Report

Assignment1: Trapezoidal Rule using MPI

제출일 2024.11.1

과목명 빅데이터처리

전공 모바일시스템공학과

학번 32204012

이름 전혜림

목차

1. Introduction (Background)

오늘날 과학 및 공학 문제의 복잡성이 증가하면서, 효율적인 수치 적분 방법과 병렬 컴퓨팅 기술의 중요성이 더욱 부각되고 있습니다. 본 과제는 이러한 필요성을 바탕으로, 분산 메모리 환경에서 여러 프로세스가 협력하여 계산을 수행하는 Message Passing Interface (MPI)와 Trapezoidal Rule 을 이용하여 적분을 계산하는 방법을 구현하고자 합니다.

Trapezoidal Rule 은 복잡한 함수를 사다리꼴 형태로 단순한 직선 구간들로 근사하여 넓이를 계산하는 방법입니다. 계산의 정밀도를 높이기 위해 구간을 세분화할수록 연산량이 증가하므로, 이를 신속하게 처리하기 위해 병렬 처리가 필수적입니다. 특히, MPI 는 병렬 처리 환경에서 각 프로세스 간의 통신을 가능하게 하여 대규모 계산을 여러 프로세스가 나누어 수행할 수 있게 합니다. 이와 같은 MPI 와 Trapezoidal Rule 을 활용하여 두 가지 주어진 함수의 적분값을 효과적으로 계산하고, 병렬 처리를 통한 성능 향상 효과를 검증하고자 합니다.

본 레포트에서는 Trapezoidal Rule 을 통한 수치 적분이 MPI 환경에서 어떻게 구현되는지를 설명하고, 실험을 통해 병렬 처리에 따른 실행 시간, 확장성, 그리고 계산의 정밀도 변화를 평가합니다. 이를 통해 시리얼 및 병렬 처리 방식이 적분 계산에 미치는 영향을 분석하고, 병렬 처리의 효율성을 논의할 것입니다.

1.1) Trapezoidal Rule 의 원리

Trapezoidal Rule은 연속 함수의 정적분을 수치적으로 계산하는 대표적인 방법으로, 주어진 구간을 작은 사다리꼴 형태의 구간으로 나눈 뒤 각 구간의 넓이를 합산하여 전체 적분값을 근사하는 방식입니다. 예를 들어, f(x)를 구간 [a,b]에서 n개의 균등한 소구간으로 나누면, 각 소구간의 너비 Δx = $\frac{b-a}{n}$ 로 계산됩니다. 이때 i번째 소구간의 넓이는 사다리꼴의 상단과 하단 길이, 즉 f(x_i)와 f(x_{i+1})의 평균에 너비 Δx를 곱하여 얻을 수 있습니다. 이를 모든 소구간에 대해 합산하면 다음과 같이 전체 적분값을 근사할 수 있습니다:

$$\int_a^b f(x)dx \approx \frac{\Delta x}{2}\sum_{i=1}^{n}(f(x_i)+f(x_{i+1}))$$

Trapezoidal Rule은 계산이 단순하고 비교적 높은 정확도를 제공하여 수치 해석의 다양한 분야에서 널리 사용됩니다. 그러나 구간을 세분화할수록 계산해야 할

사다리꼴의 수가 기하급수적으로 늘어나므로 연산량이 급격히 증가하게 됩니다. 이를 효율적으로 처리하기 위해 병렬 처리를 통해 각 프로세스가 특정 구간을 나누어 계산하는 방식이 필요하며, MPI와 같은 분산 병렬 처리 기법이 이를 가능하게 합니다.

1.2) Parallel proccesing 과 MPI

MPI(Message Passing Interface)는 분산 메모리 병렬 컴퓨팅 환경에서 여러 프로세스가 상호 통신하며 작업을 수행할 수 있도록 설계된 표준 인터페이스로, 과학 및 공학 계산 분야에서 널리 사용됩니다. MPI 는 각 프로세스가 독립적인 메모리 공간을 가지고 작업을 수행하며, 필요 시 메시지 passing 을 통해 데이터를 교환합니다. 이 인터페이스는 프로세스 생성, 데이터 전송, 동기화와 같은 주요 기능을 제공하여 분산 환경에서 다수의 프로세스가 협력하여 연산을 수행할 수 있도록 지원합니다.

1.3) 성능 분석에 사용된 지표

본 실험에서는 시리얼 및 병렬 계산 환경에서 Trapezoidal Rule 을 적용한 결과를 평가하기 위해 다음과 같은 성능 지표를 사용합니다:

속도 향상 (Speedup): 병렬 처리의 성능 향상 정도를 나타내는 지표로, 시리얼 실행 시간(Serial Execution Time,Ts) 과 병렬 실행 시간(Parallel Execution Time, Tp)의 비율로 정의됩니다. 여기서 Ts 는 모든 작업을 단일 프로세스에서 수행했을 때 걸리는 시간이며, Tp 는 병렬 처리를 통해 여러 프로세스가 협력하여 수행했을 때의 실행 시간입니다. Speedup 값이 1 보다 크면 병렬 처리가 시리얼 처리보다 빠르다는 것을 의미하며, 값이 클수록 병렬화의 효과가 큼을 나타냅니다. 이론적으로, 프로세스 수가 P 일 때 최대 Speedup 은 p 에 수렴할 수 있지만, 실제로는 통신 오버헤드, 부하 불균형 등으로 인해 이상적인 Speedup 에 도달하지 못하는 경우가 많습니다.

