Homework solution

MPI_word_count (MWC nextly) is a solution for homework problem of counting word frequencies in a set of files using MPI.

Build and run

This project is compiled using cmake, use `./build_release.sh` or `./build_debug.sh` for compilation. For debug reasons, it is possible enabling detailed logger calling `log_set_level(LOG_DEBUG);`

Run program using mpirun like so: `$ mpirun -np 8 ./mpi_word_count <files_dir> <result_file>`

High level solution description

MWC uses MPI for message passing, so in project we have two roles - master and worker. Master once received, in input, files directory performs first evaluation of the situation. Lists all files and calculates sizes. As we don't want perform intensive computing on master in preparing all things, master distributes workload using only file sizes a source of information about a "complexity". Once all work was distributed, workers (including master) can start working. Each worker, reads assigned to him files and updates internal hashmap containing word:count pairs. When reading is done, worker sorts words using merge sort and sends result back to master process. The last one (master) receives data from workers using, only merging, updates his local frequency map. Once all workers has finished sending, master produces final csv with sorted results.

Workload division between workers

Dividing workload is the most simple but most important part of this project. First, we don't want do a lot of work because in this time workers are idle, so here is needed fast and robust solution. Second, if the work is not balanced well, we can see that some workers are more loaded than others and this is not ok.

Final solution used here is Approx Algorithm (see `split_files_equally`) which gives T < 3/4(T*) guarantee. Essentially, this means that balancing used here is 3/4 worst comparing to the optimal one. Algorithm calculates all sizes, orders files in not decreasing order, and puts file one by one to the most free worker. The border line solution is when we have 1 very very big file and 1 very small file, but considering the problem, where each worker must atomically take the file, the solution will be the optimal one.

Wordcounting process

All counting processing is gathered under `worker_process_files` function, which receives a list of files and returns `WordFreq` structure which contains processing result. Internally, is used modified version of HashTable which stores frequencies and LinkedList which is used to store. HashTable when created, accepts size, this is important value, because it should be near 20% of HT items for better performance/memory. If you know that files has high entropy (essentially a lot of different words), put in `wc_constants.h` WF_HT_SIZE to approx count of words * 5.

LinkedList contains words because HT itself is not using strings as keys (this should be very slow) but crc32 of string. By the way, LinkedList with words is great, when we need to sort words in lexicographic order before returning result.

Merging process

The only one process which performs the final "merging" of results is master, this is sequential part of project, but it is incredibly fast. Workers, as said before, delivers to master just sorted results, so the one thing to do is to merge results, which can be done in O(n).

Message passing architecture

This project uses a very small subset of features available in MPI, this is caused by using very simple Map/Reduce architecture. As it is possible to see in `wc_mpi_helpers.h` was used simple standard `MPI_Send`. In reduce process is used `MPI_Pack/MPI_Unpack`, for passing complex data and `MPI_Get_count` with `MPI_Probe` for dynamic memory allocation sizes.

Why not using master as scheduler?

Workers, once finished with 1 file, should ask master for other piece of work. This should be great opportunity to have excellent workload distribution, but this can make architecture more complex, creates additional network traffic and, by the way, can't give revolutionary results.

Performing merge using binary tree

MPI works only with contiguous memory, so reducing using binary like tree adds a lot of overhead in serializing/deserializing of data structures, especially for very big and random dataset.

Proof of correctness

Correctness was validated in two different tests, one with very small dataset available in `test_dir` another with a very large dataset which can be generated with python script.

The first dataset, contains italian words ["uno", "due", "tre", "quattro", "cinque", "sei", "sette"] which are randomly putted in 4 files. Each word is repeated times as names (uno = 1, due = 2, tre = 3 and so on).

Test is launched using mpi_run with 1/2/3/4/5/6 workers, each time, resulting CSV is correctly ordered:

```
word,frequency
cinque,5
due,2
quattro,4
sei,6
sette,7
tre,3
uno,1
```

For small dataset is quiet easy check if the resulting CSV is correct, so for the bigger was used SHA256. Dataset used is `high_entropy_xxl_t1`.

