COP5615 - Project 1

1. Group Members

Prajwal Dondiganahalli Prakash (UFID 04464906)
Rishabh Khanna (UFID 13399251)

Steps to run:

1. Unzip `khanna_dondiaganahalliprakash.zip`
   `$ unzip khanna_dondiganahalliprakash.zip`
2. Change directory to the project folder in the terminal.
   `$ cd khanna_dondiganahalliprakash`
3. Run the script as
   `$ mix run proj1.exs [lower_limit] [upper_limit]`
   Example:
   `$ mix run proj1.exs 100000 200000`

2. Number of worker actors created

Each worker handles 100 numbers. Therefore for a range of 100000 numbers (between 100000 and 200000), the number of worker actors created was **1000**.

3. Size of the worker actor

The following timings were obtained for different sizes of worker actors (executing the program for the range 100000 to 200000):

Size of worker actors	**Time - real**	**Time - user**	**Time - sys**	**Parallelism**
1	0m14.177s	0m26.459s	0m1.253s	1.958
5	0m11.851s	0m39.509s	0m0.653s	3.388
10	0m10.649s	0m39.422s	0m0.463s	3.754
20	0m10.748s	0m38.801s	0m0.383s	3.645
30	0m10.414s	0m38.800s	0m0.357s	3.760
50	0m11.395s	0m37.490s	0m0.393s	3.324
100	0m10.039s	0m37.186s	0m0.307s	3.735
200	0m10.147s	0m37.688s	0m0.311s	3.745

The above runs were performed one after to simulate a consistent load on the system.

We chose the size of the worker actors to be **100** for our computer (though smaller worker sizes resulted in more parallelism) as:

- The computer was able to handle 100 numbers in each actor.
- If we choose a smaller size, there would be more actors and the process limit would be met for a larger range.
- If we choose a smaller size, more time would be spent in communication (ideally we want to spend more time in computation).

4. Sample Output

```
$ mix run proj1.exs 100000 200000
102510 510 201
104260 401 260
105210 501 210
105264 516 204
105750 705 150
108135 801 135
110758 701 158
115672 761 152
116725 725 161
117067 701 167
118440 840 141
120600 600 201
123354 534 231
124483 443 281
125248 824 152
125433 543 231
125460 615 204 510 246
125500 500 251
126027 627 201
126846 486 261
129640 926 140
129775 725 179
131242 422 311
132430 410 323
133245 423 315
134725 425 317
135828 588 231
135837 387 351
136525 635 215
```

```
136948 938 146
140350 401 350
145314 414 351
146137 461 317
146952 942 156
150300 501 300
152608 608 251
152685 585 261
153436 431 356
156240 651 240
156289 581 269
156915 951 165
162976 926 176
163944 414 396
172822 782 221
173250 750 231
174370 470 371
175329 759 231
180225 801 225
180297 897 201
182250 810 225
182650 650 281
186624 864 216
190260 906 210
192150 915 210
193257 591 327
193945 491 395
197725 719 275
```

5. Running time for the above sample output

```
$ time mix run proj1.exs 100000 200000
…

real  0m9.925s
user  0m36.048s
sys   0m0.351s
```

Ratio of CPU time to real time = (36.048 + 0.351) / 9.925 = **3.667**

6. Solving larger problems

We could solve the following problem (100,000 to 1,000,000)

```
$ time mix run proj1.exs 100000 1000000
829696 926 896
841995 945 891
939658 986 953

real  1m27.608s
user  5m7.971s
sys   0m1.461s
```

We were also able to execute the program for the following range (though the computation required for the numbers between 1000000 and 6000000 is relatively less than for numbers from 100000 to 999999).

