

# benchmark\_analysis

March 15, 2023

## 1 Benchmark Analysis

```
[1]: import sys
!{sys.executable} -m pip install -r requirements.txt
```

```
Requirement already satisfied: matplotlib in /opt/homebrew/lib/python3.11/site-
packages (from -r requirements.txt (line 1)) (3.7.0)
Requirement already satisfied: numpy in /opt/homebrew/lib/python3.11/site-
packages (from -r requirements.txt (line 2)) (1.24.2)
Requirement already satisfied: pandas in /opt/homebrew/lib/python3.11/site-
packages (from -r requirements.txt (line 3)) (1.5.3)
Requirement already satisfied: contourpy>=1.0.1 in
/opt/homebrew/lib/python3.11/site-packages (from matplotlib->-r requirements.txt
(line 1)) (1.0.7)
Requirement already satisfied: cycler>=0.10 in
/opt/homebrew/lib/python3.11/site-packages (from matplotlib->-r requirements.txt
(line 1)) (0.11.0)
Requirement already satisfied: fonttools>=4.22.0 in
/opt/homebrew/lib/python3.11/site-packages (from matplotlib->-r requirements.txt
(line 1)) (4.38.0)
Requirement already satisfied: kiwisolver>=1.0.1 in
/opt/homebrew/lib/python3.11/site-packages (from matplotlib->-r requirements.txt
(line 1)) (1.4.4)
Requirement already satisfied: packaging>=20.0 in
/opt/homebrew/Cellar/jupyterlab/3.4.8_1/libexec/lib/python3.11/site-packages
(from matplotlib->-r requirements.txt (line 1)) (21.3)
Requirement already satisfied: pillow>=6.2.0 in
/opt/homebrew/lib/python3.11/site-packages (from matplotlib->-r requirements.txt
(line 1)) (9.4.0)
Requirement already satisfied: pyparsing>=2.3.1 in
/opt/homebrew/Cellar/jupyterlab/3.4.8_1/libexec/lib/python3.11/site-packages
(from matplotlib->-r requirements.txt (line 1)) (3.0.9)
Requirement already satisfied: python-dateutil>=2.7 in
/opt/homebrew/Cellar/jupyterlab/3.4.8_1/libexec/lib/python3.11/site-packages
(from matplotlib->-r requirements.txt (line 1)) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in
/opt/homebrew/Cellar/jupyterlab/3.4.8_1/libexec/lib/python3.11/site-packages
(from pandas->-r requirements.txt (line 3)) (2022.4)
```

Requirement already satisfied: six>=1.5 in  
/opt/homebrew/opt/six/lib/python3.11/site-packages (from python-  
dateutil>=2.7->matplotlib->r requirements.txt (line 1)) (1.16.0)

[notice] A new release of pip  
available: 22.3.1 -> 23.0.1  
[notice] To update, run:  
python3.11 -m pip install --upgrade pip

```
[2]: import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import statistics
import os
from pathlib import Path
from typing import List, Dict, Any, Tuple

%matplotlib inline
```

```
[3]: matplotlib.style.use('seaborn-v0_8')
```

```
[4]: root_dir = '/Users/diego/Desktop/BENCHMARK_NEBULAC_ALL'
```

```
[5]: GCC_TBB_COLOR = 'salmon'
GCC_TBB_COLOR_SECONDARY = 'sienna'

NVC_OMP_COLOR = 'green'
NVC_OMP_COLOR_SECONDARY = 'yellowgreen'

NVC_GPU_COLOR = 'beige'
```

```
[6]: plot_save_dir = './plots'
```

## 1.1 Utils

```
[7]: def get_path(*entries):
      return os.path.join(*entries)
```

```
[8]: def ensure_file_existence(output_filename):
      """
      Checks whether the path to the file exists. If not it creates the folder_
      structure and the final file.
      :param output_filename: path to the file
      :return:
      """

      # creates dirs etc if they do not exist
```

```

output_path = Path(output_filename)
if not os.path.exists(output_path.parent):
    os.makedirs(output_path.parent)
output_path.touch(exist_ok=True) # will create file, if it exists will do
↳ nothing

```

```

[9]: def plot(name:str,save=True) -> None:
    name = name.replace(' ','-')
    if save:
        plt.savefig(get_path(plot_save_dir,name+".png"))
    plt.show()

```

```

[10]: def extraction_pandas_frame_algo(path, COMP="TODO"):
    df = pd.read_csv(path)

    # dropping columns we do not care about
    df = df.drop(['iterations', 'bytes_per_second', 'items_per_second',
↳ 'label', 'error_occurred', 'error_message'],
                axis=1)

    # adding the problem size as column
    df = df[df['name'].str.endswith(('mean', 'median', 'stddev'))]
    df['n'] = df.apply(lambda x: x[0][x[0].find('/') + 1:x[0].rfind('_')],
↳ axis=1)

    df = df.reset_index(drop=True)

    # convert to format
    #
↳ name      real_time      cpu_time      time_unit      n      median      stddev
    results_gcc = df.groupby('n').apply(lambda sf: pd.Series(sf.iloc[0])).
↳ reset_index(drop=True)
    results_gcc.n = results_gcc.n.astype(int)
    results_gcc = results_gcc.sort_values(['n'], ascending=True).
↳ reset_index(drop=True)

    results_gcc['C'] = np.arange(len(results_gcc))

    results_gcc['median_id'] = results_gcc['C'] * 3 + 1
    results_gcc['median'] = results_gcc['median_id'].apply(lambda x: df.
↳ iloc[x]['real_time'])

    results_gcc['stddev_id'] = results_gcc['C'] * 3 + 2
    results_gcc['stddev'] = results_gcc['stddev_id'].apply(lambda x: df.
↳ iloc[x]['real_time'])

```

```

results_gcc = results_gcc.drop(['C', 'median_id', 'stddev_id'], axis=1)
results_gcc['Compiler'] = COMP
results_gcc['name'] = results_gcc.apply(lambda x: x[0].replace(str(x['n']), '\n'), axis=1)
↳"".replace('/_mean', ''), axis=1)

return results_gcc

```

```

[11]: # generate filename for threading
def get_threading_file_name(benchmark_name:str, thread_nr: int, input_size:str)↳
↳-> str:
    return f"[T{thread_nr}]_{benchmark_name}_{input_size}_T{thread_nr}.csv"

# extract threaded into dictionary
def extraction_pandas_frame_algo_threaded(folder_path:str, benchmark_name:str,↳
↳threads_list:List[int], input_size:int = '1048576', COMP:str="TODO") -> Any:
    result = pd.DataFrame()

    for t_id in threads_list:
        filename =↳
↳get_threading_file_name(benchmark_name=benchmark_name,thread_nr=t_id,input_size=input_size)
        file_path = get_path(folder_path,filename)

        data_frame = extraction_pandas_frame_algo(file_path,COMP=COMP)
        data_frame['threads'] = t_id
        result = pd.concat([result, data_frame], ignore_index=True)

    result = result.rename_axis(None, axis=1)
    return result

```

```

[12]: # calculate speedup based on seq runnings
def calc_speedup_based_seq(seq_df: pd.DataFrame, threads_df: pd.DataFrame,↳
↳speedup_column_name:str, input_size:int = 1048576) -> pd.DataFrame:
    # calculate speedup
    seq_df = seq_df[seq_df['n'] == input_size]
    seq_time = seq_df['real_time'].iloc[0] # now its only a single digit

    threads_df['speedup'] = seq_time / threads_df['real_time']

    # clean df
    threads_df = threads_df.
↳drop(columns=['name', 'cpu_time', 'time_unit', 'median', 'stddev', 'Compiler', 'n', 'real_time'])
    threads_df = threads_df.rename(columns={'speedup':speedup_column_name})

    return threads_df

```

```
[13]: def calc_speedup_based_par(threads_df: pd.DataFrame, speedup_column_name:str,
    ↪input_size:int = 1048576) -> pd.DataFrame:
    base_time = threads_df[threads_df['threads'] == 1].iloc[0]['real_time']

    threads_df['speedup'] = base_time / threads_df['real_time']

    # clean df
    threads_df = threads_df.
    ↪drop(columns=['name', 'cpu_time', 'time_unit', 'median', 'stddev', 'Compiler', 'n', 'real_time'])
    threads_df = threads_df.rename(columns={'speedup':speedup_column_name})

    return threads_df
```

## 2 Nebulah all Core

```
Architecture:      x86_64
CPU op-mode(s):    32-bit, 64-bit
Byte Order:        Little Endian
Address sizes:      43 bits physical, 48 bits virtual
CPU(s):            64
On-line CPU(s) list: 0-63
Thread(s) per core: 1
Core(s) per socket: 32
Socket(s):         2
NUMA node(s):      8
Vendor ID:         AuthenticAMD
CPU family:         23
Model:             1
Model name:        AMD EPYC 7551 32-Core Processor
Stepping:          2
CPU MHz:           2404.199
CPU max MHz:       2000.0000
CPU min MHz:       1200.0000
BogoMIPS:          3992.24
Virtualization:    AMD-V
L1d cache:         32K
L1i cache:         64K
L2 cache:          512K
L3 cache:          8192K
NUMA node0 CPU(s): 0,8,16,24,32,40,48,56
NUMA node1 CPU(s): 2,10,18,26,34,42,50,58
NUMA node2 CPU(s): 4,12,20,28,36,44,52,60
NUMA node3 CPU(s): 6,14,22,30,38,46,54,62
NUMA node4 CPU(s): 1,9,17,25,33,41,49,57
NUMA node5 CPU(s): 3,11,19,27,35,43,51,59
NUMA node6 CPU(s): 5,13,21,29,37,45,53,61
NUMA node7 CPU(s): 7,15,23,31,39,47,55,63
```

Flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 c

## 2.1 H1

Some parallel backends exhibit better performance and scalability when handling nested parallelism for homogeneous workloads

### 2.1.1 Time

**Time Comparison - b1\_1\_for\_each\_linear\_par** Check how the runtime without constraining the threads develops with increasing input size

```
[14]: # load data gcc (b1_1_for_each_linear_par)
b1_1_for_each_linear_par_gcc = extraction_pandas_frame_algo(root_dir + '/'
↳GCC_TBB/DEFAULT/b1_1_for_each_linear_par__Default.csv',COMP="GCC(TBB)")

b1_1_for_each_linear_par_gcc = b1_1_for_each_linear_par_gcc.
↳drop(columns=['name','cpu_time','time_unit','median','stddev','Compiler'])
b1_1_for_each_linear_par_gcc = b1_1_for_each_linear_par_gcc.
↳rename(columns={'real_time':'GCC(TBB)'})

# load data nvhpc (b1_1_for_each_linear_par)
b1_1_for_each_linear_par_nvc_omp = extraction_pandas_frame_algo(root_dir + '/'
↳NVHPC_Multicore/DEFAULT/b1_1_for_each_linear_par__Default.csv',␣
↳COMP="NVC(OMP)")

b1_1_for_each_linear_par_nvc_omp = b1_1_for_each_linear_par_nvc_omp.
↳drop(columns=['name','cpu_time','time_unit','median','stddev','Compiler'])
b1_1_for_each_linear_par_nvc_omp = b1_1_for_each_linear_par_nvc_omp.
↳rename(columns={'real_time':'NVC(OMP)'})

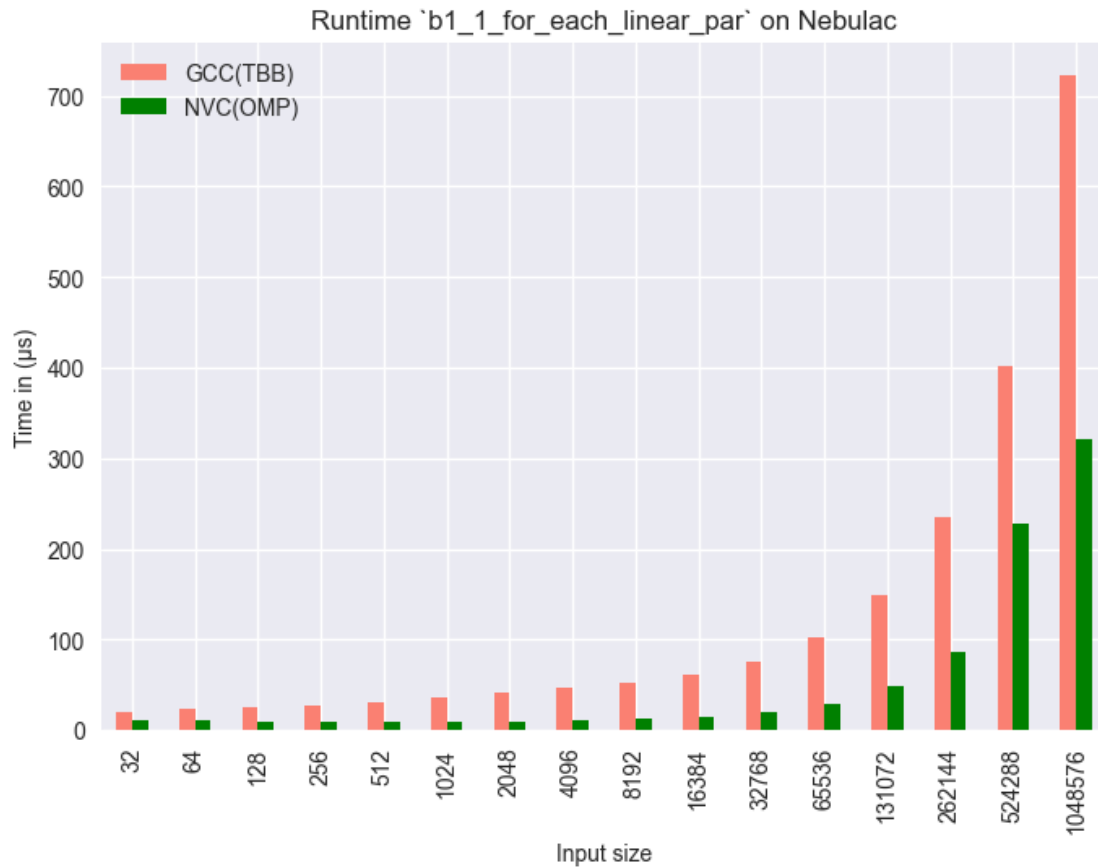
# merge for plotting
b1_1_for_each_linear_par_time_merged = pd.merge(b1_1_for_each_linear_par_gcc,␣
↳b1_1_for_each_linear_par_nvc_omp, on='n')
b1_1_for_each_linear_par_time_merged

# convert time from ns to microseconds because otherwise it will look really bad
b1_1_for_each_linear_par_time_merged['GCC(TBB)'] =␣
↳b1_1_for_each_linear_par_time_merged['GCC(TBB)'] / 1_000
b1_1_for_each_linear_par_time_merged['NVC(OMP)'] =␣
↳b1_1_for_each_linear_par_time_merged['NVC(OMP)'] / 1_000

# plot
b1_1_for_each_linear_par_time_merged.
↳plot(kind='bar',x='n',align='center',color=[GCC_TBB_COLOR,NVC_OMP_COLOR])
```

```
plt.ylabel('Time in (μs)')
plt.xlabel('Input size')
plt.title('Runtime `b1_1_for_each_linear_par` on Nebulac')

plot('Runtime `b1_1_for_each_linear_par` on Nebulac')
```



**Time Comparison - b1\_2\_for\_each\_quadratic\_outer\_std::execution::parallel\_policy\_par**  
Check how the runtime without constraining the threads develops with increasing input size

```
[15]: # load data gcc (b1_2_for_each_quadratic_outer_std::execution::
      ↪parallel_policy_par)
b1_2_for_each_quadratic_par_par_gcc = extraction_pandas_frame_algo(root_dir + '/'
      ↪GCC_TBB/DEFAULT/b1_2_for_each_quadratic_outer_std::execution::
      ↪parallel_policy_par_Default.csv', COMP="GCC(TBB)")

b1_2_for_each_quadratic_par_par_gcc = b1_2_for_each_quadratic_par_par_gcc.
      ↪drop(columns=['name', 'cpu_time', 'time_unit', 'median', 'stddev', 'Compiler'])
```

```

b1_2_for_each_quadratic_par_par_gcc = b1_2_for_each_quadratic_par_par_gcc.
↳rename(columns={'real_time': 'GCC(TBB)'})

# load data nvhpc (b1_2_for_each_quadratic_outer_std::execution::
↳parallel_policy_par)
b1_2_for_each_quadratic_par_par_nvc_omp = extraction_pandas_frame_algo(root_dir_
↳+ '/NVHPC_Multicore/DEFAULT/b1_2_for_each_quadratic_outer_std::execution::
↳parallel_policy_par_Default.csv', COMP="NVC(OMP)")

b1_2_for_each_quadratic_par_par_nvc_omp =
↳b1_2_for_each_quadratic_par_par_nvc_omp.
↳drop(columns=['name', 'cpu_time', 'time_unit', 'median', 'stddev', 'Compiler'])
b1_2_for_each_quadratic_par_par_nvc_omp =
↳b1_2_for_each_quadratic_par_par_nvc_omp.rename(columns={'real_time':
↳'NVC(OMP)'})

# merge for plotting
b1_2_for_each_quadratic_par_par_time_merged = pd.
↳merge(b1_2_for_each_quadratic_par_par_gcc,
↳b1_2_for_each_quadratic_par_par_nvc_omp, on='n')

# convert time from ns to milliseconds because otherwise it will look really bad
b1_2_for_each_quadratic_par_par_time_merged['GCC(TBB)'] =
↳b1_2_for_each_quadratic_par_par_time_merged['GCC(TBB)'] / 1_000_000
b1_2_for_each_quadratic_par_par_time_merged['NVC(OMP)'] =
↳b1_2_for_each_quadratic_par_par_time_merged['NVC(OMP)'] / 1_000_000

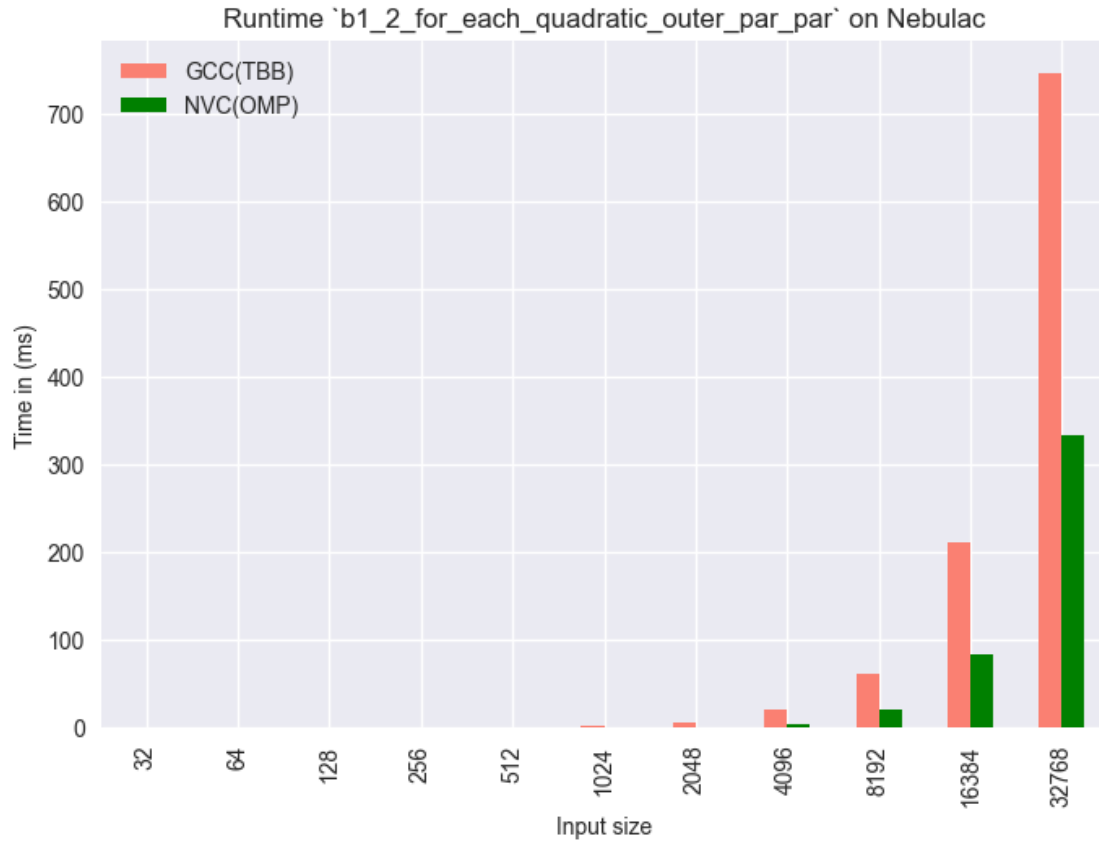
# plot
b1_2_for_each_quadratic_par_par_time_merged.
↳plot(kind='bar', x='n', align='center', color=[GCC_TBB_COLOR, NVC_OMP_COLOR])

plt.ylabel('Time in (ms)')
plt.xlabel('Input size')
plt.title('Runtime `b1_2_for_each_quadratic_outer_par_par` on Nebulac')

plot('Runtime `b1_2_for_each_quadratic_outer_par_par` on Nebulac')

```





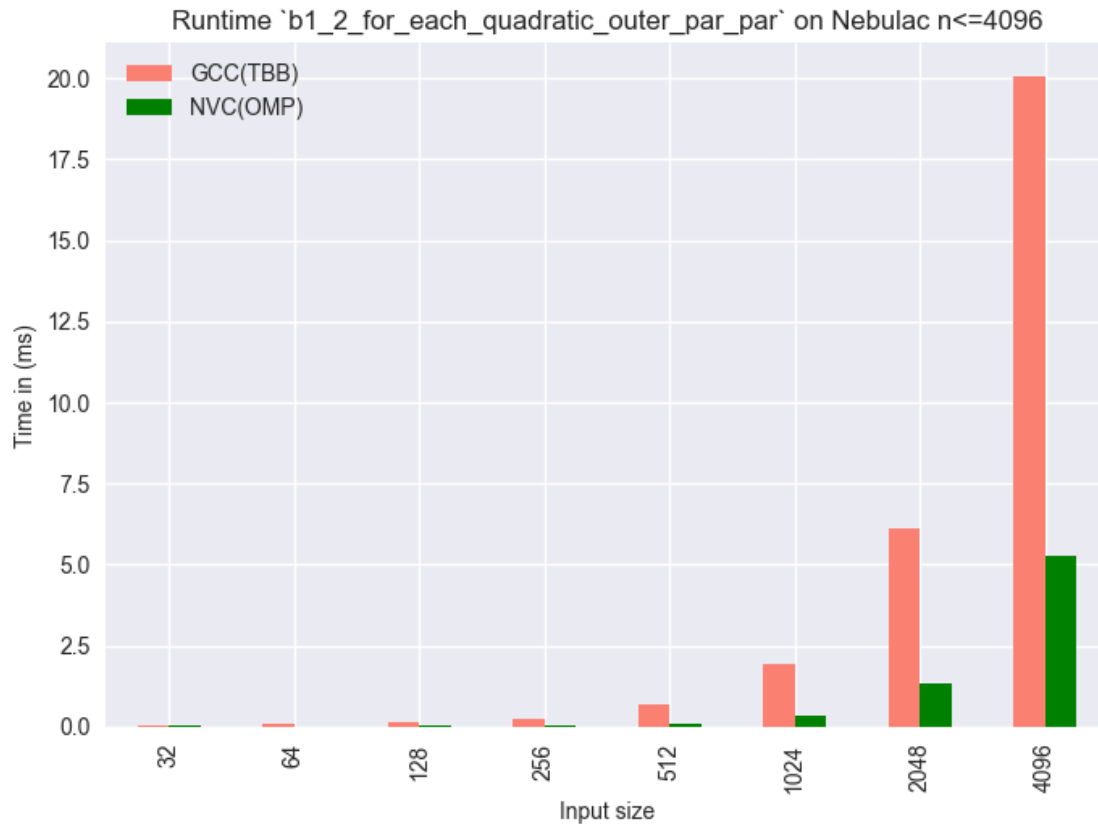
Adding a second graph because small numbers are not readable in the above graph

```
[16]: b1_2_for_each_quadratic_par_par_time_merged_sub_4096 =
↳ b1_2_for_each_quadratic_par_par_time_merged[b1_2_for_each_quadratic_par_par_time_merged['n'
↳ <= 4096]

# plot
b1_2_for_each_quadratic_par_par_time_merged_sub_4096.
↳ plot(kind='bar',x='n',align='center',color=[GCC_TBB_COLOR,NVC_OMP_COLOR])

plt.ylabel('Time in (ms)')
plt.xlabel('Input size')
plt.title('Runtime `b1_2_for_each_quadratic_outer_par_par` on Nebulac n<=4096')

plot('Runtime `b1_2_for_each_quadratic_outer_par_par` on Nebulac n<=4096')
```



**Time Comparison - b1\_4\_for\_each\_exponential\_par** Check how the runtime without constraining the threads develops with increasing input size

```
[17]: # load data gcc (b1_4_for_each_exponential_par)
b1_4_for_each_exponential_par_gcc = extraction_pandas_frame_algo(root_dir + '/'
↳GCC_TBB/DEFAULT/b1_4_for_each_exponential_par__Default.csv',COMP="GCC(TBB)")

b1_4_for_each_exponential_par_gcc = b1_4_for_each_exponential_par_gcc.
↳drop(columns=['name', 'cpu_time', 'time_unit', 'median', 'stddev', 'Compiler'])
b1_4_for_each_exponential_par_gcc = b1_4_for_each_exponential_par_gcc.
↳rename(columns={'real_time': 'GCC(TBB)'})

# load data nvhpc (b1_4_for_each_exponential_par)
b1_4_for_each_exponential_par_nvc_omp = extraction_pandas_frame_algo(root_dir +
↳'/NVHPC_Multicore/DEFAULT/b1_4_for_each_exponential_par__Default.
↳csv',COMP="NVC(OMP)")

b1_4_for_each_exponential_par_nvc_omp = b1_4_for_each_exponential_par_nvc_omp.
↳drop(columns=['name', 'cpu_time', 'time_unit', 'median', 'stddev', 'Compiler'])
```

```

b1_4_for_each_exponential_par_nvc_omp = b1_4_for_each_exponential_par_nvc_omp.
↳rename(columns={'real_time': 'NVC(OMP)'})

# merge for plotting
b1_4_for_each_exponential_par_time_merged = pd.
↳merge(b1_4_for_each_exponential_par_gcc,↳
↳b1_4_for_each_exponential_par_nvc_omp, on='n')

# convert time from ns to milliseconds because otherwise it will look really bad
b1_4_for_each_exponential_par_time_merged['GCC(TBB)'] =↳
↳b1_4_for_each_exponential_par_time_merged['GCC(TBB)'] / 1_000_000
b1_4_for_each_exponential_par_time_merged['NVC(OMP)'] =↳
↳b1_4_for_each_exponential_par_time_merged['NVC(OMP)'] / 1_000_000

print(b1_4_for_each_exponential_par_time_merged)

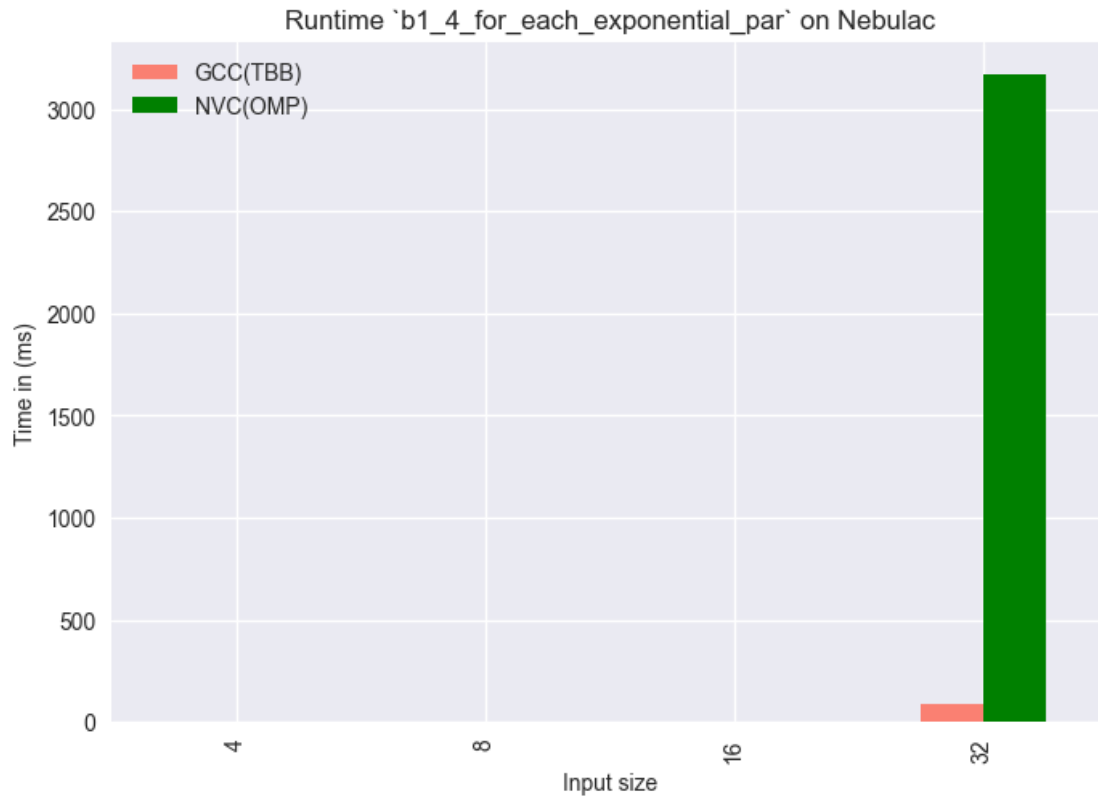
# plot
b1_4_for_each_exponential_par_time_merged.
↳plot(kind='bar',x='n',align='center',color=[GCC_TBB_COLOR,NVC_OMP_COLOR])

plt.ylabel('Time in (ms)')
plt.xlabel('Input size')
plt.title('Runtime `b1_4_for_each_exponential_par` on Nebulac')

plot('Runtime `b1_4_for_each_exponential_par` on Nebulac')

