用 OpenMP 并行实现 CYK 算法

CYK 算法（Cocke–Younger–Kasami algorithm，缩写为 CYK）是由约翰·科克，Younger 和嵩忠雄共同研究出来，大约发表于 1965 年的一个用来判定任意给定的字符串 w 是否属于一个上下文无关文法的编译算法。

普通的回溯法在最坏的情况下需要指数时间才能解决这样的问题，而 CYK 算法只需要多项式时间就够了(O(n^3)，n 为字符串 w 的长度）。CYK 算法采用了动态规划的思想。伪代码描述如下：

```
function CKY-PARSE(words, grammar) returns table

  for j ← from 1 to LENGTH(words) do
      table[j − 1, j] ← {A | A → words[j] ∈ grammar }
      for i ← from j − 2 downto 0 do
          for k ← i + 1 to j − 1 do
              table[i,j] ← table[i,j] ∪
                           {A | A → BC ∈ grammar,
                                B ∈ table[i,k],
                                C ∈ table[k, j] }
```

本作业提供了串行代码与数据样例，请同学们在串行代码的基础上完成代码并行化。串行代码仅供参考，也可以自己编写效率更高的串行代码。

不要尝试修改算法，因为编译原理课上讲的算法不能处理二义性文法，测试用例给定的文法都是有二义性的上下文无关文法。

也不可对文法进行化简，这会导致结果不正确。

【输入形式】

从“input.txt”读入。

第一行是一个整型，表示非终结符的数量 vn_num；

第二行是一个整型，表示“右侧是两个非终结符的产生式”的数量 production2_num，即代码中的 Production2 类型的数量；

接下来 production2_num 行，每行是一个 Production2 类型，非终结符的格式是“<%d>”，数字范围是 0 到(production2_num-1)，“<0>”表示开始符号；

然后下一行是一个整型，表示“右侧是一个终结符的产生式”的数量 production1_num，即代码中的 Production1 类型的数量；

接下来 production1_num 行，每行是一个 Production1 类型，终结符的格式是“%c”，范围是所有可打印的 ASCII 字符；

然后下一行是一个整型，表示被测试的字符串的长度；

最后一行是被测试的字符串。

【输出形式】

一个 32 位无符号整型，表示被测试的字符串能构成多少棵符合文法的语法树，语法树的数量可能很多。为了简单起见，若发生溢出就截取低 32 位。

【样例输入】

```
4
5
<0>::=<1><2>
<0>::=<2><3>
<1>::=<2><1>
<2>::=<3><3>
<3>::=<1><2>
3
<1>::=a
<2>::=b
<3>::=a
5
baaba
```

