운영체제 과제 #4

: Thread pool 방식을 사용하여 연립방정식 프로그램 구현

과 목	운영체제 _02
전 공	컴퓨터공학과
학 번	20190850
이 름	이수진

목 차

Ⅰ. 문제 기술

1. 문제 정의 및 해결

연립방정식의 해를 구하는 프로그램을 작성한다. 연립방정식의 풀이는 Gaussian elimination과 back substitution을 이용하고, Thread pool 방식을 사용하여 multi-processor 성능을 높인다.

Thread pool 방식은 bounded buffer 개념을 적용하였는데, 이때 Main Thread와 work thread는 producer과 consumer 관계이다. bounded buffer는 원형 큐로서 semaphore로 구현하였다. Gaussian elimination과 back substitution 함수를 수행할 때 worker thread를 두 종류로 두어 각 모듈을 수행하도록 하였다.

Ⅱ. 세부 구현사항

1. 구현 시 어려웠던 코드

(1) semaphore 개수 및 초기화

임계구역에 대해 고려하지 않고 세마포 synchronize만 가지고 구현하려고 하자 동기화가 제대로 되지 않는 문제가 발생하였다. 임계구역을 고려해 3개를 추가해도 제대로 동작하지 않았다.

```
…
sem_init(&mutex, 0, 1);
sem_init(&full, 0, 0);
sem_init(&empty, 0, max-1); // max is buffer size
sem_init(&synchronize, 0, 0);
…
```

이는 empty 초기화 함수에서 문제가 발생했다. 원형 큐로 구현된 buffer는 실제로 크기-1 한 값만 사용한다. 따라서 크기 - 1로 empty를 초기화하여 해결하였다.

2. 사용한 자료구조

Gaussian 함수와 BackSub 함수에 전달할 인자로 구조체를 선언하여 전달하였다. 구조체 안에는 벡터의 크기 N, float 포인터형의 a, b, c, 그리고 thread의 개수인 threadsize 변수가 선언되어있다.

```
typedef struct __myarg_t {
    int n;
    float *a;
    float *b;
    float *c;
    int threadsize;
} myarg_t;
```

그리고 send와 recv 당시 두 개의 인자를 동시에 queue에 넣어야 하므로 이중 포인터형으로 전역 변수인 queue를 선언하였다. 그리고 반환을 위해 구조체 value도 선언했다.

```
int** queue;
...
typedef struct value
{
    int x;
    int y;
}values;
```

3. Thread pool

main에서 특정 개수(N)의 thread를 만들고, semaphore를 초기화한다. 그리고 Gaussian elimination을 수행하고 종료, Back substitution을 수행하고 종료한다.

(1) Gaussian elimination

과제 설명에 있는 알고리즘을 사용하였다. Gaussian elimination은 연립방정식 수식을 upper triangle 형태의 방정식으로 변경하는 과정이다.

Gaussian elimination 함수에서는 for-l문을 병렬화시킬 수 있다. MTGaussian에서는 작업을 수행할 개수만큼 반복해서 l과 i를 queue에 저장하고, 모든 작업이 완료되면 sentinel value인 -1을 thread의 개수만큼 저장한다,

WTGaussian 에서는 queue에서 작업을 받아서 행렬 A, 벡터 B를 계산한다. queue로부터 읽은 명령에 sentinel value가 있으면 반복에서 빠져나온다.

