COMP4300 Individual Project - Documentation

Runpeng Luo u7109933

May 26, 2022

Table of Contents

1. Part A
    1. Block Matrix Multiplication by OpenMP
    2. SUMMA algorithm by MPI
    3. Cannon's algorithm by MPI
    4. SUMMA algorithm by Pthread & Cannon's algorithm by Pthread
2. Part B
    1. Parallel Sample Sort
3. Reference

Note: the input file will only be parsed by the main thread (Part A.1) or master process (default process 0, Part A.2, 3, Part B.1) or master thread (default thread 0, Part A.4, 5). Result is compared with sequential algorithm result. And all the output and elapsed time will be stored in output file with same manner as input. **Benchmark and result is discussed in report.md/report.pdf, related program raw data output and benchmark script is stored in benchmark_assets/benchmark_result/, you can use `datagen` and `matrixgen` to generate random input test case**

For each program, please comment out the portion below for verifying the output result

```
    /* This portion of code is appeared in the end of every program's main
method.And only used to verify the final output result by comparing to
sequential computed result manually */
    if ((err = verifyResult(A, B, m, n, p, C))){
        fprintf(stderr, "wrong result\n");
        // exit(0);
    }else{
        printf("Correct result\n");
    }
```

Part A

Block Matrix Multiplication by OpenMP

First of all, consider the matrix A has dimension m * n, and B has dimension n * p, the resulting matrix C must have dimension m * p. For general block matrix multiplication, where no restriction on where matrix dimension is divisible by blocksize (pre-defined, related to cache size, platform dependent), we first divide the matrix A, B and C into numbers of sublocks (use `ceil` function to ensure the boundary entries also be covered in a block), where each of them have size at most blocksize * blocksize.

```
# blocksize = 3, m = 10, n = 7, p = 4
# (a,b): (row size, column size) in each sublock
    |(3,3)|(3,3)|(3,1)|     |(3,2)|(3,2)|      |(3,2)|(3,2)|
A:  |(3,3)|(3,3)|(3,1)| * B:|(3,2)|(3,2)|  = C:|(3,2)|(3,2)|
    |(3,3)|(3,3)|(3,1)|     |(1,2)|(1,2)|      |(3,2)|(3,2)|
    |(1,3)|(1,3)|(1,1)|                        |(1,2)|(1,2)|
```

As the matrix has been divided by subblocks, as example shown above, notice every corresponding pair of subblocks in A and B have matching dimension for multiplcation. After that, instead of iterating each matrix entry directly through m,n,p, we iterate through every subblocks in a similar fashion. Within the nested sub-block for-loop, we perform normal matrix multiplication for matching subblock A and B pair. From example above, when computing first row, first column subblock C, we need 3 subblocks along the first row of matrix A and 3 subblocks along the first column of matrix B.

Finally, we can apply some parallelism techniques using OpenMP to speed up the code. We can use OpenMP `parallel for schedule(dynamic)` to parallel the outer subblock for-loop.

As the variation (version 2), I further parallelize the code by adding another OpenMP `parallel for schedule(dynamic)` for the inner subblock i-k-j for-loop region, in which I also use `collapse` to turn outer subblock for-loop into 1 (also for inner i-k-j for-loop), notice that since within the i-k-j for-loop, different iteration may write to same entry `C[i][j]` in parallel. To avoid the race condition, I add a OpenMP `atomic` directive to make the `C[i][j]` entry write into a critical section.

