OpenMP Programming Assignment Report

This project is to parallel the giving sequential code using OpenMP directives. Performance of the parallel code with different threads will be tested and compared.

- ## Directives used

  To create the parallel code based on the sequential code, OpenMP directives should be added. Those directives operate on the loop. If the loop could be parallelized, directive related to parallelize "for" should be added before the "for" statement as follows:

```
#pragma omp parallel for private(<private var>)
for…
```

  Or

```
#pragma omp parallel private(<private var>){
……
#pragma omp for
for(…)
}
```

  If thread-related variables are used in the loop, we should use the second way to parallel the loop and related statements. Statements including variables like thread ID also should be contained since every thread should execute these statements. Otherwise, either way is fine for loop parallelization.

  Here are some rules for whether one loop would be parallelized or not.

  (1) If number of iterations is small, that is smaller than number of threads, that loop would not be parallelized since it might not cause performance improvement but more overhead.
  (2) If the body of loop contains memory allocation or input/output operations, that loop would not be vectorized either.
  (3) If variables written in the loop have dependencies between iterations, then the loop cannot be vectorized. (That means value of that variable in the current iteration depends on the previous iteration)
  (4) If variables are independent, some variables should be private to every thread when executed in parallel. Otherwise, result might be wrong.

  Here are some ways to figure out whether variables in the loop should be private or shared:

  (1) If variable is an element in the array, and index is related to iteration variable, this variable might be shared
  (2) Iteration variables usually are private.
  (3) In the nested loop, its iteration variable might be private and array elements whose index only related to nested loop's iteration variable are private.

(4) Scalar variables written in the loop might be private since in every iteration that variable would be used and written.

Based on rules above, all possible loops could be parallelized using parallel for directives.

- Performance

Table 1 listed below is the performance of different threads from 1 to 16. Table 2 lists partial results from **`grof`** analysis for sequential code (execution time for unlisted functions are 0.00). Since the parallel code focuses on those functions costing more time which are **`smvp, main, element_matrices`** and **`mem_init`**. Therefore, those not parallelized code takes about 2.57 seconds. If every statements in this three functions are parallelized, the limit execution time for those functions are about 64.40/16 = 4.03 seconds. Therefore, expected performance for 16 threads is a little more than about 6.6 seconds since not every statements are in the loop and not every loop could be parallelized and overhead for parallelizing should be considered as well.

According to the result from Table1, performance of 16 threads is 6.86s, a little slower than the expected result 6.6s. This is because there exists serial code and loops with dependencies which cannot be parallelized. Additionally, those iteration number less than 16 have overhead which causes more time than serial code. And all loops with **`malloc`** and **`fprintf`** are not vectorized. Therefore, the final results are a little longer than expected.

Table1: Program with multi-thread execution time

# of threads	Execution time (s)
1	70.781450
2	40.656607
4	21.912493
8	12.133388
16	6.863577

Table2: Analysis for functions for serial code

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
64.05	42.96	42.96	3855	11.14	11.15	smvp
31.96	64.40	21.44				main
1.30	65.27	0.88	348904485	0.00	0.00	phi1
0.97	65.92	0.65	348904485	0.00	0.00	phi0
0.92	66.54	0.62	348904485	0.00	0.00	phi2
0.42	66.82	0.28	151173	0.00	0.00	element_matrices
0.10	66.89	0.07	1	70.02	70.02	slip
0.04	66.92	0.03	4162	0.01	0.01	abe_matrix
0.03	66.94	0.02	3857	0.01	0.01	mem_init
0.03	66.96	0.02	1	20.01	20.01	readpackfile
0.01	66.97	0.01	151173	0.00	0.00	inv_J