Exercise 01

VU Performance-oriented Computing, Summer Semester 2024

Calvin Hoy

2024-03-12

Outline/Preparation

Building

I created a new folder named `build` and ran `cmake ..` inside it to prepare the build environment.

I then ran `make -j$(nproc)` while still in the `build` directory to compile the examples.

Test environment

Benchmarks were run on both my personal computer as well as an LCC3 cluster node. The former is described in the table below; disk specifications given are for the drive used as the working directory for all benchmarks.

Component	Component description
CPU	AMD Ryzen 9 5900X
Memory	2x16GB DDR4-3200 CL16
Disk type	NVMe PCIe 3.0 SSD with DRAM cache
Disk filesystem	Btrfs (zstd-compressed)
GCC version	13.2.1

Benchmark script [B) Experiments]

I first wrote these scripts, then used them to obtain the results described below.

See bench_small_samples.sh and benchmark.sh. The former script may be used to benchmark the programs given in small-samples; it relies on the latter script to conduct the tests using `/usr/bin/env time` and store the output in JSON format.

Usage of `bench_small_samples.sh`:

```
./bench_small-samples.sh <path/to/small_samples> <workdir> <list of programs...>
```

path/to/small_samples: must be pointed to the `small-samples` directory in the Git repository.

path/to/workdir: working directory to be used by `filegen` and `filesearch`

list of programs: which programs to be benchmarked, e.g. `delannoy filegen filesearch`

Example usage:

```
./bench_small_samples.sh ../../small-samples ~/tempdir filesearch nbody
```

LCC3 Notes

On LCC3 I loaded the `gcc/12.2.0-gcc-8.5.0-p4pe45v` module before building any of the code.

I have attached the Slurm job scripts I used to run the benchmarks on LCC3's compute nodes in the `jobs/` directory.

Programs/Benchmark results

All figures for mean and variance given in the following section were taken over five runs of the program. For "wall", "user", "system", and "mem", the columns with "(var)" show the variance, while the ones without "(var)" show the mean result. Unless otherwise specified, runtime is specified in seconds and memory use in kilobytes.

I saw no noteworthy patterns in memory use with any of the programs provided.

Raw (JSON) output from each of the tests as performed on my personal computer as well as LCC3 can be found in the `results_pc/` and `results_lcc3/` directories, respectively.

`delannoy`

`delannoy.c` performs a recursive computation that scales exponentially with one given parameter `N`. It runs very fast for low values thereof, but becomes exponentially slower for larger values.

I chose to test all values of N between 1 and 15 (inclusive). Extrapolating the runtime for N=15 led me to expect a runtime in the ballpark of 10 minutes for N=16, and one hour for N=17, which I deemed simply impractical.

Results

PC:

N	wall	user	system	mem	wall (var)	user (var)	system (var)	mem (var)
1	0.000	0.000	0.000	1256.0	0.000	0.000	0.000	9144.0
2	0.000	0.000	0.000	1250.4	0.000	0.000	0.000	8068.8
3	0.000	0.000	0.000	1283.2	0.000	0.000	0.000	5379.2
4	0.000	0.000	0.000	1315.2	0.000	0.000	0.000	3.2
5	0.000	0.000	0.000	1217.6	0.000	0.000	0.000	8068.8
6	0.000	0.000	0.000	1281.6	0.000	0.000	0.000	5252.8
7	0.000	0.000	0.000	1288.8	0.000	0.000	0.000	5995.2
8	0.000	0.000	0.000	1217.6	0.000	0.000	0.000	8068.8
9	0.000	0.000	0.000	1249.6	0.000	0.000	0.000	7940.8
10	0.010	0.010	0.000	1321.6	0.000	0.000	0.000	156.8
11	0.098	0.096	0.000	1261.6	0.000	0.000	0.000	10140.8
12	0.562	0.558	0.000	1321.6	0.000	0.000	0.000	156.8
13	3.184	3.178	0.000	1249.6	0.002	0.002	0.000	7940.8
14	18.034	18.018	0.010	1314.4	0.091	0.087	0.000	4.8
15	102.066	101.920	0.128	1249.6	0.180	0.226	0.003	7940.8

The results show that indeed the runtime is exponential, with each increase of 1 in problem size increasing the computation time by a factor of roughly 5.66.