SUMMA algorithm by MPI

First of all, as sanity check, if number of processes is not a number that composited as its square root, the program will terminate the processes that not forming a square process grid (process grid dimension is refered by `r`). Otherwise, all the process will be used for computation. Consider master process input the matrix A with dimension m * k and matrix B with dimension k * n via normal file IO module, master process will divide the matrix A, B and C into subblocks (use `round` function to ensure the boundary entries also be covered by last row/column of subblocks), as example below.

```
# number of process in used = 9 = 3 * 3, r = 3, m = 10, k = 7, n = 4
# (a,b): (row size, column size) in each sublock
                                  /\
     |(3,2)|(3,2)|(3,3)|      |(2,1)|(2,1)|(2,2)|      |(3,1)|(3,1)|(3,2)|
A:  <|(3,2)|(3,2)|(3,3)|> * B:|(2,1)|(2,1)|(2,2)|  = C:|(3,1)|[3,1]|(3,2)|
     |(4,2)|(4,2)|(4,3)|      |(3,1)|(3,1)|(3,2)|      |(4,1)|(4,1)|(3,2)|
                                  \/
```

For instance, consider process grid has dimension r * r, for matrix A subblock division, first (r-1) process grid rows (columns) will get subblock with dimension row round(m/r) (column round(k/r)), where process grid rth row (column) will hold dimension row (r - (r-1) * round(m/r)) (column r - (r-1) * round(k/r)) (0-indexed), similar fashion for distributing matrix B and C into process grid. By doing so, every process within the process grid will hold a portion from global matrix respect to its relative position in the grid.

Since MPI allowed to create sub communication group, for simplicity, I create sub group `row_comm` for all process along each row (similarly for `col_comm`), process within the same `row_comm` is identified by its world rank within `MPI_COMM_WORLD` divided by process grid dimension `r` (modulo when determine column sub group `col_comm`).

Known that for computing subblock `C[i][j]`, we need i-th subblock row of matrix A and j-th subblock column of matrix B, and each row (column) will involve `r` (process grid dimension) number of subblocks, hence, we need to broadcast A's i-th row subblock along the process row and B's j-th column subblock along the process, which requires `r` iterations. Within each iteration, C's local subblock will be added once up until all required subblock A and B be received and used.

Once all the subblocks of C be computed by corresponding process, we can gather the subblocks back to master process and obtain the elpased time and final global matrix C.

As the variation (version 2), notice that during local subblock matrix multiplication, we can use OpenMP to introduce more data parallelism by parallelizing i-k-j for-loop using `omp for schedule(dynamic)`, such that the program is turned into a hybrid multi-process&thread program.

Cannon's algorithm by MPI

Similar to SUMMA algorithm, Cannon's method also require every process within the grid handle a subblock of matrix A, B and C, so the matrix distribution method (before the main algorithm start) is equivalent to SUMMA's implmentation.

For the main part, we need to left cyclic shift the matrix A's subblock row and up circular shift the matrix B's subblock column, I used `MPI_Sendrecv_replace` to achieve this.

Consider the example below, where we have 3 * 3 process grid, and 2nd row (0-indexed) is attempting to do the left cicular shift by 2. (other row just stay for simplicity)

```
# process grid = 3 * 3, r = 3, m = 10, k = 7, n = 4
# (a,b): (row size, column size) in each sublock

     |(3,2)|(3,2)|(3,3)|         |0|1|2|
A:   |(3,2)|(3,2)|(3,3)|  grid:  |3|4|5|
    >|(4,2)|(4,2)|(4,3)|<       >|6|7|8|<---- involved process row

      |a|b|c|  2nd row left circular shift by 2 |a|b|c|
A:    |d|e|f|  ---------------------------->   |d|e|f|
    <-|g|h|i|<-                                |i|g|h|
```

Consider the row wise, where process 6,7,8 has row id 0,1,2, respectively. The shift is done as follows: (px -> py means process x sends to process y) 1. p0 -> p1 2. p1 -> p2 3. p2 -> p0

Notice that during cyclic shift, every process will receive once and send once only. Now, we generalize the idea to `r` process grid dimension, consider `s`th row is attempting to perform left cyclic shift `s`th row by `s`, we pick process `j` (j is in 0..r-1, row rank) along the `s`th row, the following modular arthmetic formula can be used to compute the row ranks that process `j` is sending to and receiving from.

```
# s: s-th row in the process grid
# j: row rank, j-th process along the s-th row_comm
# r: process grid dimension
send_to_rank = (j - s + r) mod r
recv_from_rank = (j + s + r) mod r
```

The above formula can also be deployed for up-circular shift.

