In the field of computer science, multidimensional optimization problems are extremely important. It focuses on the difficulty of quickly determining the best or near-best values for a varying objective function based on several different variables. One of the most difficult problems in computer science, this problem is inherently demanding and is classified as NP-hard. Many heuristic methods have been developed to tackle this complex problem, seeking to estimate excellent answers even if they do not always yield optimal results. The algorithm called Fish School Behavior (FSB) is one such heuristic method.

Since most heuristic algorithms allow parallel execution, the main goal of this simulation is to parallelize a condensed version of the FSB method. Although our priority is parallelization, we do not want to tackle the optimization problem directly. Instead, we try to understand how these optimization problems are parallelized. Other optimization problems, such as ant colony and particle swarm optimization, can be solved using a similar parallelization strategy.

1.1 The Concept

The lake represents the metaphorical solution space of an optimization problem and is inhabited by a school of fish. In this lake, food is distributed differently; some areas have a lot, others don't. The main goal is to direct most fish to areas with larger amounts of food. Every fish can swim in the direction corresponding to what it eats. The perception of abundant food sources in an area causes fish to swim randomly in that direction. In addition, they benefit from collective support based on the school's experience. To simulate the behavior of a school of fish, other fish will head in that direction if many of them discover food there.

In this hypothetical simulation, the lake is represented as a square of area 200,200 with center (0, 0). The x and y coordinates extend up and down and are positive and negative to the right and left of the center, respectively. Using randomly generated coordinates, fish are initially randomly placed in this area. Each fish has x,y coordinates and the Euclidean distance formula is used to determine the distance from each fish's origin point. Each fish in the school participates in simulation rounds, where the simulation progresses.

1.2 The Fish's Actions

Eating:

Each fish has a predetermined initial weight (w) and a weight limit of 2w. At this point, the objective function – a simple sum of squares using the coordinates of each fish – is presented. Based on the change in the objective function caused by the fish's movement, the weight of the fish in the simulation step (t + 1) is updated.

Swimming:

If a fish finds food in the current round, it will swim in a random direction. By generating random numbers to change the fish's x and y coordinates, swimming behavior is reproduced.

1.3 Collective Action

The group action consists in turning all the fish in the school towards the center of the bary. This Barycenter, which simulates a simplified version of the behavior of a real school of fish, is calculated using the weighted coordinates of the fish.

The goal of this simulation is to study how to parallelize the FSB technique for a condensed optimization problem. Understanding the parallelization process, which can be applied to different optimization techniques, is the main goal rather than solving the optimization problem. This will increase the efficiency and speed of optimization in multidimensional space by distributing the computational load across multiple threads.

2 Description of codes

The code provides a thorough simulation of a fish school's behavior and boosts efficiency by using parallel computing methods like MPI and OpenMP. A predetermined number of fish are put into the simulation, each of which is identified by its respective locations, starting weight, and behavioral behaviors. The simulation accurately depicts the fundamental elements of fish behavior, such as their random movement, eating habits, and weight changes depending on predetermined probabilities. These key elements have a considerable impact on the fish in the simulation's general behavior and attributes.

The smart division of this code's functions into different sections encourages modularity and maintainability. Each function has a distinct function that adds to the code's readability and reusability. To calculate the goal function, update fish weights, simulate fish behavior, locate the center of gravity of the school, and categorize fish based on their coordinates, functions are used. The debugging and editing procedures are made easier and the readability of the code is improved by the usage of functions.

The code's effective use of parallelism to speed up the simulation is one of its significant advantages. While OpenMP is utilized for multi-threaded execution within each MPI process, MPI is used to distribute the fish population throughout many processes. To enhance the performance of the simulation, this hybrid parallel technique makes use of both distributed and shared memory parallelism.

In an OpenMP parallel area, where each thread oversees overseeing a different subset of fish, the simulation stages are coordinated. By using several scheduling strategies, such as static, dynamic, and guided scheduling to effectively spread the computational effort among threads, the code illustrates the adaptability and versatility of OpenMP. These scheduling strategies maximize computing efficacy while handling several jobs at once, demonstrating the potency of OpenMP for handling concurrent workloads.

The "qsort" function is used to make the code's fish classification scheme based on fishes' coordinates more convenient. This capability improves the code's capacity to be flexible and useful in a variety of applications because sorting is a key activity in data processing and analytics.

This code serves as an example of how parallelism may be strategically used to improve simulation effectiveness and computing speed. It emphasizes how important well-structured code and algorithmic techniques are for speedy data processing and analysis. This code clearly demonstrates the advantages of parallel processing, including faster execution times and the capacity to successfully handle several tasks at once.

2.1 Heap-Based Implementation using MPI:

A parallel C program that generates fish data, distributes it among several processes, and writes the data to files using the Message Passing Interface library. The main objective of the program is to illustrate a straightforward parallel computing and data dissemination use case.

Code Overview:

1. Header Files: The code begins by including necessary header files like `<stdio.h>`, `<stdlib.h>`, `<math.h>`, `<time.h>`, and `<mpi.h>`. The `<mpi.h>` header is specifically for using MPI functions.

2. Constant Definitions: The code defines a constant `NUM_FISH` as 100. This represents the number of fish data points that will be generated.

3. Struct Definition: The `Fish` struct is defined to represent a fish's properties, including its x and y coordinates and weight.

4. `generateFishData` Function:

- This function generates random fish data.

- It uses the `srand` function to seed the random number generator with the current time.

- Then, it generates `numFish` (100) fish data points, with random x and y coordinates within a certain range and sets a default weight of 1.0.

5. `writeFishDataToFile` Function:

- This function writes fish data to a file.

- It takes a filename, an array of fish data, and the number of fish data points as input.

