Myfactory

▼ Motivation

Multi-threaded programming has significantly transformed the methodologies employed in addressing computationally intensive problems. Enhancing computational speed is a primary advantage of multi-threading, which is particularly beneficial for tasks like Fast Fourier Transforms (FFT) and parallel Bitonic Sort algorithms.

Another vital application of multi-threaded programming lies in the field of simulations. Simulations play a critical role in various scientific and engineering domains, serving as tools for modeling and analyzing intricate systems.

In the current project, I emulate the production line of an industrial facility, aiming to develop an optimized strategy for augmenting output. Concurrently, it is imperative to ensure that the program design exhibits user-friendliness and reusability, facilitating its application in a broad spectrum of similar scenarios.

▼ Scenario Description

Description from professor Chen

Consider a plant, where there are m part workers (each will be implemented with a thread) whose jobs are to produce five types of parts (A, B, C, D, E). Each part worker will produce 6 pieces of all possible combinations, such as (2,0,0,2,2), (0,2,1,0,3), (6,0,0,0,0), etc., given that it takes 500, 500, 600, 600, 700 microseconds (us) to make each part of type A, B, C, D, E, respectively. For example, it takes a part worker 500**2 +600**3+700 us to make a (2,0,0,3,1) part combination. Each part worker will attempt to load the produced parts to a buffer area, which has a capacity for 7, 6, 5, 5, 4 pieces of type A, B, C, D, E parts, respectively. That is, the buffer capacity is (7,6,5,5,4).

It will take a part worker 200, 200, 300, 300,400 us to move a part of type A, B, C, D, E, respectively, to/from the buffer. Each part combination, such as (1,0,2,2,1) is referred to as a load order.The current number of parts of each type in the buffer, such as (5,2,1,3,2) is referred to as buffer state. A part worker will load the number of parts of each type to the buffer, restricted by the buffer's capacity of each type.For example, if a load order is (2,1,0,2,1) and the buffer state is (7,2,1,4,2), then the part worker can place a type B part, a type D and a type E part to the buffer; thus, the updated load order will be (1,0,0,1,0) and the updated buffer state will be (7,3,1,5,3). The part worker will wait near the buffer area for the buffer space to become available to complete the load order. If the wait time reaches MaxTimePart us (maximum wait time for a part worker), the part worker will stop waiting, move the un-loaded parts back, and randomly re-generate a new load order, which must re-use the previously un-loaded parts (that got moved back). Recall that it takes a part worker 200, 200, 300, 300, 400 us to move a part of type A, B, C, D, E, respectively. A part worker will then repeat the process to produce a brand-new load order.

In addition, there are n product workers (each implemented as a thread) whose jobs are to take the parts from the buffer area and assemble them into products. Each product assembly needs 5 pieces of parts each time; however, the 5 pieces will be from exactly two or three types of parts, such as (1,2,2,0,0), (1,0,3,1,0), (1,0,0,0,4), (0,2,3,0,0), etc. with equal occurrence probability. For example, a product worker will not generate an order of (1,1,2,1,0), (5,0,0,0), etc. Each such legal combination from a product worker is referred to as a pickup order. The time it takes a product worker to move a part of type A, B, C, D, E from the buffer is 200, 200,300,300, 400 us,respectively. Like that for part workers, partial fulfillment policy

is adopted. If the current buffer state is (4,0,2,1,3) and a pickup order is (1,1,0, 0,3), then the updated buffer state will be (3,0,2,1,0) and the updated pickup order will be (0,1,0,0,0). Note that when a product worker goes to the buffer area to pick up parts, the product worker will pick up parts and load them on the cart. At this moment, the numbers of parts on the cart will be (1,0,0,0,3), which is referred to as cart state. The product worker will wait next to the buffer area, looking to complete the pickup order. Once all needed parts are obtained, they will be moved back to assembly area andthen assembled into products. The move time for parts of each type has been described. The assembly time needed for parts of type A, B, C, D, E, will be 600, 600, 700, 700, 800 us, respectively. If the wait time reachesMaxTimeProduct us (maximum wait time for a product worker), the product worker will move back the parts already picked up and randomly re-generate a load order which has to re-use all the parts that were moved back.

