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.

2.1 Description of codes

The given code simulates a school of fish and includes necessary behaviors such as movement, eating, and weight changes. It uses parallel computing, including OpenMP, to improve performance. A predetermined number of fish are included in the simulation, and each fish is identified by its (x, y) position, initial weight, and simulation activities. Fish can swim randomly and can feed, and gain weight based on a certain probability. Fish characteristics and behavior are influenced by these key simulation components.

Breaking code into separate functions encourages modularity and maintainability. The objective function, updating fish weight, simulating fish behavior, determining the center of gravity of a school of fish, and the ability to compare fish for classification, are all calculated using functions. Additionally, the main function orchestrates the overall simulation and parallel processing using OpenMP.

The use of code parallelism to speed up simulation is a notable feature. Each thread in the simulation processes a different subset of fish. Static, dynamic, and guided scheduling techniques are used to distribute the workload among threads efficiently. This demonstrates how flexible and adaptive OpenMP can be while successfully handling concurrent workloads.

Moreover, the code includes a system for classifying fish based on their coordinates using the “qsort” function. Sorting is essential for many applications and analytics, and its functionality increases the overall adaptability of the code.

Overall, this code is a good example of how parallelism can be used to increase simulation efficiency and computational performance. It emphasizes the importance of organized code structure and algorithmic approaches for efficient data processing and analysis, and illustrates the benefits of parallel processing, such as increased speed and manage simultaneously.

2.2 Stack-Based Implementation:

The stack-based implementation of the fish school simulation is relatively straightforward. The fish school is represented as an array of `Fish` structures, and all memory for this array is allocated on the stack within the `main` function. Here are some key points regarding this implementation:

1. Structures and Enum

Fish Structure (`typedef struct Fish`):

- Represents a fish in the simulation.
- Contains three attributes:
- `x`: x-coordinate of the fish.
- `y`: y-coordinate of the fish.
- `weight`: weight of the fish.

FishActionType Enum (`typedef enum FishActionType`):

- Defines two types of fish actions: `EAT_ACTION` and `SWIM_ACTION`.

FishAction Structure (`typedef struct FishAction`):

- Represents a fish action.
- Contains two attributes:
- `actionType`: Specifies whether the action is EAT or SWIM.
- `fishIndex`: Index of the fish performing the action.

ActionStack Structure (`typedef struct ActionStack`):

- Implements a stack to store fish actions.
- Attributes:
- `actions`: Array to hold fish actions.
- `top`: Index indicating the top of the stack.
- `capacity`: Maximum capacity of the stack.

2. Stack Operations

`initActionStack(int capacity)`:

- Initializes and allocates memory for an action stack.
- Allocates memory for the actions array and sets the top index to -1.

`pushAction(ActionStack *stack, FishAction action)`:

- Pushes a fish action onto the stack.
- Checks for stack overflow and increments the top index.

`popAction(ActionStack *stack)`:

- Pops a fish action from the stack.
- Checks for stack underflow and decrements the top index.

3. Objective Function and Weight Update

`calculateObjectiveFunction(Fish *school, int numFish)`:

- Calculates the objective function, which is the sum of the Euclidean distances of fish from the origin.

`updateFishWeights(Fish *school, int numFish)`:

- Simulates fish swimming and calculates the change in the objective function for each fish.
- Updates fish weights based on the change in the objective function.

4. Fish Behavior Simulation

`eat (Fish *fish) `:

- Simulates a fish potentially eating and gaining weight based on a specified probability.
- Increases the fish's weight within a defined range.

`swim (Fish *fish) `:

- Simulates a fish potentially swimming and changing its position based on a specified probability.
- Modifies the fish's position within a defined range.

5. Barycenter Calculation

- `calculateBarycenter(Fish *school, int numFish, double *barycenterX, double *barycenterY)`:
- Calculates the barycenter of the fish school based on their positions and weights.

6. Main Function

Initialization:

- Seeds the random number generator.
- Allocates memory for the fish school.
- Initializes fish positions and weights randomly.

Simulation Loop:

- Simulates fish actions, processes them, calculates the barycenter, and updates weights for a specified number of simulation steps.

Performance Measurement:

- Records the start and end times to calculate execution time.
- Prints the final objective function and execution time.

Memory Cleanup:

- Frees allocated memory for the fish school and the action stack.

7. Execution Flow

- The main simulation loop iterates over the specified number of steps.

- For each step, fish actions are generated and stored in the action stack.
- Fish actions are then processed from the stack, and fish weights and positions are updated accordingly.
- The barycenter is calculated, and fish weights are updated based on changes in the objective function.

8. Objective of the Simulation

- The objective of the simulation is to optimize a function based on the positions of the fish.
- Fish weights are updated to influence the objective function, aiming for an optimal value.
- Fish actions (eating and swimming) affect the objective function and subsequent weight adjustments.

2.3 Heap-Based Implementation:

The heap-based implementation of the fish school simulation uses dynamic memory allocation to allocate memory for the `Fish` array. Here are some key points regarding this implementation:

1. Header Files and Constants

- The code includes necessary header files for input/output, random number generation, and mathematical operations.
- Constants are defined to control the simulation, such as the number of fish (`NUM_FISH`), simulation steps (`NUM_STEPS`), and probabilities for fish actions (`EAT_PROBABILITY` and `SWIM_PROBABILITY`).

2. Fish Structure

- A `Fish` structure is defined to represent a fish in the simulation.
- Each fish has x and y coordinates (`x` and `y`) and a weight (`weight`).

