Contents

1 ORNL - Coding challenge **1**

1.1 Steps to install and run 1

1.2 Problem 1: Extracts digits in string 1

1.3 Problem 2: Remove duplicates in order 2

1.4 Problem 3: Lookup ids 3

1.5 Question B: How might you extend your solution to process tens of millions of elements in the list of IDs? The list of specialities? Both? 4

1.5.1 Processing millions of lists of IDs in parallel 4

1.5.2 Removing duplicates in parallel 5

1.5.3 Processing millions of specialties in parallel 8

1.5.4 Millions of (id, specialty) pairs and millions of ids . . . 9

1.6 Unit tests 10

1 ORNL - Coding challenge

1.1 Steps to install and run

- Install fortran package manager from - `https://fpm.fortran-lang.org/en/install/index.html#building-from-source`

```
git clone https://github.com/fortran-lang/fpm
cd fpm
sh install.sh
```

- Change into this directory and run 'make test'

```
# This should download needed dependencies and run the tests
make test
```

1.2 Problem 1: Extracts digits in string

- Time Complexity - O(C); where C is the length of the characters
- Space Complexity - O(C)

```
pure function extract_digits(id) result(id_clean)
   character(len=*), intent(in) :: id
   character(len=len(id)) :: numbers
```

```
   character(len=:), allocatable :: id_clean
   integer :: i, pos
   integer :: count
   ! Init
   count = 0
   pos = 1
   ! Filter digits
   do i = 1, len(id)
      if (is_digit(id(i:i))) then
         pos = count + 1
         numbers(pos:pos) = id(i:i)
         count = pos
      end if
   end do
   id_clean = numbers(1:count)
end function extract_digits
```

1.3 Problem 2: Remove duplicates in order

- Time complexity - O(N)
  - Open addressing Hashmap with linear probing was used; Insertion time - O(N)
  - Amortized lookup time - O(1); Load factor of 0.5625 was used to shorten probe length
  - Hash function - fnv1a
  - During insertion, we can build a list that filters duplicates
- Space complexity - $O(N/\text{load}_{\text{factor}})$; $\text{load}_{\text{factor}} = 0.5625$
  - Load factor is a configurable value.

```
subroutine remove_duplicates(xs, ys)
   ! -- declarations omitted for brevity --
   ! Init
   count = 0
   allocate (temp, source=xs)
   slot_bits = exponent((size(xs)/load_factor))
   slot_bits = max(default_bits, slot_bits)
   call map%init(fnv_1_hasher, slots_bits=slot_bits)
```

```
   ! filter duplicates using hashmap
   do i = 1, size(xs)

      call set(key, xs(i))
      call map%map_entry(key, conflict=conflict)
      if (.not. conflict) then
         pos = count + 1
         temp(pos) = xs(i)
         count = pos
      end if
   end do
   ys = temp(1:count)
end subroutine remove_duplicates
```

1.4 Problem 3: Lookup ids

- Time complexity
  - O(N) to build mapping between ids and specialties;
  - O(2M) - to filter duplicates(while retaining order) and specialty lookup
- Space complexity
  - $O(N/load_{factor})$ for building specialties hashmap
  - $O(M/load_{factor})$ for filtering duplicates in ids with a hashmap

```
subroutine lookup_ids(specialty_ids, specialty_names, ids, specialties)
   ! -- declarations omitted for brevity --
   ! Init hashmap
   slot_bits = exponent(size(specialty_ids)/load_factor)
   slot_bits = max(default_bits, slot_bits)
   call map%init(fnv_1_hasher, slots_bits=slot_bits)

   ! Fill the map with (id, specialty)
   do i = 1, size(specialty_ids)
      call set(key, get_digits(specialty_ids(i)))
      allocate (data, source=char(specialty_names(i)))
      call set(other, data)
      call map%map_entry(key, other, conflict=conflict)
      deallocate (data)
```

```
   end do

   ! Cleanup ids
   allocate (ids_digits(size(ids)))
   do i = 1, size(ids)
      ids_digits(i) = extract_digits(ids(i))
   end do

   ! Remove duplicates
   call remove_duplicates(ids_digits, unique_ids)

   ! Build int ids
   unique_ids_int = to_integer(unique_ids)

   ! Lookup specialties
   do i = 1, size(unique_ids_int)
      call set(key, get_digits(unique_ids_int(i)))
      call map%key_test(key, is_present)
      if (is_present) then
         call map%get_other_data(key, other)
         call get(other, data)
         select type (data)
         type is (character(len=*))
            specialties(i) = data
         class default
            specialties(i) = "N/A"
         end select
      else
         specialties(i) = "N/A"
      end if
   end do
end subroutine lookup_ids
```

