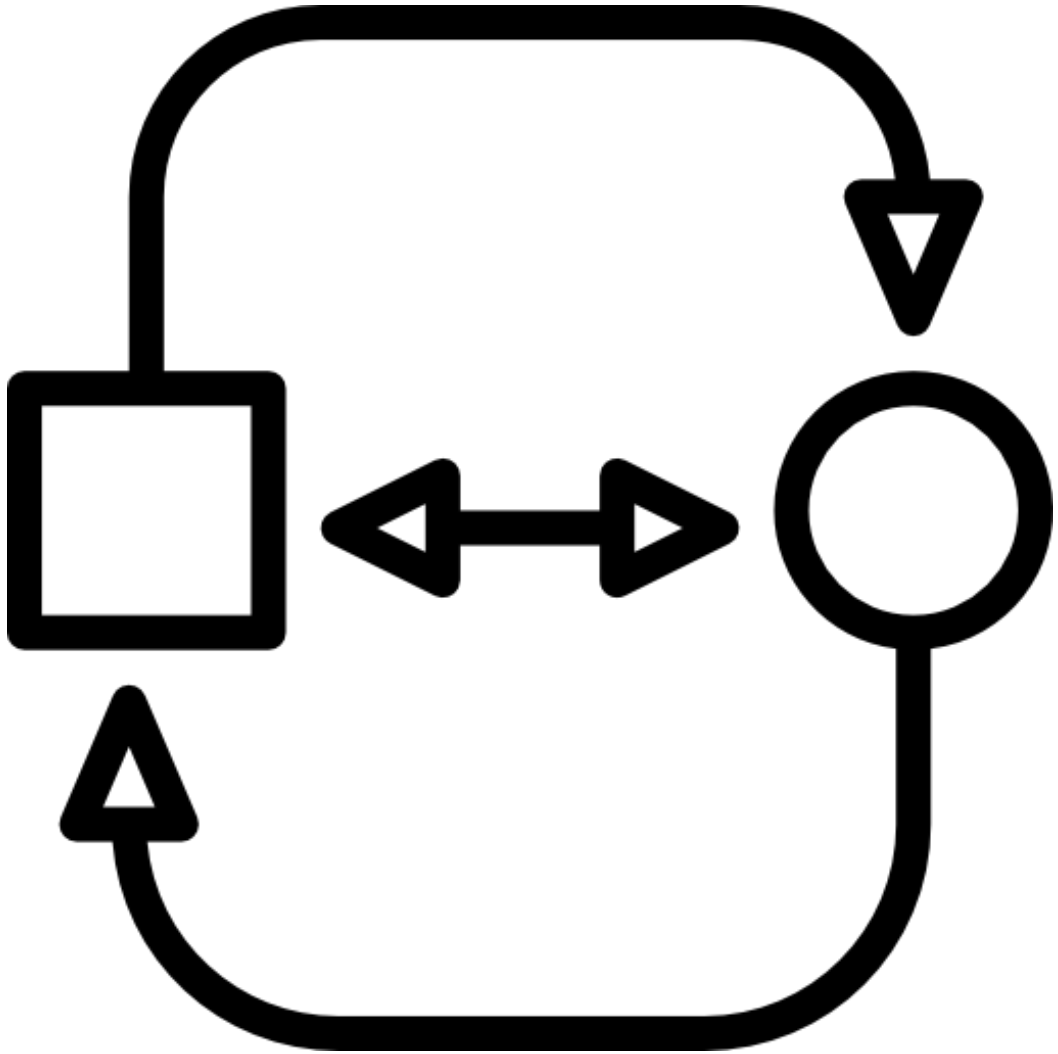


MBTI

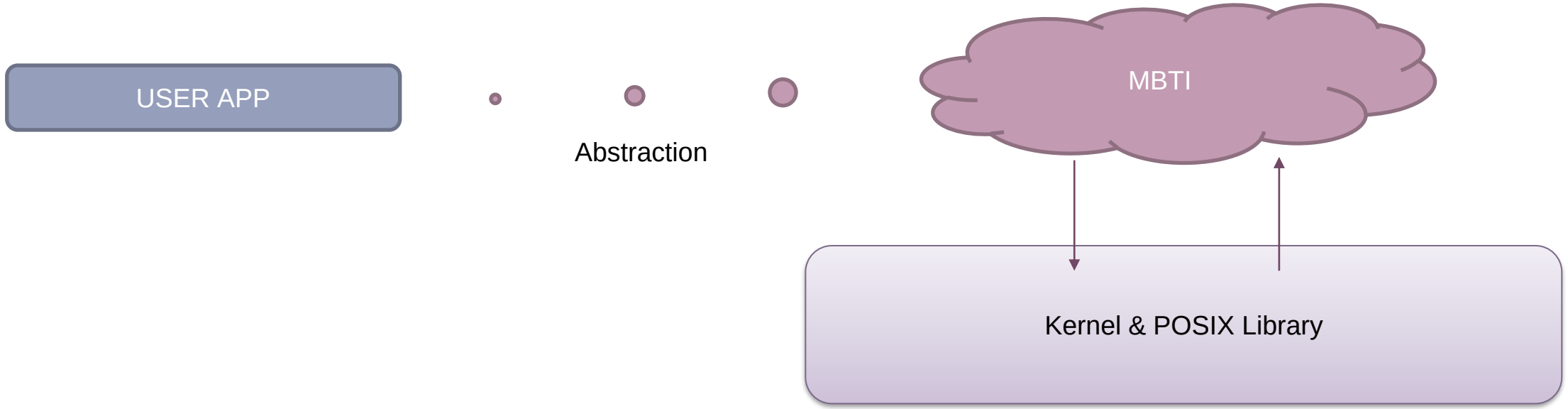
TEAM - 3

- 201511246 김상재
- 201711395 박성준
- 201511284 이종빈
- 201511290 장서연

목차

MBTI 소개

- Micro Benchmark Testing Interface.
- 시스템 콜들의 다양한 환경 속 성능을 측정한다.
- POSIX 라이브러리를 포함한 커널 내부의 메커니즘 오버헤드를 분석하기 위함이다.
- 최대한 시스템 콜 & POSIX 라이브러리의 오버헤드만을 측정하도록 노력한다.



개발 취지

유저 레벨에서 호스트 소프트웨어의 성능향상 요소를 판단하는 것이 아닌 시스템 레벨에서의 개선사항을 찾기 위한 것이다.

- 시스템 호출과 User level간의 인터페이스를 구현하는 유저, 혹은 기존의 시스템 User level 최적화를 넘어 시스템 최적화를 꾀하는 유저에게 범용적으로 쓰일 수 있다.

- 시스템 호출의 다양한 환경에 따른 성능 측정한다.
  - 코어의 개수
  - 프로세스 개수
  - 토폴로지 종류
  - 패턴의 횟수

개발 취지

MBTI는 실제 소프트웨어를 테스트하는 것이 아닌 사용자가 원하는 시스템 콜과
미리 정해진 시스템 콜들의 실행 동작 패턴을 설정하여 테스트 한다.

- 편의성
  a. 테스트 환경에 대한 오버헤드를 줄일 수 있다.
  b. 여러 환경에서 시스템 콜 성능 측정을 간단히 구성할 수 있다.
  c. CSV 파일 형식의 데이터로 부터 성능 측정의 산출물을 도출해 낼 수 있다.
- 다양성
  a. 사용자 필요에 따라 다른 환경에서 테스트의 성능 측정 산출물을 도출해낼 수 있다.
  b. 특정 시스템 콜에 집중해서 분석, 비교할 수 있다.