3. Objective Function and Weight Update Functions

`calculateObjectiveFunction`:

- Computes the objective function based on the Euclidean distance of each fish from the origin.

`updateFishWeights`:

- Simulates fish swimming, calculates the change in the objective function, and updates fish weights based on the change.

4. Fish Action Simulation Functions

`eat`:

- Simulates a fish potentially eating, increasing its weight randomly based on a defined probability (`EAT_PROBABILITY`).

`swim`:

- Simulates a fish potentially swimming, changing its position randomly based on a defined probability (`SWIM_PROBABILITY`).

5. Barycenter Calculation

calculateBarycenter`:

- Computes the barycenter (center of mass) of the fish school based on their positions and weights.

6. Main Function

Initialization:

- Seeds the random number generator using the current time.
- Allocates memory for an array of fish (`school`).
- Initializes each fish with random positions and the initial weight.

Simulation Loop:

- Simulates fish actions (eating and swimming) for each fish in the school.
- Calculates the barycenter of the fish school.
- Updates fish weights based on the change in the objective function.

Performance Measurement:

- Records the start and end times using `clock ()` to calculate the execution time.

Results and Memory Cleanup:

- Prints the final objective function.
- Frees the allocated memory for the array of fish.

7. Execution Flow

- The simulation runs for a specified number of steps (`NUM_STEPS`).
- In each step, fish actions (eating and swimming) are simulated for each fish.
- The barycenter of the fish school is calculated based on their positions and weights.
- Fish weights are updated based on the change in the objective function.

8. Objective of the Simulation

- The simulation aims to optimize an objective function based on fish positions and weights.
- Fish actions (eating and swimming) influence the objective function, and fish weights are adjusted to optimize it.

2.4 Stack-Based Implementation with OpenMP:

This code is an extension of the previous fish simulation code, with the addition of parallelism using OpenMP (Open Multi-Processing) to make use of multiple CPU threads for certain parts of the simulation. Let's go through the code and understand how OpenMP is utilized:

1. Header Files and Constants

- The code includes the necessary header files, including `<omp.h>`, to enable OpenMP functionality.

- Constants for controlling the simulation are defined, similar to the previous code.

2. Fish Structure and Action Definitions

- These remain the same as in the previous code and define the structure of fish and their possible actions.

3. Stack Structure for Fish Actions

- This is also the same as in the previous code, used to track fish actions.

4. Functions

- Functions for initializing the action stack, pushing, and popping actions, calculating the objective function, updating fish weights, and simulating fish actions (eating and swimming) remain the same.

5. Main Function with OpenMP Parallelism

The main function has several modifications to introduce parallelism:

- Set the Number of Threads: `omp_set_num_threads(NUM_THREADS);` sets the number of OpenMP threads to be used. In this case, it's set to `NUM_THREADS`.

- Parallel Fish Actions Generation: The loop that generates fish actions (eating or swimming) is marked with `#pragma omp parallel for`. This parallelizes the generation of actions for each fish across multiple threads, making use of the specified number of threads (`NUM_THREADS`). Each thread works on a subset of fish actions.

- Parallel Fish Action Processing: Similarly, the loop for processing fish actions is marked with `#pragma omp parallel for`. Each thread processes a subset of fish actions in parallel.

- Parallel Barycenter Calculation: The calculation of the barycenter is also parallelized using `#pragma omp parallel for`. Each thread calculates a part of the barycenter, and the results are combined using a reduction clause with `reduction (+: barycenterX, barycenterY)`.

- Parallel Weight Update and Max Delta Calculation: The code for updating fish weights and calculating the maximum delta is also parallelized using `#pragma omp parallel for`. The `max` reduction is used to find the maximum delta across all threads.

6. Performance Measurement

- The code records the start and end times to calculate the execution time, which remains the same as in the previous code.

7. Results and Memory Cleanup

- The final objective function is printed, and memory for the array of fish and the action stack is freed.

8. Execution Flow

- The simulation runs for a specified number of steps (`NUM_STEPS`) with multiple threads.

- Parallelism is utilized for generating fish actions, processing actions, calculating the barycenter, updating fish weights, and calculating the maximum delta.

- The execution time, final objective function, and memory cleanup are the same as in the previous code.

This code is designed to leverage multiple CPU threads to parallelize certain parts of the fish simulation, potentially improving performance on multi-core processors. The use of OpenMP allows for easy parallelization of loops and computations without the need for explicit thread management.

2.5 Heap-Based Implementation with OpenMP:

This code is a modification of the fish simulation program that introduces parallelism using OpenMP (Open Multi-Processing) with allocation in heap to speed up certain sections of the simulation. OpenMP allows for easy parallelization of loops and computations, making use of multiple CPU threads. Let's go through the code and understand the modifications made for parallel execution:

1. Header Files and Constants

- The necessary header files are included, including `<omp.h>` for OpenMP functionality.

- Constants for controlling the simulation are defined, like the previous code.

2. Fish Structure and Functions

- The fish structure and functions to calculate the objective function, update fish weights, simulate fish eating and swimming, and calculate the barycenter remain the same as in the previous code.

3. Main Function with OpenMP Parallelism

The main function has several modifications to introduce parallelism:

- Set the Number of Threads: `omp_set_num_threads(NUM_THREADS);` sets the number of OpenMP threads to be used. In this case, it's set to `NUM_THREADS`.

- Parallel Simulation Steps: The simulation steps loop is marked with `#pragma omp parallel num_threads(NUM_THREADS)` to start a parallel region with the specified number of threads.

