Towards Faster Combinatorial Search: Performance-Driven Workstealing Policy in YewPar

Hao Xie

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

A dissertation presented in part fulfilment of the requirements of the Degree of Master of Science at The University of Glasgow

31th August 2023

Abstract

Precise combinatorial search is indispensable for a wide range of applications, including constraint programming, graph matching, and computational algebra. Parallel computation serves as one of the vital means to accelerate combinatorial search. Workstealing stands as a crucial strategy for achieving parallel computation, where idle computational resources of a node exploit the residual tasks from other nodes, enhancing the overall performance. YewPar is an exemplary parallel framework tailored for combinatorial search. It offers an array of robust skeletons and policies, enabling developers to effortlessly parallelize their search algorithms. However, its workstealing strategy leans towards randomly selecting task-rich nodes for task theft, often leading to unnecessary overhead and latency. Our objective is to boost the execution speed of applications based on the YewPar framework by designing a novel workstealing strategy.

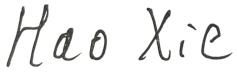
To this end, we devised and implemented a new framework termed "Performance-Driven Workstealing Policy", which dynamically adjusts the target node for task theft based on multiple performance parameters of each node. The core philosophy of this framework revolves around various strategies, such as stealing tasks from nodes with higher loads to approximate a load-balanced state, thereby shortening the time taken by the last worker to complete its task. By reducing unnecessary probing durations, the starvation time for workers without tasks is minimized, ultimately curtailing the total execution time of applications based on YewPar under parallel conditions.

Furthermore, we evaluated the enhanced YewPar with the improved Workstealing strategy. The assessment was conducted on a Beowulf cluster equipped with multi-core machines. The results indicate that the revamped YewPar, compared to its original version, achieves superior performance across varying node counts and workloads. Without compromising the search results, it can significantly reduce the execution time of search applications based on YewPar. In most scenarios, the improvements in its execution speed are quite remarkable.

Keywords: Combinatorial Search, Parallel Computation, Workstealing, YewPar, Performance-Driven Policy, Beowulf Cluster.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: Hao Xie Signature: 

Acknowledgements

It took months almost entirely to focus on this project and do this dissertation. As a student unfamiliar with the topic initially, I have done much work with effort and interest to gain knowledge regarding the YewPar framework, Combinatorial Search and Workstealing concepts related to the project.

So, I would like to thank my supervisor Blair Archibald, who helped me a lot in completing this project. He discussed the project with me every week and gave me a lot of advice, which benefited me greatly.

I would also like to thank my family for their love and support throughout my life. Also, my roommates and other friends, who encouraged me a lot during my project.

Contents

1 Introduction **5**

2 Background **7**

2.1 Modern Strategies in Distributed Concurrency 7

2.1.1 Workstealing 7

2.1.2 The Pitfalls of Locking Mechanisms and the Promise of Lock-Free Designs 9

2.2 Combinatorial Search 9

2.2.1 Parallel Combinatorial Search and YewPar 9

2.2.2 The Rationale for Designing and Employing a Novel Workstealing Strategy in YewPar 11

3 Design and implementation **12**

3.1 Overall Design 12

3.2 Performance Data Collection and Transmission 13

3.2.1 Load Status of All Workers on a Node 13

3.2.2 Collection of Residual Task Volume and Communication Latency . 16

3.3 Optimal Steal Target Calculation and Caching 18

3.4 Refresh Data and Provide Tasks Mechanism 19

4 Evaluation **22**

4.1 Search Applications 22

4.2 Experimental Setup 22

4.3 Isolated Performance Evaluation of Different Policies 23

4.3.1 Isolated Performance Evaluation of Maxclique 23

4.3.2 Isolated Performance Evaluation of NS-hivert 25

4.4 Simultaneous Performance Evaluation under Resource Contention 27

4.4.1 Simultaneous Performance Evaluation of Maxclique 28

4.4.2 Simultaneous Performance Evaluation of NS-hivert 29

4.5 Evaluation of Additional Performance Cost of Performance-driven Workstealing Policy Framework 30

5 Future Work & Conclusion **32**

5.1 Future Work 32

5.1.1 Automatic Adjustment of Parameters 32

5.1.2 Optimization for Performance Anomalies in Parallel Combined Searches 32

5.1.3 Substitute All YewPar Workstealing Policies 32

5.2 Conclusion 32

A Appendix **34**

A.1 Project description and Source code 34

A.2 Evaluation Data 34

Bibliography **42**

Chapter 1: Introduction

Precise combinatorial search is essential for a broad spectrum of applications, including constraint programming, graph matching, and computational algebra. Combinatorial problems are addressed by systematically exploring the search space, a process both theoretically and practically computationally challenging. Exact search, in particular, traverses the entire search space to offer a provably optimal solution. Conceptually, exact combinatorial search operates by generating and navigating through a (massive) tree representing potential solutions. Incorporating parallelism, on-the-fly tree generation, search heuristics, and pruning can diminish the execution time of an exact search. Due to the vast and highly irregular search trees, parallelizing exact combinatorial search poses significant challenges.

A framework known as YewPar[4] stands out as the first scalable parallel framework tailored for precise combinatorial search. YewPar aims to allow non-specialist users to benefit from parallelism, reuse parallel search patterns encoded as algorithmic skeletons, and execute across diverse parallel architectures.

Concurrently, with the proliferation of parallel computing and multi-core processors, effective task scheduling becomes increasingly paramount. A key feature of YewPar is its capability to parallelize search, accomplished by allowing multiple workers in a node to steal tasks from other nodes' task pools when their local task pools run dry. In YewPar, when a local task pool is empty, it randomly selects nodes for task theft. Workstealing, a widely researched and implemented parallel scheduling strategy, permits idle processors to "steal" tasks from their busier counterparts. However, many internal Workstealing schedulers, like YewPar, tend to adopt a somewhat random theft approach, often leading to unnecessary overhead, latency, and a vast amount of time spent in probing for tasks to steal. This often results in relative load imbalances across nodes, prolonging the overall completion time. While numerous improvements on Workstealing have emerged, many fail to adequately balance features such as lightweight design, decentralization, high performance, and generalizability.

Our objective is to enhance YewPar's performance by refining its Workstealing strategy. To this end, this paper introduces a new "Performance-Driven Workstealing Policy" framework. Its core philosophy centers on attempting to steal tasks from higher-load nodes to approximate a balanced load state, thereby reducing the time taken for the last worker to complete its task. Additionally, by minimizing unnecessary task probing and theft durations, the starvation time for task-less workers is reduced, ultimately shortening the runtime of YewPar-based applications in a parallel environment.

This framework boasts multiple features: it periodically monitors and transmits valuable data, calculates and caches the optimal node for task theft, and efficiently gathers and shares multiple performance metrics, such as node load status, remaining tasks per node, and inter-node communication latency, at a low cost using proprietary architecture and data processing algorithms. Through its in-house "Time-Optimized Workstealing Strategy" algorithm, it calculates the optimal theft node and caches it, refreshing periodically. Idle workers, if they fail to obtain tasks from the cached target, initiate an auxiliary refresh operation, assisting potential dormant refreshers in refreshing the optimal theft target cache. Collectively, these components enable the acceleration effects of the Performance-Driven Workstealing Policy.

Furthermore, we evaluated the YewPar enhanced with the refined Workstealing strategy. Assessments were conducted on a Beowulf cluster equipped with multi-core machines. Results revealed that the improved YewPar, under varying node counts and workloads, outperforms the original YewPar in terms of performance. Without compromising the search results, it can significantly reduce the execution time of search applications based on YewPar. In most scenarios, the improvements in its execution speed are quite pronounced.

Chapter 2: Background

2.1 Modern Strategies in Distributed Concurrency

2.1.1 Workstealing

Workstealing is a core concept in parallel programming. Its primary advantage lies in decentralized scheduling, where each processor autonomously manages its task queue, significantly reducing the overhead brought about by global synchronization. The idea of Workstealing can be traced back to the implementation of the Multilisp programming language in the 1980s and the work on parallel functional programming languages[7]. It has been employed in the scheduler of the Cilk programming language[6], Java's fork/join framework[16], .NET's Task Parallel Library[18], and Rust's Tokio runtime[24, 15], among many other parallel frameworks and libraries.

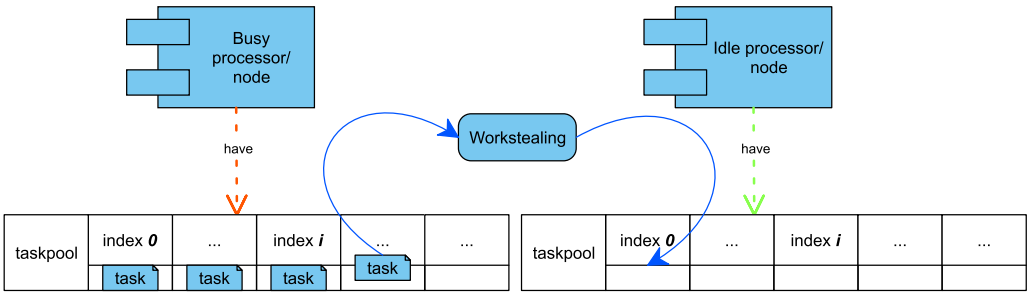


Figure 2.1: Workstealing example

As illustrated in 2.1, in the workstealing strategy, every processor within the system possesses its own queue of tasks awaiting execution. Each task consists of a series of instructions that need to be executed sequentially. During execution, a task might spawn new sub-tasks. These sub-tasks are preliminarily placed in the queue of the processor executing the parent task. When a processor's task queue becomes empty, it attempts to "steal" tasks from other processors' queues. Thus, Workstealing essentially schedules tasks to idle processors, ensuring that scheduling overhead only occurs when all processors are occupied[8].

In contrast to Workstealing stands the work-sharing strategy, another approach to dynamic multithreading scheduling. In work-sharing, newly generated tasks are immediately scheduled on a processor for execution. Compared to this, Workstealing reduces task migration between processors, as such migration doesn't occur when all processors are busy[7].

Algorithms like the randomized Workstealing algorithm proposed by Blumofe and Leiserson maintain multiple execution threads (asynchronous tasks) and schedule them on PP processors[7]. Each processor has a double-ended queue (deque) for storing threads, designating the two ends as "top" and "bottom". When a processor has a current thread to execute, it processes the instructions in the thread sequentially until encountering one of four "special" behaviors:

- A ”spawn” instruction creates a new thread. The current thread is placed at the bottom of the deque, and the processor starts executing the new thread.
- A ”stall” instruction temporarily halts the execution of its thread. The processor pops a thread from the bottom of its deque and starts its execution. If its deque is empty, it initiates workstealing.
- Some instructions might terminate a thread. The behavior in this case is the same as a stalling instruction.
- An instruction might activate another thread. The other thread is pushed to the bottom of the deque, but the processor continues executing its current thread.

Initially, a computation comprises a single thread assigned to a processor, while other processors start off idle. Any idle processor commences the actual Workstealing process, involving the following steps:

- It randomly picks another processor.
- If the other processor's deque is non-empty, it pops the topmost thread from the deque and begins its execution;
- Otherwise, the process is repeated.

However, strategies like this random theft often incur considerable overhead in complex multi-node environments. Such overhead primarily originates from the ”thief” randomly probing nodes in the cluster to locate a ”victim”. Given the uneven distribution of cluster scales, this problem becomes more pronounced, resulting in excessive system messaging and prolonged starvation periods for ”thieves” due to multiple theft failures and network latencies. While optimization attempts concerning Workstealing have never ceased, such as establishing fixed ”matchmaker” nodes and notifying them of the current states of nodes with excess tasks and those without, to enable ”matchmaker” nodes to match task-starved nodes with those abundant in tasks[23], the maintenance of these ”matchmaker” nodes and the waiting time for sending query requests to them during task starvation remain significant overheads. Additionally, they only match nodes with extra tasks to those without, without optimizing based on various performance parameters of the nodes, which might lead to frequent unnecessary thefts. Strategies specifically designed for Multicore Event-Driven Systems[12], though considering the characteristics of event-driven programming and effectively increasing parallelism in such environments, still introduce additional overhead through task randomization and delayed theft strategies and might not be well-suited for non-event-driven applications.

To date, no strategy has emerged as a universally efficient, low-overhead, lightweight, cross-platform solution. Workstealing remains primarily in the phase of targeted optimization for various application environments. Our task theft scheme design, based on node performance data, mainly employs performance data such as worker load conditions, remaining tasks of each node, and communication delays between nodes. This data can be effortlessly acquired in other frameworks employing the Workstealing strategy, ensuring excellent portability and universality. It can be easily ported to platforms and frameworks other than YewPar, and the design also possesses lightweight characteristics.

2.1.2 The Pitfalls of Locking Mechanisms and the Promise of Lock-Free Designs

Optimization in multithreading is a burgeoning field, where locking mechanisms have been primarily designed to address inconsistencies in concurrent operations. Edward A. Lee once posited that lock mechanisms are among the primary issues in multithreading[17]. Locks can induce thread waiting, leading to performance degradation. In high-concurrency scenarios, such waiting can manifest as severe performance bottlenecks. Furthermore, when multiple threads vie for the same lock, they enter a state of contention. Such a state not only diminishes performance but can also induce thread starvation, wherein certain threads may perpetually be denied lock access. Lee emphasized that while various techniques can alleviate these issues, the ultimate solution lies in seeking alternative programming methodologies and models to multithreading, with lock-free programming emerging as a promising avenue.

