ASSIGNMENT 1- SEQUENTIAL SEARCHING AND PARALLEL SEARCHING USING OPENMP REPORT

CSC4005 High Performance Computing

Abstract

I doing some coding and some tests to solve the problem.

1.introduction

OpenMP API have been developed to support parallel application. They provide the parallel application programmer with a model for parallel programming that is portable across architecture from different ventors , and it has been widely accepted as a set of Compiler Directives for multiprocessor programming of shared memory parallel systems . OpenMP supported programming languages include C, C++, and Fortran; and OpenMp-enabled compilers include Sun Compiler, GNU Compiler, and Intel Compiler. OpenMp provides a high-level abstract description of parallel algorithms. The programmer specifies the intent by adding a dedicated pragma to the source code, so that the compiler can automatically parallelize the program and add synchronization mutex where necessary. And communication. When you choose to ignore these pragmas, or if the compiler does not support OpenMp, the program can degenerate into the usual program (usually serial), the code will still work, but you can't use multithreading to speed up program execution.

In this assignment, I focus on the basic use of openMP, and discuss the performance about using openMP to a straightforward pattern matching algorithm with diffident threads. In detail, understand how straightforward pattern matching algorithm works. Generate the test cases, and coding the searching sequential.c into an OpenMP program,also need find the bug and do some optimizations on code to improve the performance.

2.1 Part A - Sequential Searching

<u>About straightforward pattern matching algorithm:</u>

The name of the straightforward pattern matching algorithm is very cool, but the efficiency is not very good, which is a simple, rude way to search.

About the Worst case of straightforward pattern matching algorithm: the time complexity of straightforward pattern matching is O(mn) ,which also have not a preprocessing stage.So that In the worst case, the straightforward pattern matching algorithm looks for a length of M in a text of length N and requires ~NM character comparisons, until it found the pattern on text at last M.

Therefore, I figured out a way to solve this, for example if the pattern is [aaab] , then the text is [aaaaaaaaaaaaab], the algorithm will do full pattern length 4 times searches from each letter, and totally do (14-4)*4 times searches until the pattern found at the end.

<u>About the code:</u>

Time function:

In some testcases, elapsed wall clock time is “0.000000” because the value returned from the time() function which in the original code couldn’t have a high accurate to the 0.000000 . After tried a few different timing function and finally a high precision time function - gettimeofday() has been choose to be used on this code because the time it returns accurate to microseconds which meet the requirement perfectly. What’s more it’s also in the time.h, so that we can introduce it easily without add new head files.

Snippet of code:

[Length of Pattern and text choose] part:'

Totally 20 patterns and texts which will result in the worst case performance and note the CPU execution time for different length(text)*length(pattern).
For the selection of pattern and text length. In the first four testcases I design the pattern length is 2 ,4, 5 ,10(the pattern is "ab","aaab","aaaab","aaaaaaaaab") and the text length is simply the 100/[pattern Length], as there is a rule that the product length(text)*length(pattern) should be approximately equal to 10^2, 10^4, 10^6, 10^8 and 10^10. So, each 4 test cases is a stage. Pattern Length will multiply by 100 in next stage, length is always following the formula: [text length] = product / [pattern Length].
There is another benefit of the design is that the totally file size is very small, less than 100 MB whatever.

And the Snippet of code as below:

```
long textLength;
long patternLength;// 2 4 5 10

if(testNumber>=0 && testNumber<4){
        if(testNumber%4==0)//0
                patternLength = 2;
        if(testNumber%4==1)//1
                patternLength = 4;
        if(testNumber%4==2)//2
                patternLength = 5;
        if(testNumber%4==3)//3
                patternLength = 10;
        textLength = 100/patternLength;

}
if(testNumber>=4 && testNumber<8){
        if(testNumber%4==0)//4
                patternLength = 20;
        if(testNumber%4==1)//5
                patternLength = 40;
        if(testNumber%4==2)//6
                patternLength = 50;
        if(testNumber%4==3)//7
                patternLength = 100;
        textLength = 10000/patternLength;

}
if(testNumber>=8 && testNumber<12){
        if(testNumber%4==0)//8
                patternLength = 200;
        if(testNumber%4==1)//9
                patternLength = 400;
        if(testNumber%4==2)//10
                patternLength = 500;
        if(testNumber%4==3)//11
                patternLength = 1000;
        textLength = 1000000/patternLength;
}
```

