1.

- **자신의 구현에 대한 설명:**

A) global 변수 part로 잡아, part에는 각 스레드가 한 작업의 마지막(즉, 한 스레드 당 최종 부분합)이 쌓이게 한다.

- parallel영역이 시작되고 for schedule(static)을 만나면 스레드 별로 루프가 분배되어 부분합이 쌓이고, part에는 각 스레드에서 작업한 최종 부분합이 쌓인다.

- 즉, in = [0,1,2,3,4,5]고 스레드가 2개면, part[0]=0, part[1] = 0+1+2 =3, part[2] = 3+4+5 = 12이고 이 스레드들의 작업은 barrier에 의해 다른 스레드까지 끝날 때까지 기다린다. 또 다시 스레드별로 setof에는 해당 스레드의 직전 스레드의 최종합까지가 누적된다. 즉, 스레드1의 setof=part[0] = 0, 스레드2의setof = part[0]+part[1] = 3. 그리고 각 스레드별로 최종 out에 setof를 넣어주면, 스레드2 기준으로 out[3],out[4],out[5]에 in[0:2]까지의 합이 들어가 더해지므로 원하는 out이 생성된다.

- **기존 과제와 달리 prefix sum 구현은 double precision floating-point 형식을 사용하였다. 그 이유가 무엇 일지 생각해보자. Single precision 형식을 사용하면 어떤 문제가 발생하는가?**

A) double precision이 오차가 더 적다.

- **N = 134217728 일 경우의 순차 버전 및 병렬화 버전의 성능 비교. 만약 병렬화 버전의 성능이 기대보다 느리다면, 그 이유는 무엇인가?**

A) (사진참고) 병렬화버전이 성능이 더 좋긴 하지만, 코드 상 barrier를 두는 등 스레드들이 자신들의 일을 하고 서로 합칠 때까지 대기하는 부분 등에서 시간이 지체되는 부분이 생긴다.

```
shpc033@login0:~/hw3/prefix_sum$ ./run.sh -n 3 -m sequential 134217728
Options:
  METHOD: sequential
  Problem Size (N): 134217728
  Number of Iterations: 3

Initializing... done!
Calculating...(iter=0) 0.474850 sec
Calculating...(iter=1) 0.455784 sec
Calculating...(iter=2) 0.477178 sec
Validating...
Result: VALID
Avg. time: 0.469271 sec
Avg. throughput: 0.286013 GFLOPS
shpc033@login0:~/hw3/prefix_sum$ ./run.sh -n 3 -m parallel 134217728
Options:
  METHOD: parallel
  Problem Size (N): 134217728
  Number of Iterations: 3

Initializing... done!
Calculating...(iter=0) 0.082190 sec
Calculating...(iter=1) 0.082989 sec
Calculating...(iter=2) 0.083802 sec
Validating...
Result: VALID
Avg. time: 0.082994 sec
Avg. throughput: 1.617203 GFLOPS
```

2.

- **자신의 병렬화 방식에 대한 설명.**

A) omp parallel for에서 각 스레드별로 루프를 나눠서 돌며 행렬곱연산을 수행한다.

- **OpenMP는 Pthread와 달리 사용자가 명시적으로 스레드 생성 함수를 호출하지 않는다. OpenMP에서 thread 생성은 어떤 식으로 이루어지는가? 컴파일러와 런타임 시스템이 각각 어떤 역할을 수행하는 지 생각해보자.**

A) 컴파일러 지시어(omp parallel)이 병렬화를 할 구간을 정해주고, 런타임 라이브러리의 omp_set_num_threads(n)나 런타임 시스템(런타임환경) 상에서 스레드 개수가 설정된다. 컴파일러는 이를 읽어 병렬화 구간에 스레드를 알맞은 개수로 생성하도록 스레드 프로그램을 generate한다. 그러면 run시 master thread가 생성되고 나머지 thread가 생성되어 task를 처리한다.

- **스레드를 1개부터 256개까지 사용하였을 때의 행렬곱 성능을 측정해 보자 (스레드 개수는 적당한 간격을 두고 측정). 스레드 개수가 늘어남에 따라 일정한 추세로 성능이 증가하는가? 이유는 무엇인가?**

Number of threads: 16	Avg. throughput: 56.066171 GFLOPS
Number of threads: 32	Avg. throughput: 86.887307 GFLOPS
Number of threads: 48	Avg. throughput: 48.375404 GFLOPS
Number of threads: 64	Avg. throughput: 56.459031 GFLOPS
Number of threads: 80	Avg. throughput: 47.800356 GFLOPS
Number of threads: 96	Avg. throughput: 51.484724 GFLOPS
Number of threads: 112	Avg. throughput: 47.127552 GFLOPS
Number of threads: 128	Avg. throughput: 49.568267 GFLOPS
Number of threads: 144	Avg. throughput: 47.038069 GFLOPS
Number of threads: 160	Avg. throughput: 48.692664 GFLOPS
Number of threads: 176	Avg. throughput: 46.768018 GFLOPS
Number of threads: 192	Avg. throughput: 50.180349 GFLOPS
Number of threads: 208	Avg. throughput: 47.352225 GFLOPS
Number of threads: 224	Avg. throughput: 51.170762 GFLOPS
Number of threads: 240	Avg. throughput: 47.007839 GFLOPS
Number of threads: 256	Avg. throughput: 47.733096 GFLOPS

