HPC Tools 2024, Exercise 6 solutions

Bruno Oliveira Cattelan, Laurent Chôné

Problem 1

In this first problem, we investigate the scaling of the Scipy eigensolver as a function of the matrix size.

For context, an eigensolver finds the eigenvalues λ and eigenvectors $\mathbf{v}$ solutions of:

$$A\mathbf{v} = \lambda\mathbf{v}$$

Many problems can be put in such a form, such as eg finding the roots of a polynomial, analysing the stability of a linear PDE, or solving the time-independent Schrödinger equation.

Here the precise form of A is not important, so it can be initialised with some freedom. The solver is simply called as:

```
    eigenValues, eigenVectors = scipy.linalg.eig(matrixA)
```

We then seek to find the $\mathcal{O}(N^a)$ scaling of CPU time as a function of matrix size N. Power laws are determined easiest when working in logarithmic scales:

$$\begin{aligned} t_{\text{CPU}} &= bN^a \\ \Leftrightarrow \ln t_{\text{CPU}} &= a \ln N + \ln b \end{aligned}$$

We therefore measure the CPU time for various matrix sizes, and find the scaling power with linear regression. Since the solver relies on matrix multiplications, we expect the scaling to be between $\mathcal{O}(N^3)$ (naïve algorithm) and $\mathcal{O}(N^{2.37})$ (current fastest matrix multiplication algorithm). BLAS is quoted as using Strassen's algorithm which scales as $\mathcal{O}(N^{2.807})$, but this information seems to be out of date. Note that for small values of N, the results departs from the scaling as the execution time is no longer dominated by the matrix operations.

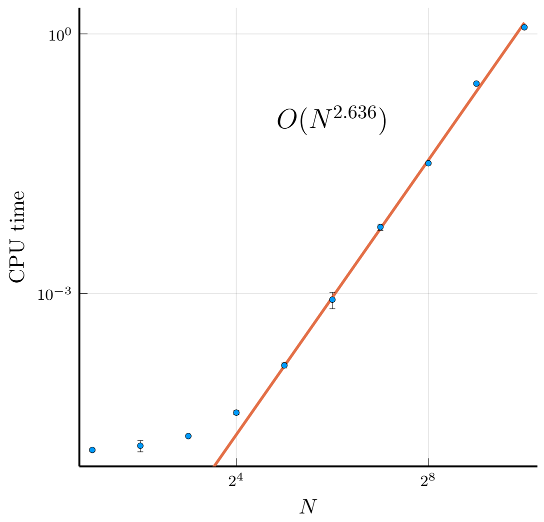


Figure 1: Measured scaling of `scipy.linalg.eig`. The best fit, restricted to $N \geq 32$ is $a = 2.63628$ and 95% interval: $[2.50888, 2.76368]$

Problem 2

For this problem we were asked to implement a parallel program with a communication flow as in Figure 2.

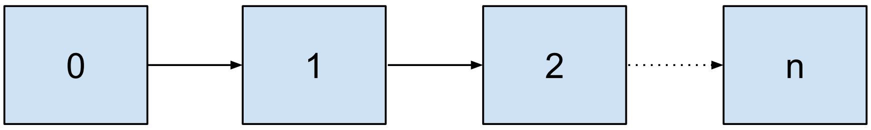
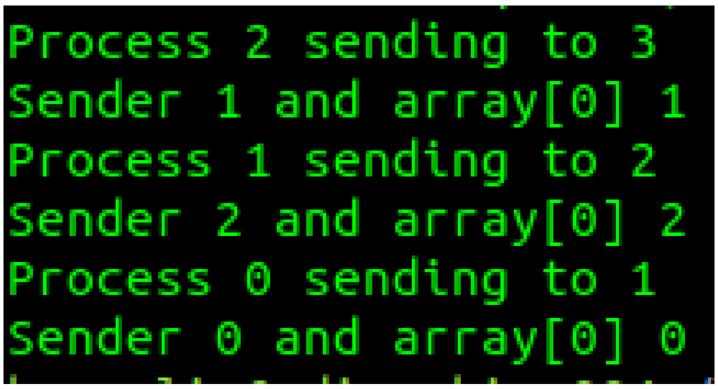


Figure 2: Communication flow - linear

The output is shown in Figure 3. Messages starting with "process" are printed when sending, and the ones starting with "sender" are printed when receiving. We see that the order in which the prints happened are somewhat random, as no synchronization was enforced.



```
Process 2 sending to 3
Sender 1 and array[0] 1
Process 1 sending to 2
Sender 2 and array[0] 2
Process 0 sending to 1
Sender 0 and array[0] 0
```

Figure 3: Output

Problem 3

For this problem we were asked to implement a parallel program with a communication flow as in Figure 4. The difference when compared to the previous one is that this program has a circular communication flow.