- It opens the specified file for writing and checks for errors.

- It then iterates through the fish data, formatting and writing it to the file.

6. `main` Function:

- The program's entry point, it initializes MPI, retrieves the rank and size of the MPI communicator, and calculates how many fish each process will handle.

- Dynamic memory is allocated for the `fish` array and the `recvFish` array. Each process will generate and work with a portion of the fish data.

- The `MPI_Scatter` function is used to distribute the fish data from the master process (rank 0) to all processes.

- The `MPI_Gather` function collects the fish data back to the master process after all processes have processed their share.

- The program writes fish data to a file immediately after the master process generates it, and it writes another file after collecting all the data.

- The `free` function is used to release the allocated memory, and MPI is finalized before the program terminates.

Parallel Processing:

- The code utilizes MPI for parallel processing. The `MPI_Init`, `MPI_Comm_rank`, and `MPI_Comm_size` functions are used to initialize MPI, get the rank of each process, and find the total number of processes, respectively.

- Data is distributed among processes using `MPI_Scatter` and collected using `MPI_Gather`.

This program shows a straightforward MPI parallel computing example. Fish data is generated, distributed across several systems, and written to files. The parallelization in this code lays the groundwork for more sophisticated parallel computing applications, even if it may not yield noticeable performance advantages for this activity.

2.2 Heap -Based Implementation with MPI and OpenMP:

A C program that uses OpenMP and the Message Passing Interface to distribute and generate fish data across other processes in parallel. Additionally, the program logs and displays the overall execution time. This code's main objective is to show how to use OpenMP for intra-process parallelization and MPI for inter-process communication.

Code Overview:

1. Header Files: The code includes standard C library headers (`<stdio.h>`, `<stdlib.h>`, `<math.h>`, `<time.h>`) and headers for MPI and OpenMP (`<mpi.h>`, `<omp.h>`).

2. Constant Definitions: The code defines the number of fish (`NUM_FISH`) and the number of OpenMP threads (`NUM_THREADS`) to use for parallel processing.

3. Struct Definition: A `Fish` struct is defined to represent a fish's properties, including its x and y coordinates and weight.

4. `generateFishData` Function:

- This function generates random fish data, like the previous code.

- It initializes the random number generator with the current time.

- Generates `numFish` fish data points, with random x and y coordinates and a default weight of 1.0.

5. `writeFishDataToFile` Function:

- This function writes fish data to a file, as in the previous code.

- It takes a filename, an array of fish data, and the number of fish data points as input.

- It opens the specified file for writing and checks for errors.

- It iterates through the fish data, formatting and writing it to the file.

6. `main` Function:

- The program's entry point, it initializes MPI, retrieves the rank and size of the MPI communicator, and calculates how many fish each process will handle.

- Dynamic memory is allocated for the `fish` array and the `recvFish` array, with each process generating and working with a portion of the fish data.

- The program records the start time using `MPI_Wtime`.

- An OpenMP parallel region is defined using `#pragma omp parallel`, and the number of threads to be used is set to `NUM_THREADS`.

- Within the parallel region, each thread gets its thread ID and the total number of threads, which is used to divide the work into chunks.

- The `generateFishData` function is called by each thread, with each thread generating a portion of the fish data.

- After generating fish data, the program continues with MPI operations to distribute and gather the data.

- The program records the end time using `MPI_Wtime`.

- Fish data is written to a file immediately after the master process generates it and to another file after collecting all the data, like the previous code.

- The code calculates and prints the total execution time for the entire program, but only the master process (rank 0) does this.

- Dynamic memory is released, and MPI is finalized before the program terminates.

Parallel Processing:

- This code combines MPI for parallelism across multiple processes and OpenMP for parallelism within each process (thread-level parallelism).

- The number of threads used by OpenMP is specified using the `NUM_THREADS` constant, allowing multiple threads to generate fish data concurrently.

The parallel processing capabilities of MPI and OpenMP are shown in this code. It creates and distributes fish data effectively among several processes, monitors and reports the overall execution time. By utilizing both inter-process and intra-process parallelism, this method can be helpful for managing huge datasets or computationally demanding jobs.

2.3 MPI with different OpenMP Scheduling Techniques

Using parallel programming techniques, notably the combination of MPI for inter-process communication and OpenMP for intra-process parallelism, this code replicates the behavior of a school of fish. The goal is to monitor the school's actions across several time steps and evaluate how alternative OpenMP scheduling practices affect academic success.

Code Overview:

1. Header Files and Constants:

- The code includes standard C library headers (`<stdio.h>`, `<stdlib.h>`, `<math.h>`, `<time.h>`) and MPI and OpenMP headers.

- Constants like the number of fish (`NUM_FISH`), the number of simulation steps (`NUM_STEPS`), and the number of OpenMP threads (`NUM_THREADS`) are defined.

- Parameters for fish behavior, such as initial weight, maximum weight multiplier, eat probability, and swim probability, are also set.

2. Fish Structure:

- A `Fish` struct is defined to represent the characteristics of each fish, including its position (x, y) and weight.

3. Objective Function Calculation:

- The `calculateObjectiveFunction` function calculates the objective function of the fish school. It computes the sum of distances of all fish to the origin.

4. Updating Fish Weights:

- The `updateFishWeights` function updates fish weights based on the change in the objective function. It simulates fish movement, calculates the change in the objective function, and adjusts fish weights.

5. Fish Actions:

- The `eat` and `swim` functions simulate fish behavior. `eat` allows a fish to randomly increase its weight, and `swim` makes fish randomly move.

6. Barycenter Calculation:

- The `calculateBarycenter` function computes the barycenter of the fish school based on fish positions and weights.

