SPM ASSIGNMENT-3

DATED:25 May 2024

Javeria XXX

1. **INTRODUCTION:**

This assignment was designed to parallelize an algorithm that determines the topk words inside a specified text file collection. The first k terms are those that appear the most frequently across all input files.

FastFlow is used to create a parallel implementation of the Word-Count-seq.cpp code.

1. **Code:**

struct Reader : ff\_monode\_t<std::vector<std::string>> {

Reader(const std::vector<std::string> &filenames\_, const uint64\_t num\_lines\_, const uint64\_t left\_workers\_) : filenames(filenames\_), num\_lines(num\_lines\_), left\_workers(left\_workers\_) {}

std::vector<std::string>\* svc(std::vector<std::string>\*) {

uint64\_t id = get\_my\_id();

uint64\_t block\_size = filenames.size() / left\_workers;

uint64\_t r = filenames.size() % left\_workers;

long lower = (id \* block\_size) + (id < r ? id : r);

long upper = lower + block\_size + (id < r ? 1 : 0);

for (uint64\_t i = lower; i < upper; i++) {

std::ifstream file(filenames[i], std::ios\_base::in);

if (file.is\_open()) {

//file read

ff\_send\_out(lines);

}

}

file.close();

} return EOS;

}

const std::vector<std::string> &filenames;

const uint64\_t num\_lines;

const uint64\_t left\_workers;

};

This Reader struct is a component of the FastFlow library used for parallel task execution. It inherits from ff\_monode\_t, indicating that it is a mono-input node, meaning it receives input from a single source.

**svc() Function**:

This function is the main processing function of the Reader struct. It receives a pointer to a vector of strings as input.

· It retrieves the ID of the current worker thread using get\_my\_id().

· It calculates the range of file indices that the current worker thread will process based on its ID and the total number of worker threads (left\_workers).

· It iterates over the files within its assigned range.

For each file:

* + It opens the file for reading.
  + It initializes a pointer to a vector of strings (lines) to store the lines read from the file.
  + It enters a loop to read lines from the file until the end of file (eof()).
  + Within the inner loop:
    - It initializes a new vector of strings (lines) to store a batch of lines from the file.
    - It reserves memory for num\_lines lines in the vector to reduce memory allocations.
    - It reads lines from the file and appends them to the vector until either num\_lines lines are read or the end of file is reached.
    - It sends the vector of lines to the next stage of processing using ff\_send\_out().
* After processing all files, it closes the file streams.

· Finally, it returns EOS (End Of Stream) to signal the end of processing.

Overall, the Reader struct is responsible for reading lines from multiple files in parallel, batching them.. It efficiently utilizes available worker threads to maximize throughput.

struct Tokenize : ff\_minode\_t<std::vector<std::string>> {

Tokenize(umap &um\_) : um(um\_) {}

std::vector<std::string>\* svc(std::vector<std::string>\* lines) {

for (auto l = lines->begin(); l != lines->end(); l++) {

char \*tmpstr;

char \*token = strtok\_r(const\_cast<char\*>(l->c\_str()), " \r\n", &tmpstr);

while(token) {

um[std::string(token)]++;

token = strtok\_r(NULL, " \r\n", &tmpstr);

total\_words++;

}

for(volatile uint64\_t j {0}; j < extraworkXline; j++);

}

delete lines;

return GO\_ON;

}

umap &um;

};

The Tokenize struct is another component of the FastFlow library used for parallel task execution. It inherits from ff\_minode\_t, indicating that it is a mono-input node that produces multiple outputs.

· **svc() Function**:

* This function is the main processing function of the Tokenize struct. It receives a pointer to a vector of strings (lines) as input.
* Inside the function:
  + It iterates over each line in the input vector of strings.
  + For each line:
    - It tokenizes the line using strtok\_r, splitting it into individual words based on the delimiters " \r\n".
    - It initializes a character pointer token to hold the current tokenized word.
    - It enters a loop to process each tokenized word:
      * It inserts the tokenized word into the unordered map um and increments its count.
      * It increments the total\_words counter, indicating the total number of words processed.
      * It retrieves the next tokenized word using strtok\_r.
    - After processing all tokenized words in the line, it performs additional work (extraworkXline) specified by the volatile uint64\_t j loop.
  + After processing all lines, it deletes the input vector of strings to free memory.
  + Finally, it returns GO\_ON to indicate that processing can continue with the next input.

Overall, the Tokenize struct is responsible for tokenizing lines of text into individual words, counting their occurrences, and updating the corresponding unordered map. It efficiently processes batches of lines in parallel and performs additional work specified by extraworkXline.

int main(int argc, char \*argv[]) {

.....