개발 환경

- GCC : gcc (Ubuntu 9.3.0-10ubuntu2) 9.3.0
- Python3 : Python Interpreter : Python 3.8.2
- OS : Ubuntu 20.04 LTS
- Kernel : 5.4.0-45-generic
- CPU: AMD Ryzen Thread Ripper 2950X, Cores/Threads: 16-Cores, 16-Threads

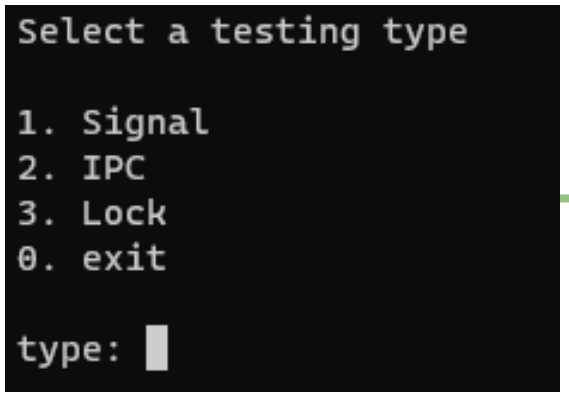
개발 목표

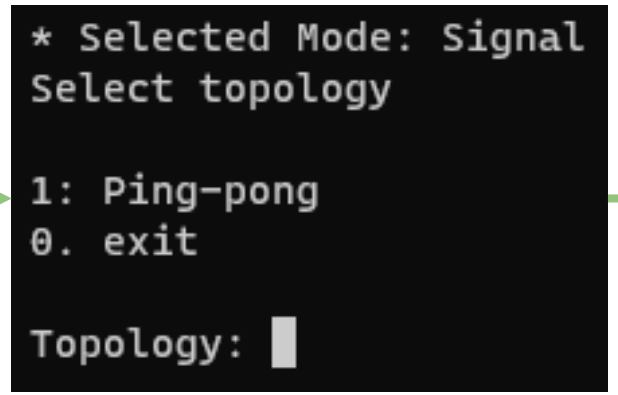
- 각 시스템콜 마이크로 벤치 마크 스위트는 공통적으로 다음과 같은 기능들을 가지고 있어야한다.
  a. 코어 수에 따른 시스템 콜의 성능을 측정한다.
  b. 프로세스 혹은 스레드 수에 따른 시스템 콜의 성능을 측정한다.
  c. 토폴로지 종류에 따른 시스템 콜의 성능을 측정한다.
  d. 패턴의 반복에 따른 시스템 콜의 성능을 측정한다.
  e. 성능 측정 결과값을 CSV파일 형식으로 저장 및 그래프를 산출한다.

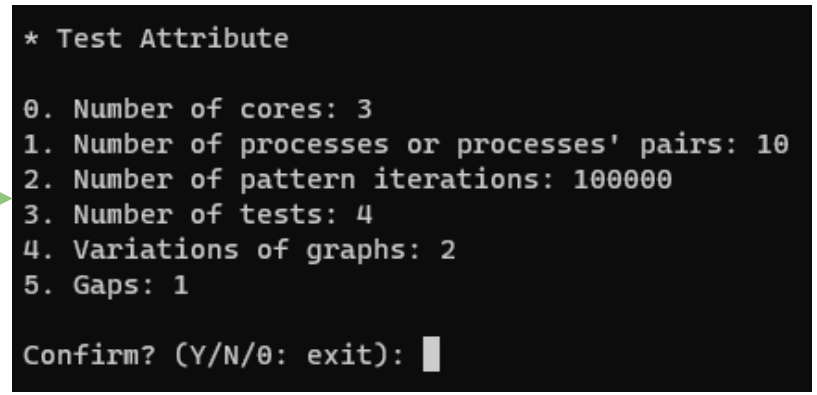
역할소개

김상재 signal 파트 구현 담당 main script 구현 담당	**박성준** message queue 파트 구현 담당
이종빈 pthread mutex lock 파트 구현 담당	**장서연** semaphore 파트 구현 담당

구현 - Main







1. 테스팅 시스템콜 설정

2. Topology 설정

3. Test 속성 입력

구현 - Main

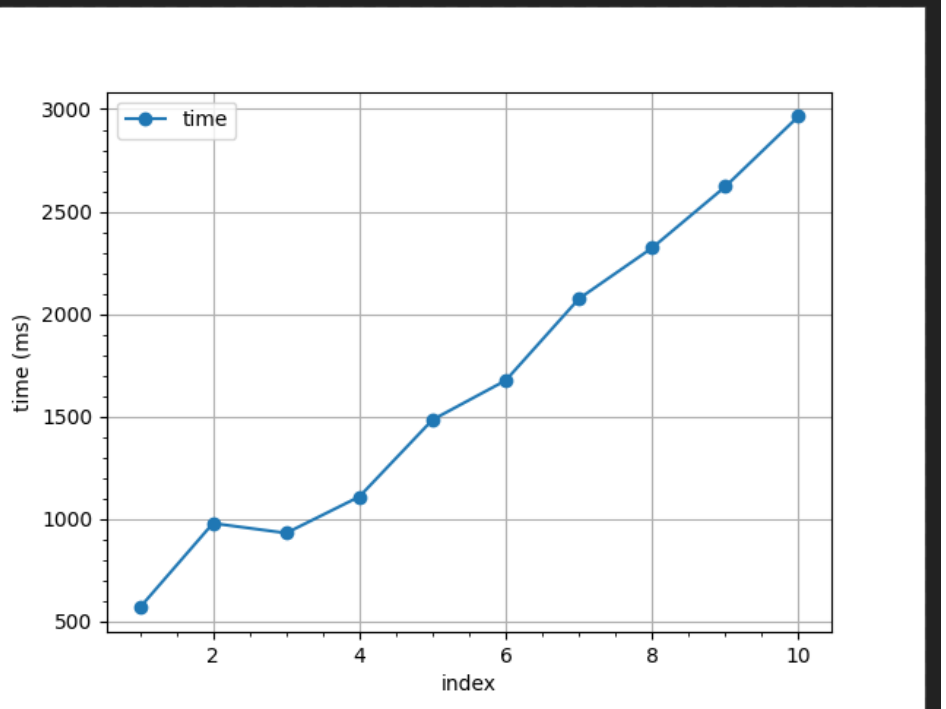
4. 측정 결과 CSV 파일 저장

```
* CSV File is saved (file name: 202097203033.csv)
Save as png file? (Y/N): Y
```

```
~/Doc/Ko/Microbench/mbti on first_srs !1 ?1 > cat 202097203033.csv
index,time
1,566.64965025
2,978.1622639999999
3,930.55032275
4,1108.1521115
5,1483.7556425
6,1675.72720275
7,2074.4664135000003
8,2322.576235
9,2621.9084705
10,2964.9910585000002
```

