Problem description

The value of π can be estimated with the following power series:

$$\pi = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right] = 4\sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

The series version of code with n-term approximation could be written as follows:

```
double factor = 1.0;
double sum = 0.0;
for (k = 0; k < n; k++) {
   sum += factor/(2*k+1);
   factor = -factor;
}
pi_approx = 4.0*sum;
```

Issue 4 threads, and complete the estimation π of with n = 1000000.

Code and explanations

1. Estimates the value of π by using the parallel directive + critical directive

```
# include<stdio.h>
# include<stdlib.h>
# include<omp.h>

int main(int argc, char *argv[]){
	int num_of_thread = atoi(argv[1]);
	int num_of_k = atoi(argv[2]);
	double pi = 0.0;
	double start_time, end_time;

	start_time = omp_get_wtime();

	# pragma omp parallel num_threads(num_of_thread)
	{
		double partial = 0;
		for (int k = omp_get_thread_num(); k < num_of_k; k += num_of_thread){
			double factor = (k % 2 == 0)? 1.0: -1.0;
			partial += factor / (2 * k + 1);
		}
		printf("Processor %d out of %d sum: %f\n", omp_get_thread_num(), num_of_thread, partial);
		# pragma omp critical
		pi += 4.0 * partial;
	}

	end_time = omp_get_wtime();

	printf("Final estimated results with n=%d: %f\n", num_of_k, pi);
	printf("Author: B0928007 余明昌\n");
	printf("Execution time: %f seconds\n", end_time - start_time);

	return 0;
}
```

一開始先將 argv 的指定 thread 數量以及 k 轉換成 local variable，接著在平行運算部分，根據先前獲得的 k 當作 for 迴圈的範圍，thread 的數量當作各個 thread 所處理的部分(thread0 處理 k = 0 + thread 數量 * n, thread1 處理 k = 1 + thread 數量 * n, …)。各個 thread 計算完各自的 partial sum 過後，先展示出各 thread 的

結果，接著以 critical directive 將 partial sum 加總。最後將π的推估結果以及花費時間印出。

2. Estimate the value of π by using the parallel for directive + critical directive

```
# include<stdio.h>
# include<stdlib.h>
# include<omp.h>

int main(int argc, char *argv[]){
        int num_of_thread = atoi(argv[1]);
        int num_of_k = atoi(argv[2]);
        double pi = 0.0;
        double start_time, end_time;
        double partial_sum[num_of_thread];
        for (int i = 0; i < num_of_thread; i++){
                partial_sum[i] = 0;
        }

        start_time = omp_get_wtime();

        # pragma omp parallel for num_threads(num_of_thread)
                for (int k = 0; k < num_of_k; k++){
                        double factor = (k % 2 == 0)? 1.0: -1.0;
                        double partial = 4 * factor / (2 * k + 1);
                        partial_sum[omp_get_thread_num()] += partial;
                        # pragma omp critical
                        pi += partial;
                }

        end_time = omp_get_wtime();

        for (int i = 0; i < num_of_thread; i++){
                printf("Processor %d out of %d sum: %f\n", i, num_of_thread, partial_sum[i]);
        }

        printf("Fianl estimated results with n=%d: %f\n", num_of_k, pi);
        printf("Author: B0928007 余明昌\n");
        printf("Execution time: %f seconds\n", end_time - start_time);

        return 0;
}
```

一開始先將 argv 的指定 thread 數量以及 k 轉換成 local variable，接著在平行運算部分，根據先前獲得的 k 當作 for 迴圈的範圍，parallel for directive 會自己將資料分配給我所指定的 thread 數量去做平行運算。接著以 critical directive 將 partial sum 加總。最後先將各個 thread 的 partial sum 給印出，再將π的推估結果以及花費時間印出。

3. Remove the critical from 2. and see how it produces a wrong result

```c
# include<stdio.h>
# include<stdlib.h>
# include<omp.h>

int main(int argc, char *argv[]){
        int num_of_thread = atoi(argv[1]);
        int num_of_k = atoi(argv[2]);
        double pi = 0.0;
        double start_time, end_time;
        double partial_sum[num_of_thread];
        for (int i = 0; i < num_of_thread; i++){
                partial_sum[i] = 0;
        }

        start_time = omp_get_wtime();

        # pragma omp parallel for num_threads(num_of_thread)
                for (int k = 0; k < num_of_k; k++){
                        double factor = (k % 2 == 0)? 1.0: -1.0;
                        double partial = 4 * factor / (2 * k + 1);
                        partial_sum[omp_get_thread_num()] += partial;
                        pi += partial;
                }

        end_time = omp_get_wtime();

        for (int i = 0; i < num_of_thread; i++){
                printf("Processor %d out of %d sum: %f\n", i, num_of_thread, partial_sum[i]);
        }

        printf("Fianl estimated results with n=%d: %f\n", num_of_k, pi);
        printf("Author: B0928007 余明昌\n");
        printf("Execution time: %f seconds\n", end_time - start_time);

        return 0;
}
```