$$S\ (Speedup) = \frac{T_{serial}}{T_{parallel}}$$

✓ $T_{parallel} = \frac{T_{serial}}{p}$, $where\ p = number\ of\ cores;\ T_{serial} = Serial\ runtime;\ T_{parallel} = Prallel\ runtime$

효율성 (Efficiency): 병렬 계산이 주어진 프로세스 수에서 얼마나 효과적으로 수행되는지를 나타내는 지표로, Speedup 을 프로세스 수로 나눈 값으로 정의됩니다. 이를 통해 병렬 처리에 사용된 자원이 얼마나 효율적으로 활용되었는지를 평가할 수 있습니다. Efficiency 값이 1 에 가까울수록 각 프로세스가 병렬 작업에 기여한 정도가

최대화되었음을 나타내며, 효율적인 병렬화가 이루어졌음을 뜻합니다. Efficiency 값이 낮을수록 병렬 처리에 사용된 자원 대비 성능 향상이 낮음을 의미합니다. 이는 작업 부하가 충분히 크지 않거나, 통신 오버헤드 및 부하 불균형 등으로 인해 자원의 활용도가 떨어졌을 가능성을 시사합니다.

$$E\,(Efficiency) = \frac{S\,(Speedup)}{p} = \frac{\frac{T_{serial}}{T_{parallel}}}{p} = \frac{T_{serial}}{p \times T_{parallel}}$$

속도 향상과 효율성은 프로세스 수가 증가할수록 성능이 향상되는지 평가하는 중요한 지표로, 본 과제에서는 다양한 프로세스 수와 사다리꼴 개수에 대해 이들을 분석하여 확장성을 평가할 것입니다.

2.Design

2.1) MPI_Trapezoid.c

```c
#include <mpi.h>
#include <stdio.h>

// Function to integrate
double f(double x) {
    return x * x * x * x * x - 3 * x * x * x + x + 4;
}

// Compute local trapezoidal approximation
double Trap(double left_endpt, double right_endpt, int trap_count, double base_len) {
    double estimate, x;
    int i;

    estimate = (f(left_endpt) + f(right_endpt)) / 2.0;
    for (i = 1; i <= trap_count - 1; i++) {
        x = left_endpt + i * base_len;
        estimate += f(x);
    }
    estimate = estimate * base_len;

    return estimate;
}

int main(int argc, char** argv) {
    int my_rank, comm_sz, n = 1024;
    double a = 0.0, b = 1.0, h, local_a, local_b;
    int local_n;
    double local_int, total_int;
    double start, finish, elapsed;

    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

    // Start timing the entire program
    start = MPI_Wtime();

    h = (b - a) / n;  // width of each trapezoid
    local_n = n / comm_sz;  // number of trapezoids for each process

    // Each process calculates its local interval
    local_a = a + my_rank * local_n * h;
    local_b = local_a + local_n * h;
```

```
    // Calculate the local integral
    local_int = Trap(local_a, local_b, local_n, h);

    // Sum up the integrals from all processes using MPI_Reduce
    MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    // End timing the entire program
    finish = MPI_Wtime();
    elapsed = finish - start;

    // Output the result and total time
    if (my_rank == 0) {
        printf("With n = %d trapezoids, our estimate of the integral from %f to %f = %.15e\n", n, a, b, total_int);
        printf("Total elapsed time = %f seconds\n", elapsed);
    }

    MPI_Finalize();
    return 0;
}
```

이 코드는 MPI(Message Passing Interface)를 사용하여 병렬로 사다리꼴 적분법을 구현한 프로그램입니다. 프로그램의 구조는 다음과 같습니다:

헤더 파일에 <mpi.h>를 포함하여 필요한 MPI 라이브러리를 사용합니다. MPI_Init()을 호출하여 MPI 환경을 초기화하고, 각 프로세스의 고유 ID와 총 프로세스 수를 획득합니다. 적분할 구간과 함수 f(x)를 정의한 후, 전체 구간을 나누어 각 프로세스에 할당합니다.

각 프로세스는 할당된 구간에 대해 Trapezoidal Rule을 적용하여 적분을 수행한 뒤, 결과를 마스터 프로세스로 전달합니다. 마스터 프로세스는 수집된 결과를 바탕으로 최종 적분값을 계산하여 출력합니다. 마지막으로 MPI_Finalize()를 호출하여 MPI 환경을 종료합니다.

시리얼 실행에서는 단일 프로세스에서 전체 구간에 Trapezoidal Rule을 적용합니다. 이 경우 MPI를 사용하지 않고 각 구간의 면적을 합산하여 적분값을 계산합니다.

