Experimenting sequential algorithms to optimize the scheduling of multidimensional array processing in Dask

1st Timothée Guédon
Department of Computer Science and Software Engineering
Concordia University
Montreal, Quebec, Canada
t_guedon@encs.concordia.ca

2nd Tristan Glatard
Department of Computer Science and Software Engineering
Concordia University
Montreal, Quebec, Canada
tristan.glatard@concordia.ca

3rd Valérie Hayot-Sasson
Department of Computer Science and Software Engineering
Concordia University
Montreal, Quebec, Canada
email address or ORCID

***Abstract*—This document is a model and instructions for LaTeX. This and the IEEEtran.cls file define the components of your paper [title, text, heads, etc.]. *CRITICAL: Do Not Use Symbols, Special Characters, Footnotes, or Math in Paper Title or Abstract.**

***Index Terms*—component, formatting, style, styling, insert**

INTRODUCTION

This document is a model and instructions for LaTeX. Please observe the conference page limits.

A. Big Data softwares in Python

Spark Dask

B. Selecting a Big Data software

C. Focusing on Dask

Scheduler selection API selection Which algorithm can be implemented? Dask constraints and requirements

RELATED WORK

D. The splitting and merging problem

E. Sequential algorithms

METHODS

F. Experiment 1 - assessing if there is space for improvement

G. Implementation details of the "Clustered" strategy

Our goal is to implement the most minor modification as possible that will provide the clustered strategy to the user. We also want our modification to be easy to use. We could try to modify internally the Dask software but a smarter way to modify the graph would be to use an optimization function. As we will see, the modification of the Dask graph by the optimization function alone would not be safe to use. That is why we need to adjust a little bit the scheduler behavior, too.

1) Optimization function: Our optimization strategy consists in modifying the Dask graphs such that we force the scheduler to load contiguous data blocks in one shot, as one buffer. A parameter has to be selected along with the use of the optimization function: the buffer size. On the one hand, the buffer size is bounded by the random access memory size available. On the other hand, the user could also want to further bound the buffer size depending on the application. For example if the application is doing a cartesian product between two vectors the memory size used by the application will explose: We will switch from $O(n)$ memory consumption to $O(n^2)$.

Before stepping into the implementation details, we need to define what an "array proxy" is. We call it array proxy inspired by the Python library nibabel. Unlike nibabel however, our proxy is not an object which can load data but it is a link to a block of the data. Going through the scheduler, this task will load the block it represents, and this block will be used for further operations.

Internally, the optimization function is applied on the dask graph. We proceed in two steps: finding the "array proxies" that are used in the graph, and modifying the dask graph accordingly to mutualize the data loading by the use of buffers.

Finding the used array proxies

At the end of the search, we end up with $utility\_dicts$ contains the following dictionaries:

- $array\_to\_original$: map array proxy names to the original arrays they represent
- $original\_array\_chunks$: map original array names to their chunk shapes
- $original\_array\_shapes$: map original array names to their shape

- $original\_array\_blocks\_shape$: map original array names to their shape in terms of blocks, i.e. the number of blocks they have in each dimension.

Creating the buffers

At this point we found the used array proxies. The idea here is to create new nodes to replace the array proxies by buffers. We can do this because the scheduler tries to leverage the data that has been loaded using a LIFO strategy: When some data has been loaded the scheduler will run all the tasks that uses this data. By doing so the data is loaded only once. By loading a large buffer in one time, we will not break this rule. We will load more data at the same time that can then be used by all the tasks that need it. Firstly, this would have been a problem if the data would be loaded several times. Secondly, it *should* maximize the parallelism by enabling the computation of many tasks at the same time. This way, we reduce the probability of seeing tasks waiting for the data loading, although the beginning of computation will be delayed by the very first data loading task.

The idea is to create buffers following the clustered strategy from Hayot-Sasson et al. As we said, we want to create buffers by merging tasks that load contiguous data blocks. We therefore need to know whiwh block is contiguous with each other on the disk. To that aim we need an algorithm to convert indices in n-dimension into indices in 1 dimension for each logical block of the array that is to be loaded by a loading task i.e. which is reprensented by an array proxy. This function takes the order in which the data is stored on disk: column-major or row-major.

Given a list of used proxies, we first need to convert the indices into numeric indices using the algorithm above. We then concatenate the blocks into buffers given a maximum buffer memory size. According to the paper of reference, we should prevent the overlaping of block slices and block rows. Algorithm 1 below show how we process to concatenate blocks loading. In the following algorithm, $blocks\_used$ designates the list of integer indices of the used array proxies found in the previous section.

As you can see, Algorithm 1 uses three sub-algorithms: $start\_new\_buffer$ (Algorithm 2), $concat\_rows$ (Algorithm 3) and $concat\_slices$. $start\_new\_buffer$ evaluates if we should append the current block being processed to the current buffer or if we should save the current buffer and start a new one. For example, if the buffer size has been reached, we cannot add more blocks to it. This suppose to have some information like the size of one data block in bytes, the size of the buffer in bytes etc. The $concat\_slice$ algorithm is of the same spirit than $concat\_rows$. In the following algorithm, MAX_BB designates the maximum number of blocks that can be load in one buffer and NB_BR designates the number of blocks per block row.