5. 측정 결과 PNG 파일 저장

202097203033.png



구현 - Signal

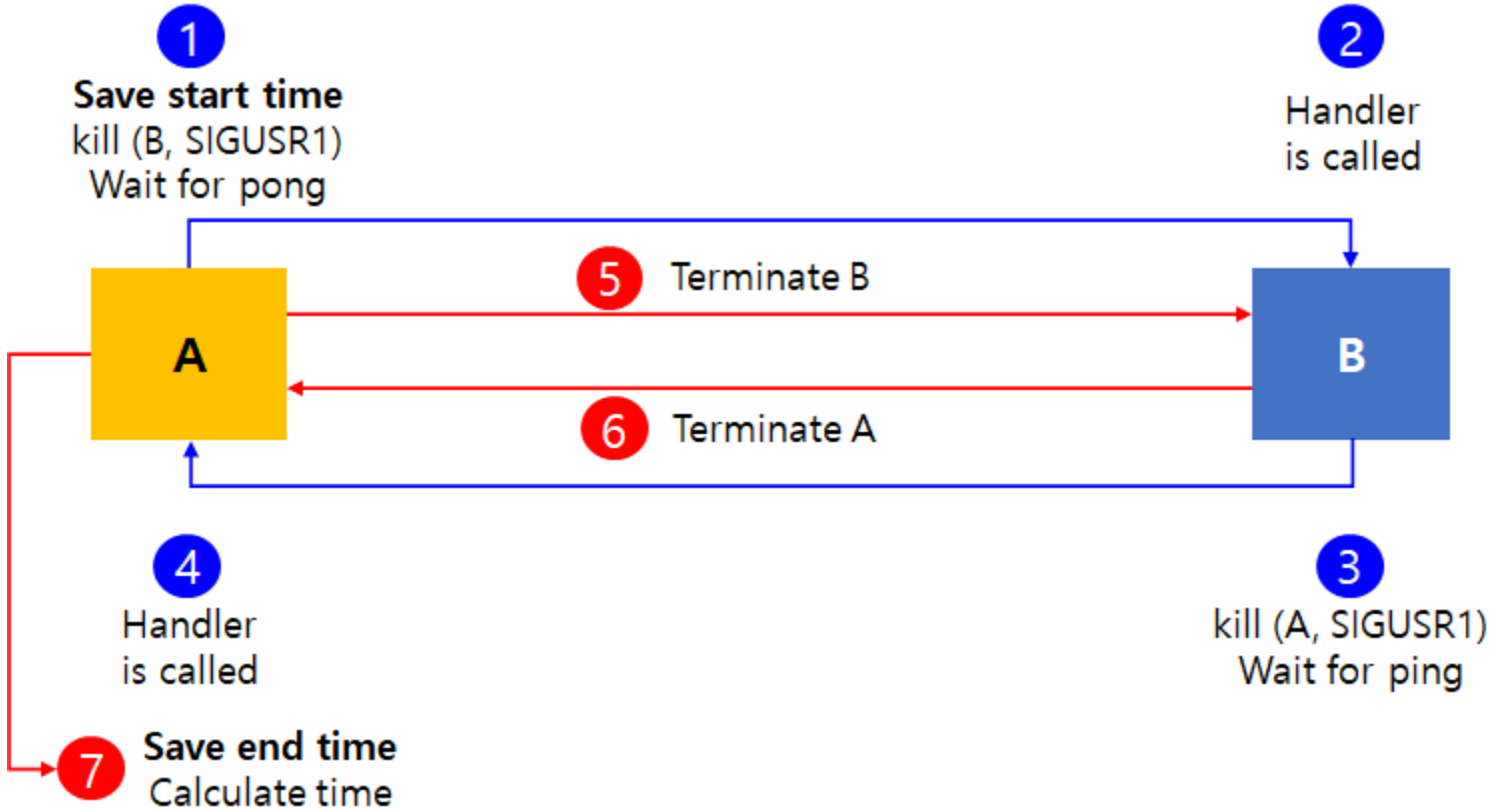
1. sig_test_init

- 해당 topology의 초기 함수를 실행한다.
- 테스팅을 위한 프로세스들 생성 후 대기 상태로 전환시킨다.

2. sig_test_exec

- 생성된 프로세스 개수 확인한다.
- 생성된 프로세스에 SIGCONT를 전송하여 동시에 실행한다.
- 각 쌍 마다 측정된 시간을 수신한다.
- 평균 시간 계산 후 반환한다.

구현 - Signal (Ping-Pong)



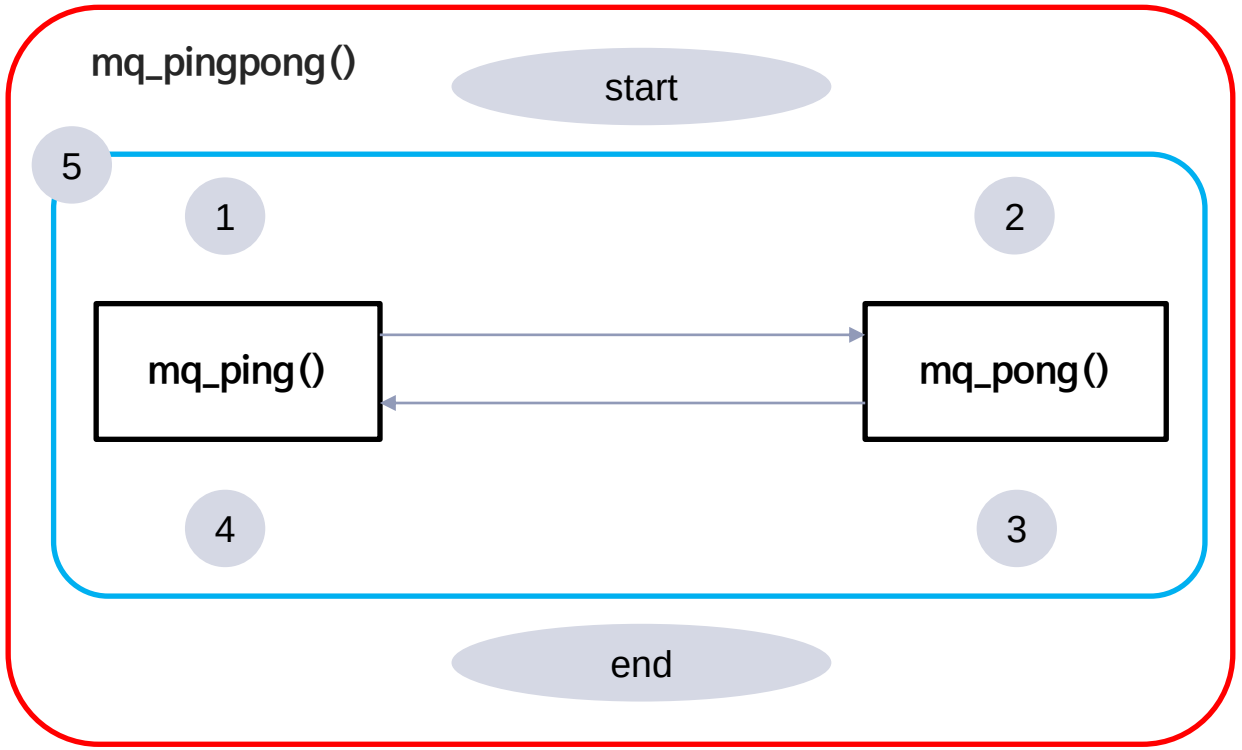
1 2 3 4

- A와 B 사이에서 ping-pong을 주고받는 과정이다.
(1->2: ping / 3->4: pong)
각 Handler는 signal이 수신될 경우 상대 Process에 signal를 송신한다.

5 6 7

- A가 pattern_iteration 수 만큼 signal을 전송했을 경우 B를 종료한다.
- A가 종료시간을 저장한 후 시간 차이를 계산한다.

