Exercises for Memory-Efficient Computing

In-memory computations: Numexpr as an accelerator of NumPy expressions

Initially, we are going to see how to optimize the computation of expressions that fit well in main memory. For the exercises in this sections we will mainly use the `poly1.py` script.

Warning: For this part, please remember to login into the remote server for performing the computations.

1. Use script `poly1.py` to check how much time it takes to evaluate the next polynomial:

   ```
   y = .25*x**3 + .75*x**2 - 1.5*x - 2
   ```

   with x in the range [-1, 1], and with 10 millions points.

   - Set the *what* parameter to "numexpr" and take note of the speed-up versus the "numpy" case. Why do you think the speed-up is so large?

2. The expression below:

   ```
   y = ((.25*x + .75)*x - 1.5)*x - 2
   ```

   represents the same polynomial than the original one, but with some interesting side-effects in efficiency. Repeat the computation for numpy and numexpr and get your own conclusions.

   - Why do you think numpy is performing much more efficiently with this new expression?
   - Why the speed-up in numexpr is not so high in comparison?
   - Why numexpr continues to be faster than numpy?

3. The C program `poly.c` does the same computation than above, but in pure C. Compile it like this:

   ```
   gcc -O3 -o poly poly.c -lm
   ```

   and execute it.

   - Why do you think it is more efficient than the above approaches?

4. Be sure that you are on a multi-processor machine and repeat the last computation in poly1.py (using numexpr) but increasing the number of threads one by one (use the *ne.set_num_threads()* function).

   - How the efficency scales?
   - Why do you think it scales that way?
   - How peformance compares with the C computation?

5. The expression:

   ```
   y = sin(x)**2 + cos(x)**2
   ```

   contains the sine and cosine, transcental functions that cannot easily be computed in terms of simple CPU operations and need a lot of cycles to complete. Compute this using numpy first. Then use numexpr with several threads.

   - How the efficency scales?
   - Why it scales differently that the previous polynomial expression?

- Modify the poly.c so that you can evaluate this transcendental expression. How it performs compared with numpy/numexpr?

Out-of-memory computations: `numpy.memmap` versus `tables.Expr`

Now, we are going to make use of the script `poly2.py` to compute the same problem than above, but using an out-of-memory paradigm.

Comparing `numpy.memmap` and `tables.Expr` approaches

6. Use script `poly2.py` to study the *compute_numpy* and *compute_tables* functions and try to understand how the different `numpy.memmap` and `tables.Expr` paradigms work.
   - Compare the times for computing the polynomial via both `numpy.memmap` and `tables.Expr` (set the *what* variable properly). Do you notice some difference? Why?
   - Compare the latter times with the times for the in-memory approach. Why do you think the out-of-memory paradigm is slower?
   - With the out-of-memory approach, try putting the result in-memory. Is the improvement noticeable?

Playing with compression

7. With the `tables.Expr` module, play with different compression levels (including 0, i.e. no compression) for the Blosc compressor.
   - Which one compresses better?
   - Which one achieves the best compression/time ratio?
   - Is this competitive in terms of speed with the non-compressed mode?
8. Compare 'blosc' with other compressors in PyTables, like 'zlib' or 'lzo'.
   - Which one compresses better?
   - Which one achieves the best compression/time ratio?

Making real "out-of-memory" computations

Of course, the advantage of the out-of-memory approach is that you can still perform your computations even if they exceed your available memory.

Warning: In order to not overload the server, please do the next exercises on your laptops only.

9. Set the number of elements (N) in vector x to some value that slightly exceeds the amount of the *physical* memory in your laptop, but still, less than the *virtual* memory.

   Hint: the working set for this problem is 2*N*size(datatype). As the datatype is a double precision one, size(datatype)=8. So, for a laptop with 1 GB of main memory, setting N=80 millions is fine.

   Warning: For this part, you should make sure that you have some swap space available (check with *free* command). If you don't, please create one.
   - Which approach (`numpy.memmap` or `tables.Expr`) is faster?
10. You will have surely noticed some important jitter while doing measurements in this section. Uncomment the:

```
os.system("sync")
```

line in *print_filesize()* function and see if measurements are a bit more reproducible.

- Why do you think it is so?

11. With this setup, try with `tables.Expr` together with Blosc and different compression levels.
    - Which compression level gives best speed? Could you explain why?

Beyond virtual memory limits

12. Finally, use a working set slightly larger than your *virtual* memory. First try `tables.Expr` and then `numpy.memmap`. Spy the memory consumption in another terminal with the "top" utility.

    Hint: In this test `numpy.memmap` will ask for more virtual memory than your system can possibly deliver, so be ready for seeing your process to be killed by the OS, or even worse, you may end with your kernel frozen for several minutes. If your are a bit faint of heart, you are not forced to check this experimentally ;-)

    - Why do you think `tables.Expr` consumes so little memory?