과제 5: 성능 측정

시스템 프로그래밍 7분반
소프트웨어학과 32170578 김산

1. 구현 설명

괄호 쌍의 개수 n이 주어질 때, n개의 괄호 쌍으로 만들 수 있는 모든 가능한 괄호 문자열 벡터를 반환하는 함수를 작성하는 프로그램을 여러 방식으로 구현하였습니다.

<코드>

gen_paren.cpp

```
vector <string> generateParenthesis(int n) {
    if (!n) return {""};
    vector <string> parens;
    for (int i =0; i < n; i ++) {
        for (string inner : generateParenthesis(i)) {
            for (string outer : generateParenthesis(n -1 - i)) {
                parens.push_back('('+ inner +')'+ outer);
            }
        }
    }
    return parens;
}
```

gen_paren_List.cpp

- vector 컨테이너에서 list컨테이너로 수정하였습니다.

```
list<string> generateParenthesis(int n) {
    if (!n) return {""};
    list<string> parens;
    for (int i =0; i < n; i ++) {
        for (string inner : generateParenthesis(i)) {
            for (string outer : generateParenthesis(n -1 - i)) {
                parens.push_back('('+ inner +')'+ outer);
            }
        }
    }
    return parens;
}
```

gen_paren_CM.cpp

- 함수호출을 줄이기 위해 for문안에 있는 generateParenthesis(i)를 밖으로 이동하였습니다.

gen_paren_CM.cpp

```
vector <string > generateParenthesis(int n) {
    if (!n) return {""};
    vector <string > parens;
    for (int i =0; i < n; i ++) {
        vector <string> inner = generateParenthesis(i);
        vector <string> outer = generateParenthesis(n -1 - i);
        for (int j =0; j < inner.size(); ++j) {
            for (int k =0; k < outer.size(); ++k) {
                parens.push_back("("+ inner[j] +")"+ outer[k]);
            }
        }
    }
    return parens;
}
```

gen_paren_CM_thread.cpp

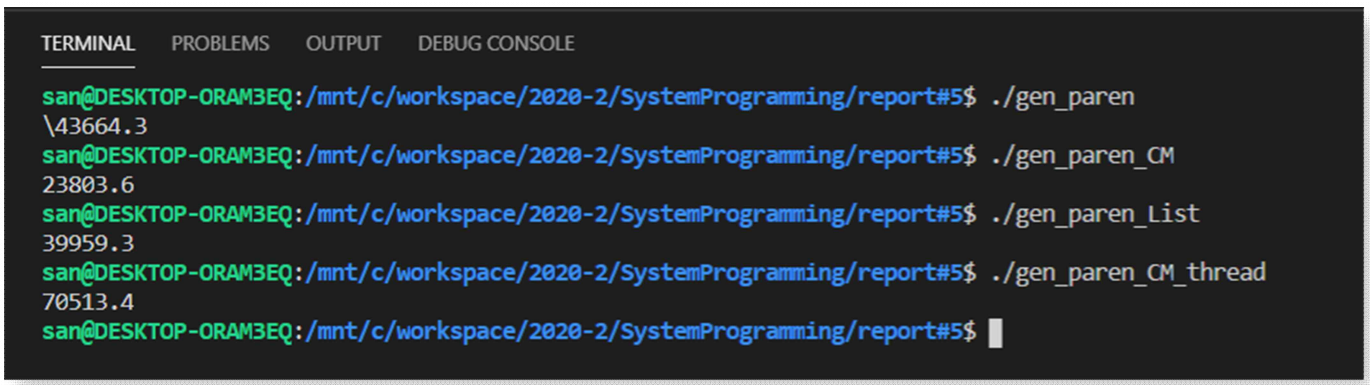
- gen_paren_CM.cpp의 for문을 openMP API로 병렬화하였습니다.

```
vector <string > generateParenthesis(int n) {
    if (!n) return {""};
    vector <string > parens, inner, outer;
    int i =0;
#pragma omp parallel
    {
    #pragma omp single private(i)
    for (int i =0; i < n; i ++) {
        #pragma omp task shared(inner)
        inner = generateParenthesis(i);
        #pragma omp task shared(outer)
        outer = generateParenthesis(n -1 - i);
        #pragma omp taskwait
        for (int j =0; j < inner.size(); ++j) {
            for (int k =0; k < outer.size(); ++k) {
                parens.push_back("("+ inner[j] +")"+ outer[k]);
            }
        }
    }
    }
    return parens;
}
```