Folder creation part, generate text and pattern, and write them to each document:

About the text and pattern generation, I used 2 for loop to create them respective:

Put ‘a’ to index (0~arrayLength-2), then put b to index arrayLength-1

And use open() function and write() function to create text file and write the array of text and pattern to the corresponding file.

And the Snippet of code as below:

```
char textName[1000];
sprintf(textName,"inputs/test%d/text.txt",testNumber);
int handle_text;
char *text = (char*)malloc(textLength * sizeof(char));
handle_text = open(textName,O_CREAT|O_RDWR,0777);
if(handle_text ==-1){
        printf("errno=%d\n",errno);
        printf("couldn't create text for test%d\n",testNumber);
}
int ch = 'a';
int i;
for(i=0;i<textLength-1;i++)
        text[i] = ch;
text[textLength-1] = 'b';
write(handle_text,text,textLength);
free(text);


char patternName[1000];
sprintf(patternName,"inputs/test%d/pattern.txt",testNumber);
int handle_pattern;
char pattern[patternLength];
handle_pattern = open(patternName,O_CREAT|O_RDWR,0777);
if(handle_pattern ==-1){
        printf("errno=%d\n",errno);
        printf("couldn't create pattern for test%d\n",testNumber);
}

pattern[patternLength-1] = 'b';
for(i=patternLength-2;i>=0;i--)
        pattern[i] = 'a';
write(handle_pattern,pattern,patternLength);
if(handle_text!=-1 && handle_pattern!=-1)
        printf("create text and pattern for test%d successfully\n",testNumber);
```

Results(output):

