[1378, 1422]	62%	4.5	[60%, 64%]
	HEFT	1350	98.6	[1330, 1370]	68%	3.8	[67%, 69%]
	PHTS	1300	85.4	[1283, 1317]	70%	3.1	[69%, 71%]
	PETS	1350	90.2	[1333, 1367]	66%	3.6	[65%, 67%]
	KaCoSTA	**1250**	**70.5**	**[1235, 1265]**	**75%**	**2.9**	**[74%, 76%]**

Table 5.5: Statistical Analysis of Algorithm Performance

Legend: Makespan – total task completion time; V – variance; 95% CI – confidence interval for makespan; RU – resource utilization; 95% CI U – confidence interval for RU.

achieved by assigning t_3 and t_6 to processor 2, t_2 and t_4 to processor 3, and the remaining tasks t_1, t_5, and t_7 to processor k_3. In comparison, Min-Min results in a makespan of 140, while both QoS-Guided Min-Min and Max-Min yield a makespan of 130, leading to a 15% improvement in this test. It is noted that the complexities of KaCoSTA and Max-Min are $\mathcal{O}(n \cdot m)$ and $\mathcal{O}(n^2 \cdot m)$, respectively, where n is the number of tasks, m is the number of processors, and MT is the calculated makespan threshold defined in Eq. 5.2.1.

KaCoSTA performs particularly well in scenarios with high task heterogeneity and varying processor capabilities. For example, in Dataset C, with 1000 tasks and 15 processors, KaCoSTA achieves a makespan of 1250, compared to 1500 for Min-Min and 1350 for PETS. This highlights the algorithm's robustness and efficiency in complex and dynamic environments.

The lower variance and narrower confidence intervals for KaCoSTA, such as a variance of 6.5 and a 95% confidence interval of [103, 107] in Dataset A, indicate more consistent performance and reliability compared to other algorithms. This is crucial for applications requiring predictable and stable task scheduling outcomes.

Algorithm KaCoSTA significantly improves upon existing task scheduling algorithms (see Table 5.7) in terms of makespan and RU. More precisely, KaCoSTA outperforms Min-Min, Max-Min, and QoS-guided Min-Min to achieve a low makespan, which is due to their unsuitability for long tasks, less efficiency in a dynamic environment, and limited scalability, respectively. In addition, despite the improvement in terms of scalability of the SOTA algorithms considered in this work, our algorithm is still the best in terms of makespan and RU. This is due to KaCoSTA's complexity, which is still the best.

Moreover, the observed makespan's decreasing and the RU's increasing are primarily due to KaCoSTA's dynamic allocation of tasks, which is based on its real-time processing ability. The experimental results of our approach demonstrate significant improvements across all key metrics. We noticed the efficiency of dynamic programming combined with the knapsack techniques, which enables KaCoSTA to handle larger task sets with better scalability and quadratic complexity.

Our approach has proven its efficiency in terms of makespan, RU. The reason is that our approach succeeds to address the issues of the tested SOTA methods (see Table 5.6).

Algorithm	Handicap
Min-Min	Performs poorly in RU due to its tendency to leave some processors underutilized. This results in a higher makespan in heterogeneous environments.
Max-Min	Balances the load better than Min-Min but still lacks efficiency in dynamic environments where task execution times vary significantly.
QoS Min-Min	Incorporates QoS constraints but sacrifices scalability and efficiency.
HEFT	Achieves better makespan and RU but suffers from complexity in implementation, making it less adaptable to varying scenarios.
PHTS	Performs well in terms of makespan and RU but has high complexity and less adaptability, limiting its effectiveness.
PETS	Effective for preemptable tasks but inefficient for non-preemptive tasks, leading to a higher makespan and lower RU.

Table 5.6: SoTA algorithms handicap

5.5.6 Discussion

The novelty of our approach lies in the new modeling and solving techniques that allow optimal results in terms of makespan and RU. More accurately, KaCoSTA models a task scheduling problem as a knapsack problem, which allows a structured and efficient task allocation. In addition, a DP technique is utilized to systematically solve the scheduling problem, breaking it down into manageable subproblems. This ensures optimal task allocation and load balancing across processors. $Makespan_{Threshold}$ Estimation is used to guide the scheduling process and to balance the trade-off between minimizing makespan and maximizing RU.

Thus, our algorithm is highly adaptable to various HCS scenarios, effectively handling different task sets and processor configurations. KaCoSTA demonstrates consistent performance with lower variance and narrower confidence intervals, ensuring reliable scheduling outcomes. This robustness is crucial for applications requiring stable and predictable task execution times.

Algorithm	**Makespan**	**RU**	**Complexity**	**Experimental Characteristics**
HEFT	Medium	High	$O(n^2 + n \cdot m)$	Task prioritization, Difficult to implement
CPOP	Medium	Medium	$O(n \log n + n \cdot m)$	Task ranking
PETS	Medium	Low	$O(n \log n + n \cdot m)$	Task prioritization, Less efficient for non-preemptive tasks
PHTS	Low	Medium	$O(n \log n + n \cdot m)$	Path-based priority, Less adaptable
KaCoSTA	**Low**	**High**	$O(n \cdot m)$	Dynamic programming, Knapsack-based model Task prioritization, Tasks' run times are known beforehand, Deterministic conditions

Table 5.7: Characteristics of SOTA scheduling algorithms compared to KaCoSTA

5.6 Conclusion

In this chapter, we presented the methodological framework underlying our second proposed approach, KaCoSTA, and described the corresponding experimental study for task

scheduling in heterogeneous computing systems. Building on the foundation laid by DyTAg, KaCoSTA introduces an advanced co-scheduling strategy that explicitly incorporates task precedence and succession constraints.

By combining dynamic programming with knapsack optimization principles, KaCoSTA effectively accounts for task interdependence constraints while optimizing execution order and minimizing overall processing time. Its adaptive scheduling mechanism enables dynamic responsiveness to varying workload patterns and heterogeneous resource capacities.

Together, DyTAg and KaCoSTA address critical challenges in task scheduling across distributed and heterogeneous environments, offering scalable, adaptable, and efficient solutions. The experimental evaluation provided in this chapter confirms the effectiveness of the knapsack-based scheduling approach and demonstrates its strong performance across a variety of computational scenarios.

Chapter 6

Conclusion

This thesis addressed the critical challenge of task scheduling in Heterogeneous Computing Systems (HCS) by proposing novel algorithmic solutions aimed at improving performance and resource utilization. The first contribution, DyTAg, is a dynamic programming-based task allocation algorithm designed to optimize task scheduling in environments with diverse processing capabilities particularly in Heterogeneous Multiprocessor Systems (HMS). It systematically identifies and classifies tasks, matches them to suitable processors, and dynamically balances workloads to minimize idle time and makespan. The results confirm that DyTAg offers a significant improvement in scheduling efficiency within heterogeneous systems.

The effectiveness of DyTAg was validated through a comparative performance analysis against established scheduling algorithms such as Min-Min, QoS-guided Min-Min, and Max-Min. The experimental results demonstrated that DyTAg consistently achieves lower computation times and improved task distribution across processors. Designed specifically for independent tasks, DyTAg leverages dynamic programming to dynamically adapt to both task and system characteristics, thereby outperforming traditional methods in terms of makespan reduction and overall scheduling efficiency.

To address more complex scheduling scenarios involving interdependent tasks with priority constraints, we extended our research to HMS by introducing KaCoSTA a knapsack-based co-scheduling algorithm. This contribution, along with KReSTA and KISTA, targets global optimization across multiple processors, integrating advanced techniques such as dynamic programming and combinatorial optimization. Together, these methods provide scalable and adaptive scheduling strategies tailored to the dynamic nature and computational diversity of modern HCS.

We first introduced KReSTA, a recursive local task scheduling algorithm designed to efficiently handle relatively small task sets (e.g., $n \lesssim 35$). KReSTA's recursive structure

enabled low-overhead scheduling with strong performance in constrained scenarios. However, It became less efficient at handling larger and more complex task sets due to the recursive overhead. To overcome these limitations, we proposed KISTA, an iterative refinement of KReSTA. KISTA significantly reduced the algorithm's time complexity, making it more suitable for larger and more complex task sets. This evolution from KReSTA to KISTA underscores the importance of adapting scheduling strategies to match the scale and heterogeneity of the system, and suggests that hybrid approaches blending recursive precision with iterative scalability can be particularly effective in heterogeneous computing environments.

Building upon the limitations identified in local scheduling approaches such as KReSTA and KISTA, we introduced KaCoSTA (Knapsack-based Co-Scheduling Algorithm for Task Allocation) a global, co-scheduling strategy designed to optimize task assignment across multiple processors. Unlike the localized decision-making of its predecessors, KaCoSTA adopts a system-wide perspective, leveraging both knapsack optimization and dynamic programming to enhance scheduling performance in HCS. The approach unfolds in three key phases: 1. Makespan Threshold Estimation (MT): Analogous to the knapsack capacity, MT defines the maximum allowable processing time for each processor, guiding the initial allocation. 2. Task Allocation: Tasks are distributed across processors such that execution times remain within the MT, balancing computational costs and task priorities. 3. Schedule Optimization: Final adjustments are made to reduce idle time and improve overall resource utilization, ensuring an efficient and balanced schedule.

Through rigorous experimentation and comparative analysis against established scheduling algorithms such as PHTS, HEFT, PETS, and SLNDD, KaCoSTA demonstrated significant performance advantages. It consistently achieved lower makespan and improved resource utilization across a wide range of benchmark datasets and heterogeneous scenarios. In particular, KaCoSTA outperformed QoS-guided Min-Min, HEFT, and SLNDD by ensuring better task distribution and minimizing execution delays. Its ability to adapt dynamically to varying task dependencies and processor configurations underscores its suitability for modern computational environments, including cloud computing, edge computing, and large-scale distributed systems.