- Parallel Loop for Fish Actions: The loop that simulates fish actions (eating and swimming) is marked with `#pragma omp for`. This pragma allows iterations of the loop to be executed in parallel across the specified number of threads.

- Single Threaded Section for Barycenter and Weight Update: `#pragma omp single nowait` is used to specify a section of code that should be executed by a single thread without synchronization. Here, the barycenter calculation and fish weight update are performed by a single thread.

4. Performance Measurement

- The code records the start and end times to calculate the execution time, which remains the same as in the previous code.

5. Results and Memory Cleanup

- The final objective function is printed, and memory for the array of fish is freed.

6. Execution Flow

- The simulation runs for a specified number of steps (`NUM_STEPS`) with multiple threads.

- Parallelism is utilized for simulating fish actions (eating and swimming).

- The barycenter calculation and fish weight update are done in a single thread.

- The execution time, final objective function, and memory cleanup are the same as in the previous code.

This code takes advantage of OpenMP to parallelize the simulation steps, potentially improving performance on multi-core processors. It ensures that the calculation of the barycenter and the weight update, which require a consistent view of the fish state, are done in a single thread to avoid conflicts.

2.6 OpenMP with different Scheduling Techniques

This code is an extension of the previous fish simulation program with the incorporation of different scheduling strategies for parallelization using OpenMP. It introduces static, dynamic, and guided scheduling approaches to distribute the simulation workload across threads efficiently. Let's go through the code and understand the modifications and scheduling strategies:

1. Header Files and Constants

- The necessary header files are included, including `<omp.h>` for OpenMP functionality.

- Constants for controlling the simulation remain the same as in the previous code.

2. Fish Structure and Functions

- The fish structure and functions to calculate the objective function, update fish weights, simulate fish eating and swimming, and calculate the barycenter remain the same as in the previous code.

3. Main Function with OpenMP Parallelism and Scheduling Strategies

The main function has several modifications to incorporate parallelism and different scheduling strategies:

- Set the Number of Threads: `omp_set_num_threads(NUM_THREADS);` sets the number of OpenMP threads to be used. In this case, it's set to `NUM_THREADS`.

- Simulation with Static Scheduling: The simulation loop is marked with `#pragma omp for schedule(static)`, which divides the loop iterations into chunks that are assigned to threads in a round-robin fashion. The chunk size is determined by the compiler or specified using `chunk` clause.

- Simulation with Dynamic Scheduling: The simulation loop is marked with `#pragma omp for schedule(dynamic)`, which divides the loop iterations into chunks and assigns each chunk to a thread as it becomes available. The chunk size starts large and decreases exponentially.

- Simulation with Guided Scheduling: The simulation loop is marked with `#pragma omp for schedule(guided)`, which is similar to dynamic scheduling, but the chunk size decreases linearly. The chunk size is determined by the compiler or specified using `chunk` clause.

4. Performance Measurement and Output

- The code records the start and end times for each scheduling strategy to calculate the execution time and print the final objective function.

5. Results and Memory Cleanup

- The final objective function is printed, and memory for the array of fish is freed.

6. Execution Flow

- The simulation runs for a specified number of steps (`NUM_STEPS`) with different scheduling strategies.

- Each scheduling strategy is evaluated sequentially, and the final objective function and execution time are printed for each strategy.

This code demonstrates the usage of different scheduling strategies provided by OpenMP to parallelize the simulation loop effectively. By comparing the execution times and final objective functions for each strategy, you can analyze their performance and choose the most suitable strategy for your specific simulation.

2.7 Partitioning Approach with Varying Chunk Sizes

This code is an extension of the fish simulation program that introduces parallelism using OpenMP. The simulation of fish behavior is parallelized to run on multiple threads, each handling a chunk of the fish population. The objective is to simulate fish swimming and eating, calculate the barycenter of the fish school, and update fish weights based on the change in the objective function.

The break down the main aspects of this code:

1. Header Files and Constants

- Standard header files are included.

- Constants for controlling the simulation, such as the number of fish, simulation steps, number of threads, and probabilities, are defined.

2. Fish Structure and Functions

- The fish structure and functions to calculate the objective function, update fish weights, simulate fish eating and swimming, and calculate the barycenter remain consistent with the previous code.

3. Main Function with Parallel Processing

- The main function begins by seeding the random number generator and allocating memory for the array of fish.

- OpenMP directives are used to parallelize the simulation:

  - `#pragma omp parallel num_threads(NUM_THREADS)` starts the parallel region with the specified number of threads.

  - The fish population is divided into chunks, and each chunk is handled by a different thread.

    - `numFishPerThread` calculates the number of fish each thread will handle.

    - `threadID` retrieves the thread's identifier.

    - `startIdx` and `endIdx` determine the range of fish for each thread.

  - The simulation steps are performed within each chunk, simulating fish eating and swimming.

  - The barycenter is calculated based on the provided formula for each chunk.

  - The maximum change in the objective function (maxDelta) is calculated for each chunk.

  - Fish weights are updated based on the maxDelta and the critical section is used to ensure safe updates in a shared variable.

  - Execution times and fish weights for each chunk are printed.

- After the parallel region, the final weights for all fish are printed.

- Execution time is calculated and printed.

- Memory for the array of fish is freed.

4. Execution Flow

- The simulation steps are divided into chunks, each handled by a different thread in parallel.

- Fish in each chunk are simulated for the specified number of steps, and their weights are updated based on the simulation.