The product worker will then re-produce a brand-new pickup order. If the parts moved back after timeout event are (1,1,0,0,0). During the next iteration, if a new pickup order (2,1,2,0,0) is generated, then the real pickup order that the product worker will bring to the buffer area is (1,0,2,0,0), while the parts (1,1,0,0,0), which were brought back during the previous iteration due to timeout event, will stay at local area; (1,1,0,0,0) will be referred to as local state.
Develop a simulation program of the activity of the above-described plant. Your implementation should be designed to improve the performance of the plant, while ensuring a fair treatment to all workers. Each part work or product worker is said to have completed one iteration when a load order or pickup order is completed, or if timeout event occurs such that an order is aborted. Your program should allow each worker to finish 5 iterations.

Clearly you need to protect the shared resource, buffer, with proper lock/mutex. Every time when a part worker thread or a product worker thread gain the access to the shared resource, we need to print information as shown below to a file call log.txt . Note that each product worker thread will also print the total number of completed product.

▼ Class Design and Method Implementation

In order to develop a reusable and abstract design that is user-friendly, I have employed object-oriented principles to structure the implementation. Additionally, I have focused on encapsulation to ensure that the design is modular and self-contained. To accomplish these objectives, I have devised fourclasses that collectively facilitate the simulation of this model in a professional and efficient manner.

▼ productWorkers

The `ProductWorker` class encapsulates the functionality for executing the assembly process described above.

productWorkers(int id, int maxtime):

This is the constructor for the productWorkers class. It takes in two parameters: the worker's ID and the maximum running time. Upon initialization, various member variables are set, including the iteration count, the time to assemble and move parts, the capacity for pickup orders, the pickup order itself, the state of the cart and local state, the time required to move and assemble each part, the current time and accumulated wait time, and the maximum wait time for a product worker. These variables are used to keep track of the product worker's progress throughout the simulation.

void gen_pickupOrder_twotype():

Helper function for generating pickup orders.

void gen_pickupOrder_threetype():

Helper function for generating pickup orders.

void gen_pickupOrder():

Create the orders specified in the process description using the helper functions `gen_pickupOrder_twotype()` and `gen_pickupOrder_threetype()`. These functions aid in generating pickup orders containing either two or three different part types.

void download_buffer(buffer& bu):

The `productWorkers::download_buffer` function facilitates product workers' access to the shared buffer using a unique lock on the buffer's mutex, ensuring thread-safe operations. Workers retrieve required parts, handle waiting times with condition variables, and update buffer and cart states. The function also incorporates timeout management, logs, and prints pertinent information throughout the execution, contributing to an efficient and robust implementation.

void productWorkers::download_bufferGreedy(buffer&bu):

This function, `download_bufferGreedy` is used by each `productWorker` to download parts from the shared buffer in a greedy manner. The function uses a while loop to continuously try to download parts until the worker has reached its `PickupOrderCapacity`. The function first checks if there are any pending notifications from the buffer. If there are notifications, it tries to download the parts in a greedy manner by selecting the notification with the highest priority. It then waits for the corresponding buffer to be non-empty before attempting to download the parts. If the wait times out, the function returns, and the worker moves to the next iteration. Otherwise, the worker downloads the parts and updates its `pickupOrder`, `cartState`, and `PickupOrderCapacity` accordingly. The function also tracks the waiting time of the worker using `AccuWaitTime`. Finally, the function sends a notification to the buffer, signaling that it has downloaded parts and the buffer is no longer full.

void productWorkers::setEnd():

The `productWorkers::setEnd()` function efficiently handles the cleanup and prepar for the next iteration or termination of the process.

void productWorkers::gen_moveTime():

The `productWorkers::gen_moveTime()` function calculates the total time required for a product worker to move the parts in the `cartState`.

void productWorkers::gen_assemble(buffer& bu):

The `productWorkers::gen_assemble(buffer& bu)` function manages the assembly process for product workers, taking into account the parts in `cartState` and `oncelocalState`. It calculates move and assembly times, resets part counts in `cartState` and `oncelocalState`, and increments the `completed_products` counter in the shared buffer. This ensures efficient assembly and prepares the worker for the next iteration or termination of the process.

print / save functions:

Print functions are responsible for outputting the current state of the simulation to the console, enabling real-time monitoring of the simulation's progress.

Save functions, on the other hand, store the simulation state to a file for later analysis or reference.

void rest_Iter()/void each_Inter(buffer& bu, bool greedy)/void Interations(buffer& bu):

These three functions are used to simulate a production iteration

The first function "rest_Iter()" resets the state of the worker after each iteration.