```

	GCC(TBB)	n	NVC(OMP)
0	0.018690	4	0.012494
1	0.121331	8	0.042570
2	0.537180	16	1.722970
3	92.126600	32	3170.760000



Adding a second graph because small numbers are not readable in the above graph

```
[18]: b1_4_for_each_exponential_par_time_merged_sub_16 =
    ↳ b1_4_for_each_exponential_par_time_merged[b1_4_for_each_exponential_par_time_merged['n']]
    ↳ <= 16]

# convert from milliseconds to microseconds
b1_4_for_each_exponential_par_time_merged_sub_16['GCC(TBB)'] =
    ↳ b1_4_for_each_exponential_par_time_merged_sub_16['GCC(TBB)'] * 1_000
b1_4_for_each_exponential_par_time_merged_sub_16['NVC(OMP)'] =
    ↳ b1_4_for_each_exponential_par_time_merged_sub_16['NVC(OMP)'] * 1_000

# plot
b1_4_for_each_exponential_par_time_merged_sub_16.
    ↳ plot(kind='bar', x='n', align='center', color=[GCC_TBB_COLOR, NVC_OMP_COLOR])

print(b1_4_for_each_exponential_par_time_merged_sub_16)

plt.ylabel('Time in (µs)')
plt.xlabel('Input size')
```

```
plt.title('Runtime `b1_4_for_each_exponential_par` on Nebulac n<=16')
plot('Runtime `b1_4_for_each_exponential_par` on Nebulac n<=16')
```

	GCC(TBB)	n	NVC(OMP)
0	18.6901	4	12.4941
1	121.3310	8	42.5701
2	537.1800	16	1722.9700

```
/var/folders/42/fk0jfryd1dd1ztdldncqc1cw0000gn/T/ipykernel_33509/2823210147.py:4
```

```
: SettingWithCopyWarning:
```

```
A value is trying to be set on a copy of a slice from a DataFrame.
```

```
Try using .loc[row_indexer,col_indexer] = value instead
```

```
See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
```

```
    b1_4_for_each_exponential_par_time_merged_sub_16['GCC(TBB)'] =
b1_4_for_each_exponential_par_time_merged_sub_16['GCC(TBB)'] * 1_000
```

```
/var/folders/42/fk0jfryd1dd1ztdldncqc1cw0000gn/T/ipykernel_33509/2823210147.py:5
```

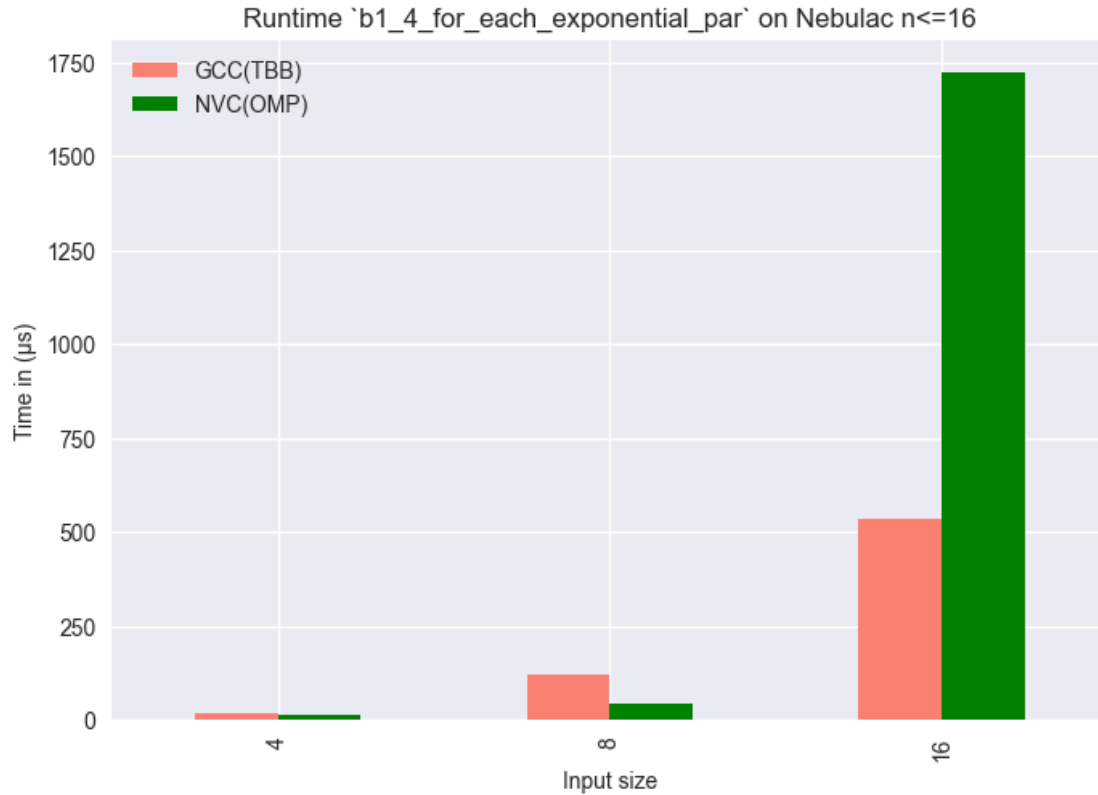
```
: SettingWithCopyWarning:
```

```
A value is trying to be set on a copy of a slice from a DataFrame.
```

```
Try using .loc[row_indexer,col_indexer] = value instead
```

```
See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
```

```
    b1_4_for_each_exponential_par_time_merged_sub_16['NVC(OMP)'] =
b1_4_for_each_exponential_par_time_merged_sub_16['NVC(OMP)'] * 1_000
```



### 2.1.2 Strong Scaling

$$S(p) = T(1) / T(p)$$

As based we use once the: \* sequential algorithm \* parallel algorithm (1 thread)

**Strong Scaling - b1\_1\_for\_each\_linear** 1 Million fixed input size with threads 1-64

Seq Base

```
[19]: # GCC
## load gcc (b1_1_for_each_linear_seq)
b1_1_for_each_linear_seq_gcc = extraction_pandas_frame_algo(root_dir + '/'
↳GCC_TBB/DEFAULT/b1_1_for_each_linear_seq_Default.csv',COMP="GCC(TBB)")

## load gcc threaded b1_1_for_each_linear_par
b1_1_for_each_linear_threads_gcc =↳
↳extraction_pandas_frame_algo_threaded(root_dir + '/GCC_TBB/
↳THREADS','b1_1_for_each_linear_par',[1,2,4,8,16,32,64],COMP="GCC(TBB)")

## calculate speedup
b1_1_for_each_linear_strong_scaling_seqbase_gcc =↳
↳calc_speedup_based_seq(b1_1_for_each_linear_seq_gcc,b1_1_for_each_linear_threads_gcc,"GCC(TBB)")
```

```

# NVC(OMP)
## load nvhpc (b1_1_for_each_linear_seq)
b1_1_for_each_linear_seq_nvc_omp = extraction_pandas_frame_algo(root_dir + '/'
↳NVHPC_Multicore/DEFAULT/b1_1_for_each_linear_seq_Default.
↳csv',COMP="NVC(OMP)")

## load nvhpc threaded b1_1_for_each_linear_par
b1_1_for_each_linear_threads_nvc_omp =↳
↳extraction_pandas_frame_algo_threaded(root_dir + '/NVHPC_Multicore/
↳THREADS', 'b1_1_for_each_linear_par', [1,2,4,8,16,32,64],COMP="NVC(OMP)")

## calculate speedup
b1_1_for_each_linear_strong_scaling_seqbase_nvc_omp =↳
↳calc_speedup_based_seq(b1_1_for_each_linear_seq_nvc_omp,b1_1_for_each_linear_threads_nvc_omp)

# merge for plotting
b1_1_for_each_linear_seq_speedup_merged = pd.
↳merge(b1_1_for_each_linear_strong_scaling_seqbase_gcc,↳
↳b1_1_for_each_linear_strong_scaling_seqbase_nvc_omp, on='threads')
print(b1_1_for_each_linear_seq_speedup_merged)

# plot strong scaling
ax = b1_1_for_each_linear_seq_speedup_merged.
↳plot(kind='bar',x='threads',align='center',color=[GCC_TBB_COLOR,NVC_OMP_COLOR])

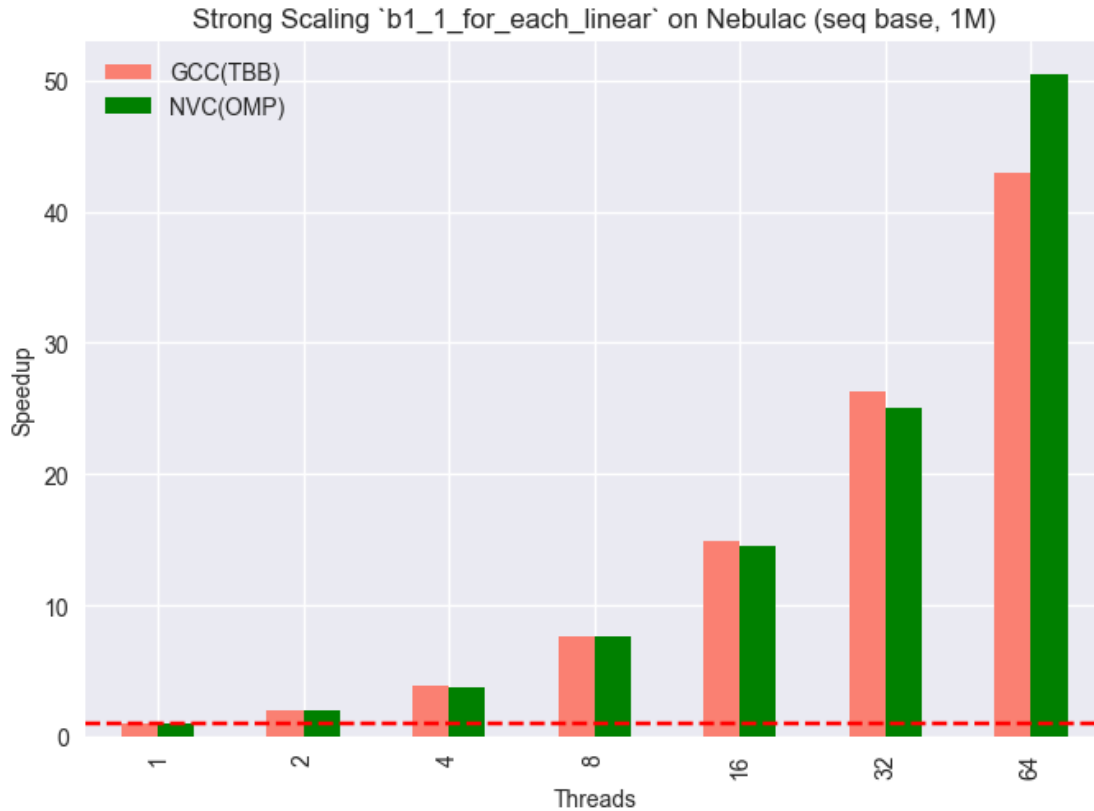
# adding horizontal line where there is speedup
ax.axhline(y=1, color='r', linestyle='--')

plt.ylabel('Speedup')
plt.xlabel('Threads')
plt.title('Strong Scaling `b1_1_for_each_linear` on Nebulac (seq base, 1M)')

plot('Strong Scaling `b1_1_for_each_linear` on Nebulac (seq base, 1M)')

```

	threads	GCC(TBB)	NVC(OMP)
0	1	0.997915	0.987061
1	2	1.993628	1.971258
2	4	3.966795	3.812405
3	8	7.715379	7.682825
4	16	14.945539	14.531737
5	32	26.258819	25.007096
6	64	42.943653	50.463093



```
[20]: ## efficiency graph
b1_1_for_each_linear_seq_efficiency = b1_1_for_each_linear_seq_speedup_merged.
    ↳copy()
b1_1_for_each_linear_seq_efficiency['GCC(TBB)'] =
    ↳b1_1_for_each_linear_seq_efficiency['GCC(TBB)'] /
    ↳b1_1_for_each_linear_seq_efficiency['threads']
b1_1_for_each_linear_seq_efficiency['NVC(OMP)'] =
    ↳b1_1_for_each_linear_seq_efficiency['NVC(OMP)'] /
    ↳b1_1_for_each_linear_seq_efficiency['threads']

print(b1_1_for_each_linear_seq_efficiency)

# plot efficiency
ax = b1_1_for_each_linear_seq_efficiency.
    ↳plot(x='threads',color=[GCC_TBB_COLOR,NVC_OMP_COLOR])

# adding horizontal line where there is speedup
ax.axhline(y=1, color='r', linestyle='--')

plt.ylim(0.4,1.05)
```



```

plt.xscale('log', base=2)
current_values = plt.gca().get_xticks()
plt.gca().set_xticklabels(['{:, .0f}'.format(x) for x in current_values])

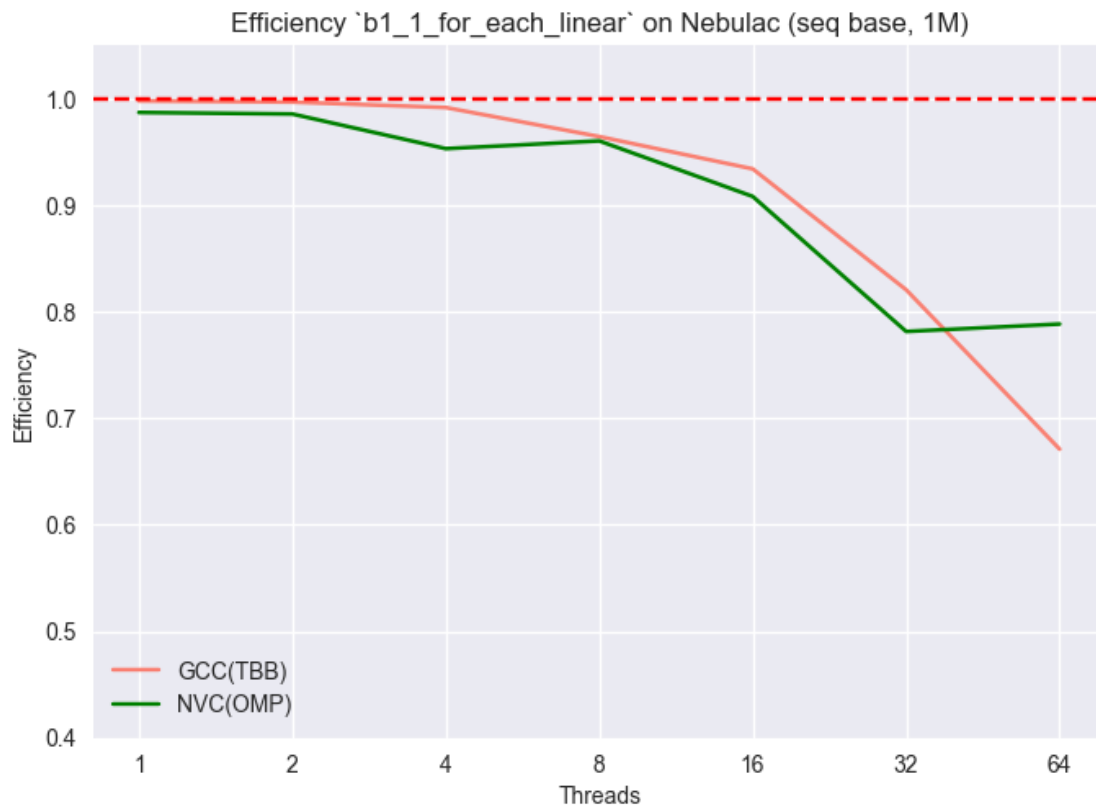
plt.ylabel('Efficiency')
plt.xlabel('Threads')
plt.title('Efficiency `b1_1_for_each_linear` on Nebulac (seq base, 1M)')

plot('Efficiency `b1_1_for_each_linear` on Nebulac (seq base, 1M)')

```

	threads	GCC(TBB)	NVC(OMP)
0	1	0.997915	0.987061
1	2	0.996814	0.985629
2	4	0.991699	0.953101
3	8	0.964422	0.960353
4	16	0.934096	0.908234
5	32	0.820588	0.781472
6	64	0.670995	0.788486

/var/folders/42/fk0jfryd1dd1ztdldncqc1cw0000gn/T/ipykernel\_33509/3891040355.py:1  
8: UserWarning: FixedFormatter should only be used together with FixedLocator  
plt.gca().set\_xticklabels(['{:, .0f}'.format(x) for x in current\_values])



## Par(1) Base

```
[21]: # GCC(TBB)
## load gcc threaded b1_1_for_each_linear_par
b1_1_for_each_linear_threads_gcc = □
↳extraction_pandas_frame_algo_threaded(root_dir + '/GCC_TBB/
↳THREADS', 'b1_1_for_each_linear_par', [1,2,4,8,16,32,64], COMP="GCC(TBB)")

## calc strong scaling
b1_1_for_each_linear_strong_scaling_parbase_gcc = □
↳calc_speedup_based_par(b1_1_for_each_linear_threads_gcc, "GCC(TBB)")

# NVC(OMP)
## load nvhpc threaded b1_1_for_each_linear_par
b1_1_for_each_linear_threads_nvc_omp = □
↳extraction_pandas_frame_algo_threaded(root_dir + '/NVHPC_Multicore/
↳THREADS', 'b1_1_for_each_linear_par', [1,2,4,8,16,32,64], COMP="NVC(OMP)")

## calc strong scaling
b1_1_for_each_linear_strong_scaling_parbase_nvc_omp = □
↳calc_speedup_based_par(b1_1_for_each_linear_threads_nvc_omp, "NVC(OMP)")

# merge for plotting
b1_1_for_each_linear_seq_parbase_speedup_merged = pd.
↳merge(b1_1_for_each_linear_strong_scaling_parbase_gcc, □
↳b1_1_for_each_linear_strong_scaling_parbase_nvc_omp, on='threads')
print(b1_1_for_each_linear_seq_parbase_speedup_merged)

# plot strong scaling
ax = b1_1_for_each_linear_seq_parbase_speedup_merged.
↳plot(kind='bar', x='threads', align='center', color=[GCC_TBB_COLOR, NVC_OMP_COLOR])

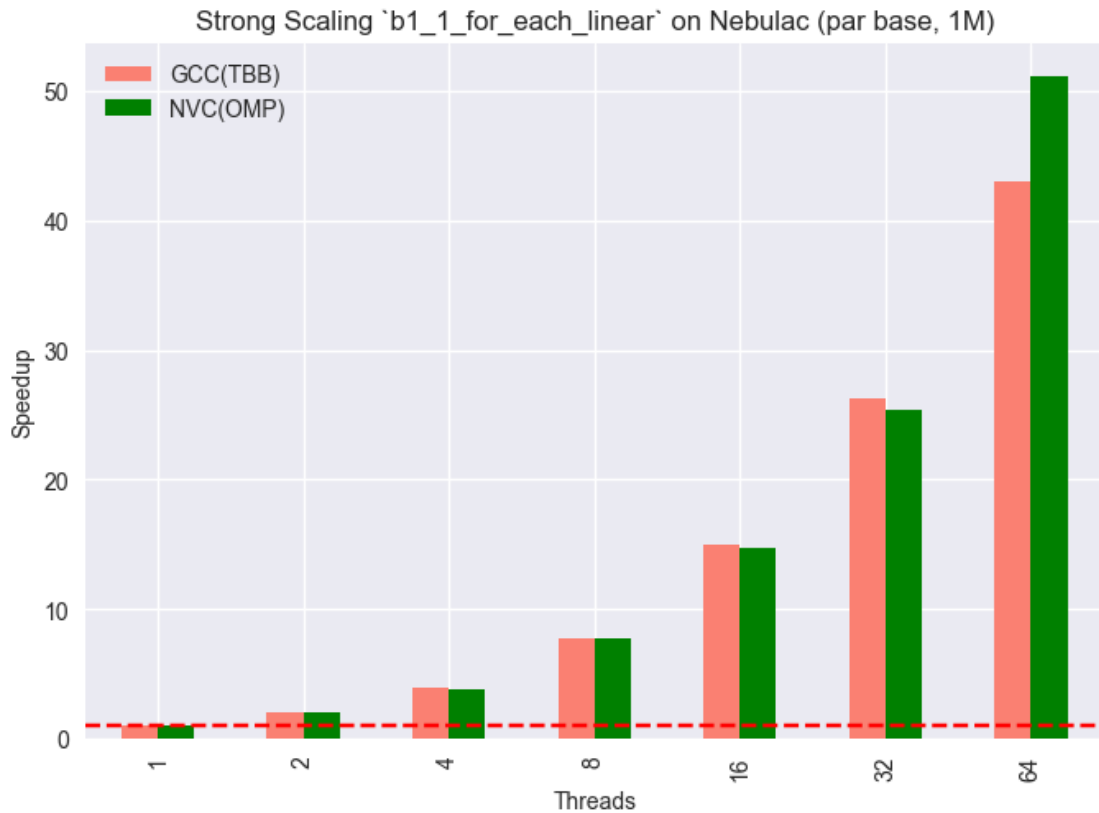
# adding horizontal line where there is speedup
ax.axhline(y=1, color='r', linestyle='--')

plt.ylabel('Speedup')
plt.xlabel('Threads')
plt.title('Strong Scaling `b1_1_for_each_linear` on Nebulac (par base, 1M)')

plot('Strong Scaling `b1_1_for_each_linear` on Nebulac (par base, 1M)')
```

	threads	GCC(TBB)	NVC(OMP)
0	1	1.000000	1.000000
1	2	1.997794	1.997099

2	4	3.975084	3.862382
3	8	7.731503	7.783538
4	16	14.976772	14.722232
5	32	26.313694	25.334912
6	64	43.033396	51.124609



```
[22]: ## efficiency graph
b1_1_for_each_linear_seq_efficiency =
    ↳ b1_1_for_each_linear_seq_parbase_speedup_merged.copy()
b1_1_for_each_linear_seq_efficiency['GCC(TBB)'] =
    ↳ b1_1_for_each_linear_seq_efficiency['GCC(TBB)'] /
    ↳ b1_1_for_each_linear_seq_efficiency['threads']
b1_1_for_each_linear_seq_efficiency['NVC(OMP)'] =
    ↳ b1_1_for_each_linear_seq_efficiency['NVC(OMP)'] /
    ↳ b1_1_for_each_linear_seq_efficiency['threads']

print(b1_1_for_each_linear_seq_efficiency)

# plot efficiency
ax = b1_1_for_each_linear_seq_efficiency.
    ↳ plot(x='threads', color=[GCC_TBB_COLOR, NVC_OMP_COLOR])
```

```

# adding horizontal line where there is speedup
ax.axhline(y=1, color='r', linestyle='--')

plt.ylim(0.4,1.05)

plt.xscale('log', base=2)
current_values = plt.gca().get_xticks()
plt.gca().set_xticklabels(['{:, .0f}'.format(x) for x in current_values])

plt.ylabel('Efficiency')
plt.xlabel('Threads')
plt.title('Efficiency `b1_1_for_each_linear` on Nebulac (par base, 1M)')

plot('Efficiency `b1_1_for_each_linear` on Nebulac (par base, 1M)')

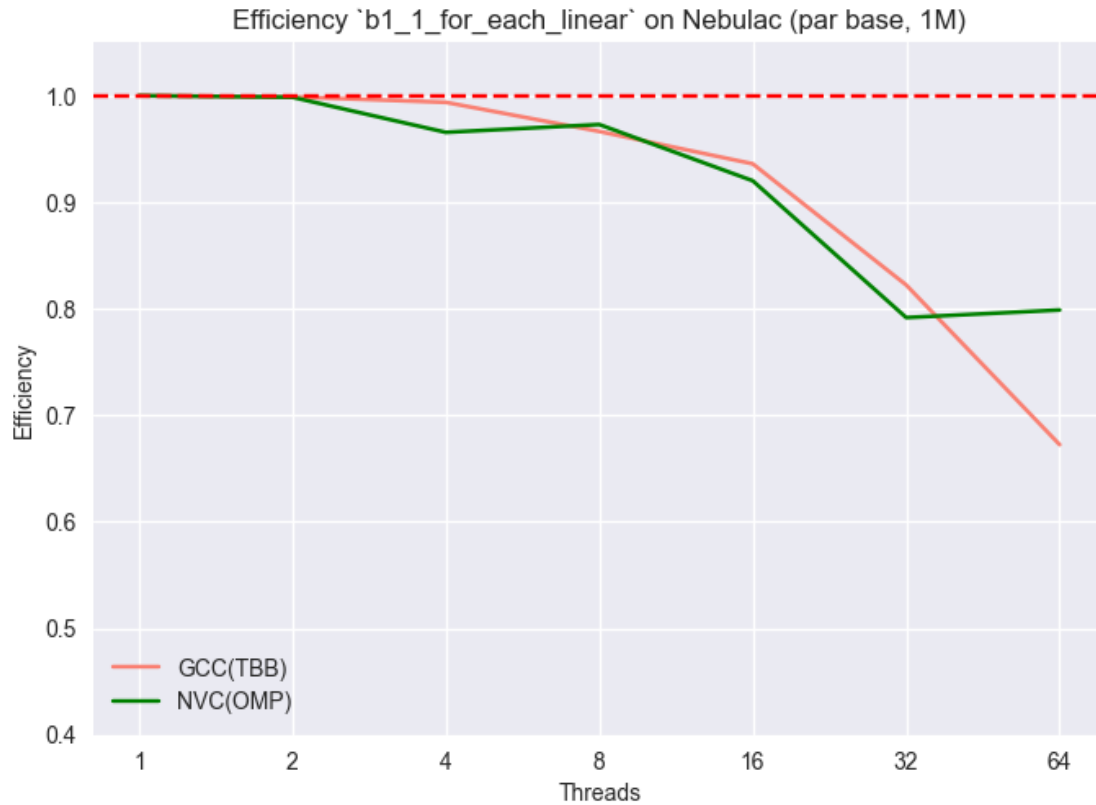
```

	threads	GCC(TBB)	NVC(OMP)
0	1	1.000000	1.000000
1	2	0.998897	0.998549
2	4	0.993771	0.965595
3	8	0.966438	0.972942
4	16	0.936048	0.920140
5	32	0.822303	0.791716
6	64	0.672397	0.798822

```

/var/folders/42/fk0jfryd1dd1ztdldncqc1cw0000gn/T/ipykernel_33509/4232767085.py:1
8: UserWarning: FixedFormatter should only be used together with FixedLocator
plt.gca().set_xticklabels(['{:, .0f}'.format(x) for x in current_values])