Since during circular shift, the buffer that stores the awaiting-to-send submatrix can also be reusd for receiving, besides the actual received submatrix may have different size compare to current holding one. I used padding to ensure that buffer has memory space that enough to store biggest submatrix hold by any process + 2 (additional 2 is for storing the row*column size of sending/receiving submatrix). Example as below:

```
# convert submatrix with size 2 * 3 into buffer.
           |1|3|2|
submatrix: |4|6|5|

# first two integers (2,3) are used to recover the matrix dimension when #
delivered
buffer: |2,3,1,3,2,4,6,5|
```

Since buffer space can be reused properly, I use `MPI_Sendrecv_replace`, which perform blocking `MPI_Send` and `MPI_Recv` operation concurrently, such that avoiding the *dining table philosophers problem*.

After the `skew A` and `skew B` circular shift process, we have a for-loop which has similar fashion as `SUMMA`, where each process compute the temporary local submatrix C value by current holding sumatrix A and B, then share it out (also receive) next submatrix for computation. By doing so, every process ends up with a complete submatrix C, which can be gathered by master process for finalization.

The variation for Cannon's algorithm implementation is similar to `SUMMA` one, where I parallelize the local matrix multiplication by a OpenMP `for schedule(dynamic)` directive, and wish to achive a better speed up by hybriding multi-process and multi-thread.

SUMMA algorithm by Pthread & Cannon's algorithm by Pthread

The Pthread version of SUMMA algorithm and Cannon's algorithm has similar idea and implementation as MPI version, communcation is achieved via customized communication method (using mutex, conditional variable and barrier), and both Pthread implementations reuse same communication method, although defined in separate program since using global variable. The main idea of algorithms are not discussed again.

For instance, I uses conditional variable and mutual exclusion lock to implement the synchronized send/recv method. I initialize the individual message buffer space (associate with corresponding mutex and conditional variable) for all threads, declared under global scope. For example, when thread `i` want to send message to thread `j`, it will first check the capacity for jth message buffer space, whether thread `i` is

current sender to thread `j`, and vice versa. If all pre-conditions are satisfied, thread `i` will simply deep-copy the message into jth message buffer space. Thread who calls the `Send` will be blocked if receiver's message queue is full, otherwise it will directly put the message and signal the receiver. Before exit, sender will be blocked until receiver acknowledge its message has been delivered, and both sender and receiver will re-initialize the variable (Who I'm sending to and receiving from). Finally, thread can safely exit the `Send` routine without blocking other thread.

Thread who wishes to receive message should call `Recv` function. In particular, it will first signal any blocked sender who is trying to send to current receiver thread. If the message buffer space is empty when the function is called, it will block until the sender who concurrently call the `Send` function fill the buffer space. When the receiver be signalled by sender thread, it will release from receiver's conditional variable blocking and deep copy the received messsage before exiting the `Recv` routine. Same as `Send` function, all the receiver's operation is protected by mutex.

