|  |
| --- |
| Multicore Computing |

-project #1 : problem1–

측정기이(가) 표시된 사진

자동 생성된 설명

|  |  |
| --- | --- |
| 과목명 | 멀티코어 컴퓨팅 |
| 교수명 | 손봉수 교수님 |
| 제출일 | 2023.04.16 |
| 학 번 | 20200283 |
| 학 과 | 소프트웨어학부 |
| 이 름 | 조범희 |

INDEX(2023 1st Semester Multicore computing)

1. Project Environment/Overview
2. Analysis

* Table
* Graph
* Static load balancing(BLOCK)
* Stack load balancing(CYCLIC)
* Dynamic load balancing

1. Conclusion

* Summary
* Full Results Screenshot

1. Project Environment/Overview

* Environment
* OS : Window11
* Processor : 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz
* Memory : Ram 8GB
* IDE : IntelliJ IDEA 2022.1.1
* Overview
* Detecting the Number of Prime Numbers from 1 to 200,000 Using Static Load Balancing (Block, Cyclic), Dynamic Load Balancing, and Multi-Threaded Approach
* The attached code can be run on Java IDE such as IntelliJ or Eclipse.This is the java file.

1. Analysis

* Result Table

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Exec time | 1 | 2 | 4 | 6 | 8 |
| Static(Block) | 2770 | 2118 | 1353 | 1165 | 1008 |
| Static(Cyclic) | 2735 | 1752 | 1566 | 1320 | 1031 |
| Dynamic | 3040 | 2513 | 1267 | 1079 | 983 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Exec time | 10 | 12 | 14 | 16 | 32 |
| Static(Block) | 977 | 1131 | 1114 | 1042 | 904 |
| Static(Cyclic) | 953 | 848 | 1008 | 914 | 911 |
| Dynamic | 992 | 901 | 884 | 835 | 975 |

Task size = 10, Unit: ms

* Result Graph

차트이(가) 표시된 사진

자동 생성된 설명 차트이(가) 표시된 사진

자동 생성된 설명

<Static load balancing (BLOCK) Execution Time, Performance>

차트이(가) 표시된 사진

자동 생성된 설명 차트이(가) 표시된 사진

자동 생성된 설명

<Static load balancing (CYCLIC) Execution Time, Performance>

차트이(가) 표시된 사진

자동 생성된 설명 차트이(가) 표시된 사진

자동 생성된 설명

<Dynamic load balancing Execution Time, Performance>

* Static load balancing (BLOCK)

텍스트이(가) 표시된 사진

자동 생성된 설명 텍스트이(가) 표시된 사진

자동 생성된 설명

<Thread #8> < Thread #10>

* Static load balancing is a technique where the workload is assigned at compile time. In other words, the programmer allocates the work in the code using a decomposition approach. Block-based load balancing may not be the best choice for this technique, but it is a low-overhead approach. On the other hand, cyclic-based load balancing may result in significant overhead, but it is a good choice for load balancing.
* Block-based load balancing is a technique where the given workload is divided into blocks based on the number of threads available for processing. In this problem, we allocated 200,000 numbers to each thread. This approach is relatively easy to implement and uses a multi-threaded approach, but it is not an efficient load balancing technique. As shown in the results above, we can see that the execution time among threads is significantly different, indicating poor load balancing.
* Looking at the table above, the longer the number of threads, the shorter the program execution time, but more than 10 times, it was similar or rather longer. The computer's process has four cores and eight logical threads, which are thought to be due to the excessive number of threads, resulting in excessive switching and overhead of threads.

텍스트이(가) 표시된 사진

자동 생성된 설명 텍스트이(가) 표시된 사진

자동 생성된 설명

* The main function declares a thread and allocates a block. It then determines the prime number from the "run" of the thread and adds a value to the global variable "counter".
* Static load balancing (CYCLIC)

텍스트이(가) 표시된 사진

자동 생성된 설명 텍스트이(가) 표시된 사진

자동 생성된 설명

< Thread #8> < Thread #10>

* Although it is a static load balancing method as above, the cyclic method is also affected by the number of threads and the number of tasks. Since there are 10 tasks here, 1 to 10, 11 to 20.. Repeat and assign to each thread. Looking at the execution time of each thread above, it can be seen that the load balancing is significantly improved compared to the block method. In addition, the overall time also showed better performance in cycles than in blocks.
* Looking at the table, we can see that the performance improves up to 12 threads, but doesn't improve much beyond that. This is likely due to the same reason as the block method. The difference is that while the block method didn't see much improvement beyond 10 threads, this time we see improvement up to 12 threads. This is likely due to better load balancing with the cyclic method.

텍스트이(가) 표시된 사진

자동 생성된 설명 텍스트이(가) 표시된 사진

자동 생성된 설명

* The task allocation code has become more complex. In the case of block allocation, we only needed to divide END\_NUM by the number of threads, but here we need to divide the number of tasks and assign them in order. In this code, we gave each thread an arrayList<arrayList<int>> to allocate the work items. I was worried that using ArrayList would slow down the performance due to access time, but it didn't have as much impact as I thought.
* Dynamic load balancing

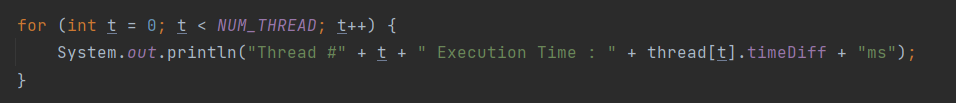
텍스트이(가) 표시된 사진

자동 생성된 설명 텍스트이(가) 표시된 사진

자동 생성된 설명

<Thread #8> < Thread #10>

* Dynamic load balancing, unlike static load balancing, is assigned tasks at runtime. In other words, it is not clear exactly what work will be assigned to each thread because the programmer created only creates situations. In addition, the difference between dynamic load balancing and static load balancing is the synchronization problem. It is important to synchronize properly because threads continue to access the same resources.
* Looking at the screenshot, we can see that load balancing is significantly better than static load balancing cyc method. The workload is evenly distributed across threads, resulting in improved performance for up to 16 threads, which is a significant improvement compared to the static method. In my opinion, dynamic load balancing may have incurred synchronization overhead, so I expected the execution time to be slower even with load balancing. However, the results were better than expected.



텍스트이(가) 표시된 사진

자동 생성된 설명 텍스트이(가) 표시된 사진

자동 생성된 설명

* In this implementation, unlike static load balancing, there is no code assigned by the main. Because it is assigned at runtime. If you look at the code, since there are 10 Tasks, you work 10 times each, determine the prime number, and update the isPrime variable through the update function. This is because isPrime is a common variable between threads. If a synchronization problem occurs here, the result will be a fatal error.

1. Conclusion

* Multi-thread programming shows faster performance compared to single threads. However, you should use a composition method for each situation. There were parts that fit my expectations and parts that didn't fit, but I think I need to study more deeply.
* All Results Screenshot

텍스트이(가) 표시된 사진

자동 생성된 설명 텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명 텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명 텍스트이(가) 표시된 사진

자동 생성된 설명

달력이(가) 표시된 사진

자동 생성된 설명 텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명 텍스트, 명판이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

<Static load balancing(Block) Execution Time ScreanShot>

텍스트이(가) 표시된 사진

자동 생성된 설명 텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명 텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트, 명판이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

<Static load balancing(Cyclic) Execution Time ScreanShot>

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명달력이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

<Dynamic load balancing Execution Time ScreanShot>