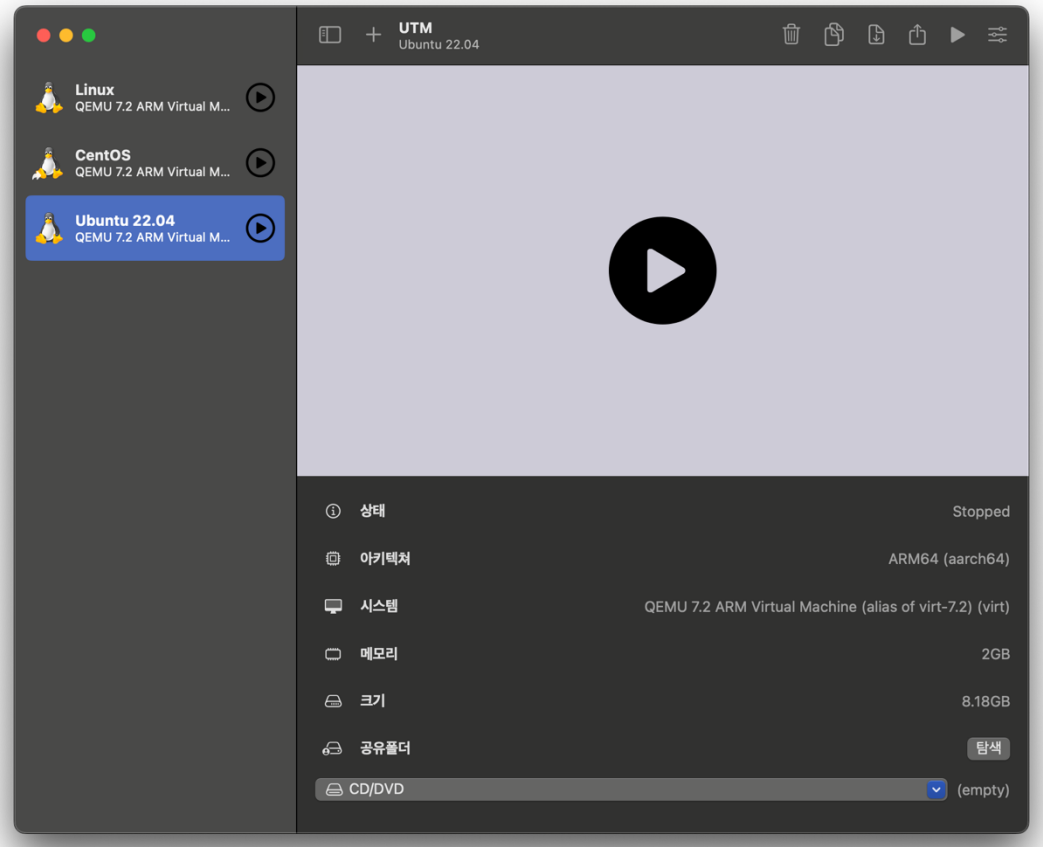
병렬 실행에서는 여러 프로세스가 할당된 구간에서 Trapezoidal Rule을 적용하여 각 구간의 적분값을 계산한 후, 이 결과를 마스터 프로세스에 전달합니다. 각 프로세스는 다음과 같은 단계를 통해 병렬로 작업을 수행합니다: 각 프로세스가 자신의 적분 구간을 나누고, 소구간별로 Trapezoidal Rule을 적용하여 해당 구간의 적분값을 계산한 후, MPI_Reduce() 또는 MPI_Gather()를 사용하여 결과를 마스터 프로세스로 전송합니다. 마스터 프로세스는 수집된 값을 집계하여 최종 결과를 도출합니다.

2.2) 시간 측정을 위한 MPI_Wtime()

MPI_Wtime() 함수는 MPI 프로그램의 실행 시간을 측정하는 데 사용됩니다. 코드의 특정 지점에서 호출되며, 호출 시점으로부터 경과된 벽 시계 시간을 반환합니다. 코드

내부에서는 시간 측정을 위해 실행의 시작과 끝에 MPI_Wtime()을 배치하여 전체 실행 시간을 측정합니다. double start_time = MPI_Wtime();는 코드가 시작될 때 호출하여 시작 시간을 기록합니다.
적분 계산이 완료된 후, double end_time = MPI_Wtime();를 통해 종료 시간을 기록합니다.
이후, 전체 실행 시간은 end_time - start_time으로 계산됩니다.
이러한 시간 측정 결과는 시리얼 및 병렬 실행 시간을 비교하는 데 활용되어 병렬화에 따른 성능 향상도를 평가할 수 있습니다.

2.3) 실행환경 구축



이번 과제는 MPI(Message Passing Interface)를 활용한 병렬 컴퓨팅과 관련된 작업을 포함하고 있어, 리눅스 기반의 환경이 필요했습니다. MacOS 자체는 Unix 기반으로 여러 개발 도구를 지원하지만, MPI와 같은 특정 병렬 컴퓨팅 라이브러리 설치와 환경 설정이 원활하지 않을 수 있어 Ubuntu 22.04를 실행 환경으로 선택했습니다.
Ubuntu는 MPI 및 병렬 프로세싱 환경을 위한 다양한 라이브러리를 쉽게 설치하고

사용할 수 있는 플랫폼으로, 안정적인 성능을 제공하기 때문에 적합한 선택이었습니다. MacOS 상에서 Ubuntu 22.04를 가상화한 이유는, 리눅스 기반의 개발 환경을 필요로 하는 MPI 과제를 효과적으로 수행하고 실험 결과를 안정적으로 기록하기 위함이었습니다. UTM을 활용하여 ubuntu 22.04 가상 환경 구축을 구축해 과제를 수행하였습니다.