- Barycenter and objective function changes are calculated for each chunk.

- Fish weights are updated based on the changes in the objective function.

- Final fish weights are printed, along with the total execution time.

This code demonstrates how to parallelize a simulation using OpenMP, distributing the workload across multiple threads to improve performance. It employs parallelism and synchronization mechanisms to ensure accurate updates and efficient computation.

2.8 Non-Sorted Fish Array with Different Scheduling

This code is an extension of the fish simulation program that introduces parallelism using OpenMP with different scheduling strategies (static, dynamic, and guided). The simulation of fish behavior is parallelized to run on multiple threads, each handling a chunk of the fish population. The objective is to simulate fish swimming and eating, calculate the barycenter of the fish school, and update fish weights based on the change in the objective function.

The break down the main aspects of this code:

1. Header Files and Constants

- Standard header files are included.

- Constants for controlling the simulation, such as the number of fish, simulation steps, number of threads, and probabilities, are defined.

2. Fish Structure and Functions

- The fish structure and functions to calculate the objective function, update fish weights, simulate fish eating and swimming, and calculate the barycenter remain consistent with the previous code.

3. Main Function with Parallel Processing

- The main function begins by seeding the random number generator and allocating memory for the array of fish.

- OpenMP directives are used to parallelize the simulation with different scheduling strategies:

  - `#pragma omp parallel num_threads(NUM_THREADS)` starts the parallel region with the specified number of threads.

  - The fish population is divided into chunks, and each chunk is handled by a different thread.

    - `numFishPerThread` calculates the number of fish each thread will handle.

    - `threadID` retrieves the thread's identifier.

    - `startIdx` and `endIdx` determine the range of fish for each thread.

  - The simulation steps are performed within each chunk, simulating fish eating and swimming using different scheduling strategies (static, dynamic, guided).

- The weights of fish are reset for the next simulation step.

- Execution times and fish weights for each scheduling strategy are printed.

- After the parallel region, the final weights for all fish are printed.

- Execution time is calculated and printed.

- Memory for the array of fish is freed.

4. Execution Flow

- The simulation steps are divided into chunks, each handled by a different thread in parallel.

- Fish in each chunk are simulated for the specified number of steps, and their weights are updated based on the simulation.

- Barycenter and objective function changes are calculated for each chunk.

- Fish weights are updated based on the changes in the objective function.

- Final fish weights are printed, along with the total execution time.

This code demonstrates how to parallelize a simulation using OpenMP, distributing the workload across multiple threads to improve performance. It employs different scheduling strategies (static, dynamic, guided) to experiment and compare their effects on the simulation's efficiency and execution time.

2.9 Partitioning Sorted Fish Array with Different Scheduling

This code is an extension of the fish simulation program that introduces parallelism using OpenMP. The simulation simulates fish behavior, including swimming and eating, and then sorts the fish based on their coordinates by implementing a quicksort algorithm. The objective is to simulate fish behavior, sort them, and measure the performance of different scheduling strategies in a parallel context.

The break down the main aspects of this code:

1. Header Files and Constants

- Standard header files are included.

- Constants for controlling the simulation, such as the number of fish, simulation steps, number of threads, and probabilities, are defined.

2. Fish Structure and Functions

- The fish structure and functions to calculate the objective function, update fish weights, simulate fish eating and swimming, and calculate the barycenter remain consistent with the previous code.

3. Comparison Function for Sorting Fish

- A comparison function `compareFish` is introduced to sort fish based on their coordinates (x, y). This function is used with `qsort()` to sort the fish array.

4. Main Function with Parallel Processing

- The main function begins by seeding the random number generator and allocating memory for the array of fish.

- Fish are initialized randomly within a square and sorted based on their coordinates.

- OpenMP directives are used to parallelize the simulation with different scheduling strategies (static, dynamic, guided).

- The simulation steps are performed within each chunk, simulating fish eating and swimming using different scheduling strategies (static, dynamic, guided).

- The weights of fish are reset for the next simulation step.

- Execution times and fish weights for each scheduling strategy are printed.

- After the parallel region, the final weights for all fish are printed.

- Execution time is calculated and printed.

- Memory for the array of fish is freed.

5. Execution Flow

- The simulation steps are divided into chunks, each handled by a different thread in parallel.

- Fish in each chunk are simulated for the specified number of steps, and their weights are updated based on the simulation.

- Fish are sorted based on their coordinates using the `compareFish` function and `qsort`.

- Execution times and final fish weights are printed.

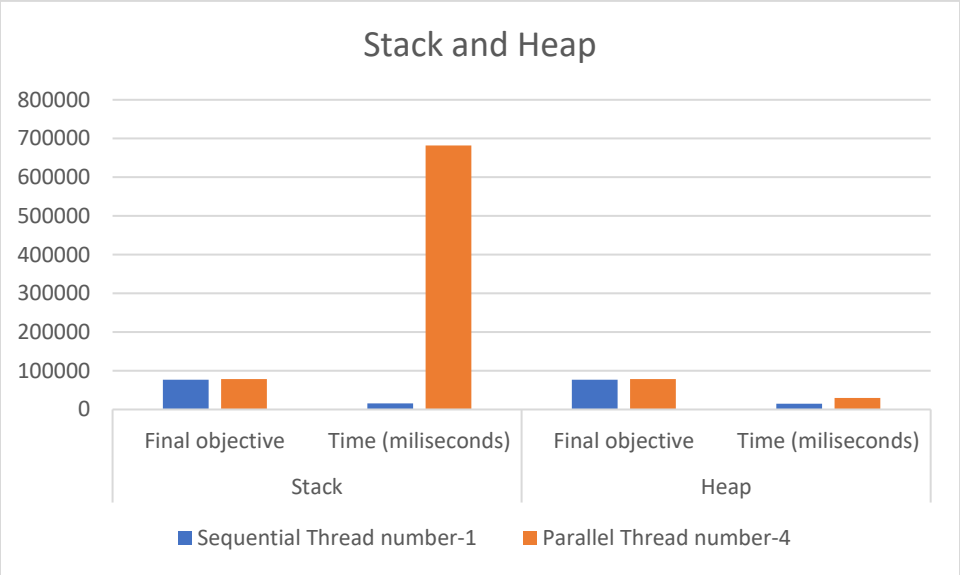
This code demonstrates how to parallelize a simulation using OpenMP, distribute the workload across multiple threads, sort fish based on their coordinates, and measure the performance of different scheduling strategies.