Currently, there are numerous exemplary implementations of lock-free designs. For instance, there exists a lock-free concurrent queue algorithm that facilitates multiple threads to access and modify the queue concurrently without necessitating locks[21]. Moreover, a dynamic hash table design based on a lock-free paradigm has been introduced. Owing to the elimination of lock overheads, this design offers stellar performance in high-concurrency settings[20].

This paper draws inspiration from various lock-free designs, introducing lock-free data structures and algorithms. Our Performance-Driven Workstealing Policy framework, while ensuring data accuracy, circumvents the performance implications of locks. To a certain extent, this ensures the framework's lightweight nature and high performance.

2.2 Combinatorial Search

In computer science and artificial intelligence, combinatorial search refers to a method employed to find solutions within a designated search space. Classic combinatorial search challenges encompass problems like the Eight Queens puzzle and assessing actions in games with extensive game trees, such as Othello or Chess. This type of search is particularly apt for problems where the solution space is vast, making it infeasible to enumerate all possible solutions. Certain algorithms guarantee the discovery of the optimal solution, while others may only yield the best solution found within the explored state space. Through branch and bound techniques or by adopting heuristic approaches, combinatorial search can effectively reduce the search space, facilitating a swifter solution discovery.

2.2.1 Parallel Combinatorial Search and YewPar

With the evolution of modern computer hardware, especially the proliferation of multi-core processors and distributed systems, leveraging parallelism to expedite combinatorial search has emerged as a research hotspot. The aim of parallel combinatorial search is to decompose the search space into multiple segments, enabling concurrent execution across various processing units, thereby accelerating solution discovery.

Parallel tree search can be categorized[13] as follows:

1. Parallel Node Processing focuses on parallelizing branching/boundary operations, as illustrated by boundary calculations for the Flowshop problem on GPUs[14].

2. Space partitioning, where parallel workers speculatively explore subtrees of the search tree. Even though subtrees are explored independently, knowledge such as improved boundaries is typically shared amongst workers to enhance performance.

3. Combinatorial methods run competitive searches in parallel, usually employing diverse heuristics or boundary approaches. Hybrid methods combining these strategies are also employed.

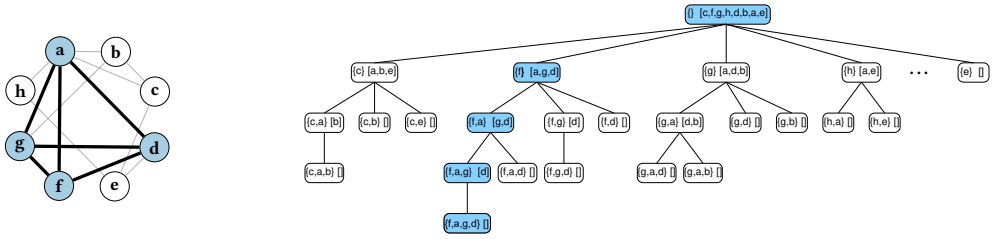


Figure 2.2: Maximum clique instance in YewPar. Input graph with clique $\{$a,d,f,g$\}$ to the left and corresponding search tree to the right. Each tree node displays the current clique and a list of candidate vertices (in heuristic order) to extend that clique. Reproduced from [4]

YewPar stands as a parallel framework explicitly designed for combinatorial search. It offers a suite of robust tools and strategies, allowing developers to effortlessly parallelize their search algorithms. YewPar's primary attributes are its flexibility and scalability, equipping it to handle a wide range of intricate search scenarios. Within YewPar, work-stealing serves as the central strategy for task scheduling, enabling processors to steal tasks from others when their local task queues are empty. This ensures all processors remain occupied, thereby optimizing overall performance.

Performance Anomalies

Search algorithms rely on sorting heuristics to swiftly identify useful nodes, such as target nodes in decision problems or a strongly constrained node in branch and bound optimization problems. To capitalize on these heuristics, searches proceed in a left-to-right order (across all tree depths). Therefore, a sequential search has complete information about the search tree, obtaining insights from all nodes on its left, and the search remains deterministic. Parallel search tentatively explores subtrees without the full left-side information and might benefit from a right-to-left information flow. However, this speculation implies that parallel searches might undertake more work than their sequential counterparts. Consequently, parallel searches are notoriously recognized for performance anomalies[9]. Detrimental anomalies occur when the runtime on w worker threads surpasses that on $w - 1$ threads. Here, the extra work might outweigh the benefits of added computational resources, or the additional computational resources might disrupt the search heuristics. Acceleration anomalies signify super-linear acceleration, often attributed to the information flow from right to left, which allows for more pruning than in sequential searches, thereby reducing the overall workload. The presence of anomalies complicates the parallel performance prediction for speculative search applications. Yew-Par aims to avoid detrimental anomalies while permitting acceleration anomalies; [2] reports a specialized search framework that carefully controls anomalies to offer replicable performance guarantees.

2.2.2 The Rationale for Designing and Employing a Novel Workstealing Strategy in YewPar

YewPar employs Workstealing as its pivotal distributed parallelization scheduling strategy, but it utilizes a biased random stealing approach[5]. Idle workers attempt to steal tasks from other nodes randomly until they secure a valid task, then lock onto that node for stealing until they can no longer pilfer tasks from it. Only then do they randomly select another node for theft. This method might induce unnecessary overheads and idle computational resources, especially amidst uneven task distributions and when nodes possess imbalanced computational assets. Given the characteristics of combinatorial searches, there might be a vast disparity in the number of tasks across nodes and a significant execution time variance between tasks. This suggests that the random stealing approach is more likely to yield prolonged starvation times for idle workers, rendering it a suboptimal choice.

For optimal utilization of computational capabilities across node cores and to curtail redundant communication overheads, YewPar necessitates a more intelligent Workstealing strategy. The proposed Performance-Driven Workstealing Policy, tailored to node performance, addresses this issue. By economically evaluating each node's performance, such as the average task execution time, this new strategy employs a stealing algorithm designed around performance parameters to precisely and dynamically update the optimal stealing target, thereby enhancing overall performance.

Chapter 3: Design and implementation

3.1 Overall Design

The Performance-Driven Workstealing Policy framework designed in this study encompasses:

- Scheduler Channel: Primarily responsible for monitoring, gathering statistics, and transmitting the load status of all workers within the node.
- Performance Monitor: Primarily tasked with collecting various performance data, transmitting processed local performance metrics, and computing and caching the ID of the optimal node for task stealing.
- Performance Policy: Mainly engaged in obtaining the optimal stealing target from the Performance Monitor when the local task pool is empty.

The relationship between these components is depicted in 3.1.

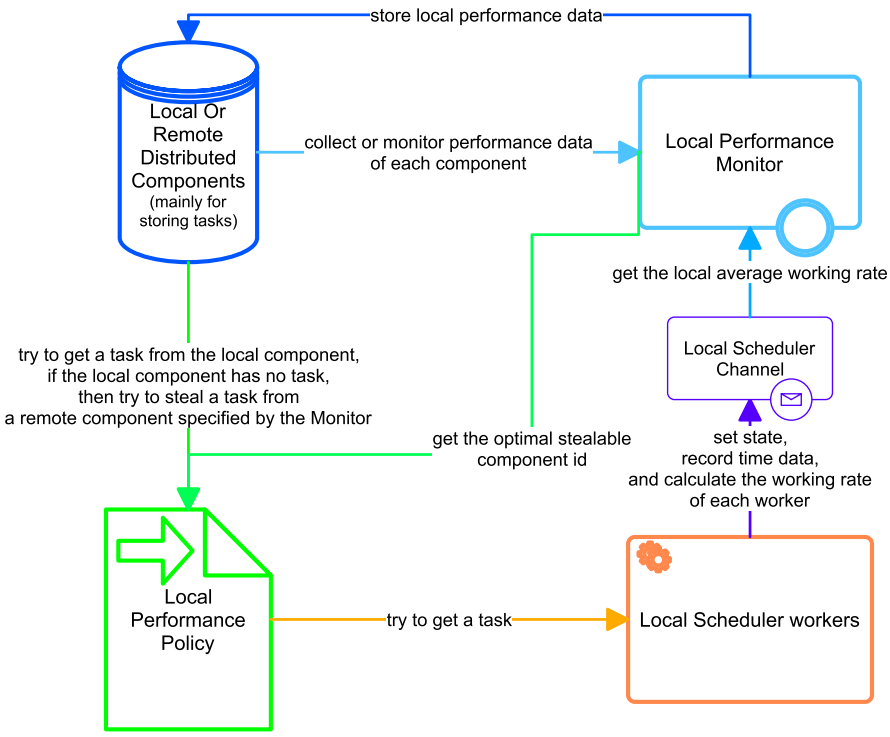


Figure 3.1: Performance-Driven Workstealing Policy Architecture

Among them, the Performance Monitor chiefly handles the gathering and transmission of performance metrics and calculates and caches the ID of the optimal node for task stealing. The distributed component is responsible for storing task pools and performance data accessible both remotely and locally, offering access and modification operations through the

HPX framework's action. The Scheduler Channel provides an interface allowing workers to update their current load status and compute their load data. Workers are tasked with obtaining executable tasks from the Performance Policy. Meanwhile, the Performance Policy is charged with drawing tasks from the local task pool or, after obtaining the optimal stealing target node from the Performance Monitor, stealing tasks from its task pool.

3.2 Performance Data Collection and Transmission

Data collection encompasses three aspects: load status across nodes, remaining task volume on each node, and the time each node takes to retrieve tasks from the task pool. These parameters will provide robust data support for subsequent optimal stealing target calculations. Furthermore, to ensure the collection of more effective data at a lower cost, this paper meticulously designs and optimizes all three aspects.

3.2.1 Load Status of All Workers on a Node

Initially, the YewPar's System Stack appears as illustrated in 3.2 [3].

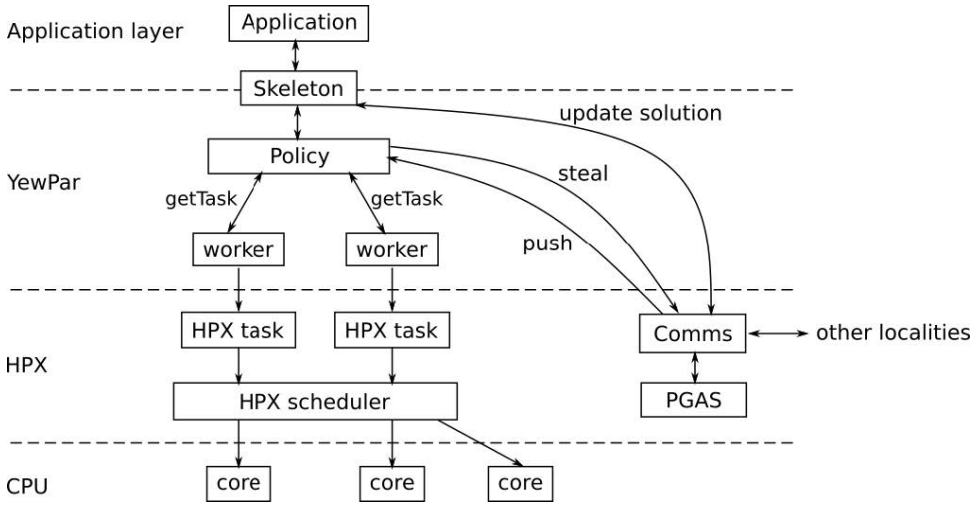


Figure 3.2: YewPar System Stack, Reproduced from [4]

Within this, the Policy is responsible for being invoked by workers, either attempting to fetch tasks from the local pool or stealing tasks from the task pool of other nodes. The actual task execution is handled by various YewPar workers. Thus, to gauge the load status of each node without relying on underlying system functions, a straightforward and effective method is to assess the load status of workers on each node. This paper analyzes the workflow of workers under the YewPar framework, as shown in 3.3.

It is evident that each worker, from creation to termination, exists within a loop comprising three possible stages:

1. Invoking the getWork function from Policy to acquire tasks;
2. If tasks are acquired, they are executed;
3. If no tasks are acquired, the worker sleeps for a duration. This duration increases with each unsuccessful task retrieval attempt and resets upon a successful retrieval.

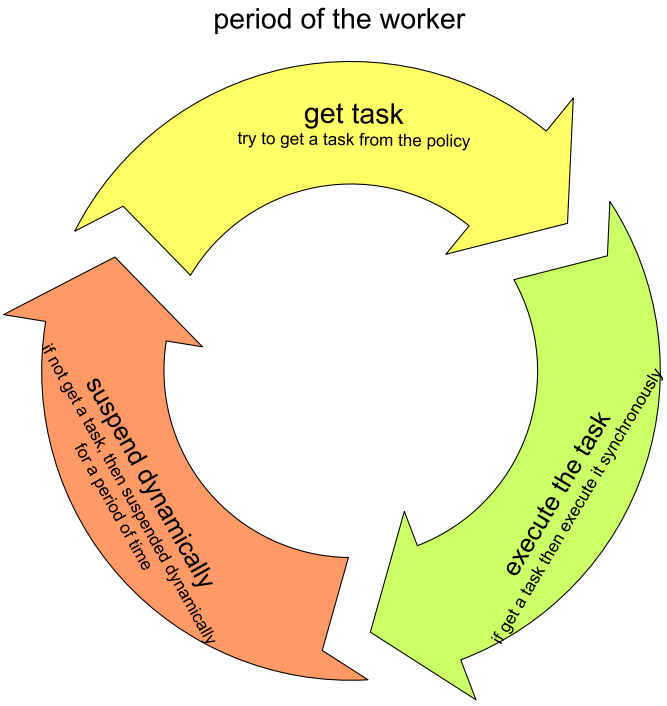


Figure 3.3: Period of the Worker

Among these stages, the first is invariably experienced in every cycle, while the second and third stages are mutually exclusive, with only one transpiring per cycle.

