Performance Measurement of Test & Poisson of Flecsi (task_local branch)

Outlines:

I. Test performance (hpx@master backend)

1. Parameters:

- Command:

```
mpirun -n 4 $executable --backend-args="--hpx:ini=hpx.max_background_threads!=$i" --backend-args="--hpx:print-bind"
```

- Build with Release type on medusa node. (mpirun -n 4 wont hang.).
- Build with Debug type on medusa node. (mpirun -n 4, mpirun -n 2 wont hang. -n 8 and more will hang though remove –hpx:threads=N).
- Background threads =[1 2 4 8 16 24 32 40] .
- Repeat **5** times.
- Executable order follows ctest order.

```
EXECUTABLES=(
    "$exe_dir/exec/kernel"
    "$exe_dir/exec/task"
    "$exe_dir/exec/future"
    "$exe_dir/run/flog"
    "$exe_dir/run/program-options"
    "$exe_dir/topo/index"
    "$exe_dir/topo/coloring"
    "$exe_dir/topo/unstructured"
    "$exe_dir/topo/fixed"
    "$exe_dir/topo/set"
    "$exe_dir/topo/narray"
    "$exe_dir/topo/ntree_geometry"
    "$exe_dir/util/array_ref"
    "$exe_dir/util/common"
    "$exe_dir/util/color_map"
    "$exe_dir/util/unit"
    "$exe_dir/util/serialize"
    "$exe_dir/util/set_utils"
    "$exe_dir/util/point"
    "$exe_dir/util/filling_curve"
```

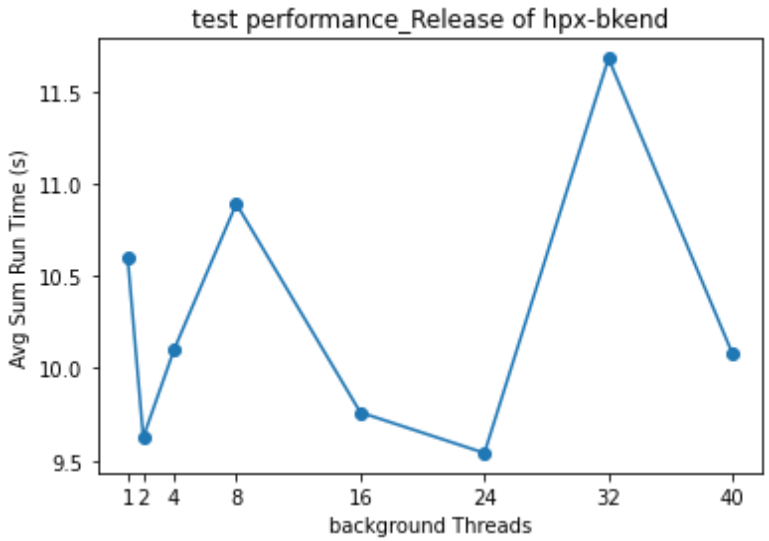
```
    "$exe_dir/util/hashtable"
    )
```

2. Results:

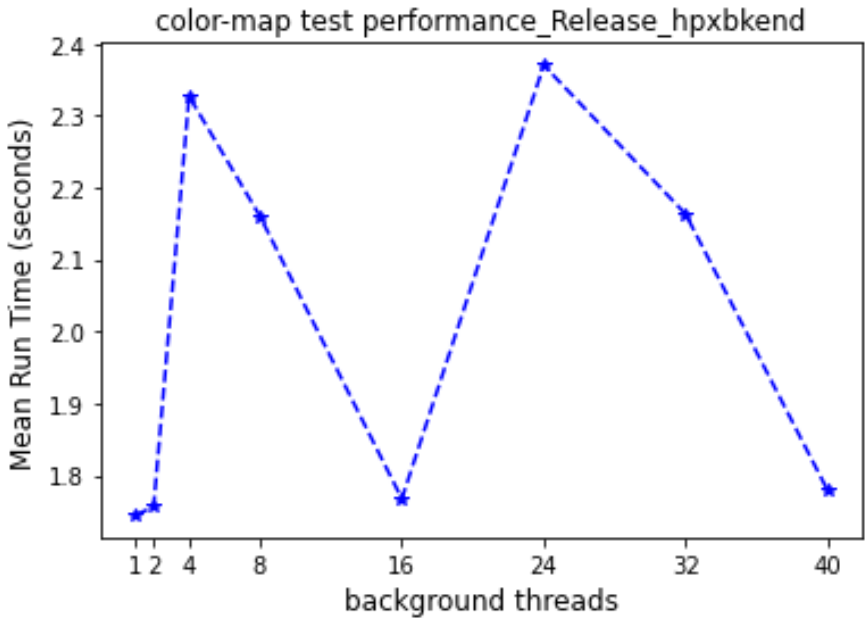
First, sum time for all tests under different threads. And the '**color_map**' test takes the longest average run time under different threads numbers under Release & Debug type.

----Release type:

All tests

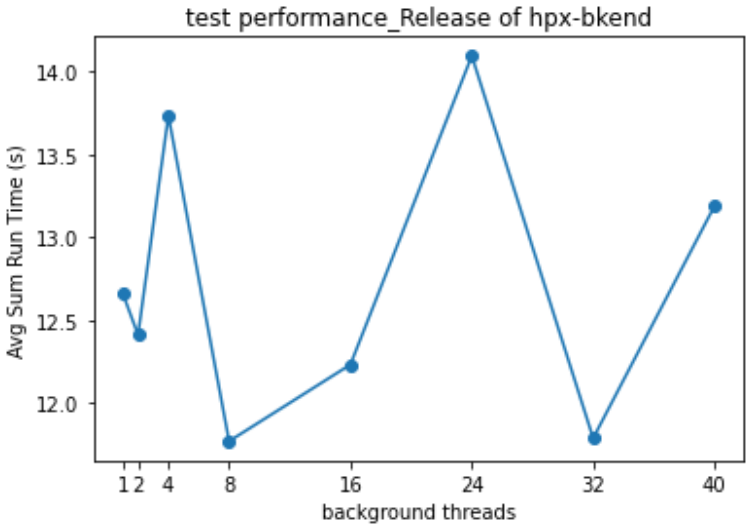


color-map test

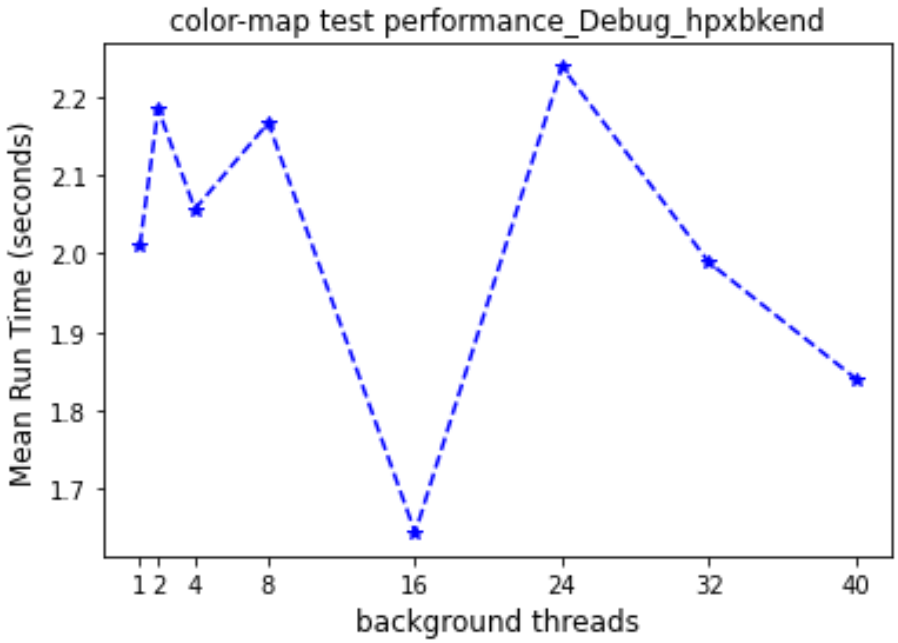


----Debug type:

All tests



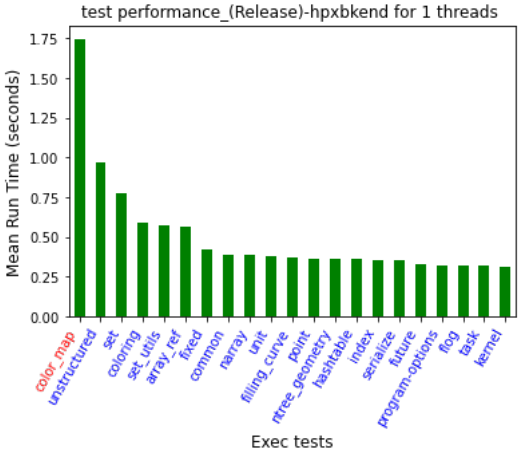
color-map test

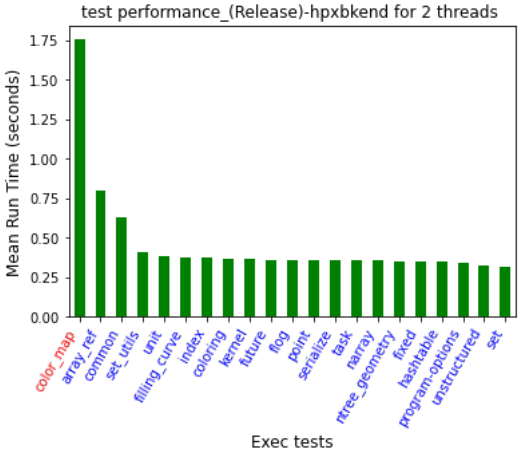


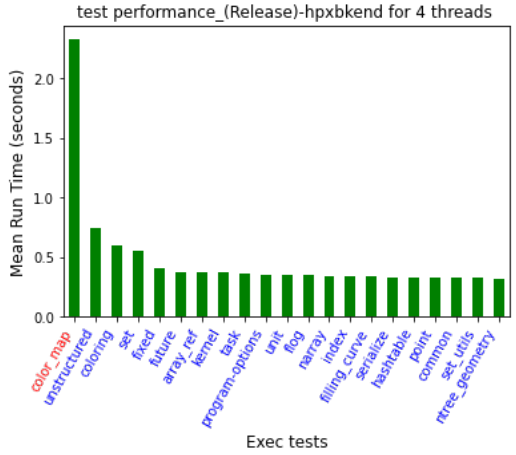
1.3 Appendix:

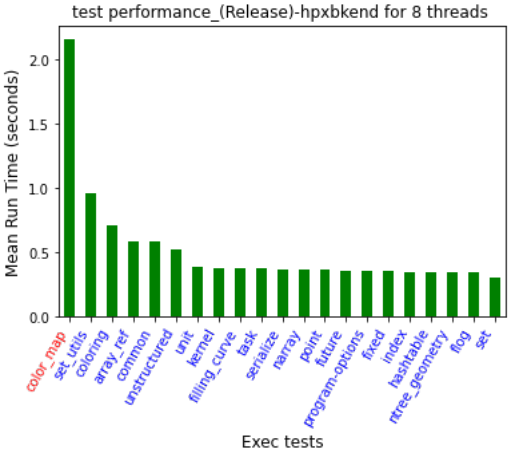
Following plots show that the average run time for various tests under each thread under Release and Debug type. '**color_map**' test among the various tests takes the longest average run time.

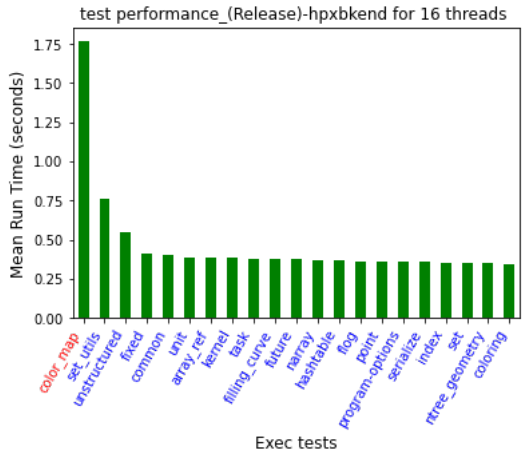
Release type:

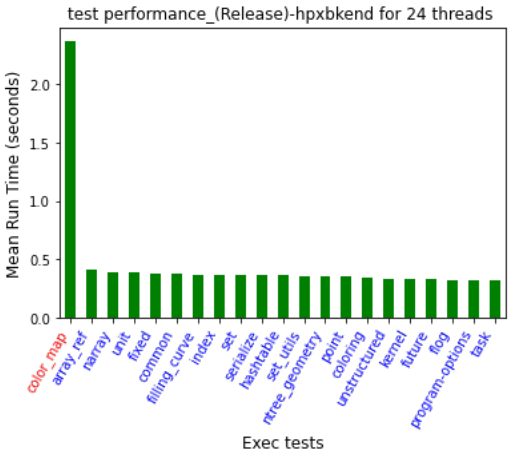


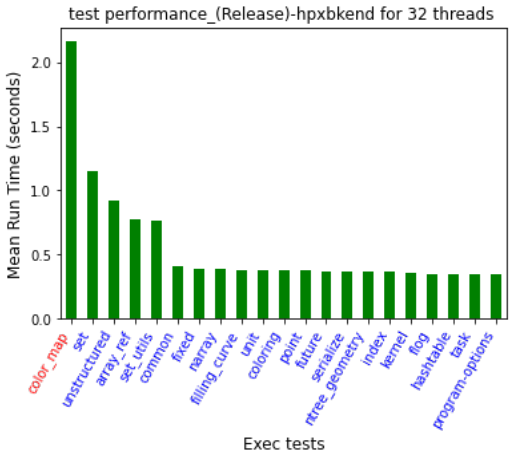


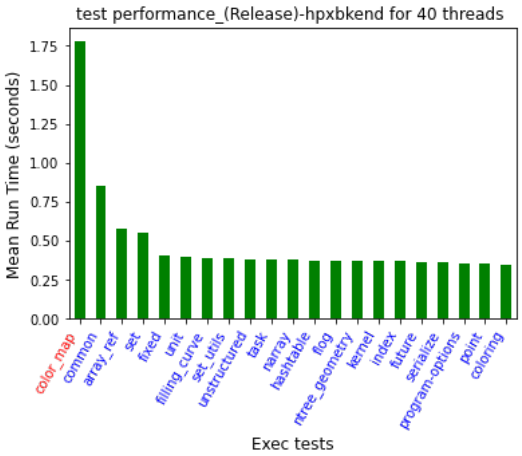




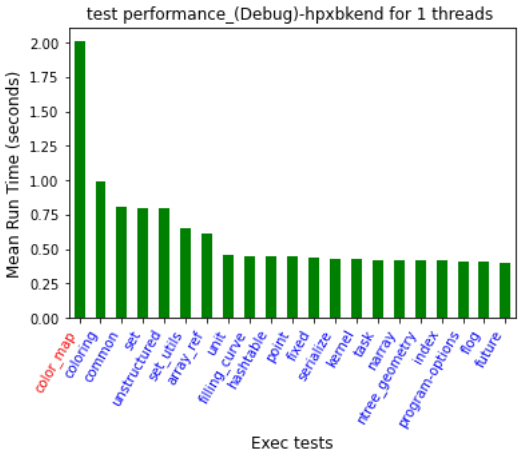


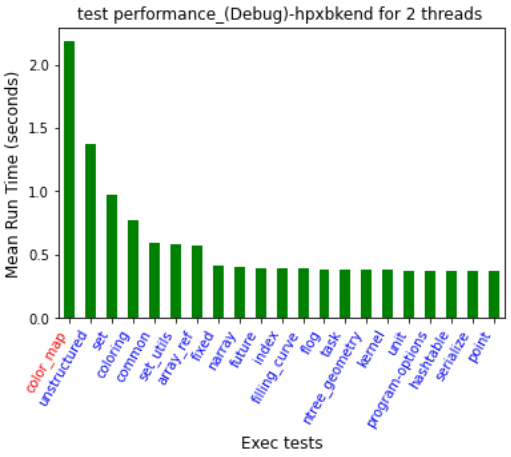


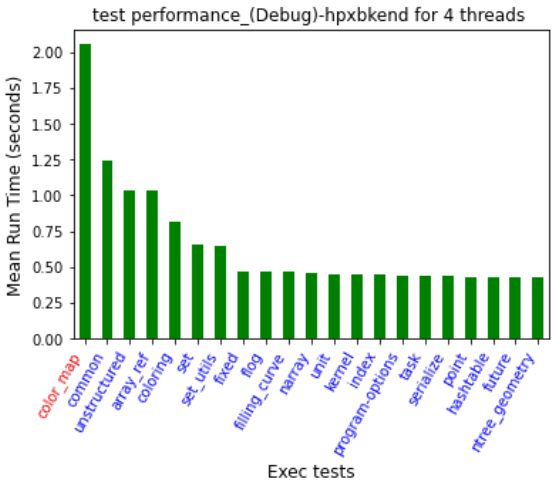

test performance_(Release)-hpxbkend for 32 threads
Mean Run Time (seconds)
Exec tests
color_map
set
unstructured
array_ref
set_utils
common
fixed
narray
filling_curve
unit
coloring
point
future
serialize
rtree_geometry
index
kernel
flog
hashtable
task
program-options

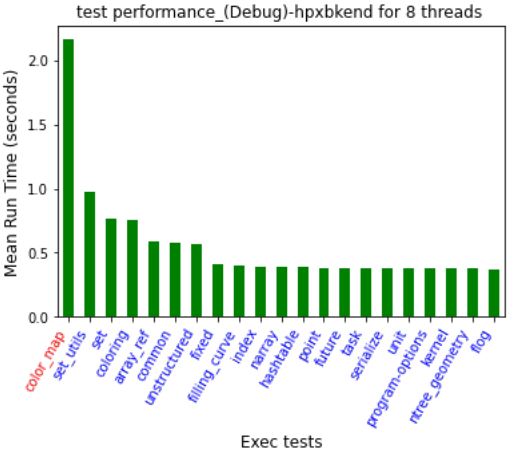

test performance_(Release)-hpxbkend for 40 threads
Mean Run Time (seconds)
Exec tests
color_map
common
array_ref
set
fixed
unit
filling_curve
set_utils
unstructured
task
narray
hashtable
flog
rtree_geometry
kernel
index
future
serialize
program-options
point
coloring

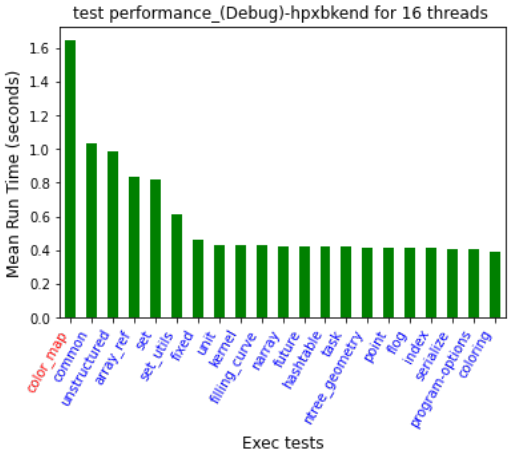
Debug type:

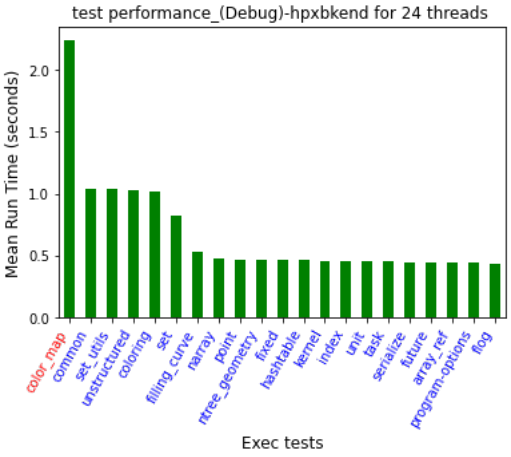

test performance_(Debug)-hpxbkend for 1 threads
Mean Run Time (seconds)
Exec tests
color_map
coloring
common
set
unstructured
set_utils
array_ref
unit
filling_curve
hashtable
point
fixed
serialize
kernel
task
narray
rtree_geometry
index
program-options
flog
future

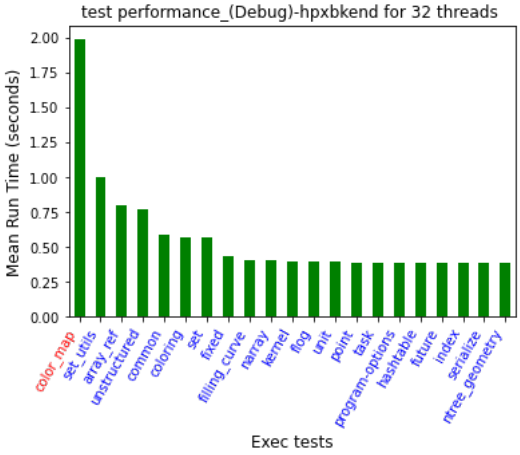

test performance_(Debug)-hpxbkend for 2 threads
Mean Run Time (seconds)
Exec tests
color_map
unstructured
set
coloring
common
set_utils
array_ref
fixed
narray
future
index
filling_curve
flog
task
rtree_geometry
kernel
unit
program-options
hashtable
serialize
point

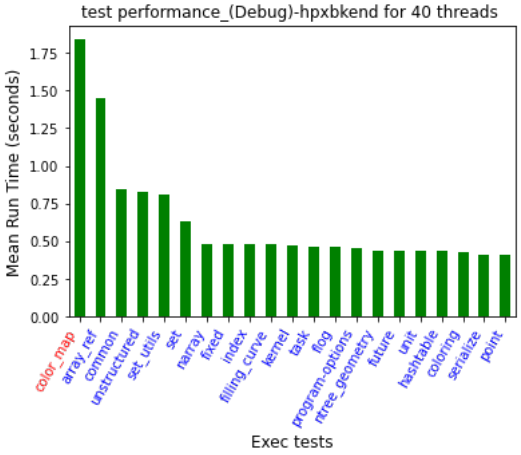

test performance_(Debug)-hpxbkend for 4 threads
Mean Run Time (seconds)
Exec tests
color_map
common
unstructured
array_ref
coloring
set
set_utils
fixed
flog
filling_curve
narray
unit
kernel
index
program-options
task
serialize
point
hashtable
future
rtree_geometry


test performance_(Debug)-hpxbkend for 8 threads
Mean Run Time (seconds)
Exec tests
color_map
set_utils
set
coloring
array_ref
common
unstructured
fixed
filling_curve
index
narray
hashtable
point
future
task
serialize
unit
program-options
kernel
rtree_geometry
flog


