

INF236 - Problem Set 1

Konstantin Rygol

February 20, 2019

1 General

1.1 Hardware

The benchmarks were computed on `ift-superbeist.klientdrift.uib.no`. This is a cluster machine at the physics institute. It has 48 cpu divided over 4 sockets. Each socket having one AMD Opteron(tm) Processor 6174. The machine is quite outdated but I think I get nice benchmark/scaling results then if I run on lyng. In the appendix you can find the output of `lscpu`. On this machine was much fewer traffic than on lyng but some calculations were running anyways disturbing my benchmarks. As the problems were quite large each program was only run once and that time taken. To ensure better benchmark results multiple runs with the same parameters would be good to obtain statistics.

1.2 Task 1 and 2

Both implementation work with an arbitrary problem size. Both require the number of processes to be even. If an uneven number of processes is given the programs will abort with a simple error message.

2 Task 1

2.1 Algorithm

Parallelizing in a distributed memory system require a lot of thought on memory management. Which variables have to be shared by all process which ones should be private. How do you divide up the data. In this task the data stored in `env` (environment) will be distributed equally over `p` processes. The other variables like the function will be on all ranks. The idea is that every rank works on its part of the environment called *env_rank*. The ranks need to share the border elements so after every iteration the first and the last element are sent to the neighbouring ranks in a torus like fashion. After the final iteration all *env_rank* are gathered on rank 0 and streamed to std output. This kind of parallelization is highly effective as there is little communication for one iteration. the

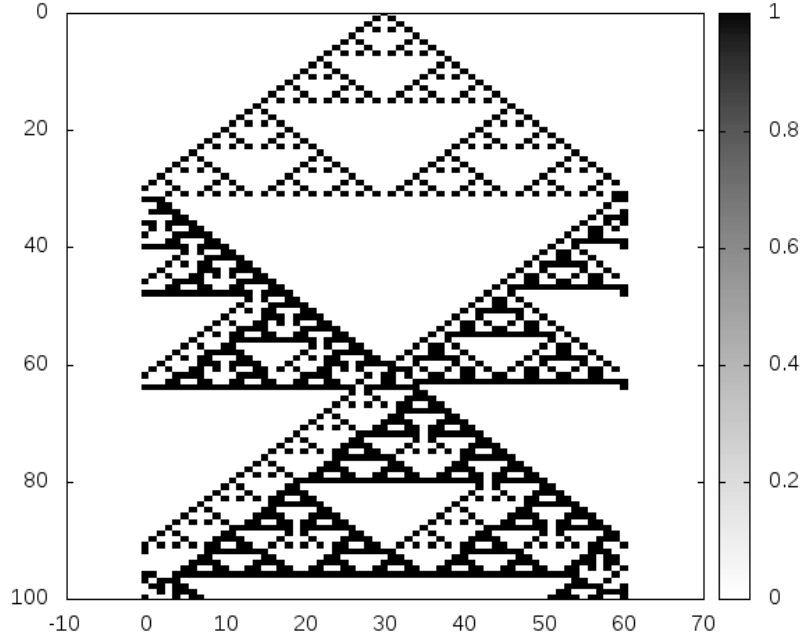


Figure 1: 100 Iterations on middle30.txt

I/O operations are all handled by rank 0. The biggest data transfers happens during the distribution of env and the collection of the env_rank.

2.2 Evaluation 1D Algorithm

For a sufficient problem size there is a speed up with the optimum being between $p=16$ and $p=32$. The computation time is linear in the input size of the starting assignment for most p . We see that for p greater than 32 a break down in performance is observed. This is easily explained by costly synchronization. After every iteration of the algorithm we have to synchronize therefore the speed of the whole simulation depends on the slowest rank if we now enlarge p over the number of cores on the machine we create some slow ranks. This is also a problem if multiple users work on one machine.

3 Algorithm 2d

In this task we expanded our algorithm to a 2d model. In the way of parallelization it does not change much. The environment is split up row wise shown in table1. Each block has length n and height n/p . After every iteration each block send their neighbouring rows to its neighbours. This is done in a torso like fashion. In the folder 2-Sequential there is a python script generating the game_of_live.txt. To ensure that the program works the blinker figure and toad of Conan's game of life were used.

	2	4	8	16	32	64	128
2048	0.839	0.667	0.671	0.945	1.684	5.158	27.228
4096	1.432	1.136	0.923	1.180	1.830	5.751	31.186
8096	2.664	1.856	1.753	1.544	2.169	6.603	31.675
16384	4.974	3.689	2.795	3.018	2.973	9.071	35.191
32768	9.665	6.902	5.421	4.604	5.686	10.830	41.028
65536	19.242	13.402	11.257	9.446	8.544	18.510	49.785
131072	38.205	27.354	21.751	19.074	18.219	24.236	64.354
262144	75.884	51.557	42.726	37.375	34.649	71.878	110.128
524288	152.416	99.968	79.004	73.948	68.455	110.441	415.160
1048576	304.220	196.013	147.866	135.491	137.772	193.754	553.927
2097152	611.076	385.591	288.120	244.230	240.036	340.373	784.973