Variance is low, as is to be expected given the deterministic nature of the algorithm.

LCC3:

N	wall	user	system	mem	wall (var)	user (var)	system (var)	mem (var)
1	0.000	0.000	0.000	1327.2	0.000	0.000	0.000	43.2
2	0.000	0.000	0.000	1349.6	0.000	0.000	0.000	1332.8
3	0.000	0.000	0.000	1349.6	0.000	0.000	0.000	1252.8
4	0.000	0.000	0.000	1369.6	0.000	0.000	0.000	372.8
5	0.000	0.000	0.000	1353.6	0.000	0.000	0.000	812.8
6	0.000	0.000	0.000	1364.0	0.000	0.000	0.000	704.0
7	0.000	0.000	0.000	1364.8	0.000	0.000	0.000	411.2
8	0.000	0.000	0.000	1368.8	0.000	0.000	0.000	3067.2
9	0.010	0.004	0.000	1356.0	0.000	0.000	0.000	584.0

N	wall	user	system	mem	wall (var)	user (var)	system (var)	mem (var)
10	0.050	0.050	0.000	1382.4	0.000	0.000	0.000	556.8
11	0.306	0.304	0.000	1348.0	0.000	0.000	0.000	256.0
12	1.718	1.716	0.000	1351.2	0.000	0.000	0.000	507.2
13	9.660	9.646	0.000	1365.6	0.002	0.002	0.000	212.8
14	54.314	54.232	0.000	1361.6	0.064	0.067	0.000	964.8
15	306.040	305.578	0.008	1360.8	0.930	0.964	0.000	1323.2

The per-core performance on LCC3 appears to be about a third of that of my personal computer. Interestingly, there appears to be generally lower variance in memory use on LCC3.

`filegen`

`filegen.c` creates a given number of directories, each containing the same specified number of files with a pseudorandom size within a given range. Each file contains pseudorandom content, generated at runtime.

The workload clearly scales with each parameter – likely linearly, but with different respective constant factors.

To see how each of the three main parameters – number of directories, number of files, and file size – affect performance, I chose to test the following sets thereof:

- 1,000 / 10,000 / 100,000 / 1,000,000 directories, 1 file, 1B
- 1 directory, 1,000 / 10,000 / 100,000 / 1,000,000 files, 1B
- 1 directory, 1 file, 10,000 / 100,000 / 1,000,000 / 10,000,000 / 100,000,000 B

I chose to keep the minimum and maximum file sizes the same, as having a "random" component to this would only serve to make scaling less consistent.

Each benchmark run was conducted using the default seed of `1234`, and all generated files were deleted in between runs.

Results

PC:

dirs	files	file size	wall	user	system	mem	wall (var)	user (var)	system (var)	mem (var)
1	1	10 kB	0.000	0.000	0.000	1508.0	0.000	0.000	0.000	0.0
1	1	100 kB	0.000	0.000	0.000	1700.0	0.000	0.000	0.000	0.0
1	1	1 MB	0.000	0.000	0.000	2320.0	0.000	0.000	0.000	22736.0
1	1	10 MB	0.060	0.046	0.004	11188.8	0.000	0.000	0.000	22387.2
1	1	100 MB	0.662	0.520	0.072	99060.0	0.016	0.000	0.000	6296.0
1	1000	1 B	0	0.000	0.020	1404.0	0.000	0.000	0.000	24888.0
1	10000	1 B	2	0.010	0.238	1436.8	0.000	0.000	0.000	9603.2
1	100000	1 B	2	0.162	2.412	1398.4	0.017	0.000	0.019	10140.8
1	1000000	1 B	12	1.704	23.852	1397.6	3.689	0.047	0.748	10308.8
1000	1	1 B	4	0.000	0.024	1480.8	0.000	0.000	0.000	5995.2
10000	1	1 B	6	0.012	0.276	1397.6	0.003	0.000	0.003	26052.8
100000	1	1 B	2	0.202	2.750	1365.6	0.275	0.000	0.280	6532.8
1000000	1	1 B	22	2.062	28.372	1436.8	51.508	0.022	35.108	9603.2

The table above shows several things:

- Each execution parameter *does* exhibit roughly linear scaling in one or multiple relevant performance metrics.
- While generation of file content impacts the time in user space as well as memory usage, creation of additional files and directories directly increases the time spent in kernel space.