```



**Strong Scaling - b1\_2\_for\_each\_quadratic** 1 Million fixed input size with threads 1-64

**Seq Base** Here we wont do it with seq base because its not really realistic

**Par(1) Base**

```
[23]: # GCC(TBB)
## load gcc threaded b1_2_for_each_quadratic_outer_std::execution::
    ↪parallel_policy_par
b1_2_for_each_quadratic_par_par_threads_gcc = □
    ↪extraction_pandas_frame_algo_threaded(root_dir + '/GCC_TBB/
    ↪THREADS','b1_2_for_each_quadratic_outer_std::execution::
    ↪parallel_policy_par',[1,2,4,8,16,32,64],COMP="GCC(TBB)",input_size=32768)

## calc strong scaling
b1_2_for_each_quadratic_par_par_scaling_parbase_gcc = □
    ↪calc_speedup_based_par(b1_2_for_each_quadratic_par_par_threads_gcc,"GCC(TBB)")

# NVC(OMP)
## load nvhpc threaded b1_2_for_each_quadratic_outer_std::execution::
    ↪parallel_policy_par
```

```

b1_2_for_each_quadratic_par_par_threads_nvc_omp =
↳extraction_pandas_frame_algo_threaded(root_dir + '/NVHPC_Multicore/
↳THREADS', 'b1_2_for_each_quadratic_outer_std::execution::
↳parallel_policy_par', [1,2,4,8,16,32,64], COMP="NVC(OMP)", input_size=32768)

## calc strong scaling
b1_2_for_each_quadratic_par_par_scaling_parbase_nvc_omp =
↳calc_speedup_based_par(b1_2_for_each_quadratic_par_par_threads_nvc_omp, "NVC(OMP)")

# merge for plotting
b1_2_for_each_quadratic_par_par_speedup_merged = pd.
↳merge(b1_2_for_each_quadratic_par_par_scaling_parbase_gcc,
↳b1_2_for_each_quadratic_par_par_scaling_parbase_nvc_omp, on='threads')
print(b1_2_for_each_quadratic_par_par_speedup_merged)

# plot strong scaling
ax = b1_2_for_each_quadratic_par_par_speedup_merged.
↳plot(kind='bar', x='threads', align='center', color=[GCC_TBB_COLOR, NVC_OMP_COLOR])

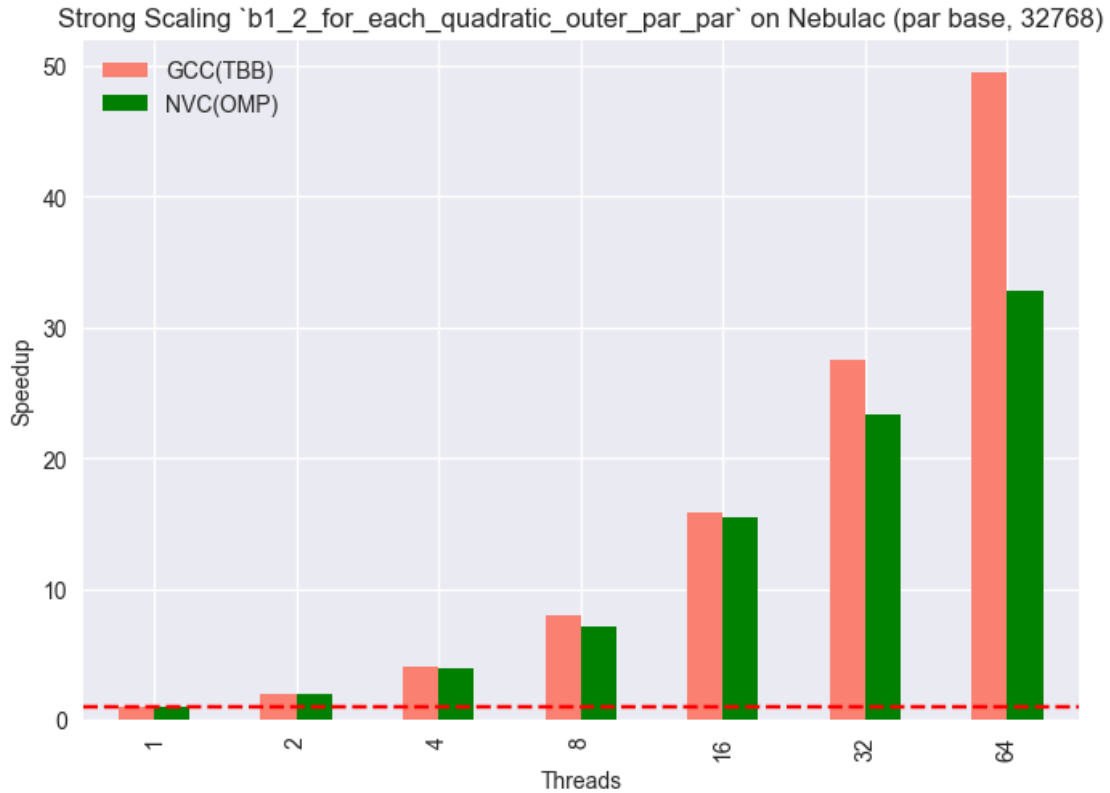
# adding horizontal line where there is speedup
ax.axhline(y=1, color='r', linestyle='--')

plt.ylabel('Speedup')
plt.xlabel('Threads')
plt.title('Strong Scaling `b1_2_for_each_quadratic_outer_par_par` on Nebulac
↳(par base, 32768)')

plot('Strong Scaling `b1_2_for_each_quadratic_outer_par_par` on Nebulac (par
↳base, 32768)')

```

	threads	GCC(TBB)	NVC(OMP)
0	1	1.000000	1.000000
1	2	2.012084	1.989333
2	4	4.028875	3.965606
3	8	8.029452	7.085557
4	16	15.840287	15.510771
5	32	27.573592	23.361771
6	64	49.508283	32.807058



```
[24]: ## efficiency graph
b1_2_for_each_quadratic_par_par_efficiency =_
↳ b1_2_for_each_quadratic_par_par_speedup_merged.copy()
b1_2_for_each_quadratic_par_par_efficiency['GCC(TBB)'] =_
↳ b1_2_for_each_quadratic_par_par_efficiency['GCC(TBB)'] /_
↳ b1_2_for_each_quadratic_par_par_efficiency['threads']
b1_2_for_each_quadratic_par_par_efficiency['NVC(OMP)'] =_
↳ b1_2_for_each_quadratic_par_par_efficiency['NVC(OMP)'] /_
↳ b1_2_for_each_quadratic_par_par_efficiency['threads']

print(b1_2_for_each_quadratic_par_par_efficiency)

# plot efficiency
ax = b1_2_for_each_quadratic_par_par_efficiency.
↳ plot(x='threads', color=[GCC_TBB_COLOR, NVC_OMP_COLOR])

# adding horizontal line where there is speedup
ax.axhline(y=1, color='r', linestyle='--')

plt.ylim(0.4, 1.05)
```

```

plt.xscale('log', base=2)
current_values = plt.gca().get_xticks()
plt.gca().set_xticklabels(['{:,.0f}'.format(x) for x in current_values])

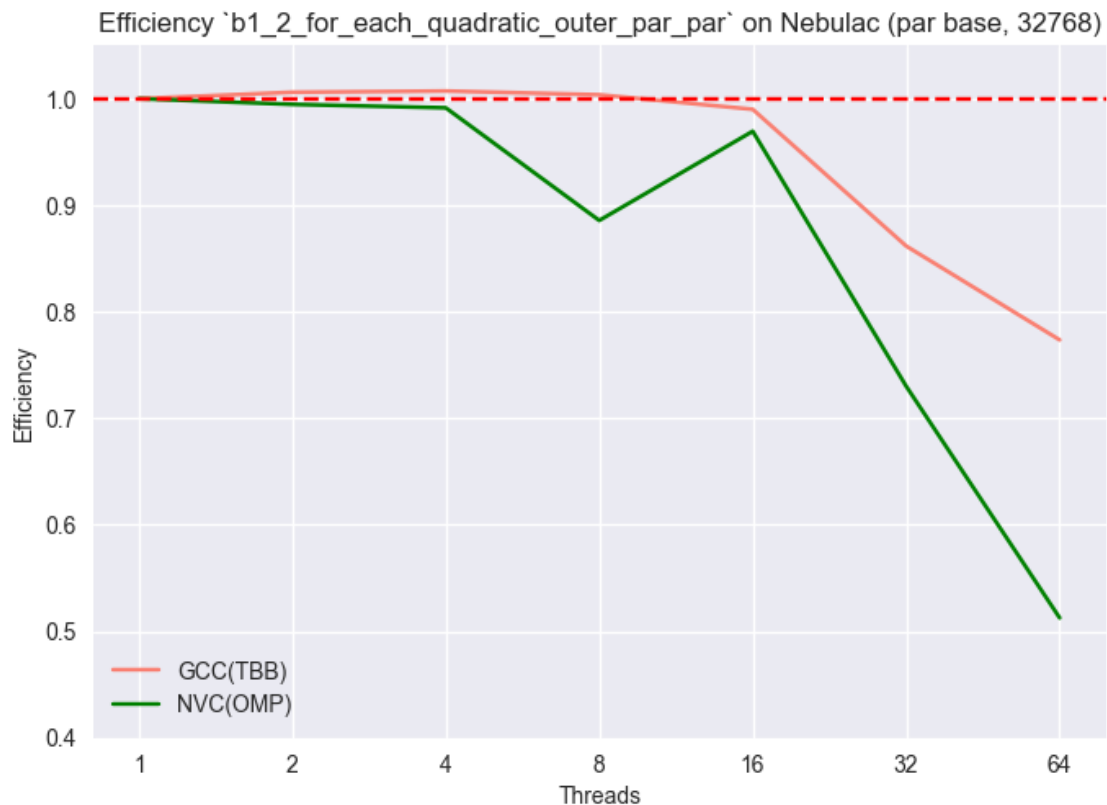
plt.ylabel('Efficiency')
plt.xlabel('Threads')
plt.title('Efficiency `b1_2_for_each_quadratic_outer_par_par` on Nebulac (par_
↪base, 32768)')

plot('Efficiency `b1_2_for_each_quadratic_outer_par_par` on Nebulac (par base,
↪32768)')

```

	threads	GCC(TBB)	NVC(OMP)
0	1	1.000000	1.000000
1	2	1.006042	0.994667
2	4	1.007219	0.991402
3	8	1.003681	0.885695
4	16	0.990018	0.969423
5	32	0.861675	0.730055
6	64	0.773567	0.512610

/var/folders/42/fk0jfryd1dd1ztdldncqc1cw0000gn/T/ipykernel\_33509/2905771063.py:1  
8: UserWarning: FixedFormatter should only be used together with FixedLocator  
plt.gca().set\_xticklabels(['{:,.0f}'.format(x) for x in current\_values])





Strong Scaling - b1\_4\_for\_each\_exponential 32 fixed input size with threads 1-64

### Seq Base

[25]:

```
# GCC
## load gcc (b1_4_for_each_exponential_seq)
b1_4_for_each_exponential_seq_gcc = extraction_pandas_frame_algo(root_dir + '/'
    ↪GCC_TBB/DEFAULT/b1_4_for_each_exponential_seq__Default.csv', COMP="GCC(TBB)")

## load gcc threaded b1_4_for_each_exponential_par
b1_4_for_each_exponential_threads_gcc =
    ↪extraction_pandas_frame_algo_threaded(root_dir + '/GCC_TBB/
    ↪THREADS', 'b1_4_for_each_exponential_par', [1,2,4,8,16,32,64], COMP="GCC(TBB)", input_size=32)

## calculate speedup
b1_4_for_each_exponential_strong_scaling_seqbase_gcc =
    ↪calc_speedup_based_seq(b1_4_for_each_exponential_seq_gcc, b1_4_for_each_exponential_threads_gcc)

# NVC(OMP)
## load nvhpc (b1_4_for_each_exponential_seq)
b1_4_for_each_exponential_seq_nvc_omp = extraction_pandas_frame_algo(root_dir +
    ↪'/NVHPC_Multicore/DEFAULT/b1_4_for_each_exponential_seq__Default.
    ↪csv', COMP="NVC(OMP)")

## load nvhpc threaded b1_4_for_each_exponential_par
b1_4_for_each_exponential_threads_nvc_omp =
    ↪extraction_pandas_frame_algo_threaded(root_dir + '/NVHPC_Multicore/
    ↪THREADS', 'b1_4_for_each_exponential_par', [1,2,4,8,16,32,64], COMP="NVC(OMP)", input_size=32)

## calculate speedup
b1_4_for_each_exponential_strong_scaling_seqbase_nvc_omp =
    ↪calc_speedup_based_seq(b1_4_for_each_exponential_seq_nvc_omp, b1_4_for_each_exponential_threads_nvc_omp)

# merge for plotting
b1_4_for_each_exponential_seq_speedup_merged = pd.
    ↪merge(b1_4_for_each_exponential_strong_scaling_seqbase_gcc,
    ↪b1_4_for_each_exponential_strong_scaling_seqbase_nvc_omp, on='threads')
print(b1_4_for_each_exponential_seq_speedup_merged)

# plot strong scaling
ax = b1_4_for_each_exponential_seq_speedup_merged.
    ↪plot(kind='bar', x='threads', align='center', color=[GCC_TBB_COLOR, NVC_OMP_COLOR])
```

```

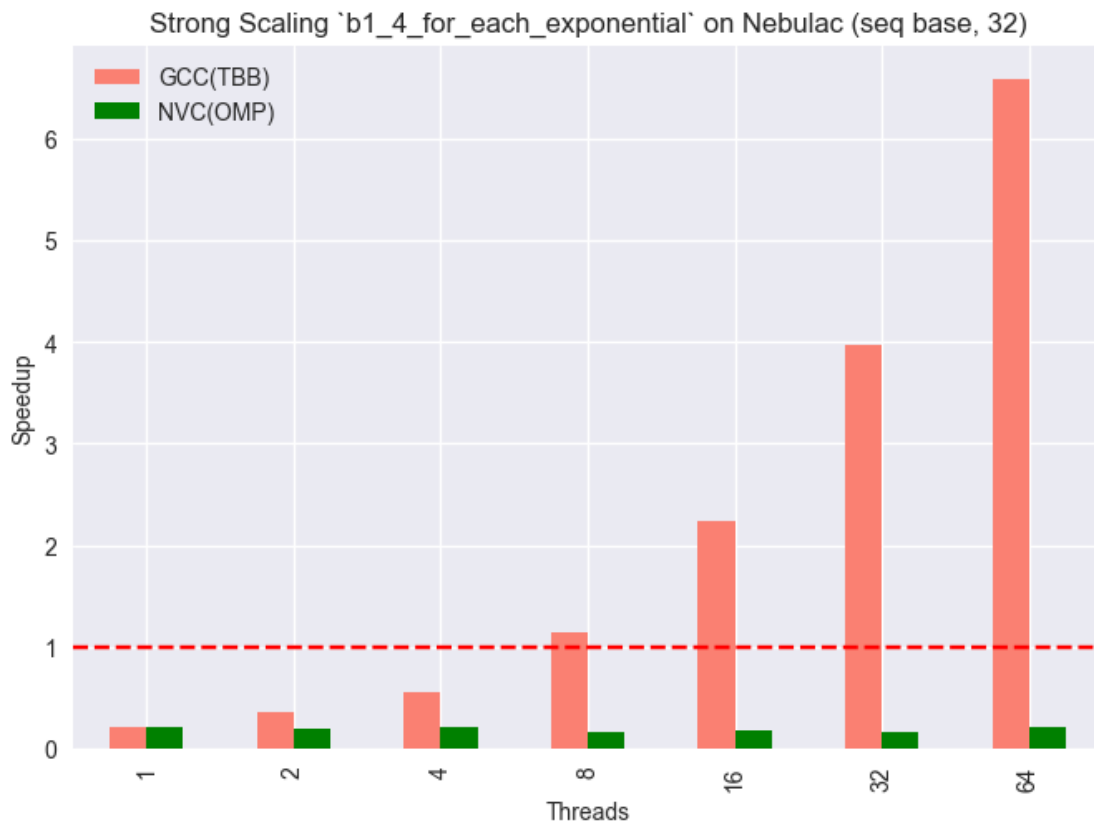
# adding horizontal line where there is speedup
ax.axhline(y=1, color='r', linestyle='--')

plt.ylabel('Speedup')
plt.xlabel('Threads')
plt.title('Strong Scaling `b1_4_for_each_exponential` on Nebulac (seq base, 32)')

plot('Strong Scaling `b1_4_for_each_exponential` on Nebulac (seq base, 32)')

```

	threads	GCC(TBB)	NVC(OMP)
0	1	0.220355	0.210447
1	2	0.362259	0.195431
2	4	0.566315	0.223379
3	8	1.145408	0.173250
4	16	2.241314	0.190904
5	32	3.970097	0.171405
6	64	6.585767	0.217449



```
[26]: ## efficiency graph
b1_4_for_each_exponential_seq_efficiency =  $\square$ 
     $\hookrightarrow$  b1_4_for_each_exponential_seq_speedup_merged.copy()
b1_4_for_each_exponential_seq_efficiency['GCC(TBB)'] =  $\square$ 
     $\hookrightarrow$  b1_4_for_each_exponential_seq_efficiency['GCC(TBB)'] /  $\square$ 
     $\hookrightarrow$  b1_4_for_each_exponential_seq_efficiency['threads']
b1_4_for_each_exponential_seq_efficiency['NVC(OMP)'] =  $\square$ 
     $\hookrightarrow$  b1_4_for_each_exponential_seq_efficiency['NVC(OMP)'] /  $\square$ 
     $\hookrightarrow$  b1_4_for_each_exponential_seq_efficiency['threads']

print(b1_4_for_each_exponential_seq_efficiency)

# plot efficiency
ax = b1_4_for_each_exponential_seq_efficiency.
     $\hookrightarrow$  plot(x='threads', color=[GCC_TBB_COLOR, NVC_OMP_COLOR])

# adding horizontal line where there is speedup
ax.axhline(y=1, color='r', linestyle='--')

plt.ylim(0, 1.05)

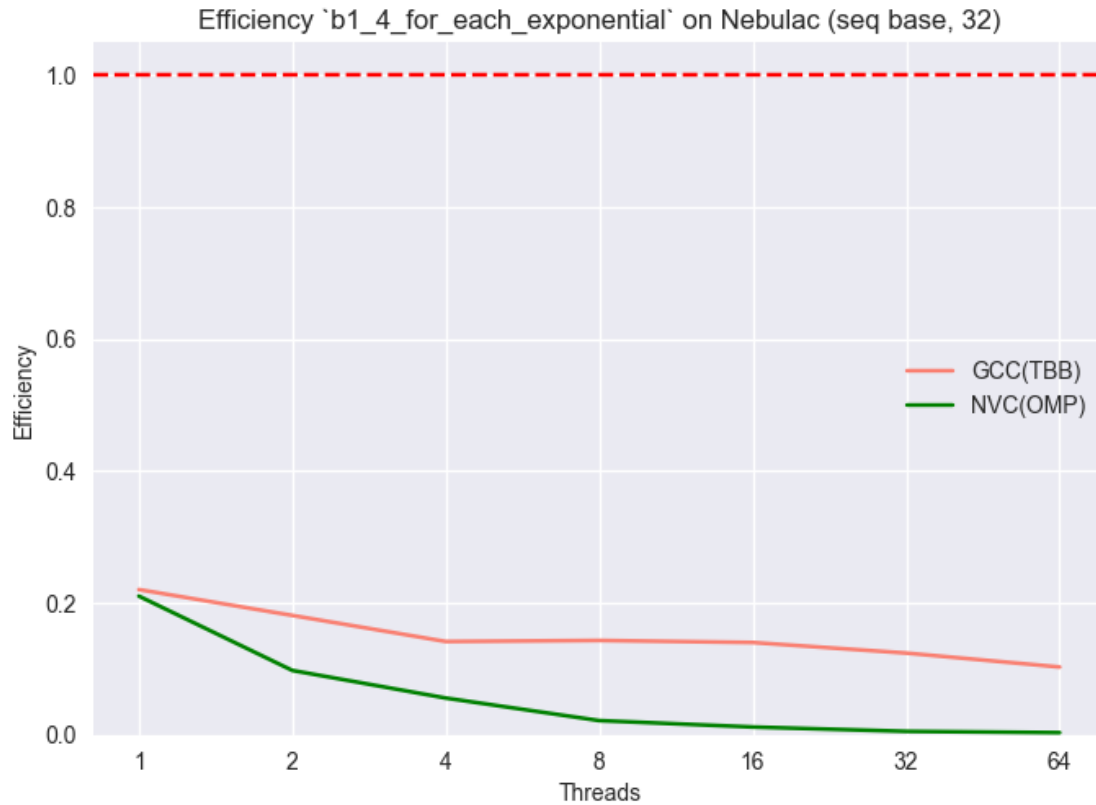
plt.xscale('log', base=2)
current_values = plt.gca().get_xticks()
plt.gca().set_xticklabels(['{:,.0f}'.format(x) for x in current_values])

plt.ylabel('Efficiency')
plt.xlabel('Threads')
plt.title('Efficiency `b1_4_for_each_exponential` on Nebulac (seq base, 32)')

plot("Efficiency `b1_4_for_each_exponential` on Nebulac (seq base, 32)")
```

	threads	GCC(TBB)	NVC(OMP)
0	1	0.220355	0.210447
1	2	0.181129	0.097715
2	4	0.141579	0.055845
3	8	0.143176	0.021656
4	16	0.140082	0.011931
5	32	0.124066	0.005356
6	64	0.102903	0.003398

```
/var/folders/42/fk0jfrdyd1dd1ztdldncqc1cw0000gn/T/ipykernel_33509/2006940372.py:1
8: UserWarning: FixedFormatter should only be used together with FixedLocator
    plt.gca().set_xticklabels(['{:,.0f}'.format(x) for x in current_values])
```



### Par(1) Base

```
[27]: # GCC
## load gcc threaded b1_4_for_each_exponential_par
b1_4_for_each_exponential_threads_gcc =
    ↪extraction_pandas_frame_algo_threaded(root_dir + '/GCC_TBB/'
    ↪THREADS', 'b1_4_for_each_exponential_par', [1,2,4,8,16,32,64], COMP="GCC(TBB)", input_size=32)

## calc strong scaling
b1_4_for_each_exponential_strong_scaling_parbase_gcc =
    ↪calc_speedup_based_par(b1_4_for_each_exponential_threads_gcc, "GCC(TBB)")

# NVC(OMP)
## load nvhpc threaded b1_4_for_each_exponential_par
b1_4_for_each_exponential_threads_nvc_omp =
    ↪extraction_pandas_frame_algo_threaded(root_dir + '/NVHPC_Multicore/'
    ↪THREADS', 'b1_4_for_each_exponential_par', [1,2,4,8,16,32,64], COMP="NVC(OMP)", input_size=32)

## calc strong scaling
```

```

b1_4_for_each_exponential_strong_scaling_parbase_nvc_omp = □
    ↪ calc_speedup_based_par(b1_4_for_each_exponential_threads_nvc_omp, "NVC(OMP)")

# merge for plotting
b1_4_for_each_linear_seq_parbase_speedup_merged = pd.
    ↪ merge(b1_4_for_each_exponential_strong_scaling_parbase_gcc, □
    ↪ b1_4_for_each_exponential_strong_scaling_parbase_nvc_omp, on='threads')
print(b1_4_for_each_linear_seq_parbase_speedup_merged)

# plot strong scaling
ax = b1_4_for_each_linear_seq_parbase_speedup_merged.
    ↪ plot(kind='bar', x='threads', align='center', color=[GCC_TBB_COLOR, NVC_OMP_COLOR])

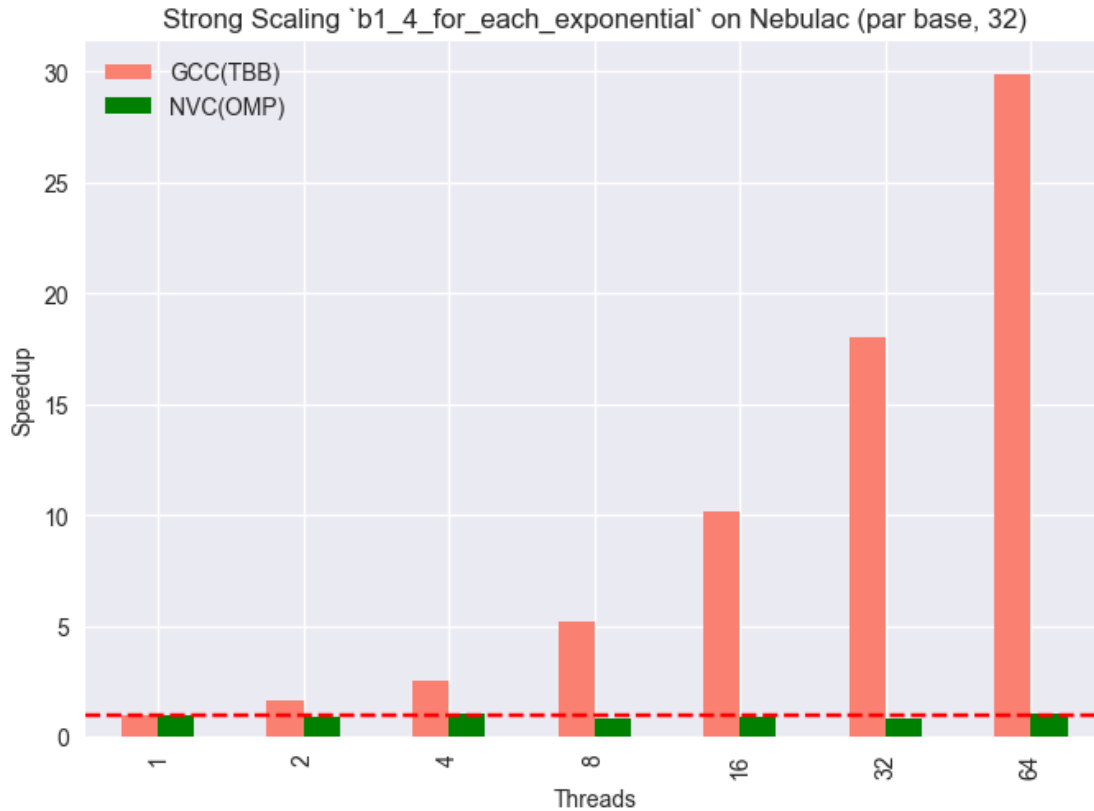
# adding horizontal line where there is speedup
ax.axhline(y=1, color='r', linestyle='--')

plt.ylabel('Speedup')
plt.xlabel('Threads')
plt.title('Strong Scaling `b1_4_for_each_exponential` on Nebulac (par base, □
    ↪ 32)')

plot('Strong Scaling `b1_4_for_each_exponential` on Nebulac (par base, 32)')