To analyze the actual load of a worker, focus must be placed on the second stage, since it represents the phase where the worker genuinely processes tasks. The time consumed during this phase represents the effective load duration, while the time consumed in other phases can be categorized as idle time, given that no tasks are executed, rendering it ineffective load time.

Considering that the original cycle might not necessarily undergo the second phase, to prevent the update of redundant data, this paper redefines the lifecycle judgment of the worker. The conclusion of one task execution marks the commencement of a new cycle, while the initiation of a task signals the end of a cycle. This ensures that each cycle invariably undergoes the second phase, which is the task execution phase.

Building upon this theoretical foundation, we have designed a comprehensive data collection scheme as depicted in 3.4.

By inserting probing code at the beginning and end positions of task execution (which, upon execution, momentarily halts the current process and proceeds with probe-related operations), a worker can update its status to the Scheduler Channel module (a channel dedicated to status transmission) before and after executing a task. In this manner, the worker can compute and update its load data in real-time, supplying the Performance Monitor with the requisite data for further calculations.

The Scheduler Channel employs a specific data structure for data aggregation. To circumvent the overheads associated with lock usage, it adopts a lock-free design strategy as discussed in 2.1.2. This design aims to mitigate contention among workers when collecting load data.

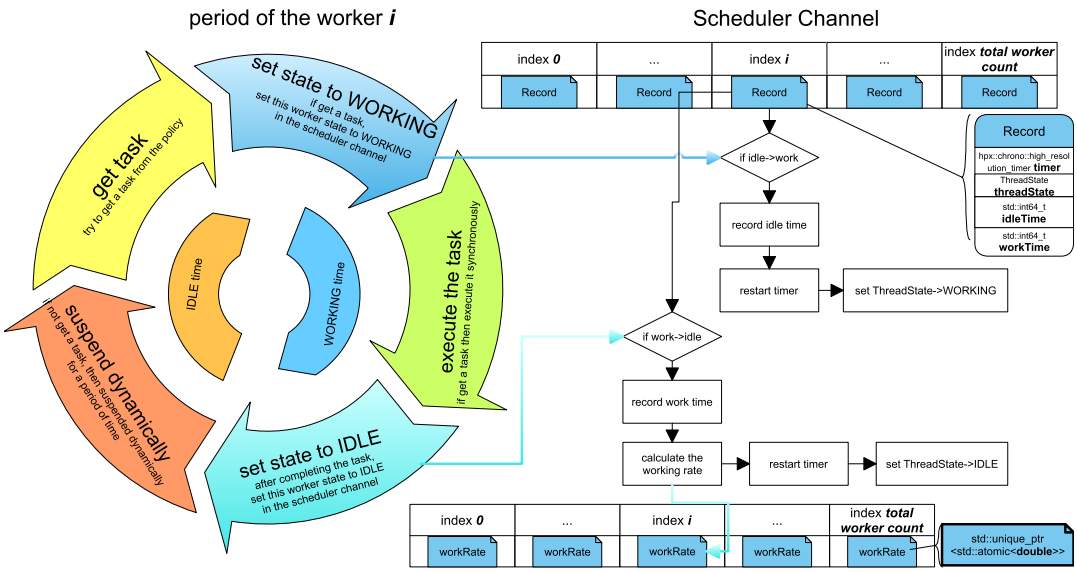


Figure 3.4: Collect Worker State to Calculate the Workload

This data structure principally consists of two arrays, each with a length equivalent to the number of local workers:

1. Record Array: This array stores distinct Record data structures. Each Record encompasses:

   - A timer, denoted as 'timer'
   - The current status, termed 'threadState'
   - Idle time for the current cycle, labeled 'idleTime'
   - Working time for the current cycle, named 'workTime'

   Such a design allows workers to log their durations under various states. As each worker corresponds to an individual record, it avoids the need for locks, thereby enhancing performance.

2. workRate Array: This array retains the worker load rate data, computed through a specific algorithm. Each 'workRate' is an atomic double type wrapped by 'unique_ptr'. Although both the worker and Monitor might access this data concurrently, the fundamental datatype nature of 'workRate' ensures data integrity in a lock-free environment using the atomic type.

The load data collection here primarily consists of two workflows:

- During the "set state to WORKING" phase, the worker primarily updates its idle time for this cycle in the channel and switches its state to WORKING.
- During the "set state to IDLE" phase, the worker primarily updates its working time for this cycle in the channel. Based on the idle and working time, along with historical workRate data, it calculates the current workRate and switches its state to IDLE.

During the "set state to IDLE" phase, the algorithm to compute the latest workRate is presented in 3.1:

$$\begin{aligned}\text{workRate} = \Bigg(&\ln\left(2.72 + \frac{\text{workTime}}{\text{workTime} + \text{idleTime}}\right) \\ &\times \ln\left(2.72 + (\text{workTime} + \text{idleTime})\right) \times 0.65\Bigg) \\ &+ \text{workRate} \times 0.35 \end{aligned} \tag{3.1}$$

The purpose of the formula is to gauge whether the current worker is busy and whether its computing speed is relatively slow. It comprises three main components:

1. $\ln\left(2.72 + \frac{\text{workTime}}{\text{workTime}+\text{idleTime}}\right)$: Within each cycle, if the duration of the WORKING state relative to the IDLE state is greater, it indicates a substantial or complex task load, necessitating more time for task execution. Conversely, a prolonged task starvation time signifies significant time wastage on unproductive load, such as speculative task stealing. Given that time data is measured in microseconds and exhibits substantial variability, a logarithmic function is employed to temper data sensitivity, and the term $(2.72 + ...)$ ensures a positive outcome.

2. $\ln\left(2.72 + (\text{workTime} + \text{idleTime})\right)$: The prior ratio doesn't adequately reflect the current worker's execution speed. There might be cases where the task load is minimal, but due to a slow execution speed, the actual load remains significant. By calculating the total duration for the current cycle and multiplying it by the previously computed ratio, one can obtain a practical load scenario for the cycle. The logarithmic function, given its microsecond-level granularity, is also employed here to reduce data sensitivity.

3. $(... \times 0.65 + \text{workRate} \times 0.35)$: Refreshing data might occur after intervals. Retaining historical load data as a reference, to prevent misjudgment due to short-term fluctuations, is essential. Since the current load rate data structure is optimized for memory efficiency and doesn't maintain structures like arrays for historical data, an exponential smoothing technique, as referenced in [11], is adopted to balance current and historical workRate data. The coefficients 0.65 and 0.35 were determined through extensive cluster environment testing to make the data more smooth and informative.

In the end, the workRate data computed for each worker is aggregated and divided by the number of workers to derive the average load data. This processed data is a lightweight double type, which is then passed to the Performance Monitor. The monitor submits this local workRate data to the local distributed component, facilitating rapid queries by other nodes.

3.2.2 Collection of Residual Task Volume and Communication Latency

Performance data also requires the collection of residual task volume and acquisition time for each node, as illustrated in 3.5.

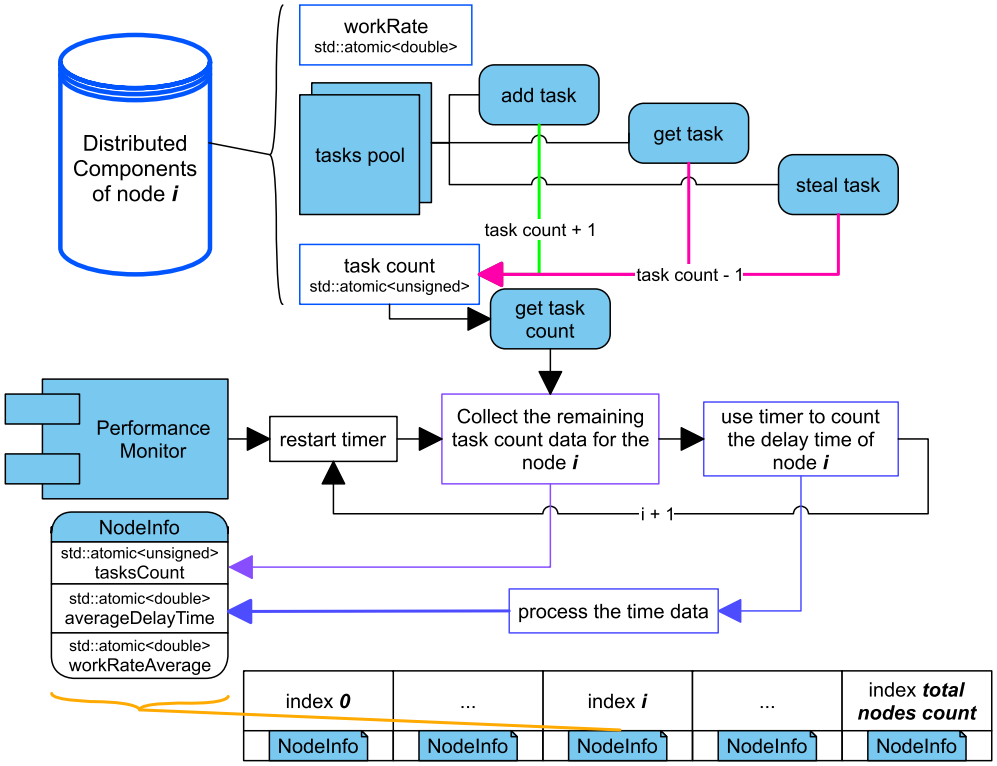


Figure 3.5: collect tasks count and delay time

The distributed component in HPX comprises a task pool and the number of residual tasks. The count of the residual tasks, introduced in a later phase for the needs of this framework, employs a lightweight atomic type to ensure data integrity. Originally, the task pool had three distributed operations to modify it: adding tasks, retrieving tasks, and stealing tasks. To maintain the accuracy of the residual task count, updates are made during these operations as depicted in 3.5. Within each node's Performance Monitor, a routine will traverse and invoke the get_task_count interface of each node's distributed component to obtain the residual task count.

To quantify the communication delay between the local node and other nodes, which represents the time required to retrieve data between distributed components, the delay in obtaining the residual task count from each node is measured. This process, while fetching distributed component data, also captures communication delay without incurring additional IO overhead.

It's worth noting that the communication delay also subtly reflects the busyness of the node's thread pool. Since the task count uses atomic types, an active thread pool may occasionally block other modification operations, resulting in increased time to retrieve the task count.

Given that the obtained communication delay is a temporal datum, which could exhibit significant values and fluctuations, it requires further processing before storage. The processing formula is:

$$\begin{aligned} \text{averageDelayTime}_i = {} & \ln\left(2.72 + \text{delayTime} \times \text{worker\_count}\right) \times 0.65 \\ & + \text{nodeInfoVector}[i]\text{->averageDelayTime} \times 0.35 \end{aligned} \tag{3.2}$$

Here, i represents the identifier for each node. As node identifiers remain consistent within the HPX framework, they can serve as array indices for easy node location. delayTime denotes the communication delay, and worker_count is the count of workers on the node. Firstly, the formula multiplies the delay time by the local worker count to reflect the node's overall communication overhead. Similar to 3.1, logarithmic functions and exponential smoothing are applied to reduce data sensitivity, retaining some impact from historical data, making the results smoother and more informative.

After processing, the data is stored. The storage data structure also adopts a lock-free design, as shown in 3.5. It's an array with a length equal to the total number of nodes, with index numbers corresponding to node identifiers. Stored within the array is a structure called NodeInfo, which contains atomic-type node load rates, residual task counts, and average communication delays for the corresponding node.

3.3 Optimal Steal Target Calculation and Caching

Upon acquiring performance data such as node load status, residual task volume on each node, and communication latency through the performance collection mechanism in the Performance Monitor, the optimal steal target node is calculated using these data. The node's ID is then cached for subsequent retrieval by the Performance Policy. The overall process is illustrated in 3.6.

