Operating Systems I Course Project

Guanzhou Hu
Student ID: 36136477
Email: hugzh1@shanghaitech.edu.cn
School of Information Science and Technology
ShanghaiTech University

Xinyu Zhang
Student ID: 82649845
Email: zhangxy3@shanghaitech.edu.cn
School of Information Science and Technology
ShanghaiTech University

The project required us to perform certain tasks on the given source data `systor17-01.tar` and optimize our code to improve the performance. This report will describe our design of the raw project, how we optimize it, and the performance results.

I. Requirements

Task 0: Decompress `systor17-01.tar` twice to get the data files.
Task 1: According to the *IOType* field, divide all the entries in the data files into two files `R.csv` and `W.csv`.
Task 2: Sort the entries in `R.csv` and `W.csv` in ascending order of *Size* respectively. For the entries with the same size, sort them by *Timestamp*.
Task 3: Collect the number of entries with each size in `R.csv` and `W.csv` respectively.

The goal is to reduce the total timespan of the whole project (measured from the start of decompression to the end of result writing) by at least 10%.

II. Design of the Raw Project

A. Data structures

1) node: A self-defined `struct` used in Task 2. When sorted, each entry is represented by a `node`. It holds three fields: `line_idx` (the original index of the entry in `R.csv` or `W.csv`), `size` (*Size*), and `time_stamp` (*Timestamp*).

2) cnt_struct: A self-defined `struct` used in Task 3. For each *IOType*, the number of entries with each different size is counted with a `cnt_struct`, where the `size` field stands for the *Size* and the `cnt` field records the count.

B. Static Variables

1) NUM: An array of length 2 recording the number of entries of the two *IOType*s. Index 0 is for *R*-type entries, 1 for *W*-type. All other arrays of length 2 use the same encoding, which will be left out afterwards.

2) CNT: An array of length 2 recording the number of different sizes for the two *IOType*s.

3) LUN_idx_arr: An array holding all the possibilities (0, 1, 2, 3, 4, 6) for the last digit before the `.csv` extension of the filenames (i.e., the digit after `LUN`). Since they are not continued, this can simplify the loops in decompressing and opening data files.

4) global_src_file: An array to store all the pointers pointing to decompressed data files. Filled after the decompressed files are opened.

5) size_cnt_arr: An array of length two where each element points to an array of `cnt_struct` instances.

6) node_arr: An array of length two where each element points to an array of `node` instances.

C. Subroutines

Instead of the four tasks, we decompose the whole project into five subroutines. These functions have neither arguments nor return values, each with a self-defined type of `PROCESS`. They are called by function `runProcess`.

1) decompress: Corresponds to Task 0. Use the Linux `execl` function to decompress the source data file `systor17-01.tar`, which results in 32 `.csv.gz` files. Call `execl` to decompress the files again to get the data files which we can finally perform the analysis on. Since `execl` takes control of the current process and anything after it will never run, we must fork a child process to call `execl` every time we want to decompress a file, and the parent process needs to wait for the termination of the child process.

2) scanStatistics: An auxiliary subroutine to collect needed information in `NUM[2]` and `CNT[2]`. Call `fscanf` to scan each entry in the decompressed data files, extract the *IOType* and *Size* field, set the corresponding element in the two-dimensional array `size_mark`, and increment the element in `NUM`.

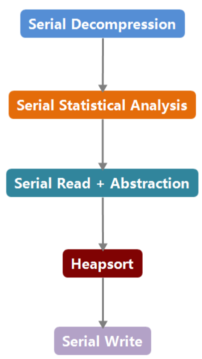
3) abstractRead: Corresponds to Task 1 but does some extra work. Create and open two intermediate files by `fopen`, `R-int.csv` and `W-int.csv`. Scan each source data file once more and write each line to the intermediate file that match the `IOType`. Meanwhile, for each entry, record the three fields `line_idx`, `size` and `time_stamp` in the corresponding node array, which will be used in the next stages.

4) sortEntries: Corresponds to Task 2 and Task 3. We use heapsort to sort the huge node arrays pointed to by element in `node_arr` in the two intermediate files, due to its $O(n \log n)$ time complexity and in-place characteristic.

5) writeResult: Writes the sorted entries and the statistic analysis result to the final files `R.csv` and `W.csv`. For each node in the node array, use the `line_idx` field and the `fseek` function to find the position of the entry, read from the $\star$-int.csv files, and write to the final files. The statistic analysis result is written as "size,cnt" pairs, each pair on a new line.

D. Main Function

First, we run the decompress subroutine, and then open the decompressed data files. Then, we run the scanStatistics subroutine. With the information collected in scanStatistics, we allocate memory for the arrays that elements of node_arr and size_cnt_arr point to. Afterwards, subroutines abstractRead, sortEntries and writeResult are called successively. The project is finally wound up with the memory freed and the files closed.



The results were uploaded to https://pan.baidu.com/s/1T2fp0MYft_vxDl9vrSKSbA. The password is bz89.

III. Optimization

A. Modified Data structures

In struct node, line_idx is replaced by offset combined with src_file_idx. Also, another new field write_offset is added.