이때 계산 후 synchronize로 sync step을 해주는 것과 별개로, queue의 상태를 알리기 위해 mutex, empty, full을 send와 recv 할 때마다 처리한다.

```
function *MTGaussian(arg)
arg <- 전달된 구조체 포인터 변수
data <- arg의 정보를 입력받는 구조체 포인터 변수
A, b <- data가 가리키는 구조체 안의 행렬과 벡터
n <- 행렬의 가로 크기 또는 벡터의 크기
empty, mutex, full, synchronize <- 세마포
threadsize <- thread 개수

        data <- arg

        for l <-0 to n-2 do
            for i <- l+1 to n-1 do
                sem_wait(&empty)
                sem_wait(&mutex)
                send(l, i)
                sem_post(&mutex)
                sem_post(&full)
            end for
            for q <- l+1 to n-1 do
                sem_wait(&synchronize);
            end for
        end for

        for k <-0 to threadsize+1 do
            sem_wait(&empty)
            sem_wait(&mutex)
            send(-1, -1)
            sem_post(&mutex)
            sem_post(&full)
        end for

End MTGaussian()
```

```
function *WTGaussian(arg)
arg <- 전달된 구조체 포인터 변수
data <- arg의 정보를 입력받는 구조체 포인터 변수
A, b <- data가 가리키는 구조체 안의 행렬과 벡터
n <- 행렬의 가로 크기 또는 벡터의 크기
empty, mutex, full, synchronize <- 세마포
threadsize <- thread 개수

        data <- arg

        while True
            sem_wait(&full)
            sem_wait(&mutex)
            recv(&l, &i)
            sem_post(&mutex)
            sem_post(&empty)

            if ( l==-1 ) && (i == -1) then
                break
            end if
            for j <- l+1 to n-1 do
                A[i*n+j] <- A[i*n+j] - (A[i*n+l] / A[l*n+l]) * A[l*n+j]
            end for
            b[i] <- b[i] - (A[i*n+l] / A[l*n+l]) * b[l]
            sem_post(&synchronize)

        end while

End WTGaussian()
```

(2) Back substitution

과제 설명에 있는 알고리즘을 사용하였다. Back Substitution은 후진 대입법으로 맨 밑의 있는 행부터 x의 값을 구해 오는 과정이다.

Back substitution 함수에서는 for-j문을 병렬화시킬 수 있다. MTBackSub 에서는 정답 벡터인 C를 계산 후 작업을 수행할 개수만큼 반복해서 i와 j를 queue에 저장하고, 모든 작업이 완료되면 sentinel value인 -1을 thread의 개수만큼 저장한다.

WTBackSub 에서는 queue에서 작업을 받아서 행렬 A, 벡터 B를 계산한다. queue로부터 읽은 명령에 sentinel value가 있으면 반복에서 빠져나온다.

이때 계산 후 synchronize로 sync step을 해주는 것과 별개로, queue의 상태를 알리기 위해 mutex, empty, full을 send와 recv 할 때마다 처리한다.

```
function *MTBackSub(arg)
arg <- 전달된 구조체 포인터 변수
data <- arg의 정보를 입력받는 구조체 포인터 변수
A, b <- data가 가리키는 구조체 안의 행렬과 벡터
c <- data가 가리키는 구조체 안의 정답 벡터
n <- 행렬의 가로 크기 또는 벡터의 크기
empty, mutex, full, synchronize <- 세마포
threadsize <- thread 개수

        data <- arg

        for i <- n-1 to 0 do
            c[i] <- b[i] / a[i*n+i]
            for j <- 0 to i do
                sem_wait(&empty)
                sem_wait(&mutex)
                send(i, j)
                sem_post(&mutex)
                sem_post(&full)
            end for
            for q <- 0 to i do
                sem_wait(&synchronize)
            end for
        end for
        for k <-0 to threadsize+1 do
            sem_wait(&empty)
            sem_wait(&mutex)
            send(-1, -1)
            sem_post(&mutex)
            sem_post(&full)
        end for

End MTBackSub()
```

```
function *WTBackSub(arg)
arg <- 전달된 구조체 포인터 변수
data <- arg의 정보를 입력받는 구조체 포인터 변수
A, b <- data가 가리키는 구조체 안의 행렬과 벡터
c <- data가 가리키는 구조체 안의 정답 벡터
n <- 행렬의 가로 크기 또는 벡터의 크기
empty, mutex, full, synchronize <- 세마포
threadsize <- thread 개수

        data <- arg

        while True
            sem_wait(&full)
            sem_wait(&mutex)
            recv(&i, &j)
            sem_post(&mutex)
            sem_post(&empty)

            if ( i==-1 ) && ( j==-1) then
                break
            end if
            b[j] <- b[j] - c[i] * a[j*n+i]
            a[j*n+i] = 0
            sem_post(&synchronize)
        end while

End WTBackSub()
```