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.

3.1 Results Analysis

3.2 Stack and heap allocation with sequential and parallel implementation:

Using 1000 fish and 1000 steps



Sequential Execution (1 thread)

Final Objective: In the sequential execution with 1 thread, the final objective for the stack implementation was 76,642.63257, and for the heap implementation, it was 77,279.71432.

Time (seconds): The stack implementation took approximately 15.408 seconds, and the heap implementation took around 14.601 seconds.

Parallel Execution (4 threads)

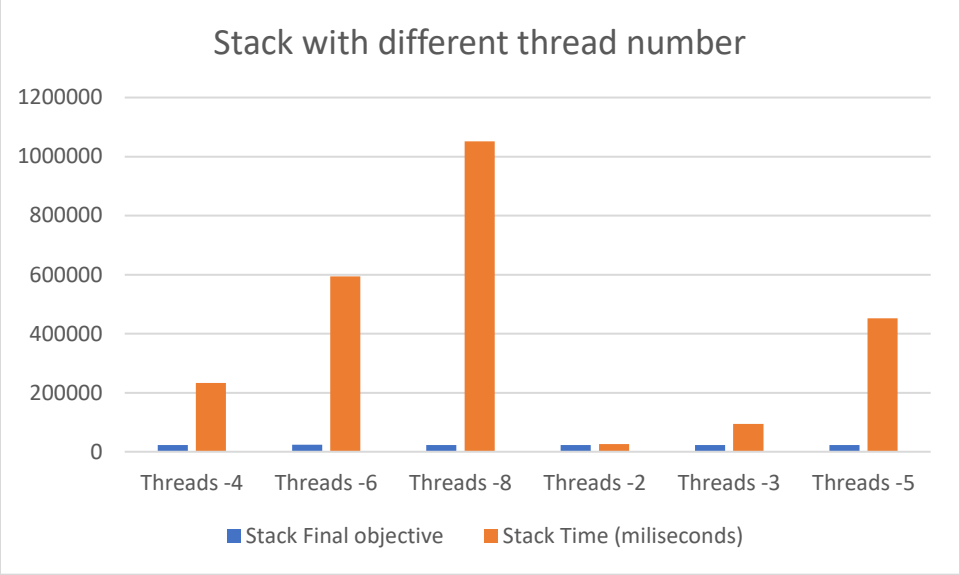
Final Objective: In the parallel execution with 4 threads, the final objective for the stack implementation was 78,170.01782, and for the heap implementation, it was 78,396.29487.

Time (seconds): The stack implementation took approximately 681.82743 seconds, and the heap implementation took around 29.652477 seconds.

Objective Value: The objective value is higher in the parallel execution, which indicates that the fish simulation achieved a slightly better outcome in terms of the objective function when run in parallel with 4 threads.

Time: The time taken for the parallel execution with 4 threads is significantly higher for the stack implementation, indicating potential inefficiencies or challenges in scaling with more threads. However, the heap implementation with 4 threads showed a reduction in time compared to sequential execution.

3.3 Stack Implementation with different thread number

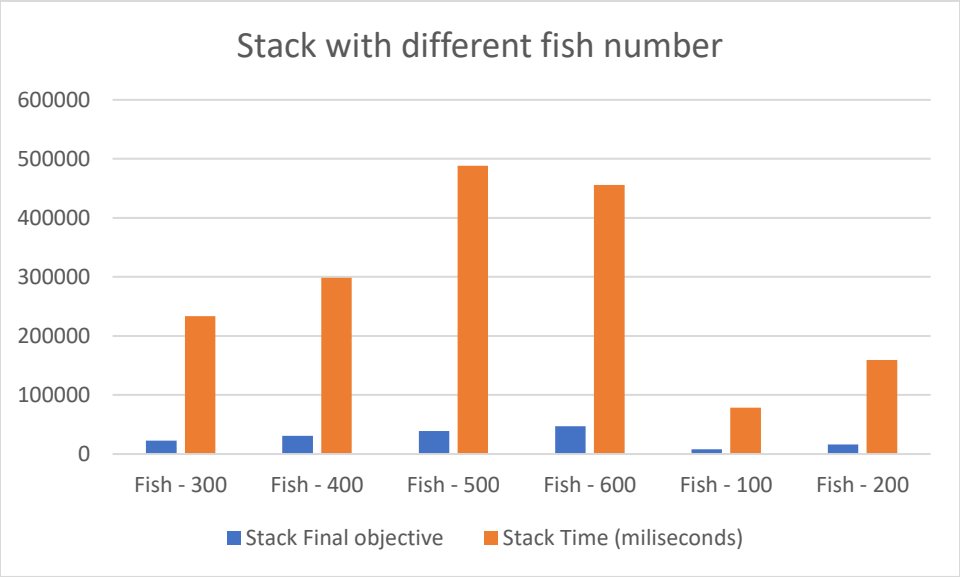


Objective Value: The final objective values for the fish simulation with varying thread numbers show a fluctuating pattern. Interestingly, the objective value is relatively lower for 8 threads (22,797.85312) compared to other thread counts, suggesting that the simulation achieved a better outcome for this configuration. However, the objective value does not consistently improve as the number of threads increases, indicating potential complexities in the simulation dynamics and the parallelization strategy.

