Motivation: Showing a performance tableau

Consider a performance table showing the service quality of 12 commercial cloud providers measured by an external auditor on 14 incommensurable performance criteria.

Legend: 0 = 'very weak', 1 = 'weak', 2 = 'fair', 3 = 'good', 4 = 'very good', 'NA' = missing data; 'green' and 'red' mark the best, respectively the worst, performances on each criterion.

Motivation: showing an ordered heat map

The same performance tableau may be optimistically colored with the highest 7-tiles class of the marginal performances and presented like a heat map, eventually linearly ordered, following for instance the Copeland ranking rule, from the best to the worst performing alternatives (ties are lexicographically resolved).

How to rank big performance tableaux?

- The Copeland ranking rule, for instance, is based on crisp net flows requiring the in- and out-degree of each node in the outranking digraph;
- When the order $n$ of the outranking digraph becomes big (several thousand or millions of alternatives), this requires handling a huge set of $n^2$ pairwise outranking situations;
- We use instead a sparse model of the outranking digraph, where we only keep a linearly ordered list of diagonal multicriteria quantiles equivalence classes with local outranking content.
HPC performance measurements HPC school 2017

digraph order standard model sparse model

<table>
<thead>
<tr>
<th>#c.</th>
<th>tg sec.</th>
<th>τg</th>
<th>#c.</th>
<th>tbg</th>
<th>τbg</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>118 6&quot;</td>
<td>+0.88</td>
<td>8 1.6&quot;</td>
<td>+0.83</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>118 15&quot;</td>
<td>+0.88</td>
<td>8 3.5&quot;</td>
<td>+0.83</td>
<td></td>
</tr>
<tr>
<td>2500</td>
<td>118 27&quot;</td>
<td>+0.88</td>
<td>8 4.4&quot;</td>
<td>+0.83</td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100000</td>
<td>(size = 10^{10})</td>
<td>118 2&quot;</td>
<td>(fill rate = 0.077%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000000</td>
<td>(size = 10^{12})</td>
<td>118 36&quot;</td>
<td>(fill rate = 0.028%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1732051</td>
<td>(size = 3 \times 10^{12})</td>
<td>118 2h17&quot;</td>
<td>(fill rate = 0.010%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2236068</td>
<td>(size = 5 \times 10^{12})</td>
<td>118 3h15&quot;</td>
<td>(fill rate = 0.010%)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Legend:
- #c. = number of cores;
- g: standard outranking digraph, bg: the sparse outranking digraph;
- τg, resp. τbg, are the corresponding constructor run times;
- τg, resp. τbg are the ordinal correlation of the Copeland ordering with the given outranking relation.

Gaia-80 November 2016 ranking record

New performance measurements Spring 2018

<table>
<thead>
<tr>
<th>q outranking relation</th>
<th>order</th>
<th>size</th>
<th>q</th>
<th>fill rate</th>
<th>nbr. cores</th>
<th>run time</th>
</tr>
</thead>
<tbody>
<tr>
<td>≿q</td>
<td>5000</td>
<td>25 \times 10^6</td>
<td>4</td>
<td>0.005%</td>
<td>28</td>
<td>0.5&quot;</td>
</tr>
<tr>
<td></td>
<td>10000</td>
<td>1 \times 10^8</td>
<td>4</td>
<td>0.001%</td>
<td>28</td>
<td>1&quot;</td>
</tr>
<tr>
<td></td>
<td>100000</td>
<td>1 \times 10^{10}</td>
<td>5</td>
<td>0.002%</td>
<td>28</td>
<td>10&quot;</td>
</tr>
<tr>
<td></td>
<td>1000000</td>
<td>1 \times 10^{12}</td>
<td>6</td>
<td>0.001%</td>
<td>64</td>
<td>2'</td>
</tr>
<tr>
<td></td>
<td>3000000</td>
<td>9 \times 10^{12}</td>
<td>15</td>
<td>0.004%</td>
<td>64</td>
<td>13'</td>
</tr>
<tr>
<td></td>
<td>6000000</td>
<td>36 \times 10^{12}</td>
<td>15</td>
<td>0.002%</td>
<td>64</td>
<td>41'</td>
</tr>
</tbody>
</table>

Successful actions for enhancing the performances - 1

- Algorithmic refinements: The pre-ranking quantiles sorting algorithm was further optimized, reducing considerably the fill rate of the sparse outranking digraphs;

These run times are achieved both:
- on the Iris -skylake nodes with 28 cores,
- on the 3TB -bigmem Gaia-183 node with 64 cores, and
- running cythonized python modules in an Intel compiled virtual Python 3.6.5 environment [GCC Intel(R) 17.0.1 –enable-optimizations c++ 6.3 mode] on Debian 8 linux.
Sparse versus standard digraph - Nov 2016

Symbol legend
⊤ outranking for certain
+ more or less outranking
· indeterminate
− more or less outranked
⊥ outranked for certain

Sparse digraph bg:
# Actions : 50
# Criteria : 7
Sorted by : 5-Tiling
Ranking rule : Copeland
# Components : 7
Minimal order : 1
Maximal order : 15
Average order : 7.1
fill rate : 20.980%
correlation : +0.7563

Sparse outranking digraph - Now

Reducing the size of python data objects

tp1 Standard Random 3 Objectives performance tableau instance with 5000 decision actions and 21 performance criteria: size(tp1) = 3 602 132 Bytes.

tp2 Same BigData Random 3 Objectives performance tableau instance: size(tp2) = 1 398 365 Bytes.

