Program Structures & Algorithms

INFO 6205 Fall 2019

Assignment 5

1. Parallel Sort Code

In this assignment, we rewrite the ParSort.java code to make multiple threads to execute the program in order to see whether using parallel sort can perform better than only one thread. Before showing the result and conclusion of this assignment, we should know that this assignment’s result is largely depend on computer processors, where my computer processor is 2.8 GHz quad-core Intel Core i7, MacBook.

In the code part, we are supposed to implement the ParSort.java and completed the parallel sort function as sort(result, 0, to - from); On the other hand, in the sort part, we considered 4 different sort conditions as xs1 and xs2 also have boundary conditions. I also changed a little of the Main.java to finish this experiment, I chose 3 sizes of the cutoff value from 10000 to 1000000 which will illustrate the efficiency of the parallel sort on different cutoff value; For the array size, I also selected 30000 to 1920000 with each time I double the array size in a for loop.

1. Run-Time Graph & Conclusion Between Array and Cutoff

Conclusion 1 : From the first “Array Size - Run Time Graph”, we can see that as the value of the array size increased by a factor of two, the processing time increased at a much slower rate. Even in the second part of the data where array size = 6000, processing time had also dropped a little. Therefore, we can determine that there had no linear relationship between the amount of the array size processed by the parallel sorting and the processing time O(n). When array size = 1920000 which is 64 times of 3000, the running time just 10 times bigger than the beginning, which indicated that parallel sorting performs better on big data scale.

图片包含 文字, 地图

描述已自动生成

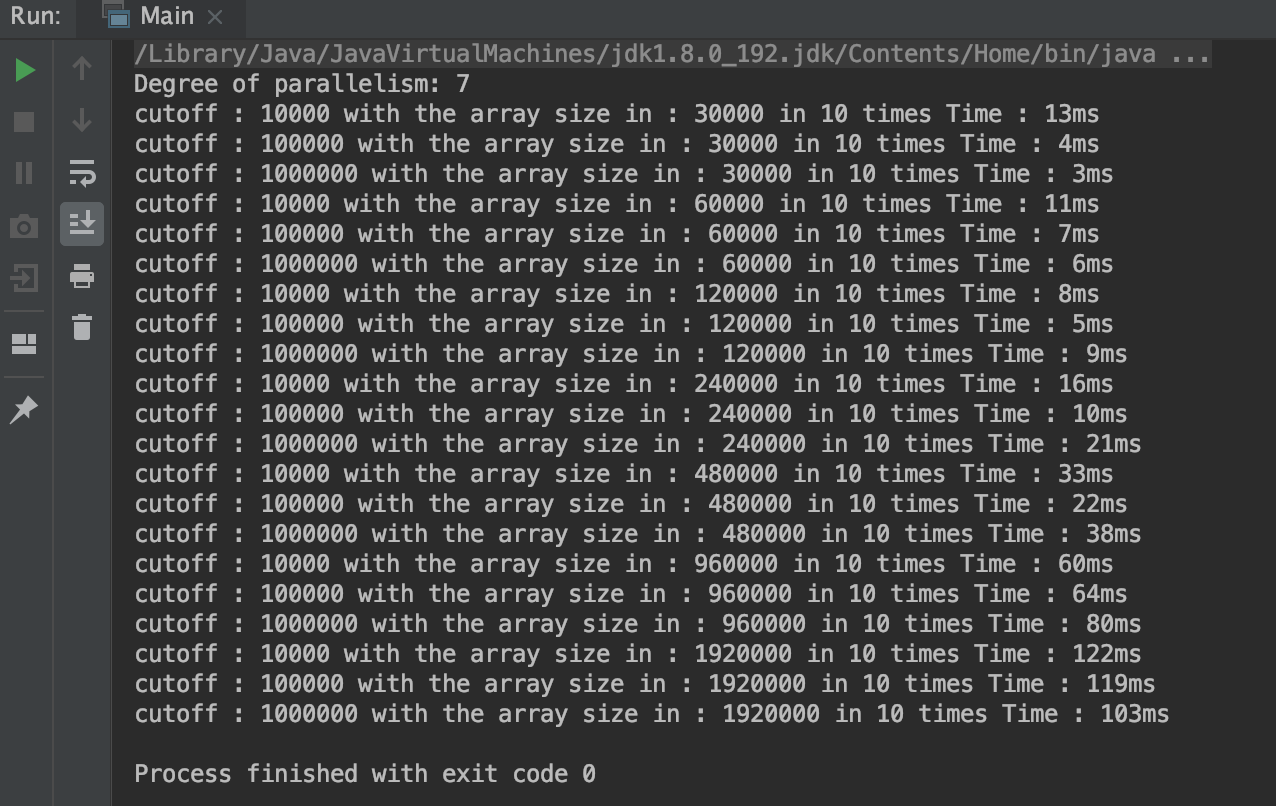
Array Size - Run Time Graph

图片包含 室内, 计算机

描述已自动生成

Cutoff Size - Run Time Graph

Conclusion 2 : From the second “Cutoff Size - Run Time Graph”, We can find that the value of the cutoff size and the amount of the array size will both affect the overall run time result. Parallel sorting did not work well when cutoff size is larger than the amount of the array size while the array size is very small (when array size = 30000). Therefore, in this case, system sorting may be a better choice. But when the cutoff size increased, the run time result got closer and closer as ignored the array size (when array size = 1920000). From this we can concluded that when using the parallel sorting, we should control the ratio of array size to cutoff size to be greater than 1 but smaller than 10 which will optimize the parallel sorting better.



Output of The Code

1. Multiple Thread & Conclusion

I also changed the degree of parallelism to different values, as my maximum value for thread is 7, I selected thread as 3, 5, 7 for test. To realize this change, I used Executor object to get different newFixedThreadPool with parameter 3, 5, 7. I changed the array size to 800000, 160000 and 320000 with the cutoff values from 1020000 to 2000000 where the cutoff values is almost 10 times of the array size, which is related my previous conclusion.

图片包含 计算机, 监视器, 室内, 笔记本电脑

描述已自动生成

Degree of parallelism = 3

图片包含 计算机, 监视器, 室内, 笔记本电脑

描述已自动生成

Degree of parallelism = 5

图片包含 计算机, 监视器, 室内, 笔记本电脑

描述已自动生成

Degree of parallelism = 7

From these 3 pictures we can drew the conclusion that the runtime of the program is depended on the array size, the lower bound of the cutoff size and the upper bound of the cutoff size, where the relationship is:

page1image66419328

Parameter Relational Graph

The data in red were the runtime that is the smallest between all the runtime in that section, although it may be a little bit different because of the arrangement of the random arrays. But when we see the last graph (when Degree of parallelism is 7), there occurred a big increasement as cutoff size is bigger than 1600000, all the runtime went to 1700ms, no matter how big the array size was and became stable after that.