Outputs:

```
$ head result-n1.csv
~/ClionProjects/mpi_word_count
word,frequency
a,1546
aa,1531
aaa,1566
aah,1436
aahed,1534
aahing,1489
aahs,1524
aal,1490
aalii,1438
.......
```

```
$ shasum -a 256 result-*
~/ClionProjects/mpi_word_count
95936ca27d921e5205a1b7c27457846c72d8261888a8402128697be9924df303  result-n1.csv
95936ca27d921e5205a1b7c27457846c72d8261888a8402128697be9924df303  result-n2.csv
95936ca27d921e5205a1b7c27457846c72d8261888a8402128697be9924df303  result-n3.csv
95936ca27d921e5205a1b7c27457846c72d8261888a8402128697be9924df303  result-n4.csv
95936ca27d921e5205a1b7c27457846c72d8261888a8402128697be9924df303  result-n5.csv
95936ca27d921e5205a1b7c27457846c72d8261888a8402128697be9924df303  result-n6.csv
95936ca27d921e5205a1b7c27457846c72d8261888a8402128697be9924df303  result-n7.csv
95936ca27d921e5205a1b7c27457846c72d8261888a8402128697be9924df303  result-n8.csv
```

This means that output resulting files remains equal even if the worker size changes.

Performance evaluation

Weak and strong scalability was calculated using AWS cluster of 4 m4.large instances with total of a 8 vCPU. Was chosen 4 different datasets (see Datasets section) where each next dataset has double complexity (not only size!) of the previous one. All timings were measured using linux `time` command; seconds are unit of measurement.

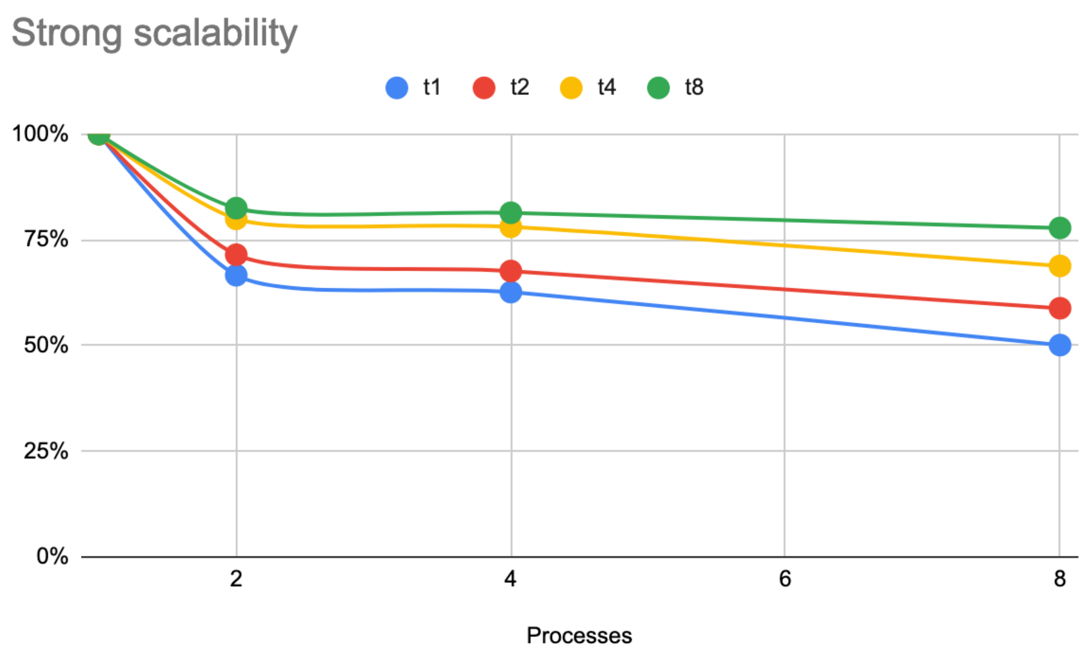
Timings

Dataset/Processe	1	2	4	8
t1	8.507	6.3846	3.3957	2.1222
t2	17.51	12.2508	6.4746	3.7215
t4	38.617	24.1488	12.366	7.0146
t8	97.16	58.9023	29.844	15.6105

Strong scalability

Strong scalability is measured when the problem size stays fixed but the number of processing elements are increased. As provided in table below, strong scalability is worst for small dataset and better for the bigger one. The explanation is that for smaller datasets there is a lot of communication and preparation (sequential operations), and only a bit of real computation (parallel). On other hand, for big dataset, speedup is not optimal but great. Essentially, for t8 dataset with 8 processors we have only 22% of loss for communication stuff, comparing to 50% of t1.