3. Experiment and Results

3.1) Part 1 – Serial vs Parallel execution time comparison

Part1에선 사다리꼴 적분법을 사용하여 두 가지 함수를 적분할 때, Serial 및 Parallel 실행 간의 성능 차이를 Speedup과 Efficiency를 통해 비교 분석합니다. 아래 두 함수를 [0, 2] 구간에서 4096개의 사다리꼴을 사용하여 적분합니다

$$f1(x) = x^2$$
$$f2(x) = x^4 - 3x^2 + x + 4$$

MPI(Message Passing Interface)를 사용하여, 각 함수를 다음 두 가지 방식으로 실행하고 MPI_Wtime()를 통해 두 실행 방식에서의 실행 시간을 측정하여 병렬화에 따른 성능 변화를 평가하였습니다.

<실행결과>

serial

```
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpicc -g -Wall -o MPI_Trapezoid1_f1 MPI_Trapezoid1_f1.c
user@ubuntu-2204:~/Documents/Bigdata_processing$ ls
MPI_Trapezoid1_f1  MPI_Trapezoid1_f1.c  MPI_Trapezoid1_f2.c
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 1 ./MPI_Trapezoid1_f1
With n = 4096 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 2.666666746139526e+00
Total elapsed time = 0.000032 seconds
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpicc -g -Wall -o MPI_Trapezoid1_f2 MPI_Trapezoid1_f2.c
user@ubuntu-2204:~/Documents/Bigdata_processing$ ls
MPI_Trapezoid1_f1  MPI_Trapezoid1_f1.c  MPI_Trapezoid1_f2  MPI_Trapezoid1_f2.c
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 1 ./MPI_Trapezoid1_f2
With n = 4096 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 8.400000397364433e+00
Total elapsed time = 0.000034 seconds
```

parallel

```
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 4 ./MPI_Trapezoid1_f1
With n = 4096 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 2.666666746139526e+00
Total elapsed time = 0.001949 seconds
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 4 ./MPI_Trapezoid1_f2
With n = 4096 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 8.400000397364245e+00
Total elapsed time = 0.000284 seconds
```

<결과분석>

함수	프로세스 수	계산된 적분값	정확한 적분값	실행 시간(초)	S	E
f1(x)	1	2.66666674613952	2.6667	0.000032	0.01641868	0.00410467
	4	2.66666674613952	2.6667	0.001949		
f2(x)	1	8.40000039736443	8.4	0.000034	0.11971831	0.02992958
	4	8.40000039736424	8.4	0.000284		

$f1$(x) 에서 직렬 실행이 병렬 실행보다 훨씬 빨랐습니다. 직렬 실행 시간은 0.000032초였고, 병렬 실행 시간은 0.001949초로 측정되었습니다.
$f2$(x) 직렬 실행이 병렬 실행보다 여전히 빠르지만, 그 차이는 조금 적었습니다. 직렬 실행 시간은 0.000034초, 병렬 실행 시간은 0.000284초였습니다.

정확도 분석:

Serial 실행과 Parallel 실행 모두 동일한 적분값을 산출하여 정확도는 동일하였습니다. 두 함수의 계산된 적분값이 정확한 적분값에 매우 가깝게 근사되어 병렬화가 정확도에는 영향을 미치지 않음을 확인할 수 있었습니다.

Performance(Speedup 및 Efficiency) 분석:

Speedup은 Serial 실행 시간 대비 Parallel 실행 시간을 통해 성능 향상을 나타냅니다. 이 경우 Serial 실행이 더 빠르므로 $f1$(x), $f2$(x) 모두 Speedup 값이 1보다 크게 나오지 않았습니다. 또한, 이 실험에서 Efficiency는 $f1$(x) 에서 0.4%, $f2$(x) 에서 약 3%로, 두 함수 모두 매우 낮은 효율성을 보였습니다.

낮은 Speedup과 Efficiency는 이 작업의 낮은 계산 복잡도로 인해 병렬화의 장점이 병렬화 오버헤드에 의해 상쇄됨을 의미합니다. 병렬화 오버헤드가 프로세스 간 통신 비용과 데이터 분할 및 통합으로 인해 발생하였으며, 사다리꼴 개수가 비교적 적기 때문에, 4개의 프로세스 간 통신 및 조정 오버헤드로 인해 병렬 실행에서 오히려 더 많은 시간이 소요된 것으로 보입니다. 작업량이 작은 경우 Serial 실행이 더 효율적일 수 있음을 확인할 수 있습니다.

3.2) Part 2 – Scalability Test

Part2에서는 주어진 구간 [0,2]에서 두 함수에 대한 MPI 기반 사다리꼴 규칙 프로그램의 확장성과 성능을 분석합니다. 두 함수에 대해 프로세스 수(1, 2, 4, 8)와 사다리꼴 개수(256, 1024, 4096, 16384)를 변화시키며 속도 향상과 효율을 측정합니다. Part 2의 목표는 병렬 처리와 작업 부하가 증가함에 따라 프로그램이 얼마나 잘 확장되는지 관찰하는 것입니다.

