Spring 2022: Advanced Topics in Numerical Analysis: High Performance Computing Assignment 3 (due Apr. 4, 2022)

0. The codes can be founded at `https://github.com/zixiao18/HPC.git` For Windows users, you may type the following in the command window for running the codes. Commands may be different for Macs.

```
make
\fast-sin.exe
\omp-scan.exe
```

   All the codes are running on a processor `11th Gen Intel(R) Core(TM) i7-1195G7 @ 2.90GHz 2.92 GHz` with `RAM = 32.0GB (31.7GB available)`

1. **Approximating Special Functions Using Taylor Series & Vectorization.** Special functions like trigonometric functions can be expensive to evaluate on current processor architectures which are optimized for floating-point multiplications and additions. In this assignment, we will try to optimize evaluation of $\sin(x)$ for $x \in [-\pi/4, \pi/4]$ by replacing the builtin scalar function in C/C++ with a vectorized Taylor series approximation,

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!} + \cdots$$

   The source file `fast-sin.cpp` in the homework repository contains the following functions to evaluate $\{\sin(x_0), \sin(x_1), \sin(x_2), \sin(x_3)\}$ for different $x_0, \ldots, x_3$:

   - `sin4_reference()`: is implemented using the builtin C/C++ function.
   - `sin4_taylor()`: evaluates a truncated Taylor series expansion accurate to about 12-digits.
   - `sin4_intrin()`: evaluates only the first two terms in the Taylor series expansion (3-digit accuracy) and is vectorized using SSE and AVX intrinsics.
   - `sin4_vec()`: evaluates only the first two terms in the Taylor series expansion (3-digit accuracy) and is vectorized using the Vec class.

   Your task is to improve the accuracy to 12-digits for **any one** vectorized version by adding more terms to the Taylor series expansion. Depending on the instruction set supported by the processor you are using, you can choose to do this for either the SSE part of the function `sin4_intrin()` or the AVX part of the function `sin4_intrin()` or for the function `sin4_vec()`.

   Extra credit: develop an efficient way to evaluate the function outside of the interval $x \in [-\pi/4, \pi/4]$ using symmetries. Explain your idea in words and implement it for the function `sin4_taylor()` and for any one vectorized version. Hint: $e^{i\theta} = \cos\theta + i \sin\theta$ and $e^{i(\theta+\pi/2)} = ie^{i\theta}$.

We improve the accuracy to 12-digits for `sin4_taylor()` and `sin4_vec` by adding more terms to the Taylor series. We use first six terms (the same as `sin4_taylor`) instead of the first two to achieve 12-digits accuracy. You may take a look at `fast-sin.cpp` for details.

```
void sin4_intrin(double* sinx, const double* x) {
  // The definition of intrinsic functions can be found at:
  // https://software.intel.com/sites/landingpage/IntrinsicsGuide/#
#if defined(__AVX__)
  __m256d x1, x2, x3, x5, x7, x9, x11;
  x1  = _mm256_load_pd(x);
  x2  = _mm256_mul_pd(x1, x1);
  x3  = _mm256_mul_pd(x1, x2);
  x5  = _mm256_mul_pd(x3, x2);
  x7 = _mm256_mul_pd(x5, x2);
  x9 = _mm256_mul_pd(x7, x2);
  x11 = _mm256_mul_pd(x9, x2);

  __m256d s = x1;
  s = _mm256_add_pd(s, _mm256_mul_pd(x3 , _mm256_set1_pd(c3 )));
  s = _mm256_add_pd(s, _mm256_mul_pd(x5 , _mm256_set1_pd(c5 )));
  s = _mm256_add_pd(s, _mm256_mul_pd(x7 , _mm256_set1_pd(c7 )));
  s = _mm256_add_pd(s, _mm256_mul_pd(x9 , _mm256_set1_pd(c9 )));
  s = _mm256_add_pd(s, _mm256_mul_pd(x11 , _mm256_set1_pd(c11 )));
  _mm256_store_pd(sinx, s);
// The AVX part omitted
}

void sin4_vector(double* sinx, const double* x) {
  // The Vec class is defined in the file intrin-wrapper.h
  typedef Vec<double,4> Vec4;
  Vec4 x1, x2, x3, x5, x7, x9, x11;
  x1  = Vec4::LoadAligned(x);
  x2  = x1 * x1;
  x3  = x1 * x2;
  x5  = x3 * x2;
  x7  = x5 * x2;
  x9  = x7 * x2;
  x11 = x9 * x2;

  Vec4 s = x1;
  s += x3  * c3 ;
  s += x5  * c5 ;
  s += x7  * c7;
  s += x9  * c9;
  s += x11 * c11;
  s.StoreAligned(sinx);
}
```