```

	threads	GCC(TBB)	NVC(OMP)
0	1	1.000000	1.000000
1	2	1.643982	0.928644
2	4	2.570017	1.061446
3	8	5.198022	0.823248
4	16	10.171399	0.907134
5	32	18.016857	0.814478
6	64	29.887137	1.033271



```
[28]: ## efficiency graph
b1_4_for_each_linear_seq_parbase_efficiency =
    ↳ b1_4_for_each_linear_seq_parbase_speedup_merged.copy()
b1_4_for_each_linear_seq_parbase_efficiency['GCC(TBB)'] =
    ↳ b1_4_for_each_linear_seq_parbase_efficiency['GCC(TBB)'] /
    ↳ b1_4_for_each_linear_seq_parbase_efficiency['threads']
b1_4_for_each_linear_seq_parbase_efficiency['NVC(OMP)'] =
    ↳ b1_4_for_each_linear_seq_parbase_efficiency['NVC(OMP)'] /
    ↳ b1_4_for_each_linear_seq_parbase_efficiency['threads']

print(b1_4_for_each_linear_seq_parbase_efficiency)

# plot efficiency
ax = b1_4_for_each_linear_seq_parbase_efficiency.
    ↳ plot(x='threads', color=[GCC_TBB_COLOR, NVC_OMP_COLOR])

# adding horizontal line where there is speedup
ax.axhline(y=1, color='r', linestyle='--')

plt.ylim(0, 1.05)
```

```

plt.xscale('log', base=2)
current_values = plt.gca().get_xticks()
plt.gca().set_xticklabels(['{:,.0f}'.format(x) for x in current_values])

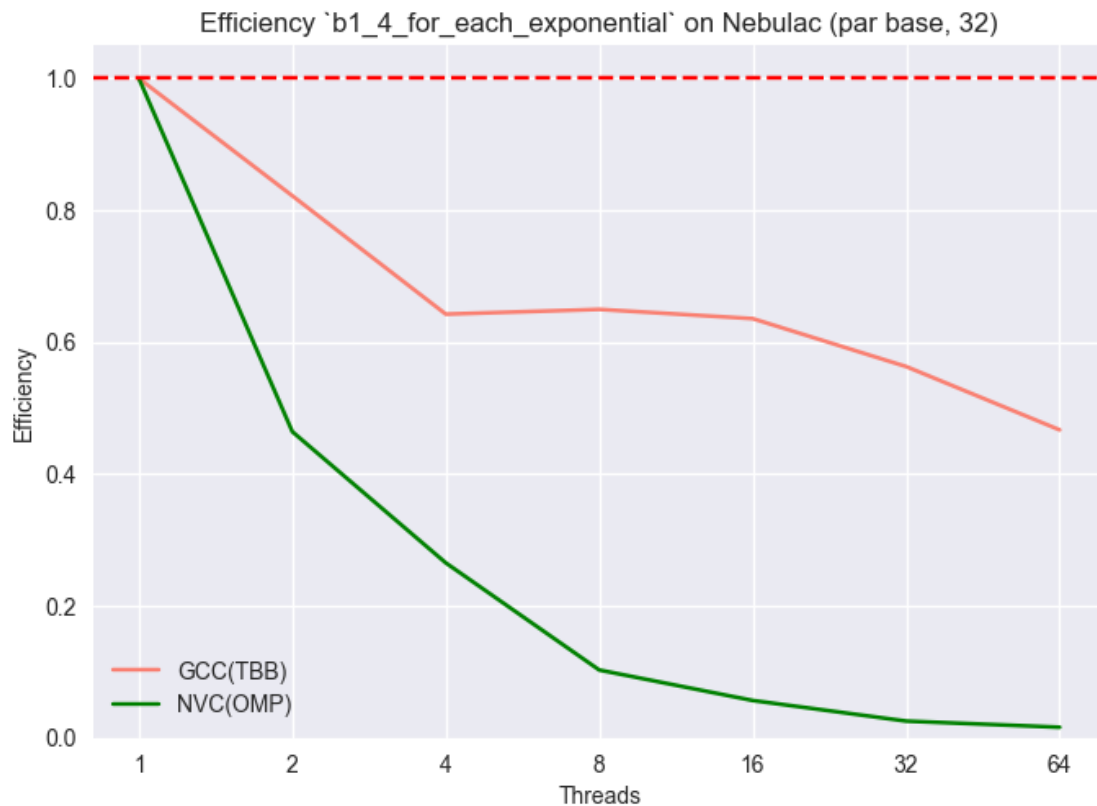
plt.ylabel('Efficiency')
plt.xlabel('Threads')
plt.title('Efficiency `b1_4_for_each_exponential` on Nebulac (par base, 32)')

plot('Efficiency `b1_4_for_each_exponential` on Nebulac (par base, 32)')

```

	threads	GCC(TBB)	NVC(OMP)
0	1	1.000000	1.000000
1	2	0.821991	0.464322
2	4	0.642504	0.265362
3	8	0.649753	0.102906
4	16	0.635712	0.056696
5	32	0.563027	0.025452
6	64	0.466987	0.016145

/var/folders/42/fk0jfryd1dd1ztdldncqc1cw0000gn/T/ipykernel\_33509/2609364247.py:1  
8: UserWarning: FixedFormatter should only be used together with FixedLocator  
plt.gca().set\_xticklabels(['{:,.0f}'.format(x) for x in current\_values])



### 2.1.3 Performance Portability Calculation (Inter Compiler)

for this group we can “calculate” a performance probability by looking at the strong scaling speedup every compiler has when using the max amount of cores. (aka running with 1M entries at max core) (inspired by [1])

example:

	achieved	perfect	efficiency
-----	-----	-----	-----
GCC(TBB)	12	16	12/16=0.75
NVC(OMP)	16	16	16/16=1
NVC(GPU)	0	0	0
Intel	14	16	14/16=0.875

Performance Portability for `{GCC(TBB), NVC(OMP), NVC(GPU), Intel}` = 0

Performance Portability for `{GCC(TBB), NVC(OMP), Intel}` =  $\frac{3}{((1/0.75) + (1/1) + (1/0.875))}$

```
[29]: max_cores :int = 64

b1_1_data = []
b1_2_data = []
b1_4_data = []

print("GCC")
# calculate efficiency for gcc on max core for `b1_1_for_each_linear`
b1_1_for_each_linear_speed_up_64_gcc = □
    ↳ b1_1_for_each_linear_strong_scaling_parbase_gcc[b1_1_for_each_linear_strong_scaling_parbase
    ↳ == 64].iloc[0]['GCC(TBB)']
b1_1_data.append(b1_1_for_each_linear_speed_up_64_gcc)

print("\tb1_1 Speedup(64):", b1_1_for_each_linear_speed_up_64_gcc)

# calculate efficiency for gcc on max core for `b1_2_for_each_quadratic`
b1_2_for_each_quadratic_par_par_speed_up_64_gcc = □
    ↳ b1_2_for_each_quadratic_par_par_scaling_parbase_gcc[b1_2_for_each_quadratic_par_par_scaling
    ↳ == 64].iloc[0]['GCC(TBB)']
b1_2_data.append(b1_2_for_each_quadratic_par_par_speed_up_64_gcc)

print("\tb1_2 Speedup(64):", b1_2_for_each_quadratic_par_par_speed_up_64_gcc)

# calculate efficiency for gcc on max core for `b1_4_for_each_exponential`
```



```

b1_4_for_each_exponential_speed_up_64_gcc =
    ↪ b1_4_for_each_exponential_strong_scaling_parbase_gcc[b1_4_for_each_exponential_strong_scaling_parbase_gcc == 64].iloc[0]['GCC(TBB)']
b1_4_data.append(b1_4_for_each_exponential_speed_up_64_gcc)

print("\tb1_4 Speedup(64):", b1_4_for_each_exponential_speed_up_64_gcc)

print("\nNVC(OMP)")
# calculate efficiency for nvhpc(mc) on max core for `b1_1_for_each_linear`
b1_1_for_each_linear_speed_up_64_nvc_omp =
    ↪ b1_1_for_each_linear_strong_scaling_parbase_nvc_omp[b1_1_for_each_linear_strong_scaling_parbase_nvc_omp == 64].iloc[0]['NVC(OMP)']
b1_1_data.append(b1_1_for_each_linear_speed_up_64_nvc_omp)

print("\tb1_1 Speedup(64):", b1_1_for_each_linear_speed_up_64_nvc_omp)

# calculate efficiency for nvhpc(mc) on max core for `b1_2_for_each_quadratic`
b1_2_for_each_quadratic_par_par_speed_up_64_nvc_omp =
    ↪ b1_2_for_each_quadratic_par_par_scaling_parbase_nvc_omp[b1_2_for_each_quadratic_par_par_scaling_parbase_nvc_omp == 64].iloc[0]['NVC(OMP)']
b1_2_data.append(b1_2_for_each_quadratic_par_par_speed_up_64_nvc_omp)

print("\tb1_2 Speedup(64):",
    ↪ b1_2_for_each_quadratic_par_par_speed_up_64_nvc_omp)

# calculate efficiency for nvhpc(mc) on max core for `b1_4_for_each_exponential`
b1_4_for_each_exponential_speed_up_64_nvc_omp =
    ↪ b1_4_for_each_exponential_strong_scaling_parbase_nvc_omp[b1_4_for_each_exponential_strong_scaling_parbase_nvc_omp == 64].iloc[0]['NVC(OMP)']
b1_4_data.append(b1_4_for_each_exponential_speed_up_64_nvc_omp)
print("\tb1_4 Speedup(64):", b1_4_for_each_exponential_speed_up_64_nvc_omp)

print("\n\n")

# calc
b1_1_perfect = max(b1_1_data)
b1_2_perfect = max(b1_2_data)
b1_4_perfect = max(b1_4_data)

# Performance portability b1_1 inter compiler
b1_1_efficiency = [x / b1_1_perfect for x in b1_1_data]
pp_b1_1 = len(b1_1_efficiency) / (sum([1 / x for x in b1_1_efficiency]))

print("Performance Portability B1_1: " , pp_b1_1)

```

```

# Performance portability b1_2 inter compiler
b1_2_efficiency = [x / b1_2_perfect for x in b1_2_data]
pp_b1_2 = len(b1_2_efficiency) / (sum([1 / x for x in b1_2_efficiency]))

print("Performance Portability B1_2: " , pp_b1_2)

# Performance portability b1_4 inter compiler
b1_4_efficiency = [x / b1_4_perfect for x in b1_4_data]
pp_b1_4 = len(b1_4_efficiency) / (sum([1 / x for x in b1_4_efficiency]))

print("Performance Portability B1_4: " , pp_b1_4)

```

GCC

```

b1_1 Speedup(64): 43.03339575922877
b1_2 Speedup(64): 49.50828342574879
b1_4 Speedup(64): 29.887136604756876

```

NVC(OMP)

```

b1_1 Speedup(64): 51.12460858208368
b1_2 Speedup(64): 32.80705823482036
b1_4 Speedup(64): 1.0332712232351373

```

```

Performance Portability B1_1: 0.914067711189745
Performance Portability B1_2: 0.7971067743386567
Performance Portability B1_4: 0.0668342558082125

```

#### 2.1.4 Findings for H1

**b1\_1** There is a significant runtime difference between parallel backends (TBB and NVC(OMP) ) when we are dealing with quite rudimentary linear homogenous workloads. As you can see in [figure of runtime comparisons](#). The larger the input size gets the worse the performance of GCC(with TBB) gets. On the other side NVC(with OMP backend) seems to scale quite good under linear homogenous workloads.

For strong scaling we can see that calculating the speedup using the parallel implementation with 1 thread and using the sequential implementation, does not make a huge difference. In fact the overhead for this kind of workload seems to be minimal. The backends scale fairly good and the absolute speedup for each number of threads does not have a tremendous difference between the two backends. We only start to notice that the more threads we utilize the larger the speedup between GCC(TBB) and NVC(OMP) gets.

For small number of threads (1-16) we see that the speedup is quite optimal (close to perfect speedup). Only later when utilizing more threads (32+) we start to see a significant performance loss for both GCC(TBB) and NVC(OMP)

Since the performance portability metric used in this hypothesis focuses on the speedup and as

observed above and the difference between speedup is not that huge, we achieve a rather high performance portability of **91%**!

Key observations: \* Significant runtime differences between GCC(TBB) and NVC(OMP) \* Speedup seems to be on same level for backends only for huge number of threads it starts to degrade \* Small number of threads nearly perfect speedup for both \* Performance portability quite high since backends behave quite good.

**b1\_2** There is a significant runtime difference between parallel backends (TBB and NVC(OMP)) when we are dealing with nested quadratic parallelism (aka nested loops with each  $O(n)$ ). As you can see in [figure of runtime comparisons](#). The larger the input size gets the worse the performance of GCC(with TBB) gets. On the other side NVC(with OMP backend) seems to scale quite good under quadratic homogenous workloads.

For this benchmark we only considered the outer parallel and inner parallel with 1thread as base to calculate the speedups. GCC(TBB) seems to have better strong scaling than NVC(OMP). It looks like that NVC(OMP) starts to degrade heavily when having high number of threads and this is also visible when looking at the efficiency.

Since the performance portability metric used in this hypothesis focuses on the speedup and as observed above and the difference between speedup is actually quite huge, we achieve a rather poor portability of **79%**!

Key observations: \* Significant runtime differences between GCC(TBB) and NVC(OMP). NVC(OMP) faster than GCC(TBB) \* Speedup difference becomes bigger with rise of threads. \* NVC(OMP) pretty much collapses at 64 threads. \* Small number of threads quite good for both backends \* Performance portability poor since NVC(OMP) collapses for high number of threads.

**b1\_4** The runtime difference between GCC(TBB) and NVC(OMP) is extreme! Since we are dealing with exponential runtime it was expected that the runtime will increase fast, but the runtimes of NVC(OMP) exploded. At first the runtime of GCC(TBB) is worse than the of NVC(OMP), but when for larger input sizes the trend turns and the runtime of NVC(OMP) exploded and making GCC(TBB) faster by a magnitude.

For this kind of nested parallelism strong scaling does looks really bad. Using the sequential algorithm or the parallel algorithm with 1 Thread as base does not have an effect on the speedup for NVC(OMP). NVC(OMP) has really bad strong scaling and often does not even break the 1x speedup. On the other hand GCC(TBB) does improve significantly with more core reaching a speedup of more than 30x.

Since NVC(OMP) scales really bad on this kind of workload but GCC(TBB) really good, we achieve a rather poor portability of **6%**!

**GPU Findings** Sadly NVC(GPU) does not support nested parallelism. Although it would be possible to run **b1\_1** with NVC(GPU) the rest of the benchmarks (**b1\_2** and **b1\_4**) do not.

**Hypothesis Findings** The hypothesis is TRUE!

## 2.2 H2

The performance is significantly impacted by the order in which parallelism is applied, whether it is outer loop sequential and inner loop parallel, or outer loop parallel and inner loop sequential.

### 2.2.1 Time

**Time Comparison - b1\_2\_for\_each\_quadratic\_outer\_std::execution::sequenced\_policy\_par vs b1\_2\_for\_each\_quadratic\_outer\_std::execution::parallel\_policy\_seq** Check how the runtime without constraining the number of threads develops with increasing the input size.

```
[30]: def get_b1_2_data_algo(compiler_location:str,compiler_name:str) -> pd.DataFrame:
    ## load b1_2_for_each_quadratic_outer_std::execution::sequenced_policy_par
    b1_2_for_each_quadratic_seq_par = extraction_pandas_frame_algo(root_dir +
    ↪f'/{compiler_location}/DEFAULT/b1_2_for_each_quadratic_outer_std::execution::
    ↪sequenced_policy_par__Default.csv',COMP=compiler_name)

    b1_2_for_each_quadratic_seq_par = b1_2_for_each_quadratic_seq_par.
    ↪drop(columns=['name','cpu_time','time_unit','median','stddev','Compiler'])
    b1_2_for_each_quadratic_seq_par = b1_2_for_each_quadratic_seq_par.
    ↪rename(columns={'real_time':f'{compiler_name} - seq_par'})

    ## load b1_2_for_each_quadratic_outer_std::execution::parallel_policy_seq
    b1_2_for_each_quadratic_par_seq = extraction_pandas_frame_algo(root_dir +
    ↪f'/{compiler_location}/DEFAULT/b1_2_for_each_quadratic_outer_std::execution::
    ↪parallel_policy_seq__Default.csv',COMP=compiler_name)

    b1_2_for_each_quadratic_par_seq = b1_2_for_each_quadratic_par_seq.
    ↪drop(columns=['name','cpu_time','time_unit','median','stddev','Compiler'])
    b1_2_for_each_quadratic_par_seq = b1_2_for_each_quadratic_par_seq.
    ↪rename(columns={'real_time':f'{compiler_name} - par_seq'})

    ## merge
    return pd.
    ↪merge(b1_2_for_each_quadratic_seq_par,b1_2_for_each_quadratic_par_seq,
    ↪on='n')

instances = [
    ('GCC_TBB','GCC(TBB)'),
    ('NVHPC_Multicore','NVC(OMP)')
]

# collect data for instances
data = [get_b1_2_data_algo(*x) for x in instances]
```

```

# merge for plotting
b1_2_for_each_quadratic_time_merged = pd.merge(*data, on='n')

# convert time from ns to microseconds because otherwise it will look really bad
for _, compiler_name in instances:
    b1_2_for_each_quadratic_time_merged[f'{compiler_name} - par_seq'] =
↳ b1_2_for_each_quadratic_time_merged[f'{compiler_name} - par_seq'] / 1_000
    b1_2_for_each_quadratic_time_merged[f'{compiler_name} - seq_par'] =
↳ b1_2_for_each_quadratic_time_merged[f'{compiler_name} - seq_par'] / 1_000

# plot
ax = b1_2_for_each_quadratic_time_merged.
↳ plot(kind='bar', x='n', align='center', color=[GCC_TBB_COLOR, GCC_TBB_COLOR_SECONDARY, NVC_OMP_C
print(b1_2_for_each_quadratic_time_merged)

# plt.yscale('log', base=10)
plt.ylabel('Time in (µs)')
plt.xlabel('Input size')
plt.title('Runtime `b1_2_for_each_quadratic` on Nebulac')

plot('Runtime `b1_2_for_each_quadratic` on Nebulac')

```

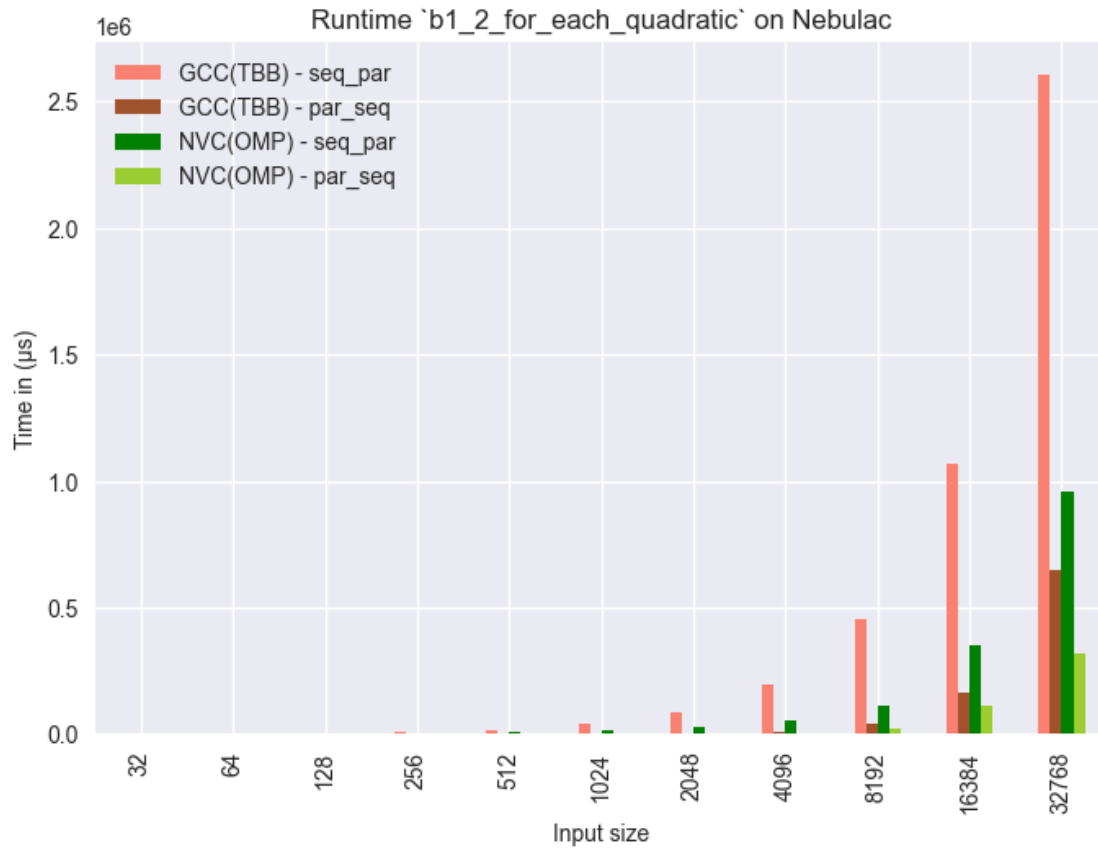
	GCC(TBB) - seq_par	n	GCC(TBB) - par_seq	NVC(OMP) - seq_par \
0	630.911	32	24.1083	350.829
1	1529.730	64	29.6303	712.506
2	3259.200	128	42.8168	1417.470
3	7237.800	256	80.5595	2827.600
4	16449.900	512	217.1550	5861.490
5	37973.700	1024	720.0160	11950.000
6	86425.400	2048	2691.5700	25747.200
7	197370.000	4096	10485.3000	52638.900
8	451837.000	8192	41183.8000	114388.000
9	1069750.000	16384	163609.0000	348804.000
10	2604950.000	32768	651184.0000	955396.000

	NVC(OMP) - par_seq
0	12.9499
1	13.3429
2	17.0725
3	32.7040
4	93.0087
5	325.7320
6	1467.6900
7	5017.4100
8	21595.6000

```

9         110340.0000
10        319242.0000

```



## 2.2.2 Strong Scaling

$$S(p) = T(1) / T(p)$$

As based we use once the: \* parallel algorithm (1 thread)

Strong Scaling - b1\_2\_for\_each\_quadratic\_outer\_std::execution::sequenced\_policy\_par  
vs b1\_2\_for\_each\_quadratic\_outer\_std::execution::parallel\_policy\_seq 32.768 fixed  
input size with threads 1-64

```

[31]: def get_b1_2_strong_scaling_algo(compiler_location:str,compiler_name:str) -> pd.
      ↳ DataFrame:
      ↳   ## Threading data
      ↳   b1_2_for_each_quadratic_seq_par_threads =
      ↳ extraction_pandas_frame_algo_threaded(root_dir + f'/{compiler_location}/
      ↳ 'THREADS',
      ↳
      ↳   'b1_2_for_each_quadratic_outer_std::execution::sequenced_policy_par',

```

```

↳ [1,2,4,8,16,32,64],
↳ COMP=compiler_name,
↳ input_size=32768
↳)

    ## calc strong scaling
    b1_2_for_each_quadratic_seq_par_strong_scaling =
↳calc_speedup_based_par(b1_2_for_each_quadratic_seq_par_threads,
↳f"{compiler_name} - seq_par",
↳input_size=32768
)

    ## load b1_2_for_each_quadratic_outer_std::execution::parallel_policy_seq
    b1_2_for_each_quadratic_par_seq_threads =
↳extraction_pandas_frame_algo_threaded(root_dir + f'/{compiler_location}/
↳THREADS',
↳ 'b1_2_for_each_quadratic_outer_std::execution::parallel_policy_seq',
↳ [1,2,4,8,16,32,64],
↳ COMP=compiler_name,
↳ input_size=32768
↳)

    ## calc strong scaling
    b1_2_for_each_quadratic_par_seq_strong_scaling =
↳calc_speedup_based_par(b1_2_for_each_quadratic_par_seq_threads,
↳f"{compiler_name} - par_seq",
↳input_size=32768
)

    ## merge
    return pd.merge(b1_2_for_each_quadratic_seq_par_strong_scaling,

```

```

        b1_2_for_each_quadratic_par_seq_strong_scaling,
        on='threads'
    )

instances = [
    ('GCC_TBB', 'GCC(TBB)'),
    ('NVHPC_Multicore', 'NVC(OMP)')
]

# collect data for instances
data = [get_b1_2_strong_scaling_algo(*x) for x in instances]

b1_2_for_each_quadratic_strong_scaling_merged = pd.merge(*data, on='threads')

print(b1_2_for_each_quadratic_strong_scaling_merged)

# plot strong scaling
ax = b1_2_for_each_quadratic_strong_scaling_merged.plot(kind='bar',
                                                         x='threads',
                                                         align='center',
                                                         color=[GCC_TBB_COLOR, GCC_TBB_COLOR_SECONDARY, NVC_OMP_COLOR, NVC_OMP_COLOR_SECONDARY]
                                                         )

# adding horizontal line where there is speedup
ax.axhline(y=1, color='r', linestyle='--')

plt.ylabel('Speedup')
plt.xlabel('Threads')
plt.title('Strong Scaling `b1_2_for_each_quadratic` on Nebulac (32768)')

plot('Strong Scaling `b1_2_for_each_quadratic` on Nebulac (32768)')

```

	threads	GCC(TBB) - seq_par	GCC(TBB) - par_seq	NVC(OMP) - seq_par \
0	1	1.000000	1.000000	1.000000
1	2	1.979862	1.998062	1.980330
2	4	3.750753	3.986084	3.855304
3	8	6.238904	7.944636	7.276533
4	16	9.478906	15.592834	11.836966
5	32	12.198081	27.519958	16.516401
6	64	14.887093	50.555355	15.592923

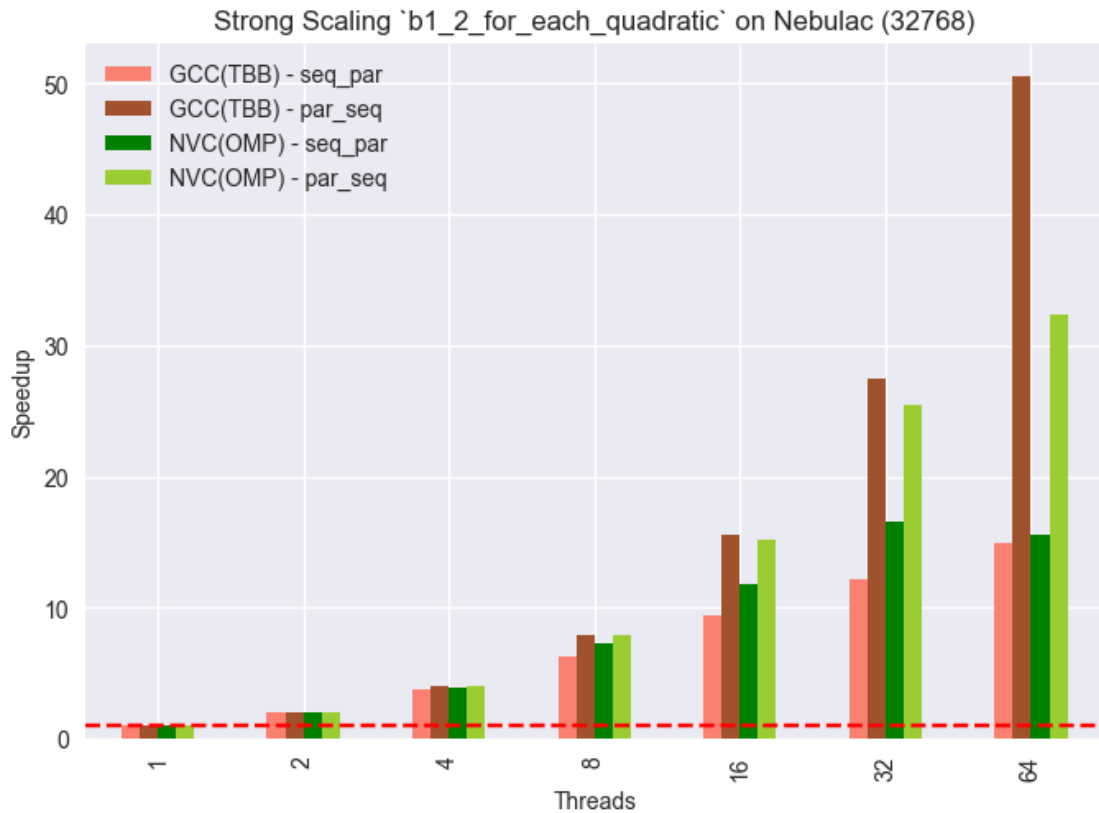
	NVC(OMP) - par_seq
0	1.000000
1	2.000584
2	3.987332



```

3         7.863144
4         15.249982
5         25.411051
6         32.338995