Figure 2: Run time in seconds of the 1d algorithm in seconds for 100000 evolutions

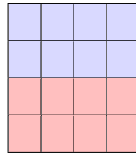


Table 1: The table represents the data distribution scheme for the 2dCellular algorithm for $p = 2$ and $n = 4$. It is a row wise decomposition of the environment

	2	4	8	16	32	64	128
1024	118.163	58.643	29.596	14.805	7.470	7.710	7.477
2048	476.921	237.184	119.531	59.733	29.821	31.148	25.282
4096	1928.160	955.603	482.584	240.910	121.122	128.958	109.26
8192	7756.810	3860.760	1951.340	973.530	485.878	484.671	387.64

Table 2: Run time in seconds of the 2d algorithm in seconds for 1000 evolutions

3.1 Evaluation 2d Algorithm

The run time depends quadratic on the input size n . Which is clear as the environment consists of n^2 elements. It is interesting to see that contrary to the 1d model the best number of processes is between $p=32$ and $p=64$. This might be due to fewer iterations leading to less idle time. Some values as the one for $P=128$ $n=8192$ might also be subject to normal fluctuations.

4 Task 3

4.1 Algorithm branching

The algorithm calculates a chain of associative operations and checks whether the results is in a set F . It takes an assignment and a branching program. The branching program defines the associative operation. The set the operation operates on, the final set and Triplets which are the subjects to the associative operation. We first reduce the triplets with the starting assignment to set of elements in S that are to be "multiplied". The idea behind and efficient parallelization is to organize the data in a tree structure. Every rank get a share of elements that he multiplies. After it is finished it sends its end element to rank zero that sums up all these elements to obtain the final element. This is allowed as the operation is associative. The limit of what we can achieve is $\mathcal{O}(\log m)$.

4.2 Proof

It is to be shown that $\text{val}(P,a)$ can be computed in $\mathcal{O}(\log m)$. If we have unlimited amount of processes p . Lets write down what the algorithm does:

1. First get $I_{a[i_1]}^1 \dots I_{a[i_m]}^m$
2. Calculate associative operation this takes $2 * k/2$ time steps
3. Repeat step 2 until all elements are summed up
4. Check whether final element is in F

Now lets see where we can save time by using multi processing:

1. If we have m processes this can be done in linear time
2. For the second step we need $2 * k / 2 = k$ processes for each element to achieve linear time
3. we have to repeat step 2 $\log(m)$ times if we order the data in a tree. (Prefix sum)
4. We need r processes to do the check in linear time

Conclusively $\mathcal{O}(\log m)$ steps are needed if we have an unlimited amount of processes.

4.3 Evaluation branching

Unfortunately my algorithm does not work correctly with the input of the branching program. I tested it successfully earlier with another operation then the multiplication on A5. My times were extremely short and the program now gives bad results. Therefore I could not make any meaning full timing runs. I suspect my implementation of the A5 multiplication in my python script is corrupted.

5 Conclusion

Generally the effects of parallelization became obvious. Communication was not time consuming in the problems but synchronization is. This becomes evident for using more processes than cores. This is also heavily influenced by other users on the same machine. For better benchmarks a scheduled system e.g. with slurm would be beneficial. In the end I would say it was a nice exercise.

6 Appendix

```

1 Architecture:          x86_64
2 CPU op-mode(s):      32-bit, 64-bit
3 Byte Order:          Little Endian
4 CPU(s):              48
5 On-line CPU(s) list: 0-47
6 Thread(s) per core:  1
7 Core(s) per socket:  12
8 Socket(s):           4
9 NUMA node(s):        8
10 Vendor ID:           AuthenticAMD
11 CPU family:          16
12 Model:               9
13 Model name:          AMD Opteron(tm) Processor 6174
14 Stepping:            1
15 CPU MHz:             2200.151
16 BogomIPS:            4400.15
17 Virtualization:      AMD-V
18 L1d cache:           64K
19 L1i cache:           64K
20 L2 cache:            512K
21 L3 cache:            5118K
22 NUMA node0 CPU(s):   0,4,8,12,16,20
23 NUMA node1 CPU(s):   24,28,32,36,40,44
24 NUMA node2 CPU(s):   3,7,11,15,19,23
25 NUMA node3 CPU(s):   27,31,35,39,43,47
26 NUMA node4 CPU(s):   2,6,10,14,18,22
27 NUMA node5 CPU(s):   26,30,34,38,42,46
28 NUMA node6 CPU(s):   1,5,9,13,17,21
29 NUMA node7 CPU(s):   25,29,33,37,41,45
30 Flags:                fpu vme de pse tsc msr pae mce cx8 apic
                        sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht
                        syscall nx mmxext fxsr_opt pdpe1gb rdtscp lm 3dnowext 3dnow
                        constant_tsc rep_good nopl nonstop_tsc cpuid extd_apicid
                        amd_dcm pni monitor cx16 popcnt lahf_lm cmp_legacy svm
                        extapic cr8_legacy abm sse4a misalignsse 3dnowprefetch osvw
                        ibs skinit wdt nodeid_msr hw_pstate vmmcall npt lbrv
                        svm_lock nrip_save pausefilter

```