The "each_Inter()" function generates a pickup order, notifies the buffer, and downloads parts from the buffer to the worker's cart. Depending on the value of the "greedy" parameter, the function will use different algorithms for downloading parts. If the worker's cart is full, the function will assemble the parts and start a new iteration.

The "Interations()" function runs the "each_Inter()" function for five iterations using the greedy algorithm.

▼ partWorkers

The **partWorkers** class encapsulates the functionality for executing the assembly process described above.

partWorkers(int id, int maxtime):

The `partWorkers` constructor takes in an ID and maximum time, and initializes various member variables. These include the ID, the iteration number, the capacity and maximum capacity for loading orders, the generation and movement times for parts, the production times and movement times for each part, the current loading order, the current time, the accumulated wait time, and the maximum wait time for parts.

void partWorkers::gen_loadOrder():

. This function generates a random order for loading parts into the production buffer. It uses the `rand()` function from the standard library to generate a random index between 0 and 4, inclusive, and increments the corresponding element of the `loadOrder` vector. The `GenPartTime` variable is also incremented by the amount of time it takes to produce the part that was added to the order. The function continues generating parts until the `loadOrderCapacity` reaches the `loadOrderMaxCapacity`, which is set to 6.

void partWorkers::gen_movePartTime():

The function `gen_movePartTime()` calculates the total time it takes to move all parts in the load order to the assembly line, based on the number of parts in the order and their corresponding move times. It iterates through the `loadOrder` vector, which represents the number of parts for each type in the current load order, and multiplies each quantity by its corresponding move time in the `EachPartMoveTime` vector. The resulting times are added together and stored in the `MovePartTime` variable.

void partWorkers::upload_buffer(buffer& bu):

This function is responsible for uploading the parts produced by the worker into the buffer. It takes a reference to a buffer object as input. The function first acquires a lock on the buffer object using a unique_lock. It then checks if the worker has any parts to upload, and if not, it simply breaks out of the loop.

If the worker has parts to upload, the function waits until the buffer is not full. It does this by checking the `not_empty` flag of the buffer object, and if it is true, it

waits on the `buffer_not_full` condition variable until it is notified or until the maximum wait time (`PartMaxiumWaitTime`) is reached. If the wait times out, the function sets the `overtime` flag to true and breaks out of the loop.

If the wait is successful, the function updates the accumulated wait time and prints and logs the pre-upload state of the buffer. It then proceeds to upload the parts into the buffer. It does this by iterating through the `loadOrder` vector, which represents the number of parts of each type that the worker has to upload. For each part type, the function iterates through the buffer slots of that type and uploads one part at a time until either all the parts of that type have been uploaded or the buffer is full.

After all the parts have been uploaded, the function sets the `not_empty` flag of the buffer object to true and notifies all waiting threads using the `buffer_not_empty` condition variable. It then prints and logs the post-upload state of the buffer.

Finally, the function returns, completing the upload process.

void partWorkers::upload_bufferGreedy(buffer& bu):

This function is similar to `upload_buffer`, but it uses a greedy approach to upload the parts to the buffer. The function takes a `buffer` object as input and uses its mutex lock to access and modify the buffer state.

The function first initializes a boolean variable `overtime` to false and sets the start and end times of the wait interval. The function then enters a while loop that continues until all parts in `loadOrder` have been uploaded to the buffer.

The function then checks if there are any parts in the `smartNotify` priority queue. If there are, it pops the top element of the priority queue and checks which buffer slot corresponds to the part. It then enters a while loop that continues until the corresponding buffer slot is not full.

If the wait time exceeds the maximum wait time `PartMaxiumWaitTime`, the function updates the `AccuWaitTime` variable and skips to the next element in the while loop. If the corresponding buffer slot is not full, the function updates the `AccuWaitTime` variable and enters another for loop that iterates over all elements in `loadOrder`. It then checks whether there are any parts in `loadOrder` that correspond to the current buffer slot, and if so, it uploads those parts to the buffer.

The function then prints out information about the upload process and notifies the appropriate condition variable for the buffer slot.

After all parts have been uploaded to the buffer using the greedy approach, the function enters another while loop that continues until the buffer is not full. The function then operates similarly to the first while loop, waiting until there is space in the buffer to upload more parts.