1.5 Question B: How might you extend your solution to process tens of millions of elements in the list of IDs? The list of specialities? Both?

1.5.1 Processing millions of lists of IDs in parallel

- Let N be the total length of list of IDs

- Let M be the number of nodes available to process the list
- We can load each node with (N / M) of elements from the list
- For example: ["7-231", "1236", "4567", "7231", "8901", "89-01"] could be divided as shown below int0 three nodes.
- Note: The order of elementss is retained in the order of the nodes. i.e. Node 1 gets the first N/M chunk, Node 3 gets the third N/M chunk

Node1	Node2	Node3
7-231	4567	8901
1236	7231	89-01

- To cleanup the ids in parallel, we can run 'extract$_{\text{digits}}$' function on each node. After cleanup, the nodes look like this

Node1	Node2	Node3
7231	4567	8901
1236	7231	8901

1.5.2 Removing duplicates in parallel

- We can remove the duplicates in M nodes in M iterations
- In each iteration m,
  - node m contains the unique list of ids
  - All M nodes, copy the unique list of ids from node m and filter the duplicates in their memory
  - The ids in node m are ready to be printed/saved.

Iteration: 1: All nodes remove duplicates in their own list. Contents of node1 contain unique ids. Contents in node2, node3 could still have duplicates

Node1	Node2	Node3
7231	4567	8901
1236	7231	

Iteration 2: Nodes 2 and Node3 can copy the list from node 1 and filter duplicates from the list in memory. Node 2 contains the unique list of ids that is ready to be printed/saved

Node1	Node2	Node3
7231	4567	8901
1236		

Iteration 3: Node 3 copies the list from node2 and filters duplicates from the list in memory. Contents of node3 can now be printed/saved

Node1	Node2	Node3
7231	4567	8901
1236		

Final list of unique ids printed by traversing nodes in order: ["7231", "1236", "4567", "8901"]

Since this distributed duplicate removal step is an important piece of the proposed solution, I implemented it and measured it's performance for problem sizes 600K,3.6M,6M,12M,24M and for $\text{node}_{\text{counts}}$ 1,2,4,6. For greater problem sizes, single node runs out of memory, but nodes that have data distributed can keep processing higher amounts

All execution times were averaged over 3 runs. To reproduce the results, you can run

```
# Follow instructions from here. http://www.opencoarrays.org/
# Install brew for mac/linux
brew install opencoarrays
export FPM_FC=<path_to_caf>
fpm build --profile=release
bash scripts/runner.sh
```

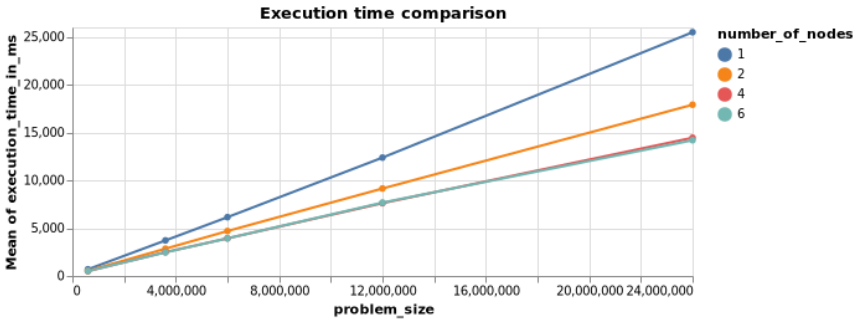
The results provided here were measured in

```
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         39 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  12
  On-line CPU(s) list:   0-11
Vendor ID:               GenuineIntel
  Model name:            Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz
```

