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

Handing in your homework: Hand in your homework as for the previous homework assignment (git repo with Makefile), answering the questions by adding a text or a LaTeX file into your repo. The git repository https://github.com/pehersto/HPCSpring2022 contains the code you can build on for this homework.

1. **Pitch your final project.** Summarize your current plan for the final project in a PDF document and send to me and Melody and Cai via email. We assume you have already talked to us about your project ideas when this homework is due and when you have sent the project description via email. Detail *what* you are planning to do, and with *whom* you will be cooperating. It is important that you call out 4-5 concrete tasks in your project description. We will request frequent updates during the rest of the semester on the progress you are making on these tasks.
   Solution:

   HPC Project Proposal, Spring 2022

   Project Description: Due to recent advancements in computer hardware, it is now possible to process graphs with billions of nodes and edges on a shared memory architecture. We will be implementing graph processing algorithms parallelly for such shared-memory multicore machines and GPUs and will analyze our results.

   The framework: Ligra proposed in the paper [1] works with applying function mapping routines on subsets of vertices or edges. These function mappings are essentially independent and can be parallelized to improve performance. The routines defined in Ligra are then used in graph algorithms to have performance gains. The bucketing technique proposed in the paper [2] extends the Ligra framework with its bucket data structure to extend the working of the routines defined in Ligra to also work on bucket-based graph algorithms.

   Using these techniques and framework we want to parallelize various graph algorithms for multi-core CPU and GPU. We would like to observe the improvement in performance compared to serial implementation and also want to analyze how the parallel graph algorithms fare on different performance units.(CPU vs GPU).

   Relevant Details: Tasks Milestones We will be implementing the graph processing algorithms discussed in frameworks: Ligra for CPU and GPU. The framework contains various functions (VertexMap, EdgeMap, EdgeMapReduce) which are parallelizable.

   Implementation of the Work Efficient Bucketing techniques discussed in the framework: Julienne

   We will be using the above-described techniques to parallelly implement various graph algorithms like Breadth-First Search, Bellman-Ford Shortest Path (SSSP), Connected Components, etc.

We will be testing our implementation with publicly available social networks and other graph datasets.

We will be analyzing the running time, speedups, bandwidth and scaling of the generated results and comparison with the available results.

3. Tools of HPC to be used: OpenMP CUDA C++ Valgrind and XCode Profiling Tools

4. Collaborators: Shuvadeep Saha (ss15592) Utkarsh Khandelwal (uk2051) Yajur Ahuja (ya2109)

5. References: L. Dhulipala, G. Blelloch and J. Shun, Julienne: A Framework for Parallel Graph Algorithms using Work-efficient Bucketing, SPAA '17: Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures, July 2017 Pages 293–304. https://people.csail.mit.edu/jshun/bucketing.pdf

J. Shun and G. Blelloch: Ligra: A Lightweight Graph Processing Framework for Shared Memory, PPoPP'13, February 23–27, 2013, Shenzhen. https://www.cs.cmu.edu/ jshun/ligra.pdf

2. **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,

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

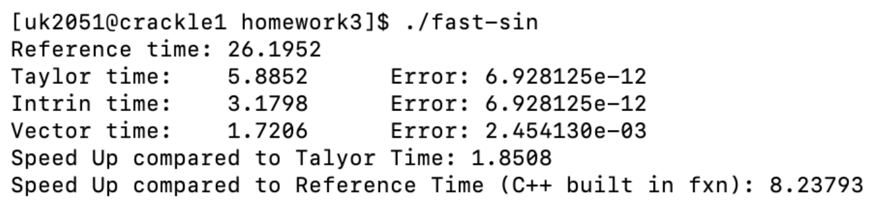
The source file `fast-sin.cpp` in the homework repository contains the following functions to evaluate $\{\sin(x_0), \sin(x_1), \sin(x_2), \sin(x_3)\}$ for different $x_0, \ldots, x_3$:

- `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()`.
Solution:

Implmented it for SSE part. Tested on following Courant Server:
Server Name: crackle1.cims.nyu.edu

Architecture: Two Intel Xeon E5630 (2.53 GHz) (16 cores) with 64 GB Memory and CentOS 7



```
[uk2051@crackle1 homework3]$ ./fast-sin
Reference time: 26.1952
Taylor time:    5.8852      Error: 6.928125e-12
Intrin time:    3.1798      Error: 6.928125e-12
Vector time:    1.7206      Error: 2.454130e-03
Speed Up compared to Talyor Time: 1.8508
Speed Up compared to Reference Time (C++ built in fxn): 8.23793
```

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 $e^{i(\theta+\pi/2)} = ie^{i\theta}$.
Solution:

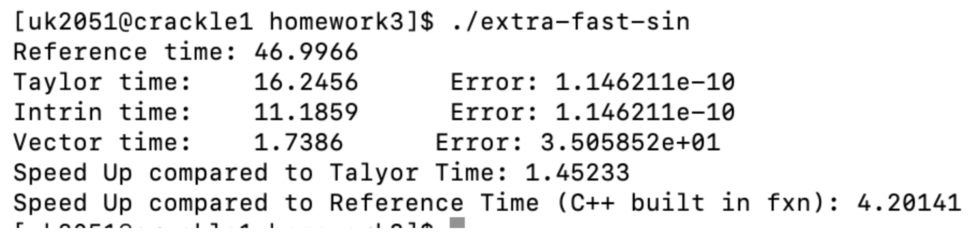
Implemented it for any value of $x \in [-2\pi, 2\pi]$ in the file "extra-fast-sin.cpp"
Approach Used:
Step 1. Checked the value of x. If it is positive used it directly, otherwise multiplied it by negative 1 and stored the factor used for multiplication. Because $\sin(-x) = -\sin(x)$ and we can multiply with -1 in the end.
Step 2. Now, we will be having only positive value of x. Use the fact that $e^{ix} = e^{i(\alpha+t\pi/2)} = i^t e^{i(\alpha)}$. Here t will be an integer. There are only four possible cases.

$$\cos(x) + i\sin(x) = e^{ix} = e^{i(\alpha+t\pi/2)} = i^t e^{i(\alpha)} = \begin{cases} e^{i(\alpha)} = \cos(\alpha) + i\sin(\alpha) \\ ie^{i(\alpha)} = i\cos(\alpha) - \sin(\alpha) \\ -e^{i(\alpha)} = -\cos(\alpha) - i\sin(\alpha) \\ -ie^{i(\alpha)} = -i\cos(\alpha) + \sin(\alpha) \end{cases}$$

Comparing the imaginary part, we will get

$$\sin(x) = \begin{cases} \sin(\alpha) \\ \cos(\alpha) \\ -\sin(\alpha) \\ -\cos(\alpha) \end{cases}$$

Step 3. At the end multiply it with the multiplication factor introduced in step 1.
Following is the error and speed up screenshot.



```
[uk2051@crackle1 homework3]$ ./extra-fast-sin
Reference time: 46.9966
Taylor time:    16.2456      Error: 1.146211e-10
Intrin time:    11.1859      Error: 1.146211e-10
Vector time:    1.7386      Error: 3.505852e+01
Speed Up compared to Talyor Time: 1.45233
Speed Up compared to Reference Time (C++ built in fxn): 4.20141
```

3. **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

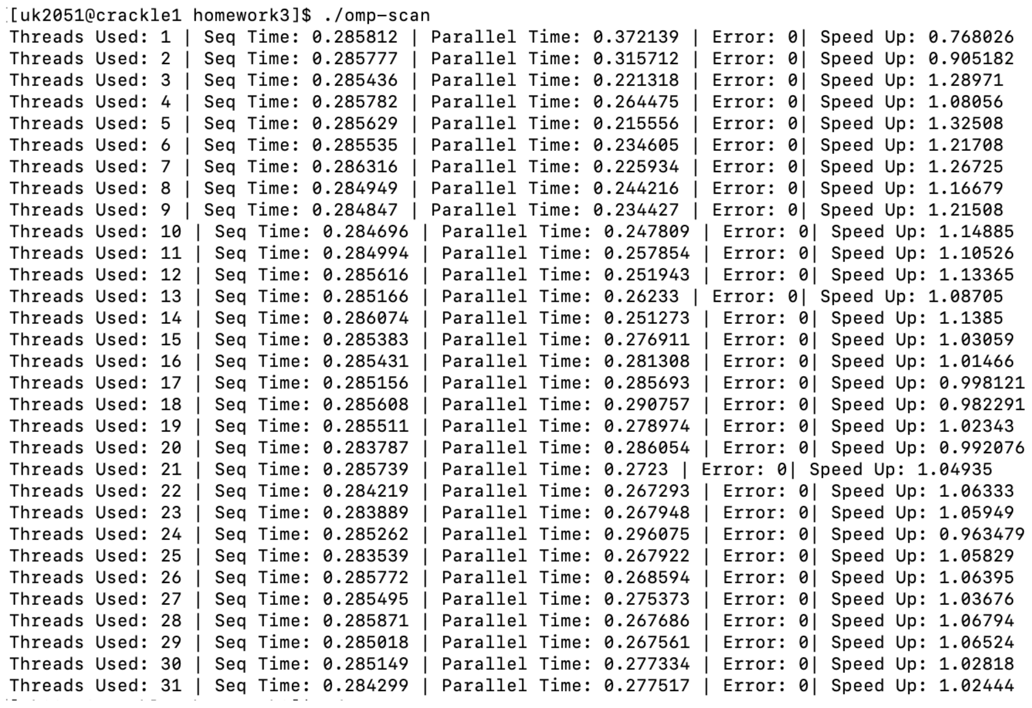
$$w_k = \sum_{i=1}^{k} v_i \text{ for } k = 1, \ldots, n.$$