<실행결과>

사다리꼴 n= 256일때

Process 1

```
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 1 ./MPI_Trapezoid1_f1
With n = 256 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 2.666687011718750e+00
Total elapsed time = 0.000009 seconds
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 1 ./MPI_Trapezoid1_f2
With n = 256 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 8.400101725012064e+00
Total elapsed time = 0.000003 seconds
```

Process 2

```
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 2 ./MPI_Trapezoid1_f1
With n = 256 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 2.666687011718750e+00
Total elapsed time = 0.000151 seconds
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 2 ./MPI_Trapezoid1_f2
With n = 256 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 8.400101725012064e+00
Total elapsed time = 0.000011 seconds
```

Process 4

```
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 4 ./MPI_Trapezoid1_f1
With n = 256 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 2.666687011718750e+00
Total elapsed time = 0.000012 seconds
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 4 ./MPI_Trapezoid1_f2
With n = 256 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 8.400101725012064e+00
Total elapsed time = 0.000591 seconds
```

Process 8

```
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 8 ./MPI_Trapezoid1_f1
With n = 256 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 2.666687011718750e+00
Total elapsed time = 0.001011 seconds
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 8 ./MPI_Trapezoid1_f2
With n = 256 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 8.400101725012064e+00
Total elapsed time = 0.013930 seconds
```

사다리꼴 n= 1024일때

Process 1

```
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 1 ./MPI_Trapezoid1_f1
With n = 1024 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 2.666667938232422e+00
Total elapsed time = 0.000013 seconds
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 1 ./MPI_Trapezoid1_f2
With n = 1024 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 8.400006357827806e+00
Total elapsed time = 0.000008 seconds
```

Process 2

```
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 2 ./MPI_Trapezoid1_f1
With n = 1024 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 2.666667938232422e+00
Total elapsed time = 0.000083 seconds
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 2 ./MPI_Trapezoid1_f2
With n = 1024 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 8.400006357827806e+00
Total elapsed time = 0.000014 seconds
```

Process 4

```
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 4 ./MPI_Trapezoid1_f1
With n = 1024 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 2.666667938232422e+00
Total elapsed time = 0.000019 seconds
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 4 ./MPI_Trapezoid1_f2
With n = 1024 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 8.400006357827806e+00
Total elapsed time = 0.000088 seconds
```

Process 8

```
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 8 ./MPI_Trapezoid1_f1
With n = 1024 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 2.666667938232422e+00
Total elapsed time = 0.000024 seconds
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 8 ./MPI_Trapezoid1_f2
With n = 1024 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 8.400006357827806e+00
Total elapsed time = 0.001612 seconds
```

사다리꼴 n= 4096일때

Process 1

```
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 1 ./MPI_Trapezoid1_f1
With n = 4096 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 2.666666746139526e+00
Total elapsed time = 0.000031 seconds
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 1 ./MPI_Trapezoid1_f2
With n = 4096 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 8.400000397364433e+00
Total elapsed time = 0.000025 seconds
```

Process 2

```
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 2 ./MPI_Trapezoid1_f1
With n = 4096 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 2.666666746139526e+00
Total elapsed time = 0.000059 seconds
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 2 ./MPI_Trapezoid1_f2
With n = 4096 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 8.400000397364259e+00
Total elapsed time = 0.000022 seconds
```

Process 4

```
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 4 ./MPI_Trapezoid1_f1
With n = 4096 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 2.666666746139526e+00
Total elapsed time = 0.000263 seconds
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 4 ./MPI_Trapezoid1_f2
With n = 4096 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 8.400000397364245e+00
Total elapsed time = 0.000021 seconds
```

Process 8

```
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 8 ./MPI_Trapezoid1_f1
With n = 4096 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 2.666666746139526e+00
Total elapsed time = 0.000626 seconds
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 8 ./MPI_Trapezoid1_f2
With n = 4096 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 8.400000397364252e+00
Total elapsed time = 0.000084 seconds
```

사다리꼴 n= 16384일때

Process 1

```
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 1 ./MPI_Trapezoid1_f1
With n = 16384 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 2.666666671633720e+00
Total elapsed time = 0.000120 seconds
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 1 ./MPI_Trapezoid1_f2
With n = 16384 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 8.400000024835263e+00
Total elapsed time = 0.000096 seconds
```

Process 2

```
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 2 ./MPI_Trapezoid1_f1
With n = 16384 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 2.666666671633720e+00
Total elapsed time = 0.000269 seconds
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 2 ./MPI_Trapezoid1_f2
With n = 16384 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 8.400000024835306e+00
Total elapsed time = 0.000059 seconds
```

Process 4

```
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 4 ./MPI_Trapezoid1_f1
With n = 16384 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 2.666666671633720e+00
Total elapsed time = 0.000176 seconds
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 4 ./MPI_Trapezoid1_f2
With n = 16384 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 8.400000024835307e+00
Total elapsed time = 0.000116 seconds
```