Notice that all above operations are protected by mutex and conditional variable, pesudo-code as below to describe the behaviour.

```
## Sender (A) send to receiver (B), with message (M), buffer space (BUF),
## mutex a, b, conditional variable conda, condb
# Recv
# 3-way handshake
lock(a)
Is_Empty(BUF) = True
signal(conda)
unlock(a)

lock(b)
while(A_s receiver /= B) wait(condb, b)
unlock(b)

lock(a)
B_s sender = A
signal(conda)
unlock(a)

# actual recv
lock(b)
while(Is_Empty(BUF)) wait(condb, b)
load message from BUF
unlock(b)

# 3-way handshake
lock(a)
Is_Empty(BUF) = True
signal(conda)
unlock(a)

lock(b)
while(A_s receiver == B) wait(condb, b)
unlock(b)
```

```
lock(a)
B_s sender = invalid
signal(conda)
unlock(a)

# Send
# 3-way handshake
lock(a)
while(not Is_Empty(BUF)) wait(conda, a)
unlock(a)

lock(b)
A_s receiver = B
signal(condb)
unlock(b)

lock(a)
while(B_s sender /= B) wait(conda, a)
unlock(a)

# actual send
lock(b)
store message to BUF
Is_Empty(BUF) = False
signal(condb)
unlock(b)

# 3-way handshake
lock(a)
while(not Is_Empty(BUF)) wait(conda, a)
unlock(a)

lock(b)
A_s receiver = invalid
signal(condb)
unlock(b)

lock(a)
while(B_s sender = A) wait(conda, a)
unlock(a)
```

For Pthread cannon's version, I implement a non-blocking version send&recv for cyclic communication to avoid to deadlock, I use barrier to simulate the `MPI_Wait`. For instance, every thread along the same row will wait until same row (and column) cyclic communication is done. The non blocking send&recv modify the 3-way handshake (as pesudocode above) to avoid the deadlock situation by only checking whether current buffer space's availability.

Each alternative implementation of Pthread program also adding with OpenMP `omp for schedule(dynamic)` directive to normal local matrix multiplication and expect to speed up the program further.

Part B

Parallel Sample Sort

First of all, master process will read in whole un-sorted list and send the list length to all processes, program will only run sample sort when the input size is larger than threshold, which is `(2 * comm_size - 1) * comm_size` (I use this threshold since every process will hold a sample and generate local splitter with size `comm_size - 1`, hence each process should hold at least `2 * comm_size - 1` sample, where we have `comm_size` processes). If the global list size is less than threshold, process 0 will simply run a local quick sort then exit.

When the input size is sufficient, sample sort is performed. Consider we have global unsorted list with size `n`, and `p` number of processes, process 0 will randomly select `p` set of sample, each with size `2*p - 1`, then distributed to every process. When process receive the sample `s`, it will first run a local quick sort, pick local splitters by considering element from sorted sample with index `1,3,5,...,2*p - 3`, with size `p - 1`. Until the local spitter is generated, I use `MPI_Allgather` to gather the local splitters, and every process will then hold all local splitters from all process, they can do the local quick sort again and generate the global splitters individually.

When global splitter is generated, every process can now determine which bucket is belong to them, process 0 will then use the global splitter information to prepare the local bucket for each process by distributing global list data, then send it to process with corresponding bucket region. After that, process holds the local bucket can perform final local sort. Until all the processes are done, process 0 can perform a `MPI_Gatherv` to collect the local sorted bucket in the order of process rank, finalize the program.

As alternative implementation, instead of distributing the global list by process 0 after the global splitter is set, I distributed the global list before the local splitter is generated, in this case, process is not required to wait until process 0 send the randomly selected sample to do the local splitter computation, but they can generate the sample themselves within the distributed global sub-list. The global list distribution can be done in O(logn) via `MPI_Scatterv`, (`v` since every process may not hold same number of element, as the input data is not restricted on size). When processes hold a portion of global list, they can do the local sampling, generate local splitter then use `MPI_Allgather` to share the splitter, and come up global splitter independently.

Since every process holds a portion of data and global splitter, they can use binary search insertion to insert the local portion into specific local buckets, then use `MPI_Alltoallv` to share with all processes. As every process receive all the data that belongs to its bucket, they can do local quicksort, and finally gathered by master process, finalize the program. The variation rountine is followed with figure 1[1] below.