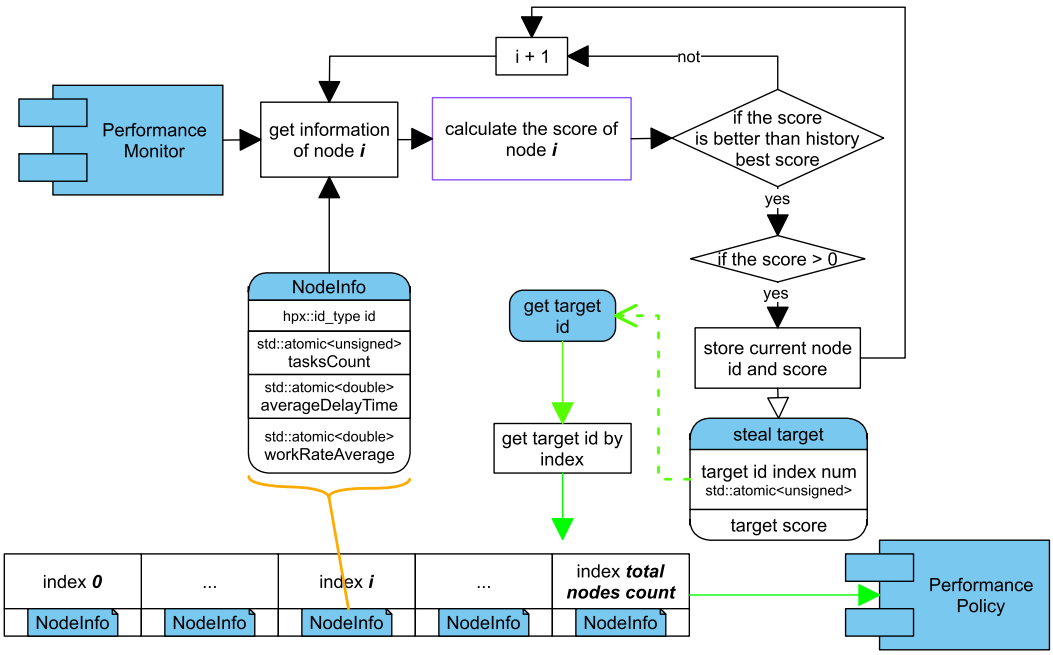


Figure 3.6: refresh the optimal steal target

The array data structure has been introduced earlier in 3.2.2.

The procedure starts by iterating through the NodeInfo data structure of each node to calculate the steal-worthiness score. A higher score indicates a higher worthiness for task stealing. Its formula is:

$$\begin{aligned} \text{score}_i = &\max\left(\text{nodeInfoVector}[i]\text{->workRateAverage}, 0.0001\right) \\ &\times \text{nodeInfoVector}[i]\text{->tasksCount} \\ &- \text{nodeInfoVector}[i]\text{->averageDelayTime} \end{aligned} \tag{3.3}$$

The core idea behind this formula is to reduce the task volume of the most heavily-loaded nodes while also minimizing the time wasted in task stealing. The formula can be broken down into:

1. Using $\max\left(\text{nodeInfoVector}[i]\text{->workRateAverage}, 0.0001\right)$ to avoid interference from negligibly small data in computations.
2. Multiplying the node's load rate by its residual task count, $\times\text{nodeInfoVector}[i]\text{->tasksCount}$, provides an estimate of the node's total load.
3. Subtracting the communication latency, $-\text{nodeInfoVector}[i]\text{->averageDelayTime}$, ensures minimal time wastage during task stealing.

A larger result suggests that the node is more deserving of task stealing because it has a larger workload and will require more time to process residual tasks. Moreover, its communication latency is smaller, meaning less time is wasted during the task-stealing process.

After iterating through all nodes, an optimal node target index is typically identified and cached. When the Performance Policy needs to retrieve the optimal steal target's id_type, it can do so through a provided query interface. This design is partly because HPX's id_type is not a basic data type and cannot be atomized. However, the corresponding array index number can be atomized. To maintain a lock-free design and avoid resource wastage due to prolonged blocking, the refresh phase only modifies the stored index number. During the query phase, the id_type of the optimal target node is retrieved using the index number. This ensures that all id_type values are only read, not modified, guaranteeing data safety in a lock-free environment.

The worthiness of the selected optimal steal target is judged based on:

- Whether there are any residual tasks left to steal from the target.
- Whether the cost of stealing outweighs the benefits, i.e., if the time spent on stealing tasks exceeds the time saved by assisting the target node in completing tasks.

Under such circumstances, the score is often less than or equal to zero. Hence, an additional condition is set to not cache the optimal steal target if the score is less than or equal to zero. In this case, under the Performance Policy, the worker will attempt to retrieve the optimal steal target again. If unsuccessful, it will enter a brief sleep mode to avoid wasting computational resources with frequent attempts.

3.4 Refresh Data and Provide Tasks Mechanism

After establishing a mechanism that spans from data collection to calculating and caching the optimal steal target, the next challenge to address is determining the frequency of data

refresh. Although the overhead of each data refresh is relatively small, YewPar's workers are designed to fully utilize local computational resources. Therefore, when workers are busy, too frequent refreshes can interrupt their operations, potentially impacting YewPar's performance. On the other hand, infrequent refreshes might reduce data reliability due to outdated information. This can result in providing inaccurate data to other nodes or relying on stale cached data from other nodes. In the worst-case scenario, this might lead to stealing tasks from already starved nodes, exacerbating imbalances.

To address this issue, this paper proposes a bidirectional dynamic refresh mechanism. This mechanism consists of two refresh methods: an automatic dynamic refresh method and an auxiliary refresh method facilitated by the Performance Policy. The general workflow is depicted in 3.7.

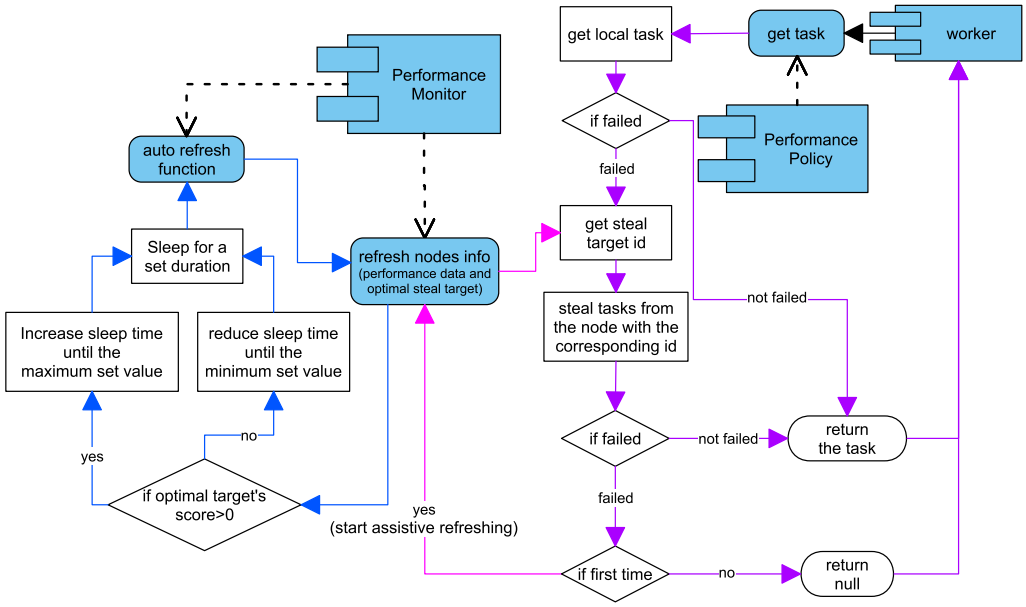


Figure 3.7: refresh data and provide tasks mechanism

On the left side of the diagram, an automatic dynamic refresh task is deployed within the Performance Monitor. At its core, this task is iterative, adjusting the timing of the next refresh based on the results of the previous one. If the refresh results are suboptimal, it's likely that some workers are still starved. In such cases, the subsequent refresh time will be significantly shortened. Conversely, if results align with expectations, indicating that most workers are likely to successfully acquire tasks, the time until the next refresh will be gradually extended to minimize unnecessary interruptions. Currently, the optimality of the refresh is primarily judged by the score. If the score is less than or equal to zero, it is considered suboptimal. According to 3.3, this is likely due to a scarcity of tasks in the pool or excessive time taken to obtain tasks. Under such conditions, the cost of task stealing might outweigh the benefits.

It's worth noting that while dynamically adjusting the refresh time yields noticeable benefits in practice, upper and lower limits should still be imposed. A lack of an upper limit might lead to using stale data, resulting in the caching of suboptimal steal targets. Without a lower limit, frequent refreshes could potentially disrupt the performance of workers with ongoing tasks or impact system-level scheduling. Through extensive experimentation, a range has been identified that minimizes impact on system performance while ensuring relative data

timeliness. Furthermore, incorporating algorithms like exponential smoothing retains the influence of historical data, ensuring that even slightly delayed data accurately reflects the overall status of nodes.

However, there's still room for improvement and optimization in the aforementioned design. Thus, as shown on the right side of 3.7, an auxiliary refresh mechanism facilitated by the Performance Policy has been introduced. The essence of the Performance Policy is a task retrieval strategy. It prioritizes acquiring local tasks since doing so allows workers to quickly obtain and execute tasks, minimizing resource wastage. If local tasks are unavailable, the policy will first query the Performance Monitor's cached optimal steal target id. While this typically results in a successful task steal, there are scenarios where very few tasks remain across all nodes, leading to occasional unsuccessful steals. If the automatic refresh mechanism is dormant and unable to promptly adjust the refresh rate, the Performance Policy, when invoked by a worker, assumes the role of an auxiliary refresher. After the data refresh is complete, another attempt is made to steal tasks based on the new target. If this too fails, it's highly probable that very few tasks are available across nodes. To conserve communication resources, the worker will then enter a sleep cycle to reduce resource consumption.

In practice, the combined effect of both mechanisms has been observed to significantly enhance system performance.

Chapter 4: Evaluation

4.1 Search Applications

We evaluated the performance of the improved workstealing strategy on representative search applications and samples for the two types of searches initially present in YewPar, as shown below:

- Enumeration: Unbalanced Tree Search (UTS) dynamically constructs synthetic irregular tree workloads based on a given branching factor, depth, and random seed[22]. Number Semigroups (NS) calculates how many number semigroups exist for a specific genus[10]. A number semigroup S is a residual set of natural numbers that includes 0 and is closed under addition; the genus of S is the size of its complement.
- Optimisation: Maximum Clique (MaxClique) identifies the largest clique in a given graph, i.e., the largest set of pairwise adjacent vertices. The 0/1 Knapsack problem determines the best combination of items, each with a profit and weight, to place in a container, ensuring maximum profit within a given weight constraint. The Traveling Salesman Problem (TSP) finds the shortest possible round trip through N cities.

To control variables, the evaluation utilized the Maxclique-16 and NS-hivert applications that come with the hpx1.8 branch of YewPar without any modifications. However, at runtime, we employed both the modified Performance-Driven Workstealing Policy and the original hpx1.8 branch's DepthPool Policy for comparison. The baseline implementations of MaxClique[19] and NS[10] employed publicly advanced algorithms. These sequential C++ implementations were provided by domain experts. A comprehensive description of the applications and instances can be found in [1].

For the Maxclique-16 application, the larger dataset brock800_2.clq was used to prolong runtime and minimize comparative errors. We adopted the Depth-Bounded skeleton, which is more suitable for the Maxclique-16 application and can utilize the Performance-Driven Workstealing Policy. We set the parameter $d = 2$ (spawn-depth: Depth in the tree to spawn at) as, in practice, we found that when d is set to 2, it offers superior performance and stability compared to other parameters.

For NS-hivert, the Budget skeleton was employed, being more suitable for the NS-hivert application and compatible with the Performance-Driven Workstealing Policy. Parameters were set as $b = 10^6$ (backtrack-budget: Number of backtracks before spawning work) and $g = 47$ (genus: Depth in the tree to count until) to ensure accurate results while increasing the workload to prolong runtime and thus reduce statistical errors.

4.2 Experimental Setup

The code used for the experiments, along with installation instructions and detailed experimental specifics (including the commands used and data obtained), can be found in the Appendix A.

We conducted measurements on the program's execution across as many as 20 machines. Each machine was equipped with a dual 8-core Intel Xeon E5-2640v2 2GHz CPU (without hyper-threading), 64GB RAM, and was running the Ubuntu 22.04.2 LTS operating system. Before testing, each machine maintained a load within 1%. Similar to the original hpx1.8 branch of YewPar, we reserved one core for task management for HPX (version 1.8), thus utilizing 15 working threads out of the 16 cores available.

It's essential to note that, due to the non-determinism caused by pruning, the search for alternative effective solutions, and the high randomness of the original YewPar's workstealing scheme, performance analysis in parallel search is highly challenging. This can potentially lead to performance anomalies, as referenced in 2.2.1, manifesting as super-linear speedups/slowdowns. To control these factors, we ran each program multiple times under various node quantities (set to 10 times) and reported the aggregated statistical data.

4.3 Isolated Performance Evaluation of Different Policies

To precisely assess the performance difference between the YewPar based on the Performance-Driven Workstealing Policy and the original version under an idle environment, we conducted individual test evaluations on two applications when all nodes were in an idle state. To minimize test errors, we opted to alternate between the modified and original versions of the application. That is, after running the program based on the modified Policy framework, we immediately ran the program based on the original Policy, followed by another run of the program based on the modified Policy framework. This cycle was repeated until all test rounds were completed. During these tests, the running program had exclusive access to all computational resources.

4.3.1 Isolated Performance Evaluation of Maxclique

The results for the run time of the Maxclique program based on the Depth-Bounded skeleton, averaged over ten runs, are presented in 4.1. The results are divided into two sections: run-time evaluation and a comparison of the performance improvement of the Performance-Driven Workstealing Policy over the original DepthPool Workstealing Policy.