```



```
[32]: ## efficiency graph
```

```

b1_2_for_each_quadratic_efficiency =
    ↳ b1_2_for_each_quadratic_strong_scaling_merged.copy()

instances = [
    ('GCC_TBB', 'GCC(TBB)'),
    ('NVHPC_Multicore', 'NVC(OMP)')
]

for _, compiler_name in instances:
    b1_2_for_each_quadratic_efficiency[f'{compiler_name} - par_seq'] =
        ↳ b1_2_for_each_quadratic_efficiency[f'{compiler_name} - par_seq'] /
        ↳ b1_2_for_each_quadratic_efficiency['threads']

```

```

    b1_2_for_each_quadratic_efficiency[f'{compiler_name} - seq_par'] = \
↳ b1_2_for_each_quadratic_efficiency[f'{compiler_name} - seq_par'] / \
↳ b1_2_for_each_quadratic_efficiency['threads']

print(b1_2_for_each_quadratic_efficiency)

# plot efficiency
ax = b1_2_for_each_quadratic_efficiency.plot(x='threads',
↳
↳ color=[GCC_TBB_COLOR,GCC_TBB_COLOR_SECONDARY,NVC_OMP_COLOR,NVC_OMP_COLOR_SECONDARY]
)

# adding horizontal line where there is speedup
ax.axhline(y=1, color='r', linestyle='--')

plt.ylim(0,1.05)

plt.xscale('log', base=2)
current_values = plt.gca().get_xticks()
plt.gca().set_xticklabels(['{:, .0f}'.format(x) for x in current_values])

plt.ylabel('Efficiency')
plt.xlabel('Threads')
plt.title('Efficiency `b1_2_for_each_quadratic` on Nebulac (32768)')

plot('Efficiency `b1_2_for_each_quadratic` on Nebulac (32768)')

```

	threads	GCC(TBB) - seq_par	GCC(TBB) - par_seq	NVC(OMP) - seq_par \
0	1	1.000000	1.000000	1.000000
1	2	0.989931	0.999031	0.990165
2	4	0.937688	0.996521	0.963826
3	8	0.779863	0.993079	0.909567
4	16	0.592432	0.974552	0.739810
5	32	0.381190	0.859999	0.516138
6	64	0.232611	0.789927	0.243639

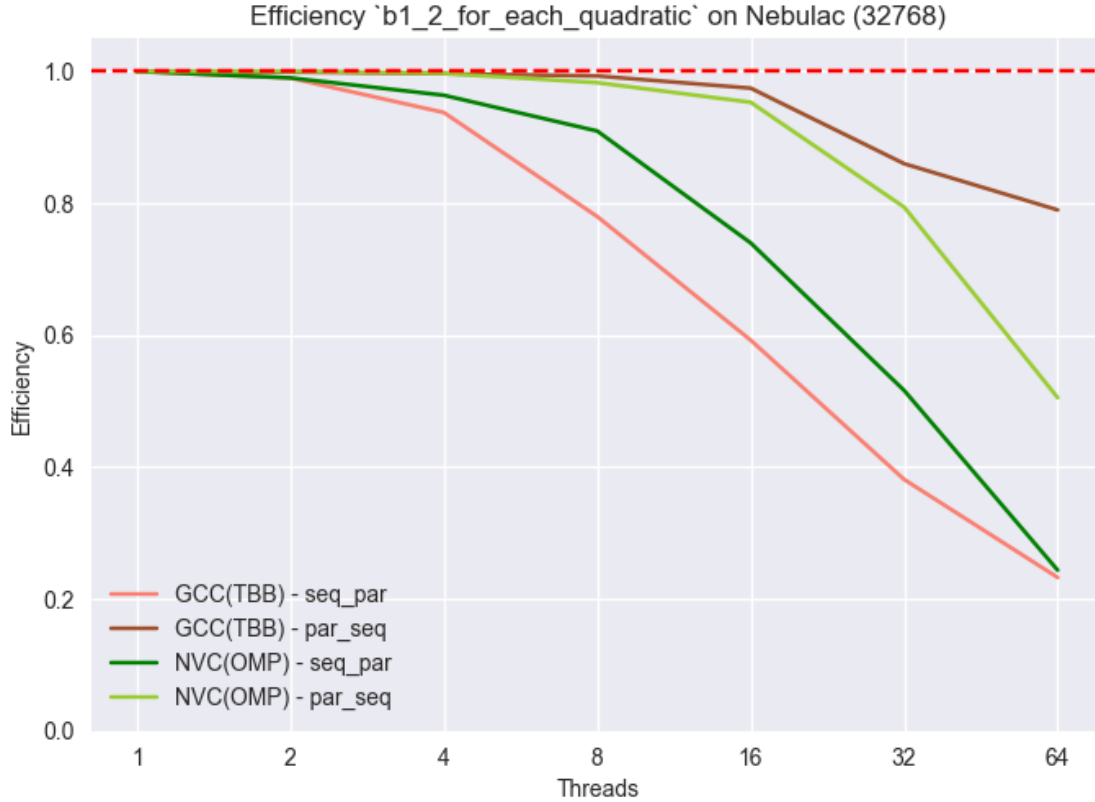
	NVC(OMP) - par_seq
0	1.000000
1	1.000292
2	0.996833
3	0.982893
4	0.953124
5	0.794095
6	0.505297

```

/var/folders/42/fk0jfryd1dd1ztdldncqc1cw0000gn/T/ipykernel_33509/956545906.py:30
: UserWarning: FixedFormatter should only be used together with FixedLocator

```

```
plt.gca().set_xticklabels(['{:,.0f}'.format(x) for x in current_values])
```



### 2.2.3 Performance Portability Calculation (Inter Compiler)

Since we know that (par, seq) will be better than (seq, par) we can check the stddev of the performance improvement from (seq, par) to (par, seq) for every compiler. For example:

	(seq, par)	(par, seq)	faster
GCC(TBB)	10s	5s	2x
NVC(OMP)	12s	8s	1.5x
NVC(GPU)	0	0	0
Intel	9	1	9x

$\text{stddev}(2, 1.5, 9) = 3.4$  indicating that the difference is quite significant when changing compilers.

$\text{stddev}(2, 1.5) = 0.25$  indicating that the difference is not significant when changing compilers.

```
[33]: # calc pp_metrics

instances = [
    ('GCC_TBB', 'GCC(TBB)'),
    ('NVHPC_Multicore', 'NVC(OMP)')
```

```

]

faster_data = []

for compiler_location, compiler_name in instances:
    print(compiler_name)

    times_faster = (b1_2_for_each_quadratic_time_merged[f'{compiler_name} -_
↪seq_par'] / b1_2_for_each_quadratic_time_merged[f'{compiler_name} -_
↪par_seq']).iloc[-1]
    faster_data.append(times_faster)

    print("\t Par_Seq faster than Seq_Par: ", times_faster)
    print()

print("\n")
pp_h2 = statistics.stdev(faster_data)

print("Performance Portability H2:",pp_h2)

```

GCC(TBB)

Par\_Seq faster than Seq\_Par: 4.000328632153124

NVC(OMP)

Par\_Seq faster than Seq\_Par: 2.9927014615871346

Performance Portability H2: 0.7125000052150252

## 2.2.4 Findings for H2

**b1\_2\_for\_each\_quadratic\_outer\_std::execution::sequenced\_policy\_par (seq\_par)** The performance of seq\_par exhibits significant variations when switching between compilers, particularly for larger input sizes of 8192+. The runtime differences become increasingly worse, and GCC(TBB) demonstrates poor performance in such scenarios.

As for strong scaling, seq\_par's performance is rather poor. Both GCC(TBB) and NVC(OMP) experience a poor speedup after 16 threads. While GCC(TBB) exhibits minor improvements with additional threads, they are insignificant. On the other hand NVC(OMP) loses speedup once 64 threads are used. The efficiency graph shows that NVC(OMP) takes a hit at 8 threads, with both collapsing at 16 threads. Notably, NVC(OMP) starts from a higher efficiency level than GCC(TBB).

Key Observations: \* Runtime of seq\_par changes a lot from compiler to compiler \* GCC(TBB) performs really bad (runtime) \* Strong scaling of GCC(TBB) is better than NVC(OMP) \* NVC(OMP) strong scaling even worse when going 32->64 threads

**b1\_2\_for\_each\_quadratic\_outer\_std::execution::parallel\_policy\_seq (par\_seq)** We see the same behaviour here as seen when using seq\_par. Notably, as the input sizes increase, GCC(TBB) shows a more rapid decline in performance compared to NVC(OMP). For instance, when transitioning from 16k to 32k, NVC(OMP) experiences a speedup of **2.89x**, whereas GCC(TBB) only shows a speedup of **3.98x**.

Moreover, the two backends exhibit different behavior when strong scaling. As the number of threads increases, GCC(TBB) benefits from the added resources, while NVC(OMP) takes a considerable hit and reaches peak performance at 32 threads.

Key Observations: \* Runtime difference visible (when changing compiler) \* GCC(TBB) runtime explodes for large input size \* NVC(OMP) runtime rises as expected for large input sizes \* GCC(TBB) strong scaling way better than NVC(OMP) \* NVC(OMP) strong scaling starts to slow down at 32 threads

### 2.2.5 General

We can observe a significant difference in runtime when we switch between execution policies for inner and outer loops. This behavior is consistent across all parallel backends. In terms of runtime, GCC(TBB) shows the most substantial improvement when switching from seq\_par to par\_seq. However, NVC(OMP) performs better in absolute runtime numbers.

Switching from seq\_par to par\_seq also affects strong scaling. For smaller input sizes, the backends show similar strong scaling behavior. However, for larger thread counts (64), GCC(TBB) still shows improvements over NVC(OMP).

When switching between backends, all of them show improvement, though to varying degrees. The extent of improvement differs between the backends, as reflected in the performance portability metric calculated for this benchmark. The stddev value of 0.71 indicates that changing compilers can lead to better performance improvements.

Key Observation: \* Changing order of execution policy has great impact. \* Strong Scaling varies a lot by compiler \* Absolute Runtime difference by compiler is a lot \* Improvement from seq\_par to par\_seq varies significantly by compiler

**GPU Findings** Sadly because this benchmarks use nested parallelism it won't work on the NVC(GPU).

**Hypothesis Findings** This hypothesis is **TRUE!**

## 2.3 H3

Some parallel backends exhibit better performance and scalability when handling nested parallelism for heterogeneous workloads

### 2.3.1 Time

**Time Comparison - b1\_1\_for\_each\_linear\_mandelbrot** Check how the runtime without constraining the number of threads develops with increasing the input size

```
[34]: def get_b1_1_mandelbrot_data_algo(compiler_location:str,compiler_name:str) ->
↳pd.DataFrame:
    ## load b1_1_for_each_linear_mandelbrot_par
    df = extraction_pandas_frame_algo(root_dir + f'/{compiler_location}/DEFAULT/
↳b1_1_for_each_linear_mandelbrot_par__Default.csv',COMP=compiler_name)

    df = df.
↳drop(columns=['name','cpu_time','time_unit','median','stddev','Compiler'])
    df = df.rename(columns={'real_time':f'{compiler_name}'})

    return df

instances = [
    ('GCC_TBB','GCC(TBB)'),
    ('NVHPC_Multicore','NVC(OMP)')
]

# collect data for instances
data = [get_b1_1_mandelbrot_data_algo(*x) for x in instances]

# merge for plotting
b1_1_mandelbrot_time_merged = pd.merge(*data, on='n')

# convert time from ns to microseconds because otherwise it will look really bad
for _, compiler_name in instances:
    b1_1_mandelbrot_time_merged[f'{compiler_name}'] =
↳b1_1_mandelbrot_time_merged[f'{compiler_name}'] / 1_000

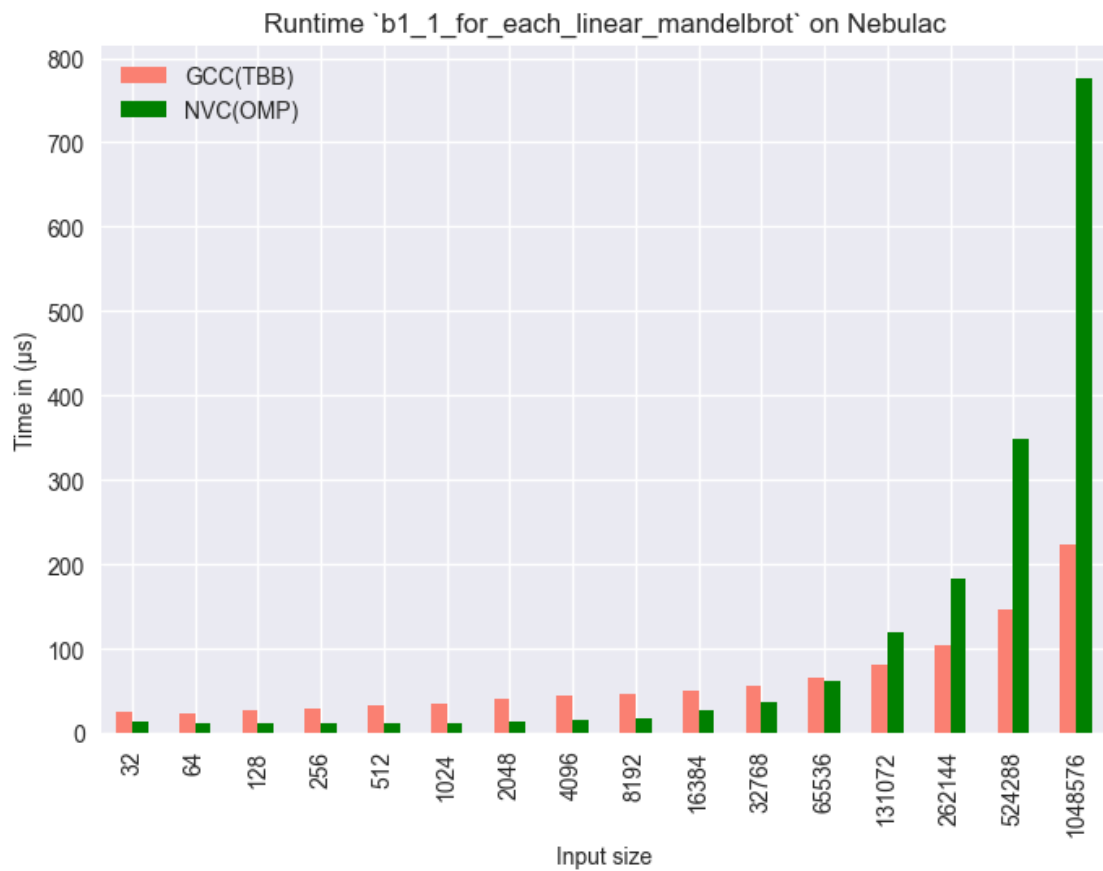
# plot
ax = b1_1_mandelbrot_time_merged.
↳plot(kind='bar',x='n',align='center',color=[GCC_TBB_COLOR,NVC_OMP_COLOR])
print(b1_1_mandelbrot_time_merged)

#plt.yscale('log', base=2)
plt.ylabel('Time in (µs)')
plt.xlabel('Input size')
plt.title('Runtime `b1_1_for_each_linear_mandelbrot` on Nebulac')

plot('Runtime `b1_1_for_each_linear_mandelbrot` on Nebulac')
```

	GCC(TBB)	n	NVC(OMP)
0	25.7110	32	13.1448
1	23.8880	64	12.0237
2	26.2309	128	11.0411

3	28.5104	256	11.0437
4	32.0780	512	11.2044
5	35.4990	1024	11.4312
6	40.0559	2048	13.2587
7	44.4702	4096	14.7847
8	46.0035	8192	18.2997
9	50.1906	16384	27.9500
10	56.3977	32768	37.2332
11	65.8956	65536	61.2572
12	80.2900	131072	119.7970
13	104.1640	262144	183.0140
14	146.4930	524288	347.8630
15	223.7350	1048576	776.4000



**Time Comparison - b1\_2\_for\_each\_quadratic\_mandelbrot** Check how the runtime without constraining the number of threads develops with increasing the input size

[35] :

```

def get_b1_2_mandelbrot_data_algo(compiler_location:str,compiler_name:str) ->
↳pd.DataFrame:
    ## load b1_2_for_each_quadratic_mandelbrot_par_par
    df_par_par = extraction_pandas_frame_algo(root_dir + f'/{compiler_location}/
↳DEFAULT/b1_2_for_each_quadratic_mandelbrot_outer_std::execution::
↳parallel_policy_par__Default.csv',COMP=compiler_name)

    df_par_par = df_par_par.
↳drop(columns=['name','cpu_time','time_unit','median','stddev','Compiler'])
    df_par_par = df_par_par.rename(columns={'real_time':f'{compiler_name}'})

    ## load b1_2_for_each_quadratic_mandelbrot_par_seq
    """df_par_seq = extraction_pandas_frame_algo(root_dir + f'/'
↳{compiler_location}/DEFAULT/b1_2_for_each_quadratic_mandelbrot_outer_std::
↳execution::parallel_policy_seq__Default.csv',COMP=compiler_name)

    df_par_seq = df_par_seq.
↳drop(columns=['name','cpu_time','time_unit','median','stddev','Compiler'])
    df_par_seq = df_par_seq.rename(columns={'real_time':f'{compiler_name}' -
↳par_seq'})"""

    return df_par_par

instances = [
    ('GCC_TBB','GCC(TBB)'),
    ('NVHPC_Multicore','NVC(OMP)')
]

# collect data for instances
data = [get_b1_2_mandelbrot_data_algo(*x) for x in instances]

# merge for plotting
b1_2_mandelbrot_time_merged = pd.merge(*data, on='n')

# convert time from ns to microseconds because otherwise it will look really bad
for _, compiler_name in instances:
    b1_2_mandelbrot_time_merged[f'{compiler_name}'] =
↳b1_2_mandelbrot_time_merged[f'{compiler_name}'] / 1_000
    #b1_2_mandelbrot_time_merged[f'{compiler_name} - par_seq'] =
↳b1_2_mandelbrot_time_merged[f'{compiler_name} - par_seq'] / 1_000

# plot

```



```

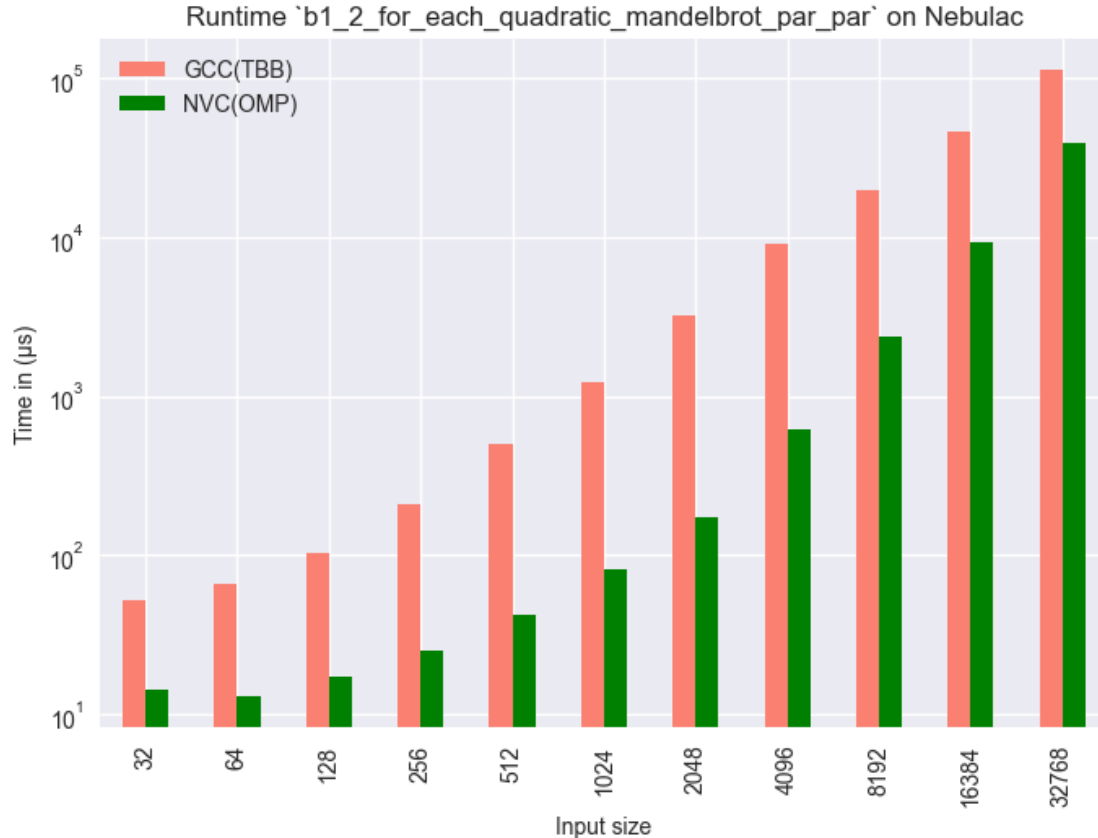
ax = b1_2_mandelbrot_time_merged.
    plot(kind='bar',x='n',align='center',color=[GCC_TBB_COLOR,NVC_OMP_COLOR])
print(b1_2_mandelbrot_time_merged)

plt.yscale('log', base=10)
plt.ylabel('Time in (μs)')
plt.xlabel('Input size')
plt.title('Runtime `b1_2_for_each_quadratic_mandelbrot_par_par` on Nebulac')

plot('Runtime `b1_2_for_each_quadratic_mandelbrot_par_par` on Nebulac')

```

	GCC(TBB)	n	NVC(OMP)
0	52.6922	32	14.3070
1	66.4682	64	12.9766
2	102.1700	128	17.3404
3	208.8140	256	25.2010
4	495.2330	512	42.5719
5	1210.1300	1024	82.0095
6	3186.5900	2048	174.3940
7	9017.7100	4096	621.8690
8	19850.9000	8192	2358.0000
9	45629.3000	16384	9165.8000
10	111974.0000	32768	39351.2000



### 2.3.2 Strong Scaling

$$S(p) = T(1) / T(p)$$

As based we use: parallel algorithm (1 thread)

**Strong scaling - b1\_1\_for\_each\_linear\_mandelbrot** 1M fixed input size with threads 1-64

```
[36]: def get_b1_1_mandelbrot_strong_scaling_algo(compiler_location:str,compiler_name:
    ↪str) -> pd.DataFrame:
        ## b1_1_for_each_linear_mandelbrot_threaded
        df = extraction_pandas_frame_algo_threaded(root_dir + f'/
    ↪{compiler_location}/THREADS',

        ↪'b1_1_for_each_linear_mandelbrot_par',
        ↪[1,2,4,8,16,32,64],
        ↪COMP=compiler_name
        )

        ## calc strong scaling
        return calc_speedup_based_par(df,f"{compiler_name}")

instances = [
    ('GCC_TBB', 'GCC(TBB)'),
    ('NVHPC_Multicore', 'NVC(OMP)')
]

# collect data for instances
data = [get_b1_1_mandelbrot_strong_scaling_algo(*x) for x in instances]

b1_1_for_each_linear_mandelbrot_strong_scaling_merged = pd.merge(*data,
    ↪on='threads')

print(b1_1_for_each_linear_mandelbrot_strong_scaling_merged)

# plot strong scaling
ax = b1_1_for_each_linear_mandelbrot_strong_scaling_merged.plot(kind='bar',
    ↪x='threads',
    ↪align='center',
    ↪color=[GCC_TBB_COLOR,NVC_OMP_COLOR]
    )

# adding horizontal line where there is speedup
```

```

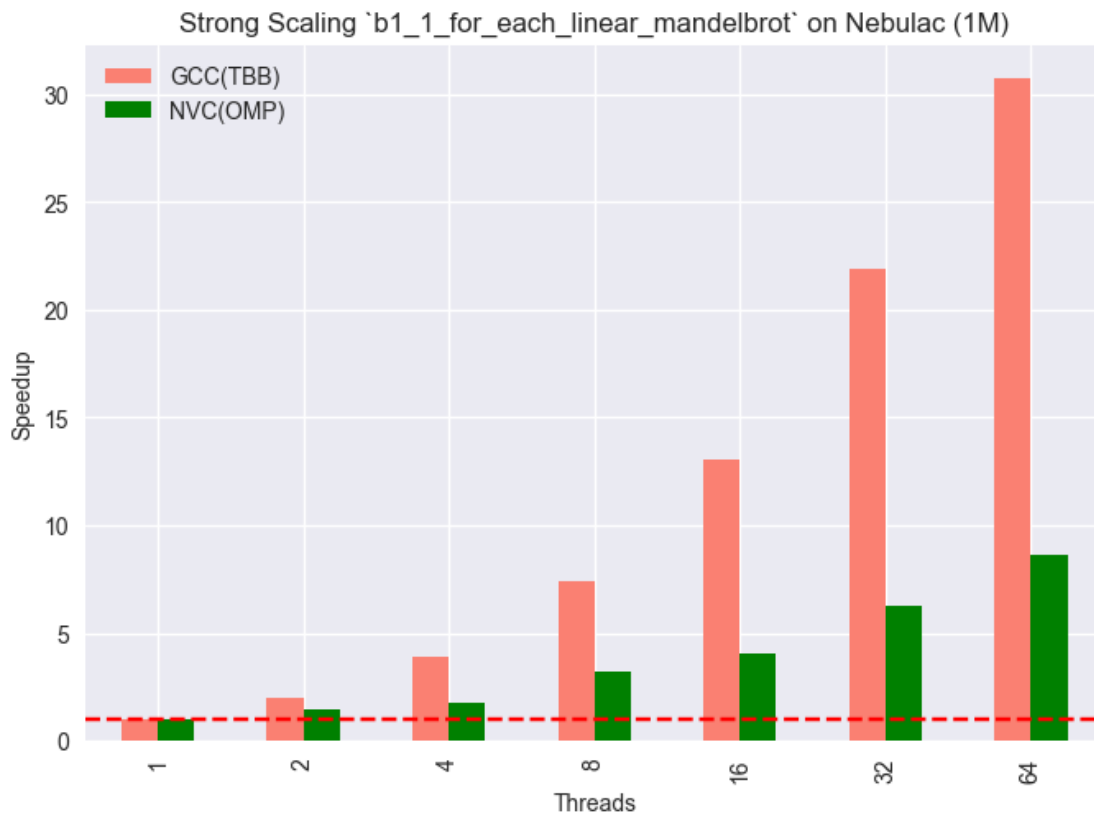
ax.axhline(y=1, color='r', linestyle='--')

plt.ylabel('Speedup')
plt.xlabel('Threads')
plt.title('Strong Scaling `b1_1_for_each_linear_mandelbrot` on Nebulac (1M)')

plot('Strong Scaling `b1_1_for_each_linear_mandelbrot` on Nebulac (1M)')