Ⅲ. 체크 포인트

1. 실행 결과

```
leesujin@ubuntu:~/os/hw4$ make
gcc -c -g -Wall hw4.c
gcc -o hw4 hw4.o -lpthread
leesujin@ubuntu:~/os/hw4$ ./hw4 5 3 a.dat b.dat c.dat
GaussianElimination Time : 6.695088
Backsubstitution Time : 5.202888
Total Time(Gaussian + Backsub) : 11.897976
Total Time(program) : 11.900274
leesujin@ubuntu:~/os/hw4$ ./diffCk x.dat c.dat 0.001
Two file (x.dat, c.dat) are the same.
```

[그림 1] 실행 결과 화면

2. 결과 분석

(1) p 크기의 thread pool을 사용할 때를 p개의 thread를 사용할 때와 비교해서 주의해야 할 점이 무엇이라고 보는가?

thread 개수뿐만이 아니라 buffer size까지도 고려해서 다양하게 테스트해 봐야 한다. 그리고 buffer가 원형 큐이기 때문에 그에 따른 size를 신경 쓸 필요가 있다.

(2) 과제명세서는 thread pool의 개념과 그 구조를 설명하였다. Thread pool의 구현을 C 코드 수준으로 구체적으로 설명하시오. 특히, Gaussian elimination과 back substitution을 수행할 때 work thread를 구현하는 방식을 두 방식 중 어떤 방식을 사용했고, 그 이유가 무엇인지 설명하시오.

1. 특정 개수 (N)의 thread를 만든다.
   main thread는 pthread_create()를 수행할 때
   start_routine으로 MTGaussian 또는 MTBackSub을 주고,
   work thread는 WTGaussian 혹은 STBackSub을 준다.
2. MTGaussian 혹은 MTBackSub은 작업을 수행할 개수만큼, 즉 send()한 만큼 반복해서 작업 명령을 배열 queue에 저장한다. 그리고 반환 전에 queue에 -1을 저장한다.
3. WTGaussian 혹은 WTBackSub은 queue에서 작업을 받아서 수행하는 동작을 반복한다.
4. WTGaussian 혹은 WTBackSub은 queue로부터 읽은 명령에 -1이 있으면 반복문을 빠져나온다.

이때, worker thread를 두 종류로 두어 각 모듈을 수행하도록 했는데, 이유는 과제 3을 비슷한 방식으로 구현했고, 간단했기 때문에 사용했다.

(3) 동기화를 위해 사용한 코드들을 구체적으로 설명하시오.

- Critical section(임계구역)은 어디인가? 그 이유는 무엇인가?
  send와 recv, 그리고 벡터와 행렬을 계산할 때다. send와 recv는 queue가 찼

거나 비어 있을 때를 위해서 해야 한다. 계산 시에는 work thread가 같은 곳을 중복해서 계산할 염려가 있기 때문이다.

• Critical section을 구현하기 위해 mutex_lock()을 사용할 수도 있고, sem_wait()를 사용할 수도 있다. 선택의 이유를 설명하시오.

sem_wait()를 사용했다. mutex_lock()을 사용해보니 전역 변수 count 때문에 복잡했기 때문이다.

• 과제 3에서 사용한 barrier 방식의 thread pool에서는 사용하기가 곤란하다. 이번 과제에서는 barrier 방식의 대안으로 어떤 방식을 사용했는가? 그 이유는 무엇인가?