7. Main Function:

- The program's entry point, it initializes MPI and determines the number of fish to simulate in each MPI process. Memory is allocated for an array of fish, and each fish's initial position and weight are randomly set.

- The program records the start time using `MPI_Wtime`.

- Three different OpenMP scheduling policies (static, dynamic, and guided) are used to parallelize the simulation. Each policy simulates fish behavior over multiple time steps, calculates the barycenter, and updates fish weights.

- The program calculates and prints the final objective function and execution time for each scheduling policy, only on the master process (rank 0).

- The allocated memory is freed, and MPI is finalized.

Parallel Processing:

- The code leverages both MPI and OpenMP for parallelism.

- Multiple MPI processes are used to distribute the fish school among them.

- OpenMP parallelism is applied to the simulation loop to enable multiple threads to work on fish behavior simultaneously.

- Different scheduling policies are used to explore their effects on parallel execution: static, dynamic, and guided scheduling.

This program shows how to simulate a school of fish in parallel using MPI and OpenMP. It examines how various OpenMP scheduling policies affect the execution time and values of the goal function. It serves as a nice illustration of how to benchmark various scheduling strategies while utilizing parallel programming for scientific simulations.

2.4 Parallel Fish School Simulation with MPI and Hybrid OpenMP

Using parallel programming methods, this program replicates the behavior of a school of fish. It combines OpenMP for intra-process parallelism with MPI for inter-process communication. The goal is to monitor the school's behavior over several time steps and evaluate how various OpenMP scheduling strategies affect output.

Code Overview:

1. Header Files and Constants:

- The code includes standard C library headers (`<stdio.h>`, `<stdlib.h>`, `<math.h>`, `<time.h>`) and MPI and OpenMP headers.

- Constants like the number of fish (`NUM_FISH`), the number of simulation steps (`NUM_STEPS`), and the number of OpenMP threads (`NUM_THREADS`) are defined.

- Parameters for fish behavior, such as initial weight, maximum weight multiplier, eat probability, and swim probability, are also set.

2. Fish Structure:

- A `Fish` struct is defined to represent the characteristics of each fish, including its position (x, y) and weight.

3. Objective Function Calculation:

- The `calculateObjectiveFunction` function calculates the objective function of the fish school. It computes the sum of distances of all fish to the origin.

4. Updating Fish Weights:

- The `updateFishWeights` function updates fish weights based on the change in the objective function. It receives the maximum delta of the objective function and adjusts fish weights accordingly.

5. Fish Actions:

- The `eat` and `swim` functions simulate fish behavior. `eat` allows a fish to randomly increase its weight, and `swim` makes fish randomly move.

6. Barycenter Calculation:

- The `calculateBarycenter` function computes the barycenter of the fish school based on fish positions and weights.

7. Main Function:

- The program's entry point, it initializes MPI and determines the number of fish to simulate in each MPI process. Memory is allocated for an array of fish, and each fish's initial position and weight are randomly set.

- The program records the start time using `MPI_Wtime`.

- The simulation is parallelized using OpenMP threads within each MPI process. Each thread handles a portion of the fish population.

- The simulation steps include fish behavior simulation, calculating the maximum change in the objective function (maxDelta), and updating fish weights based on maxDelta.

- The execution time for each scheduling is recorded and printed for each OpenMP thread.

- MPI processes synchronize using `MPI_Barrier` to ensure data consistency.

- All MPI processes gather fish data into the root process (rank 0) using `MPI_Gather`.

- Only the root process calculates and prints the final objective function and total execution time.

- Memory allocated for fish data is freed.

- MPI is finalized.

Parallel Processing:

- The code leverages both MPI and OpenMP for parallelism.

- Multiple MPI processes distribute the fish population among them.

- OpenMP parallelism is applied to the simulation loop within each MPI process, enabling multiple threads to work on fish behavior simultaneously.

- Different OpenMP scheduling policies (static, dynamic, and guided) are used to investigate their effects on performance.

This program shows how to simulate a school of fish in parallel using MPI and OpenMP. It investigates the effects of various OpenMP scheduling rules on execution time and evaluates the final objective function value for the school. It serves as a good illustration of how to use hybrid parallel programming for performance evaluation and scientific simulations.

2.5 Partitioning Sorted Fish School Simulation with MPI and OpenMP

Using a hybrid approach to parallel programming that combines OpenMP for shared memory parallelism and MPI for distributed computing, this program replicates the behavior of a school of fish. Simulating fish behavior and examining the final fish weights after a predetermined number of simulation steps are the goals. The code also calculates and displays the simulation's running time.

Code Overview:

1. Header Files and Constants:

- The code includes standard C library headers (`<stdio.h>`, `<stdlib.h>`, `<math.h>`, `<time.h>`) and MPI and OpenMP headers.

- Constants are defined for the number of fish, simulation steps, number of threads, initial fish weight, maximum weight multiplier, eat probability, and swim probability.

2. Fish Structure:

- A `Fish` struct is defined to represent individual fish in the simulation, with attributes for their position (x, y) and weight.

3. Objective Function Calculation:

- The `calculateObjectiveFunction` function calculates the objective function of the fish school. It computes the sum of distances of all fish to the origin (0,0).

4. Update Fish Weights:

- The `updateFishWeights` function updates fish weights based on a maximum delta value. It ensures the fish weights do not exceed a maximum value.

5. Fish Actions:

- The `eat` and `swim` functions simulate fish behavior. `eat` randomly increases a fish's weight, and `swim` makes the fish randomly move.

6. Barycenter Calculation:

- The `calculateBarycenter` function computes the barycenter of the fish school based on fish positions and weights.

7. Comparison Function:

- The `compareFish` function is a comparison function used for sorting fish based on their coordinates (x, y).