- File/directory creation is much slower as compared with pseudorandom content generation – certainly due to the overhead of context switching, inode allocation, etc.
- Directory creation is slower than file creation.
- For the slow test runs, there is high variance in the wall and system time measurements.

LCC3:

Due to the relatively slow disk performance on LCC3's scratch space, I reduced the number of benchmark passes to 3 – therefore, in particular the variance is not directly comparable. Further, I increased the job time limit to 1 hour, omitted all tests with >100,000 files, and moved the old files out of the way after each pass instead of deleting them.

dirs	files	file size	wall	user	system	mem	wall (var)	user (var)	system (var)	mem (var)
1	1	10 kB	0.000	0.000	0.000	3273.333	0.000	0.000	0.000	1669.333
1	1	100 kB	0.000	0.000	0.000	3281.333	0.000	0.000	0.000	2181.333
1	1	1 MB	0.020	0.010	0.000	3281.333	0.000	0.000	0.000	1797.333
1	1	10 MB	0.150	0.140	0.007	11024.000	0.000	0.000	0.000	9904.000
1	1	100 MB	1.510	1.447	0.053	98897.333	0.000	0.000	0.000	11797.333
1	1000	1 B	0.040	0.000	0.040	3268.000	0.000	0.000	0.000	1792.000
1	10000	1 B	0.930	0.033	0.383	3262.667	0.765	0.000	0.000	5141.333
1	100000	1 B	11.127	0.387	3.880	3276.000	0.778	0.000	0.000	2704.000
1000	1	1 B	0.060	0.000	0.050	3256.000	0.000	0.000	0.000	1776.000
10000	1	1 B	1.087	0.047	0.543	3264.000	0.450	0.000	0.000	112.000
100000	1	1 B	20.497	0.533	5.650	3277.333	0.305	0.001	0.001	2949.333

`filesearch`

`filesearch.c` implements a recursive (depth-first), linear directory search, outputting the name and size of the largest file found in the present working directory tree.

This workload clearly scales with the total number of files in the directory and random-access performance. `stat` is used to look up each file's size, thus the actual size of the files should not matter for performance. The number of directories should also have a sizeable impact on performance, as each `readdir`/`closedir` adds a system call and thereby a context switch.

I used `filegen` to create the following test setups, each with files sized between 1B and 10kB:

- 1 directory, 1,000 files
- 1 directory, 1,000,000 files
- 1,000 directories, 1 file each
- 1,000 directories, 1,000 files each
- 1,000,000 directories, 1 file each

Results

PC:

dirs	files	wall	user	system	mem	wall (var)	user (var)	system (var)	mem (var)
1	1000	0.000	0.000	0.000	1403.2	0.000	0.000	0.000	9323.2
1	1000000	1.608	0.260	1.348	1364.0	0.000	0.000	0.000	6736.0
1000	1	0.000	0.000	0.000	1397.6	0.000	0.000	0.000	10308.8
1000	1000	1.588	0.258	1.320	1436.8	0.000	0.000	0.000	9603.2
1000000	1	9.664	1.608	7.226	1318.4	8.736	0.001	1.321	15996.8

We can see that searching 1,000,000 files across 1,000 directories is roughly equivalent to the time taken searching the same number of files contained in a single directory. Meanwhile, searching 1,000,000 directories containing one file each is close to an order of magnitude slower.

We once again see high variance in the wall time for the test run with 1,000,000 directories, though curiously the variance in system time is not at all similar, despite the program spending 75% of its runtime in kernel space.

LCC3:

dirs	files	wall	user	system	mem	wall (var)	user (var)	system (var)	mem (var)
1	1000	0.000	0.000	0.000	1473.6	0.000	0.000	0.000	260.8
1	1000000	1.558	0.298	1.246	1384.8	0.008	0.000	0.009	5499.2
1000	1	0.002	0.000	0.000	1453.6	0.000	0.000	0.000	7956.8
1000	1000	1.506	0.308	1.188	1526.4	0.009	0.000	0.009	4044.8
1000000	1	41.626	1.604	6.502	1490.4	5771.857	0.010	1.662	1612.8

Massive variance was observed for the benchmark with 1,000,000 directories and 1 file each. This was due to the first run taking nearly 3 minutes, with each subsequent run taking only ~7 seconds.

run #	wall	system
1	177.53	8.80
2	7.54	6.00
3	7.72	6.06
4	7.81	5.78
5	7.53	5.87

I *would* suspect disk caching as the culprit, however the time spent in kernel space only varied very little, so I'm unsure what would have caused this.