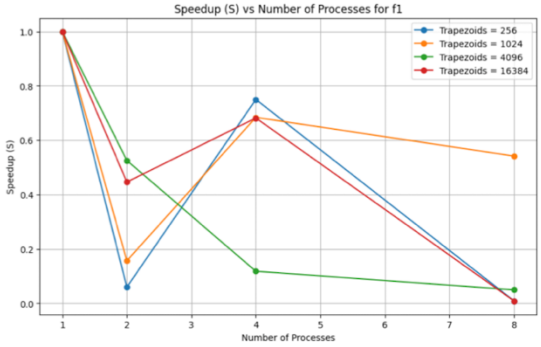
Process 8

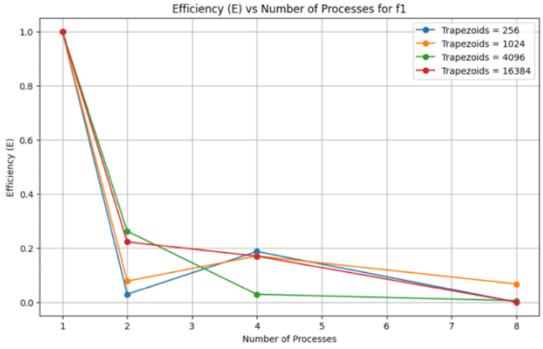
```
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 8 ./MPI_Trapezoid1_f1
With n = 16384 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 2.666666671633720e+00
Total elapsed time = 0.013807 seconds
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 8 ./MPI_Trapezoid1_f2
With n = 16384 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 8.400000024835277e+00
Total elapsed time = 0.000026 seconds
```

<결과분석>

	f1	P	1	2	4	8
0	256.0	S	1	0.059603	0.750000	0.008902
2	1024.0	S	1	0.156627	0.684211	0.541667
4	4096.0	S	1	0.525424	0.117871	0.049521
6	16384.0	S	1	0.446097	0.681818	0.008691

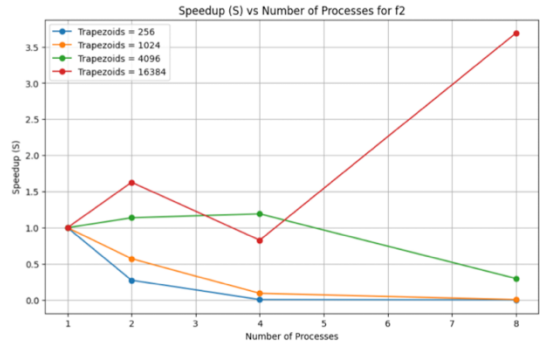
	f1	P	1	2	4	8
1	256	E	1	0.029801	0.187500	0.001113
3	1024	E	1	0.078313	0.171053	0.067708
5	4096	E	1	0.262712	0.029468	0.006190
7	16384	E	1	0.223048	0.170455	0.001086

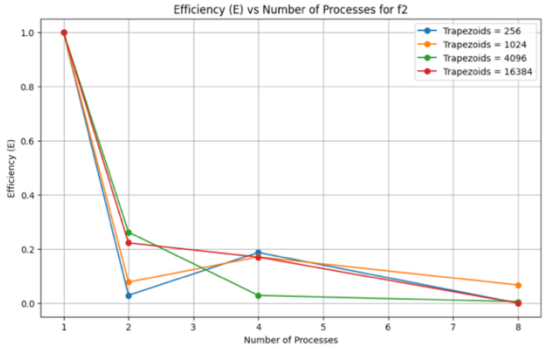




	f2	P	1	2	4	8
0	256.0	S	1	0.272727	0.005076	0.000215
2	1024.0	S	1	0.571429	0.090909	0.004963
4	4096.0	S	1	1.136364	1.190476	0.297619
6	16384.0	S	1	1.627119	0.827586	3.692308

	P	1	2	4	8	f2
1	E	1	0.029801	0.187500	0.001113	256
3	E	1	0.078313	0.171053	0.067708	1024
5	E	1	0.262712	0.029468	0.006190	4096
7	E	1	0.223048	0.170455	0.001086	16384





$\boldsymbol{f1(x) = x^2}$

N=256에서 프로세스 수가 증가함에 따라 초기에는 속도 향상이 나타나지만, 프로세스 수가 4와 8로 늘어날수록 성능이 급격히 감소합니다. 이는 작은 작업 부하에서 통신 오버헤드와 부하 불균형이 성능에 영향을 미치기 시작했음을 나타냅니다. N=1024의 경우, 2와 4 프로세스에서는 속도 향상이 있지만 8 프로세스에서는 약간의 성능 감소가 관찰됩니다. 이는 상대적으로 작은 작업 부하를 병렬 처리할 때 발생하는 수익 감소 때문일 수 있습니다. 반면, N=4096 및 16384의 경우 사다리꼴 수가 증가함에 따라 속도 향상과 효율이 더 안정적으로 유지됩니다. 이는 더 큰 작업 부하가 병렬화 오버헤드를 더 효과적으로 상쇄할 수 있기 때문입니다. 특히, N=4096에서 프로세스 수가 4개일 때 효율이 50% 이상 유지되는 경향을 보입니다. 전반적으로 속도 향상은 프로세스 수가 많아질수록 감소하는 경향이 있으며, 특히 적은 사다리꼴 수(256)에서는 병렬 처리의 비효율성이 두드러집니다. 효율은 프로세스 수가 증가할수록 꾸준히 감소하며, 이는 병렬 처리 성능의 저하를 강조하는 결과입니다.