구현 - IPC (Ping-Pong)

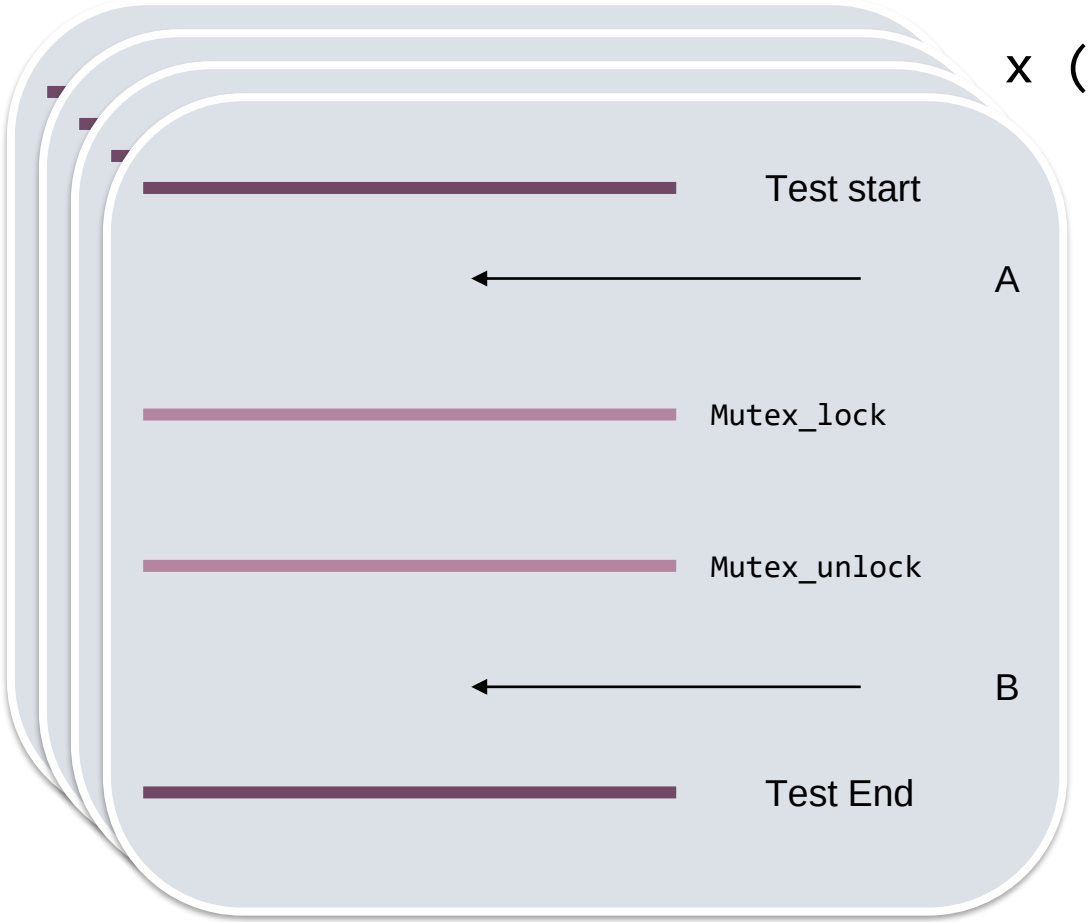


start. 반복문을 통해 생성할 프로세스 개수만큼 실행한다. 홀수 번 째는 mq_ping, 짝수 번 째일 때는 mq_pong프로세스를 생성한다.

1. mq_pong의 메시지 큐로 메시지를 보낸다.
2. mq_pong의 메시지 큐의 메시지를 받는다.
3. mq_ping의 메시지 큐로 메시지를 보낸다.
4. mq_ping의 메시지 큐의 메시지를 받는다.
5. 1~4번 과정을 반복횟수 만큼 실행하고 측정한 시간 값을 반환한다.

end. 반환 값으로 얻은 측정 값들을 종합해 핑퐁의 평균값을 최종적으로 반환한다.

구현 - pthread_Mutex(global_variable)



- 공유 변수는 하나의 Critical Section에서 하나의 스레드만 접근 가능하도록 보호되어 있다.
- 여러 Thread 들이 같은 공유 변수를 Critical Section에서 서로의 자원을 경쟁한다.
- MBTI에서 global_variable case는 A 포인트와 B포인트 사이의 시간을 측정한다.

구현 - Semaphore (SPSC)

입력된 iteration과 thread쌍의 개수를 fork()를 활용하여 SPSC 환경에 맞추어 생산자와 소비자 쓰레드를 만든다.

전역 변수로 선언된 공유 버퍼에 생산자가 만든 아이템을 넣고 소비자가 아이템을 소비한다.

```
cpu_set_t cpuset;
int num_cpus = *((int*)arg);
CPU_ZERO(&cpuset);
CPU_SET(num_cpus, &cpuset);
pthread_t current_thread = pthread_self();
int result = pthread_setaffinity_np(current_thread, sizeof(cpu_set_t), &cpuset);
```

코어의 개수 및 위치를 설정하기 위하여 생산자와 소비자 함수 각각 core affinity 관련 함수를 적용한다.

구현 - Semaphore (Ping-Pong)

```
clock_gettime(CLOCK_MONOTONIC,&sem_begin);
printf("begin time : %ldns\n",sem_begin.tv_nsec);
```

생산자 함수를 실행함과 동시에 시작점을 측정한다.

```
for(int i=0;i<sem_user_iter;i++)
{
    sem_wait(&sem_empty);
    sem_wait(&sem_mutex);
    sem_put_item();
    sem_post(&sem_mutex);
    sem_post(&sem_full);
}
```

FULL,EMPTY,MUTEX의
세가지 세마포어를 사용하여
sem_wait(), sem_post() 함수로
임계영역(본 코드에서는 공유 버퍼)에 접근한다.
<<생산자 함수 semaphore 사용 부분

```
clock_gettime(CLOCK_MONOTONIC,&sem_end);
printf("end time : %ldns\n",sem_end.tv_nsec);
```

입력된 생산자와 소비자의 iteration 수 만큼 실행 후 종료 지점을 측정한다.

Ping-Pong 결과 (Signal)