與 2.的差別在於，再加總 partial sum 時，去除了 critical directive，導致最終加總的結果發生錯誤。

4. Modify 3. to produce the correct result by using the reduction clause

```c
# include<stdio.h>
# include<stdlib.h>
# include<omp.h>

int main(int argc, char *argv[]){
        int num_of_thread = atoi(argv[1]);
        int num_of_k = atoi(argv[2]);
        double pi = 0.0;
        double start_time, end_time;
        double partial_sum[num_of_thread];
        for (int i = 0; i < num_of_thread; i++){
                partial_sum[i] = 0;
        }

        start_time = omp_get_wtime();

        # pragma omp parallel for reduction(+:pi) num_threads(num_of_thread)
                for (int k = 0; k < num_of_k; k++){
                        double factor = (k % 2 == 0)? 1.0: -1.0;
                        double partial = 4 * factor / (2 * k + 1);
                        partial_sum[omp_get_thread_num()] += partial;
                        pi += partial;
                }

        end_time = omp_get_wtime();

        for (int i = 0; i < num_of_thread; i++){
                printf("Processor %d out of %d sum: %f\n", i, num_of_thread, partial_sum[i]);
        }

        printf("Fianl estimated results with n=%d: %f\n", num_of_k, pi);
        printf("Author: B0928007 余明昌\n");
        printf("Execution time: %f seconds\n", end_time - start_time);

        return 0;
}
```

與 3.的差別在於，將 parallel for directive 加上了 reduction 並指定變數 pi 在做加總時避免多個 thread 同時修改導致錯誤產生。

Sampled outputs

1. Estimates the value of π by using the parallel directive + critical directive

```
Processor 1 out of 4 sum: -1.897902
Processor 0 out of 4 sum: 2.602213
Processor 2 out of 4 sum: 1.735241
Processor 3 out of 4 sum: -1.654154
Final estimated results with n=1000000: 3.141592
Author: B0928007 余明昌
Execution time: 0.002090 seconds
```

2. Estimate the value of π by using the parallel for directive + critical directive

```
Processor 0 out of 4 sum: 3.141589
Processor 1 out of 4 sum: 0.000002
Processor 2 out of 4 sum: 0.000001
Processor 3 out of 4 sum: 0.000000
Fianl estimated results with n=1000000: 3.141592
Author: B0928007 余明昌
Execution time: 0.043954 seconds
```

3. Remove the critical from 2. and see how it produces a wrong result

```
Processor 0 out of 4 sum: 3.141589
Processor 1 out of 4 sum: 0.000002
Processor 2 out of 4 sum: 0.000001
Processor 3 out of 4 sum: 0.000000
Fianl estimated results with n=1000000: -0.000104
Author: B0928007 余明昌
Execution time: 0.013968 seconds
```

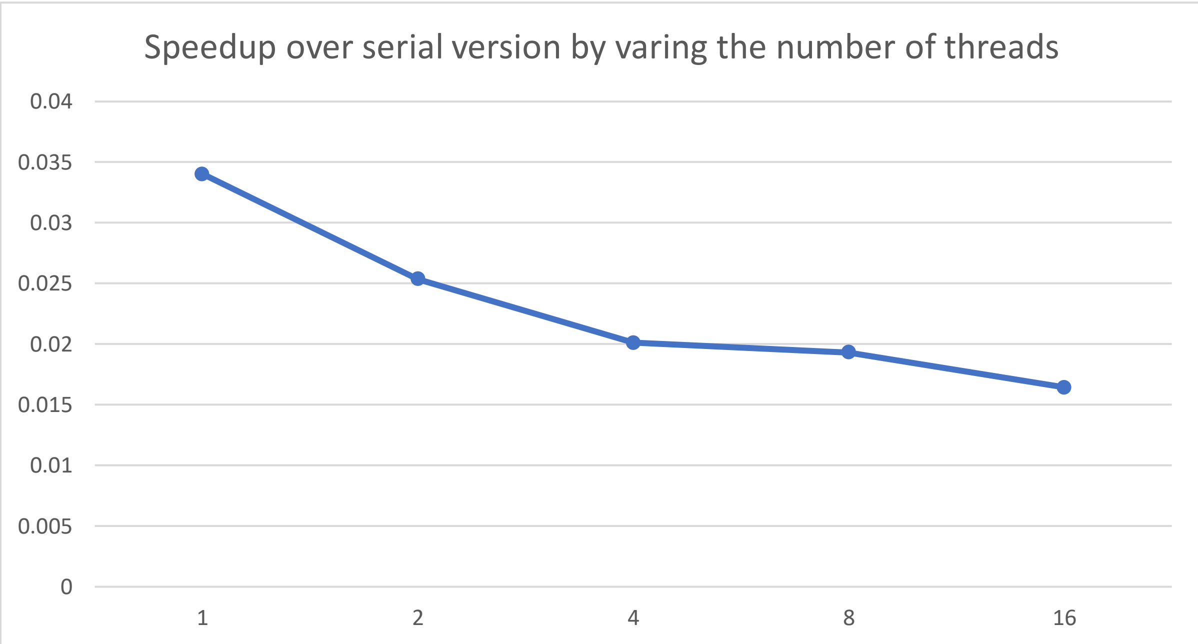
4. Modify 3. to produce the correct result by using the reduction clause

```
Processor 0 out of 4 sum: 3.141589
Processor 1 out of 4 sum: 0.000002
Processor 2 out of 4 sum: 0.000001
Processor 3 out of 4 sum: 0.000000
Fianl estimated results with n=1000000: 3.141592
Author: B0928007 余明昌
Execution time: 0.010667 seconds
```