main함수 - 다음과 같이 100번실행한 평균값을 통해 성능을 측정하였습니다.

```
int main(){
    float sum =0;
    for (int i =0; i <100; i ++)
    {
        struct timeval stime,etime,gap;
        gettimeofday(&stime,NULL);
        generateParenthesis(10);
        gettimeofday(&etime,NULL);
        gap.tv_sec = etime.tv_sec - stime.tv_sec;
        gap.tv_usec = etime.tv_usec - stime.tv_usec;
        if(gap.tv_usec <0){
            gap.tv_sec = gap.tv_sec -1;
            gap.tv_usec = gap.tv_usec +1000000;
        }
        sum += gap.tv_usec;
    }
    cout << sum /100 <<endl;
}
```

2. 수행 결과



3. 논의(discussion)

문제를 풀기 위해 처음 작성한 코드는 gen_paren.cpp입니다. 동적계획법이나 분할정복과 같은 알고리즘을 사용하여 프로그램의 성능을 높일수도 있지만 컴퓨터 시스템을 생각하며 프로그램을 최적화하는 방향으로 생각했습니다.

각 프로그램의 성능은 main 함수에서 100번 실행하고 gettimeofday()를 통해 구한 실행시간의 평균을 통해 비교하였습니다.

gen_paren_List.cpp (평균 수행시간 41226.7)

- 일반적으로 C++의 STL 컨테이너를 사용할 때 vector를 주로 사용했고, vector컨테이너가 다른컨테이너 보다 빠르다는 생각을 가지고 있었습니다. 때문에 list컨테이너보다 높은 성능을 보일것으로 기대륿 했는데 그 반대의 결과를 얻었습니다.
- 함수 내부에서 가장 많이 사용되는 push_back 함수의 경우 vector는 메모리 할당을 매번 하지 않기 때문에 매번 할당하는 list보다 pushback이 빠릅니다.
- 따라서 성능에 미치는 영향이 push_back보다는 다른곳에 원인이 있는 것 같아 push_back을 없애고 vector와 list를 재귀적으로 호출하는 다음과 같은 간단한 코드로 비교해 보았습니다.

```
list<int > list_p(int n){
    list<int > a;
    for (int i =0; i < n; i ++) {
        for (int in : list_p(i)){
            int j =0;
        }
    }
    return a;
}
```

```
vector <int > vector_p(int n){
    vector <int > a;
    for (int i =0; i < n; i ++) {
        for (int in : vector_p(i)){
            int j =0;
        }
    }
    return a;
}
```

- 1000번을 수행한 기준으로 list 컨테이너의 경우 경우 평균 수행시간이 39, vector 컨테이너의 경우 평균 수행시간이 42로 에 비해 높은 성능을 보여주었습니다.

gen_paren_CM.cpp (평균 수행시간 24049)

- 루프 안의 함수를 밖으로 가져와 함수호출횟수를 줄였습니다. 함수 호출이 줄어든 만큼 큰 성능 향상을 보여주었습니다.

gen_paren_List_thread.cpp & gen_paren_CM_thread.cpp

- 병렬처리를 하고싶었는데 재귀함수를 사용하고 있어 단순하게 코드 수정으로 병렬처리를 하는 방법을 생각해 내기가 힘들었습니다. 하둡과 맵리듀스를 보면서 병렬처리 프로그래밍 API가 있지않을까 생각해서 찾아보았는데 OpenMp라는 API가 있었습니다.
- 멀티스레드를 사용할수 있도록 코드를 구현하고 실행중에 ps -e -L을 통해 다음과 같이 프로그램이 멀티 스레드를 사용하고 있음을 확인했습니다.

- 하지만 성능이 좋아지지는 않고 비슷하거나 많이 떨어졌습니다. 구체적인 이유는 모르겠으나 추측으로는 병렬화는 가능했지만 수많은 중복계산이 발생하여 성능저하가 발생한 것 같습니다.