The formula for calculating the speed improvement is given by

$$\text{Speedup Percentage} = \left(\frac{T_{\text{DepthPool}} - T_{\text{Performance-Driven}}}{T_{\text{Performance-Driven}}} \right) \times 100\% \tag{4.1}$$

where $T_{\text{DepthPool}}$ is the run-time of the application based on the original YewPar's DepthPool Workstealing Policy, and $T_{\text{Performance-Driven}}$ is the run-time of the application based on the Performance-Driven Workstealing Policy.

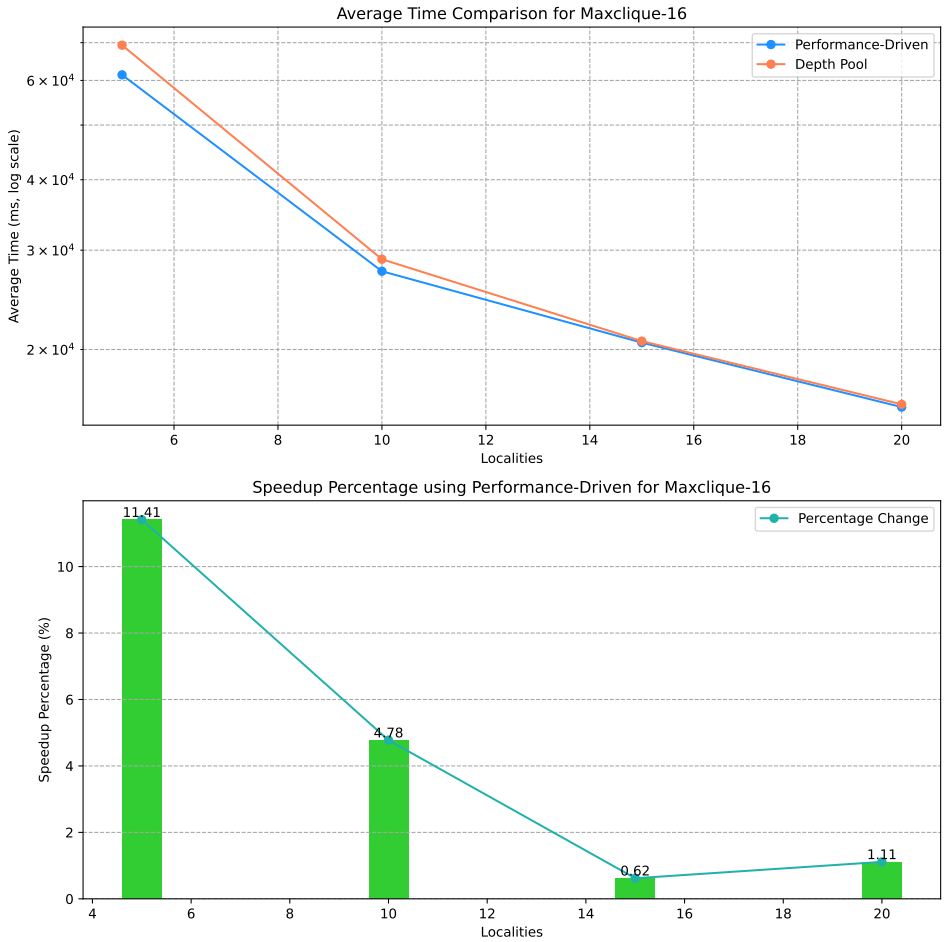


Figure 4.1: isolated time comparison for Maxclique-16

From the graph, it is evident that, when run in isolation and across varying node counts, the YewPar based on the Performance-Driven Workstealing Policy consistently outperforms the YewPar based on the original DepthPool Workstealing Policy, completing all tasks in a shorter time.

However, from the graph, it's evident that between the range of 5 to 15 nodes, the speed improvement diminishes from its peak of 11.41% at 5 nodes to a low of 0.62% at 15 nodes, only to rise again to 1.11% at 20 nodes. Theoretically, as the number of nodes increases, the potential for optimizing the Workstealing target should also grow, resulting in a more pronounced speed improvement. To understand this anomaly, we undertook an analysis:

- An analysis of the Performance-Driven Workstealing Policy revealed minor factors such as increased overhead due to data collection and computation resulting from additional nodes. However, in theory, this overhead should be negligible when compared to the benefits of optimized workstealing targets.
- A deeper investigation into the application's implementation and specific test data suggested the primary issue likely stemmed from performance anomalies referenced in 2.2.1. The Maxclique program based on the Depth-Bounded skeleton exhibited a higher probability of super-linear deceleration compared to other applications. To validate this hypothesis, beyond analyzing the YewPar code, we also studied the runtime fluctuations of the Maxclique program based on the Depth-Bounded skeleton. Using

the data from a 10-node test as an example, the fluctuations are depicted in 4.2. When

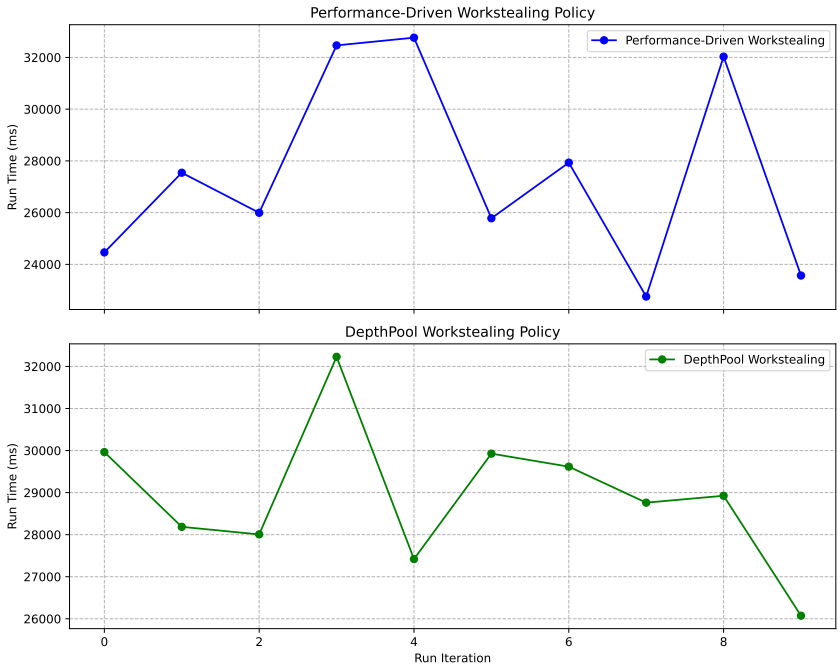


Figure 4.2: Runtime fluctuations for maxclique-16 application based on two different policies using 10 localities.

evaluating the runtime variability of the Maxclique-16 application, we calculated the percentage difference between the maximum and minimum values using the formula:

$$\text{Variability Percentage} = \left(\frac{\text{Max Value} - \text{Min Value}}{\text{Min Value}} \right) \times 100\% \tag{4.2}$$

The Maxclique-16 application based on the Performance-Driven Workstealing Policy had a maximum variability percentage of 43.98%. In contrast, the application based on the DepthPool Workstealing Policy had a maximum variability percentage of 23.61%. It's clear that the Maxclique program's runtime based on the Depth-Bounded skeleton already exhibited significant variability, but the variability of the Maxclique program based on the Performance-Driven Workstealing Policy was nearly double. Considering the content of performance anomalies in 2.2.1, it's evident that the Maxclique program might disrupt the search heuristic more with the addition of computational resources, leading to an increase in extraneous tasks.

Although the frequent changes in the Workstealing target might result in the generation of extra tasks, it's worth noting in 4.1 that between 15 and 20 nodes, there's a relative speed increase. This can likely be attributed to the benefits of optimized Workstealing targets progressively outweighing the detriments caused by the additional tasks.

4.3.2 Isolated Performance Evaluation of NS-hivert

For the NS-hivert program based on the Budget skeleton, the results after averaging the runtimes from ten iterations for each test set are depicted in 4.3. The evaluation encompasses both runtime assessment and the speed improvement evaluation comparing the Performance-Driven Workstealing Policy to the original DepthPool Workstealing Policy.

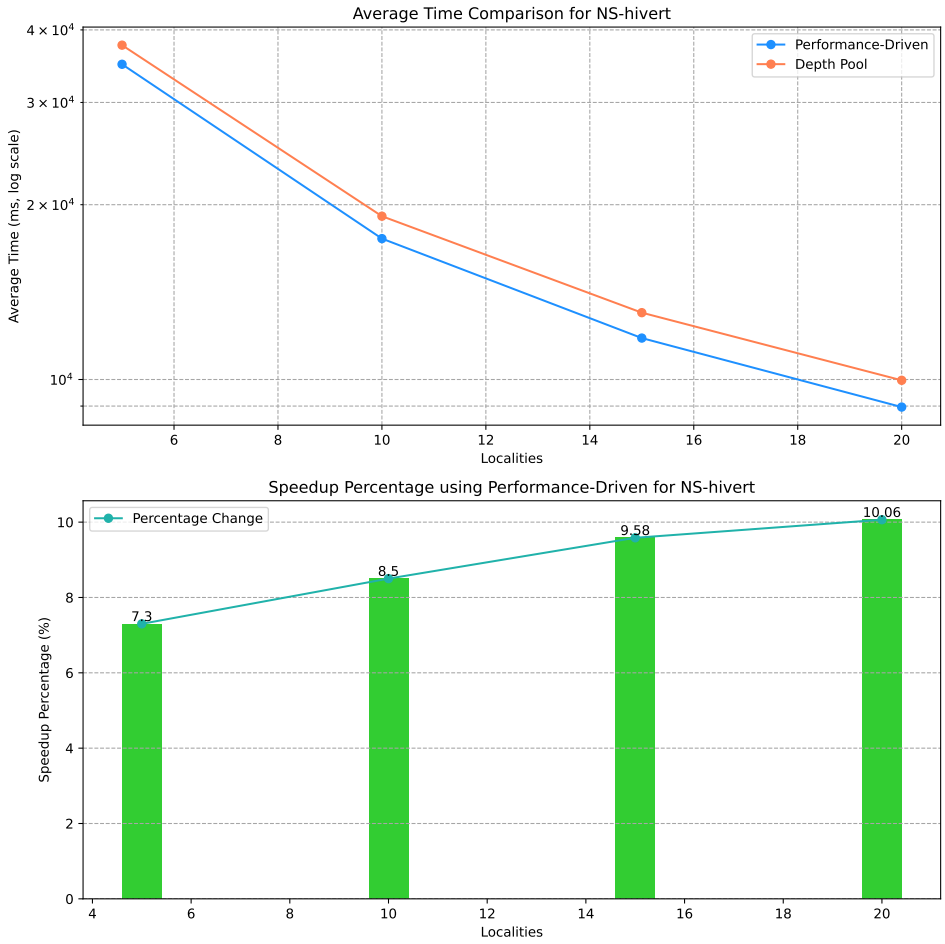


Figure 4.3: isolated time comparison for NS-hivert

From the graph, it's evident that when run in isolation, the NS-hivert program based on the Performance-Driven Workstealing Policy exhibits superior performance across different node counts, completing all tasks in a shorter duration. The speed improvement escalates with the increase in node count. At 20 nodes, there's an average speed increment of 10.06%. Considering this enhancement is solely achieved by adjusting the Workstealing Policy without any modifications to other parts of YewPar and despite the adverse impacts of performance anomalies referenced in 2.2.1, this result is commendable.

Contrasting with the Maxclique program, the speed improvements attributed to the Performance-Driven Workstealing Policy are more pronounced here as node counts increase. This is primarily because the NS-hivert program experiences fewer adverse effects from performance anomalies as discussed in 2.2.1. We also analyzed the runtime variability of the NS-hivert program, as shown in 4.4.

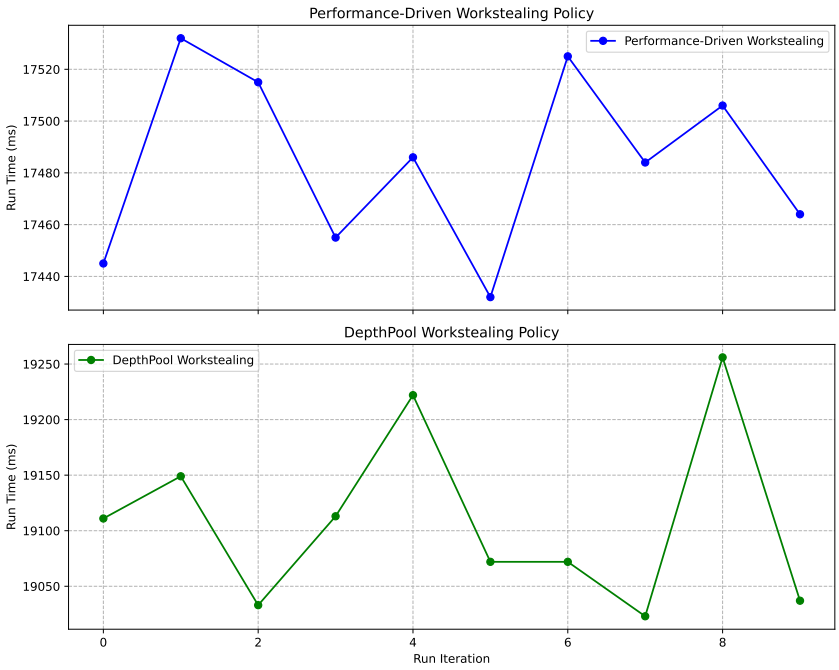


Figure 4.4: Runtime fluctuations for NS-hivert application based on two different policies using 10 localities.

