CPSC 524 Assignment 2

NetID: yl2335

Name: Yangxiaokang Liu

Environment

1. module load intel
2. module list

> [cpsc424_yl2335@grace2 ~]$ module list
>
> Currently Loaded Modules:
> 1) StdEnv (S)
> 2) GCCcore/7.3.0
> 3) binutils/2.30-GCCcore-7.3.0
> 4) icc/2018.3.222-GCC-7.3.0-2.30
> 5) ifort/2018.3.222-GCC-7.3.0-2.30
> 6) iccifort/2018.3.222-GCC-7.3.0-2.30
> 7) impi/2018.3.222-iccifort-2018.3.222-GCC-7.3.0-2.30
> 8) iimpi/2018b
> 9) imkl/2018.3.222-iimpi-2018b
> 10) intel/2018b
>
> Where:
> S: Module is Sticky, requires --force to unload or purge

Task 1: Serial Program

Row parallel to real axis

Runs	Area	Time
1st run	1.506772	75.855357
2nd run	1.506772	75.854295
3rd run	1.506772	75.883928
Average	1.506772	75.8645267

My results is 9.40000000000385e-05 greater than Dr.Sherman's result. I implememented my solution using rows parallel to the real axis. Since different random numbers are assigned to different cells when processing order changes, different processing orders may lead to slightly different results.

Column parallel to imaginary axis

Runs	Area	Time
1st run	1.506678	75.78239
2nd run	1.506678	75.78217
3rd run	1.506678	75.781115
Average	1.506678	75.7818917

When I used columns parallel to the imaginary axis, t, my result matches with Dr.Sherman's result

Task 2: OpenMP Program (Loop Directives)

Part 1: Thread-safe drand.c

I implemented the thread-safe version of drand.c using the threadprivate directive. The static variable seed will be replicated in all OpenMP threads. Since there is no guarantee of when each will be initialized, I initialized the seed inside the parallel region instead of initializing them before the parallel region. After the fix, different runs with the same number of threads produce the same result, As is shown in the table.

serial

# of threads (n)	Area	Time
1	1.506772	75.854955
2	1.506772	75.851837
4	1.506772	75.853092
10	1.506772	75.853308
20	1.506772	75.852978

default

# of threads (n)	Area	Time
1	1.506772	76.034650
2	1.506750	72.144057
4	1.506732	45.286863
10	1.506782	20.249882
20	1.506756	10.487251

From the serial table we can see that the computing time for mandseq doesn't change with the number of threads. Because the program is sequential and cannot utilize the power of multithreads, only one thread get to execute the program at the end of the day.

We can also see that the computing time for mandomp decreases as the number of threads increases. Each time the number of threads doubled, the computing time almost halved, except for the case when the number of threads is 2. Because the outerloop in which 2 threads split the work iterates over y and the major amount of work of computing the Mandelbrot set is distributed in the lower half (0 <= y <= 0.625, -2 <= x <= 0.5) of the entire region (0 <= y <= 1.25, -2 <= x <= 0.5). The poor load balancing leads to the small amount of reduced time.

Part2: Performance runs

static, 1

# of threads (n)	Area	Time
1	1.506772	76.052222
2	1.506744	38.165881
4	1.506732	19.358029
10	1.506728	7.90447
20	1.506758	4.273475

static, 10

# of threads (n)	Area	Time
1	1.506772	76.079141
2	1.506764	38.55535
4	1.50665	19.66771
10	1.50674	8.455747
20	1.506666	4.843475

dynamic

# of threads (n)	Area	Time
1	1.506772	76.036224
2	1.506786	38.454018
4	1.506682	19.457533
10	1.506714	8.027877
20	1.506772	4.266422

dynamic, 250

# of threads (n)	Area	Time
1	1.506772	76.033413
2	1.50674	38.411561
4	1.506786	38.398759
10	1.506776	38.387843
20	1.506776	38.389187

guided

# of threads (n)	Area	Time
1	1.506772	76.032796
2	1.50672	45.210924
4	1.506608	25.036404
10	1.50664	10.324011
20	1.50686	5.383943

- Almost all schedules are better than the default one beacuse of better load balancing.