Bonus:

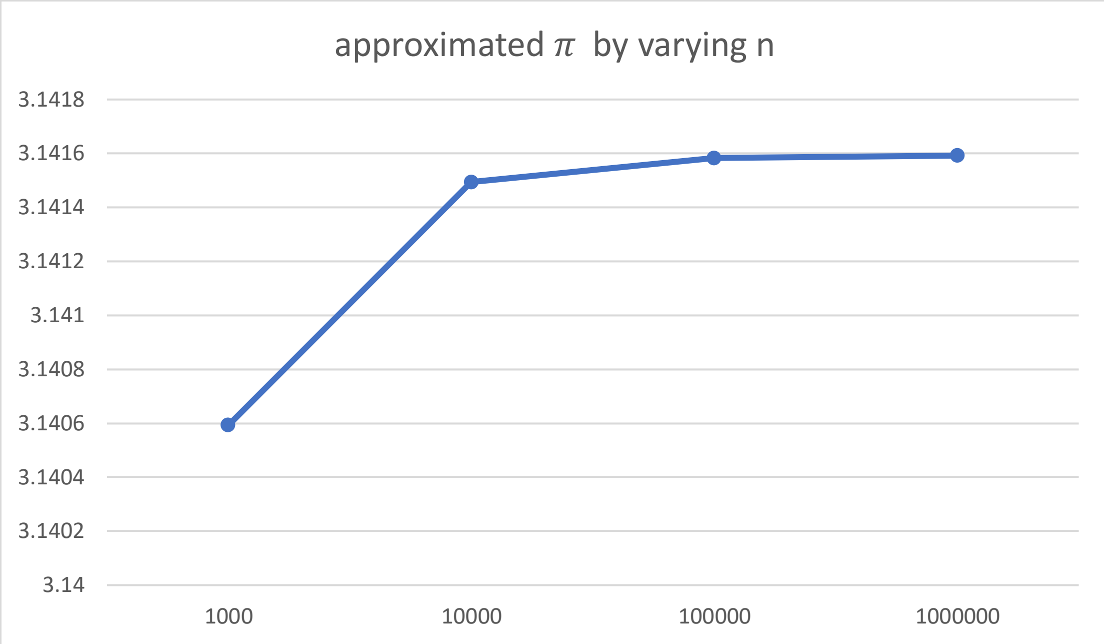
1. Recording the execution time from problem1 to problem 4
Display as the sampled outputs above.

2. Recording the speedup in problem 4 over the serial version by varying the number of threads = 1, 2, 4, 8, and 16



3. Recording the approximated π global sum of problem 4 by varying n = 1000,

10000, 100000, and 1000000



Discussions or what you have learned

1. parallel directive 可以造成平行運算但必須自己指定各個 thread 要做的事；parallel for directive 則可以自動分配給各個 thread，無需自己指定
2. 若沒有使用 critical directive 可能會導致各個 thread 在進行加總的時候，造成錯誤結果產生；加上 reduction 也可以避免上述情形產生
3. 第一種方法(parallel directive + critical directive)的速度最快，因為我們已經指定好各個 thread 要做的事情；第四種方法(parallel for directive + reduction)其次，因為需要自動分配各個 thread 做的工作，但以 reduction 避免加總錯誤產生比使用 critical directive 來得有效率，因為只指定+的運算元；第二種方法(parallel for directive + critical directive)最慢，因為要自動分配各個 thread 做的工作，並且以 critical directive 避免加總錯誤產生，增加了工作量
4. thread 的數量越多，所花費的總時間越少；k 的數值越高，π的推估準確度越高。