**SP Programming assignment 4 Report**

**B03902086 李鈺昇**

1. **The data of 24 running results (seconds)**

(The output is shown in Appendix A.)

Size = 100:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Seg. size | 1 | 4 | 10 | 20 | 50 | 100 |
| Real | 0.013 | 0.004 | 0.003 | 0.003 | 0.005 | 0.003 |
| User | 0.003 | 0.001 | 0.001 | 0.001 | 0.001 | 0.001 |
| Sys | 0.011 | 0.003 | 0.002 | 0.002 | 0.001 | 0.002 |

Size = 10000:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Seg. size | 100 | 400 | 1000 | 2000 | 5000 | 10000 |
| Real | 0.027 | 0.017 | 0.014 | 0.013 | 0.010 | 0.009 |
| User | 0.022 | 0.016 | 0.013 | 0.012 | 0.009 | 0.007 |
| Sys | 0.009 | 0.003 | 0.002 | 0.002 | 0.001 | 0.001 |

Size = 1000000:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Seg. size | 10000 | 40000 | 100000 | 200000 | 500000 | 1000000 |
| Real | 1.591 | 1.248 | 1.143 | 0.947 | 0.761 | 0.679 |
| User | 1.787 | 1.425 | 1.293 | 1.068 | 0.832 | 0.664 |
| Sys | 0.032 | 0.017 | 0.013 | 0.010 | 0.009 | 0.005 |

Size = 10000000:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Seg. size | 100000 | 400000 | 1000000 | 2000000 | 5000000 | 10000000 |
| Real | 17.301 | 15.872 | 12.379 | 10.781 | 7.716 | 7.395 |
| User | 18.414 | 16.659 | 13.698 | 11.796 | 8.534 | 7.108 |
| Sys | 0.377 | 0.363 | 0.262 | 0.188 | 0.076 | 0.064 |

1. **My observation and discovery**

If the input size is fixed (look at each individual table), then the trend is that as the segment size becomes larger, both user time and system time decrease, and hence so does real time. I think the main reason lies in the decrease of the number of segments when segment size becomes larger. Smaller number of segments means fewer times of merging two sorted lists, which is the main bottleneck of merge-sort. Although sorting with fewer segments seems less parallel, the sorting part is not what slows down the whole process. In fact, the time of merging dominates the time of sorting.

Next, if the ratio of the size of segments to the input size is fixed (look at each columns across the four tables), it seems that the real time is in proportion to the input size. This is reasonable without any question.

A question is here: why is the real time, in some cases, less than the sum of CPU time and system time? Soon I realized. Since we’re running a multi-thread program, the CPU time is the sum of all the running time of the separate CPUs, while the real time may be just around the maximum of them. This is why CPU time can exceed real time in this context.

1. **Experiment 1: The growth of time as segment size becomes smaller**

(The output is shown in Appendix B.)

When the input size is 10000000, the minimal segment size we are asked to run is 100000, which results in the longest total real time. I wonder how the real time changes as the segment size becomes smaller. I fix input size at 10000000, and let the segment size be 10000000, 1000000, 100000, …, each term being the previous term divided by ten. The result is as follows:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Seg. size | 10000000 | 1000000 | 100000 | 10000 | 1000 | 100 |
| Real | 7.591 | 11.178 | 17.652 | 21.086 | 18.639 | 159.473 |
| User | 7.216 | 12.898 | 19.127 | 23.454 | 19.536 | 13.334 |
| Sys | 0.076 | 0.137 | 0.326 | 0.420 | 0.698 | 202.654 |

\*Segment sizes under 10 are not listed, because they took too much time ( > 30mins ) without results.

Clearly, cases where segment sizes are greater than or equal to 1000 are not too far from one another. However, the case where segment size is 100 took explosive system time (user time is still fine). Two or three minutes are much longer than other cases, so the case of 10 must be a lot longer than the listed cases\*. Thus 100 might be near the critical value where the running time grows suddenly faster.

1. **Experiments 2: The comparison of “long” and “int”**

(The output is shown in Appendix C.)