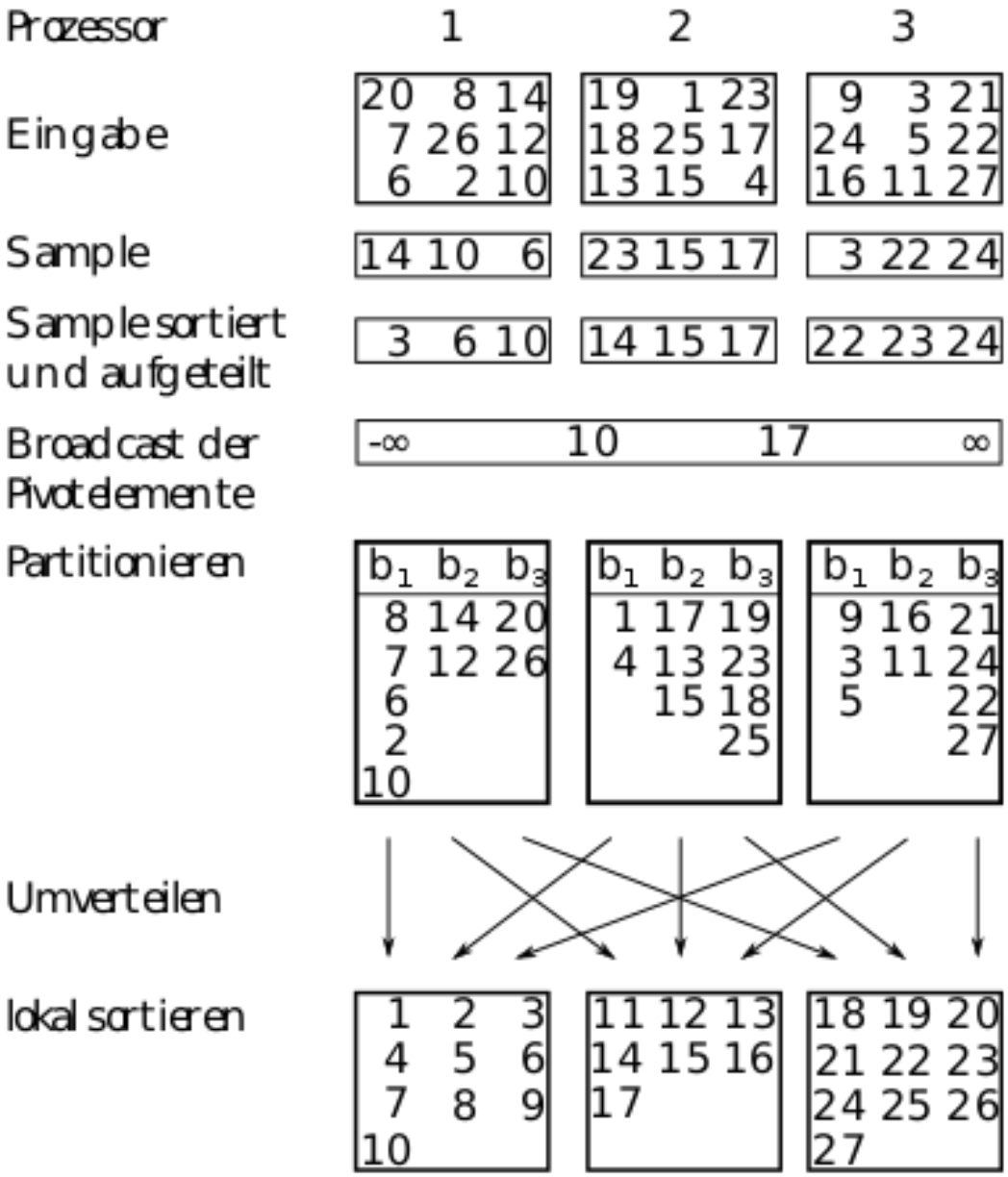

figure 1. Sample sort

Reference

[1] Wikimedia Commons, Example of parallel Samplesort on processors and an oversampling factor of . https://en.wikipedia.org/wiki/Samplesort#/mediaFile:Parallelersamplesort.svg