Dataset/Process	**1**	**2**	**4**	**8**
t1	100	66.621	62.631	50.107
t2	100	71.465	67.610	58.814
t4	100	79.956	78.071	68.815
t8	100	82.476	81.390	77.800

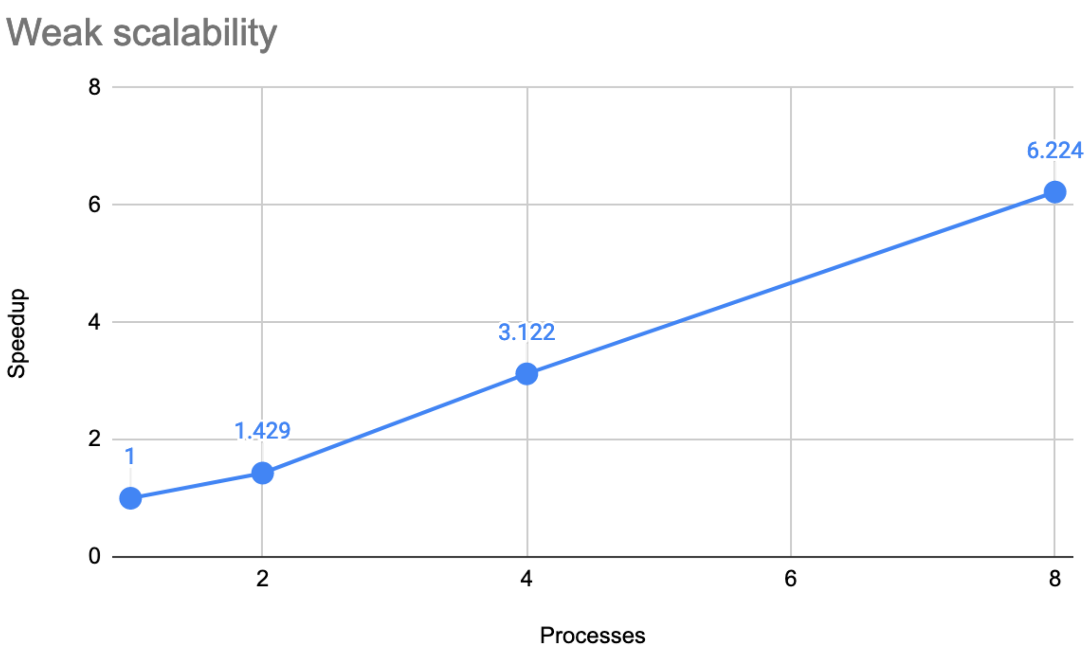


Weak scalability

In weak scalability, the workload for each processor remains constant, other processors are used to solve even bigger problem. So, giving 8 processors, we can do 622.40% more work rather than with only 1 processor. Optimal value should be 800% but the homework problem is not completely parallelizable; 800% is the theoretical maximum.

Dataset/Process	**1**	**2**	**4**	**8**
t1	**100**	133.242	250.523	400.858
t2	100	**142.929**	270.441	470.509

t4	100	159.913	**312.284**	550.523
t8	100	164.951	325.560	**622.402**



Datasets

- high_entropy_xxl_t1 = using english dictionary, first 10000 words, 1000 files each size is from 100 to 50000 words, total size 249908 kbytes (245Mb)
- high_entropy_xxl_t2 = using english dictionary, first 20000 words, 2000 files each size is from 100 to 50000 words, total size 514500 kbytes (503Mb)
- high_entropy_xxl_t4 = using english dictionary, first 40000 words, 4000 files each size is from 100 to 50000 words, total size 993120 kbytes (970Mb)
- high_entropy_xxl_t8 = using english dictionary, first 80000 words, 8000 files each size is from 100 to 50000 words, total size 2014272 kbytes (2Gb)

Dataset can be generated using python script available in `test_dir/dataset_generator.py` .

Final considerations

As mentioned in performance evaluation, achieved results are very good for real word, when we are not talking about theoretical values. Of course, solution have a lot of potential for improvement, like on the fly hashmap increasing, starting data delivery once completed, usage of master like scheduler of processes and even better memory handling.