Read test number 0

Text length = 50

Pattern length = 2

Pattern found at position 48

comparisons = 98

Test 0 elapsed wall clock time = 0.000005

Test 0 elapsed CPU time = 0.000000

Read test number 1

Text length = 25

Pattern length = 4

Pattern found at position 21

comparisons = 88

Test 1 elapsed wall clock time = 0.000005

Test 1 elapsed CPU time = 0.000000

Read test number 2

Text length = 20

Pattern length = 5

Pattern found at position 15

comparisons = 80

Test 2 elapsed wall clock time = 0.000004

Test 2 elapsed CPU time = 0.000000

Read test number 3

Text length = 10

Pattern length = 10

Pattern found at position 0

comparisons = 10

Test 3 elapsed wall clock time = 0.000004

Test 3 elapsed CPU time = 0.000000

Read test number 4

Text length = 500

Pattern length = 20

Pattern found at position 480

comparisons = 9620

Test 4 elapsed wall clock time = 0.000033

Test 4 elapsed CPU time = 0.000000

Read test number 5

Text length = 250

Pattern length = 40

Pattern found at position 210

comparisons = 8440

Test 5 elapsed wall clock time = 0.000029

Test 5 elapsed CPU time = 0.000000

Read test number 6

Text length = 200

Pattern length = 50

Pattern found at position 150

comparisons = 7550

Test 6 elapsed wall clock time = 0.000026

Test 6 elapsed CPU time = 0.000000

Read test number 7

Text length = 100

Pattern length = 100

Pattern found at position 0

comparisons = 100

Test 7 elapsed wall clock time = 0.000005

Test 7 elapsed CPU time = 0.000000

Read test number 8

Text length = 5000

Pattern length = 200

Pattern found at position 4800

comparisons = 960200

Test 8 elapsed wall clock time = 0.002755

Test 8 elapsed CPU time = 0.000000

Read test number 9

Text length = 2500

Pattern length = 400

Pattern found at position 2100

comparisons = 840400

Test 9 elapsed wall clock time = 0.002289

Test 9 elapsed CPU time = 0.000000

Read test number 10

Text length = 2000

Pattern length = 500

Pattern found at position 1500

comparisons = 750500

Test 10 elapsed wall clock time = 0.002062

Test 10 elapsed CPU time = 0.000000

Read test number 11

Text length = 1000

Pattern length = 1000

Pattern found at position 0

comparisons = 1000

Test 11 elapsed wall clock time = 0.000008

Test 11 elapsed CPU time = 0.000000

Read test number 12

Text length = 50000

Pattern length = 2000

Pattern found at position 48000

comparisons = 96002000

Test 12 elapsed wall clock time = 0.261211

Test 12 elapsed CPU time = 0.260000

Read test number 13

Text length = 25000

Pattern length = 4000

Pattern found at position 21000

comparisons = 84004000

Test 13 elapsed wall clock time = 0.228414

Test 13 elapsed CPU time = 0.230000

Read test number 14

Text length = 20000

Pattern length = 5000

Pattern found at position 15000

comparisons = 75005000

Test 14 elapsed wall clock time = 0.210776

Test 14 elapsed CPU time = 0.200000

Read test number 15

Text length = 10000

Pattern length = 10000

Pattern found at position 0

comparisons = 10000

Test 15 elapsed wall clock time = 0.000032

Test 15 elapsed CPU time = 0.000000

Read test number 16

Text length = 500000

Pattern length = 20000

Pattern found at position 480000

comparisons = 9600020000

Test 16 elapsed wall clock time = 25.954374

Test 16 elapsed CPU time = 25.900000

Read test number 17

Text length = 250000

Pattern length = 40000

Pattern found at position 210000

comparisons = 8400040000

Test 17 elapsed wall clock time = 22.770283

Test 17 elapsed CPU time = 22.709999

Read test number 18

Text length = 200000

Pattern length = 50000

Pattern found at position 150000

comparisons = 7500050000

Test 18 elapsed wall clock time = 20.294216

Test 18 elapsed CPU time = 20.190001

Read test number 19

Text length = 100000

Pattern length = 100000

Pattern found at position 0

comparisons = 100000

Test 19 elapsed wall clock time = 0.000276

Test 19 elapsed CPU time = 0.000000

Points selection:

Results(graphs):

Part B:

Test0:

1.the data below will be used as a reference point(sequential)

[40120405@login1(kelvin) assignment-1b]$ $time ./searching_sequential

Read test number 0

Text length = 2000000

Pattern length = 2000

Pattern found at position 1998000

comparisons = 3996002000

Test 0 elapsed wall clock time = 11

Test 0 elapsed CPU time = 10.680000

Read test number 1

Text length = 10000000

Pattern length = 1000

Pattern found at position 1001

comparisons = 1002000

Test 1 elapsed wall clock time = 0

Test 1 elapsed CPU time = 0.000000

Read test number 2

Text length = 10000000

Pattern length = 1000

Pattern found at position 5001001

comparisons = 5001002000

Test 2 elapsed wall clock time = 14

Test 2 elapsed CPU time = 13.380000

3.result：

When using openMP instead of sequential:

3a. Table of elapsed times and numbers of comparisons for 3 diffident scheduling strategies:

Static:

threads	elapsed wall clock time	# comparisons
1	15	3996002000
2	11	2108807405
4	11	1084215980
8	11	855766177
16	12	603062187
32	11	483029111
64	11	410961034

Dynamic:

threads	elapsed wall clock time	# comparisons
1	14	3996002000
2	21	675219377
4	14	252413741
8	15	300126896
16	16	332466081
32	13	252801616
64	14	178058564

Guided:

threads	elapsed wall clock time	# comparisons
1	15	3996002000
2	12	2130151583
4	11	1034482481
8	12	759857499
16	13	487172607
32	13	330788813
64	12	218670030

3b.