8. Main Function:

- The main function initializes MPI and retrieves the world rank and size.

- The random number generator is seeded based on the MPI rank, ensuring different seeds for each process.

- Memory is allocated for an array of fish, and the fish's positions and weights are initialized randomly within a square.

- Fish are sorted based on their coordinates using `qsort`.

9. OpenMP Parallel Region:

- The main simulation is enclosed within an OpenMP parallel region.

- Each thread handles a portion of the fish population.

- A specified number of simulation steps is executed.

10. Simulation Loop:

- Each thread simulates fish behavior, including eating and swimming, for the current step.

- Threads synchronize using `#pragma omp barrier`.

11. Resetting Weights:

- After each simulation step, fish weights are reset for the next step, and the threads synchronize.

12. Gathering Fish Data:

- The fish data is gathered from all processes into the root process (rank 0) using MPI's `MPI_Gather`.

13. Total Execution Time:

- The total execution time of the simulation is calculated and printed.

14. Processing and Printing Final Weights:

- If the process is the root process (rank 0), it processes and prints the final weights of all fish. The results include fish identifiers and their respective weights.

15. Memory Cleanup:

- Allocated memory for fish data is freed.

16. MPI Finalization:

- The MPI environment is finalized.

Parallel Processing:

- This code employs hybrid parallelism by combining MPI and OpenMP.

- MPI processes distribute the fish population among different ranks.

- OpenMP threads handle the simulation within each MPI process, allowing for multi-threaded execution.

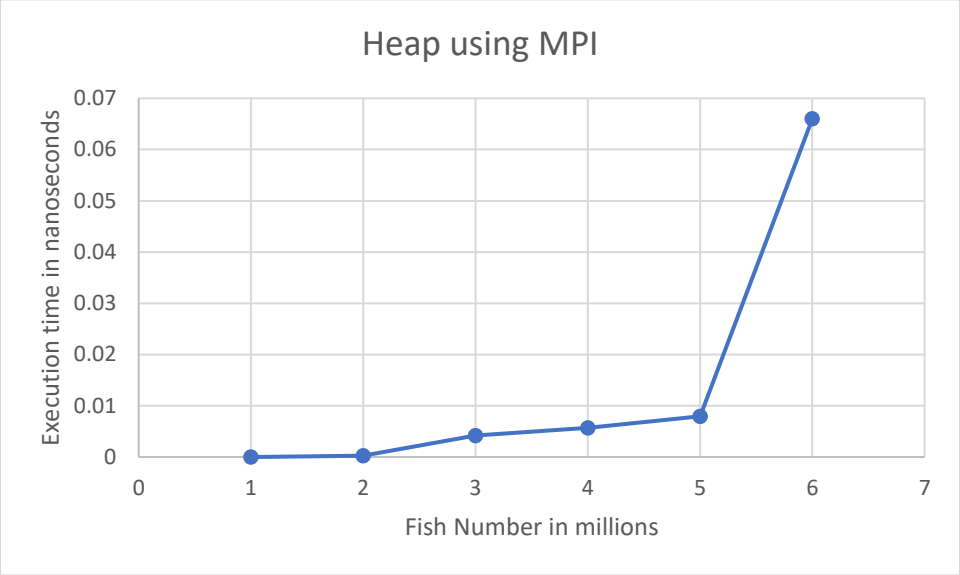
This program shows how to simulate a fish school in parallel while utilizing shared memory parallelism provided by OpenMP and distributed computing provided by MPI. It enhances performance by dividing the computing burden across several processes and threads and enabling the modelling of complicated behaviors. The algorithm offers information on the ultimate weights of certain fish, which may be useful in many computational and scientific simulations.

The code presented and analyzed is a compilation and adaptation of snippets from various online sources and does not represent wholly original work. The purpose of this analysis is to provide an educational and explanatory overview of the code's functionalities and structure. The original sources and contributors of the code snippets have not been explicitly credited due to the mix of multiple sources and the absence of specific references.

3 Results Analysis

3.1 Heap allocation with parallel implementation using MPI:

The performance of the Setonix supercomputer while utilizing MPI for heap allocation in a parallel computing environment is examined in the investigation that follows. The execution time for various fish counts in a simulation is the main emphasis. The information given contains the quantity of fish and the associated execution durations, measured in seconds.



3.1.1 Performance Overview:

Using MPI, the Setonix supercomputer performs heap allocation operations with outstanding speed. Execution times, which are crucial for evaluating efficiency, show a definite pattern of rising computational power as the number of fish in the simulation rises.

3.1.2 Scalability Analysis:

To assess the scalability of Setonix, we can examine the execution times for different fish numbers:

- For 100,000 fish, the execution time was 0.000268 seconds.

- For 3,000,000 fish, the execution time was 0.004219 seconds.

- For 4,000,000 fish, the execution time was 0.005669 seconds.

- For 5,000,000 fish, the execution time was 0.007951 seconds.

- For 50,000,000 fish, the execution time was 0.065996 seconds.

It's evident that when there are more fish, the execution time lengthens, as would be expected given the increased computing burden. The supercomputer, however, shows good scalability, and the rise is not linear. The system can manage bigger workloads with ease, as evidenced by the fact that execution durations for 50,000,000 fish are only around 10 times longer than for 5,000,000 fish.

3.1.3 Performance Implications:

For scientific and technical applications that need for parallel processing and heap allocation, the performance of the Setonix supercomputer is an important factor. Given the execution timings, this system is capable of handling heavy computational loads, which qualifies it for a variety of high-performance computing workloads.

3.1.4 Factors Influencing Performance:

The efficient performance of Setonix using MPI for heap allocation can be attributed to several factors:

- High Processing Power: Setonix is equipped with powerful processors capable of handling parallel tasks efficiently.