【样例输出】

```
2
```

【样例说明】

输入样例中的文法和字符串可以构成两棵语法树，如下页图所示。

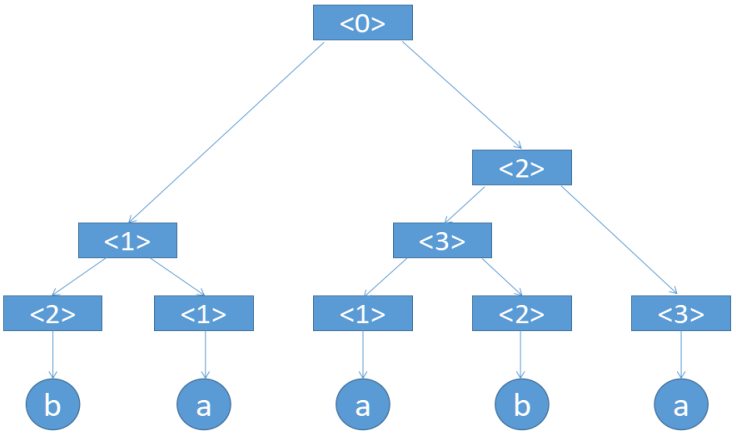
【要求】

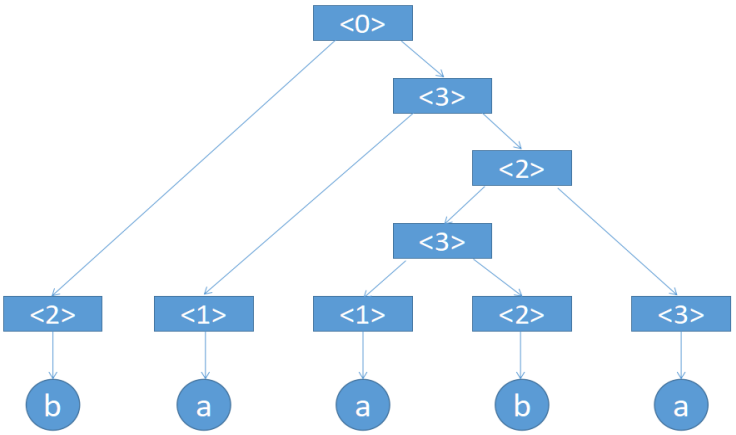
禁止抄袭（包括抄袭往届作业）。

可以使用学校的高性能计算平台的硬件资源进行实验。

完成作业时间: 4.18-4.25

同时希望大家完成一份实验报告，实验报告内容包括但不限于：



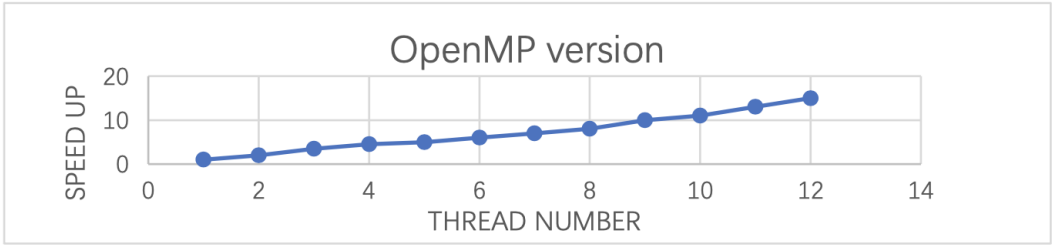


① 如何将任务进行分块的？

② 如何提升运行速度与增强程序扩展性？

③ 如何测量程序运行时间？

④ 请用 Excel 或者其他类似工具作出比较图，比较不同的因素所带来的加速比变化如进程的数量，线程的数量等。写清楚每张图的纵轴横轴是什么，使用了什么工具（如 MPI、OpenMP 还是 Pthreads）。如下图给了一张 OpenMP 版本的不同线程数的加速比参考样例，选择自己认为比较直观的图表形式即可。