Utilizing 4.2 to calculate the maximum variability percentage, for the NS-hivert application based on the Performance-Driven Workstealing Policy, the maximum variability percentage was 0.57%. For the application based on the DepthPool Workstealing Policy, it was 1.22%. Relative to the runtime fluctuations of the Maxclique program, these variations are minimal, indicating that the NS-hivert program likely has a lower probability of experiencing super-linear deceleration when task execution orders are altered. The primary cause for the speed improvement is the benefits derived from optimizing workstealing targets.

4.4 Simultaneous Performance Evaluation under Resource Contention

This section primarily assesses the performance difference between the YewPar based on the new Performance-Driven Workstealing Policy and the original version under conditions where computational resources on various nodes are imbalanced and contended. Given that real deployment environments can be complex, with different nodes having varying computational resource occupancy, comparing the performance of the Performance-Driven Workstealing Policy under resource contention can provide a more comprehensive evaluation of its capabilities.

To simulate computational resource occupancy across different nodes, we employed a straightforward method by concurrently running YewPar programs based on different Policies. Allowing both programs to execute simultaneously induces significant resource contention since the number of threads set for each program on every node matches the number of cores available on that node. Given that the original DepthPool Workstealing Policy adopts a more random workstealing strategy, it can also effectively simulate resource contention under load imbalances.

However, this approach introduces a challenge. Once one of the programs completes its ex-

ecution, the remaining program will monopolize all computational resources. Consequently, the performance improvements observed for the Performance-Driven Workstealing Policy in the subsequent evaluation might be underestimated relative to real-world scenarios. Nevertheless, these evaluation results can still shed light on the performance disparities between the two Policies under conditions of imbalanced and contended computational resources on different nodes.

4.4.1 Simultaneous Performance Evaluation of Maxclique

For the Maxclique program based on the Depth-Bounded skeleton, the results of the runtime tests under simultaneous execution conditions, averaged over ten runs, are illustrated in 4.5. The evaluation is again bifurcated into two aspects: the runtime assessment and the comparison of the speed improvement between the Performance-Driven Workstealing Policy and the original DepthPool Workstealing Policy.

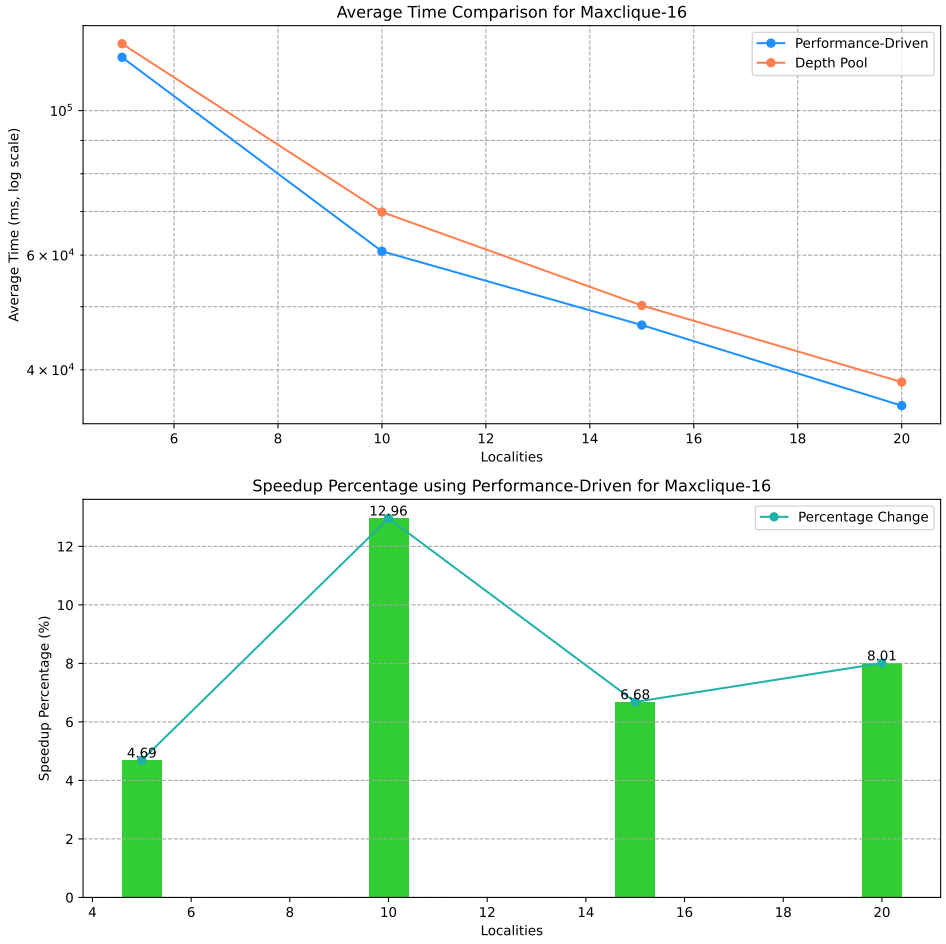


Figure 4.5: simultaneous time comparison for Maxclique-16

It is evident that while Maxclique does exhibit significant super-linear slowdowns, under conditions of imbalanced node loads, the runtime advantages of the Maxclique application based on the Performance-Driven Workstealing Policy become more pronounced. For instance, with a node count of 20, the speed improvement ratio escalates from 1.11% during isolated execution to 8.01% during simultaneous execution, an almost eight-fold increase. This enhancement is presumably because, under imbalanced loads, the benefits from optimizing workstealing targets significantly outweigh the costs introduced by additional tasks.

Furthermore, the graph indicates that, as the node count transitions from 10 to 15, the detrimental effects of additional tasks somewhat mitigate the speed improvements. However, as the number of nodes continues to increase, the advantages of optimizing workstealing targets gradually become more manifest.

4.4.2 Simultaneous Performance Evaluation of NS-hivert

For the NS-hivert program based on the Budget skeleton, the average results over ten runs for the runtime tests under simultaneous execution conditions are presented in 4.6. The evaluation encompasses both the runtime assessment and the comparison of speed improvement between the Performance-Driven Workstealing Policy and the original DepthPool Workstealing Policy.

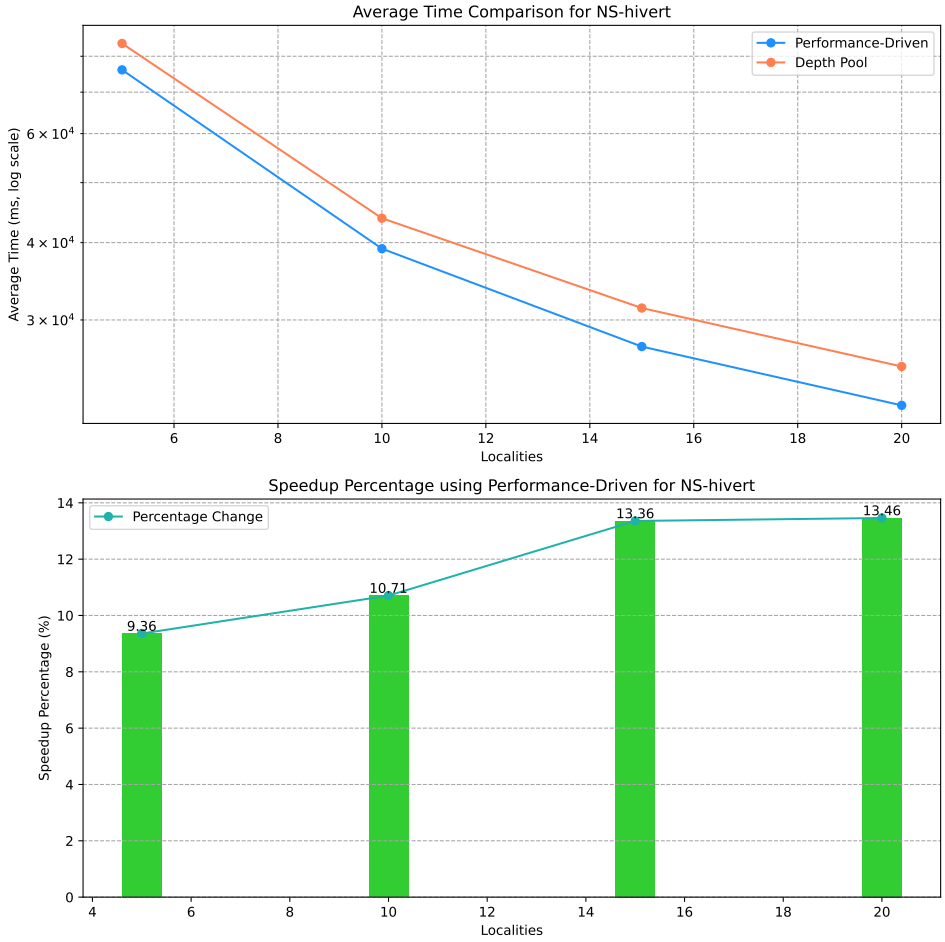


Figure 4.6: simultaneous time comparison for NS-hivert

The graph reveals that within the NS-hivert program based on the Budget skeleton, the performance enhancements from the Performance-Driven Workstealing Policy are considerably pronounced. Starting from a 9.36% improvement with 5 nodes, it steadily rises to 13.46% with 20 nodes. Given the limited optimization space that the Workstealing Policy has within the entire YewPar framework and the NS-hivert program, achieving a 13.46% performance boost is exceptionally commendable.

The modest increment observed in the graph from 15 to 20 nodes suggests that as the number of nodes increases, the percentage of performance improvement brought by Workstealing

is nearing its potential limit. Most of the program's runtime is likely consumed by task processing and other overheads associated with the YewPar framework.

4.5 Evaluation of Additional Performance Cost of Performance-driven Workstealing Policy Framework

To investigate whether the lightweight design of the Performance-Driven Workstealing Policy incurs a significant additional performance overhead, we compared the runtime of the NS-hivert application under different policies in a scenario where only one node with one worker (utilizing a single thread) was used. We opted for the NS-hivert application due to its minimal runtime fluctuations, which minimizes interference. To expedite the runtime for ease of measurement, we adjusted its parameter g to $g = 37$ (genus: Depth in the tree to count until).

The test results are displayed in 4.7.

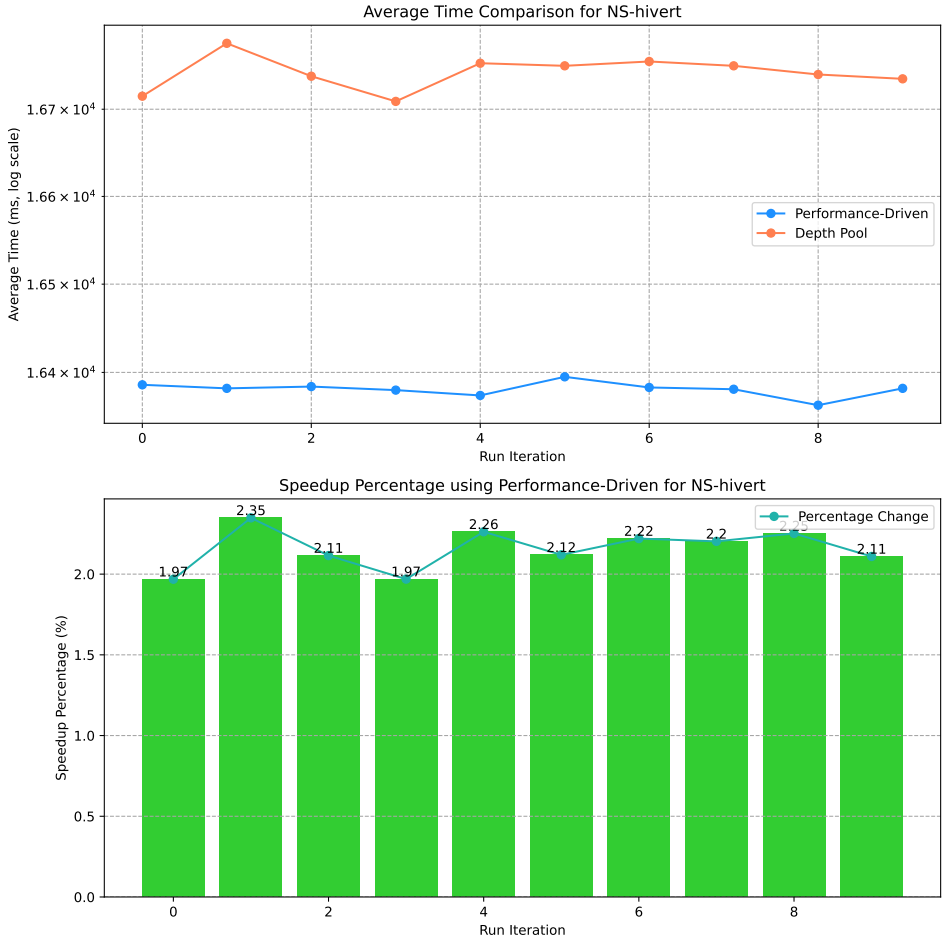


Figure 4.7: time comparison for NS-hivert using one locality with one worker

From the results, it is evident that due to the lightweight design of the Performance-Driven Workstealing Policy, which incorporates mechanisms like lock-free operations to reduce additional computational overheads, even when using a single worker (where no task stealing occurs, implying no optimization of stealing targets) and in scenarios where the Performance-Driven Workstealing Policy dynamically refreshes stealing targets in the background, it still

achieves around a 2% speed improvement over the original YewPar's Workstealing Policy. This indicates the design's tremendous success in achieving its lightweight objective, effectively reducing the performance losses introduced by the task-stealing framework.

