0.20in

PARALLEL PROGRAMMING

REPORT

NUPUR UNAVEKAR

Inhaltsverzeichnis

1 Overview

Algorithm 1: LU Decomposition

inputs : a(n,n)

outputs: π(n), l(n,n), and u(n,n)

initialize π as a vector of length n;

initialize u as an n x n matrix with 0s below the diagonal;

initialize l as an n x n matrix with 1s on the diagonal and 0s above the diagonal;

for $i = 1$ *to n* **do**

  $\pi[i] = i$;

end

for $k = 1$ *to n* **do**

  max = 0;

  for $i = k$ *to n* **do**

    if *max* $< |a(i, k)|$ **then**

      max = $|a(i, k)|$;

      k' = i;

    end

  end

  if *max == 0* **then**

    error (singular matrix);

  end

  swap $\pi[k]$ and $\pi[k']$;

  swap a(k,:) and a(k',:);

  swap l(k,1:k-1) and l(k',1:k-1);

  $u(k, k) = a(k, k)$;

  for $i = k + 1$ *to n* **do**

    $l(i, k) = a(i, k)/u(k, k)$;

    $u(k, i) = a(k, i)$;

  end

  for $i = k + 1$ *to n* **do**

    for $j = k + 1$ *to n* **do**

      $a(i, j) = a(i, j) - l(i, k) * u(k, j)$;

    end

  end

end

LU-decomposition factors a matrix A in L(lower) and U (upper) triangular matrices.

We first see the time complexity of algorithm of LU decomposition. The given algorithm includes 4 major steps which includes:-

- Initialization of Matrices L, U and pi : $O(n^2)$

- Finding max element in a column j below the diagonal : $O(n^2)$ (including outer k-loop)
- Swapping of row with max element with row j $O(n^2)$ (including outer k-loop)
- Updating value of a[i][j] $(O(n^3)$ (including outer k-loop)

Hence the last step is where most of the time is taken since it is $O(n^3)$. We will use this time complexity analysis in our parallelization implementation.

2 Reading File with `fgetc_unlocked()` and `String Buffer`

`nex_elem`

This helper function helps to read the next element from the input file into the buffer for further processing. It takes 2 inputs - f is a pointer to file object. `s` variable of type string and is used to store the element read from the file. c variable stores the next character from file using `fgetc_unlocked()` function. This function reads a character from the file without performing any locking or buffering, which nakes it more efficient for reading large files. We keep on appending `s` with `c` in every loop iteration. So at the end, `s` will have complete element from file.

3 Reading Data in Chunks from File into Vector

`get_data`

`get_data` function reads data from a given file pointer and stores in the given vector pointer arr in chunks based on the limit given. Here the variable limit is an integer representing the maximum number of characters that can be read from the file. It reads data from the file in chunks till the limit is reached. arr stores the data read in chunks.

A while loop is used that continues until either limit becomes 0 or elements becomes 0. This loop is used to read data from the file in chunks based on the given limit. The function decrements the value of elements by 1 in each iteration of the while loop until it reaches 0, indicating that all elements have been read from the file. The function then decreases the value of limit by the value of addr, which represents the size of each element in the file. This helps to keep the track of the remaining space in the limit for reading data from the file.

The function declares and initializes a temporary string s to store the element read from the file. After reading the element from the file and storing it in `s`, the function appends `s` to the end of the vector arr using the `push_back()` function for vectors, which adds `s` as a new element at the end of the vector. The function then decreases the value of limit by the size of `s`, which represents the number of characters read from the file, to keep track of the remaining space left.

4 Writing Vector of Strings to File

print_arr

This function writes the contents of a vector of strings (arr) to a file with the given filename. First declare and initialize an empty string s. Then it opens the file with the given filename in write mode and redirect the stdout to the file, and store the FILE* pointer. Then it iterates through the vector arr. The function appends the string at the current index i of arr to the s string and a newline character to s to separate each string in arr by a newline in the output file. If the size of s exceeds the value of string_max, which is the maximum size limit for writing data the function prints the contents of s to the standard output using cout, and then resets s to an empty string. After the loop, the function vector arr is cleared. At the end the function prints the remaining contents of s to the standard output (which is redirected to the file) using cout, to write any remaining data in s to the file.

5 Reading Data from File into Vector of Strings

get_data_run

This function reads data from a file pointed to by file and stores it in a vector of strings arr, up to a certain limit defined by a limit variable and a maximum size maxsize. First it clears the vector arr by removing all its elements, ensuring that it is empty for adding new elements to it. Then it runs a while loop which runs till limit and maxsize conditions are met or we can say that the maximum number of elements or the maximum size of data has been reached, respectively . The function declares a string s to store the read data from the file inside the loop. Then it stores the next element from the file in s. It updates maxsize by subtracting the size of the read element in addition to the size of string s, effectively reducing the remaining space available for reading from the file. finally s is added back to arr from back. Note it decrements limit by 1, indicating that an element has been read from the file.