If the wait time exceeds the maximum wait time `PartMaxiumWaitTime`, the function updates the `AccuWaitTime` variable and skips to the end of the function. If the buffer is not full, the function enters another for loop that iterates over all elements in `loadOrder`. It then checks whether there are any parts in `loadOrder` that correspond to the current buffer slot, and if so, it uploads those parts to the buffer.

The function then prints out information about the upload process and notifies the `buffer_not_empty` condition variable.

print / save functions:

Print functions are responsible for outputting the current state of the simulation to the console, enabling real-time monitoring of the simulation's progress.

Save functions, on the other hand, store the simulation state to a file for later analysis or reference.

void rest_Iter()/void each_Inter(buffer& bu, bool greedy)/void Interations(buffer& bu):

The `rest_Iter` function resets certain information that needs to be cleared before each iteration of the simulation. It resets the generated part time, move part time, accumulated wait time, and clears the smart notification queue.

The `each_Inter` function represents the main logic of each iteration in the simulation. It generates a load order and notification order, calculates the time it takes to move parts, records the current time, prints pre-upload information, and then calls either the `upload_buffer` or `upload_bufferGreedy` function, depending on whether or not a greedy approach is being used. After uploading the parts, it

calculates the time it takes to move the parts again, records the current time, and increments the iteration counter.

The `Interations` function calls `each_Inter` five times in a row, with a greedy approach being used each time. It represents the overall simulation of the system.

▼ buffer

This is the implementation of the `buffer` class, which represents the buffer shared by the production lines in the factory. The buffer has a `mutex` to ensure thread-safe access and five `condition_variable` objects to signal when a buffer is full or empty.

The buffer also has five flags (`a_not_empty`, `b_not_empty`, `c_not_empty`, `d_not_empty`, and `e_not_empty`) to track whether each of the five parts has products waiting to be processed in the buffer. The `completed_products` variable keeps track of the total number of products completed.

In addition, the buffer has a `bufferState` vector, which stores the current number of products for each part in the buffer. The `bufferCapacity` vector stores the maximum number of products that can be stored for each part in the buffer.

▼ myFactory

The `myFactory` class contains methods for running the factory simulation and calculating statistics based on the results.

myFactory(int m, int n):

This is the constructor for the `myFactory` class. It takes two integer arguments `m` and `n` and initializes the `NumPartWorkers` and `NumProduceWorkers` member variables of the class to those values. These variables represent the number of worker threads for producing parts and producing finished products, respectively.

void stimulation(int i):

The `stimulation()` function is the main function of the `myFactory` class that runs the simulation for the specified number of iterations.

The function takes an integer `i` as its argument, which specifies the number of times the simulation needs to be run. The for loop in the function runs `i` times.

Inside the loop, a new `buffer` object `bu` is created, and the vectors `PartW`, `ProductW`, `PartThread`, and `ProductThread` are cleared to make sure they are empty before starting the simulation.

Next, for each of the `NumPartWorkers` and `NumProduceWorkers`, a `partWorkers` and `productWorkers` object is created, respectively, with the specified maximum times. These objects are stored in the `PartW` and `ProductW` vectors.

After that, threads are created for each of the `partWorkers` and `productWorkers` objects, and these threads are stored in the `PartThread` and `ProductThread` vectors, respectively.

Then, the simulation is run by calling the `Interations()` function for each of the `partWorkers` and `productWorkers` objects, passing in the `buffer` object `bu` by reference.

Finally, after the simulation completes, the number of completed products is stored in the `cache` vector, after converting it to a `double` type.

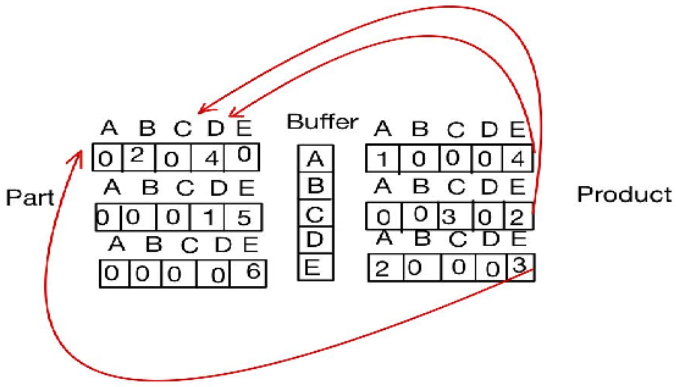
These are helper functions used by the `myFactory` class to calculate statistical measures of the completed products generated by the simulation.