At first, I encounter the problem of conversion, or type casting, between int and void\*. Converting between int and void\* generated warnings, which I disliked. Then I found that this problem could be resolved if int was replaced with long. Then everything really works fine. But I wonder how the performance of them differ, so I changed long back to int, ignoring the warnings, and then re-complie, re-run again.

At first I guessed that int version would be faster than long version. Surprisingly, the long version is faster overall, in many cases! Then I thought the reason behind this might be the time spent on converting between int and void\*, which may take longer time than between long and void\*.

**Appendix A: Original output**

size: 100, seg\_size: 1

real 0m0.013s

user 0m0.003s

sys 0m0.011s

============================================

size: 100, seg\_size: 4

real 0m0.004s

user 0m0.001s

sys 0m0.003s

============================================

size: 100, seg\_size: 10

real 0m0.003s

user 0m0.001s

sys 0m0.002s

============================================

size: 100, seg\_size: 20

real 0m0.003s

user 0m0.001s

sys 0m0.002s

============================================

size: 100, seg\_size: 50

real 0m0.005s

user 0m0.001s

sys 0m0.001s

============================================

size: 100, seg\_size: 100

real 0m0.003s

user 0m0.001s

sys 0m0.002s

============================================

size: 10000, seg\_size: 100

real 0m0.027s

user 0m0.022s

sys 0m0.009s

============================================

size: 10000, seg\_size: 400

real 0m0.017s

user 0m0.016s

sys 0m0.003s

============================================

size: 10000, seg\_size: 1000

real 0m0.014s

user 0m0.013s

sys 0m0.002s

============================================

size: 10000, seg\_size: 2000

real 0m0.013s

user 0m0.012s

sys 0m0.002s

============================================

size: 10000, seg\_size: 5000

real 0m0.010s

user 0m0.009s

sys 0m0.001s

============================================

size: 10000, seg\_size: 10000

real 0m0.009s

user 0m0.007s

sys 0m0.001s

============================================

size: 1000000, seg\_size: 10000

real 0m1.591s

user 0m1.787s

sys 0m0.032s

============================================

size: 1000000, seg\_size: 40000

real 0m1.248s

user 0m1.425s

sys 0m0.017s

============================================

size: 1000000, seg\_size: 100000

real 0m1.143s

user 0m1.293s

sys 0m0.013s

============================================

size: 1000000, seg\_size: 200000

real 0m0.947s

user 0m1.068s

sys 0m0.010s

============================================

size: 1000000, seg\_size: 500000

real 0m0.761s

user 0m0.832s

sys 0m0.009s

============================================

size: 1000000, seg\_size: 1000000

real 0m0.679s

user 0m0.664s

sys 0m0.005s

============================================

size: 10000000, seg\_size: 100000

real 0m17.301s

user 0m18.414s

sys 0m0.377s

============================================

size: 10000000, seg\_size: 400000

real 0m15.872s

user 0m16.659s

sys 0m0.363s

============================================

size: 10000000, seg\_size: 1000000

real 0m12.379s

user 0m13.698s

sys 0m0.262s

============================================

size: 10000000, seg\_size: 2000000

real 0m10.781s

user 0m11.796s

sys 0m0.188s

============================================

size: 10000000, seg\_size: 5000000

real 0m7.716s

user 0m8.534s

sys 0m0.076s

============================================

size: 10000000, seg\_size: 10000000

real 0m7.395s

user 0m7.108s

sys 0m0.064s

============================================

**Appendix B: Output of experiment 1**

(The last case didn’t complete, and was terminated.)