6 Merging Multiple Sorted Runs of Data

do_runs

This function reads data from multiple files into a vector, maintains a priority queue to keep track of the smallest element from each file, and then writes the output to a file. It handles merging multiple sorted runs of data into a single sorted output. It uses pointers, vectors, and priority queues to efficiently manage the reading and writing of large amounts of data from multiple files.
It initiates following intial variables -

- element_avaliable is a vector that contains number of available elements in each run, initialized with elems values.
- n represents number of runs to be merged

- Files represents a vector of file pointers to the runs to be merged, obtained by opening the files with names specified in names using `fopen` function
- Pointers represents a vector of integers to keep track of the current pointers to the elements in each run, initialized with 0.
- Data represents a vector of vectors of strings to store the data read from each run, initialized as empty vector.
- The Heap represents a priority queue of pairs of strings and integers, where the strings represent the elements from the runs.
- The integers represent the index of the run in the data vector.
- The priority queue is used to keep the smallest element at the top for efficient merging

It reads data from each run into the data vector using the `get_data_run` function, which reads data from the files pointed by files variable into the corresponding vectors represented by data, considering the available elements specified in `elements_avaliable` and then the maximum size of data to be read per run is calculated as `max_size/n`. It populates the heap priority queue with the first element from each data vector, along with the index of the run. `s` is initiated as empty string to store the merged output. It then opens a file with the name specified in `name_this` for writing. It then runs a loop that runs until the heap priority queue is empty or we say that when all elements from all runs have been processed and merged into the output.
The loop works as -

- It first pops the smallest element from the heap priority queue, which is stored in the temp pair.
- It then appends the string value of temp starting with the smallest element first to the s string, followed by a newline character, to represent the merged output.
- It checks if the size of s exceeds the maximum size of a string allowed (string_max). If so, it prints the current contents of s to the redirected output using cout and clears s to avoid exceeding the allowed string size.
- It then retrieves the index of the run from temp.second. it checks if there are no more elements available in the current run (elements_avaliable = 0).
- If so, it sets the pointer of the current run to -1, indicating that the run has been fully processed. if not and if there are still elements available in the current run, it reads the next batch of data from the file pointed by the current run using the get_data_run function, updates the pointer to 0, and pushes the first element from the updated.

7 External Mergesort with stops

`external_merge_sort_withstop`

This code is main function and implementation of an external merge sort algorithm with some additional features like stopping the merging process after a certain number of merges (`num_merges parameter`) and using a variable `k` to determine the maximum number of runs that can be merged in a single pass.

- It first opens the input file (input) in read mode and initialize some variables, like `num_present` which stores the number of elements in each run, and names which stores the names of temporary files generated during the sorting process.
- When there are still elements to sort, read a chunk of data from the input file with a maximum size of `max_size` into a temporary vector temp, sort it, and write it to a temporary file named as "temp.0."followed by the size of the `num_present vector`.
- It updates `num_present` with the size of temp and append the temporary file name to names.
- If this is the last chunk of data to be sorted and there is only one run `(num_present.size() = 1)`, print the sorted data to the output file `(output)` using the `print_arr()` function and return the number of merges performed (merges).
- It then initializes a variable 'level' to 1. It runs a while loop till number of runs `(num_present.size())` is reduced to 1, which means all runs have been merged into a single sorted output.
- It iterates over the runs in chunks of k (this depends on number of runs remaining). For each chunk, create a temporary file name based on the current level `(temp."+to_string(level)+"."+to_string(id))` and call the `do_runs()` function to merge the runs within the chunk, update merges, and store the resulting merged run in the temporary file.
- If `num_merges` is greater than 0 and the number of merges performed till now is equal to `num_merges`, return merges and stop the merging process.
- Update `num_present_temp` with the counts of elements in the merged runs, and append the temporary file name to `names_temp`. Increment id to keep track of the temporary file names.
- After it is done with merging of all the chunks of runs at the current level, it increments level and updates `num_present` and names with `num_present_temp and names_temp`, respectively.
- Finally the loop ends by returning 0, indicating that the sorting process is complete.

8 Performance

	English-subset.txt (1000000)	Random.txt (1000000)	Large_file_600mb.txt (1255325)	Large_file_1gb.txt (2090674)	Large_file_5gb.txt (10457193)
K =2	T = 1.33 s MRS= 82468	T = 5.86 s MRS=573244	T = 7.21 s MRS = 715380	T = 28.81 s MRS = 827288	T = 363.59 s MRS = 845112
K = 8	T = 0.76 s MRS=82468	T = 5.91 s MRS=573236	T = 7.10 s MRS = 715388	T = 29.39 s MRS = 827292	T = 187.58 s MRS = 845728
K = 16	T= 0.78 s MRS = 82468	T = 5.54 s MRS=573264	T = 7.27 s MRS = 715384	T = 33.52 s MRS = 827268	T = 176.59 s MRS = 845124

Abbildung 1: Performance