- Increment process test

CPU: Intel Core i3-6100

Cores/Threads: 2-Cores, 4-Threads

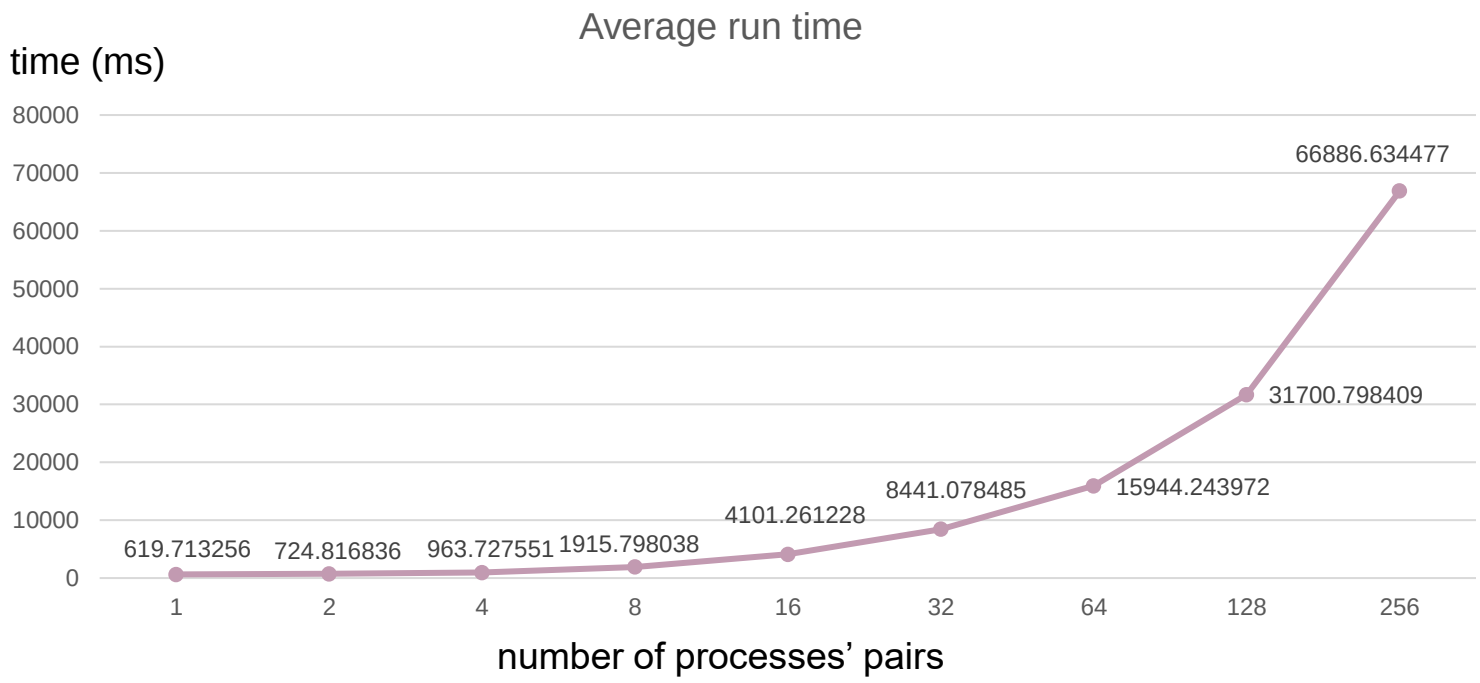
OS: Ubuntu 20.04 x64

Pattern iterations: 100,000

Number of runs per processes: 10

Number of cores: 2-Cores, 4-Threads

- 실행 코어 수를 고정하고 프로세스 쌍의 개수를 증가시킨 실험이다.



Ping-Pong 결과 (Signal)

- Increment core test

CPU: AMD Ryzen ThreadRipper 2950X

Cores/Threads: 16-Cores, 16-Threads

OS: Ubuntu 20.04 x64

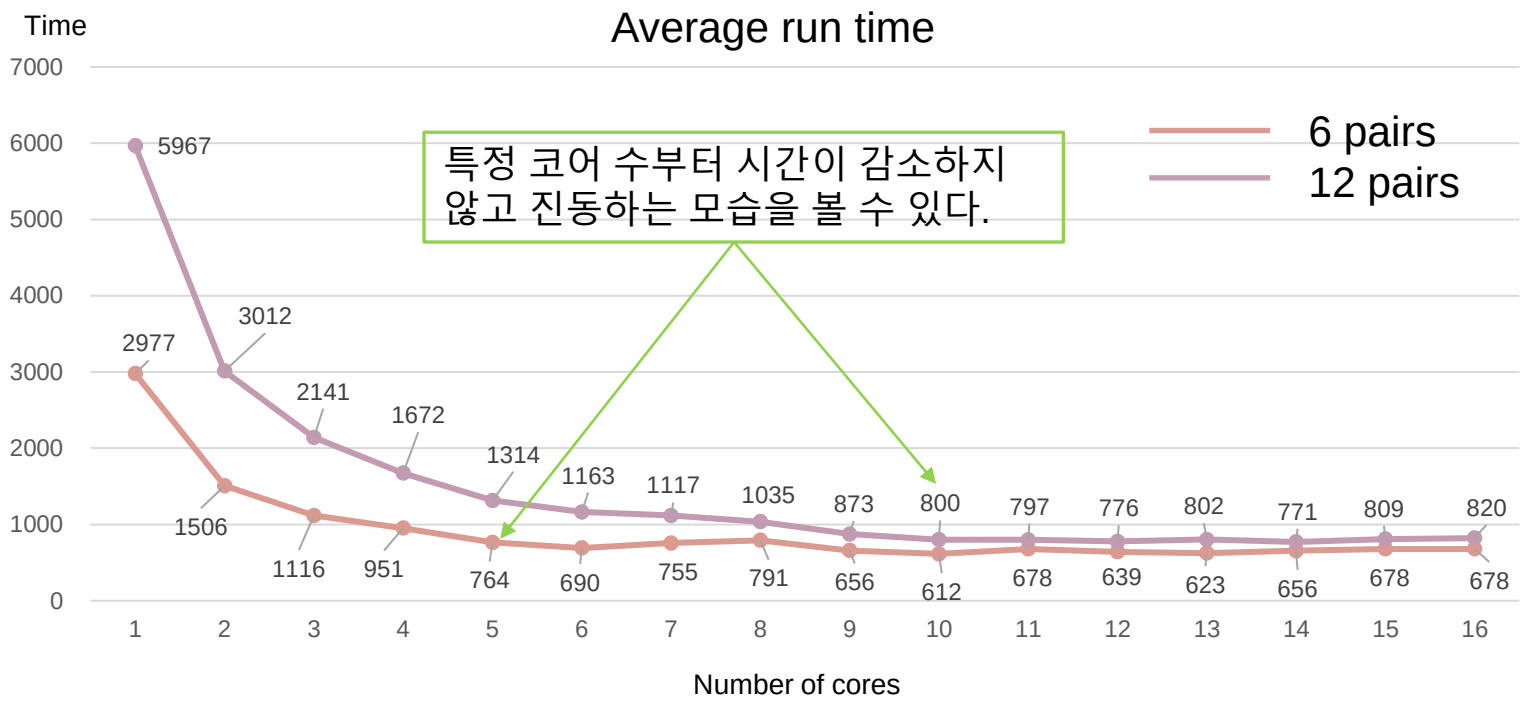
Pattern iterations: 100,000

Number of runs per core: 10

Processes' pairs:

6 (12 processes), 12 (24 processes)

- 실행 프로세스 쌍의 개수를 고정하고 실행 코어 수를 증가시키었다.



Ping-Pong 결과 (Signal)

- Core position test

CPU: AMD Ryzen ThreadRipper 2950X

Cores/Threads: 16-Cores, 16-Threads

- CPU의 물리적인 구조에 의한 Overhead가 발생할 수 있을 것 같아서 추가적으로 실험을 진행하였다.