Chapter 5: Future Work & Conclusion

5.1 Future Work

5.1.1 Automatic Adjustment of Parameters

In the previous 3 section, some fixed parameters in the algorithm, such as the proportion of historical data in the exponential smoothing algorithm, were determined based on test results to be optimal values. However, in more complex real-world scenarios, these parameters might have better alternatives and present more room for optimization. Hence, one could consider designing an algorithm that automatically adjusts these parameters to ensure that all parameters within the Performance-Driven Workstealing Policy are always in an optimal state.

5.1.2 Optimization for Performance Anomalies in Parallel Combined Searches

From our previous evaluations, it was observed that under the Maxclique program, the Performance-Driven Workstealing Policy frequently disrupted the heuristic search methods by frequently changing stealing targets, jumbling the task execution sequence, leading to the generation of more additional tasks for context speculation, and subsequently slowing down the overall program execution speed. It might be beneficial to design an algorithm that takes into account the preservation of task execution order when calculating stealing targets. This might reduce the generation of extra tasks and further boost the program's execution speed in multi-node, multi-threaded scenarios.

5.1.3 Substitute All YewPar Workstealing Policies

Currently, the Performance-Driven Workstealing Policy has replaced policies in YewPar such as the DepthPool Policy, thereby enhancing the performance of some of YewPar's skeletons. However, due to time constraints, we have not yet attempted to replace policies like the Priority Ordered Policy. As a result, not all skeletons in YewPar have benefited from the Performance-Driven Workstealing Policy. This presents an avenue for future exploration and work.

5.2 Conclusion

Our objective was to enhance the performance of YewPar by refining its Workstealing strategy. To this end, we devised and implemented a Performance-Driven Workstealing Policy, which optimizes the stealing targets during Workstealing based on performance data from each node. This involved the collection, processing, and transmission of performance data, as well as calculations and caching for the optimal stealing targets, complemented by a refreshing mechanism that offers two distinct methods. Through optimization tailored for multi-threading and inter-node communication, we achieved improvements in program execution speed at a minimal cost, while maintaining excellent compatibility.

As per our evaluation results, the Performance-Driven Workstealing Policy framework is lightweight, incurring minimal overheads. Its optimization in the realm of Workstealing is remarkably evident. In scenarios free of computational resource occupation, the overall

speed improvement of the program can surpass 8%. Moreover, in situations where node resources are unevenly utilized and there's contention, speed enhancements can exceed 13%. Considering that these improvements were achieved merely by optimizing YewPar's Workstealing Policy, and given that most of the program's runtime is spent processing tasks and generating new ones, it's evident that adjusting the stealing targets based on the performance data of each node is a highly effective approach. Therefore, the Performance-Driven Workstealing Policy stands out as an exceptional Workstealing strategy and can seamlessly replace YewPar's original Workstealing Policy.

Appendix A: Appendix

A.1 Project description and Source code

The source code and other descriptions for this project can be found at :

`https://github.com/shadowxiehao/YewPar_with_PerformancePolicy`.

You can find specific instructions on the project such as installation instructions in the Readme file.

A.2 Evaluation Data

The following data is the commands for evaluation and the 10 results(running time, Displayed by arrays, unit: ms) of running corresponding commands. The commands should vary according to the actual environment.

```
============================================================

Isolated evaluation:


Maxclique:
==========

mpiexec.openmpi -n 20 --host gpgnode-01,gpgnode-02,gpgnode
    -03,gpgnode-04,gpgnode-05,gpgnode-06,gpgnode-07,gpgnode
    -08,gpgnode-09,gpgnode-10,gpgnode-11,gpgnode-12,gpgnode
    -13,gpgnode-14,gpgnode-15,gpgnode-16,gpgnode-17,gpgnode
    -18,gpgnode-19,gpgnode-20 /cluster/gpg/hao/YewPar/build/
    install/bin/maxclique-16 -f /cluster/gpg/
    maxclique_instaces/brock800_2.clq --skeleton depthbounded
    -d 2 --hpx:threads 16

[14829,12791,16863,17266,16410,13920,14489,17646,21343,12549]

mpiexec.openmpi -n 20 --host gpgnode-01,gpgnode-02,gpgnode
    -03,gpgnode-04,gpgnode-05,gpgnode-06,gpgnode-07,gpgnode
    -08,gpgnode-09,gpgnode-10,gpgnode-11,gpgnode-12,gpgnode
    -13,gpgnode-14,gpgnode-15,gpgnode-16,gpgnode-17,gpgnode
    -18,gpgnode-19,gpgnode-20 /cluster/gpg/YewPar/build/
    install/bin/maxclique-16 -f /cluster/gpg/
    maxclique_instaces/brock800_2.clq --skeleton depthbounded
    -d 2 --hpx:threads 16

[16893,15822,17997,15739,16006,13570,17827,17189,16246,12595]
```

```
---

mpiexec.openmpi -n 15 --host gpgnode-01,gpgnode-02,gpgnode
    -03,gpgnode-04,gpgnode-05,gpgnode-06,gpgnode-07,gpgnode
    -08,gpgnode-09,gpgnode-10,gpgnode-11,gpgnode-12,gpgnode
    -13,gpgnode-14,gpgnode-15 /cluster/gpg/hao/YewPar/build/
    install/bin/maxclique-16 -f /cluster/gpg/
    maxclique_instaces/brock800_2.clq --skeleton depthbounded
    -d 2 --hpx:threads 16

[21533,19751,20677,25336,21294,25132,18389,17850,19703,16009]

mpiexec.openmpi -n 15 --host gpgnode-01,gpgnode-02,gpgnode
    -03,gpgnode-04,gpgnode-05,gpgnode-06,gpgnode-07,gpgnode
    -08,gpgnode-09,gpgnode-10,gpgnode-11,gpgnode-12,gpgnode
    -13,gpgnode-14,gpgnode-15 /cluster/gpg/YewPar/build/
    install/bin/maxclique-16 -f /cluster/gpg/
    maxclique_instaces/brock800_2.clq --skeleton depthbounded
    -d 2 --hpx:threads 16

[21678,22525,19146,18976,20766,22809,19951,20347,21012,19747]

---

---

mpiexec.openmpi -n 10 --host gpgnode-01,gpgnode-02,gpgnode
    -03,gpgnode-04,gpgnode-05,gpgnode-06,gpgnode-07,gpgnode
    -08,gpgnode-09,gpgnode-10 /cluster/gpg/hao/YewPar/build/
    install/bin/maxclique-16 -f /cluster/gpg/
    maxclique_instaces/brock800_2.clq --skeleton depthbounded
    -d 2 --hpx:threads 16

[24463,27543,25995,32465,32764,25782,27933,22756,32031,23569]

mpiexec.openmpi -n 10 --host gpgnode-01,gpgnode-02,gpgnode
    -03,gpgnode-04,gpgnode-05,gpgnode-06,gpgnode-07,gpgnode
    -08,gpgnode-09,gpgnode-10 /cluster/gpg/YewPar/build/
    install/bin/maxclique-16 -f /cluster/gpg/
    maxclique_instaces/brock800_2.clq --skeleton depthbounded
    -d 2 --hpx:threads 16

[29963,28186,28006,32229,27420,29927,29617,28761,28925,26073]

---

---

mpiexec.openmpi -n 5 --host gpgnode-01,gpgnode-02,gpgnode-03,
    gpgnode-04,gpgnode-05 /cluster/gpg/hao/YewPar/build/
    install/bin/maxclique-16 -f /cluster/gpg/
```

```
    maxclique_instaces/brock800_2.clq --skeleton depthbounded
    -d 2 --hpx:threads 16

[51414,64284,58569,66468,65849,55616,62883,64843,63617,60311]

mpiexec.openmpi -n 5 --host gpgnode-01,gpgnode-02,gpgnode-03,
    gpgnode-04,gpgnode-05 /cluster/gpg/YewPar/build/install/
    bin/maxclique-16 -f /cluster/gpg/maxclique_instaces/
    brock800_2.clq --skeleton depthbounded -d 2 --hpx:threads
    16

[79945,50156,51396,53098,99086,99490,75471,51503,64685,68091]

---

==========

NS-hivert:

==========

---

mpiexec.openmpi -n 20 --host gpgnode-01,gpgnode-02,gpgnode
    -03,gpgnode-04,gpgnode-05,gpgnode-06,gpgnode-07,gpgnode
    -08,gpgnode-09,gpgnode-10,gpgnode-11,gpgnode-12,gpgnode
    -13,gpgnode-14,gpgnode-15,gpgnode-16,gpgnode-17,gpgnode
    -18,gpgnode-19,gpgnode-20 /cluster/gpg/hao/YewPar/build/
    install/bin/NS-hivert --skeleton budget -b 1000000 -g 47
    --hpx:threads 16

[8942,9064,8978,8932,8904,8932,8966,9013,8956,8962]

mpiexec.openmpi -n 20 --host gpgnode-01,gpgnode-02,gpgnode
    -03,gpgnode-04,gpgnode-05,gpgnode-06,gpgnode-07,gpgnode
    -08,gpgnode-09,gpgnode-10,gpgnode-11,gpgnode-12,gpgnode
    -13,gpgnode-14,gpgnode-15,gpgnode-16,gpgnode-17,gpgnode
    -18,gpgnode-19,gpgnode-20 /cluster/gpg/YewPar/build/
    install/bin/NS-hivert --skeleton budget -b 1000000 -g 47
    --hpx:threads 16

[9882,9948,9954,9978,9945,9989,9871,10042,10016,10056]

---

mpiexec.openmpi -n 15 --host gpgnode-01,gpgnode-02,gpgnode
    -03,gpgnode-04,gpgnode-05,gpgnode-06,gpgnode-07,gpgnode
    -08,gpgnode-09,gpgnode-10,gpgnode-11,gpgnode-12,gpgnode
    -13,gpgnode-14,gpgnode-15 /cluster/gpg/hao/YewPar/build/
    install/bin/NS-hivert --skeleton budget -b 1000000 -g 47
    --hpx:threads 16
```

```
[11710,11826,11760,11772,11750,11760,11975,11757,11801,11746]

mpiexec.openmpi -n 15 --host gpgnode-01,gpgnode-02,gpgnode
    -03,gpgnode-04,gpgnode-05,gpgnode-06,gpgnode-07,gpgnode
    -08,gpgnode-09,gpgnode-10,gpgnode-11,gpgnode-12,gpgnode
    -13,gpgnode-14,gpgnode-15 /cluster/gpg/YewPar/build/
    install/bin/NS-hivert --skeleton budget -b 1000000 -g 47
    --hpx:threads 16

[13005,13007,13054,13031,13051,13061,13039,12982,13024,13096]

---

mpiexec.openmpi -n 10 --host gpgnode-01,gpgnode-02,gpgnode
    -03,gpgnode-04,gpgnode-05,gpgnode-06,gpgnode-07,gpgnode
    -08,gpgnode-09,gpgnode-10 /cluster/gpg/hao/YewPar/build/
    install/bin/NS-hivert --skeleton budget -b 1000000 -g 47
    --hpx:threads 16

[17445,17532,17515,17455,17486,17432,17525,17484,17506,17464]

mpiexec.openmpi -n 10 --host gpgnode-01,gpgnode-02,gpgnode
    -03,gpgnode-04,gpgnode-05,gpgnode-06,gpgnode-07,gpgnode
    -08,gpgnode-09,gpgnode-10 /cluster/gpg/YewPar/build/
    install/bin/NS-hivert --skeleton budget -b 1000000 -g 47
    --hpx:threads 16

[19111,19149,19033,19113,19222,19072,19072,19023,19256,19037]

---

mpiexec.openmpi -n 5 --host gpgnode-01,gpgnode-02,gpgnode-03,
    gpgnode-04,gpgnode-05 /cluster/gpg/hao/YewPar/build/
    install/bin/NS-hivert --skeleton budget -b 1000000 -g 47
    --hpx:threads 16

[34922,34849,34898,34913,34891,34926,34960,34891,34919,34859]

mpiexec.openmpi -n 5 --host gpgnode-01,gpgnode-02,gpgnode-03,
    gpgnode-04,gpgnode-05 /cluster/gpg/YewPar/build/install/
    bin/NS-hivert --skeleton budget -b 1000000 -g 47 --hpx:
    threads 16

[37633,37703,37710,37650,37613,37668,37720,37571,37682,37563]

---

==========

============================================================
```

```
Simultaneous evaluation:

Maxclique:
==========

mpiexec.openmpi -n 20 --host gpgnode-01,gpgnode-02,gpgnode
    -03,gpgnode-04,gpgnode-05,gpgnode-06,gpgnode-07,gpgnode
    -08,gpgnode-09,gpgnode-10,gpgnode-11,gpgnode-12,gpgnode
    -13,gpgnode-14,gpgnode-15,gpgnode-16,gpgnode-17,gpgnode
    -18,gpgnode-19,gpgnode-20 /cluster/gpg/hao/YewPar/build/
    install/bin/maxclique-16 -f /cluster/gpg/
    maxclique_instaces/brock800_2.clq --skeleton depthbounded
    -d 2 --hpx:threads 16

[29660,40314,37144,33314,37146,35187,39280,38449,30695,31253]

mpiexec.openmpi -n 20 --host gpgnode-01,gpgnode-02,gpgnode
    -03,gpgnode-04,gpgnode-05,gpgnode-06,gpgnode-07,gpgnode
    -08,gpgnode-09,gpgnode-10,gpgnode-11,gpgnode-12,gpgnode
    -13,gpgnode-14,gpgnode-15,gpgnode-16,gpgnode-17,gpgnode
    -18,gpgnode-19,gpgnode-20 /cluster/gpg/YewPar/build/
    install/bin/maxclique-16 -f /cluster/gpg/
    maxclique_instaces/brock800_2.clq --skeleton depthbounded
    -d 2 --hpx:threads 16

[34990,37954,42202,35970,39627,37308,47165,34588,36364,36959]

---

mpiexec.openmpi -n 15 --host gpgnode-01,gpgnode-02,gpgnode
    -03,gpgnode-04,gpgnode-05,gpgnode-06,gpgnode-07,gpgnode
    -08,gpgnode-09,gpgnode-10,gpgnode-11,gpgnode-12,gpgnode
    -13,gpgnode-14,gpgnode-15 /cluster/gpg/hao/YewPar/build/
    install/bin/maxclique-16 -f /cluster/gpg/
    maxclique_instaces/brock800_2.clq --skeleton depthbounded
    -d 2 --hpx:threads 16

[49062,45182,50122,40388,38596,53504,61126,34767,52585,43253]

mpiexec.openmpi -n 15 --host gpgnode-01,gpgnode-02,gpgnode
    -03,gpgnode-04,gpgnode-05,gpgnode-06,gpgnode-07,gpgnode
    -08,gpgnode-09,gpgnode-10,gpgnode-11,gpgnode-12,gpgnode
    -13,gpgnode-14,gpgnode-15 /cluster/gpg/YewPar/build/
    install/bin/maxclique-16 -f /cluster/gpg/
    maxclique_instaces/brock800_2.clq --skeleton depthbounded
    -d 2 --hpx:threads 16

[54296,57342,45869,52949,50054,51638,67120,37218,42635,43020]

---
```

```
---

mpiexec.openmpi -n 10 --host gpgnode-01,gpgnode-02,gpgnode
    -03,gpgnode-04,gpgnode-05,gpgnode-06,gpgnode-07,gpgnode
    -08,gpgnode-09,gpgnode-10 /cluster/gpg/hao/YewPar/build/
    install/bin/maxclique-16 -f /cluster/gpg/
    maxclique_instaces/brock800_2.clq --skeleton depthbounded
    -d 2 --hpx:threads 16

[66531,66494,60935,50364,56088,58545,60509,55681,63372,69744]

mpiexec.openmpi -n 10 --host gpgnode-01,gpgnode-02,gpgnode
    -03,gpgnode-04,gpgnode-05,gpgnode-06,gpgnode-07,gpgnode
    -08,gpgnode-09,gpgnode-10 /cluster/gpg/YewPar/build/
    install/bin/maxclique-16 -f /cluster/gpg/
    maxclique_instaces/brock800_2.clq --skeleton depthbounded
    -d 2 --hpx:threads 16

[62531,91967,80865,55159,56090,60476,77641,82621,61687,69789]

---

---

mpiexec.openmpi -n 5 --host gpgnode-01,gpgnode-02,gpgnode-03,
    gpgnode-04,gpgnode-05 /cluster/gpg/hao/YewPar/build/
    install/bin/maxclique-16 -f /cluster/gpg/
    maxclique_instaces/brock800_2.clq --skeleton depthbounded
    -d 2 --hpx:threads 16

[104810,106080,114325,127025,125424,119856,129894,121873,129900,128109]


mpiexec.openmpi -n 5 --host gpgnode-01,gpgnode-02,gpgnode-03,
    gpgnode-04,gpgnode-05 /cluster/gpg/YewPar/build/install/
    bin/maxclique-16 -f /cluster/gpg/maxclique_instaces/
    brock800_2.clq --skeleton depthbounded -d 2 --hpx:threads
    16

[102019,158790,103343,130167,164195,114820,132898,108817,116030,135588]


---

==========

NS-hivert:

==========
```

```
mpiexec.openmpi -n 20 --host gpgnode-01,gpgnode-02,gpgnode
    -03,gpgnode-04,gpgnode-05,gpgnode-06,gpgnode-07,gpgnode
    -08,gpgnode-09,gpgnode-10,gpgnode-11,gpgnode-12,gpgnode
    -13,gpgnode-14,gpgnode-15,gpgnode-16,gpgnode-17,gpgnode
    -18,gpgnode-19,gpgnode-20 /cluster/gpg/hao/YewPar/build/
    install/bin/NS-hivert --skeleton budget -b 1000000 -g 47
    --hpx:threads 16

[22083,22416,22299,20514,20333,22946,23012,21001,20655,23120]

mpiexec.openmpi -n 20 --host gpgnode-01,gpgnode-02,gpgnode
    -03,gpgnode-04,gpgnode-05,gpgnode-06,gpgnode-07,gpgnode
    -08,gpgnode-09,gpgnode-10,gpgnode-11,gpgnode-12,gpgnode
    -13,gpgnode-14,gpgnode-15,gpgnode-16,gpgnode-17,gpgnode
    -18,gpgnode-19,gpgnode-20 /cluster/gpg/YewPar/build/
    install/bin/NS-hivert --skeleton budget -b 1000000 -g 47
    --hpx:threads 16

[25004,25496,25477,25484,24855,25586,25632,24856,24180,25778]

mpiexec.openmpi -n 15 --host gpgnode-01,gpgnode-02,gpgnode
    -03,gpgnode-04,gpgnode-05,gpgnode-06,gpgnode-07,gpgnode
    -08,gpgnode-09,gpgnode-10,gpgnode-11,gpgnode-12,gpgnode
    -13,gpgnode-14,gpgnode-15 /cluster/gpg/hao/YewPar/build/
    install/bin/NS-hivert --skeleton budget -b 1000000 -g 47
    --hpx:threads 16

[26867,27067,28235,27508,25723,25471,28208,28128,26254,28272]

mpiexec.openmpi -n 15 --host gpgnode-01,gpgnode-02,gpgnode
    -03,gpgnode-04,gpgnode-05,gpgnode-06,gpgnode-07,gpgnode
    -08,gpgnode-09,gpgnode-10,gpgnode-11,gpgnode-12,gpgnode
    -13,gpgnode-14,gpgnode-15 /cluster/gpg/YewPar/build/
    install/bin/NS-hivert --skeleton budget -b 1000000 -g 47
    --hpx:threads 16

[32329,31555,31457,30250,30670,30431,31528,31524,31236,32640]

mpiexec.openmpi -n 10 --host gpgnode-01,gpgnode-02,gpgnode
    -03,gpgnode-04,gpgnode-05,gpgnode-06,gpgnode-07,gpgnode
    -08,gpgnode-09,gpgnode-10 /cluster/gpg/hao/YewPar/build/
    install/bin/NS-hivert --skeleton budget -b 1000000 -g 47
    --hpx:threads 16

[39736,37963,38072,40323,38470,39852,37845,40796,38552,39475]

mpiexec.openmpi -n 10 --host gpgnode-01,gpgnode-02,gpgnode
    -03,gpgnode-04,gpgnode-05,gpgnode-06,gpgnode-07,gpgnode
    -08,gpgnode-09,gpgnode-10 /cluster/gpg/YewPar/build/
    install/bin/NS-hivert --skeleton budget -b 1000000 -g 47
    --hpx:threads 16
```

```
[44152,42980,44011,43471,43557,43119,44620,43576,45429,43059]

mpiexec.openmpi -n 5 --host gpgnode-01,gpgnode-02,gpgnode-03,
    gpgnode-04,gpgnode-05 /cluster/gpg/hao/YewPar/build/
    install/bin/NS-hivert --skeleton budget -b 1000000 -g 47
    --hpx:threads 16

[75818,76999,76251,77031,75488,76587,76455,75528,75964,74382]

mpiexec.openmpi -n 5 --host gpgnode-01,gpgnode-02,gpgnode-03,
    gpgnode-04,gpgnode-05 /cluster/gpg/YewPar/build/install/
    bin/NS-hivert --skeleton budget -b 1000000 -g 47 --hpx:
    threads 16

[84444,82186,83962,84170,85396,81858,81957,85656,84645,84748]

==========


============================================================

NS-hivert using only one thread:

/cluster/gpg/hao/YewPar/build/install/bin/NS-hivert --
    skeleton budget -b 1000000 -g 37 --hpx:threads 1

[16386,16382,16384,16380,16374,16395,16383,16381,16363,16382]

/cluster/gpg/YewPar/build/install/bin/NS-hivert --skeleton
    budget -b 1000000 -g 37 --hpx:threads 1

[16715,16776,16738,16709,16753,16750,16755,16750,16740,16735]

============================================================
```

Bibliography

[1] Blair Archibald. *Skeletons for Exact Combinatorial Search at Scale*. PhD thesis, University of Glasgow, 2018.

[2] Blair Archibald, Patrick Maier, Ciaran McCreesh, Robert Stewart, and Phil Trinder. Replicable parallel branch and bound search. *Journal of Parallel and Distributed Computing*, 113:92–114, 2018.

[3] Blair Archibald, Patrick Maier, Robert Stewart, and Phil Trinder. Implementing yewpar: A framework for parallel tree search. In *Euro-Par 2019: Parallel Processing: 25th International Conference on Parallel and Distributed Computing, Göttingen, Germany, August 26–30, 2019, Proceedings*, pages 184–196, Berlin, Heidelberg, 2019. Springer-Verlag.

[4] Blair Archibald, Patrick Maier, Robert Stewart, and Phil Trinder. Yewpar: Skeletons for exact combinatorial search. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '20, pages 292–307, New York, NY, USA, 2020. Association for Computing Machinery.

[5] Blair Archibald, Patrick Maier, Phil Trinder, and Robert Stewart. Yewpar: Skeletons for exact combinatorial search [data collection]. 2019. (2019).

[6] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[7] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, sep 1999.

[8] Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C. Mowry, and Chris Wilkerson. Scheduling threads for constructive cache sharing on cmps. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 105–115, New York, NY, USA, 2007. Association for Computing Machinery.

[9] A. de Bruin, G. A. P. Kindervater, and H. W. J. M. Trienekens. Asynchronous parallel branch and bound and anomalies. In Afonso Ferreira and José Rolim, editors, *Parallel Algorithms for Irregularly Structured Problems*, pages 363–377, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[10] Jean Fromentin and Florent Hivert. Exploring the tree of numerical semigroups. *Math. Comp.*, 85(301):2553–2568, 2016.

[11] Everette S. Gardner. Exponential smoothing: The state of the art—part ii. *International Journal of Forecasting*, 22(4):637–666, 2006.

[12] Fabien Gaud, Sylvain Genevès, Renaud Lachaize, Baptiste Lepers, Fabien Mottet, Gilles Muller, and Vivien Quéma. Efficient workstealing for multicore event-driven systems. In *2010 IEEE 30th International Conference on Distributed Computing Systems*, pages 516–525, 2010.

[13] Bernard Gendron and Teodor Gabriel Crainic. Parallel branch-and-branch algorithms: Survey and synthesis. *Operations Research*, 42(6):1042–1066, 1994.

[14] Jan Gmys, Rudi Leroy, Mohand Mezmaz, Nouredine Melab, and Daniel Tuyttens. Work stealing with private integer-vector-matrix data structure for multi-core branch-and-bound algorithms. *Concurrency and Computation: Practice and Experience*, 28(18):4463–4484, 2016.

[15] Paul Krill. Tokio rust runtime reaches 1.0 status. *InfoWorld*, Jan 2021. Retrieved 2021-12-26.

[16] Doug Lea. A java fork/join framework. In *ACM Conference on Java*, 2000.

[17] E.A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[18] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. *ACM SIGPLAN Notices*, 44(10):227, 2009. CiteSeerX 10.1.1.146.4197.

[19] Ciaran McCreesh and Patrick Prosser. Multi-threading a state-of-the-art maximum clique algorithm. *Algorithms*, 6(4):618–635, 2013.

[20] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, page 73–82, New York, NY, USA, 2002. Association for Computing Machinery.

[21] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, page 267–275, New York, NY, USA, 1996. Association for Computing Machinery.

[22] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P Sadayappan, and Chau-Wen Tseng. Uts: An unbalanced tree search benchmark. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 235–250. Springer, 2006.

[23] Hrushit Parikh, Vinit Deodhar, Ada Gavrilovska, and Santosh Pande. Efficient distributed workstealing via matchmaking. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, New York, NY, USA, 2016. Association for Computing Machinery.

[24] tokio.rs contributors. What is tokio? `https://tokio.rs/tokio/tutorial`. Retrieved 2020-05-27.