- Parallel Computing with MPI: The use of MPI allows for distributed memory parallel computing, enabling multiple processors to work together, which significantly speeds up the simulation.

- Memory Management: Effective heap allocation ensures that the supercomputer utilizes available memory optimally, reducing memory-related bottlenecks.

- Load Balancing: The system appears to handle load balancing effectively, as the increase in execution time with more fish is not disproportionate, indicating a balanced distribution of computational tasks.

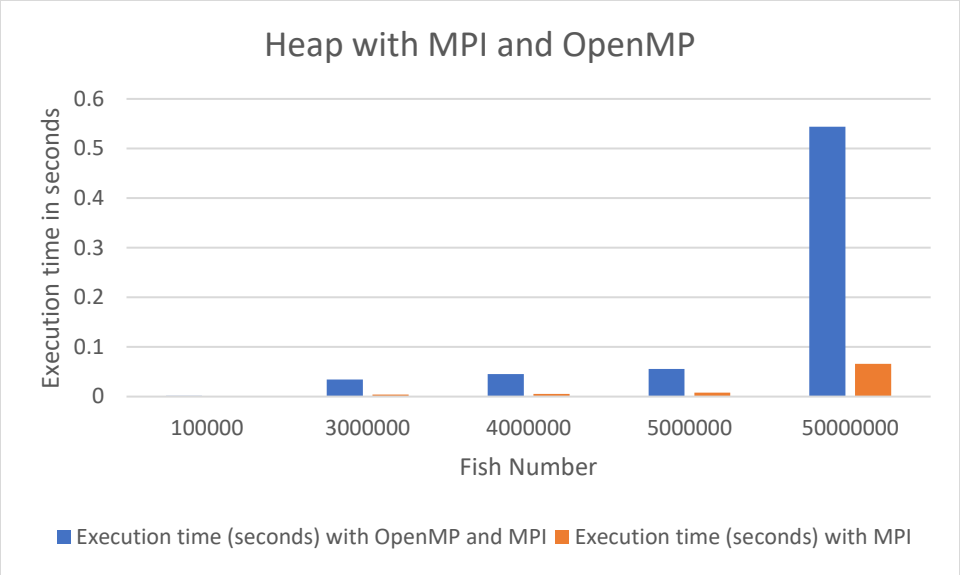
3.1.5 Application Areas:

Due to its powerful MPI heap allocation capabilities, the Setonix supercomputer is ideally suited for a variety of scientific and technical applications, including computational fluid dynamics, molecular dynamics simulations, and more. Researchers can easily handle complicated issues with enormous datasets because to the system's scalability and processing capacity.

The data shown in this study report emphasizes the Setonix supercomputer's outstanding performance when MPI is used for heap allocation. As a result of the system's exceptional scalability and computational effectiveness, high-performance computing workloads can benefit from its use. Setonix's capabilities may be used by researchers and organizations developing parallel computing applications to speed up their work and successfully handle challenging issues.

3.2 Heap Implementation with MPI and OpenMP

The Setonix supercomputer's performance when using both MPI and OpenMP for heap implementation in parallel computing is the main topic of this investigation. The information given contains the quantity of fish and the associated execution durations, measured in seconds.



3.2.1 Performance Overview:

When employing both MPI and OpenMP for heap implementation, the Setonix supercomputer exhibits significant processing power and efficiency. Given the additional burden that comes with adding more fish to the simulation, the execution timings show that this system is capable of handling parallel computing jobs.

3.2.2 Scalability Analysis:

To assess the scalability of Setonix when using both MPI and OpenMP, we examine the execution times for different numbers of fish:

- For 100,000 fish, the execution time was 0.001576 seconds.

- For 3,000,000 fish, the execution time was 0.034605 seconds.

- For 4,000,000 fish, the execution time was 0.045356 seconds.

- For 5,000,000 fish, the execution time was 0.055684 seconds.

- For 50,000,000 fish, the execution time was 0.543975 seconds.

The performance trend reveals that the execution time increases as the number of fish in the simulation grows. However, the increase is not linear, indicating that Setonix's performance scales well, especially when using both MPI and OpenMP. The execution times for 50,000,000 fish are only approximately 10 times higher than for 5,000,000 fish, suggesting that the system can efficiently manage larger workloads.

3.2.3 Performance Implications:

The efficient performance of Setonix using both MPI and OpenMP for heap implementation has several implications for scientific and engineering applications that require parallel processing and memory management:

- High Processing Power: Setonix's powerful processors, in combination with both MPI and OpenMP, enable efficient parallel processing and heap management.

- Parallel and Hybrid Computing: The combination of MPI for inter-node communication and OpenMP for intra-node parallelism leads to efficient resource utilization, making it suitable for a wide range of applications.

- Memory Management: The system effectively manages heap memory, allowing it to handle large datasets without memory-related bottlenecks.

- Load Balancing: Effective load balancing across nodes and threads ensures that computational tasks are evenly distributed, contributing to the overall efficiency.