Ping-Pong 결과 (Signal)

- **Core position test**

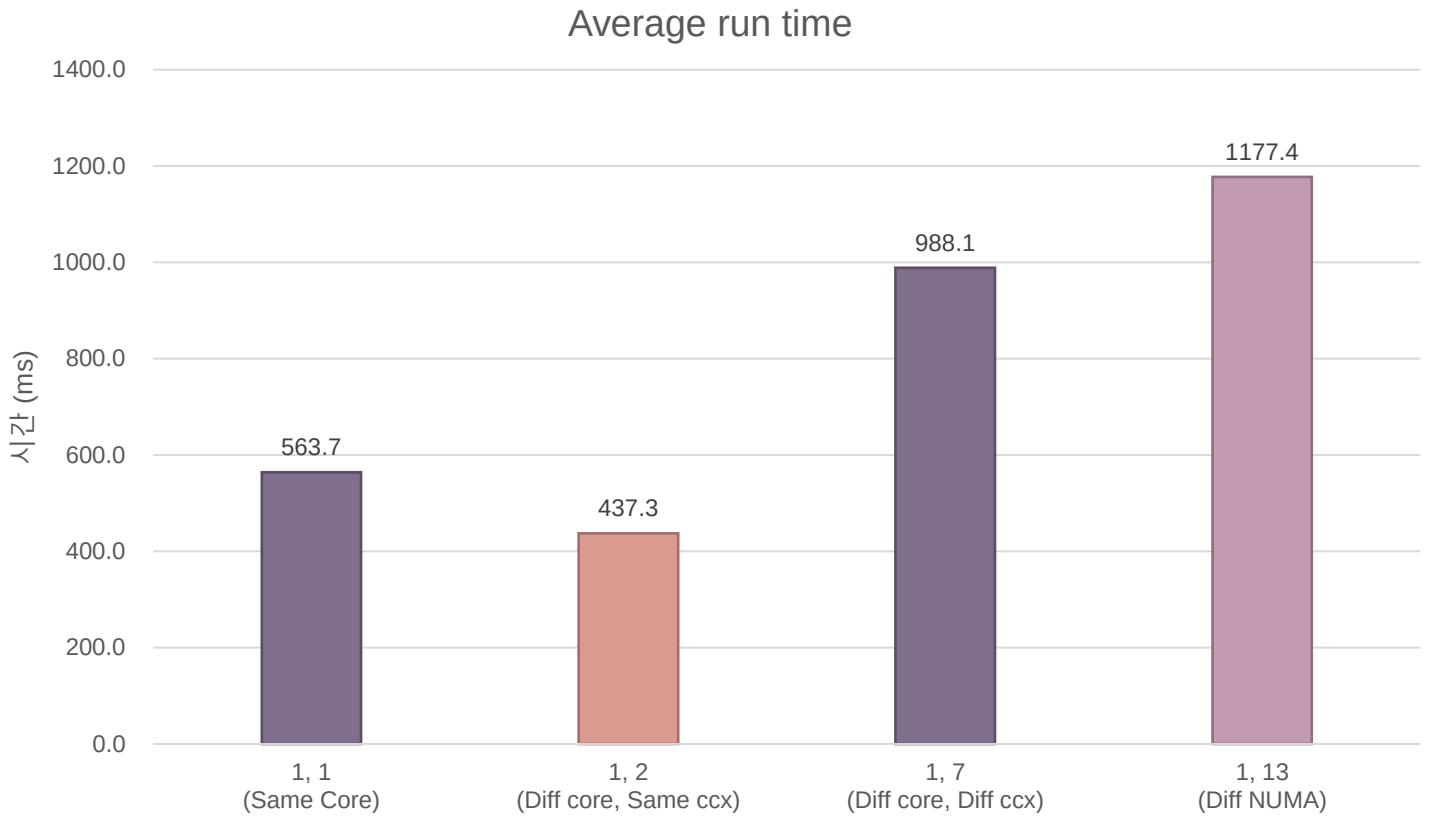
CPU: AMD Ryzen ThreadRipper 2950X

Cores/Threads: 16-Cores, 16-Threads

OS: Ubuntu 20.04 x64

Pattern iterations: 100,000

Processes' pairs: 1 (2 processes)



Ping-Pong 결과 (IPC Message Queue)

- Increment process test

CPU: AMD Ryzen ThreadRipper 2950X

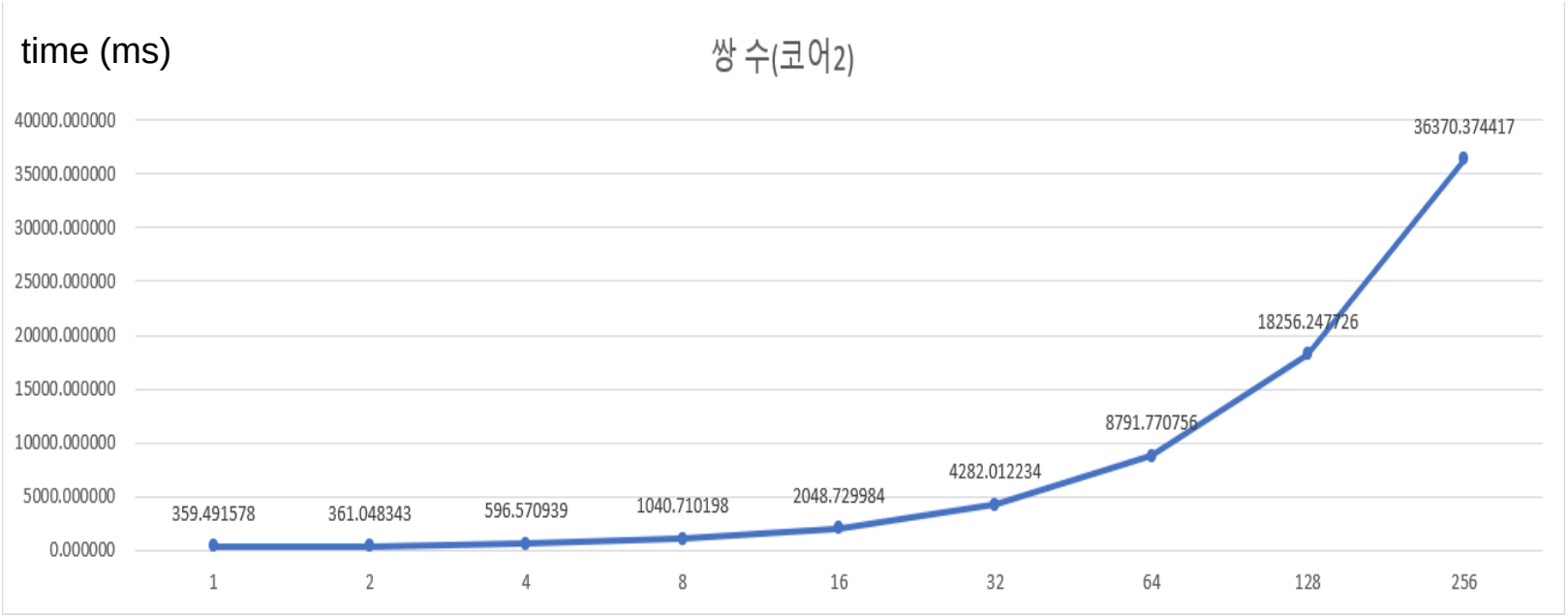
Cores/Threads: 16-Cores, 16-Threads

OS: Ubuntu 20.04 x64

Pattern iterations: 100,000

Number of runs per test: 10

Number of cores: 2-Cores, 2-Threads



Ping-Pong 결과 (IPC Message Queue)