- The use of dynamic and guided doesn't bring much performance boost compared to static schedule in this case.
- Small chunk size leads to better load balancing than large chunk size. For the extreme case of dynamic, 250, the computing time almost stays the same after n = 2. From n = 2 to n = 4, the computing time stays the same because the majority of the work is distributed in the region where 0 <= y <= 0.5. After n = 4, the computing almost stays the same because there are only 1250 / 250 = 5 tasks at total.
- Guided schedule doesn't increase the performance in this case.

Part 3: Experiments using collapse clause

default

# of threads (n)	Area	Time
1	1.506712	76.032630
2	1.506730	72.054531
4	1.506894	45.209548
10	1.506826	20.259200
20	1.506858	10.396324

static, 1

# of threads (n)	Area	Time
1	1.506712	76.03371
2	1.506810	38.310022
4	1.506596	19.466276
10	1.506614	8.0367960
20	1.506814	4.2629540

static, 10

# of threads (n)	Area	Time
1	1.506712	76.027889
2	1.506812	38.226444
4	1.506886	19.427644
10	1.506800	7.7725730
20	1.506834	4.2067640

dynamic

# of threads (n)	Area	Time
1	1.506712	76.031209
2	1.506736	38.638000
4	1.506618	19.959599
10	1.506784	8.4838120
20	1.506730	5.0071310

dynamic, 250

# of threads (n)	Area	Time
1	1.506712	76.031486
2	1.506856	38.384654
4	1.506720	19.436210
10	1.506850	7.9761290
20	1.506708	4.1642360

guided

# of threads (n)	Area	Time
1	1.506712	76.030111
2	1.506860	45.213450
4	1.506744	25.039883
10	1.506598	10.374510
20	1.506610	5.2255970

- With the default schedule, the performance almost stays the same because work is almost divided in the same way.
- Using collapse clause, the total number of chunks increases and work is distributed in finer granularity. Generally, load balancing could benefit from this fact.
- For **dynamic, 250**. Previously when the outerloop iterates over y and each unit of work is a row, there are only 5 chunks. Using collapse clause, there are 12500 chunks now and this allows more parallelism. The performance increases with more threads.
- Guided schedule doesn't boost the performance very much.

Task 3: OpenMP Program (Tasks)

single & cell task

# of threads (n)	Area	Time
1	1.506772	78.249794
2	1.506774	39.881293
4	1.506756	20.792411
10	1.50653	9.773973
20	1.506572	6.944453

single & row task

# of threads (n)	Area	Time
1	1.506772	77.990811
2	1.506718	39.199241
4	1.506706	19.626059
10	1.506612	7.879785
20	1.50662	4.336417

multiple & row task

# of threads	Area	Time
1	1.506772	77.994866
2	1.506728	39.138514
4	1.50675	19.654831
10	1.50671	8.002386
20	1.50663	4.194199

- Single thread creating cell tasks performs worse than properly scheduled parallelized for loops because of the overhead of creating the large of number

tasks (one task per cell).
- Single thread creating row tasks performs almost the same as properly scheduled parallelized for loops. It performs better than single thread creating cell tasks because the number of tasks is smaller than that of using cell tasks.
- Multiple threads creating row tasks performs almost the same as single thread creating row tasks. It only parallelizes load creation and doesn't make much difference in this case.

Task 4: Parallel Random Number Generation

I added the following lines of code to the RPG so that different thread will use a different seed and generate a distinct sequence of random numbers.

```
if ( omp_in_parallel() ) {
    int thread_id = omp_get_thread_num();
    seed = (seed & 0xFFFFFFFFFFFFFFE0) | thread_id;
}
```

running time using 20 threads

schedule	Area	Time
default	1.506644	10.716049
static,1	1.50676	4.858447
static,10	1.50669	4.898498
dynamic	1.506682	4.848627
guided	1.506824	5.362572

From the table, I cannot see significant differences compared with the original version of RNG.

Directives to run

1. sh submit.sh