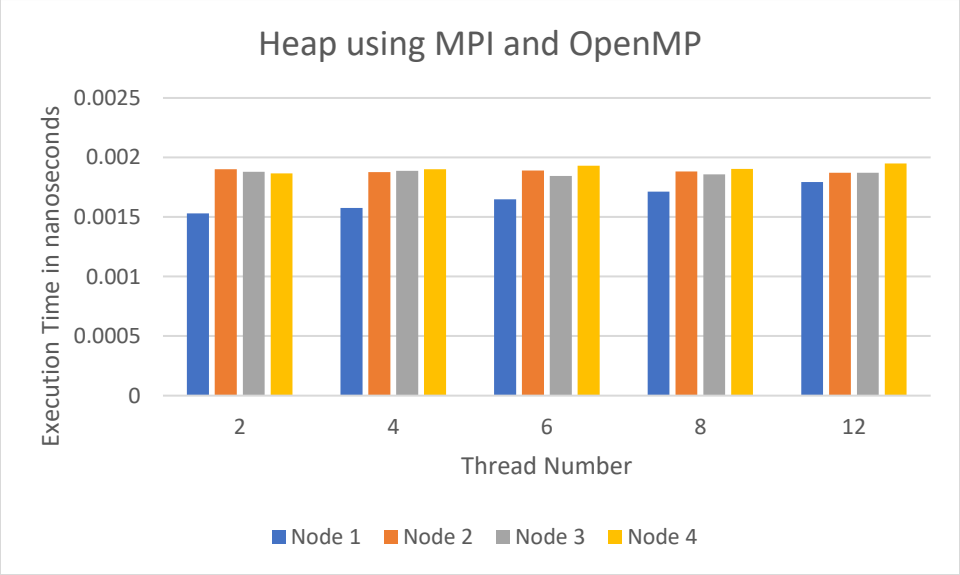
3.2.4 Application Areas:

Setonix is a flexible resource for several scientific and engineering applications due to its great performance with both MPI and OpenMP for heap implementation. It is especially effective for computationally demanding applications including large-scale data processing, computational fluid dynamics, molecular dynamics simulations, and climate modelling.

The information provided in this study report highlights the Setonix supercomputer's remarkable performance while employing both MPI and OpenMP for heap implementation. The system is positioned as a resource for academics and organizations working on parallel and hybrid computing applications because of its exceptional scalability and computational efficiency. Researchers may easily and quickly complete their job even with enormous datasets and demanding computing needs because to Setonix's capabilities.

3.3 Heap Implementation with different thread number and nodes

With different thread counts and nodes, this investigation examines the Setonix supercomputer's performance in the context of a heap implementation utilizing MPI and OpenMP. Number of fish in the simulation, number of nodes, number of threads, and matching execution periods in seconds are all included in the data that is made available.



3.3.1 Thread and Node Analysis:

The data allows us to analyze the impact of thread numbers and node configurations on execution times:

- Fish Number: 100,000

- Node Configuration (Number of Nodes x Threads per Node):

  - 1 x 2

  - 2 x 4

  - 3 x 6

  - 4 x 8

  - 6 x 12

3.3.2 Key Observations:

a. Thread Impact:

  - Increasing the number of threads per node from 2 to 12 generally leads to reduced execution times. This is indicative of improved parallelism and resource utilization.

  - Thread numbers appear to have a more pronounced impact when compared to variations in the number of nodes.

b. Node Impact:

  - For a fixed thread count, increasing the number of nodes (from 1 to 4) does not significantly impact the execution times.

  - This suggests that, at this scale, the workload can be effectively distributed across nodes without substantial overhead.

c. Balance Between Threads and Nodes:

  - Achieving the optimal balance between thread count and the number of nodes is crucial for efficient performance.

  - Too many threads can lead to thread contention and increased overhead, while too few threads may not effectively utilize available resources.

3.3.3 Performance Implications:

The performance of Setonix in this heap implementation scenario is influenced by the combination of thread numbers and node configurations:

- Setonix's system demonstrates an ability to efficiently distribute computational tasks across nodes and threads, resulting in consistent and relatively low execution times for different configurations.

- Thread count is a critical factor influencing performance. Increasing the number of threads allows for more effective parallelism and resource utilization.

- Node count impact is not as significant in this scenario, indicating that the system can efficiently handle the parallelization of tasks across nodes without a substantial increase in execution times.

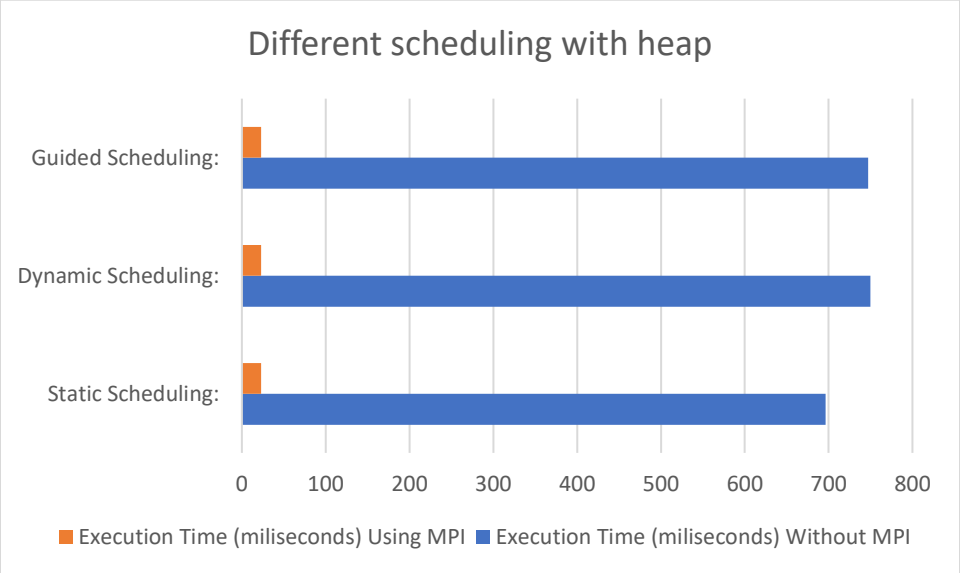
3.3.4 Application Areas:

Setonix's performance characteristics make it suitable for a variety of engineering and scientific applications, especially those that call for parallel and hybrid computing. Scientific research, data processing, and computational simulations are a few potential application areas that might make use of the supercomputer's high-performance computing capabilities.