- Increment core test

CPU: AMD Ryzen ThreadRipper 2950X

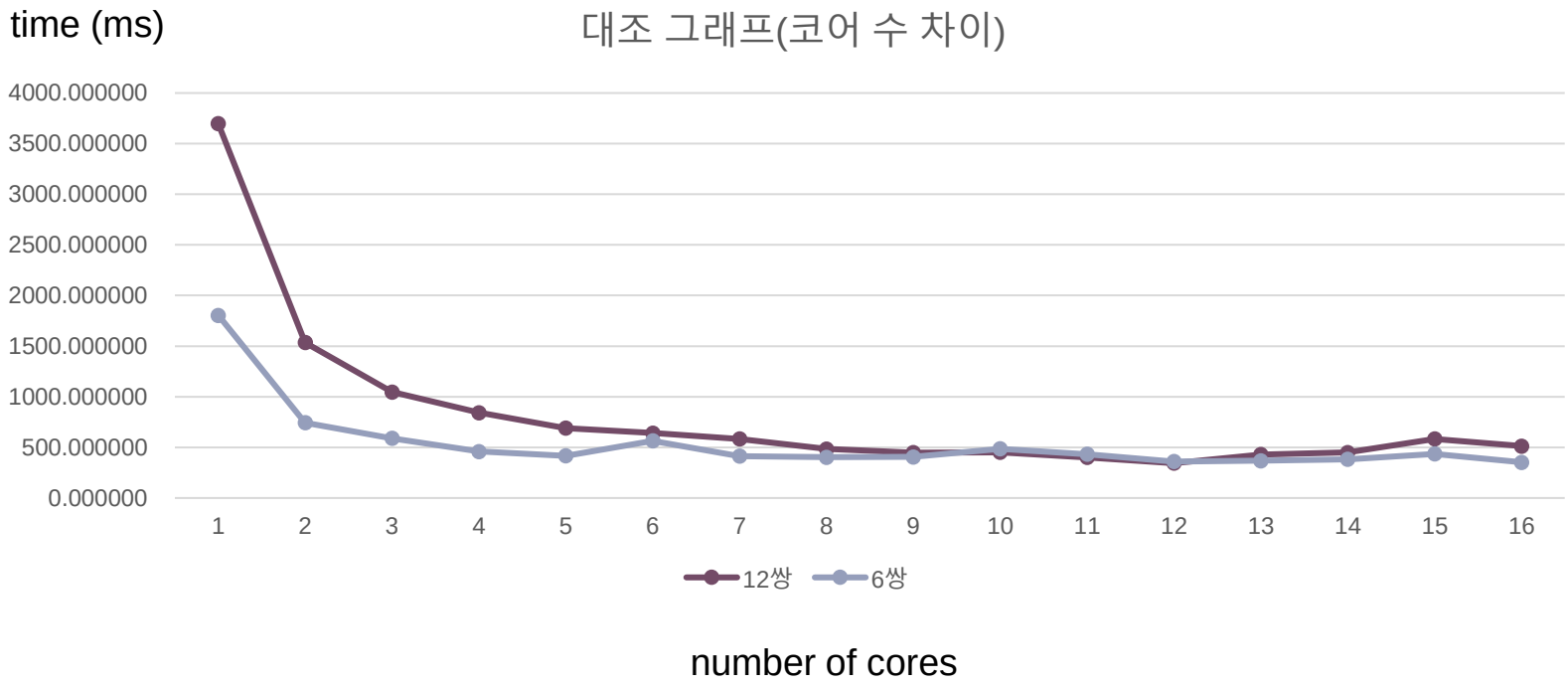
Cores/Threads: 16-Cores, 16-Threads

OS: Ubuntu 20.04 x64

Pattern iterations: 100,000

Processes' pairs:

6 (12 processes), 12 (24 processes)



SPSC 결과 (Semaphore)

- Increment process test

CPU: AMD Ryzen ThreadRipper 2950X

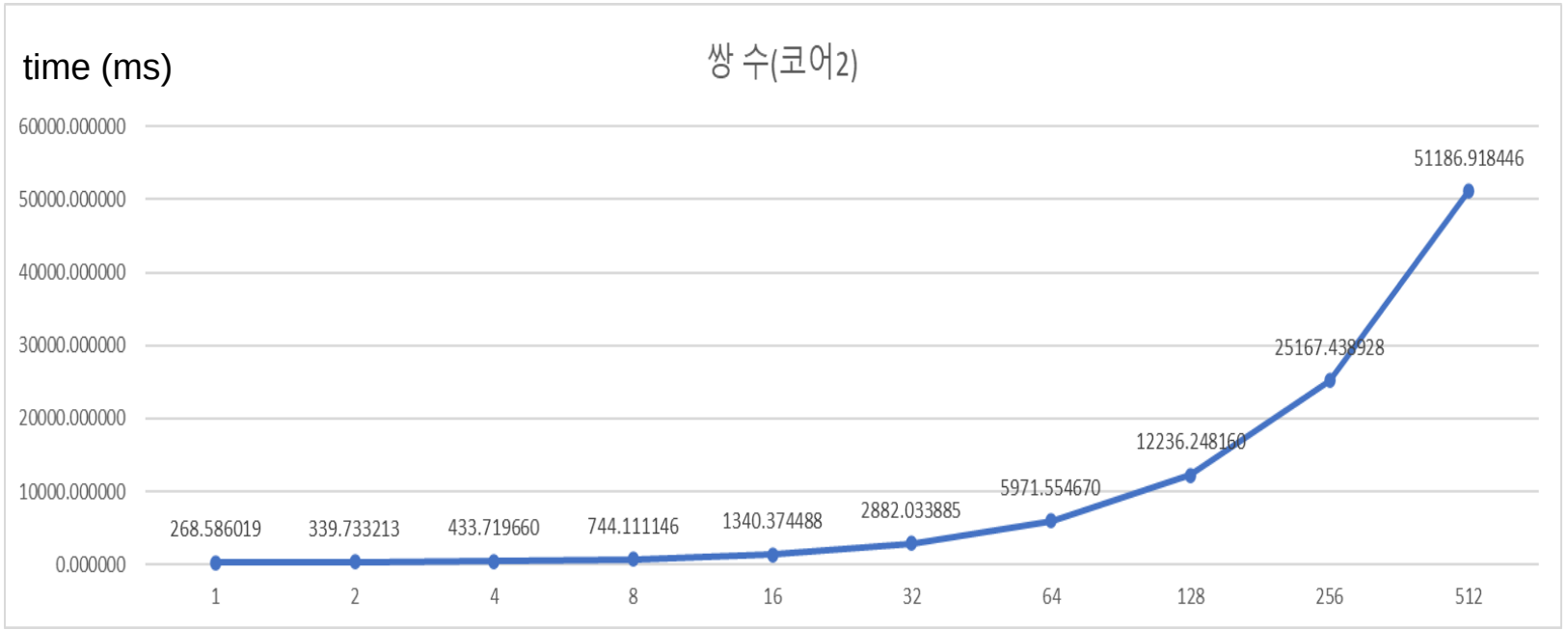
Cores/Threads: 16-Cores, 16-Threads

OS: Ubuntu 20.04 x64

Pattern iterations: 100,000

Number of runs per test: 10

Number of cores: 2-Cores, 2-Threads



SPSC 결과 (Semaphore)