//create an instance of All-to-All

ff\_a2a a2a;

std::vector<ff\_node\*> left\_w;

std::vector<ff\_node\*> right\_w;

// create instances of workers and organize them into two sets: the first set and the second set.

for (uint64\_t i = 0; i < left\_workers; i++)

left\_w.push\_back(new Reader(filenames, num\_lines, left\_workers));

for (uint64\_t i = 0; i < right\_workers; i++)

right\_w.push\_back(new Tokenize(umap\_results[i]));

a2a.add\_firstset(left\_w, 1); //add\_firstset method to add the first set of workers to the all-to-all communication pattern.

a2a.add\_secondset(right\_w);//add\_secondset method to add the second set of workers to the all-to-all communication pattern.

// execute the communication pattern using the run\_and\_wait\_end method

if (a2a.run\_and\_wait\_end() < 0) {

error("running a2a\n");

return -1; }

ffTime(STOP\_TIME);

auto time1 = ffTime(GET\_TIME) / 1000;

ffTime(START\_TIME);

if (right\_workers > 1) {

for (uint64\_t id = 1; id < right\_workers; id++)

for (auto i = umap\_results[id].begin(); i != umap\_results[id].end(); i++)

umap\_results[0][i->first] += i->second;}

// sorting in descending order

ranking rank;

rank.insert(umap\_results[0].begin(), umap\_results[0].end());

ffTime(STOP\_TIME);

auto time2 = ffTime(GET\_TIME) / 1000;

std::printf("Total time (s) %f\nCompute time (s) %f\nSorting time (s) %f\n", time1 + time2, time1, time2);

if (showresults) {

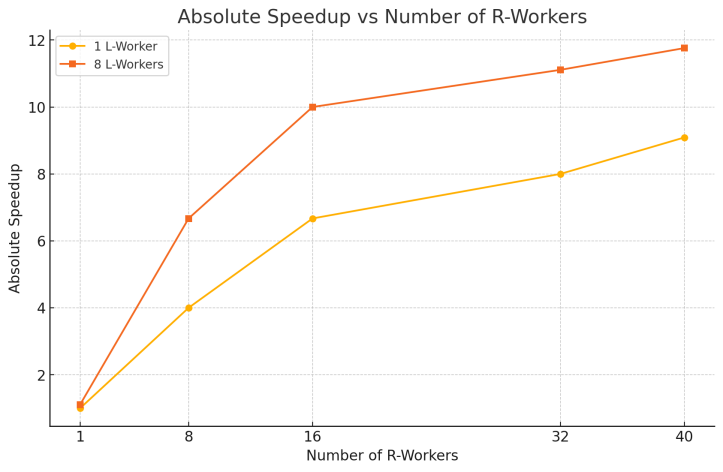
// show the results }

return 0;

}

In the main function, it can be seen that the business logic of the program is realized through the all-to-all building block, which is capable of emulating a farm if the number of L-Workers is equal to one. L-Workers are realized through the multi-output Reader node and R-Workers with the multi-input Tokenize node. L-Workers use the on-demand scheduling policy. Each R-Worker, which will then be a thread, stores the number of words in its own local umap. After all-to-all termination, the various umaps are merged into a single umap, and then sorted. The *Reader* node (L-Worker), is responsible for reading a set of files assigned to it using a block distribution. For each file, it reads num lines rows, storing them on the heap, until the end of the file is reached, and for each set of rows sends the pointer to them to an R-Worker via the function *ff send out()*. The *Tokenize* node (R-Worker), tokenizes each row in the input vector and stores the number of words encountered in the local umap.

**Evaluation:**

Absolute Speedup is the metric used to evaluate the performance of the parallel version.Tests were performed by varying the num lines and extraworkXline parameters. The graphs show the speedup as the number of R-Workers and L-Workers changes for the *spmcluster.unipi.it* cluster, which has a maximum of 40 logical cores. For L-Workers the values used are 1(Farm) and 8(A2A) so as to appreciate the difference between *farm* and *all-to-all*. 

The speedup achieved with the farm is comparable with the OpenMP version. Instead, it is possible to observe that with all-to-all the speedup is significantly higher. It is therefore evident how increasing the degree of parallelism for L-Workers the execution time found is greatly reduced. In conclusion, the version built through FastFlow achieves higher performance than the version built with OpenMP tasks.

**Conclusion:**

The parallel implementation of the Word-Count-seq.cpp code using FastFlow demonstrates significant performance improvements over the OpenMP version, particularly when utilizing an all-to-all communication model with multiple L-Workers. By efficiently reading and tokenizing lines from multiple files in parallel, the FastFlow version maximizes throughput and minimizes execution time. The observed speedup, especially with the all-to-all model, highlights the advantage of increasing parallelism in the L-Workers, resulting in a more efficient processing pipeline. Overall, the FastFlow implementation proves to be superior, showcasing its capability to handle high degrees of parallelism and achieve higher performance metrics compared to the traditional OpenMP approach.