2) Scheduler constraint on memory: **The idea** : Now that we have bufferized the data loading according to the clustered strategy, there is still a problem remaining: the scheduler is a threaded one. It implies that there will be several buffers loaded in "parallel" threads, although the RAM is shared by all the threads. In the clustered strategy paper, we choose the buffer size according to the amount of RAM available. If we do that while usign several threads to load the data, we end up loading several buffers at the same time with each buffer reaching the size of the maximum memory available. Therefore, our buffering will not work unless we force the scheduler to load the buffers sequentially. We need to constrain only the buffer loading tasks and not the other one in order not to loose the parallelism for computation.

Algorithm 1 $create\_buffers(blocks\_used)$

```
blocks ← sorted(blocksused)
buffers ← list()
curr_buffer ← list()
prev_block ← None
while length(blocks) > 0 do
  b ← blocks.pop(0)
  if start_new_buffer()andnotempty(curr_buffer) then
    buffers.append(curr_buffer)
    curr_buffer ← list()
    prev_block ← None
  end if
  curr_buffer.append(b)
  prev_block ← b
end while
if buffer_size >= block_row_size then
  buffers ← concat_rows(buffers)
  if buffer_size >= block_slice_size then
    buffers ← concat_slices(buffers)
  end if
end if
return buffers
```

Algorithm 2 $start\_new\_buffer(b, prev\_block, MAX\_BB, NB\_BR)$

```
if length(curr_buffer) == MAX_BB then
  return True
end if
if b! = prev_block + 1 then
  return True
end if
if prev_block! = 0 and prev_block%NB_BR then
  return True
end if
return False
```

Dask's threaded Scheduler : Let us consider the so-called "Asynchronous Shared-Memory Scheduler for Dask Graphs". From the docstrings of Dask we find that the state of the scheduler is divided in two collections of data structures regarding if they are "Constant states" or "Changing states". "Changing states" are divided into two categories: "data-related" states and "job-related" states.

Algorithm 3 $concat\_rows()$

$merged\_buffer \leftarrow list()$
for $buffer in buffers$ **do**
  if $is\_complete\_row(buffer)$ **then**
    $start\_new\_buffer \leftarrow False$
    if $overlap\_slices(curr\_buffer, buffer)$ **then**
      $start\_new\_buffer \leftarrow True$
    end if
    if $length(curr\_buffer) + length(buffer) > buffer\_size$ **then**
      $start\_new\_buffer \leftarrow True$
    end if
    if $start\_new\_buffer$ **then**
      $merged\_buffers.append(curr\_buffer.copy())$
      $curr\_buffer \leftarrow list()$
    end if
    $curr\_buffer.concat(buffer)$
  else
    $merged\_buffers.append(buffer)$ {Do nothing}
  end if
end for

"Constant states"

- Dependencies: $x : [a, b, c]$ a,b,c, must be run before x
- Dependents: $a : [x, y]$ a must run before x or y

Data-related states

- Cache: available, concrete data. (Dictionary)
- Released: Data that have been loaded, used and deleted and that we no longer need.

Job-related states

- Ready: FIFO stack of ready-to-run tasks.
- Running: Set of tasks that are being processed.
- Finished: Set of finished tasks.
- Waiting: Real time equivalent of dependencies.
- Waiting data: Data that is available for use. Real time equivalent of dependents.

Modification of the scheduler for safe buffering : Our goal is to load one buffer task at a time. To do this, we add two sets: $loading\_data$ and $delayed\_loading$. In the first set we store the keys of the data loading tasks including one buffer task together with the memory size of the data being loaded. After having identified a task as being a buffer task, the pseudo-code to whether or not to launch this task is very simple as shown in Algorithm 4: We predict if the amount of data stored in RAM will be too big after having loaded the buffer task in question. If so, we delay its processing by putting it into the $delayed\_loading$ set. Every time a buffer task has finished its processing, we look at the byte size of the first buffer task waiting in $delayed\_loading$ to see if it can be loaded.

In case the buffering task would not be desirable by any application, it is disactivated by default. To activate the scheduler optimization using clustered strategy, one can simply set the $io-optimizer.scheduler\_opti$ configuration setting to $True$.

Algorithm 4 $main\_scheduling\_function()$

while $size(ready) > 0$ **do**
  $next\_task \leftarrow ready.pop(0)$
  if $is\_buffer(next\_task)$ **then**
    if $launch\_buffer\_task(next\_task)$ **then**
      $run(next\_task)$
    else
      $delayed\_loading.add(next\_task)$
    end if
  else
    $run(next\_task)$
  end if
end while

Algorithm 5 $launch\_buffer\_task(next\_buffer\_task)$

$mem\_print \leftarrow size(cache) + size(waiting\_data) + size(loading\_data)$
if $already\_loading()$ {We are supposed to load only one buffer at a time} **then**
  return $False$
end if
if $mem\_print + size(next\_buffer\_task) > memory\_size$ **then**
  return $False$
end if
return $True$

H. Experiment 2 - Assessing the splitting performance of our implementation

I. Experiment 3 - Assessing the reproducibility with other file formats

RESULTS

DISCUSSION

CONCLUSION

ACKNOWLEDGMENT

Todo

REFERENCES

Todo