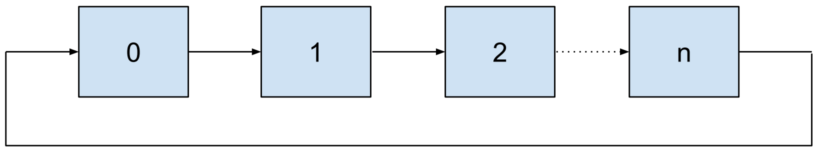
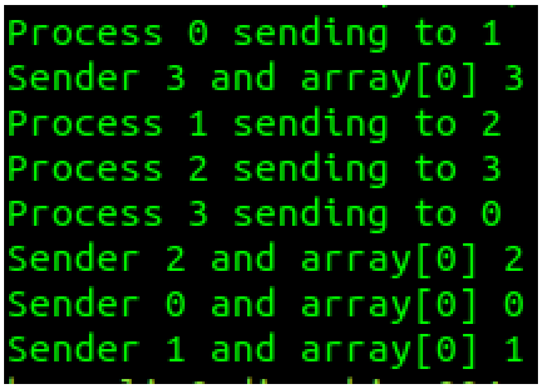


Figure 4: Communication flow - circular

The output is shown in Figure 5. Messages starting with "process" are printed when sending, and the ones starting with "sender" are printed when receiving. We see that the order in which the prints happened are somewhat random, as no synchronization was enforced. We see new messages from the new communication between the last process and the first one. Since by default the sending is non-synchronous, we can write this program with no issue. However, if for some reason we had enforced the sending to be synchronous (meaning they would block until receiving from another process) we end up with the output shown in Figure 6. Meaning that process 0 can only receive once its message is received by process 1, which can only receive once its message is received by the next process and so on. This is an example of a deadlock.



```
Process 0 sending to 1
Sender 3 and array[0] 3
Process 1 sending to 2
Process 2 sending to 3
Process 3 sending to 0
Sender 2 and array[0] 2
Sender 0 and array[0] 0
Sender 1 and array[0] 1
```

Figure 5: Output - working

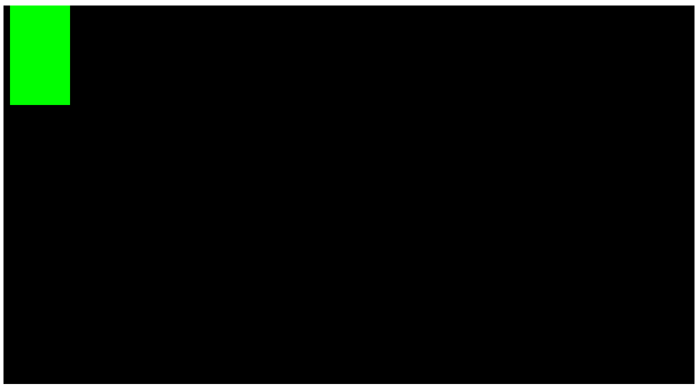


Figure 6: Output - deadlock

Problem 4

In this problem, we attempt to empirically measure the bandwidth and latency of MPI communications, using a program performing ping-pong communications (one can adapt the program from Exercise 5 Problem 4, to send an array instead of a single value).

As mentioned in the exercise statement, the CPU time should be linearly proportional to the message size N, with the parameters b and l the bandwidth and latency, respectively.

$$t_{\text{CPU}} = \frac{N}{b} + l$$

We therefore measure the time fore several executions of the program with increasing memory size, and perform a linear regression on the result to find the parameters.

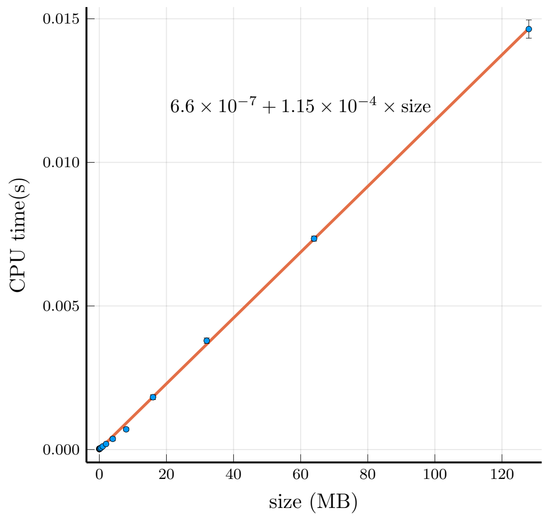


Figure 7: Measured scaling of MPI message exchange as a function of message size.

From the regression, we find:

$$l = 6.60307 \times 10^{-7}\ (\text{s}), \quad 95\%\ \text{int.}: \left[-3.23207 \times 10^{-5}, 3.36413 \times 10^{-5}\right]$$
$$b = 8.53\ (\text{GB/s}), \quad 95\%\ \text{int.}: [8.45, 8.6]$$