Differences:

When calculating f(x) for 2300, the float implementation overflows resulting in NAN. The double implementation is still capable of handling the large numbers required to do 2300^10 and does not overflow resulting in an answer. Its interesting to note that after 10 iterations, the Taylor series expansion is still very far off for the values of the sin(2300) (resulting in answers on the order of e55).

Representation in IEEE 754 Format

```
Float: s e_8 m_23
Float: s e_11 m_52

1:

Float:
- s: 0
- e: 127
- m: 0

Double:
- s: 0
- e: 1023
- m: 0

2300:

Float:
- s: 0
- e: 138
- m: 1032192 (0001111110...0) (rpt. 0 until 52 bits)

Double:
- s: 0
- e: 1034
- m: 554153860399104 (0001111110...0) (rpt. 0 until 52 bits)

-.45:

Float:
- s: 1
- e: -2 -> 127 + -2 = 125
- m: 1100...1100 = 6710886 (rpt. 1100 until 23 bits)
```

```
Double:
- s: 1
- e: -2 -> 1023 + -2 = 1021
- m: (3602879701896396) = 1100...1100 (rpt. 1100 until 52 bits)
```

Comparision

Converting the dining philsopher's solution to OpenMP had both difficulties and benefits. In OpenMP we lose some control over some of the explicit behvaior and controls that we had while using pthreads at the benefit of having an easier time spawning threads. In this case, threads were much easier to spawn (didn't need to jump through hoops with static functions to set up and launch tasks for to create and launch threads, didn't need to explicitly create barriers, etc). OpenMP also made some funcitonalities easier. Because of the explicit pragmas that affect all of the OpenMP calls, we could synchronize threads without passing objects through to threads. This meant we didn't need to create explicit barriers and locks acround certain regions since we could just use `omp barrier` and `omp critical`. Furthermore, we could change the number of threads that would be launched using a simple `omp_set_num_threads()`. The difficult part about using OpenMP is that all threads now need to run the same task.

Performance

- OMP: 9002 ms, 5 philosophers, 2 eats per
- Pthread: 9003 ms, 5 philosophers, 2 eats per

I did not see any noticeable performance difference when using OMP vs pthreads. My OpenMP run ran in 9002 ms while the pthread instance ran slightly longer at 9003 ms. These measurments include the creation and destruction of the threads. This may be due to my implementation using very minimal creation and destruction of threads (single creation point, single destruction point). The work done in both implementations are very similar, in some cases, swapping out a `pthread_mutex_t` for an `omp_lock_t`. This shows that for accessing simple parallelism, OpenMP can give large returns for little less devleopment cost than pthreads.

Summary: The Impact of Taskyield on the Design of Tasks Communicating Through MPI

`taskyield`: The taskyield construct specifies that the current task can be suspended in favor of execution of a different task. The benefit of this is the ability to hide latencies that may occur during the execution of a task.

Tasks allow OpenMP programs to be structure in such a way that one task can handle the MPI communication while other tasks handle data parallelism.

The paper can be summarized in three main components.

1. Explanation of different `taskyield` paradigms and how they can be used within MPI ranks to facilitate communicaiton. Also a explains how the different paradigmns can affect CORRECTNESS of a program!
2. Explain a series of blackbox tests that the researchers did to determine the `taskyield` paradigm being used by different implementations of OpenMP.
3. Performance Benchmarks of the Cholesky factorization on matrices using different communication/taskyield protocols. Results show that a correct hybridized implementation of OpenMP and MPI using OpenMP tasks and `taskyield`'s can have significant boosts in performance. However, being able to properly optimize the tasks and calls depends on a deep knowledge of the implementation. Furthermore the authors expose an issue and potential fix: without knowing the OpenMP implemenation being used (and by extension, the `taskyield` implementation) a program cannot know how to best handle the communication tasks. Certain implementations will even result in INCORRECT programs which deadlock due to the nature of the `taskyield` implementation. Further optimizations can be made depending on the result of the task yield implementation. The resulting fix for this problem is to expose some method in OpenMP that allows users to query the `taskyield` policy enable better optimizations.