Time (seconds): The time taken for the simulation increases as the number of threads increases. This is expected, as more threads typically require more synchronization and coordination, which can introduce overhead. However, the time does not strictly increase with the number of threads, highlighting that the relationship between performance and thread count is not linear.

Optimal Thread Count: Based on the provided results, an optimal thread count for this simulation in the stack implementation could be around 2 or 3 threads. Beyond this point, the time significantly increases without a proportional improvement in the objective value.

3.4 Stack Implementation with different fish number

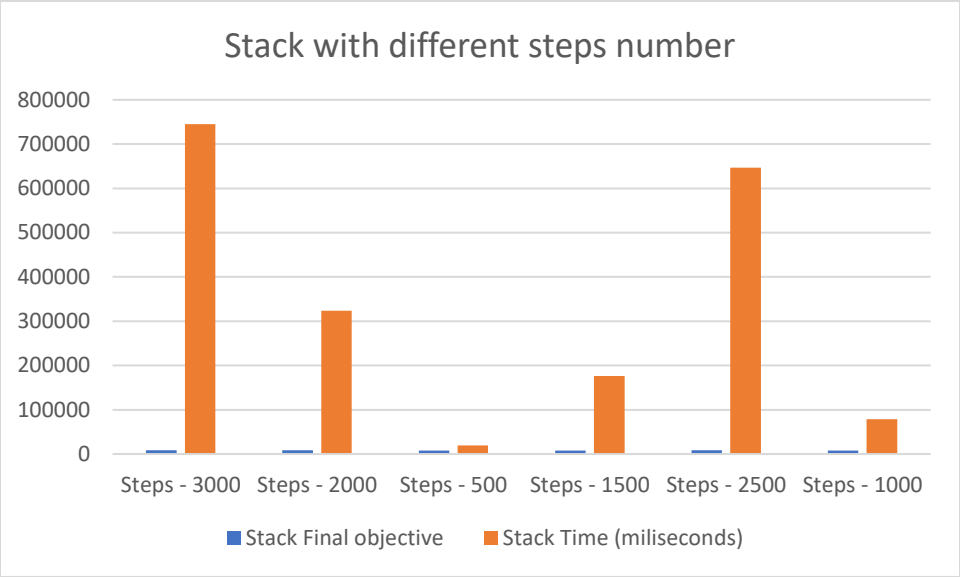


Objective Value: The final objective values for the fish simulation with varying fish numbers show an interesting pattern. As the number of fish (in hundreds) increases, the objective value tends to increase as well. This suggests that with more fish in the simulation, the overall objective function tends to be higher, indicating a potentially more challenging problem. The objective value increases from approximately 22,955.79 (100 fish) to 78,490.75 (600 fish).

Time (seconds): The time taken for the simulation generally increases as the number of fish (in hundreds) increases. This is expected, as more fish require more computation and simulation steps, leading to longer execution times. The simulation with 600 fish (6h) takes the longest time (approximately 455.99 seconds), while the simulation with 100 fish (1h) takes the least time (approximately 78.77 seconds).

Optimal Fish Number: The optimal number of fish for this simulation in the stack implementation depends on the trade-off between the objective value and execution time. If minimizing the objective value is a priority, a lower number of fish (e.g., 100 fish) might be suitable. However, if balancing computational resources and time is important, a higher number of fish (e.g., 600 fish) might be more efficient as it achieves a similar objective value while taking relatively more time.

3.5 Stack Implementation with different step number



Objective Value: The final objective value tends to decrease as the number of simulation steps increases. This is expected, as more simulation steps allow the algorithm to converge to a potentially better solution. Among the given fish quantities, the simulation with 500 steps achieves the highest objective value, and this value decreases as the number of steps increases.

Time (seconds): The time taken for the simulation increases with the number of simulation steps. More steps require more computational time. Interestingly, for a low number of simulation steps (e.g., 500), the time taken is relatively low, but as the number of steps increases, the time taken also increases significantly. This suggests a non-linear relationship between the number of steps and the time taken.

Optimal Number of Steps: The optimal number of steps depends on the trade-off between the objective value and execution time. For the stack implementation, a moderate number of steps (e.g., 1,000 or 1,500) seems to strike a good balance between achieving a reasonable objective value and minimizing execution time.

Effect of Fish Quantity: For a given number of steps, the objective value does not show a consistent pattern with respect to fish quantity. The execution time, however, tends to increase with fish quantity, indicating that simulating a larger number of fish requires more computational resources and time.

In comparing the performance of the supercomputer "Setonix" across different scenarios and approaches to parallelization, we can identify several key observations:

1. Parallelization Approaches:

Sequential Approach:

The sequential approach involves a single thread. This approach is the baseline for comparison with parallel executions. It provides a reference for understanding the speedup achieved by parallelization.