size: 10000000, seg\_size: 10000000

real 0m7.591s

user 0m7.216s

sys 0m0.076s

============================================

size: 10000000, seg\_size: 1000000

real 0m11.178s

user 0m12.898s

sys 0m0.137s

============================================

size: 10000000, seg\_size: 100000

real 0m17.652s

user 0m19.127s

sys 0m0.326s

============================================

size: 10000000, seg\_size: 10000

real 0m21.086s

user 0m23.454s

sys 0m0.420s

============================================

size: 10000000, seg\_size: 1000

real 0m18.639s

user 0m19.536s

sys 0m0.698s

============================================

size: 10000000, seg\_size: 100

real 2m39.473s

user 0m13.334s

sys 3m22.654s

============================================

size: 10000000, seg\_size: 10

**Appendix C: Output of experiment 2**

size: 100, seg\_size: 1

real 0m0.010s

user 0m0.003s

sys 0m0.010s

============================================

size: 100, seg\_size: 4

real 0m0.006s

user 0m0.001s

sys 0m0.004s

============================================

size: 100, seg\_size: 10

real 0m0.004s

user 0m0.001s

sys 0m0.002s

============================================

size: 100, seg\_size: 20

real 0m0.012s

user 0m0.001s

sys 0m0.002s

============================================

size: 100, seg\_size: 50

real 0m0.005s

user 0m0.001s

sys 0m0.002s

============================================

size: 100, seg\_size: 100

real 0m0.003s

user 0m0.001s

sys 0m0.001s

============================================

size: 10000, seg\_size: 100

real 0m0.026s

user 0m0.022s

sys 0m0.009s

============================================

size: 10000, seg\_size: 400

real 0m0.018s

user 0m0.017s

sys 0m0.003s

============================================

size: 10000, seg\_size: 1000

real 0m0.017s

user 0m0.014s

sys 0m0.003s

============================================

size: 10000, seg\_size: 2000

real 0m0.013s

user 0m0.011s

sys 0m0.002s

============================================

size: 10000, seg\_size: 5000

real 0m0.010s

user 0m0.009s

sys 0m0.001s

============================================

size: 10000, seg\_size: 10000

real 0m0.009s

user 0m0.007s

sys 0m0.001s

============================================

size: 1000000, seg\_size: 10000

real 0m1.648s

user 0m1.838s

sys 0m0.036s

============================================

size: 1000000, seg\_size: 40000

real 0m1.378s

user 0m1.570s

sys 0m0.021s

============================================

size: 1000000, seg\_size: 100000

real 0m1.179s

user 0m1.316s

sys 0m0.018s

============================================

size: 1000000, seg\_size: 200000

real 0m1.014s

user 0m1.126s

sys 0m0.015s

============================================

size: 1000000, seg\_size: 500000

real 0m0.785s

user 0m0.857s

sys 0m0.010s

============================================

size: 1000000, seg\_size: 1000000

real 0m0.703s

user 0m0.691s

sys 0m0.008s

============================================

size: 10000000, seg\_size: 100000

real 0m18.917s

user 0m20.368s

sys 0m0.430s

============================================

size: 10000000, seg\_size: 400000

real 0m16.482s

user 0m16.893s

sys 0m0.443s

============================================

size: 10000000, seg\_size: 1000000

real 0m12.898s

user 0m14.062s

sys 0m0.289s

============================================

size: 10000000, seg\_size: 2000000

real 0m11.529s

user 0m12.234s

sys 0m0.272s

============================================

size: 10000000, seg\_size: 5000000

real 0m10.083s

user 0m9.877s

sys 0m0.258s

============================================

size: 10000000, seg\_size: 10000000

real 0m9.613s

user 0m8.044s

sys 0m0.213s

============================================

**Appendix D: Shell script to generate Appendix A, C**

(in10, …, in10000000 are the random input files.)

#!/bin/bash

size[0]=100

size[1]=10000

size[2]=1000000

size[3]=10000000

nseg[0]=100

nseg[1]=25

nseg[2]=10

nseg[3]=5

nseg[4]=2

nseg[5]=1

for i in {0..3}

do

for j in {0..5}

do

echo "size: ${size[$i]}, seg\_size: $((${size[$i]}/${nseg[$j]}))"

time ./merger $((${size[$i]}/${nseg[$j]})) < in${size[$i]} > /dev/null

echo "============================================"

done

done

**Appendix E: Shell script to generate Appendix B**

#!/bin/bash

size[0]=10000000

size[1]=1000000

size[2]=100000

size[3]=10000

size[4]=1000

size[5]=100

size[6]=10

size[7]=1

for i in {0..7}

do

echo "size: 10000000, seg\_size: ${size[$i]}"

time ./merger ${size[$i]} < in10000000 > /dev/null

echo "============================================"

done