One of the standout features of KaCoSTA is its reliance on dynamic programming, which systematically evaluates all possible task allocations to find the optimal solution. This approach, combined with the mathematical modeling inspired by the knapsack problem, ensures structured, efficient, and scalable task allocation. The experimental results revealed that KaCoSTA not only excels in theoretical benchmarks but also aligns well with real-world scenarios, making it a robust choice for practical deployment.

Beyond its technical contributions, this thesis highlights KaCoSTA's potential as a

foundation for future research and development. By overcoming the limitations of traditional scheduling algorithms and introducing a dynamic, adaptive paradigm, KaCoSTA establishes a flexible and robust approach for handling complex task allocation in heterogeneous systems. Its versatility makes it a valuable tool for both researchers and practitioners aiming to design efficient, scalable solutions in diverse computing environments.

Looking ahead, several avenues for future research and development emerge. A particularly promising direction is the integration of Artificial Intelligence (AI) techniques to further enhance KaCoSTA's adaptability. For instance, AI-driven dynamic estimation of the MT could enable the algorithm to respond in real time to changing workloads and processor conditions, leading to even more efficient task allocation. Moreover, incorporating machine learning models could refine priority assignment by accurately predicting task execution times and inter-task dependencies. These enhancements would increase KaCoSTA's robustness and make it highly suitable for real-time systems, high-performance computing, and cloud-based environments.

In conclusion, this thesis advances the field of task scheduling by introducing innovative methodologies that overcome the limitations of traditional algorithms. By leveraging dynamic programming and optimization techniques inspired by the knapsack problem, we have developed scheduling approaches that are both theoretically robust and practically scalable. We began by introducing DyTAg, a dynamic programming-based algorithm designed for task allocation in heterogeneous multiprocessor systems with independent tasks. Building on this, we developed KaCoSTA, the central contribution of this work, which integrates co-scheduling principles with knapsack-based optimization to efficiently handle interdependent tasks and priority constraints. KaCoSTA stands out as a highly efficient and adaptable solution, demonstrating significant improvements in makespan reduction, resource utilization, and overall system performance across diverse and dynamic heterogeneous computing environments. This research lays a solid foundation for future advancements in task scheduling, with promising implications for resource management and computational efficiency in high-performance multiprocessor and distributed systems.

Chapter 7

Perspectives and Future Work

The field of distributed systems continues to evolve, driven by the need for high-performance computing, scalability, and adaptability in dynamic environments. Optimizing scheduling in heterogeneous computing systems remains a critical challenge, requiring strategies that efficiently allocate tasks while adapting to changing workloads, resource availability, and system constraints.

In this work, we have proposed DyTAg and KaCoSTA as dynamic task scheduling approaches to improve distributed system performance. DyTAg utilizes dynamic programming to enhance task allocation efficiency, while KaCoSTA extends the scheduling strategy to consider all processors simultaneously, optimizing resource utilization and minimizing execution time. While these approaches have demonstrated promising results, several directions for improvement can further enhance their effectiveness and applicability in large-scale, real-world environments.

7.1 Artificial Intelligence for Adaptive Scheduling

One promising direction for future work is the integration of artificial intelligence (AI) and machine learning (ML) techniques into scheduling algorithms. AI-driven scheduling models can learn from system behavior, predicting workload patterns and dynamically adjusting task assignments based on historical data. Reinforcement learning (RL) could be explored to develop autonomous scheduling policies that optimize performance under varying workloads. Neural networks and deep learning models could further enhance decision-making by recognizing complex task dependencies and processor capabilities, improving scheduling efficiency beyond traditional optimization techniques.

7.2 Exploring Cloud and Edge Computing Environments

As modern distributed systems increasingly shift toward cloud and edge computing infrastructures, future enhancements to DyTAg and KaCoSTA could focus on optimizing task scheduling within cloud-based heterogeneous computing environments. Cloud computing introduces unique challenges, such as dynamic resource provisioning, variable bandwidth, and cost efficiency, requiring scheduling algorithms that can adapt in real-time. Edge computing presents another frontier where latency-sensitive tasks must be allocated efficiently across distributed nodes. Extending DyTAg and KaCoSTA to handle cloud-edge hybrid environments would enable better performance for applications in data-intensive computing, IoT, and real-time analytics.

7.3 Energy-Aware Scheduling for Sustainable Computing

Energy consumption has become a significant concern in distributed computing, particularly in large-scale heterogeneous environments. Future extensions of DyTAg and KaCoSTA could incorporate energy-aware constraints to minimize power usage while maintaining high performance. Techniques such as dynamic voltage and frequency scaling (DVFS) could be integrated into the scheduling model to balance computation speed with energy efficiency. AI-driven energy prediction models could also help optimize task placement, ensuring workloads are distributed in a way that minimizes energy consumption without sacrificing system performance.