```

	threads	GCC(TBB)	NVC(OMP)
0	1	1.000000	1.000000
1	2	1.996081	1.451250
2	4	3.926398	1.795026
3	8	7.410093	3.234524
4	16	13.015673	4.066618
5	32	21.870544	6.280458
6	64	30.747455	8.625019



```
[37]: # efficiency graph
```

```

b1_1_for_each_linear_mandelbrot_efficiency = 
    ↪ b1_1_for_each_linear_mandelbrot_strong_scaling_merged.copy()

instances = [
    ('GCC_TBB', 'GCC(TBB)'),
    ('NVHPC_Multicore', 'NVC(OMP)')
]

for _, compiler_name in instances:
    b1_1_for_each_linear_mandelbrot_efficiency[f'{compiler_name}'] = 
    ↪ b1_1_for_each_linear_mandelbrot_efficiency[f'{compiler_name}'] / 
    ↪ b1_1_for_each_linear_mandelbrot_efficiency['threads']

print(b1_1_for_each_linear_mandelbrot_efficiency)

# plot efficiency
ax = b1_1_for_each_linear_mandelbrot_efficiency.plot(x='threads',
                                                    color=[GCC_TBB_COLOR, NVC_OMP_COLOR]
                                                    )

# adding horizontal line where there is speedup
ax.axhline(y=1, color='r', linestyle='--')

plt.ylim(0, 1.05)

plt.xscale('log', base=2)
current_values = plt.gca().get_xticks()
plt.gca().set_xticklabels(['{:,.0f}'.format(x) for x in current_values])

plt.ylabel('Efficiency')
plt.xlabel('Threads')
plt.title('Efficiency `b1_1_for_each_linear_mandelbrot` on Nebulac (1M)')

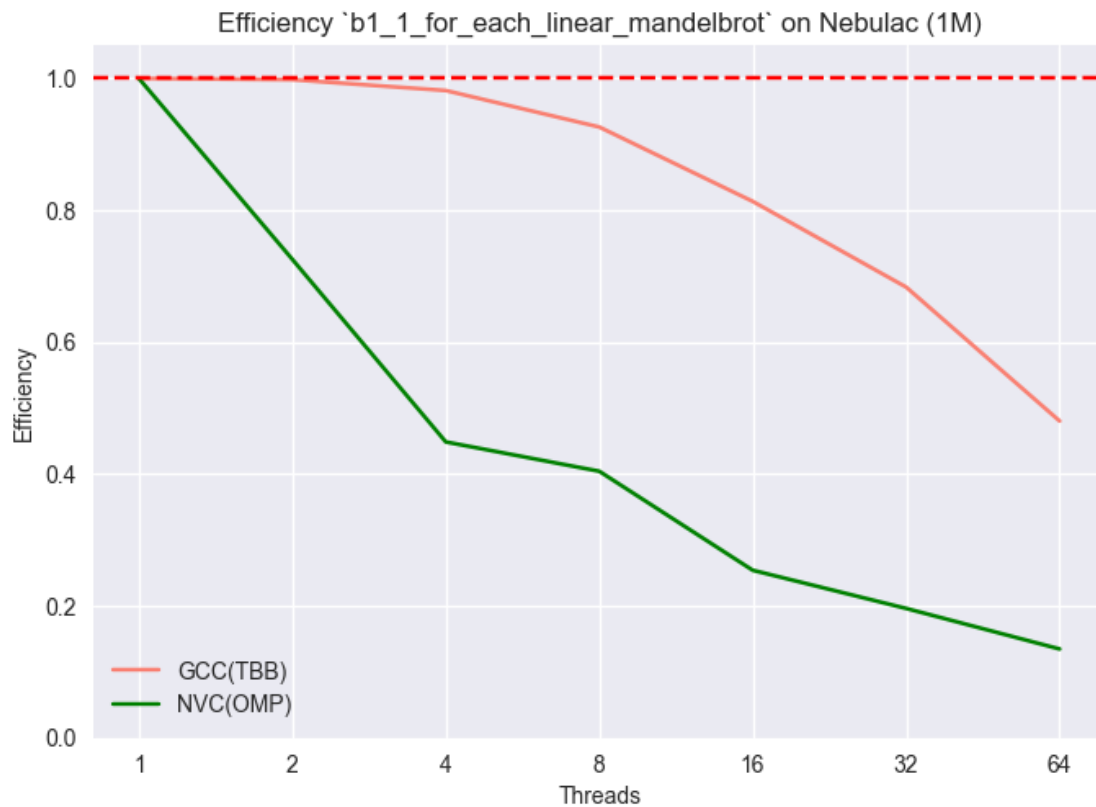
plot('Efficiency `b1_1_for_each_linear_mandelbrot` on Nebulac (1M)')

```

	threads	GCC(TBB)	NVC(OMP)
0	1	1.000000	1.000000
1	2	0.998040	0.725625
2	4	0.981600	0.448757
3	8	0.926262	0.404316
4	16	0.813480	0.254164
5	32	0.683455	0.196264
6	64	0.480429	0.134766

/var/folders/42/fk0jfrydidd1ztdldncqc1cw0000gn/T/ipykernel\_33509/3534785426.py:3

0: UserWarning: FixedFormatter should only be used together with FixedLocator  
 plt.gca().set\_xticklabels(['{:,.0f}'.format(x) for x in current\_values])



Strong scaling - b1\_2\_for\_each\_quadratic\_mandelbrot 32768 fixed input size with threads 1-64

```
[38]: def get_b1_2_mandelbrot_strong_scaling_algo(compiler_location:str,compiler_name:
↳str) -> pd.DataFrame:
    ## Threading data par_par
    par_par_threads = extraction_pandas_frame_algo_threaded(root_dir + f'/'
↳{compiler_location}/THREADS',
↳
↳    'b1_2_for_each_quadratic_mandelbrot_outer_std::execution::
↳parallel_policy_par',
↳
↳    [1,2,4,8,16,32,64],
↳
↳    COMP=compiler_name,
↳
↳    input_size=32768
```

```

↪)

## calc strong scaling
return calc_speedup_based_par(par_par_threads,
                               f"{compiler_name}",
                               input_size=32768
                               )

# load b1_2_for_each_quadratic_mandelbrot_threaded

instances = [
    ('GCC_TBB', 'GCC(TBB)'),
    ('NVHPC_Multicore', 'NVC(OMP)')
]

# collect data for instances
data = [get_b1_2_mandelbrot_strong_scaling_algo(*x) for x in instances]

b1_2_for_each_quadratic_mandelbrot_strong_scaling_merged = pd.merge(*data,
↪on='threads')

print(b1_2_for_each_quadratic_mandelbrot_strong_scaling_merged)

# plot strong scaling
ax = b1_2_for_each_quadratic_mandelbrot_strong_scaling_merged.plot(kind='bar',
                                                                    x='threads',
                                                                    align='center',
                                                                    ↪
↪color=[GCC_TBB_COLOR, NVC_OMP_COLOR]
                                                                    )

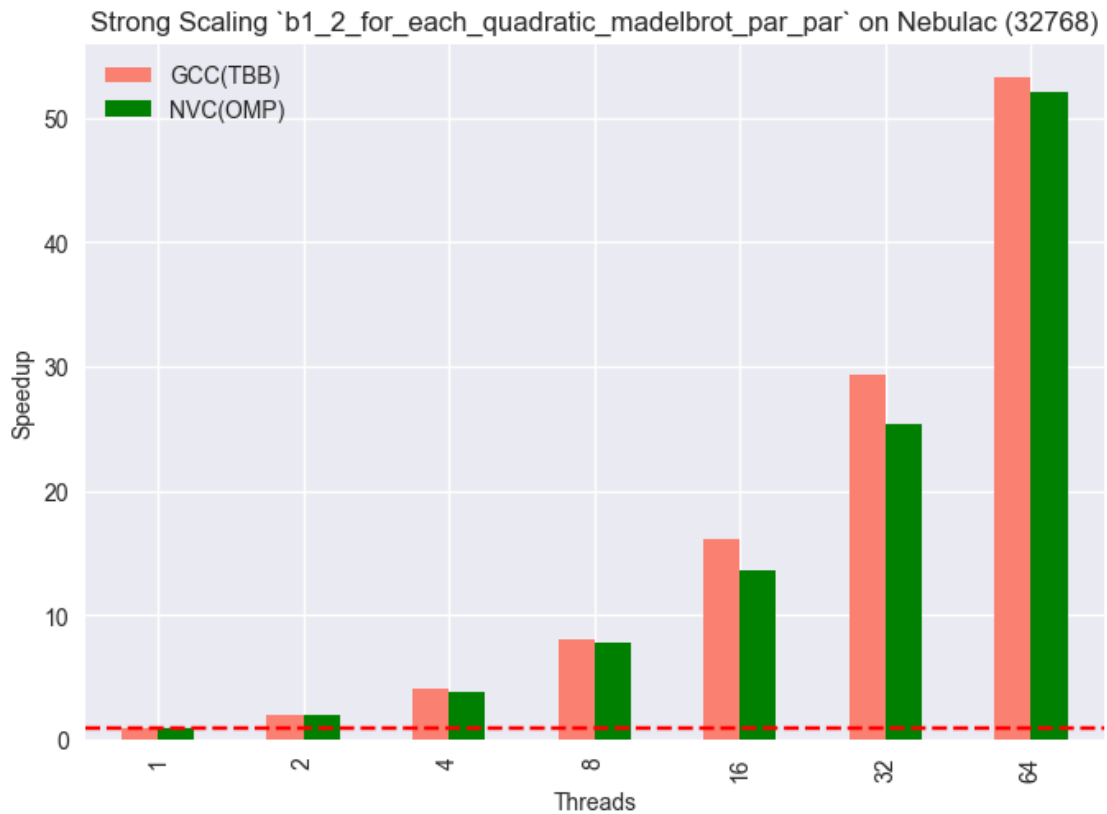
# adding horizontal line where there is speedup
ax.axhline(y=1, color='r', linestyle='--')

plt.ylabel('Speedup')
plt.xlabel('Threads')
plt.title('Strong Scaling `b1_2_for_each_quadratic_mandelbrot_par_par` on ↪
↪Nebulac (32768)')

plot('Strong Scaling `b1_2_for_each_quadratic_mandelbrot_par_par` on Nebulac ↪
↪(32768)')

```

	threads	GCC(TBB)	NVC(OMP)
0	1	1.000000	1.000000
1	2	2.049499	1.971748
2	4	4.059636	3.828575
3	8	8.035183	7.838173
4	16	16.078846	13.673510
5	32	29.304273	25.362917
6	64	53.257833	52.008864



[39]: *# efficiency graph*

```
b1_2_for_each_quadratic_mandelbrot_efficiency = _
    ↪ b1_2_for_each_quadratic_mandelbrot_strong_scaling_merged.copy()

instances = [
    ('GCC_TBB', 'GCC(TBB)'),
    ('NVHPC_Multicore', 'NVC(OMP)')
]
```

```

for _, compiler_name in instances:
    b1_2_for_each_quadratic_mandelbrot_efficiency[f'{compiler_name}'] =
↳ b1_2_for_each_quadratic_mandelbrot_efficiency[f'{compiler_name}'] /
↳ b1_1_for_each_linear_mandelbrot_efficiency['threads']

print(b1_2_for_each_quadratic_mandelbrot_efficiency)

# plot efficiency
ax = b1_2_for_each_quadratic_mandelbrot_efficiency.plot(x='threads',
                                                         color=[GCC_TBB_COLOR, NVC_OMP_COLOR]
                                                         )

# adding horizontal line where there is speedup
ax.axhline(y=1, color='r', linestyle='--')

plt.ylim(0, 1.10)

plt.xscale('log', base=2)
current_values = plt.gca().get_xticks()
plt.gca().set_xticklabels(['{:,.0f}'.format(x) for x in current_values])

plt.ylabel('Efficiency')
plt.xlabel('Threads')
plt.title('Efficiency `b1_2_for_each_quadratic_mandelbrot_par_par` on Nebulac
↳ (32768)')

plot('Efficiency `b1_2_for_each_quadratic_mandelbrot_par_par` on Nebulac
↳ (32768)')

```

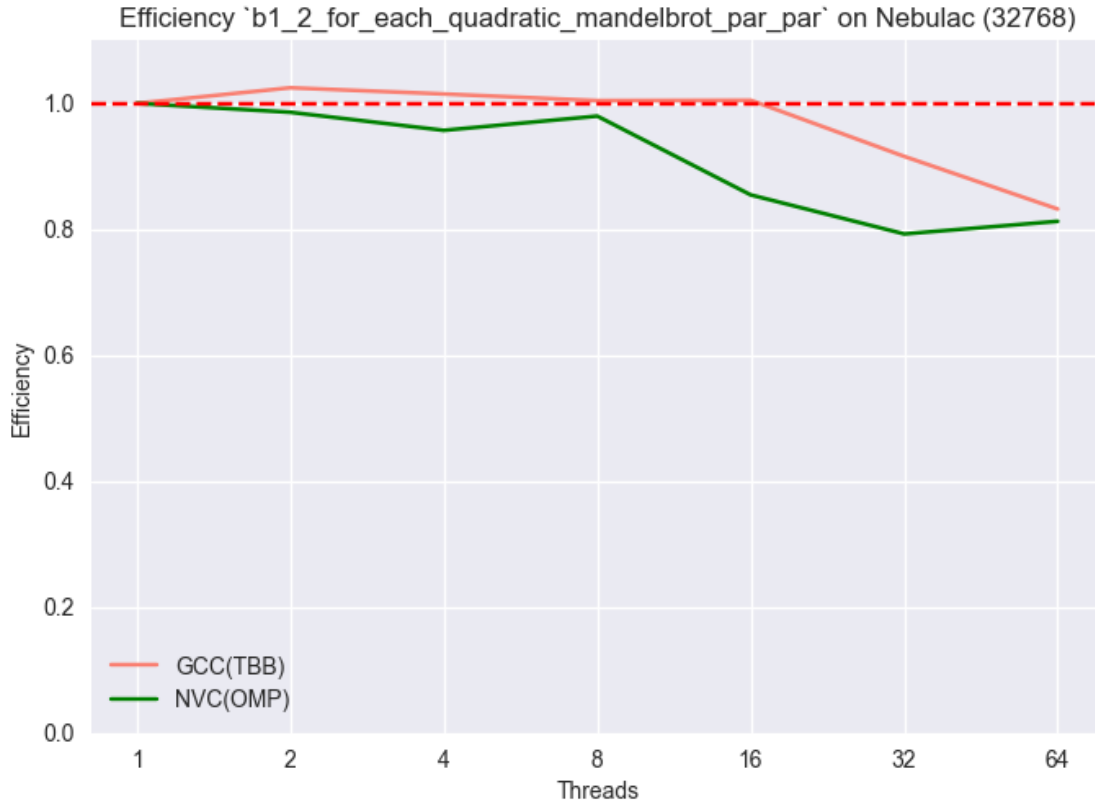
	threads	GCC(TBB)	NVC(OMP)
0	1	1.000000	1.000000
1	2	1.024749	0.985874
2	4	1.014909	0.957144
3	8	1.004398	0.979772
4	16	1.004928	0.854594
5	32	0.915759	0.792591
6	64	0.832154	0.812639

```

/var/folders/42/fk0jfryd1dd1ztdldncqc1cw0000gn/T/ipykernel_33509/1668598656.py:3
1: UserWarning: FixedFormatter should only be used together with FixedLocator
plt.gca().set_xticklabels(['{:,.0f}'.format(x) for x in current_values])

```





### 2.3.3 Instructions per second (Ips)

//TODO

[40]: *#TODO: do this!!!*

### 2.3.4 Performance Portability Calculation (Inter Compiler)

NONE

### 2.3.5 Findings for H3

**Findings b1\_1\_for\_each\_linear\_mandelbrot** From our runtime analysis we can see that GCC(TBB) has significant better performance than NVC(OMP). For small input sizes up to  $2^{16}$  the performance is about the same for both backends. For larger heterogenous workloads the performance of NVC(OMP) collapses and GCC(TBB) shines with its rather fast runtime. I cannot confirm this because I have not read through the code but I can remember to have read that GCC(TBB) does work stealing. This would explain why GCC(TBB) performs so good in comparison.

This poor performance of NVC(OMP) continues when looking at its strong scaling behavior. Already for small number of threads (4) the speedup is really poorly leading to efficiency of 40% and less. Compared to NVC(OMP), GCC(TBB) has fairly good strong scaling. Although for higher

number of threads it seems to slowly top off. Moving from 32 to 64 threads does not bring a huge improvement.

//TODO: IPS

Key Observations: \* GCC(TBB) works really well with heterogenous workloads. \* NVC(OMP) struggles a lot with large input sizes \* NVC(OMP) has really bad strong scaling \* GCC(TBB) great scaling but slows down at 32->64 threads

**Findings b1\_2\_for\_each\_quadratic\_mandelbrot** The runtime analysis shows that GCC(TBB) has quite bad runtime compared to NVC(OMP). The runtime of NVC(OMP) especially on smaller input sizes is by a magnitude faster. For larger input sizes NVC(OMP) slowly gets worse but still way better than GCC(TBB).

Strong Scaling is quite interesting for GCC(TBB). For up to 16 threads we have perfect speedup, for 32 and 64 threads we have around 80-90% efficiency. NVC(OMP) nearly follows this trend, since perfect speedup stops at 8 threads and even before we seem to be around 99% all the time. From 16 till 64 threads the speedup seems to be stable not with an efficiency of about 80%.

//TODO: IPS

Key Observations: \* GCC(TBB) runtime is quite bad comparing to NVC(OMP) \* NVC(OMP) has great runtime for small input sizes but for larger input sizes it collapses. \* Strong Scaling of GCC(TBB) is quite strange. Perfect speedup until 16 threads. Then slowly degrades \* NVC(OMP) Strong scaling nearly perfect speedup until 8 Threads. Then collapses really fast.

**GPU Findings** Although it is possible to rewrite the code of b1\_1\_linear\_mandelbrot to run on gpus nebula does not support GPUS.

### 2.3.6 Hypothesis Findings

This hypothesis is **true!**

## 2.4 H4

Different compilers/backends may fallback to sequential algorithms, leading to better performance.

### 2.4.1 Time

**Time Comparison - b4\_1\_merge\_cutoff\_wrapper\_par** Check how the runtime without constraining the threads develops with increasing input size

```
[41]: def get_b4_1_merge_data_algo(compiler_location:str,compiler_name:str) -> pd.
      ↪DataFrame:
      # load b4_1_merge_cutoff_wrapper_par
      df = extraction_pandas_frame_algo(root_dir + f'/{compiler_location}/DEFAULT/
      ↪b4_1_merge_cutoff_wrapper_par__Default.csv',COMP=compiler_name)

      df = df.
      ↪drop(columns=['name','cpu_time','time_unit','median','stddev','Compiler'])
```

```

df = df.rename(columns={'real_time':f'{compiler_name}'})

return df

instances = [
    ('GCC_TBB', 'GCC(TBB)'),
    ('NVHPC_Multicore', 'NVC(OMP)')
]

# collect data for instances
data = [get_b4_1_merge_data_algo(*x) for x in instances]

# merge for plotting
b4_1_merge_time_merged = pd.merge(*data, on='n')

# plot
ax = b4_1_merge_time_merged.
    plot(kind='bar', x='n', align='center', color=[GCC_TBB_COLOR, NVC_OMP_COLOR])
print(b4_1_merge_time_merged)

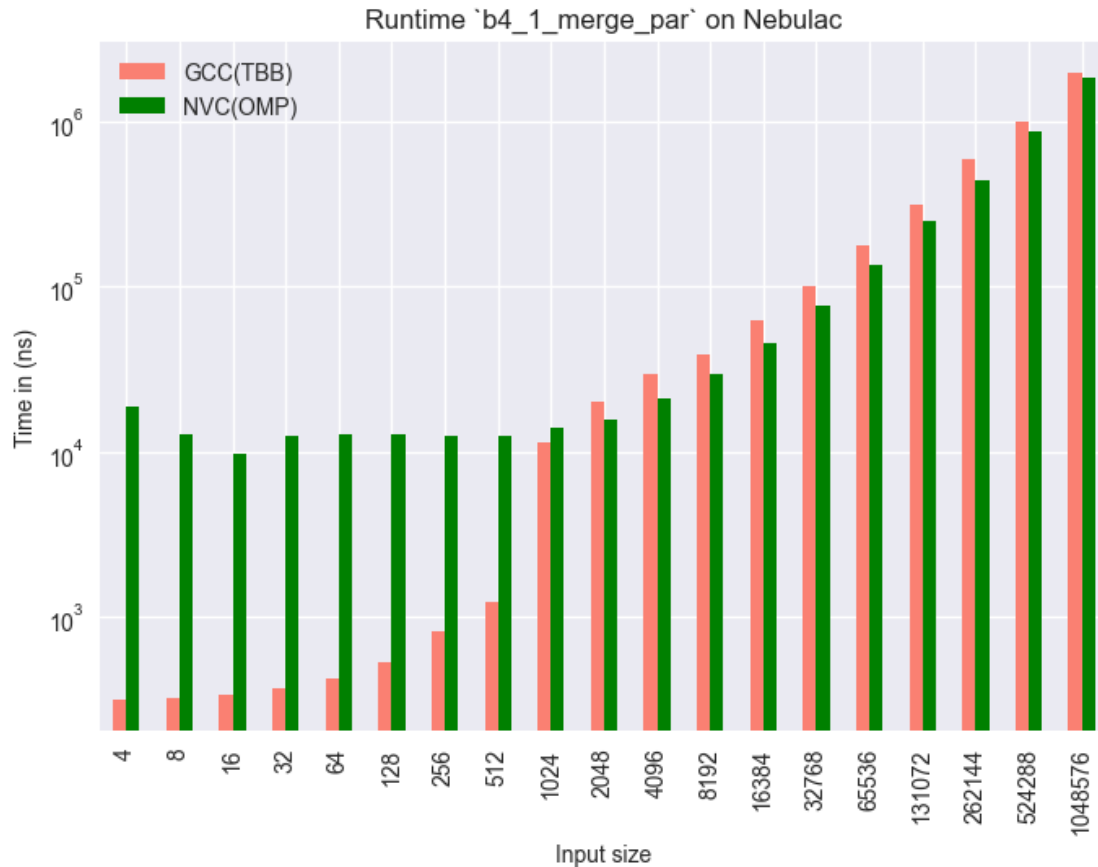
ax.set_yscale('log', base=10)
ax.set_ylabel('Time in (ns)')
ax.set_xlabel('Input size')
ax.set_title('Runtime `b4_1_merge_par` on Nebulac')

plot('Runtime `b4_1_merge_par` on Nebulac')

```

	GCC(TBB)	n	NVC(OMP)
0	316.468	4	18604.10
1	321.506	8	12710.60
2	335.089	16	9659.61
3	369.409	32	12364.40
4	420.418	64	12592.80
5	534.943	128	12694.20
6	808.866	256	12412.20
7	1234.930	512	12569.90
8	11440.400	1024	13785.60
9	19857.800	2048	15758.90
10	29289.600	4096	21013.90
11	38446.200	8192	29414.00
12	62327.600	16384	45686.90
13	100653.000	32768	76358.20
14	178404.000	65536	133716.00
15	312728.000	131072	247368.00
16	589382.000	262144	433974.00
17	987794.000	524288	854998.00

18 1962430.000 1048576 1808490.00



**Time Comparison - b4\_2\_stable\_sort\_cutoff\_already\_sorted\_par** Check how the runtime without constraining the threads develops with increasing input size

```
[42]: def get_b4_2_stable_sort_cutoff_already_sorted_data_algo(compiler_location:
      ↪str, compiler_name: str) -> pd.DataFrame:
      # load b4_2_stable_sort_cutoff_already_sorted_par
      df = extraction_pandas_frame_algo(root_dir + f'/{compiler_location}/DEFAULT/
      ↪b4_2_stable_sort_cutoff_already_sorted_par_Default.csv', COMP=compiler_name)

      df = df.
      ↪drop(columns=['name', 'cpu_time', 'time_unit', 'median', 'stddev', 'Compiler'])
      df = df.rename(columns={'real_time': f'{compiler_name}'})

      return df

instances = [
```

```

    ('GCC_TBB', 'GCC(TBB)'),
    ('NVHPC_Multicore', 'NVC(OMP)')
]

# collect data for instances
data = [get_b4_2_stable_sort_cutoff_already_sorted_data_algo(*x) for x in
        instances]

# merge for plotting
b4_2_stable_sort_cutoff_already_sorted_time_merged = pd.merge(*data, on='n')

b4_2_stable_sort_cutoff_already_sorted_time_merged

```

```

[42]:
GCC(TBB)      n  NVC(OMP)
0      368.944    4   35715.0
1      377.802    8   38181.3
2      442.495   16  46927.8
3      517.475   32  57606.2
4      716.980   64  67822.5
5     1068.450  128  70569.7
6     1728.000  256  68832.1
7    11583.600  512  69968.2
8    18797.100 1024  71040.6
9    24115.900 2048  77355.3
10   29091.800 4096  89864.3
11   34338.600 8192 106995.0
12   40210.900 16384 138450.0
13   48028.000 32768 209635.0
14   70874.700 65536 344905.0
15  111750.000 131072 1133700.0
16  133280.000 262144 2146620.0
17  247739.000 524288 4340560.0
18  450064.000 1048576 8852250.0

```

**Time Comparison - b4\_2\_stable\_sort\_cutoff\_decrement\_sorted\_par** Check how the run-time without constraining the threads develops with increasing the input size

```

[43]: def b4_2_stable_sort_cutoff_decrement_sorted_data_algo(compiler_location:
        str, compiler_name: str) -> pd.DataFrame:
        # load b4_2_stable_sort_cutoff_decrement_sorted_par
        df = extraction_pandas_frame_algo(root_dir + f'/{compiler_location}/DEFAULT/
        b4_2_stable_sort_cutoff_decrement_sorted_par__Default.
        csv', COMP=compiler_name)

        df = df.
        drop(columns=['name', 'cpu_time', 'time_unit', 'median', 'stddev', 'Compiler'])

```

```

df = df.rename(columns={'real_time':f'{compiler_name}'})

return df

instances = [
    ('GCC_TBB', 'GCC(TBB)'),
    ('NVHPC_Multicore', 'NVC(OMP)')
]

# collect data for instances
data = [b4_2_stable_sort_cutoff_decrement_sorted_data_algo(*x) for x in instances]

# merge for plotting
b4_2_stable_sort_cutoff_decrement_sorted_time_merged = pd.merge(*data, on='n')

b4_2_stable_sort_cutoff_decrement_sorted_time_merged

```

```

[43]:
      GCC(TBB)      n  NVC(OMP)
0      338.949      4    60174.3
1      349.494      8    40802.2
2      411.047     16    47735.6
3      488.623     32    54474.7
4      680.957     64    66238.7
5     1065.140    128    66247.3
6     1774.640    256    76998.2
7    11271.100    512   89525.2
8    21225.700   1024   78420.0
9    26749.200   2048   79426.2
10   31401.600   4096   86835.0
11   36597.300   8192  108990.0
12   40235.400  16384  127920.0
13   50542.100  32768  194359.0
14   76841.300  65536  316589.0
15  122539.000 131072  674739.0
16  179121.000 262144 1282040.0
17  232129.000 524288 3355980.0
18  356085.0001048576 8735320.0

```

**Time Comparison - b4\_2\_stable\_sort\_cutoff\_not\_sorted\_par** Check how the runtime without constraining the threads develops with increasing the input size

```

[44]: def b4_2_stable_sort_cutoff_not_sorted_data_algo(compiler_location:
      ↪str, compiler_name:str) -> pd.DataFrame:
      # load b4_2_stable_sort_cutoff_not_sorted_par

```

```

    df = extraction_pandas_frame_algo(root_dir + f'/{compiler_location}/DEFAULT/
↳b4_2_stable_sort_cutoff_not_sorted_par__Default.csv',COMP=compiler_name)

    df = df.
↳drop(columns=['name','cpu_time','time_unit','median','stddev','Compiler'])
    df = df.rename(columns={'real_time':f'{compiler_name}'})

    return df

instances = [
    ('GCC_TBB','GCC(TBB)'),
    ('NVHPC_Multicore','NVC(OMP)')
]

# collect data for instances
data = [b4_2_stable_sort_cutoff_not_sorted_data_algo(*x) for x in instances]

# merge for plotting
b4_2_stable_sort_cutoff_not_sorted_time_merged = pd.merge(*data, on='n')

b4_2_stable_sort_cutoff_not_sorted_time_merged

