CSE 5441
Programming Assignment 2
Submitted by: Biplob Biswas (biswas.102)

In this assignment, a comparison is presented among three different implementations of the Adaptive Mesh Refinement program. The three implementations are - 1. Serial, 2. Parallel with disposable threads and 3. Parallel with persistent threads. All the implementations use the same data structure, so the comparison of performance would be more accurate. To prevent false sharing among the CPU cores running different threads and better cache utilization, both of the parallel implementations use block distribution method for updating the DSVs.

Output of the serial version of the program:
(The timings and parameters should work as a reference for the performance comparison.)

```
[biplob@owens-login04 lab2]$ make
icc -O3 -lrt -pthread biswas_biplob_serial.c amr.c -o serial
icc -O3 -lrt -pthread biswas_biplob_disposable.c amr.c -o disposable
icc -O3 -lrt -pthread biswas_biplob_persistent.c amr.c -o persistent
```

*The third parameter (i.e. num_threads) is ignored in serial implementation

```
[biplob@owens-login04 lab2]$ time ./serial 0.02 0.02 0 < testgrid_400_12206

**************************************************************************
dissipation converged in 794818 iterations,
    with max DSV = 0.0846372 and min DSV = 0.0829445
    affect rate  = 0.020000; epsilon = 0.020000

elapsed convergence loop time  (clock): 200490000
elapsed convergence loop time   (time): 200
elapsed convergence loop time (chrono): 200512858.000000

**************************************************************************

real    3m20.569s
user    3m20.520s
sys     0m0.017s
```

Summary of timing results with disposable threads:
(All tests are run on input `testgrid_400_12206` with `AFFECT_RATE = 0.02` and `EPSILON = 0.02`, and each of them converged after 794818 iterations)

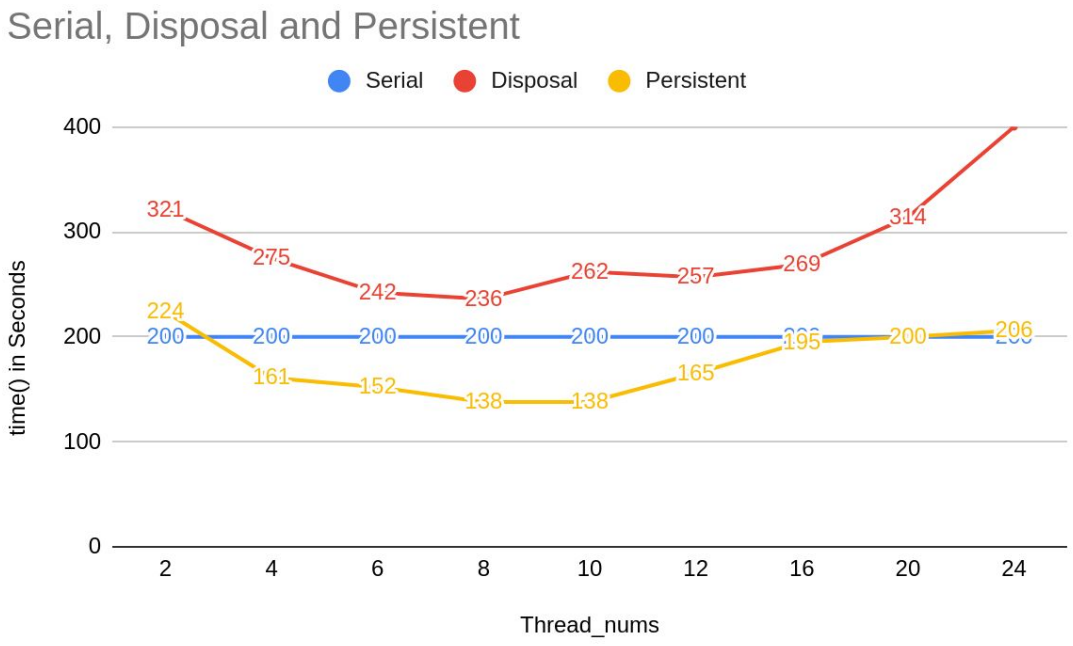
No. of threads	System calls		POSIX real time library	Unix time utility		
	time()	clock()		real	user	sys
2	321	456940000	320885812	5m20.952s	7m4.725s	0m32.252s
4	275	583380000	274243681	4m34.318s	8m27.811s	1m15.631s
6	242	618990000	242027647	4m2.094s	8m34.388s	1m44.652s
8	236	682500000	236146959	3m56.209s	9m3.992s	2m18.546s
10	262	872940000	262119052	4m22.198s	11m12.824s	3m20.150s
12	257	867540000	256652764	4m16.715s	10m45.278s	3m42.310s
16	269	928230000	269101962	4m29.184s	10m45.806s	4m42.470s
20	314	991730000	314705390	5m14.782s	10m13.734s	6m18.040s
24	400	1125770000	399996993	6m40.307s	12m3.235s	6m42.577s

Summary of timing results with persistent threads:

(All tests are run on input `testgrid_400_12206` with `AFFECT_RATE = 0.02` and `EPSILON = 0.02`, and each of them converged after 794818 iterations)

No. of threads	System calls		POSIX real time library	Unix time utility		
	time()	clock()		real	user	sys
2	224	339390000	224254047	3m44.325s	5m22.658s	0m16.789s
4	161	347230000	161393855	2m41.453s	5m19.882s	0m27.387s
6	152	376250000	151668244	2m31.775s	5m37.777s	0m38.519s
8	138	378160000	137553298	2m17.622s	5m28.649s	0m49.557s
10	138	405350000	138133376	2m18.227s	5m46.783s	0m58.594s
12	165	502040000	164587221	2m44.651s	7m5.261s	1m16.823s
16	195	665240000	194683760	3m14.747s	9m20.770s	1m44.512s
20	200	711810000	199554656	3m19.660s	9m46.975s	2m4.881s
24	206	766870000	206762962	3m26.850s	10m25.087s	2m21.828s

Observations:



- As can be seen from the above figure, the parallel version specifically one with persistent threads works better than the sequential version if the thread number is within 4-16. Otherwise, the sequential one performs better.
- Program with disposable threads performs best when it uses 8 threads although its fastest version is ~18% slower than the serial program. But with lower or higher number of threads, its performance gets worse.
- The program with persistent thread performs most effectively with 8 or 10 threads (~31% faster than the serial program) and in that case it outperforms all other program performance. With a lower (<4) or higher (>16) number of threads, it slows down, but still performs better than disposable threads.
- Between the two parallel versions, the one with persistent threads is the most effective one. It performs best with 8 threads (~41% faster than the fastest program with disposable). With 20 threads, its running time is almost the same as the serial program.
- It is expected that the parallel version would work faster than the serial one in multiprocessor system. The result of the parallel program with persistent thread matches the expectation. However, the reason behind the underperformance of disposable thread is the overhead to destroy and create threads after each iteration of convergence loop.
- About timing methods: The running times of the programs were measured in 3 ways: (i) using the `time()` and `clock()` system calls, (ii) using the POSIX real time (`rt`) library and (iii) using the Unix command `time`. Among these, the system call `time()`, the `rt` library and the `real` component of time measured by the `time` command measures the “wall clock” time (merely the difference of timestamps of starting and finishing the convergence loop). So the time measured in these three ways is almost equal. But the `clock()` system call measures the actual processing (active CPU usage) time of the program, where the processing in each CPU core counts. This value is almost equal to the sum of the `user` and `sys` components of `time` command. In both aspects, the time measured by different means is very coherent.
- In the serial version of the program, the wall clock time and the active CPU usage time are almost equal. This means it runs in a single core. The `sys` component is very low compared to the other two, which means most of the work is done in user mode.
- In both of the parallel versions, the active CPU usage time is significantly more than the wall clock time. This means they are utilizing multiple cores of the CPU. Another thing to notice is the increased value of the `sys` component. This means these programs spend more time in kernel mode.
- The wall clock time does not properly reflect the performance of multithreaded programs. The active CPU usage time should be taken into account. This is why `clock()` is better than `time()` or the `rt` library. But the best way for this analysis is to use the `time` command. This includes the information provided by all the methods previously mentioned, and clearly distinguishes between the work done in user mode and kernel mode.

Opinions and Explanations:

- In the sequential version, the value `sys` is almost negligible compared to the value of `user`. This means almost all the tasks in the convergence loop is done in user mode.
- In all cases, the `user` time is larger for parallel programs than it is for the serial program. The reason is, the parallel program does everything the serial program does, with some extra work for thread management. But the parallel programs utilize multiple CPU cores, which minimizes the effect.
- For both versions of parallel programs, `user > real` for any number of threads. This indicates the utilization of multiple cores.
- In the parallel versions, the value of `sys` is significantly larger and it increases with the number of threads. This means they do more work in kernel mode in the convergence loop. The common reason for both of them is the switching context among threads.
- For disposable threads, the `sys` value grows rapidly with the number of threads. It even exceeds the `user` time for a higher number of threads. Creating and joining threads include some kernel-specific tasks. Since the threads are created and destroyed inside the convergence loop, the 794818 iterations cause the program to create and destroy each thread 794818 times. Also, the `sys` time grows in a linear manner with the number of threads (doubling the number of threads almost doubles the `sys` time).
- Creating and destroying threads also contain a small amount of work in user mode, which explains the rise of `user` time with the number of threads.
- **For these reasons, using disposable threads for this computation is not a good idea. The system overhead for managing the threads shadows the performance achieved by multiple threads.**
- This problem can be avoided by using persistent threads. Here the kernel-mode tasks are pausing and releasing the threads at barriers (waiting time at barriers does not count in `sys`, but it is reflected in `real`).
- With persistent threads, the `sys` time grows a little with number of threads because of keeping the creation and destruction of threads outside the convergence loop.
- The `user` time for persistent threads shows little variation for any number of threads. The reason might be the blocking and releasing of threads being managed almost fully in kernel mode, so the amount of work done in user mode is almost the same.