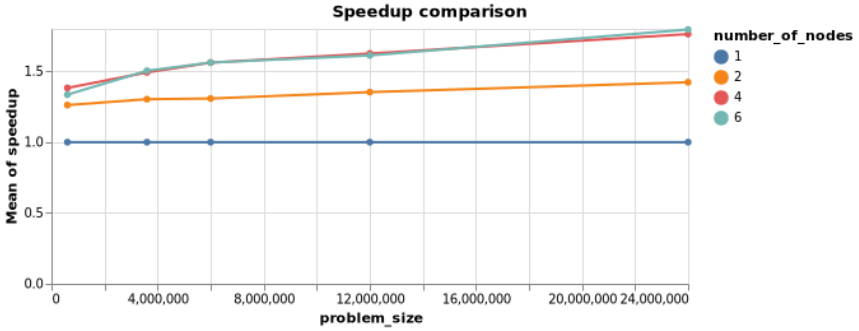
To generate visualizations, install vega-cli

```
npm i vega-cli
<path_to_vega_cli>/bin/vg2png --base <full_absolute_path_to_this_directory> recipe/exe
<path_to_vega_cli>/bin/vg2png --base <full_absolute_path_to_this_directory> recipe/spe
```

Execution time comparison:



Speedup comparison



As you can see from these charts that as size increases the distributed implementation is faster. But the speedup is not linear. As number of nodes increases upto 6, the speedup achieved is only 1.5.

1. Runtime analysis

   - Number of iterations for the distributed data removal is equal to the number of nodes
   - In each iteration, we pass through the entire list O(N). Since we do this for M iterations, total number of times the list gets examined is O(M x N)
   - In each iteration m, we also copy O(N/M) elements from one node to all (m, M) nodes
     - In the first iteration, there is a many to one copy of O(N/M) elements from node 1 to all (m-1) nodes
     - In the second iteration, there is a many to one copy of O(N/M) from node 2 to all (m $>$ 2) nodes

    - In each iteration the number of nodes involved in the many-to-one copy transfer shrinks. (Starting from m-1 nodes in the first iteration to 0 in the last iteration)
  - In each iterations, node m contains the unique list of elements. This can be saved in parallel at the end using a distributed file system or the elements can be copied to node 1 and saved (as is common to do so, in HPC systems)

1.5.3 Processing millions of specialties in parallel

- The millions of specialties and their corresponding ids can be hosted in parallel.
- Since the (id, specialty) pair can be assumed to be unique, we can use a simple routing function to route any (id, specialty) pair to a node m (in a cluster of M nodes)
- Example distribution function: node_{id} = modulo(id, M)
- Example (id, specialty) pairs

```
[
(7231, "algorithms"),
(2134, "security"),
(4532, "compilers"),
(3000, "journalism")
]
```

- After applying the routing function, node_{id} = module(id, M). Assuming M = 3

Node0	Node1	Node2
(3000, "journalism")	(7231, "algorithms") (2134, "security")	(4532, "compilers")

- Now the same routing function can be applied on the list of IDs before lookup and routed to the appropriate node for the specialties lookup
- Each node m, can also use a separate hashmap on the id facilitating faster specialty lookup