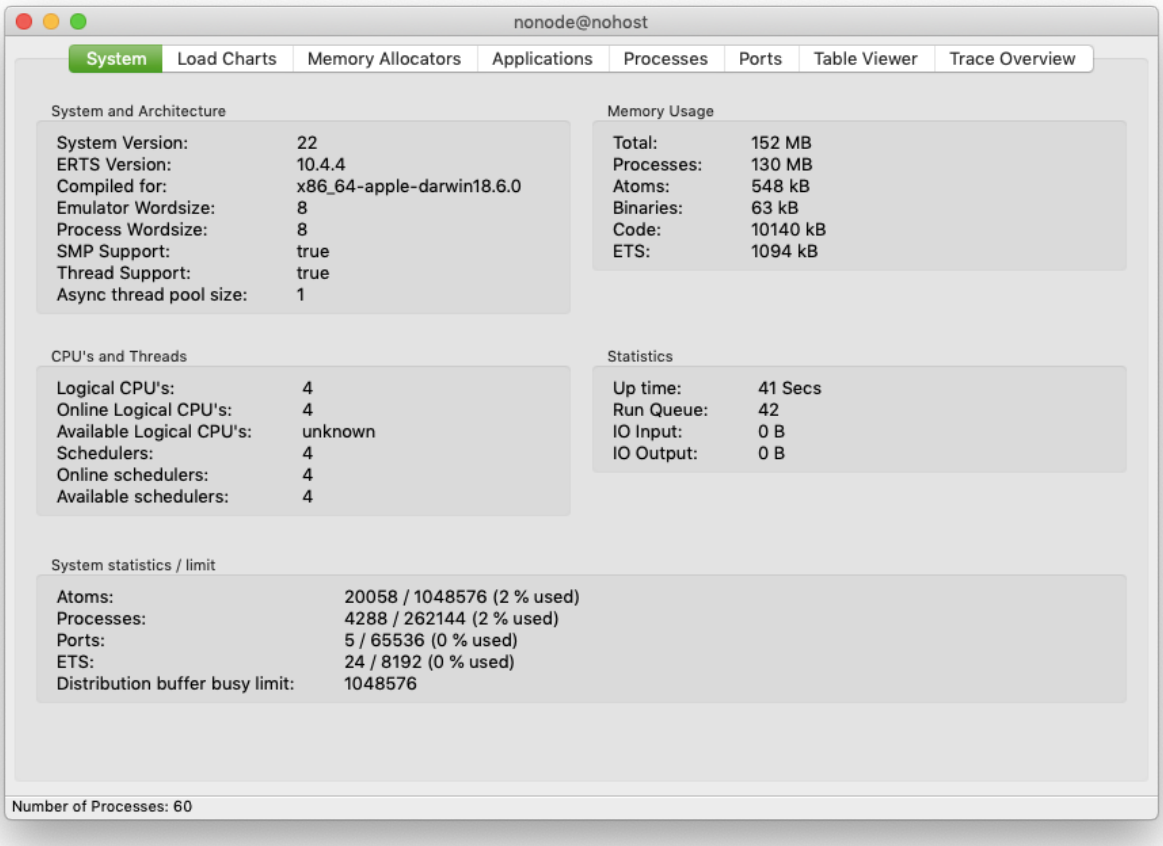
```
$ time mix run proj1.exs 100000 6000000
...
809964 906 894
815958 951 858
829696 926 896
841995 945 891
939658 986 953

real  1m25.514s
user  5m18.036s
sys   0m1.684s
```

The last vampire number we could find was 939658 (when we executed the program for the range 100000 to 6000000)