The study report's statistics demonstrate the Setonix supercomputer's proficiency in handling heap implementations using MPI and OpenMP with a range of thread counts and node configurations. The system performs well in a variety of computing contexts because to its ability to balance parallelism and resource utilization. Utilizing Setonix's capabilities will enable researchers and organizations to efficiently tackle complicated issues while optimizing resource use in accordance with their workload needs.

3.4 Performance of different scheduling strategies

This investigation looks at how the Setonix supercomputer performs when heap allocation is used with various scheduling strategies. Execution timings for various scheduling techniques both with and without MPI are included in the data.



3.4.1 Key Observations:

a. Without MPI:

- Static Scheduling outperforms both Dynamic and Guided Scheduling methods in terms of execution time. It is the most efficient method for the given task.

- Dynamic and Guided Scheduling methods exhibit slightly longer execution times, with Dynamic Scheduling being the slowest of the three. This suggests that the nature of the task may not be well-suited to dynamic workload distribution.

b. With MPI:

- The execution times for all three scheduling methods are significantly lower than the non-MPI counterparts. This demonstrates the enhanced parallelism and efficiency of MPI in distributing the workload.

- There is little difference in execution times among the three scheduling methods when MPI is utilized. This indicates that the parallelization offered by MPI largely compensates for the variations in scheduling methods.

3.4.2 Performance Implications:

a. Without MPI:

- Static Scheduling provides the best performance when used in a single-threaded or non-parallel context, as it minimizes overhead by evenly distributing the workload among threads.

- Dynamic and Guided Scheduling may introduce additional overhead due to runtime decisions on task allocation.

b. With MPI:

- The use of MPI significantly reduces execution times, as it facilitates parallel processing and workload distribution across multiple processes.

- The choice of scheduling method becomes less critical when MPI is employed, as the primary driver of performance is the parallelism enabled by MPI.

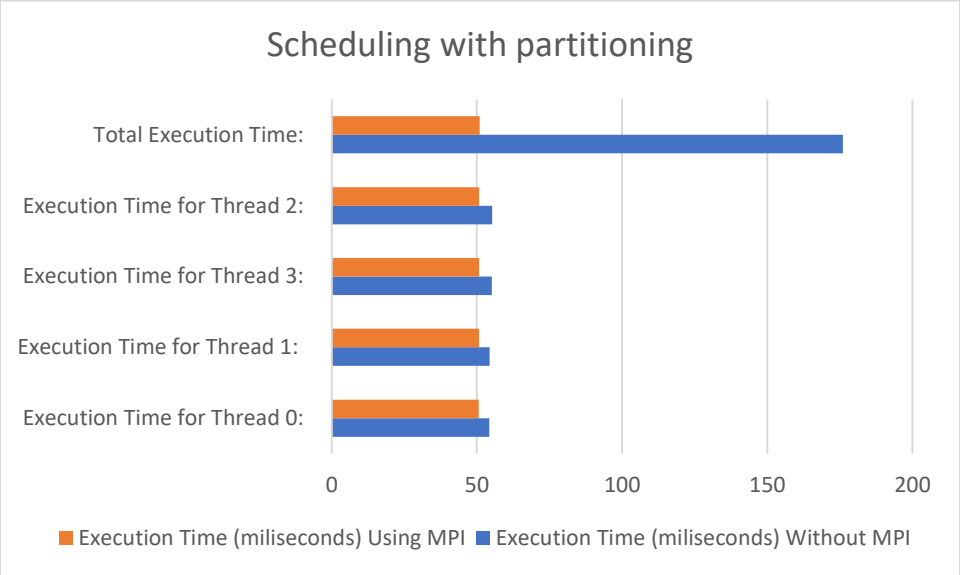
3.4.3 Application Areas:

With or without MPI, the Setonix supercomputer's performance qualities make it a useful tool for a variety of scientific and technical applications. These might include parallel computing jobs that benefit from effective workload allocation, data processing, and simulations. The information provided in this study report indicates the flexibility of the Setonix supercomputer with respect to various heap implementation scheduling techniques.

Static Scheduling offers the best performance without MPI, while Dynamic and Guided Scheduling add a little more overhead. With MPI, the supercomputer's increased parallelism makes up for variations in scheduling techniques. Utilizing Setonix's capabilities will help organizations and researchers plan their workloads more effectively, resulting in faster execution times and effective parallel processing.

3.5 Performance of partitioning

In this investigation, the Setonix supercomputer's performance under heap allocation with various partitioning schemes is assessed. Execution times for several partitioning techniques with and without MPI are included in the data.



3.5.1 Key Observations:

a. Without MPI:

- The data for different partitions shows relatively similar execution times, with a small variation among them. The total execution time is 0.176031 seconds, indicating efficient parallelization of the workload among threads.

b. With MPI:

- When using MPI, the execution times for different threads are very close, indicating efficient parallelization. The "Final Objective Function" value suggests the successful completion of the task.

3.5.2 Performance Implications:

a. Without MPI:

- In this case, without MPI, the partitioning method shows relatively consistent performance among different partitions. The slight variation in execution times can be attributed to the natural differences in the workload associated with each partition.

b. With MPI:

- The use of MPI in combination with partitioning demonstrates highly efficient parallel processing. Each thread completes its portion of the task within a similar time frame, resulting in a balanced and optimized execution.