test performance_(Debug)-hpxbkend for 16 threads
Mean Run Time (seconds)
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
color_map
common
unstructured
array_ref
set
set_utils
fixed
unit
kernel
filling_curve
narray
future
hashtable
task
rtree_geometry
point
flog
index
serialize
program-options
coloring
Exec tests


test performance_(Debug)-hpxbkend for 24 threads
Mean Run Time (seconds)
0.0
0.5
1.0
1.5
2.0
color_map
common
set_utils
unstructured
coloring
set
filling_curve
narray
point
rtree_geometry
fixed
hashtable
kernel
index
unit
task
serialize
future
array_ref
program-options
flog
Exec tests


test performance_(Debug)-hpxbkend for 32 threads
Mean Run Time (seconds)
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
color_map
set_utils
array_ref
unstructured
common
coloring
set
fixed
filling_curve
narray
kernel
flog
unit
point
task
program-options
hashtable
future
index
serialize
rtree_geometry
Exec tests


test performance_(Debug)-hpxbkend for 40 threads
Mean Run Time (seconds)
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
color_map
array_ref
common
unstructured
set_utils
set
narray
fixed
index
filling_curve
kernel
task
flog
program-options
rtree_geometry
future
unit
hashtable
coloring
serialize
point
Exec tests

II. Poisson performance

1.Parameters:

- Build with Release type on medusa node
- Request localities before measurement: srun -p medusa -N 1 -n **localities** --pty /bin/bash -l
- Matrix = [64 64], [128 128], [256 256]
- Background threads =[1 2 4 8 16 24 32 40]
- hpx backend:

```
mpirun -n $localities $executable $size $size --backend-args="--hpx:ini=hpx.max_background_threads!=$i" --backend-args="--hpx:print-bind"
```

- mpi backend :

```
mpirun -n $localities $executable $size $size
```

2. Results:

Goal:

>Now, the next thing would be to compare the performance of the poisson application for different backends (HPX and MPI). This should be done in release mode with the number of background threads set to whatever gives the best perf for the HPX backend.

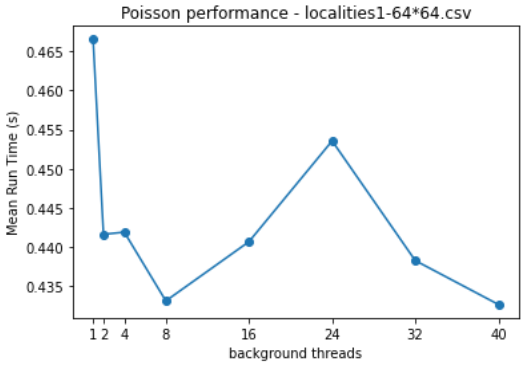
>Could you try three sizes for Poisson? Perhaps 64*64, 128*128, and 256*256? For both, MPI and HPX? And for varying numbers of localities?

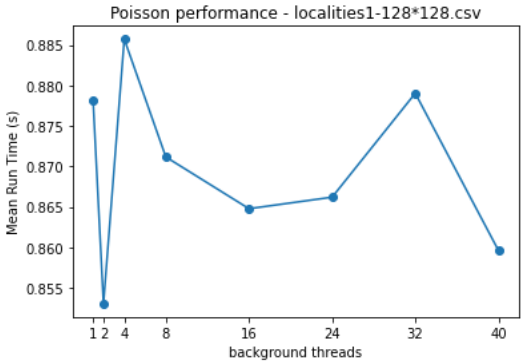
2.1 HPX backend:

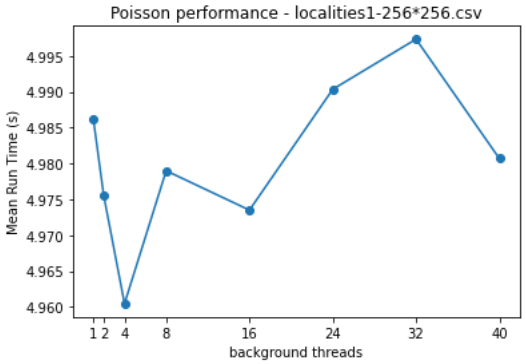
2.1.1 Poisson performance with fixed localities under different threads

