Lab3 c5441aa

Omid Asudeh
OSUID: 500149733

Compiling command: make
Run command: You may run the program using the following commands:
./disposable affect_rate epsilon num_threads < input_test_file
./persistent affect_rate epsilon num_threads < input_test_file
==========
Please read the the whole report! Because I got wired results at the beginning and then managed the experiments in a way that generates interesting correct results!

- I have used Epsilon = 0.3 and affect_rate = 0.3 on the file **testgrid_400_12206.**
- **The experiment was done with 2,4,8,16,24, and 32 threads.**
- **The following shows the comparable result for both lab 2 and 3.**

# of threads	(clock)		(time)		(chrono)		real	sys	user
sequential (lab 1)	16480000		16		16523.8		2m33.593s	2m14.144s	0m0.385s
	lab 2	lab 3	lab 2	lab 3	lab 2	lab 3	lab 3	lab 3	lab 3
disposable 2	87120000	24980000	46	12	46265.1	12507.6	0m12.937s	0m25.734s	0m0.013s
disposable 4	87790000	26670000	26	7	25752.9	6906.39	0m7.215s	0m27.619s	0m0.056s
disposable 8	88630000	25110000	44	7	43578.5	7002.25	0m7.236s	0m25.729s	0m0.171s
disposable 16	90750000	25230000	41	8	40755.2	8500.35	0m8.640s	0m26.119s	0m0.275s
disposable 24	96820000	25620000	29	9	29081.1	8639.46	0m8.740s	0m26.296s	0m0.338s
disposable 32	94120000	25960000	28	9	28045.5	8608.47	0m9.484s	0m26.599s	0m0.428s
persistent 2	87800000	24832000	46	12.4	45457.2	12493.44	0m12.505s	0m24.888s	0m0.013s
persistent 4	90630000	27704000	27	7	27129.1	7356.22	0m7.115s	0m27.049s	0m0.043s
persistent 8	90870000	25244000	27	7.2	26666	7464.552	0m7.219s	0m25.083s	0m0.277s
persistent 16	87130000	25582000	37	9.6	36836.1	9853.652	0m9.617s	0m25.172s	0m0.441s
persistent 24	91330000	26092000	26	11	26128.2	10781.4	0m9.957s	0m25.376s	0m0.714s
persistent 32	87580000	26482000	33	11	33047.2	10773.48	0m10.376s	0m25.940s	0m0.855s

1. **Note**: I've repeated the experiment 5 times for each case, e.g. 5 times ran the experiment for 2 threads, 5 times for 4, etc. and the values in the table above show the average values for the 5 experiments in order to normalize the results.

2. *As you see it seems that OMP is **three times faster. But wait there are something interesting I will explain next.***
3. **Important:** Something really weird I saw during the experiment was that, I saw that timings are severely changing. Actually, once I ran the code for OMP for both disposal and persistent versions, it was 3 times faster than the timing I got from pthreads for the lab 2. The other day I tried to do the whole experiment again. This time OMP was magically too slow. The only reasonable justification that comes to my mind is that may be it depends on the **server**! Finally, I could get a good reason for the problem. I noticed that **when we connect to stdlinux we are randomly assigned to a server e.g. alpha, beta, eta, sl6** etc..**These machines has different performance and this directly affects the running time.** For example, **I found "sl6" three times faster that "eta"** that admits my claim.
4. **So probably the results I got for the lab two in the table above may also was ran on a slower machine and that's why it is 3 times slower not because OMP is much faster.**
5. ***Too important : The only solution for this problem to get comparable results for both lab 2 and 3 is that we run both lab 2 and 3 once we connected to stdlinux. Actually this way we are running both experiments on one machine.***
6. ***Ok. I have done what I explained in 5. This time I connected to stdlinux (sl17) and I reran lab2 and lab 3 both for persistent and disposable with epsilon =0.3 and affect rate = 0.3 for thread numbers = 2,4,8,16,24,32. Also I have done this 3 times for each thread. (E.g. 3* times ran the experiment for 2 threads, 3 times for 4, etc. and the values in the table below show the average values for the 3 experiments in order to normalize the results.)**