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.
  Solution:
  Implemented using the given algorithm and ran it on the following machine.
  Server Name: crackle1.cims.nyu.edu
  Architecture: Two Intel Xeon E5630 (2.53 GHz) (16 cores) with 64 GB Memory and CentOS 7 Following is the screenshot showing time taken on different threads.



```
[uk2051@crackle1 homework3]$ ./omp-scan
Threads Used: 1 | Seq Time: 0.285812 | Parallel Time: 0.372139 | Error: 0| Speed Up: 0.768026
Threads Used: 2 | Seq Time: 0.285777 | Parallel Time: 0.315712 | Error: 0| Speed Up: 0.905182
Threads Used: 3 | Seq Time: 0.285436 | Parallel Time: 0.221318 | Error: 0| Speed Up: 1.28971
Threads Used: 4 | Seq Time: 0.285782 | Parallel Time: 0.264475 | Error: 0| Speed Up: 1.08056
Threads Used: 5 | Seq Time: 0.285629 | Parallel Time: 0.215556 | Error: 0| Speed Up: 1.32508
Threads Used: 6 | Seq Time: 0.285535 | Parallel Time: 0.234605 | Error: 0| Speed Up: 1.21708
Threads Used: 7 | Seq Time: 0.286316 | Parallel Time: 0.225934 | Error: 0| Speed Up: 1.26725
Threads Used: 8 | Seq Time: 0.284949 | Parallel Time: 0.244216 | Error: 0| Speed Up: 1.16679
Threads Used: 9 | Seq Time: 0.284847 | Parallel Time: 0.234427 | Error: 0| Speed Up: 1.21508
Threads Used: 10 | Seq Time: 0.284696 | Parallel Time: 0.247809 | Error: 0| Speed Up: 1.14885
Threads Used: 11 | Seq Time: 0.284994 | Parallel Time: 0.257854 | Error: 0| Speed Up: 1.10526
Threads Used: 12 | Seq Time: 0.285616 | Parallel Time: 0.251943 | Error: 0| Speed Up: 1.13365
Threads Used: 13 | Seq Time: 0.285166 | Parallel Time: 0.26233 | Error: 0| Speed Up: 1.08705
Threads Used: 14 | Seq Time: 0.286074 | Parallel Time: 0.251273 | Error: 0| Speed Up: 1.1385
Threads Used: 15 | Seq Time: 0.285383 | Parallel Time: 0.276911 | Error: 0| Speed Up: 1.03059
Threads Used: 16 | Seq Time: 0.285431 | Parallel Time: 0.281308 | Error: 0| Speed Up: 1.01466
Threads Used: 17 | Seq Time: 0.285156 | Parallel Time: 0.285693 | Error: 0| Speed Up: 0.998121
Threads Used: 18 | Seq Time: 0.285608 | Parallel Time: 0.290757 | Error: 0| Speed Up: 0.982291
Threads Used: 19 | Seq Time: 0.285511 | Parallel Time: 0.278974 | Error: 0| Speed Up: 1.02343
Threads Used: 20 | Seq Time: 0.283787 | Parallel Time: 0.286054 | Error: 0| Speed Up: 0.992076
Threads Used: 21 | Seq Time: 0.285739 | Parallel Time: 0.2723 | Error: 0| Speed Up: 1.04935
Threads Used: 22 | Seq Time: 0.284219 | Parallel Time: 0.267293 | Error: 0| Speed Up: 1.06333
Threads Used: 23 | Seq Time: 0.283889 | Parallel Time: 0.267948 | Error: 0| Speed Up: 1.05949
Threads Used: 24 | Seq Time: 0.285262 | Parallel Time: 0.296075 | Error: 0| Speed Up: 0.963479
Threads Used: 25 | Seq Time: 0.283539 | Parallel Time: 0.267922 | Error: 0| Speed Up: 1.05829
Threads Used: 26 | Seq Time: 0.285772 | Parallel Time: 0.268594 | Error: 0| Speed Up: 1.06395
Threads Used: 27 | Seq Time: 0.285495 | Parallel Time: 0.275373 | Error: 0| Speed Up: 1.03676
Threads Used: 28 | Seq Time: 0.285871 | Parallel Time: 0.267686 | Error: 0| Speed Up: 1.06794
Threads Used: 29 | Seq Time: 0.285018 | Parallel Time: 0.267561 | Error: 0| Speed Up: 1.06524
Threads Used: 30 | Seq Time: 0.285149 | Parallel Time: 0.277334 | Error: 0| Speed Up: 1.02818
Threads Used: 31 | Seq Time: 0.284299 | Parallel Time: 0.277517 | Error: 0| Speed Up: 1.02444
```