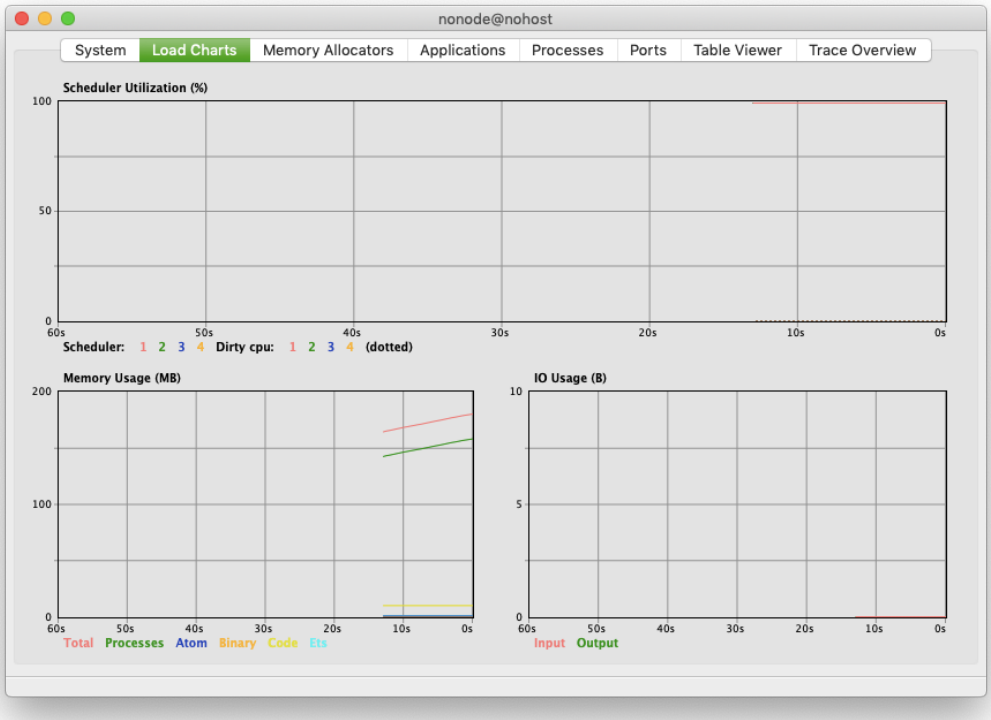
7. Observer Screenshots

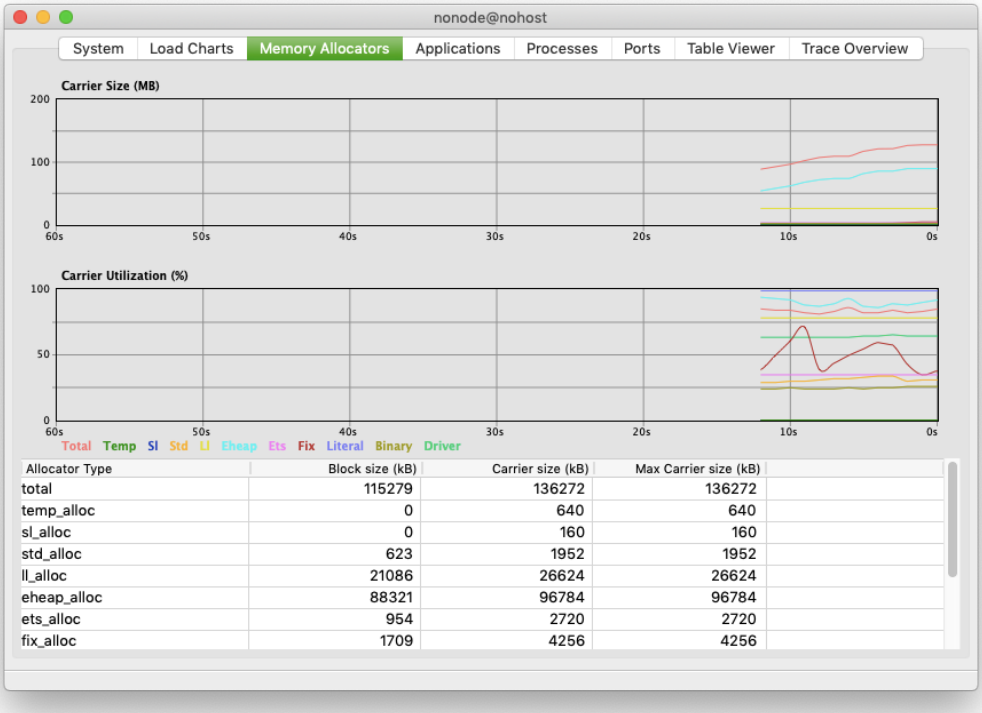
The following screenshots were captured during the execution of the command `mix run proj1.exs 200000 1000000`.

Note: All runs were performed on 1.8 GHz Intel Core i5 (i5-5350U) with 2 physical cores and 4 threads.