#of thrds	(clock)		(time)		(chrono)		real		user		sys	
seq.l (lab 1)	16480000		16		16523.8		2m33.593 s		2m14.144 s		0m0.385s	
	lab 2	lab 3	lab 2	lab 3	lab 2	lab 3	lab 2	lab 3	lab 2	lab 3	lab 2	lab 3
disp 2	38380000	27290000	20	13.66666667	20099.73333	13712.2	24.244	13.758	45.298	27.117	0.49	0.028
disp 4	28446666.67	28896666.67	9	7.333333333	8762.93	7585.896667	9.254	7.723	29.519	29.133	0.52	0.082

disp 8	29670000	30126666.67	9.333333333	9.666666667	9455.606667	10028.27667	11.134	8.047	33.343	29.501	1.931	0.087
disp 16	33626666.67	34120000	11	11.66666667	10524.03333	11333.86667	10.8	12.844	30.057	38.897	4.26	0.461
disp 24	36760000	32653333.33	11.66666667	10.33333333	11504.8	10712.73333	11.966	11.131	31.105	33.623	6.651	0.499
disp 32	39606666.67	33750000	12.66666667	11	12828.76667	11128.3	14.064	11.282	31.87	33.035	10.473	0.586
pers2	27196666.67	26270000	14.66666667	13.33333333	14501.73333	13217.33333	0.219	13.516	25.038	26.459	0.219	0.053
pers4	28400000	28436666.67	8.666666667	7.333333333	8870.343333	7454.646667	8.675	7.602	29.042	28.675	0.152	0.072
pers8	28783333.33	30393333.33	9	9.333333333	8432.71	9203.056667	8.412	9.212	28.087	30.036	0.236	0.383
pers16	29966666.67	42266666.67	10	14.33333333	10081.44	14773.7	10.798	14.968	31.707	41.991	0.378	0.861
pers24	35923333.33	47536666.67	11.66666667	16.66666667	12079.66667	16652.53333	13.319	16.829	38.21	46.557	0.66	1.386
pers32	35013333.33	49113333.33	12	17	11949.3	17285.93333	12.75	17.963	36.356	48.914	0.731	1.786

7. As you see this time values in lab two and three are more similar to each other.

- Another interesting observation was that OMP has created **the same number** of threads that I have requested. This is against what I had expected (arbitrary number of threads). I guess this suggests that OMP assigns as much threads as user requests, as long as it has resources for it. So, when the processors are free, (my case probably) OMP is generous; yet if there is a high processing traffic, the user might not get the amount s/he requests.
- **Which mechanism has the best results**? As the results in the second table shows, for 2 and 4 threads OMP is a little bit faster, but for 8, 16, 24, 32 Pthread is faster. **It seems that as the number of threads increases, the OMP performance decrease!**
- **Which one was the easiest? Obviously**, OMP was easier than pthreads. Although, it took some hours for me to learn the syntax, but once I learnt it, I took me 2 hours to see the results with no errors! OMP is more high-level. As an analogy OMP is like Java and pthreads is like assembly.
- **What to choose for similar applications**? I would definitely choose OMP for the similar applications. Because, it gives me the opportunity to think more on the problem itself than

the syntax and it makes nested parallelization easier. For example, I can save time to try to parallelize more for loops.

- **Under what circumstance I choose pthreads**? there may be cases that I need a guarantee on the number of threads, or I wanted more flexibility, or if I want to make super secure code, in these cases pthreads may outperform OMP. **Most importantly, if the results I got are credible, it is better to use pthreads as the number of threads increases.**
- **What made me surprises? Go read the section 2. :).**

I hope you enjoyed reading this. I put a lot of time doing this lab, and also enjoyed it. (changing the code was easy 2 hours, but solving the problem I explained above and doing the comprehensive experiment for both lab 2 and 3 took 2 days).