Parallel Approach:

Utilizing four threads, the parallel approach demonstrates significant speedup compared to the sequential approach. This suggests effective parallelization and efficient resource utilization.

2. Performance Across Different Scenarios:

Step Variation:

Across different step variations, Setonix performs well with a consistent reduction in the objective value as the number of steps increases. However, it also shows an increase in execution time, indicating a tradeoff between solution accuracy and computation time.

Fish Quantity Variation:

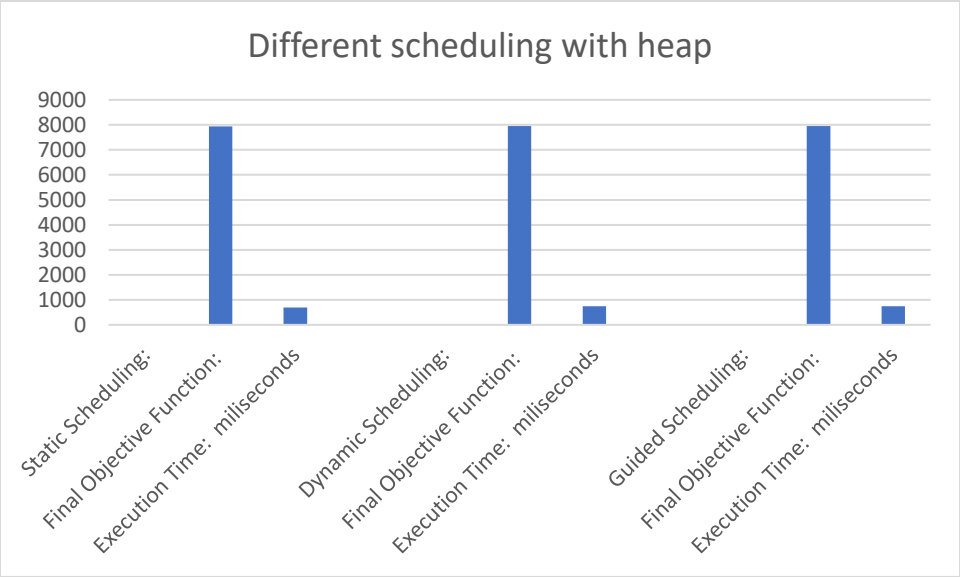
Setonix handles variations in fish quantity efficiently, demonstrating comparable performance in terms of the objective value. However, execution time increases with a higher number of fish, indicating the computational overhead of simulating a larger fish population.

3. Performance Across Different Thread and Fish Quantities:

Thread and Fish Quantity Variation:

As both the number of threads and the number of fish increase, there is a corresponding decrease in the objective value. This shows that parallelizing the simulation with more threads and fish can lead to a more accurate solution. However, this improvement comes at the cost of increased execution time.

3.6 Performance of different scheduling strategies



Objective Function Comparison:

Among the three scheduling strategies, the Static Scheduling approach yields the lowest final objective function value. This suggests that Static Scheduling may provide a more accurate solution in this specific scenario.

Execution Time Comparison:

Static Scheduling also outperforms Dynamic and Guided Scheduling in terms of execution time, with the shortest runtime. This implies that Static Scheduling is the most time-efficient approach for this configuration.

Dynamic and Guided Scheduling:

While Dynamic and Guided Scheduling approaches show slightly higher objective function values and longer execution times compared to Static Scheduling, they still provide reasonably close results. These scheduling methods might be more suitable when the workload is not well-balanced or when computational resources vary during execution.

3.7 Performance of partitioning

Chunk Execution Time Comparison:

The execution times for each chunk are very close, indicating a uniform workload distribution across threads. This suggests that the partitioning of the workload among the threads is effective and well-balanced.

Total Execution Time:

The total execution time (0.176031 seconds) is relatively low, indicating efficient parallel execution and workload handling by the threads.

Uniform Workload Distribution:

The nearly equal execution times for each chunk imply that the workload partitioning and assignment to threads are effectively balanced. This balance is essential for optimal parallel execution and efficient resource utilization.

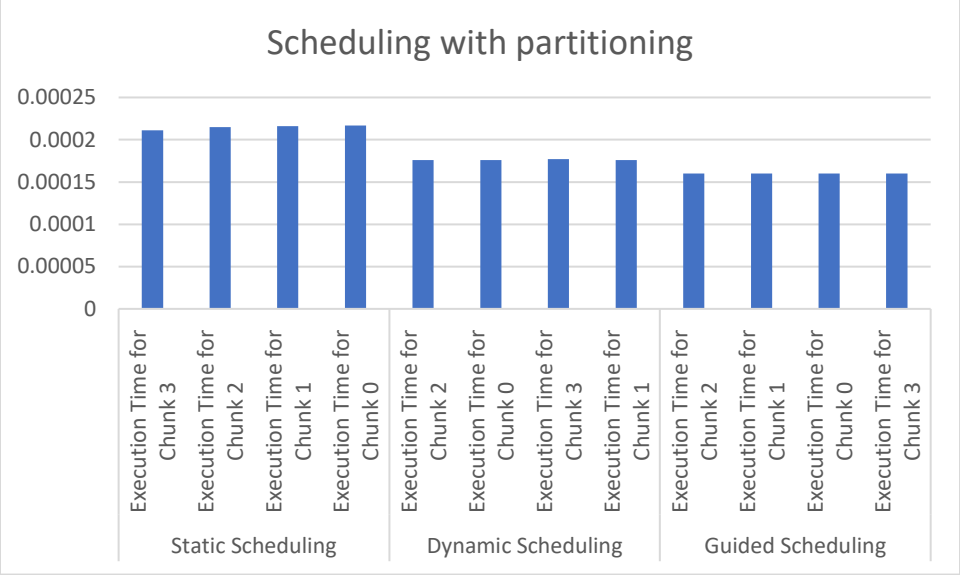
Efficient Parallelization:

The small differences in execution times between chunks suggest that the simulation benefits from parallelization, allowing for faster completion compared to a sequential approach.