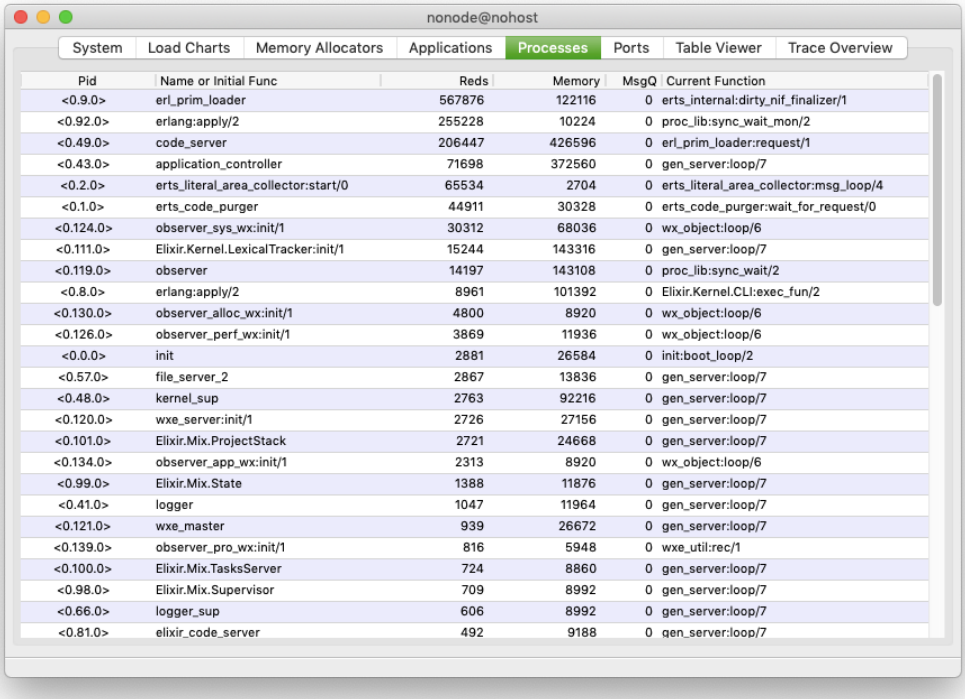


The process count goes from 60 to 7909 as the execution proceeds.





Allocator Type	Block size (kB)	Carrier size (kB)	Max Carrier size (kB)
total	115279	136272	136272
temp_alloc	0	640	640
sl_alloc	0	160	160
std_alloc	623	1952	1952
ll_alloc	21086	26624	26624
eheap_alloc	88321	96784	96784
ets_alloc	954	2720	2720
fix_alloc	1709	4256	4256


nonode@nohost
System
Load Charts
Memory Allocators
Applications
Processes
Ports
Table Viewer
Trace Overview
Pid Name or Initial Func Reds Memory MsgQ Current Function
<0.9.0> erl_prim_loader 567876 122116 0 erts_internal:dirty_nif_finalizer/1
<0.92.0> erlang:apply/2 255228 10224 0 proc_lib:sync_wait_mon/2
<0.49.0> code_server 206447 426596 0 erl_prim_loader:request/1
<0.43.0> application_controller 71698 372560 0 gen_server:loop/7
<0.2.0> erts_literal_area_collector:start/0 65534 2704 0 erts_literal_area_collector:msg_loop/4
<0.1.0> erts_code_purger 44911 30328 0 erts_code_purger:wait_for_request/0
<0.124.0> observer_sys_wx:init/1 30312 68036 0 wx_object:loop/6
<0.111.0> Elixir.Kernel.LexicalTracker:init/1 15244 143316 0 gen_server:loop/7
<0.119.0> observer 14197 143108 0 proc_lib:sync_wait/2
<0.8.0> erlang:apply/2 8961 101392 0 Elixir.Kernel.CLI:exec_fun/2
<0.130.0> observer_alloc_wx:init/1 4800 8920 0 wx_object:loop/6
<0.126.0> observer_perf_wx:init/1 3869 11936 0 wx_object:loop/6
<0.0.0> init 2881 26584 0 init:boot_loop/2
<0.57.0> file_server_2 2867 13836 0 gen_server:loop/7
<0.48.0> kernel_sup 2763 92216 0 gen_server:loop/7
<0.120.0> wxe_server:init/1 2726 27156 0 gen_server:loop/7
<0.101.0> Elixir.Mix.ProjectStack 2721 24668 0 gen_server:loop/7
<0.134.0> observer_app_wx:init/1 2313 8920 0 wx_object:loop/6
<0.99.0> Elixir.Mix.State 1388 11876 0 gen_server:loop/7
<0.41.0> logger 1047 11964 0 gen_server:loop/7
<0.121.0> wxe_master 939 26672 0 gen_server:loop/7
<0.139.0> observer_pro_wx:init/1 816 5948 0 wxe_util:rec/1
<0.100.0> Elixir.Mix.TasksServer 724 8860 0 gen_server:loop/7
<0.98.0> Elixir.Mix.Supervisor 709 8992 0 gen_server:loop/7
<0.66.0> logger_sup 606 8992 0 gen_server:loop/7
<0.81.0> elixir_code_server 492 9188 0 gen_server:loop/7