$\boldsymbol{f2(x) = x^4 - 3x^2 + x + 4}$

이 함수는 더 높은 차수의 다항 함수로, 계산 비용이 더 크기 때문에 병렬화에 의한

성능 향상을 얻을 가능성이 높습니다. N=256에서 프로세스 수가 많아질수록 속도 향상과 효율이 크게 감소하는 경향이 있으며, 이는 함수의 복잡성으로 인해 작은 작업 부하에서 통신 오버헤드의 영향이 더욱 두드러지기 때문입니다. N=1024에서는 2와 4 프로세스에서 속도 향상이 조금 더 일관되지만, 8 프로세스로 이동하면 여전히 성능이 감소합니다. 이는 작업 부하에 비해 오버헤드가 커졌음을 나타냅니다. 반면, N=4096 및 16384의 경우 더 높은 사다리꼴 수에서는 속도 향상이 더 안정적이며, 특히 N=4096에서 효율이 높게 유지됩니다. N=16384에서는 예상보다 높은 속도 향상과 효율이 관찰되는데, 이는 시스템 최적화나 캐시 효과와 같은 요인에 기인할 수 있습니다.

속도 향상은 더 큰 workload에서 더 잘 확장되는 경향이 있으며, 이는 더 복잡한 함수일수록 다수의 프로세스를 활용하는 것이 효율적임을 나타냅니다. 효율은 큰 workload에서 상대적으로 높게 유지되지만, 작은 workload에서는 병렬 처리의 효과가 감소합니다. 이 함수는 복잡성 덕분에 병렬화의 이점을 더 많이 누리지만, 작은 workload에서는 여전히 통신 비용이 확장성을 저해합니다.

두 함수 모두 작은 workload에서 확장성에 한계가 있으며, 프로세스 수가 증가함에 따라 통신 오버헤드로 인해 병렬 처리의 장점이 감소합니다. 특히, 사다리꼴 개수가 많을수록 복잡한 함수에서 더 나은 속도 향상과 효율이 나타납니다. 반면, 적은 사다리꼴 개수에서는 프로세스 수가 증가할수록 성능 이점이 빠르게 사라집니다. 복잡한 함수는 병렬화로 인한 이점을 더 많이 누릴 수 있으며, 이는 더 많은 계산 workload가 통신과 동기화 오버헤드를 상쇄할 수 있기 때문입니다.
이 프로그램은 더 큰 workload에서 더 나은 확장성을 보여주며, 프로세스 수 증가에 따른 수익이 감소하는 경향이 있습니다. 따라서 병렬 효율을 극대화하기 위해서는 workload 크기와 프로세스 수 간의 균형이 중요합니다. 고성능 응용 프로그램에서는 MPI 기반 병렬화를 최대한 활용하기 위해 적절한 workload 분배와 프로세스 수 조정이 필요합니다.

3.3) Part 3 – Precision Test

프로그램을 2개의 process로 실행하여 사다리꼴 개수를 10, 40, 160, 640으로 증가시키며 각 함수의 적분 실행결과와 정확한 적분값을 비교하고, 그 절대 오차를 기록하여 사다리꼴 개수가 정밀도에 미치는 영향을 분석하였습니다.

함수: $\boldsymbol{f1(x) = x^2}$ over the interval [0,2]

$\boldsymbol{f2(x) = x^4 - 3x^2 + x + 4}$ over the interval [0,2]

Process 수: 2

exact integral values: f1(x): $\frac{8}{3} \approx 2.6667$, f2(x): $\frac{42}{5} = 8.4$

<실행결과>

n=10

```
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 2 ./MPI_Trapezoid1_f1
With n = 10 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 2.680000000000001e+00
Total elapsed time = 0.000050 seconds
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 2 ./MPI_Trapezoid1_f2
With n = 10 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 8.466560000000001e+00
Total elapsed time = 0.000046 seconds
```

n=40

```
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 2 ./MPI_Trapezoid1_f1
With n = 40 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 2.667500000000000e+00
Total elapsed time = 0.000042 seconds
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 2 ./MPI_Trapezoid1_f2
With n = 40 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 8.404166250000001e+00
Total elapsed time = 0.000011 seconds
```

n=160

```
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 2 ./MPI_Trapezoid1_f1
With n = 160 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 2.666718750000001e+00
Total elapsed time = 0.000044 seconds
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 2 ./MPI_Trapezoid1_f2
With n = 160 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 8.400260415039062e+00
Total elapsed time = 0.000011 seconds
```

n=640

```
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 2 ./MPI_Trapezoid1_f1
With n = 640 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 2.666669921875000e+00
Total elapsed time = 0.000042 seconds
user@ubuntu-2204:~/Documents/Bigdata_processing$ mpiexec -n 2 ./MPI_Trapezoid1_f2
With n = 640 trapezoids, our estimate of the integral from 0.000000 to 2.000000 = 8.400016276035309e+00
Total elapsed time = 0.000043 seconds
```

<결과 분석>

함수	사다리꼴 개수 (n)	계산된 적분값	정확한 적분값	절대 오차
f1(x)	10	2.68	2.6667	0.0133
	40	2.6675	2.6667	0.0008
	160	2.66671875	2.6667	0.00001875
	640	2.666669922	2.6667	0.00003008
f2(x)	10	8.46656	8.4	0.06656
	40	8.40416625	8.4	0.00416625
	160	8.400260415	8.4	0.000260415
	640	8.400016276	8.4	0.000016276