7.4 Scalability and Fault-Tolerance Enhancements

Ensuring the scalability of DyTAg and KaCoSTA is crucial for their deployment in large-scale distributed systems. Future research could focus on optimizing these algorithms to handle increasing numbers of tasks and processors efficiently. Additionally, introducing fault-tolerant mechanisms, such as predictive failure detection and self-healing scheduling, would improve system reliability in the presence of node failures or network disruptions. Leveraging AI-driven anomaly detection could further enhance fault-tolerance, allowing proactive task reassignment before failures occur.

7.5 Real-Time and QoS-Aware Scheduling

Many distributed computing applications, such as autonomous systems, financial modeling, and healthcare analytics, require real-time task execution with strict Quality of Service

(QoS) constraints. Future research could explore integrating QoS-aware scheduling mechanisms into DyTAg and KaCoSTA, ensuring that time-sensitive tasks receive prioritized execution. Adaptive scheduling policies that dynamically adjust resource allocation based on QoS parameters, such as latency, throughput, and response time, would make these approaches more applicable to real-world scenarios.

Enhancing distributed system performance through dynamic adaptation remains an ongoing research challenge, particularly in heterogeneous computing environments. By incorporating AI-driven optimization, expanding to cloud and edge computing, integrating energy-aware constraints, improving scalability and fault tolerance, and introducing real-time QoS-aware scheduling, DyTAg and KaCoSTA can be significantly enhanced. These future developments will contribute to more efficient, intelligent, and sustainable distributed computing solutions, ensuring that scheduling strategies remain adaptable to the evolving needs of modern computing infrastructures.

Bibliography

[1] H. Topcuoglu, S. Hariri, and M.-Y. Wu, *Performance-effective and low-complexity task scheduling for heterogeneous computing*, *IEEE transactions on parallel and distributed systems* **13** (2002), no. 3 260–274.

[2] K. Singh, M. Alam, and S. K. Sharma, *A survey of static scheduling algorithm for distributed computing system*, *International Journal of Computer Applications* **129** (2015), no. 2 25–30.

[3] J. Fox and S. Clarke, *Exploring approaches to dynamic adaptation*, in *Proceedings of the 3rd International DiscCoTec Workshop on Middleware-Application Interaction*, pp. 19–24, 2009.

[4] R. Eswari, S. Nickolas, and M. Arock, *A path priority-based task scheduling algorithm for heterogeneous distributed systems*, *International Journal of Communication Networks and Distributed Systems* **12** (2014), no. 2 183–201.

[5] I. Foster, C. Kesselman, and S. Tuecke, *The anatomy of the grid: Enabling scalable virtual organization*, *The International Journal of High Performance Computing Applications* **15** (2001), no. 3 200–222.

[6] L. F. Bittencourt, E. R. Madeira, and N. L. da Fonseca, *Resource management and scheduling*, in *Cloud Services, Networking, and Management*, pp. 243–267. John Wiley & Sons, Inc., 2015.

[7] P. Kurp, *Green computing*, *Communications of the ACM* **51** (2008), no. 10 11–13.

[8] M. A. Rappa, *The utility business model and the future of computing services*, *IBM Systems Journal* **43** (2004), no. 1 32–42.

[9] M. Van Steen and A. S. Tanenbaum, *A brief introduction to distributed systems*, *Computing* **98** (2016) 967–1009.

[10] A. S. Tanenbaum and H. Bos, *Modern operating systems*. Pearson Education, Inc., 2015.

[11] A. K. Muppalla and K. Srinivasa, *Guide to High Performance Distributed Computing: Case Studies with Hadoop, Scalding and Spark*. Springer, 2015.

[12] D. Peleg, *Distributed Computing: A Locality-Sensitive Approach*. 2000.

[13] M. Singhal and A. D. Kshemkalyani, *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2008.

[14] D. B. Kahanwal and D. T. P. Singh, *The distributed computing paradigms: P2p, grid, cluster, cloud, and jungle*, *arXiv preprint arXiv:1311.3070* (2013).

[15] P. Trunfio, D. Talia, H. Papadakis, and P. Fragopoulou, *Peer-to-peer resource discovery in grids: Models and systems*, *Future Generation Computer Systems* (2007).

[16] I. J. Taylor and A. Harrison, *From P2P and grids to services on the web: evolving distributed communities*. Springer, 2008.

[17] H. Ponnapalli, A. Belapurkar, and A. Chakrabarti, *Distributed Systems Security: Issues, Processes, and Solutions*. Springer, 2009.

[18] P. K. Sinha, *Distributed Operating Systems: Concepts and Design*. Prentice Hall, 1998.

[19] J. E. Stone, D. Gohara, and G. Shi, *Opencl: A parallel programming standard for heterogeneous computing systems*, *Computing in science & engineering* **12** (2010), no. 3 66.