Static:

parallel speedup(1) = 15/15=1

parallel speedup(2) = 15/11=1.364

parallel speedup(4) = 15/11=1.364

parallel speedup(8) = 15/11=1.364

parallel speedup(16) = 15/12=1.25

parallel speedup(32) = 15/11=1.364

parallel speedup(64) = 15/11=1.364

dynamic:

parallel speedup(1) = 14/14=1

parallel speedup(2) = 14/21=0.667

parallel speedup(4) = 14/14=1

parallel speedup(8)=14/15=0.933

parallel speedup(16)=14/16=0.875

parallel speedup(32)=14/13=1.077

parallel speedup(64)=14/14=1

guided:

parallel speedup(1) = 15/15=1

parallel speedup(2) = 15/12=1.25

parallel speedup(4) = 15/11=1.364

parallel speedup(8)=15/12=1.25

parallel speedup(16)=15/13=1.154

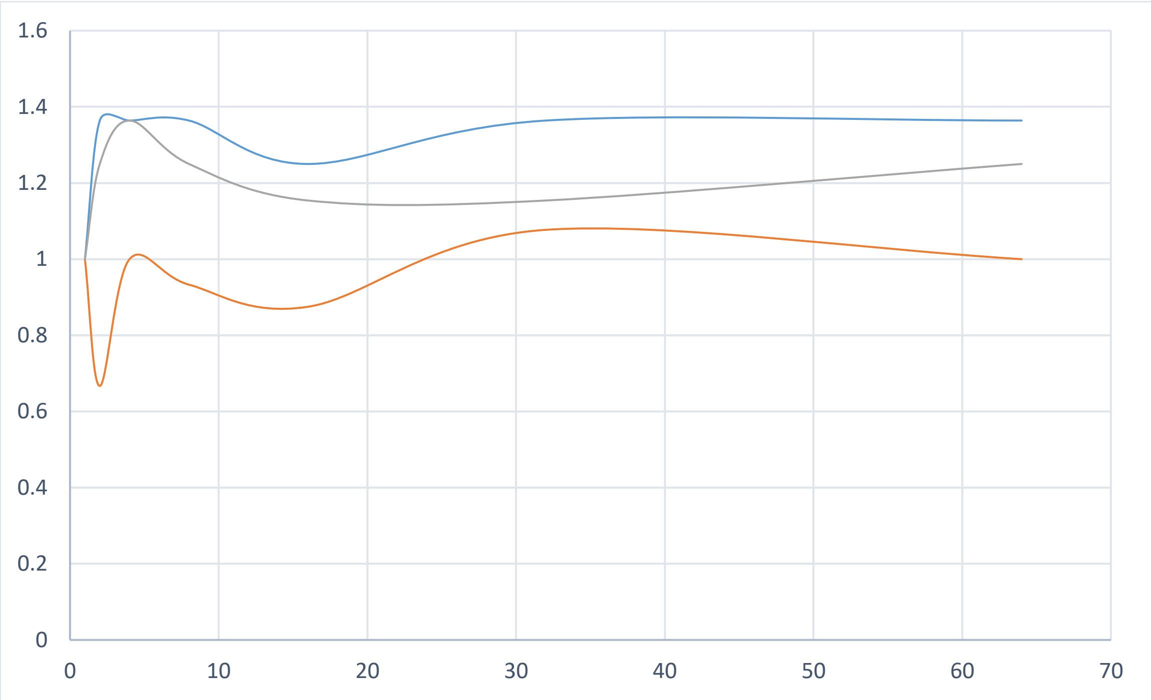
parallel speedup(32)=15/13=1.154

parallel speedup(64)=15/12=1.25

graph of the parallel speedup vs the number of threads:

x: the number of threads

y: the parallel speedup(blue:static,red:dynamic,gray:guided)



3c.