```

```

[44]:
GCC(TBB)      n  NVC(OMP)
0      352.603    4   39425.8
1      360.778    8  40345.2
2      414.446   16  50330.8
3      493.368   32  57209.6
4      697.705   64  64031.9
5     1066.040  128  69020.2
6     1832.720  256  69004.0
7    15730.300  512  76250.8
8    17388.200 1024  78127.8
9    23525.200 2048  82192.8
10   29030.300 4096  88511.0
11   34924.300 8192 105989.0
12   39211.700 16384 137135.0
13   52055.800 32768 268668.0
14   77458.800 65536 447598.0
15   89930.000 131072 802818.0
16  140852.000 262144 2235230.0
17  252975.000 524288 4314820.0
18  453777.000 1048576 8693540.0

```

```

[45]: ### Total graphs

```

```

fig, axes = plt.subplots(nrows=1, ncols=3,figsize=(16, 6))

# already sorted
ax_1 = b4_2_stable_sort_cutoff_already_sorted_time_merged.
    ↪plot(kind='bar',x='n',align='center',color=[GCC_TBB_COLOR,NVC_OMP_COLOR],ax=axes[0])

ax_1.set_yscale('log', base=10)
ax_1.set_ylabel('Time in (ns)')
ax_1.set_xlabel('Input size')
ax_1.set_title('already sorted')

# decrement sorted
ax_2 = b4_2_stable_sort_cutoff_decrement_sorted_time_merged.
    ↪plot(kind='bar',x='n',align='center',color=[GCC_TBB_COLOR,NVC_OMP_COLOR],ax=axes[1])

ax_2.set_yscale('log', base=10)
ax_2.set_ylabel('Time in (ns)')
ax_2.set_xlabel('Input size')
ax_2.set_title('decremented sorted')

# not sorted
ax_3 = b4_2_stable_sort_cutoff_not_sorted_time_merged.
    ↪plot(kind='bar',x='n',align='center',color=[GCC_TBB_COLOR,NVC_OMP_COLOR],ax=axes[2])

ax_3.set_yscale('log', base=10)
ax_3.set_ylabel('Time in (ns)')
ax_3.set_xlabel('Input size')
ax_3.set_title('not sorted')

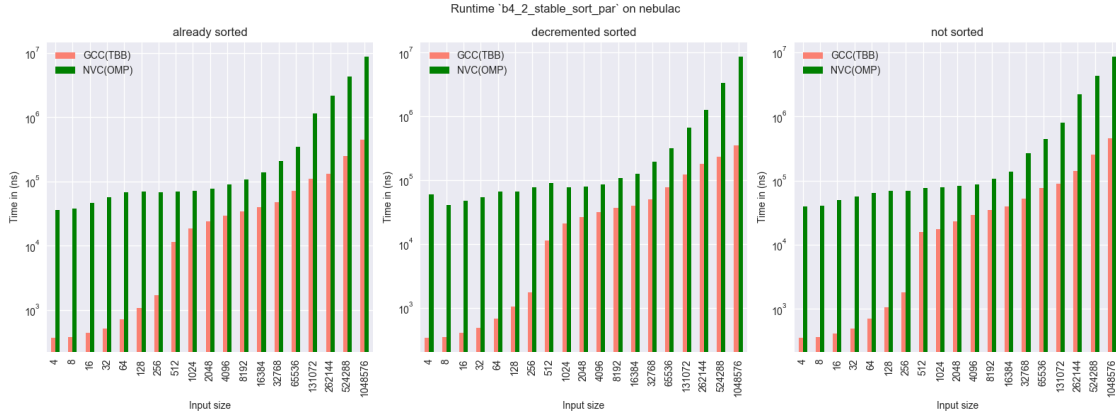
fig.suptitle("Runtime `b4_2_stable_sort_par` on nebulac")

fig.tight_layout()

plot("Runtime `b4_2_stable_sort_par` on nebulac")

```





**Time Comparison - b4\_3\_set\_union\_cutoff\_one\_empty\_par** Check how the runtime without constraining the threads develops with increasing the input size

```
[46]: def b4_3_set_union_cutoff_one_empty_data_algo(compiler_location:
    ↪str, compiler_name:str) -> pd.DataFrame:
    # load b4_3_set_union_cutoff_one_empty_par
    df = extraction_pandas_frame_algo(root_dir + f'/{compiler_location}/DEFAULT/
    ↪b4_3_set_union_cutoff_one_empty_par_Default.csv', COMP=compiler_name)

    df = df.
    ↪drop(columns=['name', 'cpu_time', 'time_unit', 'median', 'stddev', 'Compiler'])
    df = df.rename(columns={'real_time': f'{compiler_name}'})

    return df

instances = [
    ('GCC_TBB', 'GCC(TBB)'),
    ('NVHPC_Multicore', 'NVC(OMP)')
]

# collect data for instances
data = [b4_3_set_union_cutoff_one_empty_data_algo(*x) for x in instances]

# merge for plotting
b4_3_set_union_cutoff_one_empty_time_merged = pd.merge(*data, on='n')

b4_3_set_union_cutoff_one_empty_time_merged
```

```
[46]:      GCC(TBB)      n  NVC(OMP)
0      326.087      4    10851.0
```

1	324.204	8	10833.2
2	321.997	16	10766.7
3	325.720	32	11349.5
4	329.574	64	11904.8
5	340.452	128	11927.6
6	365.500	256	10750.7
7	451.627	512	11298.5
8	40197.000	1024	13247.5
9	45220.000	2048	16316.1
10	52436.000	4096	18658.5
11	58877.600	8192	24456.8
12	69873.900	16384	34021.1
13	85240.800	32768	47653.7
14	114969.000	65536	73370.5
15	167449.000	131072	122924.0
16	259516.000	262144	229783.0
17	440010.000	524288	271679.0
18	734519.000	1048576	539714.0

**Time Comparison - b4\_3\_set\_union\_cutoff\_one\_wholly\_greater\_par** Check how the run-time without constraining the threads develops with increasing the input size

```
[47]: def b4_3_set_union_cutoff_one_wholly_greater_data_algo(compiler_location:
    ↪str, compiler_name:str) -> pd.DataFrame:
        # load b4_3_set_union_cutoff_one_wholly_greater_par
        df = extraction_pandas_frame_algo(root_dir + f'/{compiler_location}/DEFAULT/
    ↪b4_3_set_union_cutoff_one_wholly_greater_par__Default.
    ↪csv', COMP=compiler_name)

        df = df.
    ↪drop(columns=['name', 'cpu_time', 'time_unit', 'median', 'stddev', 'Compiler'])
        df = df.rename(columns={'real_time': f'{compiler_name}'})

        return df

instances = [
    ('GCC_TBB', 'GCC(TBB)'),
    ('NVHPC_Multicore', 'NVC(OMP)')
]

# collect data for instances
data = [b4_3_set_union_cutoff_one_wholly_greater_data_algo(*x) for x in
    ↪instances]

# merge for plotting
```

```
b4_3_set_union_cutoff_one_wholly_greater_time_merged = pd.merge(*data, on='n')

b4_3_set_union_cutoff_one_wholly_greater_time_merged
```

```
[47]:
```

	GCC(TBB)	n	NVC(OMP)
0	311.719	4	11793.4
1	314.916	8	10319.4
2	328.652	16	10256.8
3	348.850	32	11443.7
4	390.262	64	12244.2
5	468.751	128	11400.5
6	679.696	256	11192.1
7	49155.600	512	11568.9
8	50183.300	1024	14354.5
9	57610.800	2048	18264.4
10	69642.300	4096	23981.7
11	80578.400	8192	29836.2
12	95897.400	16384	44702.5
13	124799.000	32768	71892.7
14	173364.000	65536	127640.0
15	271154.000	131072	237458.0
16	392556.000	262144	399929.0
17	659520.000	524288	787602.0
18	1308430.000	1048576	1606560.0

**Time Comparison - b4\_3\_set\_union\_cutoff\_front\_overhang\_par** Check how the runtime without constraining the threads develops with increasing the input size

```
[48]: def b4_3_set_union_cutoff_front_overhang_data_algo(compiler_location:
    ↪str, compiler_name:str) -> pd.DataFrame:
    # load b4_3_set_union_cutoff_front_overhang_par
    df = extraction_pandas_frame_algo(root_dir + f'/{compiler_location}/DEFAULT/
    ↪b4_3_set_union_cutoff_front_overhang_par__Default.csv', COMP=compiler_name)

    df = df.
    ↪drop(columns=['name', 'cpu_time', 'time_unit', 'median', 'stddev', 'Compiler'])
    df = df.rename(columns={'real_time': f'{compiler_name}'})

    return df

instances = [
    ('GCC_TBB', 'GCC(TBB)'),
    ('NVHPC_Multicore', 'NVC(OMP)')
]

# collect data for instances
```

```

data = [b4_3_set_union_cutoff_front_overhang_data_algo(*x) for x in instances]

# merge for plotting
b4_3_set_union_cutoff_front_overhang_time_merged = pd.merge(*data, on='n')

b4_3_set_union_cutoff_front_overhang_time_merged

```

```

[48]:
GCC(TBB)      n  NVC(OMP)
0      313.944    4   14325.9
1      322.226    8   10614.5
2      319.306   16   10838.6
3      343.501   32   10981.2
4      381.680   64   11951.3
5      459.557  128   12308.7
6      650.691  256   12733.4
7    106766.000  512   12774.6
8    104800.000 1024   13876.9
9    109756.000 2048   16749.7
10   113822.000 4096   20691.7
11   122026.000 8192   29327.8
12   136699.000 16384  43153.2
13   162981.000 32768  76306.1
14   194553.000 65536 134725.0
15   284624.000 131072 252869.0
16   465580.000 262144 439851.0
17   721783.000 524288 773505.0
18  1663210.000 1048576 1708470.0

```

```

[49]: # all set_union graphs

y_lim = 5*pow(10,6)
fig, axes = plt.subplots(nrows=1, ncols=3,figsize=(16, 6))

# already sorted
ax_1 = b4_3_set_union_cutoff_one_empty_time_merged.
    plot(kind='bar',x='n',align='center',color=[GCC_TBB_COLOR,NVC_OMP_COLOR],ax=axes[0])

ax_1.set_yscale('log', base=10)
ax_1.set_ylabel('Time in (ns)')
ax_1.set_xlabel('Input size')
ax_1.set_title('One empty')
ax_1.set_ylim(0,y_lim)

# decrement sorted

```

```

ax_2 = b4_3_set_union_cutoff_one_wholly_greater_time_merged.
    plot(kind='bar',x='n',align='center',color=[GCC_TBB_COLOR,NVC_OMP_COLOR],ax=axes[1])

ax_2.set_yscale('log', base=10)
ax_2.set_ylabel('Time in (ns)')
ax_2.set_xlabel('Input size')
ax_2.set_title('One wholly greater')
ax_2.set_ylim(0,y_lim)

# not sorted
ax_3 = b4_3_set_union_cutoff_front_overhang_time_merged.
    plot(kind='bar',x='n',align='center',color=[GCC_TBB_COLOR,NVC_OMP_COLOR],ax=axes[2])

ax_3.set_yscale('log', base=10)
ax_3.set_ylabel('Time in (ns)')
ax_3.set_xlabel('Input size')
ax_3.set_title('Front overhang')
ax_3.set_ylim(0,y_lim)

fig.suptitle("Runtime `b4_3_set_union` on nebulac")

fig.tight_layout()

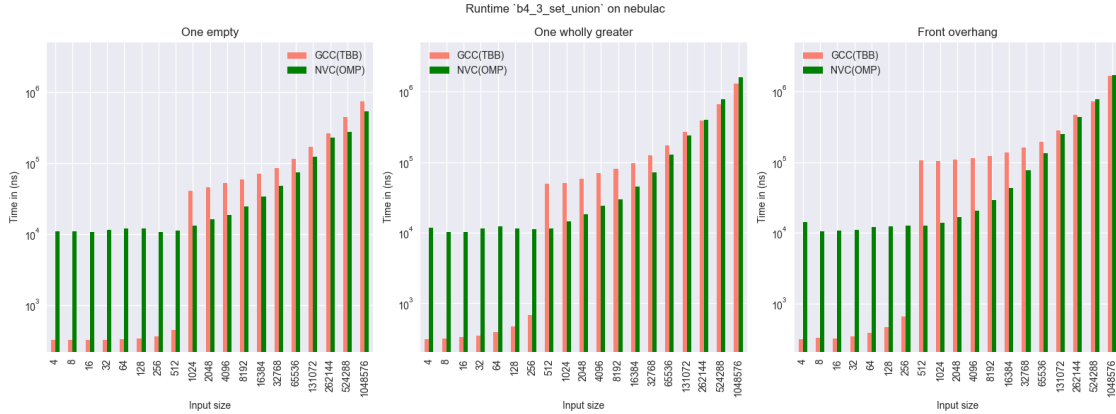
plot("Runtime `b4_3_set_union` on nebulac")

```

```

/var/folders/42/fk0jfryd1dd1ztdldncqc1cw0000gn/T/ipykernel_33509/2834800862.py:1
5: UserWarning: Attempt to set non-positive ylim on a log-scaled axis will be
ignored.
    ax_1.set_ylim(0,y_lim)
/var/folders/42/fk0jfryd1dd1ztdldncqc1cw0000gn/T/ipykernel_33509/2834800862.py:2
4: UserWarning: Attempt to set non-positive ylim on a log-scaled axis will be
ignored.
    ax_2.set_ylim(0,y_lim)
/var/folders/42/fk0jfryd1dd1ztdldncqc1cw0000gn/T/ipykernel_33509/2834800862.py:3
4: UserWarning: Attempt to set non-positive ylim on a log-scaled axis will be
ignored.
    ax_3.set_ylim(0,y_lim)

```



**Time Comparison - b4\_4\_set\_difference\_cutoff\_left\_empty\_par** Check how the runtime without constraining the threads develops with increasing the input size

```
[50]: def b4_4_set_difference_cutoff_left_empty_data_algo(compiler_location:
    ↪str, compiler_name:str) -> pd.DataFrame:
    # load b4_4_set_difference_cutoff_left_empty_par
    df = extraction_pandas_frame_algo(root_dir + f'/{compiler_location}/DEFAULT/
    ↪b4_4_set_difference_cutoff_left_empty_par_Default.csv', COMP=compiler_name)

    df = df.
    ↪drop(columns=['name', 'cpu_time', 'time_unit', 'median', 'stddev', 'Compiler'])
    df = df.rename(columns={'real_time': f'{compiler_name}'})

    return df

instances = [
    ('GCC_TBB', 'GCC(TBB)'),
    ('NVHPC_Multicore', 'NVC(OMP)')
]

# collect data for instances
data = [b4_4_set_difference_cutoff_left_empty_data_algo(*x) for x in instances]

# merge for plotting
b4_4_set_difference_cutoff_left_empty_time_merged = pd.merge(*data, on='n')

b4_4_set_difference_cutoff_left_empty_time_merged
```

```
[50]:      GCC(TBB)      n  NVC(OMP)
0      271.681      4    276.387
```

1	271.339	8	276.634
2	270.805	16	277.559
3	271.044	32	276.882
4	271.582	64	275.909
5	271.316	128	275.138
6	271.256	256	276.296
7	271.808	512	275.656
8	270.898	1024	278.104
9	271.789	2048	277.384
10	272.240	4096	278.012
11	271.826	8192	278.387
12	272.477	16384	274.661
13	271.932	32768	278.474
14	271.805	65536	276.292
15	272.744	131072	275.395
16	272.043	262144	275.628
17	272.034	524288	275.536
18	271.120	1048576	274.451

**Time Comparison - b4\_4\_set\_difference\_cutoff\_right\_empty\_par** Check how the runtime without constraining the threads develops with increasing the input size

```
[51]: def b4_4_set_difference_cutoff_right_empty_data_algo(compiler_location:
      ↪str, compiler_name:str) -> pd.DataFrame:
      ↪    # load b4_4_set_difference_cutoff_right_empty_par
      ↪    df = extraction_pandas_frame_algo(root_dir + f'/{compiler_location}/DEFAULT/
      ↪    b4_4_set_difference_cutoff_right_empty_par__Default.csv', COMP=compiler_name)

      ↪    df = df.
      ↪    drop(columns=['name', 'cpu_time', 'time_unit', 'median', 'stddev', 'Compiler'])
      ↪    df = df.rename(columns={'real_time': f'{compiler_name}'})

      ↪    return df

instances = [
    ('GCC_TBB', 'GCC(TBB)'),
    ('NVHPC_Multicore', 'NVC(OMP)')
]

# collect data for instances
data = [b4_4_set_difference_cutoff_right_empty_data_algo(*x) for x in instances]

# merge for plotting
b4_4_set_difference_cutoff_right_empty_time_merged = pd.merge(*data, on='n')
```

```
b4_4_set_difference_cutoff_right_empty_time_merged
```

```
[51]:
```

	GCC(TBB)	n	NVC(OMP)
0	5754.57	4	11364.4
1	10159.30	8	10020.1
2	15649.10	16	11201.5
3	20277.10	32	11820.0
4	23944.70	64	12178.8
5	25915.80	128	11533.3
6	29137.80	256	10760.6
7	32367.20	512	12566.4
8	37000.80	1024	11279.5
9	43527.10	2048	16256.3
10	51315.10	4096	19715.2
11	57438.80	8192	23013.1
12	69387.70	16384	30760.4
13	88699.70	32768	44154.9
14	122034.00	65536	72543.7
15	173592.00	131072	125295.0
16	269756.00	262144	239945.0
17	456072.00	524288	288062.0
18	784725.00	1048576	539903.0

**Time Comparison - b4\_4\_set\_difference\_cutoff\_wholly\_greater\_par** Check how the run-time without constraining the threads develops with increasing the input size

```
[52]: def b4_4_set_difference_cutoff_wholly_greater_data_algo(compiler_location:
    ↪str, compiler_name:str) -> pd.DataFrame:
    # load b4_4_set_difference_cutoff_wholly_greater_par
    df = extraction_pandas_frame_algo(root_dir + f'/{compiler_location}/DEFAULT/
    ↪b4_4_set_difference_cutoff_wholly_greater_par__Default.
    ↪csv', COMP=compiler_name)

    df = df.
    ↪drop(columns=['name', 'cpu_time', 'time_unit', 'median', 'stddev', 'Compiler'])
    df = df.rename(columns={'real_time': f'{compiler_name}'})

    return df

instances = [
    ('GCC_TBB', 'GCC(TBB)'),
    ('NVHPC_Multicore', 'NVC(OMP)')
]

# collect data for instances
```



```

data = [b4_4_set_difference_cutoff_wholly_greater_data_algo(*x) for x in
↳instances]

# merge for plotting
b4_4_set_difference_cutoff_wholly_greater_time_merged = pd.merge(*data, on='n')

b4_4_set_difference_cutoff_wholly_greater_time_merged

```

```

[52]:
GCC(TBB)      n      NVC(OMP)
0      7170.88      4      319.097
1     14216.20      8      322.365
2     20393.90     16      325.312
3     25043.20     32      334.878
4     28750.90     64      383.077
5     30114.30    128      431.048
6     32277.40    256      535.874
7     35688.70    512      752.974
8     39109.00   1024     1163.280
9     44349.20   2048     1932.540
10    51768.20   4096     3478.980
11    58051.70   8192     6571.730
12    69575.60  16384    12755.400
13    87667.30  32768    25096.800
14   122154.00  65536    49990.800
15   171661.00 131072    99869.600
16   282821.00 262144   198914.000
17   375885.00 524288   444354.000
18   798900.00 1048576  866868.000

```

**Time Comparison - b4\_4\_set\_difference\_cutoff\_intersected\_par** Check how the runtime without constraining the threads develops with increasing the input size

```

[53]: def b4_4_set_difference_cutoff_intersected_data_algo(compiler_location:
↳str, compiler_name:str) -> pd.DataFrame:
    # load b4_4_set_difference_cutoff_intersected_par
    df = extraction_pandas_frame_algo(root_dir + f'/{compiler_location}/DEFAULT/
↳b4_4_set_difference_cutoff_intersected_par__Default.csv', COMP=compiler_name)

    df = df.
↳drop(columns=['name', 'cpu_time', 'time_unit', 'median', 'stddev', 'Compiler'])
    df = df.rename(columns={'real_time': f'{compiler_name}'})

    return df

instances = [

```

```

    ('GCC_TBB','GCC(TBB)'),
    ('NVHPC_Multicore','NVC(OMP)')
]

# collect data for instances
data = [b4_4_set_difference_cutoff_intersected_data_algo(*x) for x in instances]

# merge for plotting
b4_4_set_difference_cutoff_intersected_time_merged = pd.merge(*data, on='n')

b4_4_set_difference_cutoff_intersected_time_merged

```

```

[53]:
      GCC(TBB)      n  NVC(OMP)
0      323.870      4    323.365
1      332.182      8    325.776
2      355.723     16    331.814
3      416.589     32    341.784
4      500.004     64    383.400
5      669.305    128    443.247
6      996.399    256    542.849
7    113858.000    512    769.255
8     95178.900   1024   1174.490
9     93406.200   2048   1935.620
10    92759.300   4096   3554.790
11    95812.800   8192   6648.150
12    95677.300  16384  12838.700
13    98291.000  32768  25300.200
14   103170.000  65536  50257.800
15   119785.000 131072  99864.000
16   142569.000 262144 199209.000
17   181368.000 524288 416028.000
18   282373.000 1048576 882834.000

```

```

[54]: # all set_difference graphs

fig, axes = plt.subplots(nrows=2, ncols=2,figsize=(16, 9))

# left empty
ax_1 = b4_4_set_difference_cutoff_left_empty_time_merged.
    ↪plot(kind='bar',x='n',align='center',color=[GCC_TBB_COLOR,NVC_OMP_COLOR],ax=axes[0,0])

ax_1.set_ylabel('Time in (ns)')
ax_1.set_xlabel('Input size')
ax_1.set_title('Left empty')
ax_1.set_ylim(0,500)

```

```

# right empty
ax_2 = b4_4_set_difference_cutoff_right_empty_time_merged.
    ↪plot(kind='bar',x='n',align='center',color=[GCC_TBB_COLOR,NVC_OMP_COLOR],ax=axes[0,1])

ax_2.set_yscale('log', base=10)
ax_2.set_ylabel('Time in (ns)')
ax_2.set_xlabel('Input size')
ax_2.set_title('Right empty')

# wholly greater
ax_3 = b4_4_set_difference_cutoff_wholly_greater_time_merged.
    ↪plot(kind='bar',x='n',align='center',color=[GCC_TBB_COLOR,NVC_OMP_COLOR],ax=axes[1,0])

ax_3.set_yscale('log', base=10)
ax_3.set_ylabel('Time in (ns)')
ax_3.set_xlabel('Input size')
ax_3.set_title('Wholly greater')

# intersected
ax_4 = b4_4_set_difference_cutoff_intersected_time_merged.
    ↪plot(kind='bar',x='n',align='center',color=[GCC_TBB_COLOR,NVC_OMP_COLOR],ax=axes[1,1])

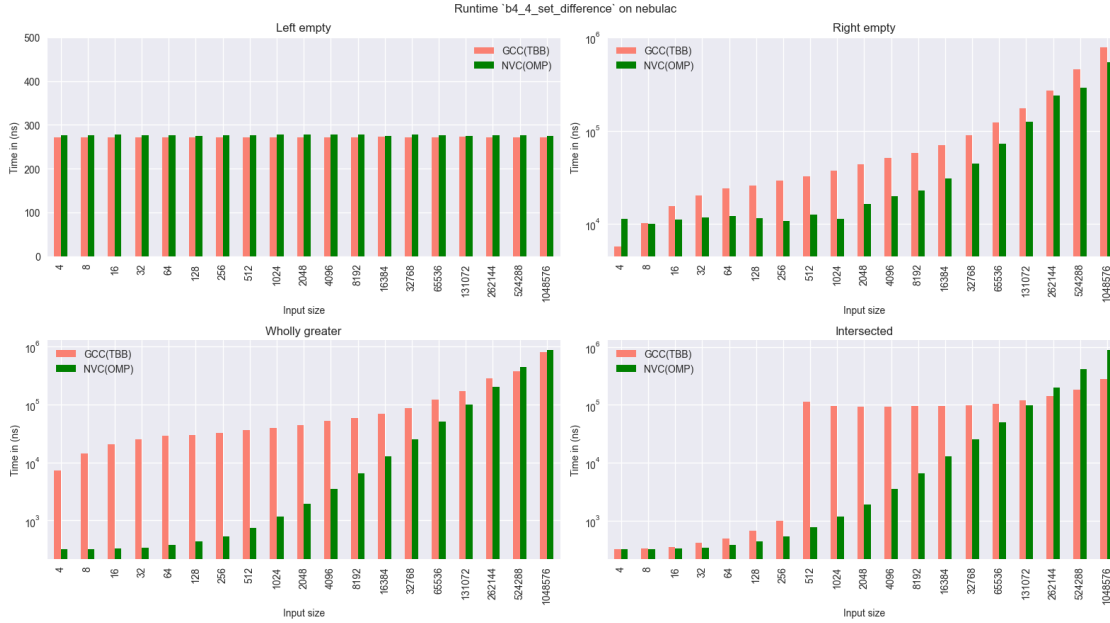
ax_4.set_yscale('log', base=10)
ax_4.set_ylabel('Time in (ns)')
ax_4.set_xlabel('Input size')
ax_4.set_title('Intersected')

fig.suptitle("Runtime `b4_4_set_difference` on nebulac")

fig.tight_layout()

plot("Runtime `b4_4_set_difference` on nebulac")

```



## 2.4.2 Strong Scaling

$$S(p) = T(1) / T(p)$$

As based we use the parallel algorithm (1 thread)

**Strong Scaling - b4\_1\_merge\_cutoff\_wrapper\_par** 1 Million fixed input size with threads 1-64

```
[55]: def get_b4_1_merge_cutoff_wrapper_scaling_algo(compiler_location:
    ↪str, compiler_name:str) -> pd.DataFrame:
        ## b4_1_merge_cutoff_wrapper_par
        df = extraction_pandas_frame_algo_threaded(root_dir + f'/'
    ↪{compiler_location}/THREADS',
                                                    'b4_1_merge_cutoff_wrapper_par',
                                                    [1,2,4,8,16,32,64],
                                                    COMP=compiler_name
        )

        ## calc strong scaling
        return calc_speedup_based_par(df, f"{compiler_name}")

instances = [
    ('GCC_TBB', 'GCC(TBB)'),
    ('NVHPC_Multicore', 'NVC(OMP)')
]
```

```

# collect data for instances
data = [get_b4_1_merge_cutoff_wrapper_scaling_algo(*x) for x in instances]

b4_1_merge_cutoff_wrapper_strong_scaling_merged = pd.merge(*data, on='threads')

print(b4_1_merge_cutoff_wrapper_strong_scaling_merged)

# plot strong scaling
ax = b4_1_merge_cutoff_wrapper_strong_scaling_merged.plot(kind='bar',
                                                         x='threads',
                                                         align='center',
                                                         color=[GCC_TBB_COLOR, NVC_OMP_COLOR]
                                                         )

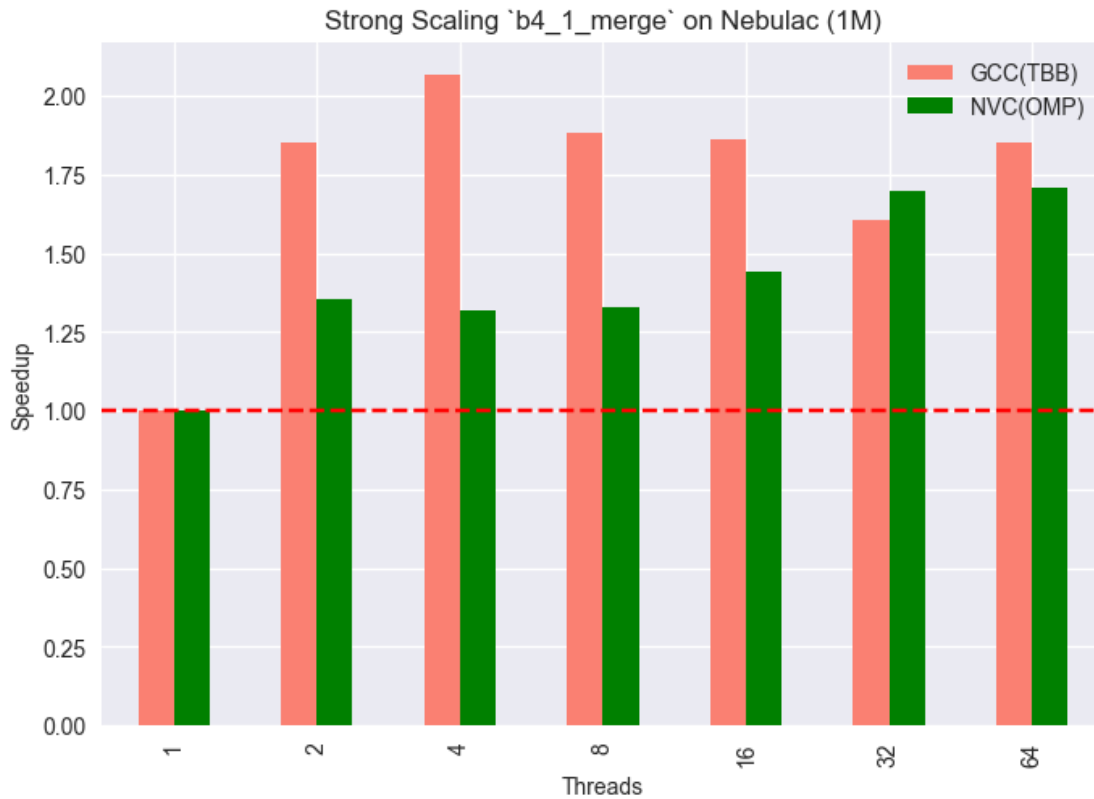
# adding horizontal line where there is speedup
ax.axhline(y=1, color='r', linestyle='--')

plt.ylabel('Speedup')
plt.xlabel('Threads')
plt.title('Strong Scaling `b4_1_merge` on Nebulac (1M)')

plot('Strong Scaling `b4_1_merge` on Nebulac (1M)')