Localities = 1, 2, 4, 8, 10, 20

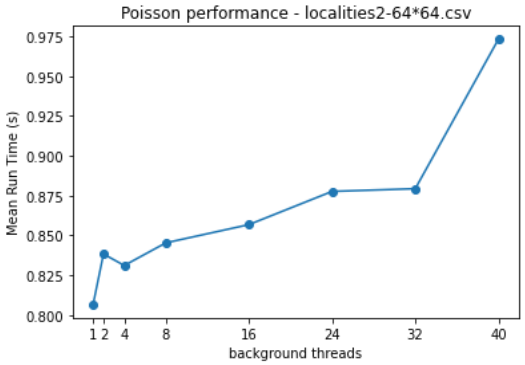
> Localities = 1

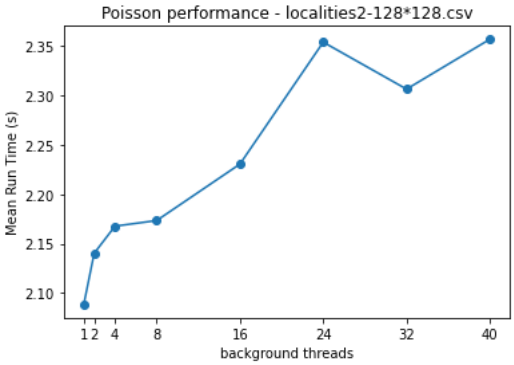


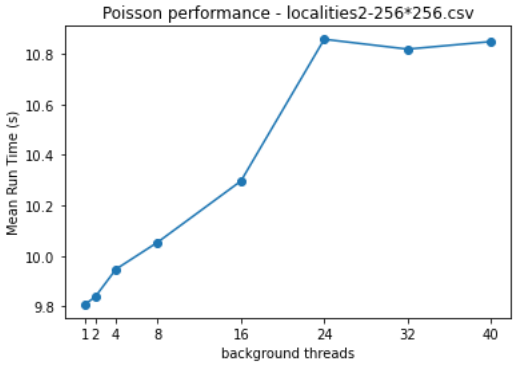




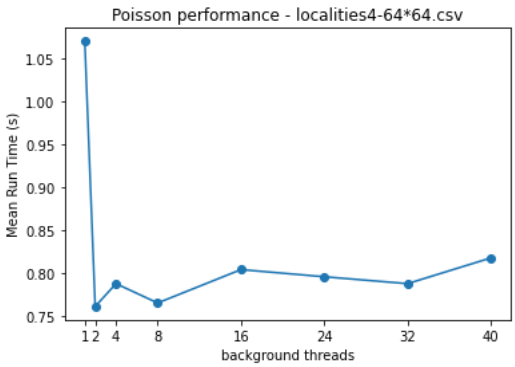
> Localities = 2

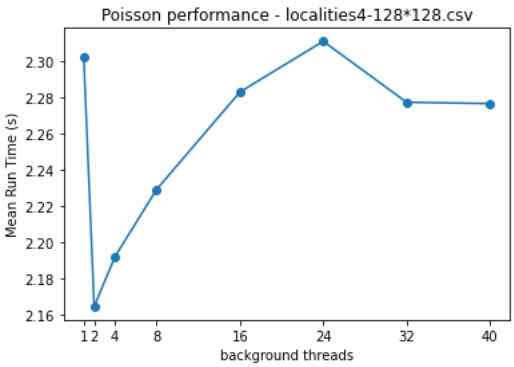


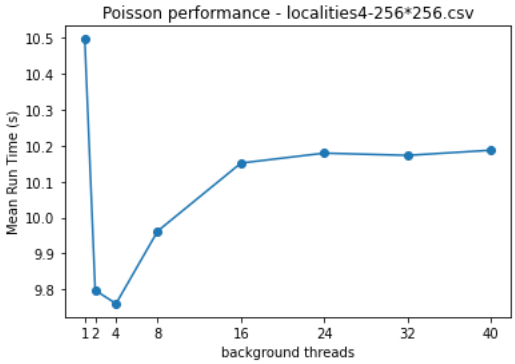




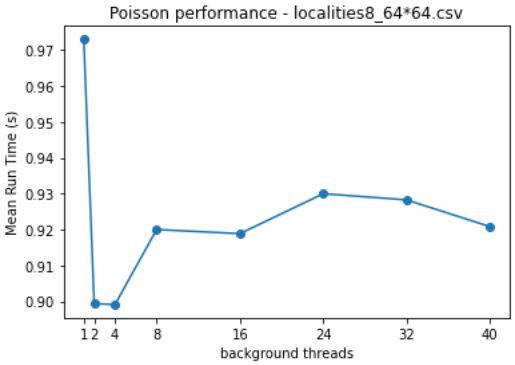
> Localities = 4

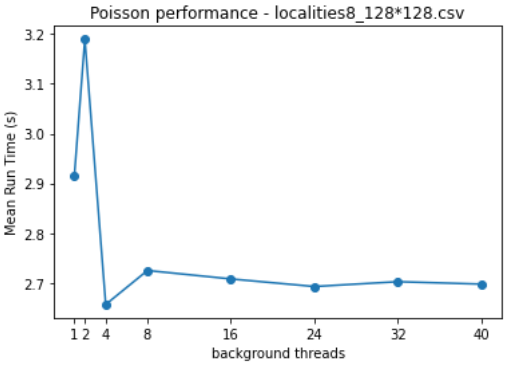


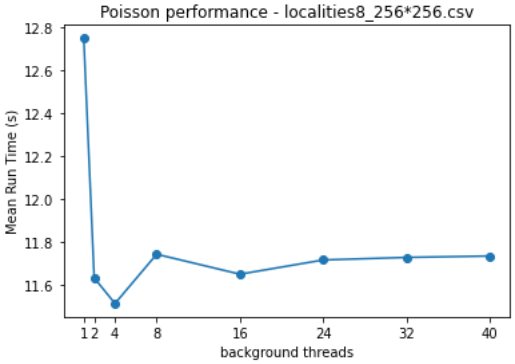




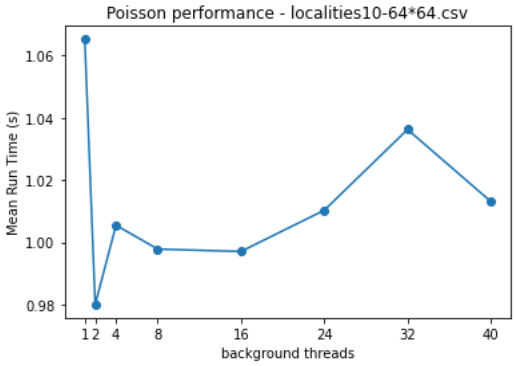
> **Localities = 8**

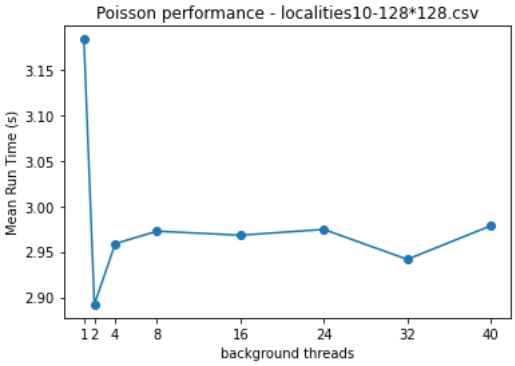


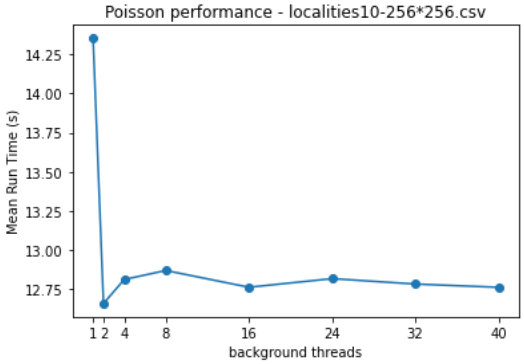




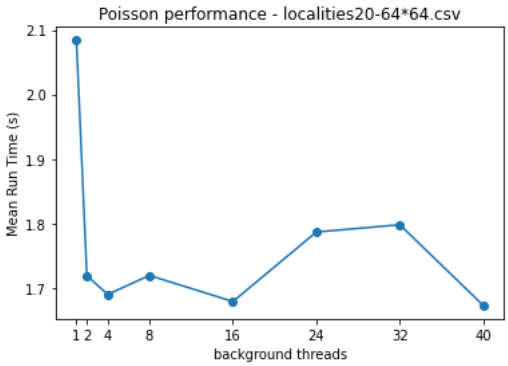
> Localities = 10

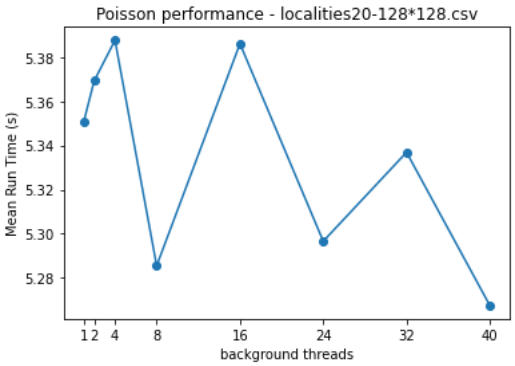


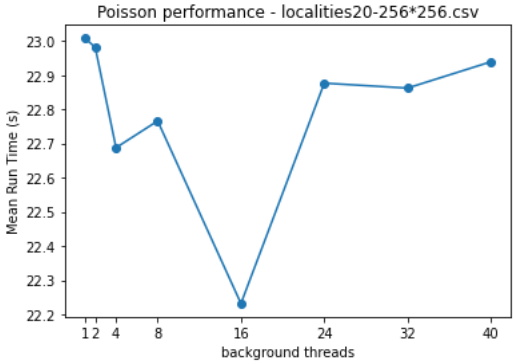




> Localities = 20



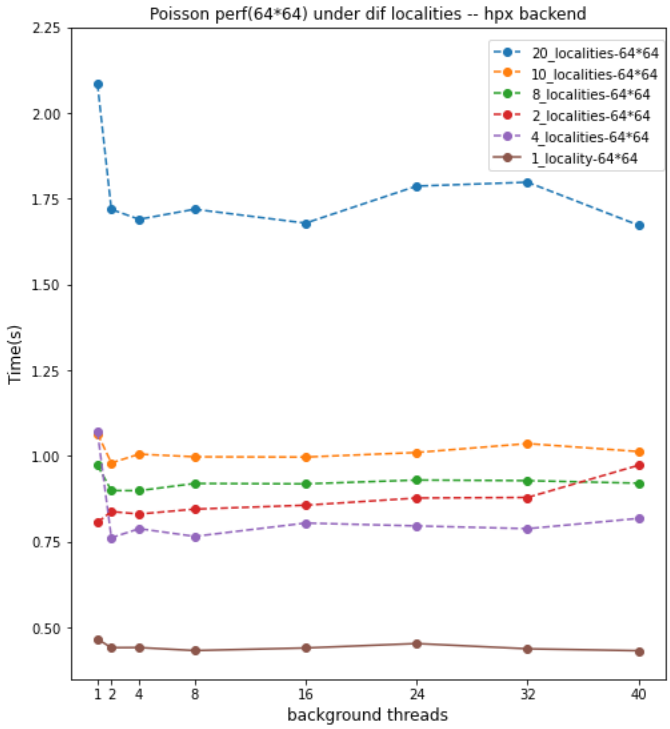




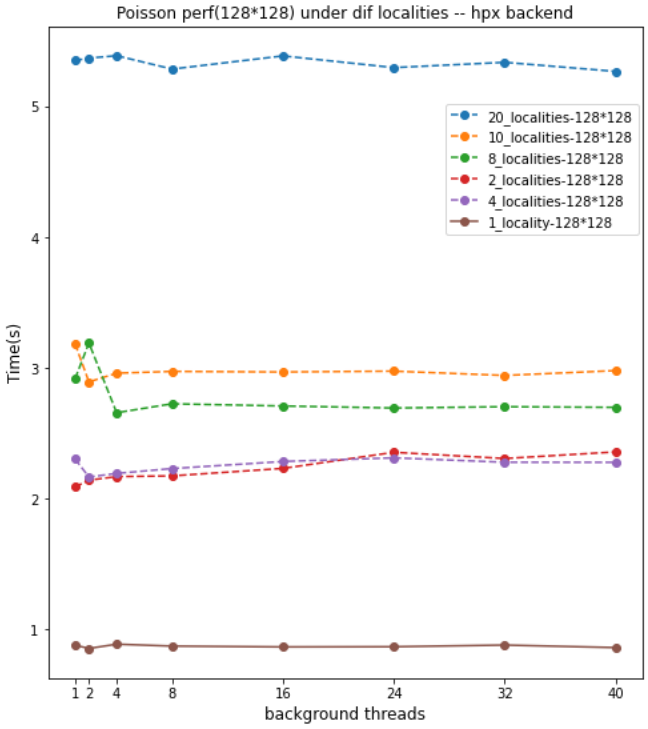
2.1.2 Poisson performance with various localities under different threads

Conclusion: The less localities, the better performance of Poisson.

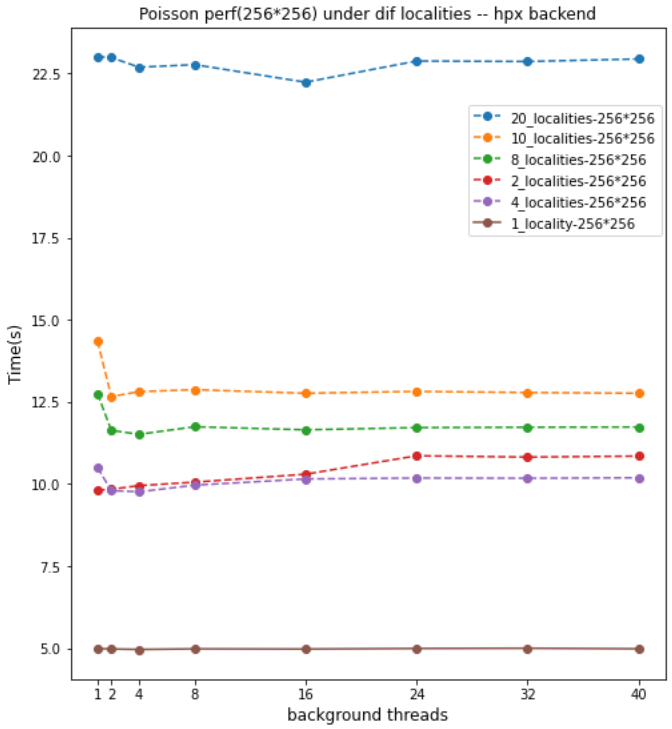
MATRIX = [64 64]



MATRIX = [128 128]

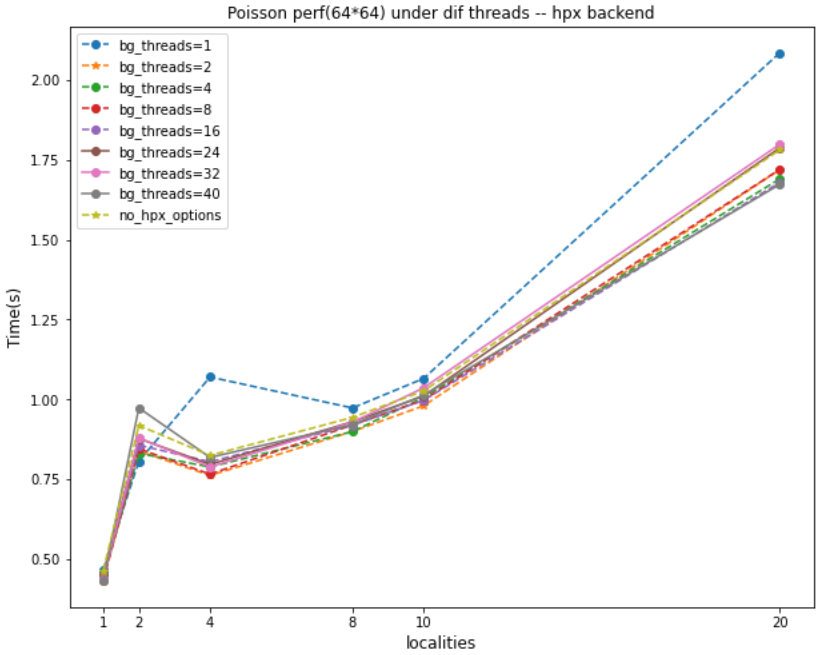


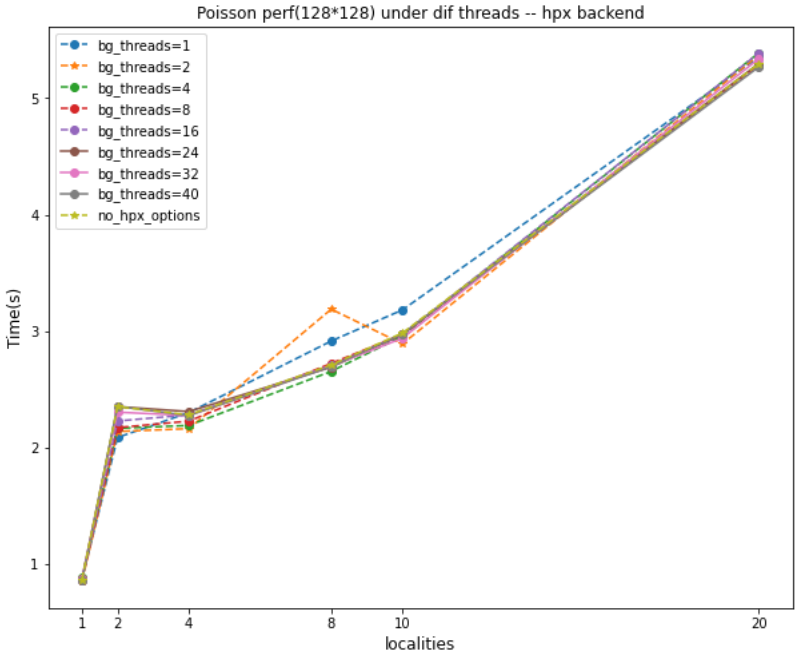
MATRIX = [256 256]

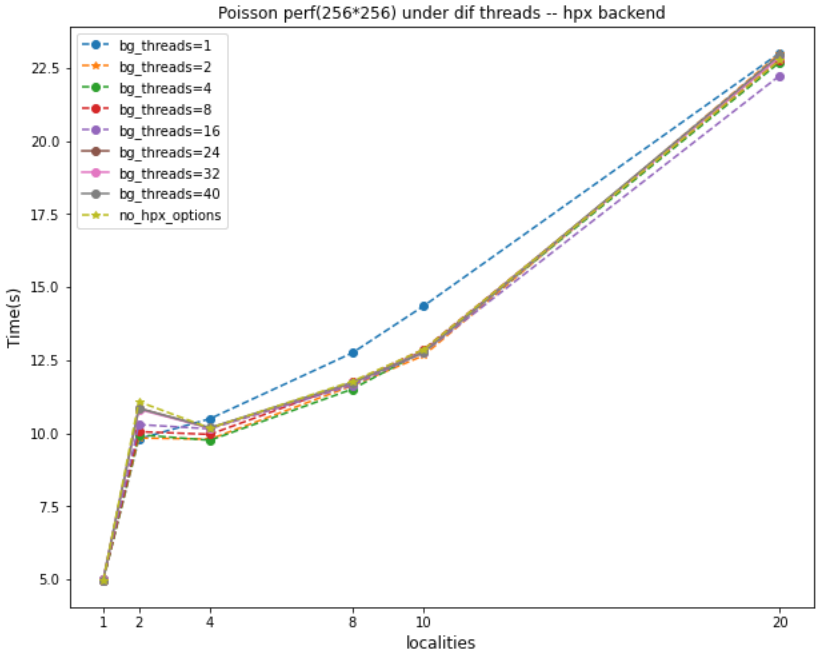


2.1.3 Poisson performance with different threads under various localities

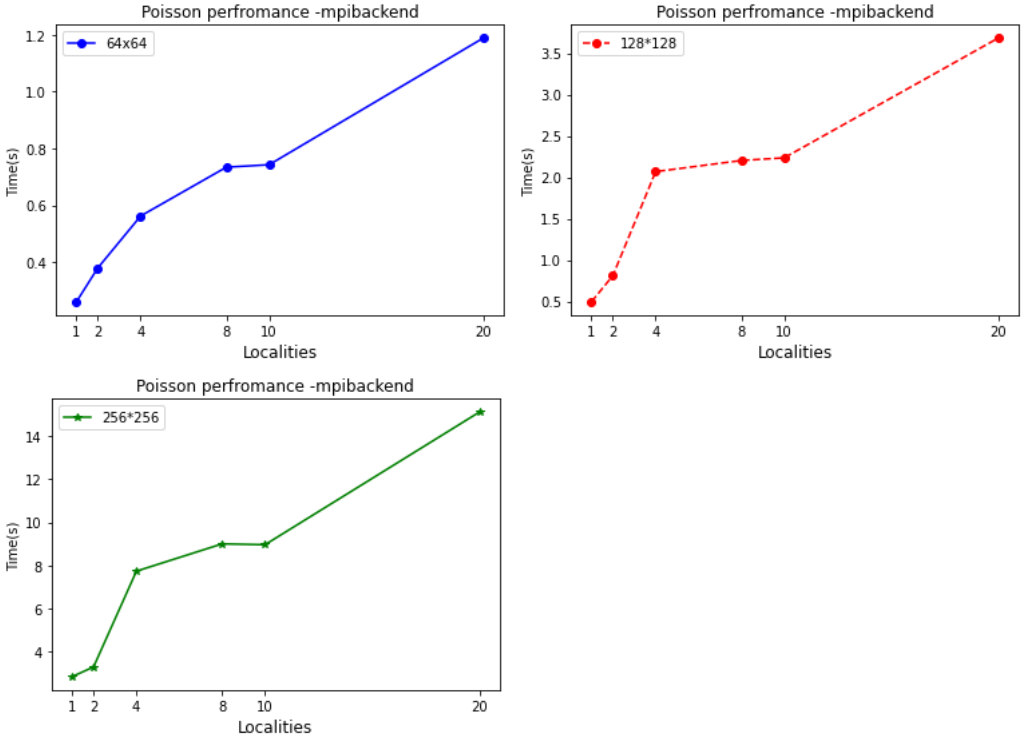
Conclusion: Poisson has better performance when background threads = 4.

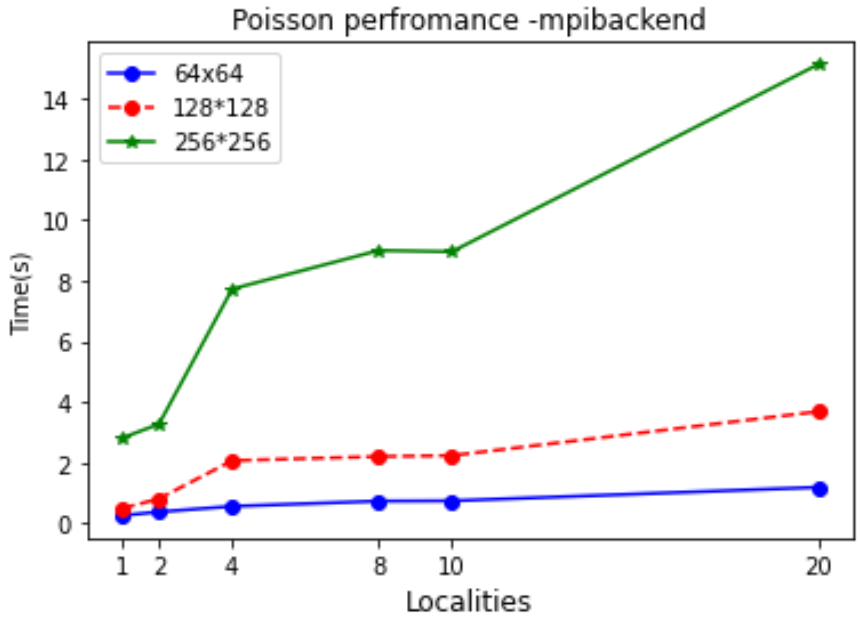

Poisson perf(64*64) under dif threads -- hpx backend
bg_threads=1
bg_threads=2
bg_threads=4
bg_threads=8
bg_threads=16
bg_threads=24
bg_threads=32
bg_threads=40
no_hpx_options
Time(s)
2.00
1.75
1.50
1.25
1.00
0.75
0.50
1
2
4
8
10
20
localities


Poisson perf(128*128) under dif threads -- hpx backend
bg_threads=1
bg_threads=2
bg_threads=4
bg_threads=8
bg_threads=16
bg_threads=24
bg_threads=32
bg_threads=40
no_hpx_options
Time(s)
5
4
3
2
1
1
2
4
8
10
20
localities

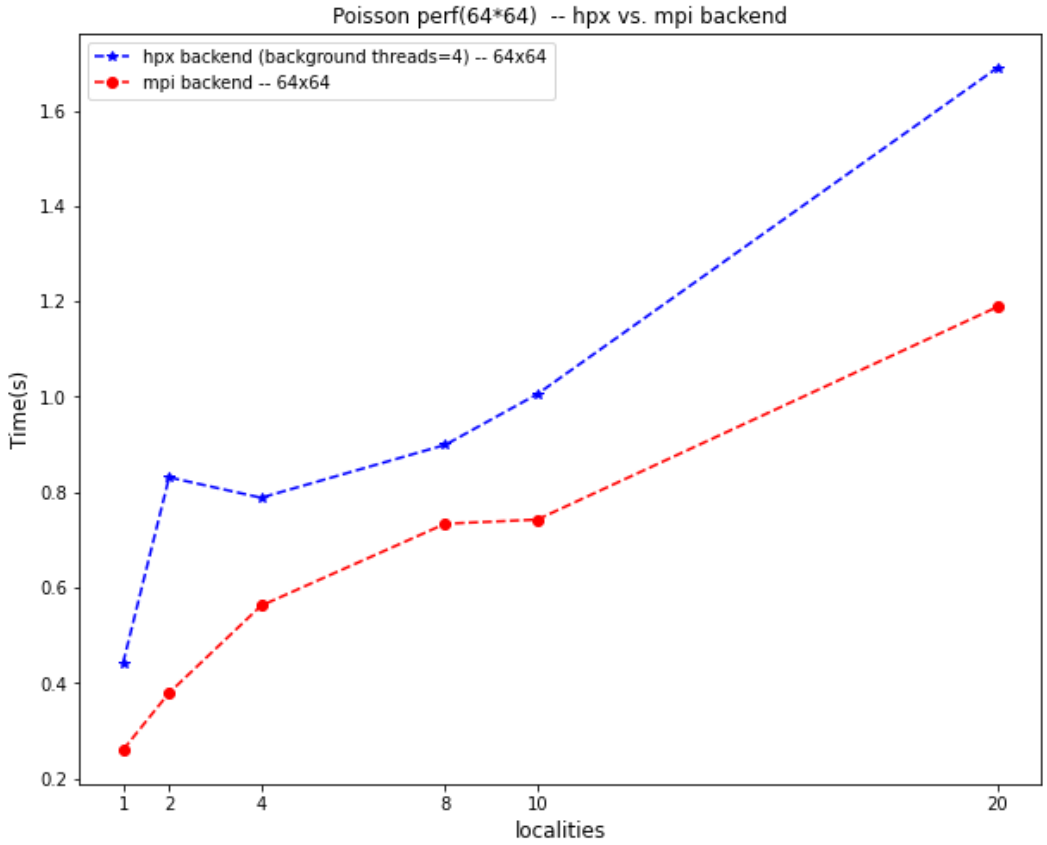


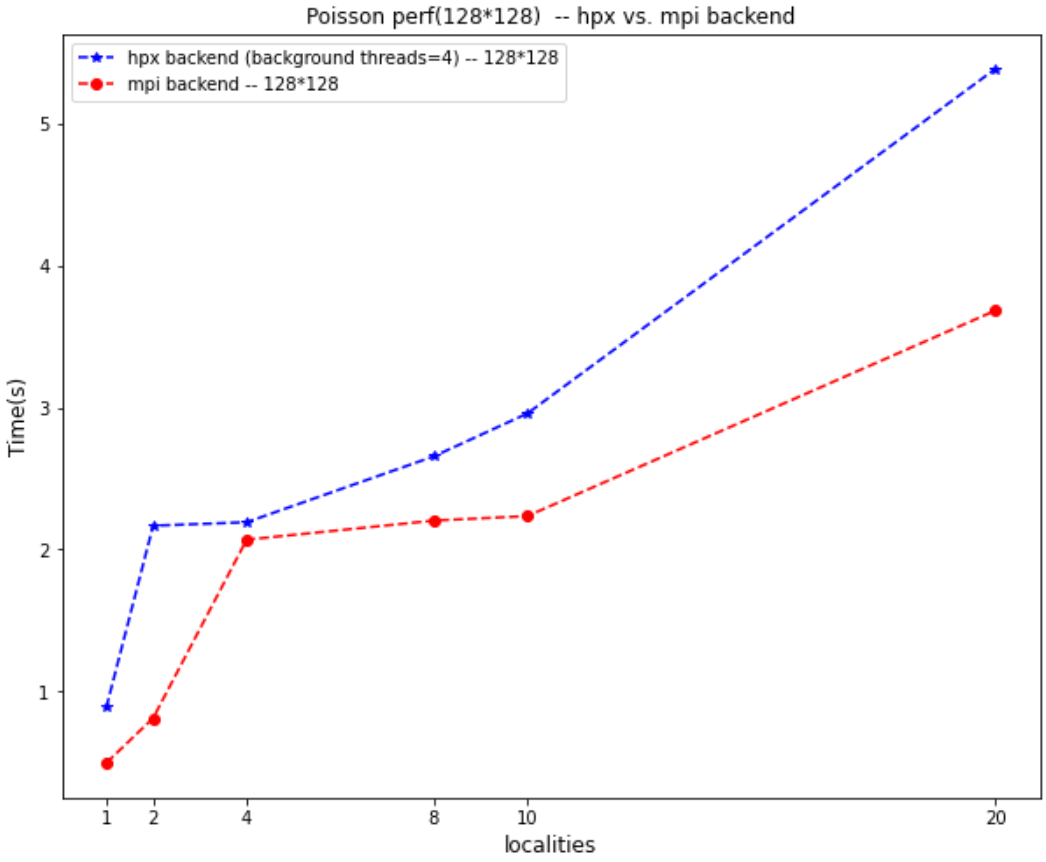
2.2 MPI backend

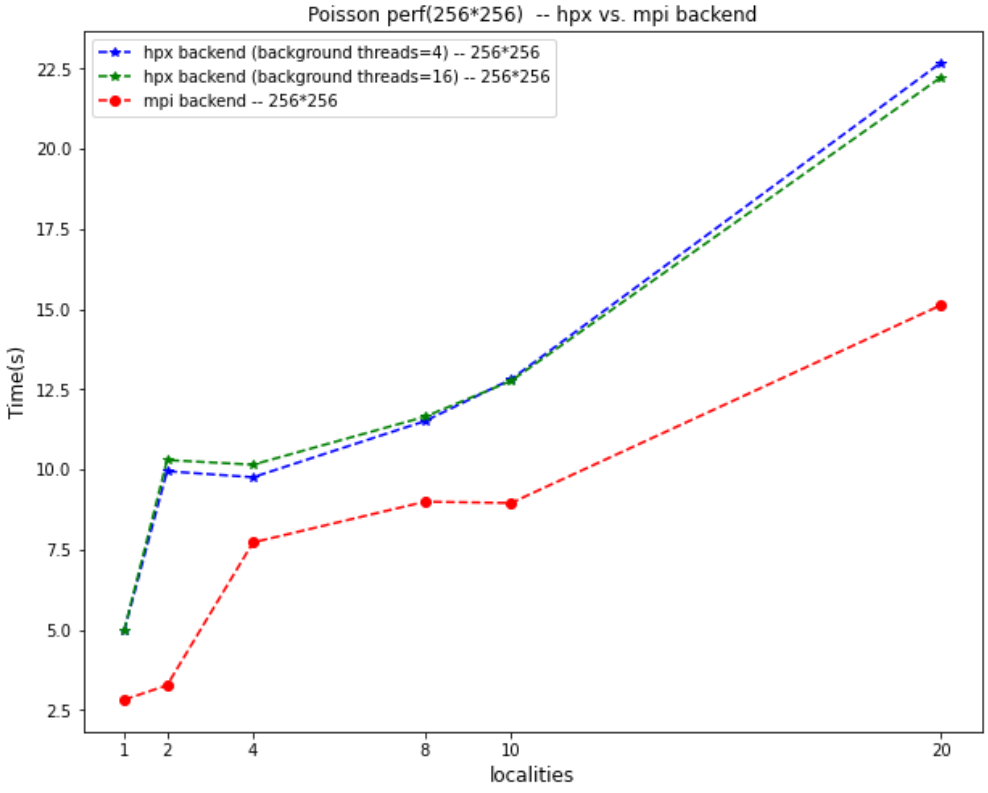




2.3 Compare the best performance of hpx backend with mpi backend




Poisson perf(128*128) -- hpx vs. mpi backend
hpx backend (background threads=4) -- 128*128
mpi backend -- 128*128
Time(s)
1
2
3
4
5
1
2
4
8
10
20
localities


Poisson perf(256*256) -- hpx vs. mpi backend
hpx backend (background threads=4) -- 256*256
hpx backend (background threads=16) -- 256*256
mpi backend -- 256*256
Time(s)
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
22.5
1
2
4
8
10
20
localities