- `mean(const std::vector<double>& values)` calculates the arithmetic mean of the input vector of values.
- `variance(const std::vector<double>& values)` calculates the variance of the input vector of values.
- `standard_deviation(const std::vector<double>& values)` calculates the standard deviation of the input vector of values, which is the square root of the variance.
- `max_value(const std::vector<double>& values)` returns the maximum value in the input vector of values.

- `min_value(const std::vector<double>& values)` returns the minimum value in the input vector of values.

▼ Strategy exploration

▼ Baseline:Strategy/ Chaos



In the original design, the wakeup mechanism lacked a coherent strategy and was essentially random. As depicted in the provided example, waking up a thread with the intention of improving its performance may have a high probability of success, but it may not necessarily improve production efficiency, leading to a wasteful utilization of resources.

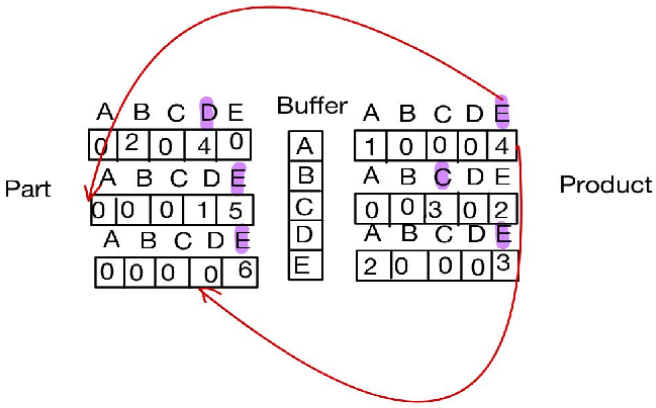
simulations result:

I have conducted 100 simulations with 10 part workers and 10 product workers：

```
********************************
Mean: 17.3
Sd: 7.53724
Min:6
Max:28
********************************

C:\Users\wangx\source\repos\hw3\x64\Debug\hw3.exe (process 2648) exited with code 0.
Press any key to close this window . . .
```

▼ New Strategy / Greedy



In the latest strategy, we implemented a wakeup/notify mechanism based on a greedy approach, using a priority_queue data structure. This mechanism wakes up threads based on their potential to provide the highest production/demand, thus reducing idle times and improving overall production efficiency.

Under the previous strategy, threads were randomly woken up, resulting in wasted resources and suboptimal production. However, with the new mechanism, the priority_queue allows us to select the most suitable thread to wake up and complete a task, which leads to a significant reduction in idle times and an increase in overall production capacity.

simulations result:

I have conducted 100 simulations with 10 part workers and 10 product workers：

```
********************************
Mean: 24.5
Sd: 4.23764
Min:9
Max:50
********************************

C:\Users\wangx\source\repos\Project36\x64\Debug\Project36.exe (process 21188) exited with code 0.
Press any key to close this window . . .
```

By implementing the new greedy wakeup/notify mechanism, we observe a remarkable performance improvement in terms of the average(17.3→24.5) and standard deviation(7.5→4.2). The results show that we can reach the theoretical maximum of **50** and achieve a much more efficient production system. This optimization is significant in terms of resource utilization and overall system performance.

▼ Conclusion

I have gained knowledge in the following areas from this well-designed project:

1. Multithreading: The project heavily relies on multithreading to enable concurrent execution of different parts of the simulation, such as the workers, buffers, and production steps.
2. Object-oriented programming: The project uses object-oriented programming (OOP) principles to encapsulate data and behaviors into classes such as `buffer`, `partWorkers`, and `productWorkers`.
3. STL containers and algorithms: The project utilizes the Standard Template Library (STL) containers and algorithms, such as `vector`, `mutex`,

`condition_variable`, `accumulate`, `min_element`, `max_element`, and `sort`.

4. C++11 features: The project makes use of some C++11 features, such as `chrono` for time measurement and `atomic` for thread-safe operations.
5. Simulation techniques: The project employs simulation techniques to model the production line and simulate the behavior of the workers and buffers. Specifically, it uses discrete event simulation, where each event represents a change in the state of the system, and Monte Carlo simulation, where random events are introduced to the simulation to capture real-world variability.
6. Statistical analysis: The project also performs statistical analysis on the results of the simulation to gain insights into the performance of the production line. This includes calculating metrics such as mean, standard deviation, minimum, and maximum.