Successful actions for enhancing the performances - 2

- **Algorithmic refinements**: The pre-ranking quantiles sorting algorithm was further optimized, reducing considerably the fill rate of the sparse outranking digraphs;

- **Reducing the size of python data objects**: A special bigData performance tableau model with integer dictionary keys and float evaluations is used for optimized Cython and C compiler variable typing;
Reducing the size of python data objects

tp1 Standard Random 3 Objectives performance tableau instance with 5000 decision actions and 21 performance criteria: \( \text{size}(tp1) = 3\,602\,132 \) Bytes.

tp2 Same BigData Random 3 Objectives performance tableau instance: \( \text{size}(tp2) = 1\,398\,365 \) Bytes.

bg1 Standard pre-ranked outranking digraph instance generated from tp1: \( \text{size}(bg1) = 9\,471\,896 \) Bytes.

bg2 BigData pre-ranked outranking digraph instance generated from tp2: \( \text{size}(bg2) = 1\,791\,755 \) Bytes.

Efficient Cython inline function declaration with variable typing

cdef inline int _localConcordance(float d, float ind, float wp, float p):
    """ None = -1.0 ""
    if p > -1.0:
        if d <= -p:
            return -1
        elif ind > -1.0:
            if d >= -ind:
                return 1
            else:
                return 0
        elif wp > -1.0:
            if d > -wp:
                return 1
            else:
                return 0
    else:
        if d < 0.0:
            return -1
        else:
            return 1

Successful actions for enhancing the performances - 3

- **Algorithmic refinements**: The pre-ranking quantiles sorting algorithm was further optimized, reducing considerably the fill rate of the sparse outranking digraphs;

- **Reducing the size of python data objects**: A special bigData performance tableau model with integer dictionary keys and float evaluations is used for optimized Cython and C compiler variable typing;

- **Efficient sharing of static data**: Global python variables allow to efficiently communicate static data objects to parallel threads when using -bigmem nodes;

Successful actions for enhancing the performances - 4

- **Algorithmic refinements**: The pre-ranking quantiles sorting algorithm was further optimized, reducing considerably the fill rate of the sparse outranking digraphs;

- **Reducing the size of python data objects**: A special bigData performance tableau model with integer dictionary keys and float evaluations is used for optimized Cython and C compiler variable typing;

- **Efficient sharing of static data**: Global python variables allow to efficiently communicate static object data to parallel threads when using -bigmem nodes;

- **Using a multiprocessing tasks queue**: Sorting tasks in decreasing durations and using an automatic multithreading mechanism (see the multiprocessing python3 documentation)
Using a multiprocessing tasks queue

```python
with TemporaryDirectory(dir=tempDir) as tempDirName:
    # tasks queue and workers launching
    NUMBER_OF_WORKERS = nbrOnCPUs
    tasksIndex = [(i, len(decomposition[i][1])) for i in range(nc)]
    tasksIndex.sort(key=lambda pos: pos[1], reverse=True)
    TASKS = [(Comments, (pos[0], nc, tempDirName)) for pos in tasksIndex]
    task_queue = Queue()
    for task in TASKS:
        task_queue.put(task)
    for i in range(NUMBER_OF_WORKERS):
        Process(target=_worker, args=(task_queue,)).start()
    if Comments:
        print('started')
    for i in range(NUMBER_OF_WORKERS):
        task_queue.put('STOP')
    while active_children() != []:
        pass
    if Comments:
        print('Exit %d threads' % NUMBER_OF_WORKERS)
```

Successful actions for enhancing the performances - 5

- **Algorithmic refinements**: The pre-ranking quantiles sorting algorithm was further optimized, reducing considerably the fill rate of the sparse outranking digraphs;

- **Reducing the size of python data objects**: A special bigData performance tableau model with integer dictionary keys and float evaluations is used for optimized Cython and C compiler variable typing;

- **Efficient sharing of static data**: Global python variables allow to efficiently communicate static object data to parallel threads when using -bigmem nodes;

- **Using a multiprocessing task queue**: Sorting tasks in decreasing durations and using an automatic multithreading mechanism.

- **Efficient UL HPC cluster equipments and staff**: Thank you for your support :)

Further documentation resources

Our cythonized Python HPC modules are freely available under the cython directory on:

- [https://github.com/rbis dorff/Digraph3](https://github.com/rbis dorff/Digraph3) and on
- [https://sourceforge.net/projects/digraph3/](https://sourceforge.net/projects/digraph3/)

Tutorials and technical documentation + source code listings may be consulted on: