운영체제 2차과제

-Multiprocessing, Multithreading을 이용한 Kmeans algorhithm의 구현-

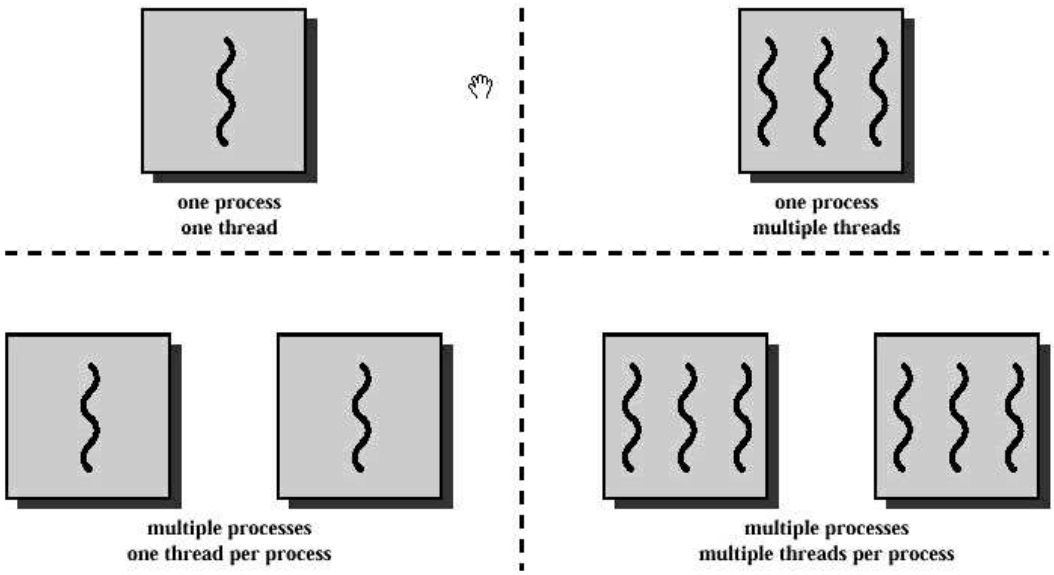
목차

2014147550
컴퓨터과학과
강효림

1. 사전 보고서

가. Process와 Thread의 차이

1) Task: 자원 소유권의 단위
2) Process: 동작 중인 프로그램을 말하며, 커널로부터 할당받은 “자신만의” 자원을 갖고, CPU가 기계어 명령들을 실행함에 따라 끊임없이 변화하는 동적인 존재이다.
3) Thread: 동작 수행의 단위를 말하며, 하나의 제어 흐름을 나타낸다. 한 프로세스 내에 여러 개의 쓰레드가 존재할 수 있다. 이 때, 같은 프로세스 내의 쓰레드들은 프로세스의 주소 공간 등 자원을 공유하며 실행된다.



나. Process와 Thread의 리눅스 상 구현

프로세스를 생성하는 fork, vfork함수와, Thread를 생성하는 pthread_create함수는 모두 최종적으로는 커널 내부 함수인 do_fork를 호출한다. 이는 리눅스가 **프로세스와 쓰레드를 모두 'task'를 생성하는 것**으로 그 본질을 같게 보고 있기 때문인데, 따라서 리눅스 상에서 둘의 차이는 새로 생성되는 task들이 서로 자원을 얼마나 공유하는가에 불과하다고 볼 것이다. 한편, do_fork함수는 task_struct 구조체를 생성하여 새로 생성되는 테스크의 성질을 정의한다.

task가 do_fork함수로 하여금 생성되면, 운영체제는 그 task에 고유한 pid를 부여해 준다. 이 때, 사용자가 프로세스를 원하는 경우라면, 그 task의 tgid에 앞서 언급한 (방금 생성된) pid값을 그대로 넣어주고, 쓰레드를 원하는 경우라면 tgid값을 부모 프로세스의 tgid값으로 설정한다. 이로서 리눅스는 thread와 process를 구분할 수 있다.

다. 멀티프로세싱, 멀티쓰레딩

앞서 살피었듯, Thread와 Process는 자원 공유의 관점에서 차이를 보인다. Process는 독자적인 자원을 갖고 실행되므로, 멀티프로세싱은 타 프로세스 간 자원 공유가 비교적 적은, 소위 '**독립적인**' 작업을 수행하는데 적합하고, Thread는 서로 자원을 공유하므로, 멀티스레딩은 서로 간 **자원의 공유가 잦은 작업**의 수행에 적합하다.