Static:

parallel efficiency (1) = 1/1=1

parallel efficiency (2) = 1.364/2=0.682

parallel efficiency (4) = 1.364/4=0.341

parallel efficiency (8) = 1.364/8=0.175

parallel efficiency (16) = 1.25/16=0.078

parallel efficiency (32) = 1.364/32=0.043

parallel efficiency (64) = 1.364/64=0.021

dynamic:

parallel efficiency (1) = 1/1=1

parallel efficiency (2) = 0.667/2 = 0.334

parallel efficiency (4) = 1/4=0.25

parallel efficiency (8) = 0.933/8=0.117

parallel efficiency (16) = 0.875/16=0.055

parallel efficiency (32) = 1.077/32=0.034

parallel efficiency (64) = 1/64=0.016

guided:

parallel efficiency (1) = 1/1=1

parallel efficiency (2) = 1.25/2=0.625

parallel efficiency (4) = 1.364/4=0.341

parallel efficiency (8) = 1.25/8=0.156

parallel efficiency (16) = 1.154/16=0.072

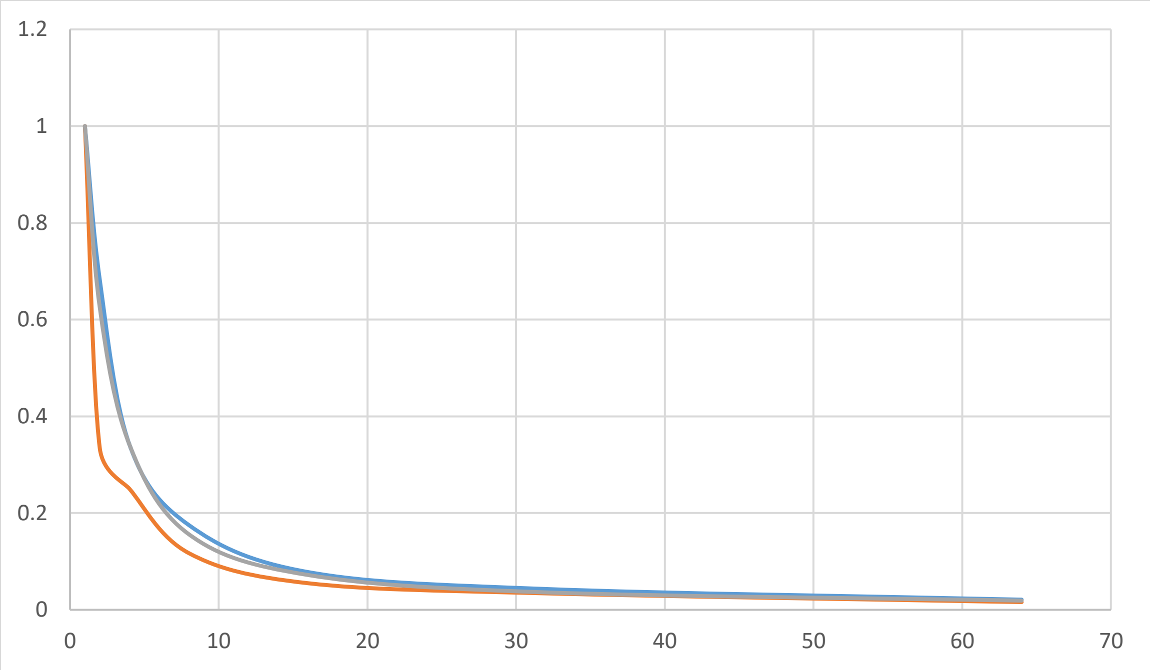
parallel efficiency (32) = 1.154/32=0.036

parallel efficiency (64) = 1.25/64=0.019

graph of the parallel efficiency vs the number of threads:

x: number of threads:

y: parallel efficiency (blue: static,red:dynamic,gray:guided)



Conclusion: According to the 2 graphs of PS vs P and PE vs P , there is no too much diffidence regarding to 3 scheduling strategies, and the performance of static schedule is slightly better than other 2's. However, while the threads increase, the parallel efficiency is decline, the relationship of them is nearly y = 1/x.