

# Scalability Report

In the following an assessment of the scalability of the `respy` function `_full_solution` is provided. Input data for the function was generated by two simulated models within `respy`: `kw_94_one` and `kw_97_basic`. Data on wages, nonpecuniaries, continuation values, optimization parameters (time preference parameter  $\delta$  and ambiguity parameter  $\eta$ ) is available for periods 1, 8, 18, 28, 38 (and 48 for the `kw_97_basic` model).

Listing 1: function `_full_solution`

---

```
@parallelize_across_dense_dimensions
def _full_solution(
    wages, nonpecs, continuation_values, period_draws_emax_risk, optim_paras
):
    period_expected_value_functions = calculate_expected_value_functions(
        wages,
        nonpecs,
        continuation_values,
        period_draws_emax_risk,
        optim_paras["delta"],
        optim_paras["eta"],
    )

return period_expected_value_functions
```

---

The function `_full_solution` is fully guvectorized through `calculate_expected_value_functions`. Hence for any scalability analysis it is necessary to set the following environment variables (Listing 2) **before** importing `_full_solution`, and in particular before importing any `numba` and `numpy`-related libraries.

This holds true for any intended change of the environment variables. To test the computation time with respect to different number of maximal threads it is necessary to re-load the module. The environment variables will be stores and applied whenever `numba` and `numpy`-related libraries are imported. Otherwise changes will not come into place. This can be implemented with a caller module.

Listing 2: Environment settings - number of used threads

---

```
update_ = dict.fromkeys(
    [
        "NUMBA_NUM_THREADS",
        "MKL_NUM_THREADS",
        "OMP_NUM_THREADS",
        "OPENBLAS_NUM_THREADS",
        "NUMEXPR_NUM_THREADS",
    ],
    num_threads,
)
```

---

# Results

Results were produced on a 2.6 GHz Quad-Core Intel Core i7 (with virtually 8 cores). Noteworthy, each physical core of the i7 performs hyper-threading. Hence, four i7 physical cores translate to eight virtual cores. With the command `$ sysctl hw.physicalcpu hw.logicalcpu` it is possible to check the number of physical vs. virtual cores available to the local machine. For purpose of comparability, the timing analysis was conducted with 100 iterations of the function call.

Results for `kw_94_one` are illustrated in Figure 1 and indicate that the computing time decreases with the number of maximum involved threads. After the last maximum number of cores are utilized (eight) there is no possible gain in computational time. If more cores would be available the computational time could be reduced.

Comparison between Subfigure 1(a) and Subfigure 1(b) uncovers another instance: Depending on the structure of the input data the physical cores reach their limit at some point. Setting up new virtual memory leads to overhead costs and consequently to an increase in computational time.

For the larger model `kw_97_basic` two data sets were generated, labeled one and two. Results are illustrated in Figure 2. Computation time decreases until the fourth core (last physical core) is reached. For more than four core virtual memory has to be created, which causes additional overhead costs and so increases computation time. Once the costs incurred computation time again reduces and stays constants after the eighth virtual core was exploited. There is no discernible difference between the two datasets.

## CPU Monitoring

To gain a deeper understanding of the kernel usage CPU monitoring of the local machine may be beneficial. In the following pages the results are presented. The function `_full_solution` was monitored with sample input data of the `kw_94_one` period 38. The CPU monitoring indicates the core usage under hyper-threading. Any CPU usage in the range  $[x \cdot 100\%, (x \cdot 100 + 99)\%]$  shows that  $x$  cores are utilized. The number of total threads (not only for the computation) is shown in Python3.7. With each page the maximum number of utilizable cores is increased. **Need to redo the graphs with a clearer performance picture. Some flaws because of unnecessary processes in current implementation.**

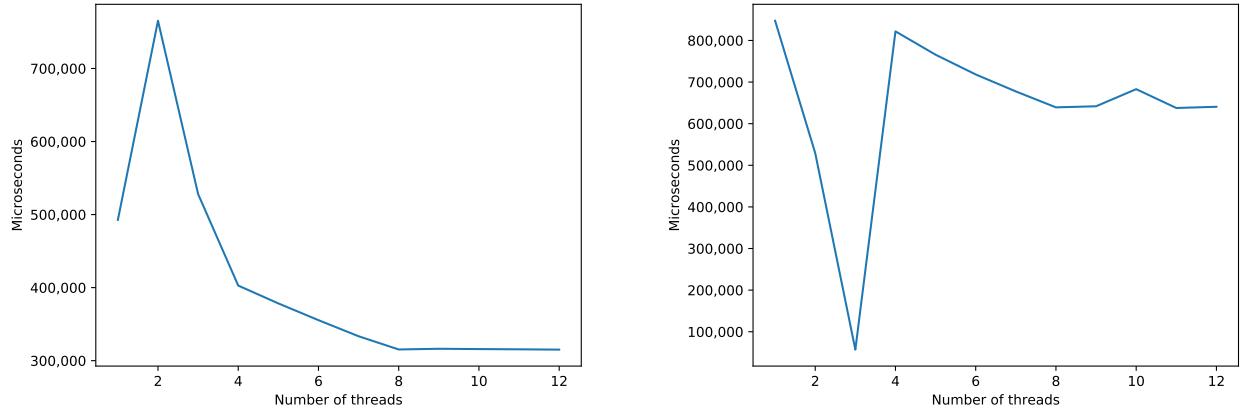


Figure 1: Illustration of computation time vs. maximum available threads for `kw_94_one`.

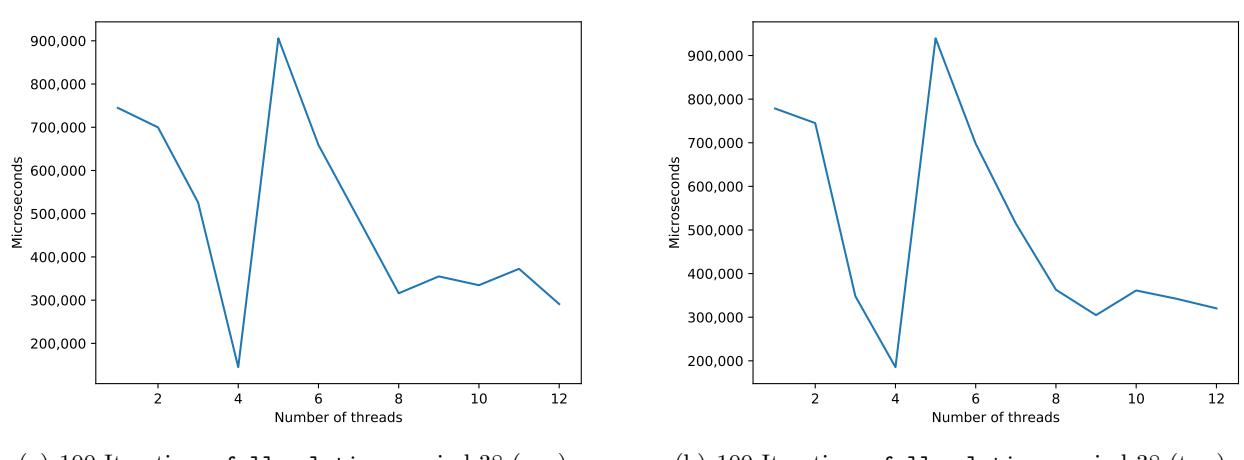


Figure 2: Illustration of computation time vs. maximum available threads for `kw_97_basic`.