[20] S. Deniziak, *Cost-efficient synthesis of multiprocessor heterogeneous systems*, *Control and Cybernetics* **33** (2004), no. 2 341–355.

[21] J. Wei, Q. Kang, and H. He, *An effective iterated greedy algorithm for reliability-oriented task allocation in distributed computing systems*, *Journal of Parallel and Distributed Computing* **73** (2013) 1–10.

[22] M. B. Dhaou and D. Fayard, *The Task Allocation Problem*. Wiley, 2014.

[23] O. Sinnen, *Task Scheduling for Parallel Systems*. Wiley, 2006.

[24] A. K. Tripathi, D. P. Vidyarthi, B. K. Sarker, and L. T. Yang, *Analysis, Design, and Models*. 2014.

[25] M. Drozdowski, *Scheduling for Parallel Processing*. Springer, 2009.

[26] Y.-K. Kwok and I. Ahmad, *Static scheduling algorithms for allocating directed task graphs to multiprocessors*, *ACM Computing Surveys (CSUR)* **31** (1999), no. 4 406–471.

[27] F. de Oliveira Lucchese, E. J. Yero, F. S. Sambatti, and M. A. Henriques, *An adaptive scheduler for grids*, *Journal of Grid Computing* **4** (2006), no. 1 1–17.

[28] L. F. Bittencourt and E. R. Madeira, *A performance-oriented adaptive scheduler for dependent tasks on grids*, *Concurrency and Computation: Practice and Experience* **20** (2008), no. 9 1029–1049.

[29] V. Hamscher, U. Schwiegelshohn, A. Streit, and R. Yahyapour, *Evaluation of job-scheduling strategies for grid computing*, in *GRID 2000: Proceedings of the First IEEE/ACM International Workshop on Grid Computing*, pp. 191–202, Springer, 2000.

[30] J. Yu and R. Buyya, *Scheduling scientific workflow applications with deadline and budget constraints using genetic algorithms*, *Scientific Programming* **14** (2006), no. 3-4 217–230.

[31] S. F. Smith, *Is scheduling a solved problem?*, in *Multidisciplinary Scheduling: Theory and Applications: 1 st International Conference, MISTA'03 Nottingham, UK, 13–15 August 2003 Selected Papers*, pp. 3–17, Springer, 2005.

[32] J. A. Torkestani, *A new distributed job scheduling algorithm for grid systems*, *Cybernetics and Systems* (2013).

[33] M. L. Pinedo, *Scheduling: Theory, Algorithms, and Systems*. Springer, 2013.

[34] S. Cook, *The p versus np problem*, *Citeseer* (2000).

[35] Merriam-Webster, *Heuristic definition*, 2016. Accessed from http://www.merriam-webster.com/dictionary/heuristic.

[36] R. L. Henderson, *Job scheduling under the portable batch system*, in *Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 279–294, Springer, 1995.

[37] Ł. Sobaszek, A. Gola, and A. Świć, *Predictive scheduling as a part of intelligent job scheduling system*, in *Intelligent Systems in Production Engineering and Maintenance–ISPEM 2017: Proceedings of the First International Conference on Intelligent Systems in Production Engineering and Maintenance ISPEM 2017 1*, pp. 358–367, Springer, 2018.

[38] D. Liang, P.-J. Ho, and B. Liu, *Scheduling in distributed systems*, *Department of Computer Science and Engineering University of California, San Diego* (2000).

[39] Y.-K. K. Y.-K. Kwok and I. Ahmad, *A static scheduling algorithm using dynamic critical path for assigning parallel algorithms onto multiprocessors*, in *1994*

Internatonal Conference on Parallel Processing Vol. 2, vol. 2, pp. 155–159, IEEE, 1994.

[40] B. Andersson, S. Baruah, and J. Jonsson, *Static-priority scheduling on multiprocessors*, in *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001)(Cat. No. 01PR1420)*, pp. 193–202, IEEE, 2001.

[41] V. Suresh and D. Chaudhuri, *Dynamic scheduling—a survey of research*, *International journal of production economics* **32** (1993), no. 1 53–63.

[42] D. Ouelhadj and S. Petrovic, *A survey of dynamic scheduling in manufacturing systems*, *Journal of scheduling* **12** (2009) 417–431.

[43] L. Renke, R. Piplani, and C. Toro, *A review of dynamic scheduling: context, techniques and prospects*, *Implementing Industry 4.0: The Model Factory as the Key Enabler for the Future of Manufacturing* (2021) 229–258.

[44] V. Sdralia, C. Smythe, P. Tzerefos, and S. Cvetkovic, *Performance characterisation of the mcns docsis 1.0 catv protocol with prioritised first come first served scheduling*, *IEEE Transactions on broadcasting* **45** (1999), no. 2 196–205.

[45] G. D. Bibu and G. C. Nwankwo, *Comparative analysis between first-come-first-serve (fcfs) and shortest-job-first (sjf) scheduling algorithms*, .

[46] M. Pedemonte, P. Ezzatti, and Á. Martín, *Accelerating the min-min heuristic*, in *Parallel Processing and Applied Mathematics: 11th International Conference, PPAM 2015, Krakow, Poland, September 6-9, 2015. Revised Selected Papers, Part II*, pp. 101–110, Springer, 2016.

[47] X. He, X. Sun, and G. Von Laszewski, *Qos guided min-min heuristic for grid task scheduling*, *Journal of computer science and technology* **18** (2003), no. 4 442–451.

[48] Q. Tang, R. Xie, F. R. Yu, T. Chen, R. Zhang, T. Huang, and Y. Liu, *Distributed task scheduling in serverless edge computing networks for the internet of things: A learning approach*, *IEEE Internet of Things Journal* **9** (2022), no. 20 19634–19648.

[49] A. Kritikakou, F. Catthoor, V. Kelefouras, and C. Goutis, *A systematic approach to classify design-time global scheduling techniques*, *ACM Computing Surveys (CSUR)* **45** (2013), no. 2 1–30.

[50] B. B. Brandenburg and M. Gül, *Global scheduling not required: Simple, near-optimal multiprocessor real-time scheduling with semi-partitioned reservations*, in *2016 IEEE Real-Time Systems Symposium (RTSS)*, pp. 99–110, IEEE, 2016.

[51] A. C. Arpaci-Dusseau, *Implicit coscheduling: coordinated scheduling with implicit information in distributed systems*, *ACM Transactions on Computer Systems (TOCS)* **19** (2001), no. 3 283–331.

[52] M. A. Palis, J.-C. Liou, and D. S. L. Wei, *Task clustering and scheduling for distributed memory parallel architectures*, *IEEE Transactions on Parallel and Distributed Systems* **7** (1996), no. 1 46–55.

[53] H. Topcuoglu, S. Hariri, and M.-Y. Wu, *Task scheduling algorithms for heterogeneous processors*, in *Proceedings. Eighth Heterogeneous Computing Workshop (HCW'99)*, pp. 3–14, IEEE, 1999.

[54] W. Xiang, *A multi-objective aco for operating room scheduling optimization*, *Natural Computing* **16** (2017) 607–617.

[55] K. Al-Saqabi, S. Sarwar, and K. Saleh, *Distributed gang scheduling in networks of heterogenous workstations*, *Computer communications* **20** (1997), no. 5 338–348.

[56] G. C. Sih and E. A. Lee, *A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures*, *IEEE transactions on Parallel and Distributed systems* **4** (1993), no. 2 175–187.

[57] P. Ezzatti, M. Pedemonte, and Á. Martín, *An efficient implementation of the min-min heuristic*, *Computers & operations research* **40** (2013), no. 11 2670–2676.

[58] D. P. Vidyarthi and A. K. Tripathi, *Maximizing reliability of distributed computing system with task allocation using simple genetic algorithm*, *Journal of Systems Architecture* **47** (2001), no. 6 549–554.

[59] I. Gupta, M. S. Kumar, and P. K. Jana, *Efficient workflow scheduling algorithm for cloud computing system: a dynamic priority-based approach*, *Arabian Journal for Science and Engineering* **43** (2018), no. 12 7945–7960.

[60] K. Etminani and M. Naghibzadeh, *A min-min max-min selective algorihtm for grid task scheduling*, in *2007 3rd IEEE/IFIP International Conference in Central Asia on Internet*, pp. 1–7, IEEE, 2007.

[61] O. K. J. Mohammad and B. M. Salih, *Improving task scheduling in cloud datacenters by implementation of an intelligent scheduling algorithm*, *Informatica* **48** (2024), no. 10.

[62] T. Stützle and H. H. Hoos, *Max–min ant system*, *Future generation computer systems* **16** (2000), no. 8 889–914.

[63] B. A. Al-Maytami, P. Fan, A. Hussain, T. Baker, and P. Liatsis, *A task scheduling algorithm with improved makespan based on prediction of tasks computation time algorithm for cloud computing*, *IEEE Access* **7** (2019) 160916–160926.

[64] S. K. Panda and P. K. Jana, *Uncertainty-based qos min–min algorithm for heterogeneous multi-cloud environment*, *Arabian Journal for Science and Engineering* **41** (2016), no. 8 3003–3025.

[65] Q. Jiang, J. Xu, and Y. Chen, *A genetic algorithm for scheduling in heterogeneous multicore system integrated with fpga*, in *2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*, pp. 594–602, IEEE, 2021.

[66] H. Sabet, M. Gholami, and M.-R. Akbari, *An improved artificial bee colony algorithm for the multiple knapsack problem*, *Swarm and Evolutionary Computation* **41** (2018) 124–133.

[67] Y. Ding, X. Qin, L. Liu, and T. Wang, *Energy efficient scheduling of virtual machines in cloud with deadline constraint*, *Future Generation Computer Systems* **50** (2015) 62–74.

[68] C. Shyalika, T. Silva, and A. Karunananda, *Reinforcement learning in dynamic task scheduling: A review*, *SN Computer Science* **1** (2020), no. 6 306.

[69] V. Vimal, *Synergizing ai and multiprocessor systems for task scheduling in industry 4.0*, *IEEE Xplore* (2024).

[70] A. O. Al-Zaghameem, *An expressive approach to distributed applications dynamic adaptation*, *International Journal of Computer Science Issues (IJCSI)* **9** (2012), no. 4 57.

[71] Z. Samsudin and M. D. Ismail, *The concept of theory of dynamic capabilities in changing environment*, *International journal of academic research in business and social sciences* **9** (2019), no. 6 1071–1078.

[72] K. M. Newell, G. Mayer-Kress, S. L. Hong, and Y.-T. Liu, *Adaptation and learning: Characteristic time scales of performance dynamics*, *Human movement science* **28** (2009), no. 6 655–687.

[73] G. H. Alférez, V. Pelechano, R. Mazo, C. Salinesi, and D. Diaz, *Dynamic adaptation of service compositions with variability models*, *Journal of Systems and Software* **91** (2014) 24–47.

[74] S. M. Sadjadi and P. K. McKinley, *Act: An adaptive corba template to support unanticipated adaptation*, in *24th International Conference on Distributed Computing Systems, 2004. Proceedings.*, pp. 74–83, IEEE, 2004.

[75] Y. Wang and A. Rausch, *An approach to automatic adaptation of daisi component interfaces*, .

[76] J. R. Syed, *An adaptive framework for knowledge work*, *Journal of Knowledge Management* **2** (1998), no. 2 59–69.

[77] S. Taherizadeh, A. C. Jones, I. Taylor, Z. Zhao, and V. Stankovski, *Monitoring self-adaptive applications within edge computing frameworks: A state-of-the-art review*, *Journal of Systems and Software* **136** (2018) 19–38.

[78] O. Gheibi, D. Weyns, and F. Quin, *Applying machine learning in self-adaptive systems: A systematic literature review*, *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* **15** (2021), no. 3 1–37.

[79] E. Ilavarasan, P. Thambidurai, and R. Mahilmannan, *Performance effective task scheduling algorithm for heterogeneous computing system*, in *The 4th international symposium on parallel and distributed computing (ISPDC'05)*, pp. 28–38, IEEE, 2005.

[80] N. Galimyanova, *Experimental investigations of combined algorithms of branch and bound method and dynamic programming method for knapsack problems*, *Journal of Computer and Systems Sciences International* **47** (2008), no. 3 422–428.

[81] D. M. Abdelkader and F. Omara, *Dynamic task scheduling algorithm with load balancing for heterogeneous computing system*, *Egyptian Informatics Journal* **13** (2012), no. 2 135–145.

[82] L. Bendiaf, A. Harbouche, and M. A. Tahraoui, *Dynamic adaptation for independent task scheduling using dynamic programming in multiprocessor systems*, *Revue Nature et Technologie* **17** (2025), no. 1 09–16.

[83] C. Wu, Y. Wang, A. Zhao, and T. Qiu, *Research on task allocation strategy and scheduling algorithm of multi-core load balance*, in *2013 Seventh International Conference on Complex, Intelligent, and Software Intensive Systems*, pp. 634–638, IEEE, 2013.

[84] L. M. Bendiaf, A. Harbouche, A. M. Tahraoui, and F. Z. Lebbah, *An innovative task scheduling method using the knapsack algorithm in heterogeneous computing systems*, *Informatica* **48** (2024), no. 16.

[85] L. Lalami, S. Hanafi, and H. Elmaghraby, *A recursive constructive algorithm for the multiple knapsack problem*, *Computers & Operations Research* **43** (2014) 68–77.

[86] A. Fukunaga and R. E. Korf, *Bin completion algorithms for multicontainer packing, knapsack, and covering problems*, *Journal of Artificial Intelligence Research* **28** (2007) 393–429.

[87] A. Fukunaga, *A branch-and-bound algorithm for mkp with dominance and symmetry breaking*, *Computers & Operations Research* **37** (2010), no. 3 392–401.

[88] S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, 1990.

[89] D. Sitarz, *Multiple criteria dynamic programming approach to the multiple knapsack problem*, *European Journal of Operational Research* **252** (2016), no. 1 129–138.

[90] M. Hickman and T. Easton, *New facet-defining inequalities for the multiple knapsack problem*, *Operations Research Letters* **38** (2010), no. 6 539–543.

[91] M. Dell'Amico, S. Martello, and D. Vigo, *An arc-flow model for the multiple knapsack problem*, *Networks* **68** (2016), no. 2 109–117.