⑤ 分析加速比随自变量变化的趋势，及发生这种变化的原因。

⑥ 最佳的优化方案是什么，该方案的优势是什么？

串行代码

serial_CYK.cpp

输入样例

input1.txt

input2.txt

input3.txt

【参考答案】

```
#include <iostream>
#include <algorithm>
#include <omp.h>
#include <time.h>
#include <memory.h>

using namespace std;

#define MAX_PRODUCTION2_NUM 512
#define MAX_PRODUCTION1_NUM 128
#define MAX_VN_NUM 128
#define MAX_VT_NUM 128
#define MAX_STRING_LENGTH 1024

struct BeginAndNum
{
    int begin;
    unsigned num;
};

struct Production2
{
    int parent;
    int child1;
    int child2;
} production2[MAX_PRODUCTION2_NUM];

struct Production1
{
    int parent;
    char child;
} production1[MAX_PRODUCTION1_NUM];

BeginAndNum vnIndex[MAX_VN_NUM][MAX_VN_NUM];
BeginAndNum vtIndex[MAX_VT_NUM];

char str[MAX_STRING_LENGTH];

struct SubTree
{
    int root;
    unsigned num;
} subTreeTable[MAX_STRING_LENGTH][MAX_STRING_LENGTH][MAX_VN_NUM];

int subTreeNumTable[MAX_STRING_LENGTH][MAX_STRING_LENGTH];

int vn_num;
int production2_num;
int production1_num;
int string_length;

int main()
{
```

```
    freopen("input.txt", "r", stdin);
    scanf("%d\n", &vn_num);
    scanf("%d\n", &production2_num);
    for (int i = 0; i < production2_num; i++)
        scanf("<%d>::=<%d><%d>\n", &production2[i].parent, &production2[i].child1,
&production2[i].child2);
    scanf("%d\n", &production1_num);
    for (int i = 0; i < production1_num; i++)
        scanf("<%d>::=%c\n", &production1[i].parent, &production1[i].child);
    scanf("%d\n", &string_length);
    scanf("%s\n", str);

    sort(production1, production1 + production1_num, [](const Production1& a, const
Production1& b)
    {
        return a.child == b.child ? a.parent < b.parent : a.child < b.child;
    });
    for (int i = 0; i < MAX_VT_NUM; i++)
    {
        vtIndex[i].begin = -1;
        vtIndex[i].num = 0;
    }
    for (int i = 0; i < production1_num; i++)
    {
        int t = production1[i].child;
        if (vtIndex[t].begin == -1)
            vtIndex[t].begin = i;
        vtIndex[t].num++;
    }
    for (int i = 0; i < string_length; i++)
    {
        int t = str[i];
        int begin = vtIndex[t].begin;
        int end = begin + vtIndex[t].num;
        for (int j = begin; j < end; j++)
        {
            SubTree subTree;
            subTree.root = production1[j].parent;
            subTree.num = 1;
            subTreeTable[i][i][subTreeNumTable[i][i]++] = subTree;
        }
    }

    sort(production2, production2 + production2_num, [](const Production2& a, const
Production2& b)
    {
        return a.child1 == b.child1 ?
            (a.child2 == b.child2 ? a.parent < b.parent : a.child2 < b.child2)
            : a.child1 < b.child1;
    });
    for (int i = 0; i < vn_num; i++)
    {
        for (int j = 0; j < vn_num; j++)
        {
            vnIndex[i][j].begin = -1;
            vnIndex[i][j].num = 0;
```

```
        }
    }
    for (int i = 0; i < production2_num; i++)
    {
        int n1 = production2[i].child1;
        int n2 = production2[i].child2;
        if (vnIndex[n1][n2].begin == -1)
            vnIndex[n1][n2].begin = i;
        vnIndex[n1][n2].num++;
    }
    for (int len = 2; len <= string_length; len++)
    {
        for (int left = 0; left <= string_length - len; left++)
        {
            SubTree subTreeBuf[2][MAX_STRING_LENGTH];
            //memset(subTreeBuf, 0, sizeof subTreeBuf);
            int curr = 0;
            int last = 1;
            int oldTreeNum = 0;
            for (int right = left + 1; right < left + len; right++)
            {
                //printf("[%d, %d] = [%d, %d] + [%d, %d]\n", left, left + len, left,
right, right, left + len);
                for (int i1 = 0; i1 < subTreeNumTable[left][right - 1]; i1++)
                {
                    SubTree subTreeChild1 = subTreeTable[left][right - 1][i1];
                    for (int i2 = 0; i2 < subTreeNumTable[right][left + len - 1];
i2++)
                    {
                        SubTree subTreeChild2 = subTreeTable[right][left + len -
1][i2];
                        int begin =
vnIndex[subTreeChild1.root][subTreeChild2.root].begin;
                        int end = begin +
vnIndex[subTreeChild1.root][subTreeChild2.root].num;
                        if (begin == end)
                        {
                            continue;
                        }
                        swap(last, curr);
                        int newTreeNum = 0;
                        int k = 0;
                        for (int j = begin; j < end; j++)
                        {
                            SubTree subTreeParent;
                            subTreeParent.root = production2[j].parent;
                            subTreeParent.num = subTreeChild1.num * subTreeChild2.num;
                            while (k < oldTreeNum && subTreeParent.root >
subTreeBuf[last][k].root)
                                subTreeBuf[curr][newTreeNum++] =
subTreeBuf[last][k++];
                            if (k < oldTreeNum && subTreeParent.root ==
subTreeBuf[last][k].root)
                                subTreeParent.num += subTreeBuf[last][k++].num;
                            subTreeBuf[curr][newTreeNum++] = subTreeParent;
                        }
```

```
                    while (k < oldTreeNum)
                    {
                        subTreeBuf[curr][newTreeNum++] = subTreeBuf[last][k++];
                    }
                    oldTreeNum = newTreeNum;
                }
            }
        }
        subTreeNumTable[left][left + len - 1] = oldTreeNum;
        if (subTreeNumTable[left][left + len - 1] > 0)
        {
            memcpy(subTreeTable[left][left + len - 1], subTreeBuf[curr],
subTreeNumTable[left][left + len - 1] * sizeof(SubTree));
        }
    }
}
unsigned treeNum = 0;
if (subTreeNumTable[0][string_length - 1] > 0)
{
    if (subTreeTable[0][string_length - 1][0].root == 0)
    {
        treeNum = subTreeTable[0][string_length - 1][0].num;
    }
}
printf("%u\n", treeNum);
return 0;
}
```