**Report**

Firstly, we have to consider the scheme that when the length of array is less than the cutoff, we use system sort instead. Actually, it’s easy to make it. We can test different cutoffs but we have to know that the cutoff is related to the original length of array. I means that when the given array is huge, the cutoff we have to come up with is also big. So in my program, I pass the value of cutoff to Parsort’s static method to sort the randomly generated array. And the formula to set the cutoff is “*cutoff* = 10000 \* (j + 1)”(50<j<100).

Secondly, we have to think about the recursion depth. We can know that for our computers, generally the default parallelism is 7 which can be proved by my code’s running result. And I tried to test the different efficiencies of the parsort with different parallelism. I set the concurrent thread count as 8,4 and 2 respectively. Then when a processor splits the task into two parts and pass each one of them to another processor, the count is divided by two. When the count is 1, it means that there are 8 threads solving the smallest problem concurrently, and it is no way to divide the problem again. And in my code, I will pass the (count/2) to the thread which receives the smaller task and this is my way to implement my idea.

|  |  |  |  |
| --- | --- | --- | --- |
| 0.005 | 114 | 117.1 | 189 |
| 0.01 | 89 | 90.2 | 178.7 |
| 0.015 | 89.1 | 85.9 | 168.1 |
| 0.02 | 91.8 | 84.5 | 165.1 |
| 0.025 | 90.6 | 85.9 | 168.6 |
| 0.03 | 87.1 | 86 | 167 |
| 0.035 | 87.5 | 82.8 | 166.5 |
| 0.04 | 92.4 | 84.1 | 169.9 |
| 0.045 | 92.1 | 82.8 | 170.8 |
| 0.05 | 89.4 | 85.2 | 175.2 |
| 0.055 | 96.8 | 87.5 | 178.6 |
| 0.06 | 90.7 | 88.5 | 182.3 |
| 0.065 | 90.6 | 92.2 | 170.7 |
| 0.07 | 85.6 | 88 | 169 |
| 0.075 | 90.7 | 86 | 179.4 |
| 0.08 | 88.8 | 84.3 | 181.3 |
| 0.085 | 85.9 | 92.1 | 191.5 |
| 0.09 | 87.5 | 87.2 | 184.3 |
| 0.095 | 85.9 | 89 | 180.2 |
| 0.1 | 86 | 85.1 | 174.3 |

This is the result I got. And you can see that the second column is the result when the threadcount is 8, the third one is the result when the count is 4,and finally the fourth is the result when the count is 2. The results of the first two are close. But the difference between the last one’s result and another one’s is quite big.

Generally speaking, when the parallelism is bigger, the result should be more satisfying. So why the results of the first two are close? Actually in my code, the length of array is 2000000 and the cutoff is equal to or bigger than 510000. This means that the array is keeping to split into 2 parts until the length of subarray is 500000, for example. At this time, we can see the count of thread is divided for two times, And whatever the count is 8 or 4, we don’t have to use system sort after confirming that count is equal to 1. Before it, this one, “(size < *cutoff*)”, help us to call the system sort. So there is no big difference between the results of 8 and 4. But the case is totally different when the thread is 2. So we can see that the running time is longer when the thread count is 2.

And I want to show the efficiency of parsort so I use merge sort to make a comparison.

|  |  |
| --- | --- |
| 0.005 | 301.5 |
| 0.01 | 291.5 |
| 0.015 | 283.1 |
| 0.02 | 278 |
| 0.025 | 276.8 |
| 0.03 | 275.3 |
| 0.035 | 276.4 |
| 0.04 | 278.7 |
| 0.045 | 275.3 |
| 0.05 | 275.6 |
| 0.055 | 279.6 |
| 0.06 | 281.7 |

As you can see, parsort is two times faster than merge sort. So we can see that we can deal with the problems using the parallelism scheme which is faster than the normal methods. And in some cases, when the length of array is small enough, just use the system sort rather than use a complicated method.