A) (우측 사진) 32까지 올라가다가 이후로 다시 떨어져 fluctuation현상을 보인다. 스레드 간의 task의 대기시간(및 join)으로 인한 병목효과와 스레드개수간의 어떤 trade-off가 가장 적은 개수가 32 개라고 사료된다.

- **가장 높은 성능을 보이는 스레드 개수에서 행렬곱 성능은 1번 문제에서 계산한 peak performance 대비 어느 정도인가? 더 높은 성능을 달성하기 위해선 어떤 점을 개선해야 하는가?**

A) 약 14-30배정도 차이난다. (즉, 14-30배 안 좋다) 개선된 행렬곱 알고리즘 자체를 쓰거나, blocksize단위로 읽고 쓰기를 진행하도록 스레드별로 강제할 수 있다면 더 개선될 것이라 생각된다.

- **OpenMP의 loop scheduling 방식에 대해 알아보자. static, dynamic, guided 방식이 각각 어떤 것인지 서술하고, 실험을 통해 성능을 비교하라.**

A) static: 병렬화 section의 태스크를 일정 덩어리로 '실행 전'에 나눠서 스레드에게 분배한다.

Dynamic: 덩어리별로 나눠두되, 구동 중 태스크가 끝난 스레드에게 남은 일을 분배한다.
Guided: static과 dynamic의 혼합으로, 작업이 할당될 수록 할당되는 덩어리의 크기가 감소한다는 차이점이 있다.

```
shpc033@login0:~/hw3/matmul$ ./run_performance.
Options:
  Problem size: M = 4096, N = 4096, K = 4096
  Number of threads: 32
  Number of iterations: 10
  Print matrix: off
  Validation: on

Initializing... done!
Calculating...(iter=0) 1.638216 sec
Calculating...(iter=1) 1.645770 sec
Calculating...(iter=2) 1.523815 sec
Calculating...(iter=3) 1.652877 sec
Calculating...(iter=4) 1.522272 sec
Calculating...(iter=5) 1.524132 sec
Calculating...(iter=6) 1.651203 sec
Calculating...(iter=7) 1.513019 sec
Calculating...(iter=8) 1.520548 sec
Calculating...(iter=9) 1.637562 sec
Validating...
Result: VALID
Avg. time: 1.582941 sec
Avg. throughput: 86.825043 GFLOPS
```

<- guided

```
~/.Xauthority
:~/hw3/matmul$ ./run_performance.sh
Options:
  Problem size: M = 4096, N = 4096, K = 4096
  Number of threads: 32
  Number of iterations: 10
  Print matrix: off
  Validation: on

Initializing... done!
Calculating...(iter=0) 1.666904 sec
Calculating...(iter=1) 1.674095 sec
Calculating...(iter=2) 1.531446 sec
Calculating...(iter=3) 1.676003 sec
Calculating...(iter=4) 1.533561 sec
Calculating...(iter=5) 1.537787 sec
Calculating...(iter=6) 1.665927 sec
Calculating...(iter=7) 1.528223 sec
Calculating...(iter=8) 1.536707 sec
Calculating...(iter=9) 1.660074 sec
Validating...
Result: VALID
Avg. time: 1.601073 sec
Avg. throughput: 85.841799 GFLOPS
```

<- static

```
shpc033@login0:~/hw3/matmul$ ./run_performance.sh
Options:
  Problem size: M = 4096, N = 4096, K = 4096
  Number of threads: 32
  Number of iterations: 10
  Print matrix: off
  Validation: on

Initializing... done!
Calculating...(iter=0) 1.501590 sec
Calculating...(iter=1) 1.370851 sec
Calculating...(iter=2) 1.511991 sec
Calculating...(iter=3) 1.379565 sec
Calculating...(iter=4) 1.511306 sec
Calculating...(iter=5) 1.378245 sec
Calculating...(iter=6) 1.507356 sec
Calculating...(iter=7) 1.369415 sec
Calculating...(iter=8) 1.373493 sec
Calculating...(iter=9) 1.374880 sec
Validating...
Result: VALID
Avg. time: 1.427869 sec
Avg. throughput: 96.254587 GFLOPS
```

<- dynamic

결과, dynamic이 제일 좋다.