멀티스레딩은 자원 공유도 쉽고 프로세스에 비해 상대적으로 경량이나(오버헤드가 적으나), 각 스레드 간 자원 공유는 필연적으로 **동기화 문제**를 수반하므로, 언제나 멀티스레딩이 정답만은 아니라 할 것이다.

라. IPC

1) 개요

프로세스는 비록 서로 자원을 공유하지는 아니하나, 서로 간 통신을 해야 할 필요가 있을 때가 있다. 이 때 그 방법론을 통틀어 Inter-Process Communication이라고 칭하는데, 대표적으로는 signal, socket, (named) pipe, shared memory 등이 있다. 과제와 관련된 signal, (named) pipe, shared memory를 아래 자세히 검토한다

2) signal

프로세스 간, 또는 프로세스 내 다른 쓰레드끼리 '비동기적으로' 전달되는 신호로써, 그 처리를 signal handler로 하여금 하게 할 수 있

다. Linux에서는 signal 함수에 사용자 정의 함수의 함수포인터를 줌으로써 당해 signal에 대한 처리를 사용자 임의로 정할 수 있다.

3) (named) pipe

fifo 방식으로 프로세스 간 data를 주고받을 수 있으며, '단방향' 통신이 원칙이다. named pipe의 경우 pipe의 이름으로써 부모 자식 프로세스 외에도 임의의 프로세스가 당해 fifo에 접근 가능하다는 장점이 있다. data를 쓰고자 하는 프로세스는 named pipe를 O_WRONLY 옵션으로 열고, 읽고자 하는 프로세스는 그 named pipe를 O_RDONLY로 연다. 이 때, pipe의 data는 file system에 위치하는 것이 아니여서(언제나 0kb, 다만 이름은 특수 file 형태로 존재하는데, 프로세스가 pipe에 접근하기 위한 adress로만 쓰임), **open()함수는 당해 pipe의 다른 한쪽 끝이 열릴 때까지 block된다는 점**을 주의한다. 이렇게 open된 pipe의 read함수는 그 pipe에 data가 쓰이기 전까지 block됨을 이용하여 과제의 process간 동기화를 꾀하였다.

4) shared memory

프로세스 간에 공유되는 메모리로써, shmget함수로 생성 가능하다. 각 프로세스에서는 각각의 shared memory에 접근할 수 있는 key값을 shmget 함수에 줌으로 하여 shmid를 얻고, 이를 인수로 준 shmat함수를 이용하여 process에 그 shared memory를 attach하여 그 메모리를 통상의 프로세스 메모리와 같이 이용할 수 있다. **IPC중에 가장 속도가 빠르고**, 과제에서는 data parallel 처리를 이용하므로 같은 데이터에 동시접근의 문제가 발생할 소지가 적은 바, 과제에서 당해 IPC 메소드를 이용하였다.

마. 기타 고려사항 - 과제의 설계

멀티태스킹으로 연산 속도의 향상을 꾀할 부분은 각 점이 어떠한 cluster에 속하는지를 판단하는, 소위 determineCluster 연산 부분이다. 각 점의 연산은 다른 점의 연산 결과에 영향 받지 아니함으로, thread pool보다는 data parallelism의 방식이 더 적합하다 보았으므로, 총 n개 thread, 또는 process는 각각 Point 배열의 $\frac{1}{n}$ 분량만큼의 작업을 처리하도록 구현할 것이다. 다만, iteration 내에서 Thread들을 생성하는 multithreading model과는 달리, multiprocessing의 경우는 fork의 오버헤드를 고려하여, **iteration 밖에서 n개의 프로세스를 생성**한 뒤, 부모 프로세스와 named pipe의 blocking 기능을 통한 동기화를 꾀할 것이다.

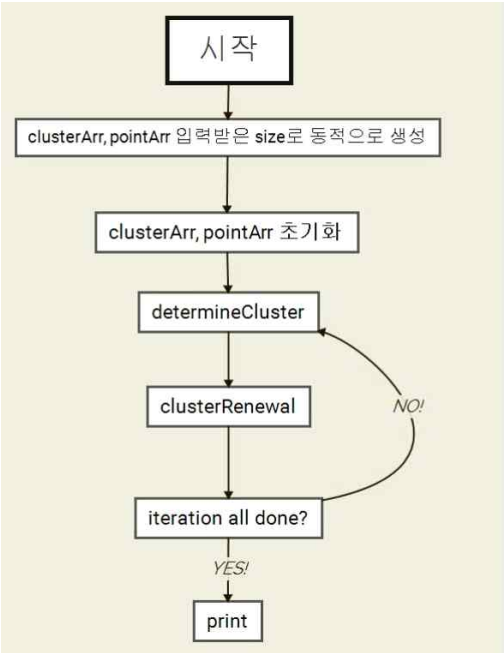
2. 결과 보고서

가. 프로그램의 동작 과정

1) noparallel.c

입력값을 받은 뒤, kmeans.h 파일에 정의된 noparallel_process 함수에 받은 인수를 넘겨주어 noparallel_process를 수행하도록 한다. 위 과정을 testCase 숫자만큼 반복한다.

noparallel_process 함수는 pointArr과 clusterArr을 동적으로 생성하여 받아낸 입력값으로 초기화한 뒤 iteration 수만큼 각 점 및 cluster 배열에 대하여 determineCluster function과 clusterRenewal function을 순서대로 수행한다. 여기서 **determineCluster는 당해 점과 가장 가까운 cluster를 결정하는 함수**이고, **clusterRenewal**은 그 iteration에서 모든 점에 대하여 determineCluster가 수행된 후, **cluster point의 좌표를 갱신하는 함수**이다. 한편 이 때 cluster에 속하는 점의 개수가 0인 cluster는 그 좌표를 (100000, 100000)으로 하여, (과제 전제사항이 point 좌표값의 절대값은 10000을 넘지 않는다 하였음) 차후 어떠한 점도 당해 cluster에 속할 수 없도록 배제한다. 이후 결과값을 print하고, 할당한 메모리를 free 해 준다.



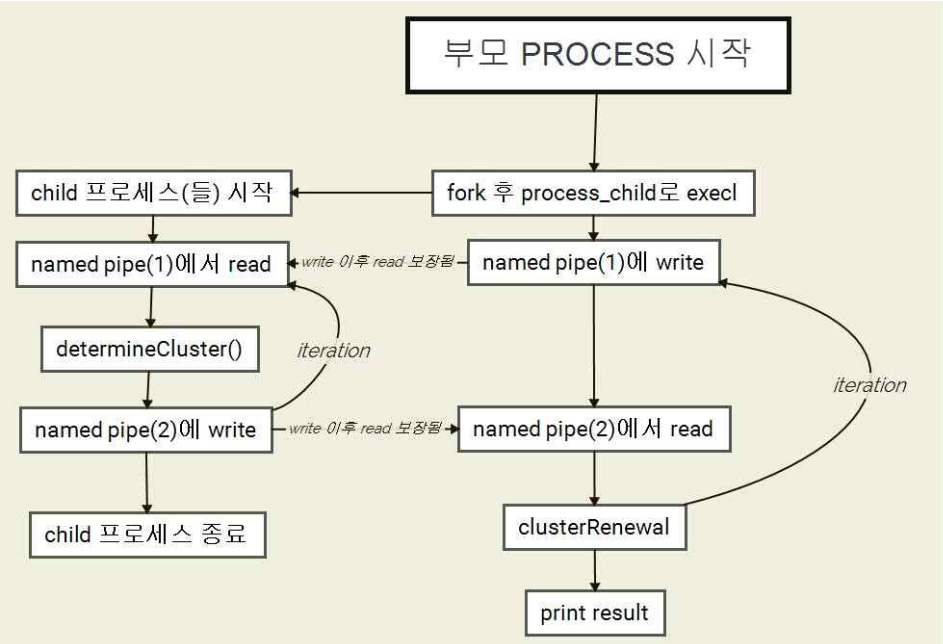
<간단한 noparallel_process의 순서도>

2) process.c

pointNum이 일정 number 이하라면, multiprocessing으로 처리하지 아니하고, 위에서 언급한 noparallel_process로 작업을 처리하게 한다. 여기서 그 일정 수는, NOPARAL_NUM으로써, kmeans.h 헤더 파일에 '10'으로 정의되어 있다. 만일 point의 개수가 이를 넘어선다면, multiprocessing으로 처리하게 된다.

프로세스간 point 배열과 cluster 배열을 공유해야 하기에, shared memory에 당해 배열을 위한 공간을 할당한다. 그 후 PROCNUM만큼의 프로세스를 fork한 뒤(**iteration 밖!**), execl함수로 하여금 fork된 process의 binary image를 바꿔준다. 이 때, 각 child process가 필요한 인수는 execl 함수의 인수로써 전달해 준다. 한편, iteration에 들어가기 전에 parent process와 child process가 통신할 수 있는 named pipe를 생성하는데, 이 때 전에 생성된 pipe를 먼저 지워준다.

iteration의 안에서 child process가 determineCluster function을 수행한 뒤, parent process가 clusterRenewal function을 수행하도록 서로 동기화시켜야 하는데, 여기서 iteration 밖에서 생성한 named pipe를 이용한다. named pipe의 read함수는 당해 pipe의 다른 쪽 끝에서 write 되기 전까지는 block된다는 점을 통하여 동기화를 구현한다. 주요 기능을 요약한 간략한 block diagram은 아래와 같다.

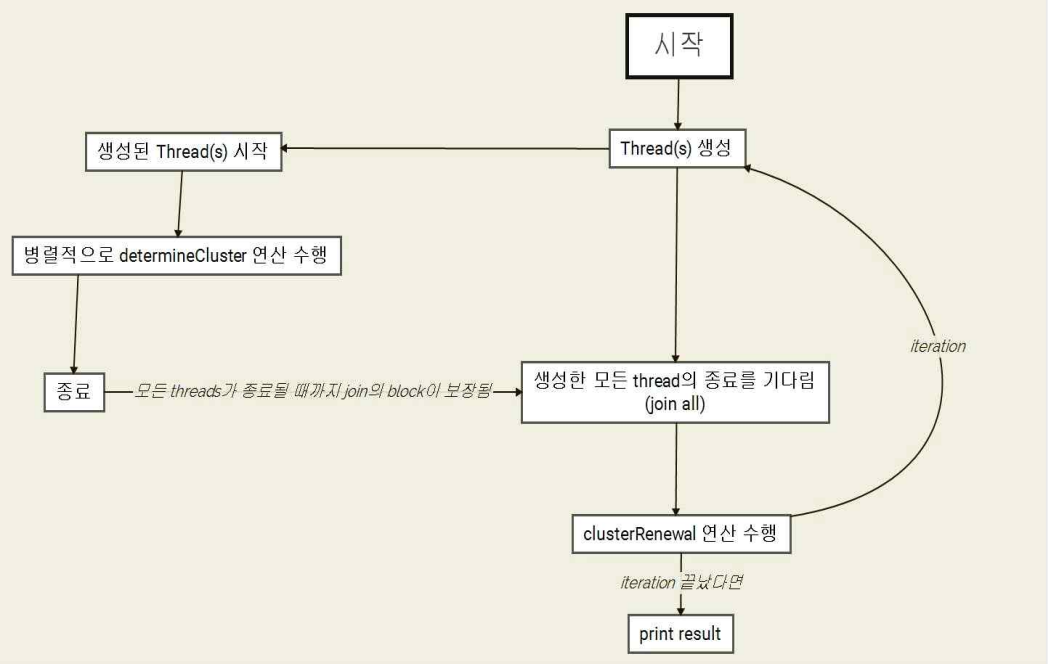


<간단한 process.c의 순서도>

각 자식 프로세스들은 공유메모리에 저장되어 있는 point배열의 startIndex부터 endIndex까지 point들에 대하여 determineCluster 연산을 수행한다. (Data parallelism) 따라서 각 자식 프로세스는 $\frac{pointNum}{PROCNUM}$만큼의 determineCluster연산을 병렬적으로 수행하게 되는 것이다.

3) thread.c

process.c와 대체적으로 비슷한 동작을 보이며 실행된다. process.c와 마찬가지로 NOPARAL_NUM 이하의 pointNum에 대해선 noparal_process로 실행되고, pthread_create로 생성된 thread들은 data parallelism 방식으로 각자 determineCluster 연산들을 수행한다. 다만 process.c와 달리, threads는 iteration 안에서 생성되므로 매 iteration마다 새로 만들어진다. 따라서 자식 thread를 중간에 **'멈추었다가 재개'할 필요는 없으며, 다만 부모 thread는 모든 자식 thread에 대해 join해 주면 된다.** 간단한 블록 다이어그램은 아래와 같다.



<간단한 thread.c의 순서도>[1)]

1) 순서도에는 입력을 받거나 변수를 할당하는 등의 묵시적으로 당연히 수행되어야 하는 작업들이 생략되어 있을 수 있음

나. Makefile 설명

1) Makefile 내용

```
all: noparallel process process_child thread
noparallel: noparallel.o kmeans.o
	gcc -o noparallel noparallel.o kmeans.o -lm
noparallel.o: noparallel.c kmeans.c
	gcc -c noparallel.c kmeans.c -lm
kmeans.o: kmeans.c
	gcc -c kmeans.c -lm
process: process.o kmeans.o
	gcc -o process process.o kmeans.o -lm
process.o: process.c kmeans.c
	gcc -c process.c kmeans.c -lm
process_child: process_child.o kmeans.o
	gcc -o process_child process_child.o kmeans.o -lm
process_child.o: process_child.c kmeans.c
	gcc -c process_child.c kmeans.c -lm
thread: thread.o kmeans.o
	gcc -o thread thread.o kmeans.o -lm -pthread
thread.o: thread.c kmeans.c
	gcc -c thread.c kmeans.c -lm -pthread
clean:
	rm *.o noparallel process process_child thread

```

2) Makefile 설명

noparallel.c process.c process_child.c thread.c 는 모두 내가 직접 제작한 kmeans.h 헤더를 포함하고, kmeans.h는 kmeans.c에 구현되어있으므로, 언제나 kmeans.c에 대하여 의존성을 갖는다. 따라서 위와 같은 Makefile의 의존관계가 나타나게 된다.

한편, **process_child**는 process에서 fork된 자식 process를 프로그램 내부에서 execl하기 위한 실행파일이다. **별도로 터미널에서 실행시키는 파일이 아니다.**

모든 실행파일은 위 Makefile의 내용에서 볼 수 있다시피, object file을 link하여 만드는 방식으로 만들었으므로 build 과정에서 필연적으로 object file이 생기게 된다. make clean으로 이러한 object file과 실행파일을 지워주기 위하여 clean을 위와 같이 정의하였다.

다. uname -a 실행화면(개발환경)

```
root@ubuntu:~/OSA# uname -a
Linux ubuntu 4.4.0-119-generic #143-Ubuntu SMP Mon Apr 2 16:08:24 UTC 2018 x86_64 x86_64 x86_64 GNU/Linux
```

(VMware 가상환경에서 상기 OS 상에서 vi로 개발하였다.)

라. 결과분석(주로 수행시간의 관점에서)

1) Thread, Process 생성의 오버헤드 분석

가) 분석을 위한 code

```c
#include "kmeans.h"
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/wait.h>

#define NUM 1000

void* myfunc(void* par){}

int main(void){
    int num = NUM;
    int fn, status;
    pthread_t tid[NUM];

    printf("making 1000 thread\n");
    long bt = getTime();
    for (int i=0;i<num;i++){
        pthread_create(&(tid[i]),NULL,myfunc,NULL);
    }
    long at = getTime();
    long rt = at - bt;
    for (int i=0;i<num;i++){
        pthread_join(tid[i], NULL);
    }
    printf("thread making time: %ld\n", rt);
    printf("making 1000 processes\n");
    bt = getTime();
    for (int i=0;i<num;i++){
        fn = fork();
        if (fn == 0){
            return 0;
        }
    }
    at = getTime();
    rt = at - bt;
    while(wait(&status)>0);
    printf("process making time: %ld\n", rt);
}
```

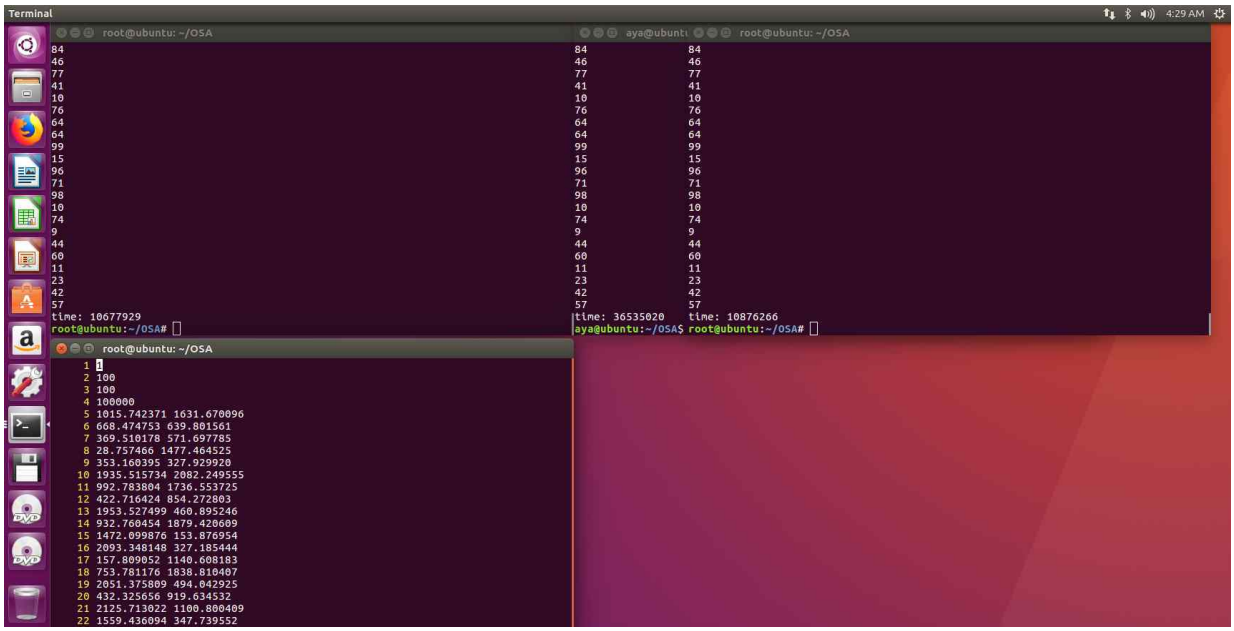
1000개의 thread를 생성하는 시간과, 1000개의 process를 fork하는 시간을 microsecond 단위로 각각 출력해주는 code로써, 그 생성시간을 측정하는데 의의가 있기에, thread routine은 빈 함수이고(myfunc) fork된 자식 process는 바로 종료된다.