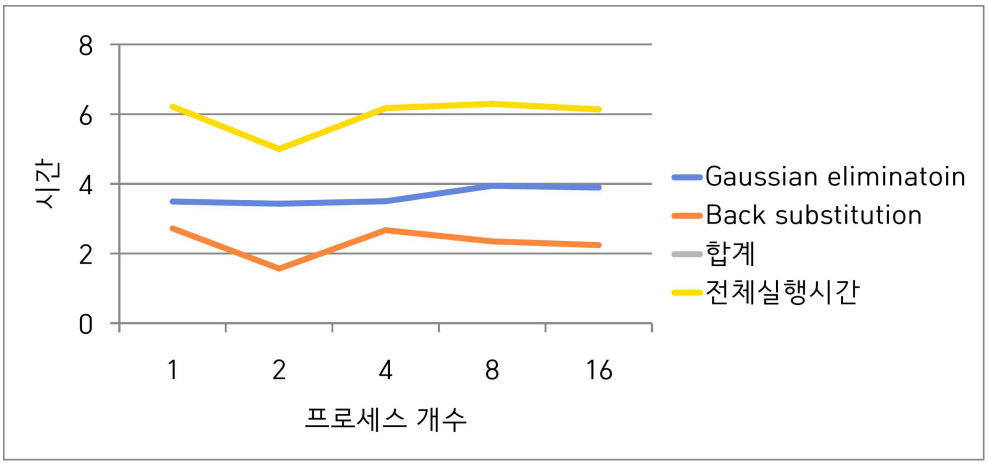
Semaphore를 사용했다. 이유는 Condition variable보다 구현이 간단해서이다.

3. 성능 평가

(1) 성능 평가 : thread pool의 크기가 1, 2, 4, 8, 16개 일 때의 실행 시간을 구해 표 1을 완성하고 전체 실행 시간의 변화를 그래프로 표시하시오. (bounded buffer의 크기는 10으로 고정하시오.)

[표 1] 실행 시간 표

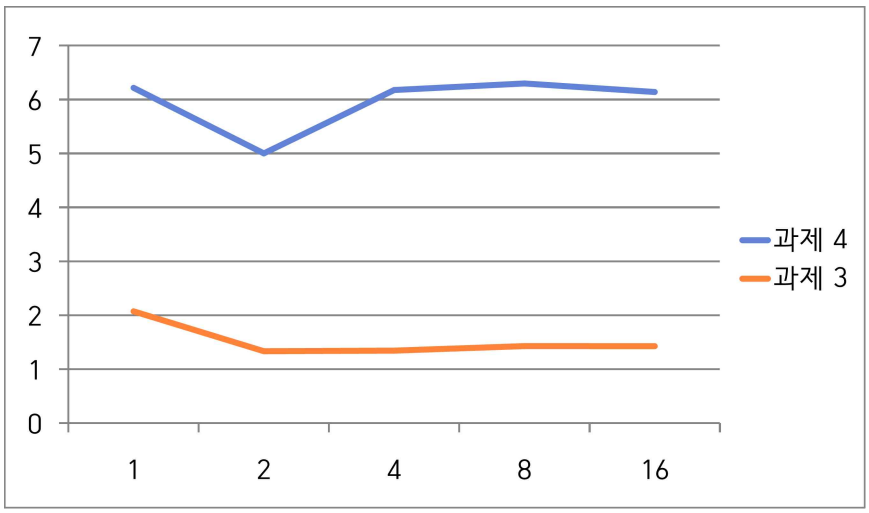
thread pool	계산 시간			전체실행시간 (I/O 포함)
	Gaussian elimination	Back substitution	합계	
1	3.493849	2.721095	6.214943	6.217491
2	3.430637	1.568470	4.999108	5.001368
4	3.502302	2.671805	6.174107	6.176428
8	3.945846	2.350125	6.295971	6.298015
16	3.895083	2.242754	6.137837	6.139638



[그림 3] 실행 시간 그래프

(2) 성능 비교 : 과제 3의 전체 실행 시간과 이번 과제의 전체 실행 시간을 thread 개수 / thread pool의 크기가 변함에 따라 비교하는 그래프를 그리시오. 그 결과가 나온 이유를 분석하시오.

과제 4가 과제 3보다 확연하게 오래 걸리는 결과를 보여주었다. 이는 임계영역이 늘어남으로 생기는 성능 저하일 수도 있고, 프로그램 코드상의 문제로 인한 것일 수 있다.



Ⅳ. 결론

thread pool 방식은 multi-process를 사용한 과제 2와 비교해 구현이 간단하나, 본 문제에서는 multi-thread를 사용한 과제 3보다는 시간이 조금 더 오래 걸린다. 이는 임계영역 처리 등으로 발생하는 문제로, 본 문제 해결은 multi-thread 방식을 사용하는 것이 성능 향상에 도움이 된다.