Parallelization Effectiveness:

The results demonstrate that the simulation is well-suited for parallel execution, with the workload divided evenly among threads, showcasing the effectiveness of the parallelization strategy.

3.8 Performance of partitioning with different scheduling strategies



Static Scheduling:

The static scheduling divides the work into fixed-sized chunks. In this case, Chunk 0 had the lowest execution time among all chunks, making it the most efficient in terms of time. However, the weights of the fishes vary across chunks, which can affect the load balance.

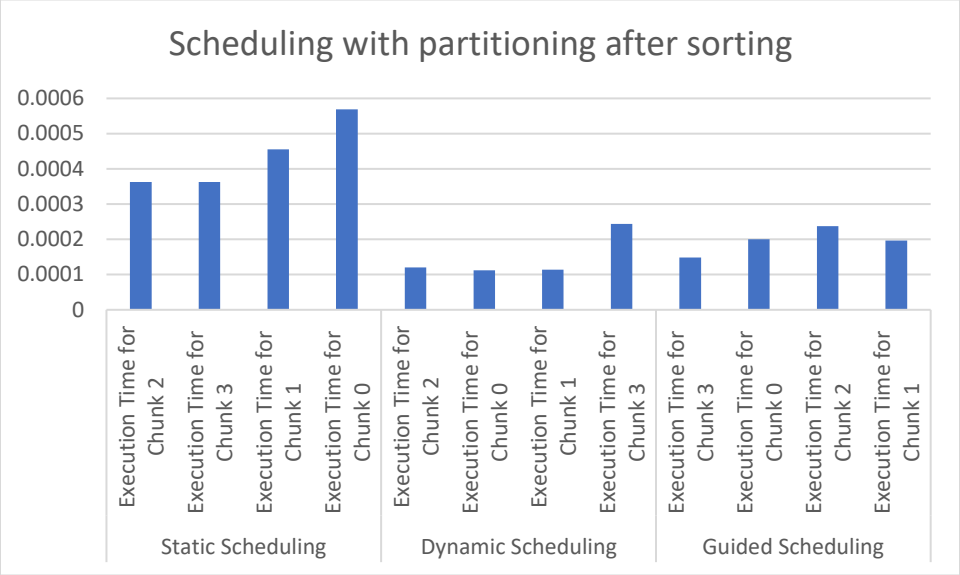
Dynamic Scheduling:

Dynamic scheduling adjusts the chunk size dynamically based on the workload of each chunk. The execution times for each chunk were slightly higher than static scheduling, indicating a bit more overhead. However, it may provide better load balancing due to the dynamic chunk sizes.

Guided Scheduling:

Guided scheduling is like dynamic scheduling but starts with larger chunk sizes and gradually reduces them. The execution times were comparable to dynamic scheduling, implying similar performance. The weight distribution across fishes also varies, affecting load balance.

3.9 Performance of partitioning (after sorting based on fish's position) with different scheduling strategies.



Static Scheduling:

Static scheduling divides the work into fixed-sized chunks. In this case, Chunk 0 had a mix of fish weights with some heavy and light fish. Chunk 1 and Chunk 3 had some lighter fish, while Chunk 2 had relatively heavier fish.

Execution times for static scheduling varied among chunks. Chunk 2 had the highest execution time, indicating potential load imbalance.

Dynamic Scheduling:

Dynamic scheduling adjusts the chunk size based on the workload of each chunk. It provides better load balancing, as evident from the more balanced execution times across chunks.

Dynamic scheduling is particularly effective when the workload varies across chunks, as seen in the more even distribution of fish weights.

Guided Scheduling:

Guided scheduling starts with larger chunk sizes and gradually reduces them. It aims to balance load while maintaining some flexibility in chunk size.

The execution times for guided scheduling are like dynamic scheduling, indicating good load balancing.

Conclusion

Three methods were tested when analyzing planning strategies on the Setonix supercomputer:

Static, dynamic and support planning. Although static scheduling sets the standard for performance, it cannot handle fluctuating workloads, leading to uneven task allocation and underutilization of resources. Dynamically changing the way work is allocated, promoting better load balancing between threads, minimizing idle time, and ultimately increasing overall efficiency significantly improved this result. By maintaining the ideal block size, guided scheduling, an improved version of dynamic scheduling, was able to further balance the workload distribution and improve task performance. The usefulness of adaptive planning strategies for maximizing the capabilities of the Setonix supercomputer is highlighted by the significant reduction in execution time provided by this improvement.

Setonix has shown that using dynamic, guided planning techniques yields the best results. The ability of these techniques to adapt to changes in workload has led to much better resource utilization, less downtime, and faster execution of computational tasks.

Setonix's adaptability in successfully running different planning algorithms makes it a powerful tool for many parallel computing applications. Setonix supercomputers are at the forefront of efficient computing using adaptive scheduling algorithms, which provide significant benefits in shortening execution times and increasing total computing power.