CS 5220 HW 2

Laura Herrle lmh95, Sijia Ma sm2462, Kaiyan Xiao kx58

October 19, 2015

1 Profiling

1.1 VTune Amplifier

We attempted to set up and use VTune Amplifier. Since we all have Macs, and VTune can only be used on Linux or Windows, we attempted to run it in a virtual machine (that had plenty of resources allocated). However, this ended up being incredibly difficult to use. We first attempted to use the profiler remotely with the cluster, but that took too much of our time to set up. We then tried to run it locally on the VM, and while we could set it up, attempting to run it locked up the VM. In the end, we decided to time the code manually (this ended up going much more quickly).

1.2 Manual timing

We timed how long each of the four method calls in `run` took, and furthermore how long each section of `compute_step` took. The results for one run of `make big` are in Table 1.

Based on these results, there are two main bottlenecks where we should concentrate our efforts: `limited_drivers` and the corrector section of `compute_step`.

`apply_periodic`	`compute_fg_speeds`	`limited_derivs`	`compute_step`	predictor	corrector	copy
0	19	72	61	13	44	3
0	21	63	60	13	44	2
0	18	62	52	10	38	3
0	29	71	51	10	38	2
0	16	71	52	10	38	3
0	21	70	56	12	39	3
0	21	69	54	10	41	2
0	18	69	54	13	36	4
0	20	70	40	9	28	2
0	15	70	41	10	28	2
0	17	65	52	13	35	3

0	21	71	47	10	32	3
0	20	73	60	13	42	2
0	18	71	55	13	38	3
0	15	70	51	12	35	3
0	21	67	52	11	37	2
0	17	70	61	12	45	2
0	19	62	49	9	37	2
0	19	71	50	13	34	2
0	14	69	63	12	46	3
0	20	63	53	13	36	3
0	22	71	57	10	43	3
0	17	70	62	14	44	4
0	16	78	48	12	31	3
0	16	68	72	13	54	4
0	24	64	49	11	34	3
0	22	71	58	14	41	2
0	20	69	55	11	42	2
0	16	73	62	11	47	3
0	22	71	55	13	38	3
0	22	69	69	14	42	2
0	15	73	53	13	37	2
0	16	62	50	13	34	2
0	15	75	55	11	41	2
0	19	73	66	13	49	3
0	21	74	58	13	42	2
0	21	83	57	11	41	4
0	17	68	47	13	31	2
0	20	68	63	13	47	2
0	16	70	54	14	34	5
0	18	80	58	14	40	3
0	21	61	54	13	38	2
0	21	75	44	12	29	1
0	18	63	55	9	41	3
0	22	66	55	11	40	3
0	21	73	46	11	32	2
0	19	75	64	9	51	3
0	21	67	48	13	32	2
0	16	69	52	10	38	3

0	21	65	47	13	31	2

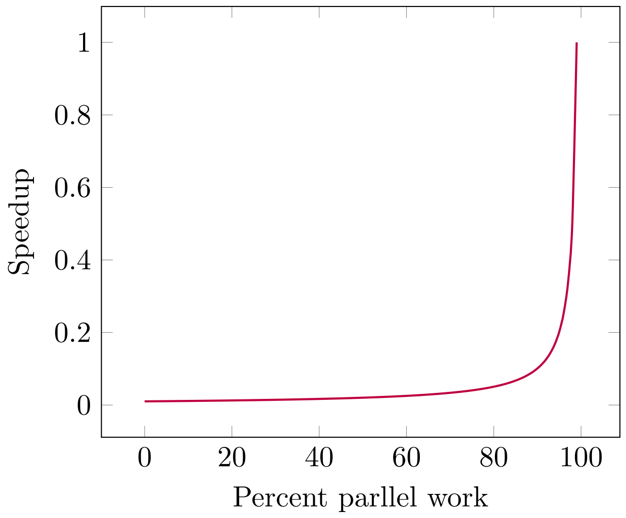
Table 1: Time per Section

2 Parallelization

2.1 Theory

In theory, the more work that is parallel, the better the speedup is. Our issue is that there is a cap on the percent of work that is parallel - we have communication overhead that means that we actually end up with more work overall, so you need to increase the total amount of parallel work for simple or naive parallelization to help much.

Speedup vs. Percent of work that is parallel



2.2 Parallelize Using OpenMP

In `central_2D`, we can just simply use `#pragama omp parallel` outside the while loop to parallelize the code. In order to do the parallelization for both outer and inner loop, we changed `#pragama omp parallel` to `#pragama omp parallel collapse(2)`. We found that for the cases where nx = 200, we found that parallelization actually makes the efficiency even worse. The reason is that, when the number of grid is small, the parallelization overhead, such as syncronization, work distribution, and waiting time will compensate the improvement caused by parallelization. Then we changed nx to be 1000, and this time the improvement is fairly clear: previously the running time for each frame was about 4.5 seconds, and with parallelization scheme, the running time for each frame is about 2 seconds. The code for this parallelization is in `central2d_para.h`. There is a table for these runtimes at Table 2.

Frame	**Initial**	**Parallel**
Average	4.52950854	1.96618996
1	4.468308	1.743288
2	4.468923	1.867762
3	4.471406	1.8865
4	4.471785	1.896273
5	4.472018	1.913024
6	4.472501	2.022954
7	4.501573	1.995248
8	4.473841	1.993301
9	4.473514	1.916126
10	4.474656	1.997498
11	4.474307	2.064316
12	4.47591	2.005893
13	4.479168	2.086459
14	4.478428	1.993293
15	4.324145	1.984681
16	4.289	1.902854
17	4.297817	1.91143
18	4.292165	2.025037
19	4.478597	1.90237
20	4.479466	2.027551
21	4.477933	1.841221
22	4.663032	1.902691
23	4.698583	1.988525
24	4.66134	1.996222
25	4.661559	2.029625
26	4.672745	1.902008
27	4.485846	1.914268
28	4.487038	1.908988
29	4.486155	1.91036
30	4.697128	1.91864
31	4.673596	1.836446
32	4.673456	1.899665
33	4.67664	1.912791
34	4.67343	1.975058
35	4.673302	2.004581

36	4.673729	1.991741
37	4.673215	1.978486
38	4.704125	1.998218
39	4.673623	2.028727
40	4.673856	2.090691
41	4.673854	2.106572
42	4.677842	2.023616
43	4.673541	2.00494
44	4.486652	2.003848
45	4.512664	1.946543
46	4.486836	2.035645
47	4.486642	2.009643
48	4.299649	2.006134
49	4.299835	1.998772
50	4.300053	2.008975

Table 2: Time per Frame

2.3 Domain Decomposition

The basic idea for parallelizing this shallow water problem is to divide the domain into small subdomains and compute each subdomain separately in different threads. And we also need ghost cells around each subdomain to update several steps before communicate with other threads. The number of ghost cell decide the communication frequency and therefore affect the efficiency of parallelization.

We divide the whole area into nblocks * nblocks = 4 * 4 subareas, where each block contains 50 * 50 grids, and we take nblkghost to be 4. At each step, we advance each of the subareas by 4 substeps, using the same algorithm as what we in the serial version, except for that now we are advancing inside the subarea instead of the whole area. And then we synchronize among all those subareas, and update the values of all the ghost cells of all the subareas, and hence completes one advancing step of the whole area.

Another trick for speeding up is using 1-dimensional array to store 2-dimensional matrix, because in this way we can best take advantage of the vectorization.

3 Conclusions

Parallelization makes a lot more sense for large problems than small ones, because the communication overhead is less of a percentage of serial work the larger the problem is. This is true even when work looks easy to parallelize.