**The title of the university: HDU-ITMO**

**Group member: Wang Xinyue, Fan Xulin, Shi Chiye**

**Laboratory research #4 ”Loop parallelization using OpenMP technology”**

目录

[1 Brief decription of the task to be completed 3](#_Toc38918854)

[2 Processor description 4](#_Toc38918855)

[3 Full text of the resulting parallel program and the scripts 5](#_Toc38918856)

[4 Graphs of program execution time functions 11](#_Toc38918857)

[5 Detailed conclusions with an analysis of the graphs and the results obtained. 12](#_Toc38918858)

# 1 Brief decription of the task to be completed

Based on Experiment 3, change the Generate stage so that the generated set of random numbers does not depend on the number of threads running the program. Then replace the gettimeofday function calls to omp\_get\_wtime. After that, parallelize the calculations at the Sort stage. Then write a function that outputs a message to the console about the current percentage of program shutdown once a second and run the specified function. At last, run the program with different N.

# 2 Processor description

Processor: Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz

Operating system: ubuntu18.04

GCC compiler used for experiments: gcc version 7.5.0 (Ubuntu 7.5.0-3ubuntu1~18.04)

bit depth: 64

number of cores: 4

RAM capacity: 8G

# 3 Full text of the resulting parallel program and the scripts

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <math.h>
4. #include <time.h>
5. #include <unistd.h>
6. #include <pthread.h>
8. #ifdef \_OPENMP
9. #include <omp.h>
10. #else
11. **int** omp\_get\_wtime() { **return** 1; }
12. #endif

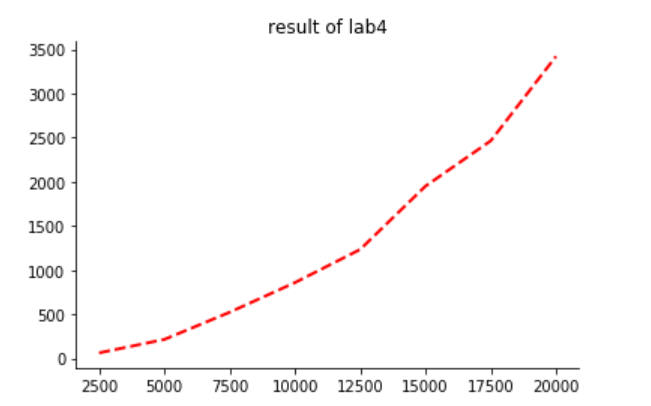

16. **int** main(**int** argc, **char** \*argv[])
17. {
19. **int** i=0, N,j;
20. **int** M;
21. //struct timeval T1, T2;
23. **long** delta\_ms;
25. N = atoi(argv[1]); /\* N绛変簬绗竴涓懡浠よ鍙傛暟N equals the first command-line parameter \*/
26. M = 3;
28. //N = 10000;
30. **double** T1 = omp\_get\_wtime(); /\* 璁颁綇褰撳墠鏃堕棿T1 remember the current time T1 \*/
31. **double** temp=T1;
32. unsigned **int** seed = 0;
33. #pragma omp parallel num\_threads(4) shared(i)
34. { /\* 50 experiments50 experiments \*/
35. **if**(omp\_get\_thread\_num()==0){
37. **while**(i<=50)
38. {
39. **if**(omp\_get\_wtime() - temp>1){
40. printf("%lf%%\n", i/50.0\*100);
41. temp = omp\_get\_wtime();
42. }
43. }
44. }**else**{
45. **while**(i<50)
46. {
47. **double** M1[N] ,M2[(**int**)(N/2)];
48. srand((unsigned)time(NULL)\*10); /\* 鍒濆鍖朢NG鐨勫垵濮嬪€?initialize the initial value of the RNG\*/
49. **int** A = 24;
50. #pragma omp parallel for private(j) shared(M1)  schedule(guided,M)
51. **for**(j = 0; j < N; j++){/\* 鐢∟涓暟鎹～鍏呭垵濮嬫暟缁?Fill the initial data array with size N \*/
52. M1[j] = rand\_r(&seed)%A;
53. **while**(M1[j]==0.0){
54. M1[j] = rand\_r(&seed)%A;
55. }
56. }
58. #pragma omp parallel for private(j) shared(M2) schedule(guided,M)
59. **for**(j = 0; j < N/2; j++){
60. M2[j] = rand\_r(&seed)%(A\*10);
61. }
63. /\* 瀹屾垚浠诲姟锛岀敤缁撴灉濉厖鏁扮粍 Complete the task, fill in the array with the results \*/
64. /\* Map Stage. Apply Hyperbolic cotangent of number鈥檚 root to each element in the M1 array \*/
65. #pragma omp parallel for private(j) shared(M1) schedule(guided,M)
66. **for**(j = 0; j < N; j++){
67. M1[j] = sqrt(1.0/tanh(M1[j]));
68. }
69. /\* in the M2 array, add each element in turn with the previous one ,and apply Sine modulus to the result of the addition\*/
70. **double** M2\_temp[(**int**)(N/2)+1];
71. M2\_temp[0] = 0;
73. #pragma omp parallel for private(j) shared(M2\_temp,M2)  schedule(guided,M)
74. **for**(j = 1; j < N/2+1; j++){
75. M2\_temp[j] = M2[j-1];
76. }
78. #pragma omp parallel for private(j) shared(M2) schedule(guided,M)
79. **for**(j = 0; j < N/2; j++){
80. M2[j] = fabs(sin(M2\_temp[j]+M2\_temp[j+1]));
82. }
84. /\* Merge Stage. In arrays M1 and M2 apply Raising to a power to all elements in pairs with the same indexes in \*/
85. #pragma omp parallel for private(j) shared(M2,M1) schedule(guided,M)
86. **for**(j = 0; j < N/2; j++){
87. //#pragma omp atomic
88. M2[j] = pow(M1[j],M2[j]);
89. }