1. Runtime analysis

- The (id, specialty) pair can be processed in parallel in M nodes during the table construction in memory.
  - Runtime: O(N/M)
- Communication between nodes is dependent on the skewness of the data. If there are lot more ids that end with 1 after applying the routing function, then node 1 will get a lot of (id,specialty) pairs to host
- This design works well only when the data/routing function balances the data equally among M nodes

1.5.4 Millions of (id, specialty) pairs and millions of ids

- We can use the design above to host millions of (id, specialty) pairs in M nodes
- We can use the "distributed duplicate removal" design above to filter ids of duplicates. This retains the order of inputs ids while removing duplicates
- Example specialties table

Node0	Node1	Node2
(3000, "journalism")	(7231, "algorithms")	(4532, "compilers")
	(2134, "security")	

- Example unique ids in memory. Note that the routing function hasn't been applied to the ids yet. So, the ids in node m may not find the specialties in node m and may have to communicate with another node to find it's specialty

Node0	Node1	Node2
7231	2134	3000
4532		

- For each iteration m in range(0, M)
  - node m - acts as the sender
  - all nodes in range (m, M) nodes act as the receiver
  - Get all ids in node m, and apply the routing function. Send lookup request to the corresponding node. Receive specialty name from the

    * Example: In node 0
    * routing$_{\text{fn}}$(7231) -> node1 -> lookup$_{\text{specialty}}$(incoming$_{\text{id}}$) -> return$_{\text{result from node1 to node0}}$

- After M iterations, the specialties in M nodes look as below

Node0	Node1	Node2
algorithms	security	journalism
compilers		

- These results can be printed in order by traversing nodes 0 to M-1

1.6 Unit tests

- Unit tests are in file test/check.f90. You can run them with 'make test'

```
arul@arul-Serval ~/d/ornl-assignment (main)> fpm test
ornl-assignment.f90                          done.
parallel-work.f90                            done.
check.f90                                    done.
libornl-assignment.a                         done.
main.f90                                     done.
ornl-assignment                              done.
check                                        done.
[100%] Project compiled successfully.
# Testing: extract_digits_suite
  Starting random_string(length=0) ... (1/4)
       ... random_string(length=0) [PASSED]
  Starting random_string(length=5) ... (2/4)
       ... random_string(length=5) [PASSED]
  Starting random_string(length=50) ... (3/4)
       ... random_string(length=50) [PASSED]
  Starting random_string(length=500) ... (4/4)
       ... random_string(length=500) [PASSED]
# Testing: remove_duplicates_suite
  Starting rm_duplicate_string(list_size=2) ... (1/3)
       ... rm_duplicate_string(list_size=2) [PASSED]
  Starting rm_duplicate_string(list_size=10) ... (2/3)
       ... rm_duplicate_string(list_size=10) [PASSED]
  Starting rm_duplicate_string(list_size=100) ... (3/3)
       ... rm_duplicate_string(list_size=100) [PASSED]
```

```
# Testing: specialties_lookup
  Starting ids_string_to_int ... (1/7)
       ... ids_string_to_int [PASSED]
  Starting specialties_lookup(table_size=1, ids_list=10) ... (2/7)
       ... specialties_lookup(table_size=1, ids_list=10) [PASSED]
  Starting specialties_lookup(table_size=10, ids_list=10) ... (3/7)
       ... specialties_lookup(table_size=10, ids_list=10) [PASSED]
  Starting specialties_lookup(table_size=100, ids_list=10) ... (4/7)
       ... specialties_lookup(table_size=100, ids_list=10) [PASSED]
  Starting specialties_lookup(table_size=1, ids_list=1) ... (5/7)
       ... specialties_lookup(table_size=1, ids_list=1) [PASSED]
  Starting specialties_lookup(table_size=1, ids_list=0) ... (6/7)
       ... specialties_lookup(table_size=1, ids_list=0) [PASSED]
  Starting specialties_lookup(table_size=1, ids_list=100) ... (7/7)
       ... specialties_lookup(table_size=1, ids_list=100) [PASSED]
```