[92] J. V. de Carvalho, *Lp models for bin packing and cutting stock problems*, *European Journal of Operational Research* **141** (2002), no. 2 253–273.

[93] P. Detti, *New upper bounds for the multiple knapsack problem*, *Computers & Industrial Engineering* **101** (2016) 49–57.

[94] C. Chekuri and S. Khanna, *A ptas for the multiple knapsack problem*, *SIAM Journal on Computing* **31** (2001), no. 3 806–825.

[95] K. Jansen, *An efficient ptas for the multiple knapsack problem*, *Journal of Scheduling* **13** (2010), no. 5 493–500.

[96] H. Wang and Y. Xing, *An approximation algorithm for the multiple knapsack problem based on iterative filling*, *Applied Mathematics and Computation* **217** (2010), no. 6 2422–2429.

[97] V. Khutoretskii, S. Popov, and I. Gorbunov, *A 0.5-approximation algorithm for the multiple knapsack problem*, *Journal of Optimization Theory and Applications* **167** (2015) 345–357.

[98] H. Shah-Hosseini, *Solving the multiple knapsack problem using a population-based algorithm*, *Applied Soft Computing* **11** (2011), no. 4 3124–3132.

[99] F. Wei and J. Zhang, *Solving the multiple knapsack problem using an artificial bee colony algorithm*, *Engineering Optimization* **46** (2014), no. 12 1677–1694.

[100] Y. Liu, J. Hou, and W. He, *An artificial fish swarm optimization algorithm for the multiple knapsack problem*, *Computers & Industrial Engineering* **110** (2017) 191–202.

[101] D. Zouache, A. Moussaoui, and F. B. Abdelaziz, *A cooperative swarm intelligence algorithm for multi-objective discrete optimization with application to the knapsack problem*, *European Journal of Operational Research* **264** (2018), no. 1 74–88.

[102] L. Qin, F. Ouyang, and G. Xiong, *Dependent task scheduling algorithm in distributed system*, in *2018 4th international conference on computer and technology applications (iccta)*, pp. 91–95, IEEE, 2018.

[103] R. M. Sahoo and S. K. Padhy, *A novel algorithm for priority-based task scheduling on a multiprocessor heterogeneous system*, *Microprocessors and Microsystems* **95** (2022) 104685.

[104] H. Arabnejad, "List based task scheduling algorithms on heterogeneous systems – an overview." https://paginas.fe.up.pt/~prodei/dsie12/papers/paper_30.pdf, 2013. Accessed: 2024-05-14.

[105] S. AlEbrahim and I. Ahmad, *Task scheduling for heterogeneous computing systems*, *The Journal of Supercomputing* **73** (2017) 2313–2338.

[106] E. C. da Silva and P. H. Gabriel, *A comprehensive review of evolutionary algorithms for multiprocessor dag scheduling*, *Computation* **8** (2020), no. 2 26.

[107] S. Forrest, *Genetic algorithms*, *ACM computing surveys (CSUR)* **28** (1996), no. 1 77–80.

[108] S. Mirjalili and S. Mirjalili, *Genetic algorithm*, *Evolutionary algorithms and neural networks: Theory and applications* (2019) 43–55.

[109] A. Lambora, K. Gupta, and K. Chopra, *Genetic algorithm-a literature review*, in *2019 international conference on machine learning, big data, cloud and parallel computing (COMITCon)*, pp. 380–384, IEEE, 2019.

[110] J. Kennedy and R. Eberhart, *Particle swarm optimization*, in *Proceedings of ICNN'95-international conference on neural networks*, vol. 4, pp. 1942–1948, ieee, 1995.

[111] M. Clerc, *Particle swarm optimization*, vol. 93. John Wiley & Sons, 2010.

[112] D. Wang, D. Tan, and L. Liu, *Particle swarm optimization algorithm: an overview*, *Soft computing* **22** (2018), no. 2 387–408.

[113] E. Aarts, J. Korst, and W. Michiels, *Simulated annealing*, *Search methodologies: introductory tutorials in optimization and decision support techniques* (2005) 187–210.

[114] D. Bertsimas and J. Tsitsiklis, *Simulated annealing*, *Statistical science* **8** (1993), no. 1 10–15.

[115] I. Kucukkoc, *Milp models to minimise makespan in additive manufacturing machine scheduling problems*, *Computers & Operations Research* **105** (2019) 58–67.

[116] T. D. Braun, H. Siegal, N. Beck, L. L. Boloni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, et al., *A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems*, in *Proceedings. Eighth Heterogeneous Computing Workshop (HCW'99)*, pp. 15–29, IEEE, 1999.

[117] M. S. Arif, Z. Iqbal, R. Tariq, F. Aadil, and M. Awais, *Parental prioritization-based task scheduling in heterogeneous systems*, *Arabian Journal for Science and Engineering* **44** (2019), no. 4 3943–3952.