Task1 Solution

The computation is based on a numerical integration. It starts with the trigonometric statement $\tan\left(\frac{\pi}{4}\right) = 1$ from this we attempt to make the mathematical solution discrete.
The sum we reach enables us to move forward with the computation.

Our provided serial solution performs instructions one after the other. The goal is to parallelize such a solution in a way that results in faster execution time.

If one part of the code takes a long time to run the rest of the code will not run until that part is finished. We can make our solution much faster through parallelization if we run different parts simultaneously. This approach exposed the possible parallel regions so that we can exploit them to run faster.

First, we start by dividing the "for" loop into subtasks and distribute them to worker processes. In other words, we need to evenly distribute the task of evaluating the integral resulting sum at a number of points.

This can be conveniently achieved by first providing the start, stop and step arguments to an inner function called here `pi_comp( )`.

1st integer is the number of integration sequence start
2nd integer is the end of the sequence
3rd integer is the step between adjacent elements and is set as the number of processes to avoid duplicate counting.

We can still further continue with this approach to maximize parallelism and minimize communication between tasks by exploring a recursive approach.

This approach can be achieved by adopting a design paradigm by which we recursively break down a problem into subproblems of the same related type.

This method is called Divide & Conquer. It starts with the big picture and then breaks it down to smaller segments. To use such an approach in parallel with OpenMP, we need to use task constructs.

The OpenMP tasking model was proposed to allow users to exploit the parallelism of irregular and dynamic problem structures such as unbounded loops, recursive algorithms and producer–consumer patterns.

In the proposed parallel solution first, a thread encounters the `parallel` directive and then creates a team of threads based on the fork-join model. The `single` directive ensures that only one thread in the team can enter the single construct. The other threads in the team will become work threads which are possible candidates for the execution of the generated tasks.

In addition to that when a task may be temporarily suspended when a thread encounters a task scheduling point. This can be explicitly set by the `barrier`, `taskyield` or in this case the `taskwait` directive.

Task 1 Results

	1	6	12	24	48
24 000 000	0.084	0.041	0.041	0.041	0.045
48 000 000	0.159	0.065	0.047	0.041	0.056
96 000 000	0.310	0.115	0.074	0.048	0.055

Table 1: Execution times for proposed problem sizes and thread number

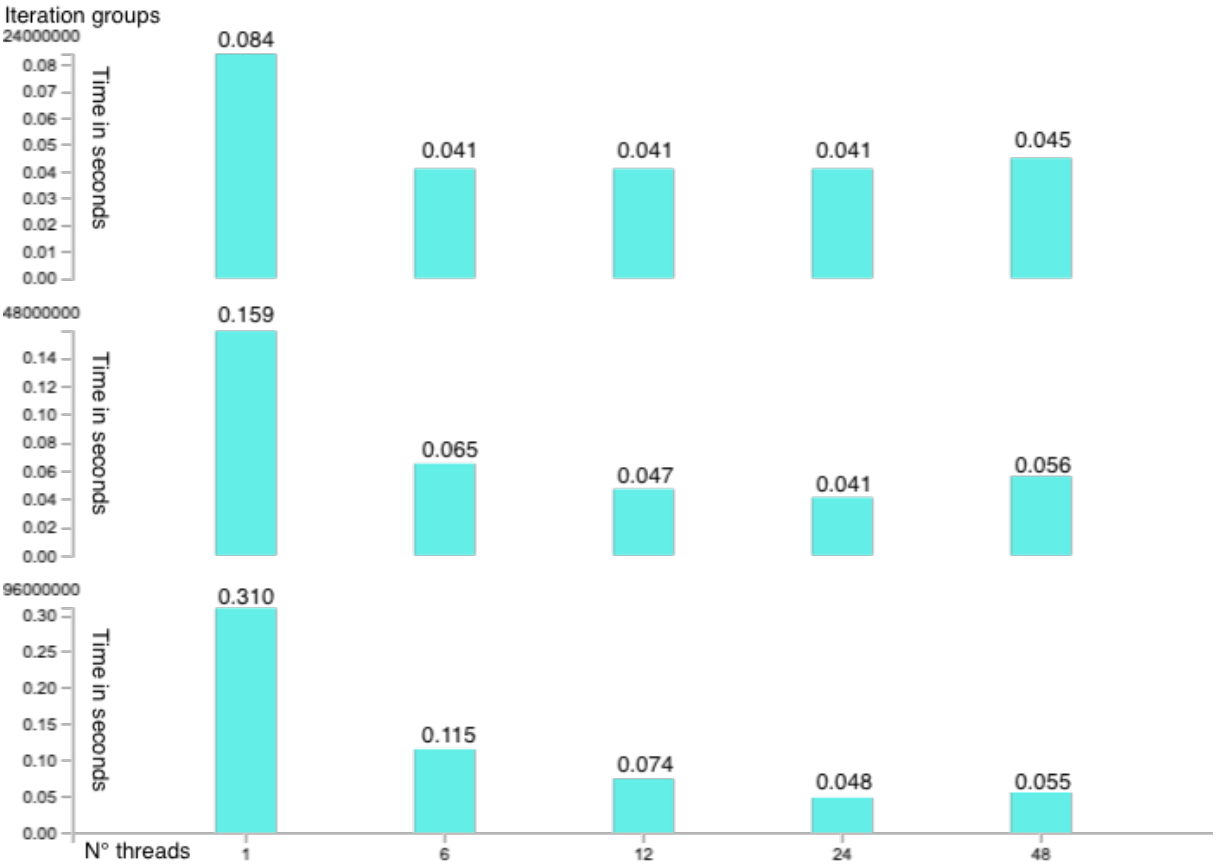

Figure 1: Execution times for proposed problem sizes and thread number in contrast

Instructions for running and testing the program

I have used the GCC compiler previously to compile the code on bash terminal execute:

Cd hmidi/project/
. /divideConquer