`mmul`

`mmul.c` performs matrix multiplication in a serial fashion. It takes no command-line arguments, thus we have no means of scaling the workload without alterin, `nbody`g the macro `S` determining the size of the matrices being multiplied.

Results

PC:

	wall	user	system	mem
mean	2.608	2.602	0.000	24681.6
variance	0.002	0.002	0.000	9444.8

LCC3:

	wall	user	system	mem
mean	5.780	5.760	0.000	24568.800
variance	0.006	0.006	0.000	2219.200

`nbody`

`nbody.c` models a particle physics simulation with a fixed number of particles in a finite space over a fixed number of iterations. We again have no way of changing the size of the workload without modifying the code; in this case, the macros `N`, `M`, `L` and `SPACE_SIZE`.

Results

PC:

	wall	user	system	mem
mean	1.650	1.646	0.000	1744.8
variance	0.001	0.001	0.000	9835.2

LCC3:

	wall	user	system	mem
mean	4.764	4.758	0.000	1833.600
variance	0.000	0.000	0.000	2116.800

qap

`qap.c` implements a recursive algorithm solving the Quadratic Assignment Problem. Input is given via `.dat` files in the `problems/` directory. As the problem at hand is NP-hard, we cannot determine an upper bound for the runtime based on input size.

After observing a runtime of well over 1 hour for a problem size of 18, I chose to only benchmark the input files up to a problem size of 15 on my PC.

Results

PC:

input	wall	user	system	mem	wall (var)	user (var)	system (var)	mem (var)
chr10a.dat	0.000	0.000	0.000	1442.4	0.000	0.000	0.000	8068.8
chr12a.dat	0.040	0.040	0.000	1480.8	0.000	0.000	0.000	5995.2
chr12b.dat	0.040	0.040	0.000	1492.0	0.000	0.000	0.000	6992.0
chr12c.dat	0.060	0.060	0.000	1486.4	0.000	0.000	0.000	6532.8
chr15a.dat	4.634	4.624	0.002	1360.0	0.004	0.004	0.000	9680.0
chr15b.dat	1.252	1.252	0.000	1479.2	0.000	0.000	0.000	5899.2
chr15c.dat	4.208	4.200	0.004	1398.4	0.006	0.006	0.000	10140.8

We can see from the results for a problem size of 15 that problem size does not directly correlate with time taken – `chr15b.dat` took only 1/4~1/3 as long to process as `chr15a.dat` and `chr15c.dat`. All benchmarks show very low variance, which is to be expected as there are no random or pseudo-random factors affecting computation time.

LCC3:

On LCC3 I submitted several additional jobs with non-exclusive allocation and a 1-hour time limit, one each for all input files with a problem size of 18 or greater. I allocated all jobs to the same node, took just a single measurement (rather than five) and let the jobs run until they either finished or exceeded the time limit. Only the job for `chr18a.dat` finished, its runtime is included in the table below.

input	wall	user	system	mem	wall (var)	user (var)	system (var)	mem (var)
chr10a	0.010	0.010	0.000	1484.0	0.000	0.000	0.000	512.0
chr12a	0.160	0.160	0.000	1507.2	0.000	0.000	0.000	259.2
chr12b	0.146	0.140	0.000	1490.4	0.000	0.000	0.000	220.8
chr12c	0.210	0.210	0.000	1479.2	0.000	0.000	0.000	1427.2
chr15a	15.442	15.412	0.000	1485.6	0.050	0.050	0.000	1764.8
chr15b	4.170	4.160	0.000	1512.0	0.000	0.000	0.000	632.0

input	wall	user	system	mem	wall (var)	user (var)	system (var)	mem (var)
chr15c	13.950	13.926	0.000	1495.2	0.000	0.000	0.000	395.2
chr18a	879.550	877.720	0.030	1492.0	N/A	N/A	N/A	N/A