- Increment core test

CPU: AMD Ryzen ThreadRipper 2950X

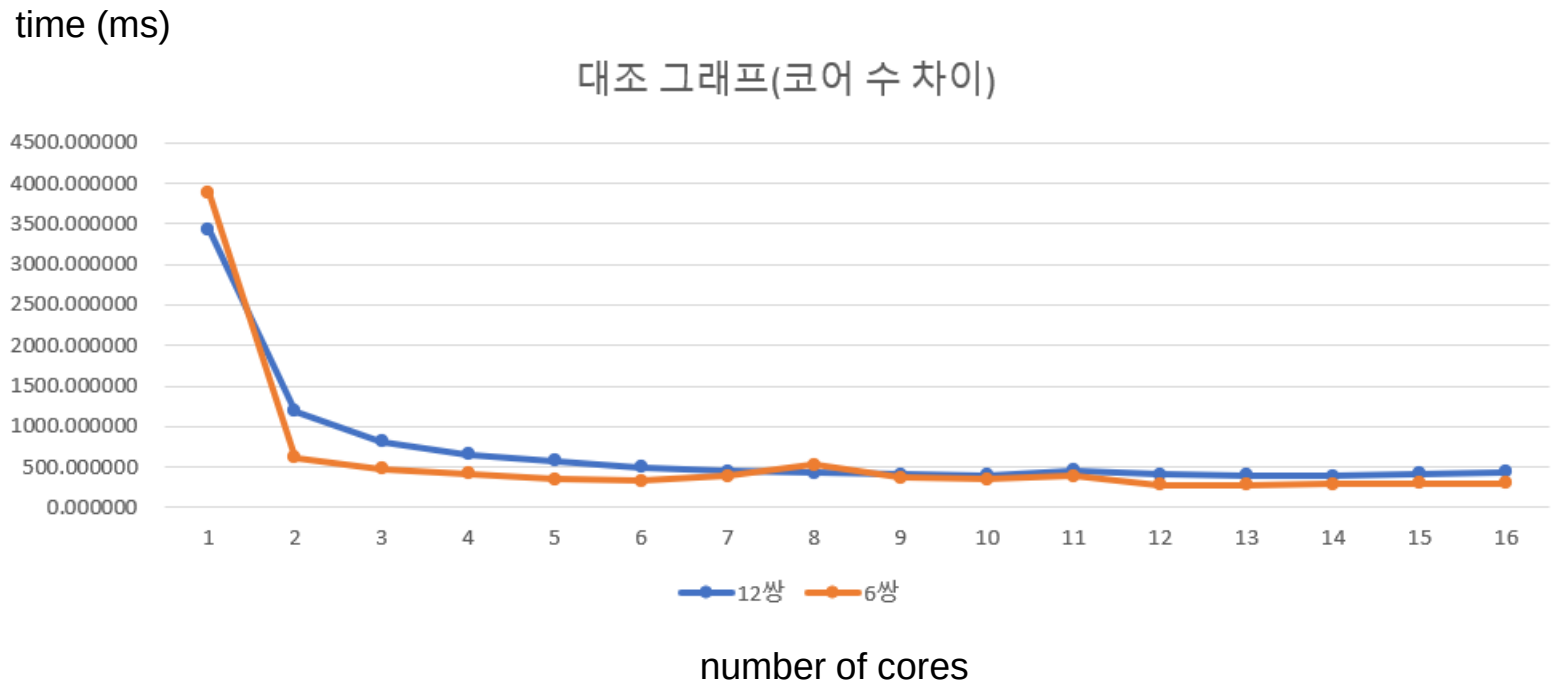
Cores/Threads: 16-Cores, 16-Threads

OS: Ubuntu 20.04 x64

Pattern iterations: 100,000

Processes' pairs:

6 (12 processes), 12 (24 processes)



pthread_Mutex(global_variable)

- Increment process test

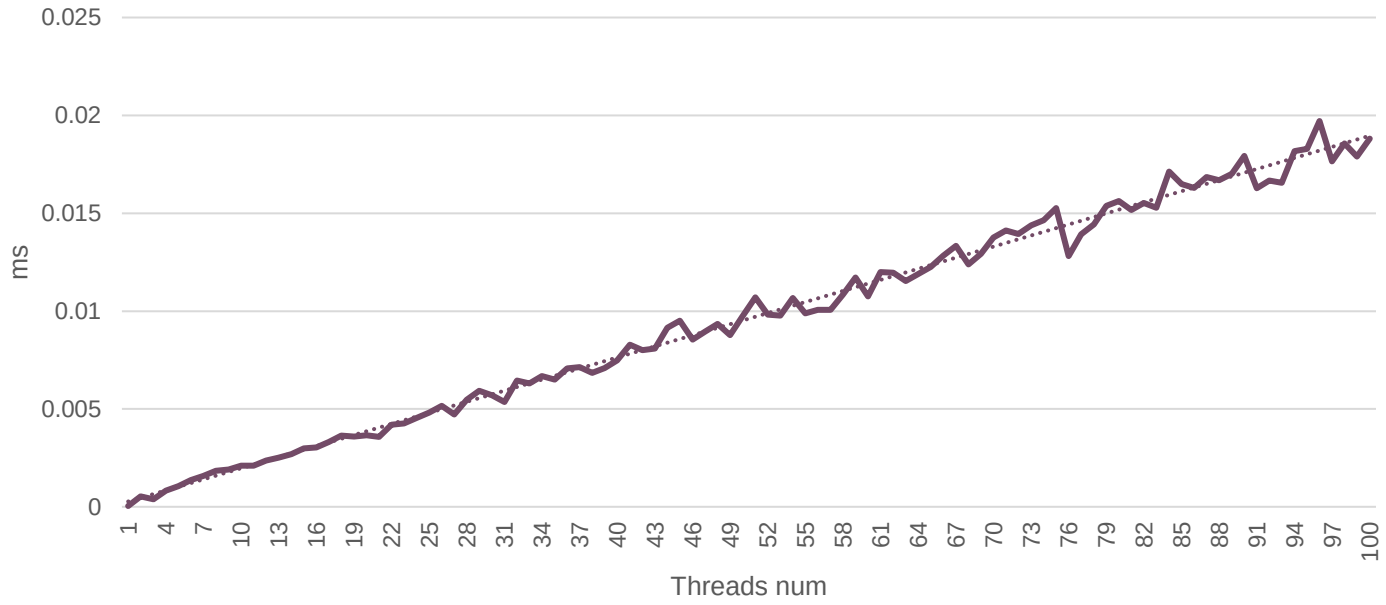
CPU: Intel(R) Core(TM) i7 860

Cores/Threads: 4-Cores, 8-Threads

OS: Ubuntu 20.04 x64

Pattern iterations: 1,000,000

Number of cores: 4-Cores, 8-Threads



number of threads

결과 분석

- **Increment process test**
  - 프로세스 수 증가에 따라서 시간도 전체적으로 Linear하게 증가함을 볼 수 있다.
- **Increment core test**
  - 특정 코어 수부터는 시간이 감소하지 않고 진동하는 것을 볼 수 있다.
- **Core position test**
  - 단순히 Signal에 따른 오버헤드가 아닌 프로세스가 실행되는 Core, CCX, NUMA에 따라서 실행 시간이 달라진다.
  - CPU의 구조와 프로세스가 위치한 Core에 따라도 실행 시간이 달라지는 것을 알 수 있었다.

향후 계획

- 특정한 software의 Microbenchmark를 추가로 구현한다.
  - Overhead가 어디에서 주로 발생하는 지 확인 할 예정이다.
    - System Call, Scheduler, Data I/O, 물리적인 구조에 의한 Overhead
  - 필요할 경우 System call이 수행되는 Kernel code를 분석할 예정이다.
  - Message queue SPSC 토폴로지 추가로 구현한다.
- 노드 외 통신 프로토콜별 시스템 콜 추가 구현한다.
- Pthread_mutex 기반 Ping-Pong 결과값 출력할 예정이다.