Here are the results

```
PS C:\Users\zixiao\HPC\HPC-hw3> .\fast-sin.exe
Reference time: 0.0388
Taylor time:    1.4339      Error: 1.402833e+43
Intrin time:    0.0021      Error: 1.402833e+43
Vector time:    0.0020      Error: 1.402833e+43
```

The intrinic and vectorized methods are far faster than the Tylor implement. Both achieve 12-digits accuracy because they have the same error as the referrence method with 12-digits accuracy.

Extra Credits
Observe that $sin(x+\pi) = -sin(x)$. Hence, we can force x to be in the interval $[-\pi/4, 3\pi/4)$ by subtracting pi continuously.
If $x \in [-\pi/4, \pi/4]$, then we calculate x the same way we proceed before. If $x \in (\pi/4, 3pi/4)$, then we use the formula $sin(x+\pi/2) = -cos(x)$ to calculate for $cos(x)$ instead. We use the following formula to compute $cos(x)$ for 12-digits accuracy.

$$cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \frac{x^{10}}{10!} + \frac{x^{12}}{12!}$$

However, this part is not implemented successfully.

2. **Parallel Scan in OpenMP.** This is an example where the shared memory parallel version of an algorithm requires some thinking beyond parallelizing for-loops. We aim at parallelizing a scan-operation with OpenMP (a serial version is provided in the homework repo). Given a (long) vector/array $\boldsymbol{v} \in \mathbb{R}^n$, a scan outputs another vector/array $\boldsymbol{w} \in \mathbb{R}^n$ of the same size with entries
$$w_k = \sum_{i=1}^{k} v_i \text{ for } k = 1, \ldots, n.$$
To parallelize the scan operation with OpenMP using p threads, we split the vector into p parts $[v_{k(j)}, v_{k(j+1)-1}]$, $j = 1, \ldots, p$, where $k(1) = 1$ and $k(p+1) = n+1$ of (approximately) equal length. Now, each thread computes the scan locally and in parallel, neglecting the contributions from the other threads. Every but the first local scan thus computes results that are off by a constant, namely the sums obtains by all the threads with lower number. For instance, all the results obtained by the the r-th thread are off by
$$\sum_{i=1}^{k(r)-1} v_i = s_1 + \cdots + s_{r-1}$$
which can easily be computed as the sum of the partial sums $s_1, \ldots, s_{r-1}$ computed by threads with numbers smaller than r. This correction can be done in serial.

   - Parallelize the provided serial code. Run it with different thread numbers and report the architecture you run it on, the number of cores of the processor and the time it takes.

We set $p \in \{3, 4, 6, 8\}$ different threads for the project using the codes below.

```
void scan_omp(long* prefix_sum, const long* A, long n) {
  // TODO: implement multi-threaded OpenMP scan
  int n_thread = 3;
  long* partial_sum = (long*) malloc((n_thread-1) * sizeof (long));
  long split = n / n_thread;

  #pragma omp parallel for num_threads(n_thread)
  for (long thd=1; thd<n_thread; thd++){

     #pragma omp atom write
     prefix_sum[split*thd] = 0;
     partial_sum[thd] = 0;
  for (long j = split*thd+1; j < split*(thd+1); j++) {
   #pragma omp atom update
    prefix_sum[j] = prefix_sum[j-1] + A[j-1];
    partial_sum[thd] += A[j-1];
  }
  partial_sum[thd] += A[split*(thd+1)-1];
  }
  // add partial_sum back
  long to_add = 0;
  for (long thd=1; thd<n_thread; thd++){
   to_add += partial_sum[thd];
   for (long j=split*thd+1; j<split*(thd+1); j++){
   prefix_sum[j] += to_add;
  }
  }
}
```

Below is the results that we found.

Number of Threads	Time (s)	Error
1	0.081	2.147e+10
2	0.132	2.147e+10
4	0.173	2.147e+10
6	0.193	2.147e+10
8	0.199	2.147e+10