92. **double** left[N/4],right[N/4];
93. **int** l=0, r=0;
94. #pragma omp parallel sections
95. {
96. #pragma omp section
97. {
98. **int** x=0;
99. **while** (x<N/4-1)
100. {
101. **if** (x == 0 || M2[x - 1] <= M2[x])
102. x++;
103. **else**
104. {
105. **double** tmp = M2[x];
106. M2[x] = M2[x - 1];
107. M2[--x] = tmp;
108. }
109. }
110. }
111. #pragma omp section
112. {
113. **int** x=N/4;
114. **while** (x<N/2)
115. {
116. **if** (x == 0 || M2[x - 1] <= M2[x])
117. x++;
118. **else**
119. {
120. **double** tmp = M2[x];
121. M2[x] = M2[x - 1];
122. M2[--x] = tmp;
123. }
124. }
125. }
127. }
128. **for**(j=0;j<N/4;j++){
129. left[j]=M2[j];
130. right[j]=M2[j+N/4];
131. }
132. j=0;
133. **while**(l<N/4&&r<N/4){
134. **if**(left[l]<right[r]){
135. M2[j]=left[l];
136. j++;
137. l++;
138. }**else**{
139. M2[j]=right[r];
140. j++;
141. r++;
142. }
143. }
144. **while**(l<N/4){
145. M2[j]=left[l];
146. j++;
147. l++;
148. }
149. **while**(r<N/4){
150. M2[j]=right[r];
151. j++;
152. r++;
153. }
155. /\* Reduce Stage. determining parity \*/
156. **double** min = A\*10;
158. #pragma omp parallel for private(j) shared(M2,min) schedule(guided,M)
159. **for**(j = 0; j < N/2; j++){/\* Find the minimum \*/
160. **if**(min > M2[j] && M2[j]!=0)
162. #pragma omp critical
163. min = M2[j];
164. //printf(" %f ",M2[j]);
165. }
166. //printf("min: %f\n",min);
167. **double** sum = 0;
168. #pragma omp parallel for private(j) reduction(+:sum)  shared(M2) schedule(guided,M)
169. **for**(j = 0; j < N/2; j++){
170. **int** tmp = (**int**)(M2[j]/min);
171. **if**(tmp%2==0.0)
172. sum = sum + sin(M2[j]);
173. }
174. i=i+1;
175. }
176. //printf("X: %f\n",sum);
177. }
178. }

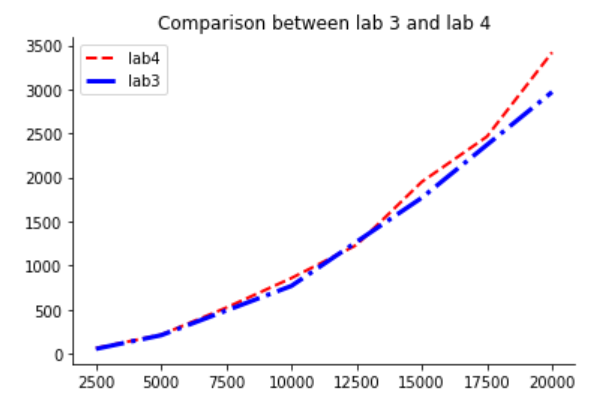
181. **double** T2 = omp\_get\_wtime(); /\* remember the current time T2 \*/

184. /\* remember the current time T2 \*/
186. //delta\_ms = 1000 \* (T2.tv\_sec - T1.tv\_sec) + (T2.tv\_usec - T1.tv\_usec) / 1000;
187. delta\_ms = 1000 \* (T2-T1);
188. printf("\nN=%d. Milliseconds passed: %ld\n", N, delta\_ms); /\* T2 -T1 \*/
189. **return** 0;
190. }
191. #!/bin/bash
193. **for** i in 2500 5000 7500 10000 12500 15000 17500 20000
194. **do**
195. ./lab4 ${i}
196. done
197. s

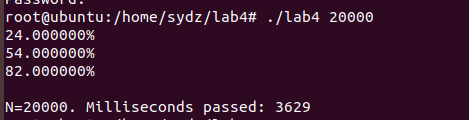
# 4 Graphs of program execution time functions



**Figure 1 result of lab4**



**Figure 2 comparison between lab3 and lab4**



**Figure 3 current percentage of program shutdown**

# 5 Detailed conclusions with an analysis of the graphs and the results obtained.

From the comparison between the result of lab3 and lab4, we can infer that although we divided the sorting stage into two parts. We do the two parts in the same time, then merge the answer array. But the efficiency of program did not improve at all. On the contrary, it even takes a little more time to do the job. The reason may be that the merge stage also takes a lot of time. And the merge can’t be executed parallelly.

From figure 3, in the first second the program does the least job of the loop. But later the completion per second is almost same. The reason can be that initially the creating of threads takes a lot time.