B. New Static Variables

1) NUM_ARR: A two-dimensional version of the static NUM array. Records the number of entries of the two *IOType*s in each data file.

2) NUM_THREADS: Number of threads used by OpenMP.

C. Subroutines

Except sortEntries, all subroutines are parallelized with OpenMP.

1) decompress: The first decompression remains the same. The decompression of .csv.gz files are parallelized with the directive #pragma omp parallel for num_threads(NUM_THREADS). Instead of waiting for each forked child processes to terminate sequentially, the OpenMP threads leave waiting child processes to the initial thread, with a file_cnt counting the number of files not decompressed yet. The initial thread will wait until file_cnt turns 0.

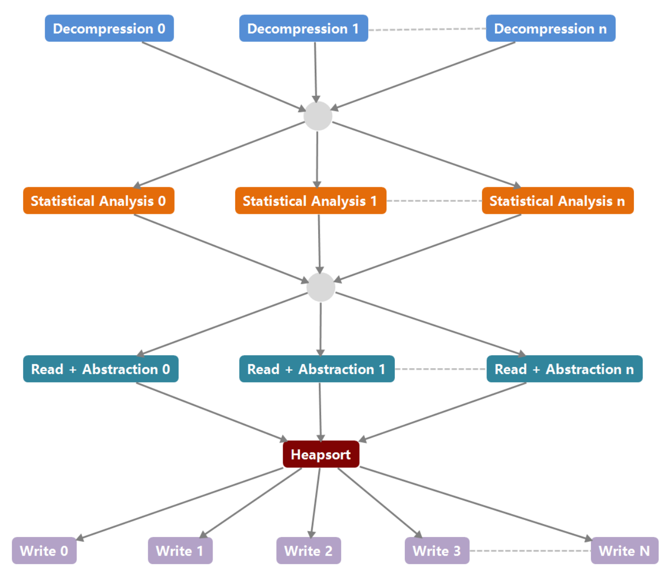
2) scanStatistics: We declare two temporary variables for NUM, R_sum_tmp and W_sum_tmp. With OpenMP for directive, the threads are assigned separate files, so only synchronization on R_sum_tmp and W_sum_tmp is needed. Thus reduction(+:R_num_tmp, W_num_tmp) is added to the directive in decompress, which will make private copies of the two variables for each thread and write the sum of them into the global shared variables. During the scan of each file, elements in NUM_ARR rather than NUM is incremented. Values of R_sum_tmp and W_sum_tmp are copied into NUM after all the threads join.

3) sortEntries: During heapsort, the element at index 0 is never used. Here a dummy head is set with write_offset equal to the length of the header line, and all other fields 0. We will use this feature to parallelize writeResult.

4) writeResult: First we calculate the position in destination files where each node will be written at. Inside the block parallelized by #pragma omp parallel num_threads(NUM_THREADS) source files and destination files are opened locally. Since the written area of each thread is independent of each other's, there is no synchronization problem when writing the entries. The last thread alone write the statistic analysis result.

D. Main Function

Nothing changed except that before any of the subroutines, NUM_THREADS is set to the smaller value of the number of available processors and half the number of data files.



IV. Performance Results

A. Analysis Method

1) Total Timespan: For the *total timespan* measurement, we used timeval struct in system library time.h to acquire and record time consumed by each subroutine. We add them up at analysis stage and get total timespan across the whole project, which starts from the beginning of decompression and ends when the writing to result files is done.

2) CPU Utilization and I/O Performance: For the other performance statistics, including *CPU utilization rate* (proportion of CPU time spent on project programs instead of staying idle), *I/O bandwidth* (sum of read and write bytes per second) and ratio of I/O time to the total timespan, we used the Linux command line tool `iostat`. At the start of running, we start with `iostat -x` with 10 seconds interval between sampling. Then we truncate the performance results within the total timespan, which should be valid recordings. The average performance results are then calculated.

B. Testing Environment and Results

1) **Dell XPS 15 Laptop**
   - 4 cores (8 threads)
   - SSD disk
   - All cores are used by OpenMP
2) **Intel Xeon Gold 6132 Server**
   - 28 cores (56 threads) on a single node
   - HDD disk
   - Only 8 cores (16 threads) are used

The results are presented in the following two tables:

TABLE I
DELL XPS 15 LAPTOP

	Unoptimized	Optimized
Timespan (secs)	300.50	114.28
% CPU Util Rate	10.268	40.133
% I/O Time Used	6.568	10.978
Bandwidth (kB/s)	56257.0	116923.5

TABLE II
INTEL XEON GOLD 6132 SERVER

	Unoptimized	Optimized
Timespan (secs)	393.51	92.45
% CPU Util Rate	1.427	5.439
% I/O Time Used	10.359	20.926
Bandwidth (kB/s)	33649.4	65407.6

It can be clearly seen that the reduce of total timespan under two testing environments are $\frac{300.50-114.28}{300.50} \approx 51.99\%$ and $\frac{393.51-92.45}{393.51} \approx 76.51\%$ respectively, both reaching the requirement of 10%. CPU utilization rate and I/O bandwidth are also significantly increased.

ACKNOWLEDGMENT

We are grateful to Yuzhuo Jing, who helped us set up the channel of the server.