$\boldsymbol{f1(x) = x^2}$

사다리꼴 개수가 10으로 적을 때는 오차가 비교적 크게 나타났지만, n=40부터 정확한 값에 근접했습니다. 특히, n=160과 n=640에서 계산된 적분값이 정확한 값과 거의 동일하게 나왔으며, 오차가 거의 0.0000에 가깝게 줄어들었습니다. 하지만 n=640일 때 오차가 약간 증가하여 0.00003008이 되는 현상이 관찰됩니다. 이는 부동소수점 오차 누적 또는 연산 정확도 한계로 인한 자연스러운 변동일 수 있습니다. 전체적으로,

사다리꼴 개수 n이 증가함에 따라 계산된 적분값이 정확한 값에 가까워지며, 절대 오차도 급격히 줄어듭니다. 이 이유는 사다리꼴 개수가 많아질수록, 각각의 사다리꼴의 너비가 좁아지면서 함수의 곡선을 더 정확하게 포착할 수 있게 되므로 오차가 줄어들어 정밀도가 높아지는걸로 예상됩니다.

$\boldsymbol{f2(x) = x^4 - 3x^2 + x + 4}$

f1(x)보다 복잡한 다항식을 가지고 있어, 정확한 적분값에 도달하기 위해 더 많은 사다리꼴 개수가 필요할 것으로 예상했습니다. 결과를 보면, n=10일 때는 오차가 0.0666으로 다소 컸지만, n=40 이상부터는 점차 정확한 값에 근접했습니다. 특히, n=640에서 거의 정확한 적분값 8.4에 도달했으며, 오차가 0에 가깝게 줄어들었습니다. f1(x)의 경우와 마찬가지로, n이 증가할수록 추정값이 정확한 값에 가까워지고 절대 오차도 현저히 감소합니다. f2(x)는 f1(x)보다 높은 차수의 다항함수이기 때문에 사다리꼴 개수가 적을 때는 오차가 더 크게 나타났습니다. 그러나 n이 증가함에 따라 곡선의 형태를 더 잘 반영하여 추정값이 정확한 값에 근접하게 되고, 정밀도가 향상되었음을 관찰할 수 있습니다.

본 테스트를 통해 사다리꼴 적분법의 정밀도는 사다리꼴 개수가 증가할수록 향상되며, 적분의 정확도가 높아진다는 것을 확인할 수 있었습니다. 두 함수 모두 사다리꼴 개수가 증가할수록 계산된 적분값이 점차 정확한 값에 수렴하였으며, 오차가 감소했습니다. 단, 오차의 미세한 증가와 같은 현상은 부동소수점 연산 한계로 인한 작은 변동도 발생할 수 있습니다. 다항식의 차수가 높을수록, 초기 오차가 크게 발생하므로, 정밀한 결과를 위해 더 많은 사다리꼴 개수가 요구됩니다. 사다리꼴 규칙을 적용할 때 적절한 사다리꼴 수 선택이 중요하다는 점을 알 수 있었습니다.

4. Conclusion.

본 과제에서는 MPI를 활용한 병렬 컴퓨팅 환경에서 Trapezoidal Rule을 통해 두 함수의 적분을 계산하고, 성능 및 정밀도를 평가하였습니다.

실험 결과, 병렬화의 이점은 workload가 증가할수록 확연히 나타났으며, 이는 더 큰 구간 분할을 통한 연산량 증가가 병렬화 오버헤드를 상쇄할 수 있기 때문임을 확인했습니다. 특히 차수가 높은 다항함수 $\boldsymbol{f2(x) = x^4 - 3x^2 + x + 4}$는 단순한 함수인 $\boldsymbol{f1(x) = x^2}$에 비해 병렬 처리의 효율성을 더 많이 활용할 수 있었습니다. 하지만 작은 workload에서 병렬화는 오히려 비효율적인 결과를 초래할 수 있음을 확인하였습니다. 이는 프로세스 간의 통신 오버헤드와 작업 분할로 인한 부하 불균형 때문이며, 적은 수의 구간에서는 직렬 처리 방식이 더 효율적일 수 있음을 보여줍니다. 따라서 병렬 처리가 큰 이점을 발휘하려면, 문제의 계산 복잡도와 적절한 프로세스

수의 균형을 맞추는 것이 중요합니다.

정밀도 측면에서, 사다리꼴 개수를 증가시킴에 따라 계산된 적분값이 정확한 값에 더욱 가까워지는 경향을 보였으며, 이는 Trapezoidal Rule의 구간 세분화가 정확도 향상에 직접적인 영향을 미침을 확인할 수 있었습니다. 다만, 사다리꼴 개수가 지나치게 많아질 경우 부동소수점 오차로 인해 오차가 소폭 증가하는 현상도 나타났는데, 이는 수치 연산의 한계로 인한 자연스러운 현상으로 이해됩니다.

이번 과제를 통해 Trapezoidal Rule을 MPI 기반 병렬 처리 환경에서 활용할 경우, 문제의 특성과 병렬 환경을 고려한 최적의 작업 분할과 프로세스 처리방식의 선택의 중요성을 깨달았습니다. 지금껏 빅데이터 처리 수업에서 배웠던 내용들을 직접 실습하는 과정에서 과목에 대한 큰 흥미를 느낄 수 있었습니다. 또한, 병렬 처리의 효율성과 그에 따른 성능 차이를 직접 실험해보며, 앞으로의 데이터 처리 및 분석 작업에 있어서도 이를 활용할 수 있는 방법이 무엇일지 고민하게 되었습니다.