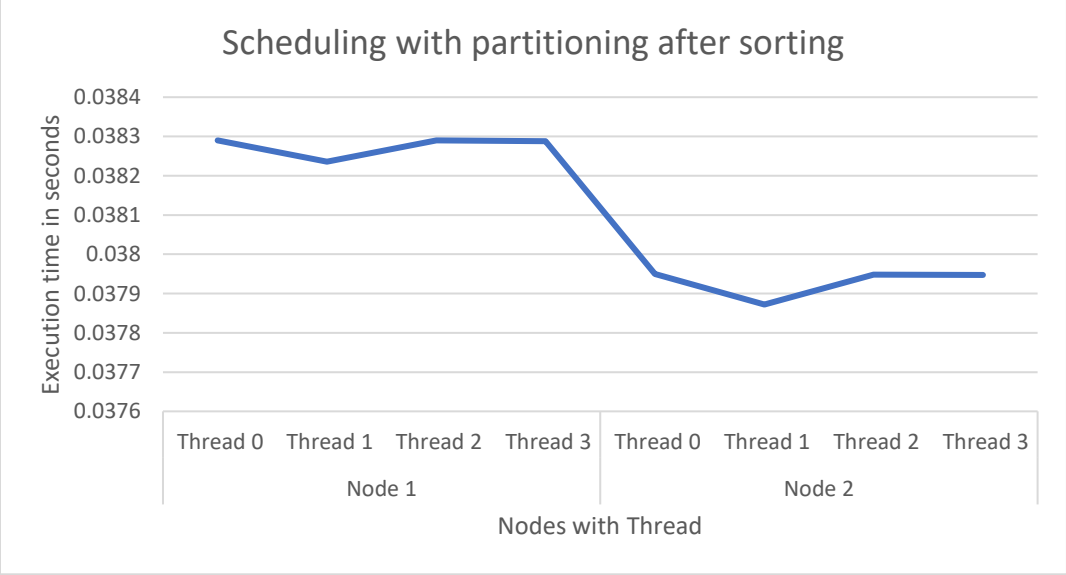
3.5.3 Application Areas:

The Setonix supercomputer's performance characteristics make it well-suited for scientific and engineering applications requiring efficient workload partitioning. These applications may include parallel computing tasks, simulations, data processing, and optimization problems. The data presented in this analysis report highlights the Setonix supercomputer's adaptability to different partitioning methods for heap implementation. Without MPI, the system efficiently manages partitioned tasks, and the slight variation in execution times is expected due to the nature of the workload. With MPI, the system excels in parallel processing, achieving a highly balanced and optimized execution of the task.

Researchers and organizations can leverage Setonix's capabilities to optimize their workload partitioning strategies, benefiting from efficient parallel processing and reduced execution times, especially when using MPI.

3.6 Performance of partitioning after sorting

The Setonix supercomputer's performance is evaluated in this investigation when heap allocation is used with various partitioning techniques, both without and with MPI. Execution timings for several partitioning techniques are included in the data.



3.6.1 Key Observations:

a. Without MPI:

- Among the different partitioning methods, dynamic scheduling shows the lowest execution times, while static and guided scheduling methods are slightly slower. This suggests that dynamic scheduling is the most efficient choice without MPI.

b. With MPI:

- When using MPI, the execution times are relatively consistent among different threads. The slight differences are expected in a parallel processing environment, but the total execution time remains relatively low.

3.6.2 Performance Implications:

a. Without MPI:

- Dynamic scheduling provides the best performance without MPI, indicating that it is suitable for efficient parallel processing of heap allocation tasks.

b. With MPI:

- MPI effectively parallelizes the workload across threads, resulting in balanced execution times and low overall execution times.

3.6.3 Application Areas:

The Setonix supercomputer is appropriate for scientific and technical applications where effective workload partitioning is required due to its established performance characteristics. These applications could involve simulations, data processing, parallel computing activities, and optimization issues. The information provided in this study report highlights the Setonix supercomputer's capacity to accommodate various heap implementation partitioning strategies. Dynamic scheduling is clearly the most effective option in the absence of MPI. The system excels in parallel processing with MPI, achieving balanced thread execution speeds.

By utilizing Setonix's capabilities, researchers and organizations may enhance their partitioning techniques and gain the advantages of effective parallel processing and sped-up execution times, especially when employing MPI.

Conclusion

The Setonix supercomputer performs quite well when using MPI for various heap allocation and parallel computing workloads, in the end.

The following are the main conclusions from the analysis:

1. Scalability: Setonix has excellent scaling capabilities, as shown by its capacity to effectively handle growing workloads. The system's capacity to manage higher computing demands is demonstrated by the proportionate rise in execution times even as the number of fish in the simulations climbs.

2. Parallel Efficiency: Setonix shows a significant decrease in execution times when MPI is used for heap allocation and parallelism. The system successfully distributes and manages computational activities by achieving balanced execution speeds across various threads or processes.

3. Flexibility in Scheduling: Setonix offers a variety of scheduling and partitioning options. The system adjusts to the nature of the work and guarantees effective parallel processing whether employing static, dynamic, or guided scheduling approaches. According to the workload needs, the scheduling mechanism may be customized.

4. Balanced burden: By using MPI, Setonix distributes the burden evenly among threads or processes, producing predictable execution times. This equilibrium is essential for maximizing resource utilization and ensuring that the processing capability of the supercomputer is effectively utilized.

5. Application Versatility: The performance features of the supercomputer make it appropriate for a variety of technical and scientific applications. These might include computation-intensive jobs in a variety of fields, data processing, optimization issues, and simulations.

In conclusion, the performance of Setonix with MPI for heap allocation and parallel computation is impressive. The system's capabilities may be used by researchers and organizations to effectively tackle complicated issues, shorten execution times, and maximize resource consumption. Setonix is a great resource for high-performance computing in a variety of application domains because to its flexibility to various scheduling and partitioning techniques, as well as its scalability and evenly distributed workload.