Exercises for Memory-Efficient Computing

Finding optimal block

1. Use the script *cpu_vs_memory.py* in order to find the best block size for computing the evaluation of a certain expression (polynomial).
   - Which is the best block size? How does that compare to the size of L1 and L2 cache?
   - How is the speed-up compared with a *raw* evaluation?
   - Why do you think the blocked calculation is faster than the raw one (for optimal blocksize)?

Optimizing arithmetic expressions

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 doing 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?

Parallelism with threads

4. Be sure that you are on a multi-processor machine and repeat the last computation in poly1.py but increasing the number of threads one by one (change the number in the `for nt in range(1):` loop).
   - How the efficiency scales?
   - Why do you think it scales that way?
   - How performance compares with the pure C computation?
5. With the same multi-processor, recompile the above poly.c, but with OpenMP support:

```
gcc -O3 -o poly poly.c -lm -fopenmp    # notice the new -fopenmp flag!
```

   and execute it for several numbers of threads:

```
OMP_NUM_THREADS=desired_number_of_threads ./poly
```

Compare its performance with the parallel numexpr.

- How the efficiency scales?
- Which is the asymptotic limit?

6. Modify poly.c so that it just copies vector *x* in *y*:

```
y[i] = x[i]
```

and run it for a different number of threads.

- Compare the performance of this with polynomial evaluation.
- Why it scales very similarly than the polynomial evaluation?
- Could you have a guess at the memory bandwidth of this machine?