나) 실행결과

```
root@ubuntu:~/OSA# ./pt.out
making 1000 thread
thread making time: 70198
making 1000 processes
process making time: 109685
root@ubuntu:~/OSA# ./pt.out
making 1000 thread
thread making time: 65624
making 1000 processes
process making time: 111019
root@ubuntu:~/OSA# ./pt.out
making 1000 thread
thread making time: 73010
making 1000 processes
process making time: 107263
root@ubuntu:~/OSA# ./pt.out
making 1000 thread
thread making time: 72012
making 1000 processes
process making time: 109999
root@ubuntu:~/OSA# ./pt.out
making 1000 thread
thread making time: 56251
making 1000 processes
process making time: 103961
```

실행 때마다 차이는 있지만 **thread 생성의 오버헤드가 process 생성의 오버헤드보다는 적다는 점**을 그 수행시간 비교를 통해 확인할 수 있었다. (thread 생성 시간은 같은 갯수 process 생성 시간의 약 **70%**)

2) 결과 화면 예시



터미널 창이 총 4개가 있는데, 왼쪽 위부터 실행파일 process, noparallel, thread에 '<'명령어를 사용하여 input.txt를 입력하여 실행한 결과이고, 왼쪽 아래의 터미널 창은 프로그램으로 직접 생성한 input.txt이다. 여기서는 testCase == 1, iteration == 100, clusterNum == 100, pointNum == 100000으로 설정되었다. 예시 결과 화면에서는 수행시간 확인의 편의를 위하여 마지막에 time을 한번 더 출력하도록 printf문을 넣어주었는데, 보다시피 같은 input에 대하여 **noparallel은 가장 느린 수행속도**를 보이고, thread, process등 **멀티테스킹을 활용한 실행파일은 그에 비하여 3.6배 정도 빠른 수행속도를 보이는 것**을 확인할 수 있다. 다만 iteration 밖에서 fork한 뒤, named fifo의 입출력을 통하여 동기화를 꾀한 **process가**, 각 iteration마다 새롭게 thread를 생성한 **thread보다** 약 200,000microseconds, 즉 **0.2초정도 빠르게 수행**되는 것을 확인할 수 있다.

생각건대 iteration 밖에서 fork한 process는, thread에 비해 매 iteration마다 새롭게 task를 생성하는 오버헤드가 제거되어 성능이 미약하게나마 향상된 것으로 보인다. 이하 보다 정량적으로 검토한다.

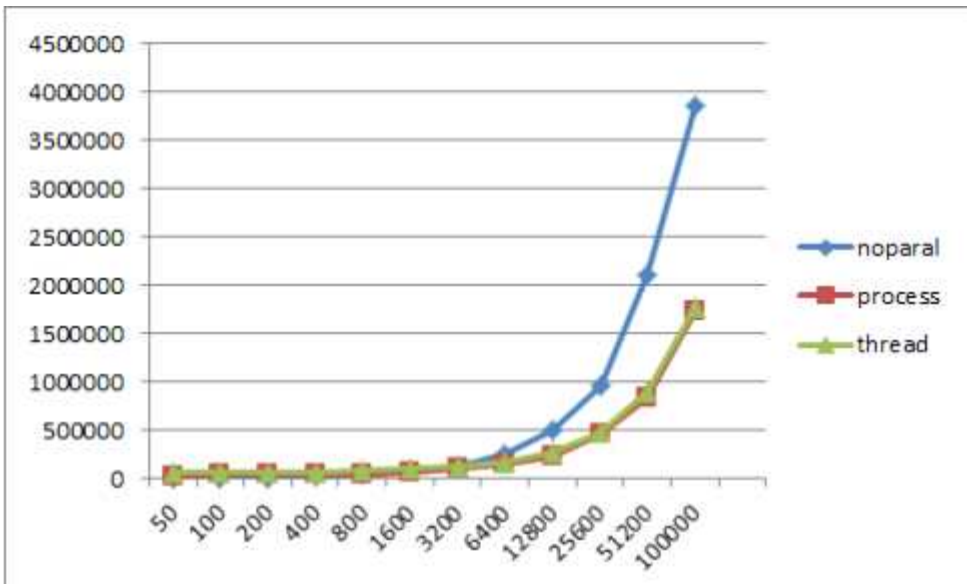
3) 수행 시간의 정량적 검토

가) dataNum에 따른 각 실행파일 별 실행시간 표

dataNum	50	100	200	400	800	1600	3200	6400	12800	25600	51200	100000
noparal	2941	5983	14093	21876	37683	78221	135988	249453	500495	955341	2113975	3852740
process	31300	44398	40373	47709	51800	69287	100399	148306	240356	456887	848017	1729109
thread	65307	61927	66172	70885	79811	98657	129162	175746	273498	481586	899269	1765896

input.txt의 pointNum을 과제 spec상 최대 pointNum인 100,000까지 지수적으로 늘려가며 (2배씩) 각 실행파일 별로 그 수행시간을 microsecond 단위로 측정한 결과값을 표로 나타낸 것이다. 이를 그래프로 나타내면 아래와 같다. (다만 iteration은 100, clusterNum은 20으로 고정하였다.)

나) 그래프를 통한 시각화



다) 결과 분석

처리해야 할 **data의 양이 적을 때는 noparallel 방식이, data의 양이 클 때는 multiprocessing/multithreading 방식이 좋은 성능을 보이는 경향**을 확인할 수 있다. 구체적으로는, process의 경우에 pointNum이 1600일 때 최초로 noparallel 성능을 앞질렀고, thread의 경우에는 pointNum이 3200일 때 최초로 noparallel의 성능을 앞지른다. 그리고 언제나 process가 thread보다는 (미약하게나마) 빠른 성능을 보였는데, 앞서 검토하였듯 이는 process에선 iteration 마다 새로운 task를 생성하는 오버헤드가 없기 때문이다.

라) 추가 분석 - process, thread의 개수와 성능

kmeans.h 헤더파일에서 PROCNUM 과 THNUM을 define함으로써 fork되는 process 수와 생성되는 thread 수를 정의하고 있으므로, 이를 바꾸고 재컴파일해 가면서 각 실행파일에 일정한 input.txt (결과 예시 화면의 그것)를 넣어주면서 그 실행시간을 비교해 보았다. 결과를 표로 나타내면 다음과 같다.

process, thread num	5(default)	10	20	40
process	10677929	8494671	7552251	7575272
thread	10876266	9889202	7954176	7995068

(시간 단위: microsecond)

일단 **병렬적으로 수행되는 task의 개수가 늘어난다 하여**, 그 수행시간에 그에 **정확하게 반비례하여 줄어드는 것은 아님**이 명확하게 관찰되었고(다만 그 개수가 늘어남에 따라, 어느 정도는 미약하게나

마 줄어드는 경향성은 확인할 수 있었음), 그마저도 **병렬적으로 수행되는 task들이 지나치게 많아지면 더 이상 성능이 좋아지지 않는 현상**도 관찰되었다.(표에서는 PROCNUM, THNUM == 40의 경우)

생각건대, 병렬적으로 수행되는 task는 앞서 검토한 결과에서 보듯 다다익선이 아니므로, 작업의 특성에 따라 적절한 수치를 정하여 그 병렬처리 방법론을 고민해야 할 것이다.

마. 과제 수행 과정에서의 문제점, 및 해결방안

1) 문제점

process.c의 작성에 있어, 부모 프로세스와 자식 프로세스 간의 자료(여기서는 pointArr과 clusterArr)공유는 IPC중 shared memory를 활용함으로써 어렵지 않게 해결할 수 있었으나, iteration 밖에서 fork하였으므로, **양 프로세스의 작업 "순서"를 동기화 시키는 작업**이 난해하였다. 단순히 wait(&status) system call을 사용하여 해결할 수도 없는 것이, 부모 process는 자식 process의 종료를 기다리는 것이 아니고, determineCluster의 연산 작업의 수행이 끝나기를 기다리는 것이기 때문이다.

2) 시도 1 - signal 사용

kill() 과 pause(), 그리고 signal handler를 사용하여 양 프로세스 간 능동적인 동기화를 꾀하였다. 그러나 부모 process에서 자식 process로의 signal 전송을 통한 자식 process 작업 재개는 구현이 잘 되었으나, 여러 개의 자식 process로부터 비동기적으로 수신되는 여러 signal을 부모 프로세스에서 처리하는 것에 실패하였다. CPU scheduling 문제 때문인 것으로 보인다. 즉 자식 process에서 자신의 작업 완료를 알리는 의미로 부모 process에게 kill함수를 통하여 SIGUSR1 signal을 전송하고 pause()되면 cpu가 부모 process에게 할당되어야 그 자식 process가 보낸 signal을 처리할 텐데, 그 대신 CPU가 다른 자식 process에게 할당되고, 새롭게 CPU를 할당 받은 자식 프로세스도 자신의 작업을 끝내고 완료의 의미로 SIGUSR1 signal을 보내버려 부모 process는 먼저 온 signal을 놓쳐 버리게 되는 결과가 나와 deadlock에 빠지는 것이다.

3) 시도 2 - parentsWait() 함수 사용

공유 메모리에 flag의 의미로 자식 프로세스 개수만큼 정수 배열을 할당하고, 이를 0으로 초기화 시킨다. 각 자식 프로세스(0~PROCNUM-1 까지의 정수 중 각각 자신 고유의 index를 갖음)는 자신의 작업이 완료되면 0으로 세팅된 자신 index의 flag를 1로 바꾼다. 부모 프로세스는 busy waiting 상태로 이 flag 배열을 상시 검사하고 있다가 모든 flag가 1이면 무한루프를 break 하고, 다시 모든 flag를 0으로 세팅한다. 여기서 busy waiting으로 낭비되는 시간을 줄이기 위해 cpu 할당 시간을 인위적으로 양보하는 sched_yield함수를 사용한다. parentsWait()의 코드는 다음과 같다.

```
void parentsWait(int* flagArr, int size){
    long cnt = 0;
    while(1){
        int sum = 0;
        for (int i=0;i<size;i++){
            sum += flagArr[i];
        }
        if (sum==size){
            for (int i=0;i<size;i++){
                flagArr[i] = 0;
            }
            break;
        }
        if (cnt > 10000){
            printf("arr: %d %d %d %ld\n",flagArr[0], flagArr[1], flagArr[2], cnt);
        }
        cnt++;
        sched_yield();
    }
}
```

그러나, 이유를 알 수 없게도 아주 가끔씩 무한 루프에서 나오지 못하는 deadlock이 관찰되어 폐기되었다.

4) 해결 - named pipe 사용

어차피 busy waiting은 좋지 않은 코딩이고, 웬만하면 event driven으로 구현하고 싶어 방법을 찾다가 named fifo의 read()에서의 blocking 기능을 활용하고자 하였다. named pipe의 close()과정에서 for문의 index를 하나 잘못 적어 test과정에서 OS가 허용하는 최대 file open 갯수를 넘어 named pipe의 open()함수에서 deadlock이 발생하는 시행착오가 있었으나, 결국 성공적으로 동기화에 성공하였다.

바. 결론 및 느낀점

과제를 수행하며, 여타 과제와 비교를 불허하는 수의 시행착오를 겪으면서 동기화 문제의 난해함과 cpu scheduling의 불확실성과 같은 병렬 프로그래밍의 어려운 점, 및 IPC에 대해 꽤나 많은 것을 학습할 수 있었던 기회가 되었다. 그러나 앞서 살피었듯 그러한 어려움에도 불구하고 병렬 프로그래밍이 특정 작업의 수행에 있어서는 확실히 수행시간의 측면에서 이득을 가져온다는 것이 가시적으로 확인되는 바, 문제 해결에 있어 언제나 하나의 대안으로서 염두해 두고 있어야 할 것이다.

사. Reference

1) 사전 레포트 첫 번째 장의 thread, process 비교 사진: https://sites.google.com/site/sureshdevang/thread-vs-process
2) 리눅스 커널 내부구조 (백승제, 최종무)
3) 리눅스 시스템 프로그래밍(로버트 러브)
4) https://ko.wikipedia.org/wiki/%ED%94%84%EB%A1%9C%EC%84%B8%EC%8A%A4_%EA%B0%84_%ED%86%B5%EC%8B%A0
(IPC 관련 내용 참고)