```

	threads	GCC(TBB)	NVC(OMP)
0	1	1.000000	1.000000
1	2	1.853313	1.354363
2	4	2.067409	1.320676
3	8	1.883675	1.330278
4	16	1.863775	1.440116
5	32	1.607514	1.699778
6	64	1.851790	1.709192



```
[56]: # efficiency b4_4_set_difference_cutoff_intersected_par
b4_1_merge_cutoff_wrapper_efficiency = ☐
↳ b4_1_merge_cutoff_wrapper_strong_scaling_merged.copy()

instances = [
    ('GCC_TBB', 'GCC(TBB)'),
    ('NVHPC_Multicore', 'NVC(OMP)')
]

for _, compiler_name in instances:
    b4_1_merge_cutoff_wrapper_efficiency[f'{compiler_name}'] = ☐
    ↳ b4_1_merge_cutoff_wrapper_efficiency[f'{compiler_name}'] / ☐
    ↳ b4_1_merge_cutoff_wrapper_efficiency['threads']

# plot efficiency
ax = b4_1_merge_cutoff_wrapper_efficiency.plot(x='threads',
```

```

        color=[GCC_TBB_COLOR,NVC_OMP_COLOR]
    )

    # adding horizontal line where there is speedup
    ax.axhline(y=1, color='r', linestyle='--')

    plt.ylim(0,1.05)

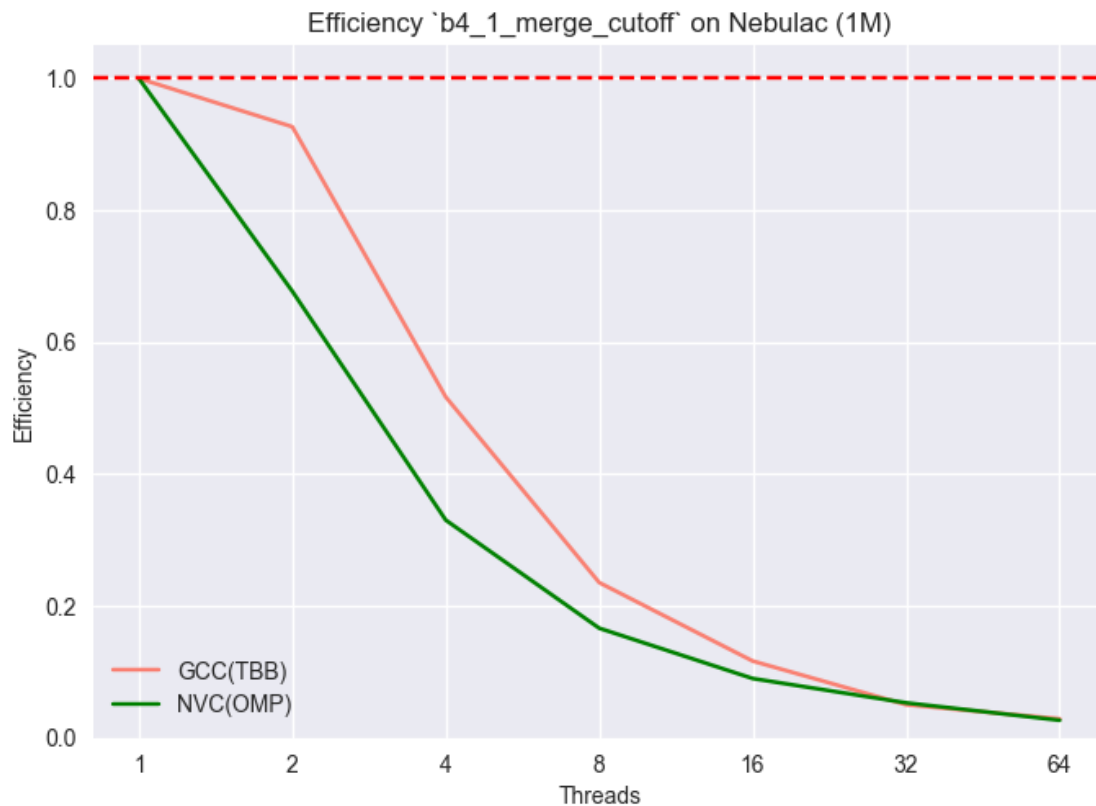
    plt.xscale('log', base=2)
    current_values = plt.gca().get_xticks()
    plt.gca().set_xticklabels(['{:,.0f}'.format(x) for x in current_values])

    plt.ylabel('Efficiency')
    plt.xlabel('Threads')
    plt.title('Efficiency `b4_1_merge_cutoff` on Nebulac (1M)')

    plot('Efficiency `b4_1_merge_cutoff` on Nebulac (1M)')

```

/var/folders/42/fk0jfyd1dd1ztdldncqc1cw0000gn/T/ipykernel\_33509/2248215454.py:2  
9: UserWarning: FixedFormatter should only be used together with FixedLocator  
plt.gca().set\_xticklabels(['{:,.0f}'.format(x) for x in current\_values])



**Strong Scaling - b4\_2\_stable\_sort\_cutoff\_already\_sorted\_par** 1 Million fixed input size with threads 1-64

```
[57]: def get_b4_2_stable_sort_cutoff_already_sorted_scaling_algo(compiler_location:
      ↪str, compiler_name:str) -> pd.DataFrame:
          ## b4_2_stable_sort_cutoff_already_sorted_par
          df = extraction_pandas_frame_algo_threaded(root_dir + f'/'
          ↪{compiler_location}/THREADS',
          ↪
          ↪'b4_2_stable_sort_cutoff_already_sorted_par',
          ↪
          ↪[1,2,4,8,16,32,64],
          ↪COMP=compiler_name
          ↪)

          ## calc strong scaling
          return calc_speedup_based_par(df, f"{compiler_name}")

instances = [
    ('GCC_TBB', 'GCC(TBB)'),
    ('NVHPC_Multicore', 'NVC(OMP)')
]

# collect data for instances
data = [get_b4_2_stable_sort_cutoff_already_sorted_scaling_algo(*x) for x in
      ↪instances]

b4_2_stable_sort_cutoff_already_sorted_strong_scaling_merged = pd.merge(*data,
      ↪on='threads')

b4_2_stable_sort_cutoff_already_sorted_strong_scaling_merged
```

```
[57]:
```

	threads	GCC(TBB)	NVC(OMP)
0	1	1.000000	1.000000
1	2	1.957849	1.663389
2	4	3.857910	2.264551
3	8	6.665196	3.109137
4	16	11.034630	3.400085
5	32	15.435886	2.811039
6	64	16.482198	2.533306

```
[58]: # efficiency b4_2_stable_sort_cutoff_already_sorted_par
b4_2_stable_sort_cutoff_already_sorted_efficiency =
      ↪b4_2_stable_sort_cutoff_already_sorted_strong_scaling_merged.copy()

instances = [
```



```

    ('GCC_TBB','GCC(TBB)'),
    ('NVHPC_Multicore','NVC(OMP)')
]

for _,compiler_name in instances:
    b4_2_stable_sort_cutoff_already_sorted_efficiency[f'{compiler_name}'] =
↳b4_2_stable_sort_cutoff_already_sorted_efficiency[f'{compiler_name}'] /
↳b4_2_stable_sort_cutoff_already_sorted_efficiency['threads']

b4_2_stable_sort_cutoff_already_sorted_efficiency

```

```

[58]:
      threads  GCC(TBB)  NVC(OMP)
0         1    1.000000    1.000000
1         2    0.978924    0.831695
2         4    0.964478    0.566138
3         8    0.833150    0.388642
4        16    0.689664    0.212505
5        32    0.482371    0.087845
6        64    0.257534    0.039583

```

**Strong Scaling - b4\_2\_stable\_sort\_cutoff\_decrement\_sorted\_par** 1 Million fixed input size with threads 1-64

```

[59]: def get_b4_2_stable_sort_cutoff_decrement_sorted_scaling_algo(compiler_location:
↳str,compiler_name:str) -> pd.DataFrame:
    ## b4_2_stable_sort_cutoff_decrement_sorted_par
    df = extraction_pandas_frame_algo_threaded(root_dir + f'/'
↳{compiler_location}/THREADS',
    ↳
↳'b4_2_stable_sort_cutoff_decrement_sorted_par',
    [1,2,4,8,16,32,64],
    COMP=compiler_name
    )

    ## calc strong scaling
    return calc_speedup_based_par(df,f"{compiler_name}")

instances = [
    ('GCC_TBB','GCC(TBB)'),
    ('NVHPC_Multicore','NVC(OMP)')
]

# collect data for instances

```

```

data = [get_b4_2_stable_sort_cutoff_decrement_sorted_scaling_algo(*x) for x in
↳instances]

b4_2_stable_sort_cutoff_decrement_sorted_strong_scaling_merged = pd.
↳merge(*data, on='threads')

b4_2_stable_sort_cutoff_decrement_sorted_strong_scaling_merged

```

```

[59]:
  threads  GCC(TBB)  NVC(OMP)
0         1    1.000000    1.000000
1         2    1.925032    1.723710
2         4    3.721133    2.249129
3         8    6.610669    2.742405
4        16   11.008026    3.019839
5        32   15.217989    2.817539
6        64   13.512579    2.421528

```

```

[60]: # efficiency b4_2_stable_sort_cutoff_decrement_sorted_par
b4_2_stable_sort_cutoff_decrement_sorted_efficiency =
↳b4_2_stable_sort_cutoff_decrement_sorted_strong_scaling_merged.copy()

instances = [
    ('GCC_TBB', 'GCC(TBB)'),
    ('NVHPC_Multicore', 'NVC(OMP)')
]

for _, compiler_name in instances:
    b4_2_stable_sort_cutoff_decrement_sorted_efficiency[f'{compiler_name}'] =
↳b4_2_stable_sort_cutoff_decrement_sorted_efficiency[f'{compiler_name}'] /
↳b4_2_stable_sort_cutoff_decrement_sorted_efficiency['threads']

b4_2_stable_sort_cutoff_decrement_sorted_efficiency

```

```

[60]:
  threads  GCC(TBB)  NVC(OMP)
0         1    1.000000    1.000000
1         2    0.962516    0.861855
2         4    0.930283    0.562282
3         8    0.826334    0.342801
4        16    0.688002    0.188740
5        32    0.475562    0.088048
6        64    0.211134    0.037836

```

**Strong Scaling - b4\_2\_stable\_sort\_cutoff\_not\_sorted\_par** 1 Million fixed input size with threads 1-64

```
[61]: def get_b4_2_stable_sort_cutoff_not_sorted_scaling_algo(compiler_location:
      ↪str, compiler_name:str) -> pd.DataFrame:
      ↪    ## b4_2_stable_sort_cutoff_not_sorted_par
      ↪    df = extraction_pandas_frame_algo_threaded(root_dir + f'/'
      ↪    {compiler_location}/THREADS',
      ↪
      ↪    'b4_2_stable_sort_cutoff_not_sorted_par',
      ↪
      ↪    [1,2,4,8,16,32,64],
      ↪    COMP=compiler_name
      ↪    )

      ↪    ## calc strong scaling
      ↪    return calc_speedup_based_par(df, f"{compiler_name}")

instances = [
    ('GCC_TBB', 'GCC(TBB)'),
    ('NVHPC_Multicore', 'NVC(OMP)')
]

# collect data for instances
data = [get_b4_2_stable_sort_cutoff_not_sorted_scaling_algo(*x) for x in
      ↪instances]

b4_2_stable_sort_cutoff_not_sorted_strong_scaling_merged = pd.merge(*data,
      ↪on='threads')

b4_2_stable_sort_cutoff_not_sorted_strong_scaling_merged
```

```
[61]:
```

	threads	GCC(TBB)	NVC(OMP)
0	1	1.000000	1.000000
1	2	1.962643	1.198911
2	4	3.831584	1.266255
3	8	7.071408	1.286807
4	16	11.656235	1.285389
5	32	15.986008	1.181633
6	64	14.573870	1.079698

```
[62]: # efficiency b4_2_stable_sort_cutoff_not_sorted_par
b4_2_stable_sort_cutoff_not_sorted_efficiency =
      ↪b4_2_stable_sort_cutoff_not_sorted_strong_scaling_merged.copy()

instances = [
    ('GCC_TBB', 'GCC(TBB)'),
    ('NVHPC_Multicore', 'NVC(OMP)')
]
```

```

for _, compiler_name in instances:
    b4_2_stable_sort_cutoff_not_sorted_efficiency[f'{compiler_name}'] = 
    ↪ b4_2_stable_sort_cutoff_not_sorted_efficiency[f'{compiler_name}'] / 

```

```

[62]:
      threads  GCC(TBB)  NVC(OMP)
0         1  1.000000  1.000000
1         2  0.981322  0.599456
2         4  0.957896  0.316564
3         8  0.883926  0.160851
4        16  0.728515  0.080337
5        32  0.499563  0.036926
6        64  0.227717  0.016870

```

```

[63]: ## stable sort strong scaling graphs

fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(16, 6))

# already sorted
ax_1 = b4_2_stable_sort_cutoff_already_sorted_strong_scaling_merged.
    ↪ plot(kind='bar', x='threads', align='center', color=[GCC_TBB_COLOR, NVC_OMP_COLOR], ax=axes[0])

ax_1.set_ylabel('Speedup')
ax_1.set_xlabel('Input size')
ax_1.set_title('already sorted')
ax_1.axhline(y=1, color='r', linestyle='--')

# decrement sorted
ax_2 = b4_2_stable_sort_cutoff_decrement_sorted_strong_scaling_merged.
    ↪ plot(kind='bar', x='threads', align='center', color=[GCC_TBB_COLOR, NVC_OMP_COLOR], ax=axes[1])

ax_2.set_ylabel('Speedup')
ax_2.set_xlabel('Input size')
ax_2.set_title('decremented sorted')
ax_2.axhline(y=1, color='r', linestyle='--')

# not sorted

```

```

ax_3 = b4_2_stable_sort_cutoff_not_sorted_strong_scaling_merged.
    plot(kind='bar',x='threads',align='center',color=[GCC_TBB_COLOR,NVC_OMP_COLOR],ax=axes[2])

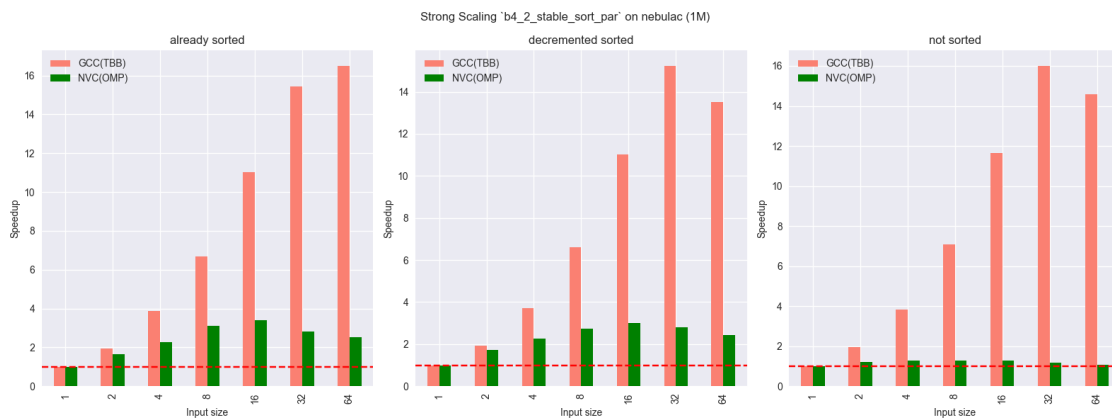
ax_3.set_ylabel('Speedup')
ax_3.set_xlabel('Input size')
ax_3.set_title('not sorted')
ax_3.axhline(y=1,color='r', linestyle='--')

fig.suptitle("Strong Scaling `b4_2_stable_sort_par` on nebulac (1M)")

fig.tight_layout()

plot("Strong Scaling `b4_2_stable_sort_par` on nebulac (1M)")

```



```

[64]: ### efficiency stable sort

fig, axes = plt.subplots(nrows=1, ncols=3,figsize=(16, 6))

# already sorted
ax_1 = b4_2_stable_sort_cutoff_already_sorted_efficiency.
    plot(x='threads',color=[GCC_TBB_COLOR,NVC_OMP_COLOR],ax=axes[0])

ax_1.set_ylabel('Efficiency')
ax_1.set_xlabel('Input size')
ax_1.set_title('already sorted')
ax_1.axhline(y=1,color='r', linestyle='--')

ax_1.set_ylim(0,1.05)

ax_1.set_xscale('log', base=2)
current_values = ax_1.get_xticks()

```

```

ax_1.set_xticklabels(['{:,.0f}'.format(x) for x in current_values])

# decrement sorted
ax_2 = b4_2_stable_sort_cutoff_decrement_sorted_efficiency.
    plot(x='threads',color=[GCC_TBB_COLOR,NVC_OMP_COLOR],ax=axes[1])

ax_2.set_ylabel('Efficiency')
ax_2.set_xlabel('Input size')
ax_2.set_title('decremented sorted')
ax_2.axhline(y=1,color='r', linestyle='--')

ax_2.set_ylim(0,1.05)

ax_2.set_xscale('log', base=2)
current_values = ax_2.get_xticks()
ax_2.set_xticklabels(['{:,.0f}'.format(x) for x in current_values])

# not sorted
ax_3 = b4_2_stable_sort_cutoff_not_sorted_efficiency.
    plot(x='threads',color=[GCC_TBB_COLOR,NVC_OMP_COLOR],ax=axes[2])

ax_3.set_ylabel('Efficiency')
ax_3.set_xlabel('Input size')
ax_3.set_title('not sorted')
ax_3.axhline(y=1,color='r', linestyle='--')

ax_3.set_ylim(0,1.05)

ax_3.set_xscale('log', base=2)
current_values = ax_3.get_xticks()
ax_3.set_xticklabels(['{:,.0f}'.format(x) for x in current_values])

fig.suptitle("Efficiency `b4_2_stable_sort_par` on nebulac (1M)")

fig.tight_layout()

plot("Efficiency `b4_2_stable_sort_par` on nebulac (1M)")

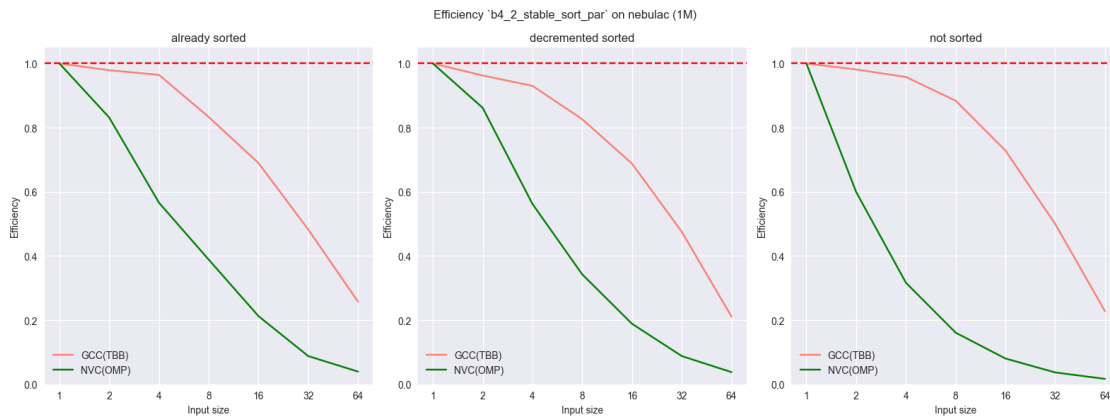
```

```

/var/folders/42/fk0jfryd1dd1ztdldncqc1cw0000gn/T/ipykernel_33509/582381719.py:18
: UserWarning: FixedFormatter should only be used together with FixedLocator
    ax_1.set_xticklabels(['{:,.0f}'.format(x) for x in current_values])
/var/folders/42/fk0jfryd1dd1ztdldncqc1cw0000gn/T/ipykernel_33509/582381719.py:33
: UserWarning: FixedFormatter should only be used together with FixedLocator
    ax_2.set_xticklabels(['{:,.0f}'.format(x) for x in current_values])

```

```
/var/folders/42/fk0jfryd1dd1ztdldncqc1cw0000gn/T/ipykernel_33509/582381719.py:48
: UserWarning: FixedFormatter should only be used together with FixedLocator
ax_3.set_xticklabels(['{:,.0f}'.format(x) for x in current_values])
```



**Strong Scaling** - b4\_3\_set\_union\_cutoff\_one\_empty\_par 1 Million fixed input size with threads 1-64

```
[65]: def get_b4_3_set_union_cutoff_one_empty_scaling_algo(compiler_location:
    ↪str, compiler_name:str) -> pd.DataFrame:
        ## b4_3_set_union_cutoff_one_empty_par
        df = extraction_pandas_frame_algo_threaded(root_dir + f'/'
    ↪{compiler_location}/THREADS',
        ↪'b4_3_set_union_cutoff_one_empty_par',
        ↪[1,2,4,8,16,32,64],
        ↪COMP=compiler_name
        ↪)

        ## calc strong scaling
        return calc_speedup_based_par(df, f"{compiler_name}")

instances = [
    ('GCC_TBB', 'GCC(TBB)'),
    ('NVHPC_Multicore', 'NVC(OMP)')
]

# collect data for instances
data = [get_b4_3_set_union_cutoff_one_empty_scaling_algo(*x) for x in instances]

b4_3_set_union_cutoff_one_empty_strong_scaling_merged = pd.merge(*data,
    ↪on='threads')
```

```
b4_3_set_union_cutoff_one_empty_strong_scaling_merged
```

```
[65]:
```

	threads	GCC(TBB)	NVC(OMP)
0	1	1.000000	1.000000
1	2	1.572384	1.912834
2	4	1.359916	3.271521
3	8	0.840827	4.702175
4	16	0.814790	4.711323
5	32	0.875187	4.333537
6	64	0.852408	5.756391

```
[66]: # efficiency b4_3_set_union_cutoff_one_empty_par
b4_3_set_union_cutoff_one_empty_efficiency =
↳ b4_3_set_union_cutoff_one_empty_strong_scaling_merged.copy()

instances = [
    ('GCC_TBB', 'GCC(TBB)'),
    ('NVHPC_Multicore', 'NVC(OMP)')
]

for _, compiler_name in instances:
    b4_3_set_union_cutoff_one_empty_efficiency[f'{compiler_name}'] =
↳ b4_3_set_union_cutoff_one_empty_efficiency[f'{compiler_name}'] /
↳ b4_3_set_union_cutoff_one_empty_efficiency['threads']

b4_3_set_union_cutoff_one_empty_efficiency
```

```
[66]:
```

	threads	GCC(TBB)	NVC(OMP)
0	1	1.000000	1.000000
1	2	0.786192	0.956417
2	4	0.339979	0.817880
3	8	0.105103	0.587772
4	16	0.050924	0.294458
5	32	0.027350	0.135423
6	64	0.013319	0.089944

**Strong Scaling - b4\_3\_set\_union\_cutoff\_one\_wholly\_greater\_par** 1 Million fixed input size with threads 1-64

```
[67]: def get_b4_3_set_union_cutoff_one_wholly_greater_scaling_algo(compiler_location:
↳ str, compiler_name: str) -> pd.DataFrame:
    ## b4_3_set_union_cutoff_one_wholly_greater_par
```



```

    df = extraction_pandas_frame_algo_threaded(root_dir + f'/{
↳{compiler_location}/THREADS',

                                □
↳'b4_3_set_union_cutoff_one_wholly_greater_par',
                                [1,2,4,8,16,32,64],
                                COMP=compiler_name
                                )

    ## calc strong scaling
    return calc_speedup_based_par(df,f"{compiler_name}")

instances = [
    ('GCC_TBB', 'GCC(TBB)'),
    ('NVHPC_Multicore', 'NVC(OMP)')
]

# collect data for instances
data = [get_b4_3_set_union_cutoff_one_wholly_greater_scaling_algo(*x) for x in
↳instances]

b4_3_set_union_cutoff_one_wholly_greater_strong_scaling_merged = pd.
↳merge(*data, on='threads')

b4_3_set_union_cutoff_one_wholly_greater_strong_scaling_merged

```

```

[67]:
   threads  GCC(TBB)  NVC(OMP)
0         1  1.000000  1.000000
1         2  1.695344  1.592076
2         4  1.869541  2.099766
3         8  1.399180  1.916400
4        16  1.391153  2.082648
5        32  1.419301  2.049050
6        64  1.367540  2.517495

```

```

[68]: # efficiency b4_3_set_union_cutoff_one_wholly_greater_par
b4_3_set_union_cutoff_one_wholly_greater_efficiency =
↳b4_3_set_union_cutoff_one_wholly_greater_strong_scaling_merged.copy()

instances = [
    ('GCC_TBB', 'GCC(TBB)'),
    ('NVHPC_Multicore', 'NVC(OMP)')
]

for _, compiler_name in instances:

```

```

    b4_3_set_union_cutoff_one_wholly_greater_efficiency[f'{compiler_name}'] =
↳ b4_3_set_union_cutoff_one_wholly_greater_efficiency[f'{compiler_name}'] /
↳ b4_3_set_union_cutoff_one_wholly_greater_efficiency['threads']

```

b4\_3\_set\_union\_cutoff\_one\_wholly\_greater\_efficiency

```

[68]:
  threads  GCC(TBB)  NVC(OMP)
0         1  1.000000  1.000000
1         2  0.847672  0.796038
2         4  0.467385  0.524941
3         8  0.174898  0.239550
4        16  0.086947  0.130166
5        32  0.044353  0.064033
6        64  0.021368  0.039336

```

**Strong Scaling** - b4\_3\_set\_union\_cutoff\_front\_overhang\_par 1 Million fixed input size with threads 1-64

```

[69]: def get_b4_3_set_union_cutoff_front_overhang_scaling_algo(compiler_location:
↳ str, compiler_name: str) -> pd.DataFrame:
    ## b4_3_set_union_cutoff_front_overhang_par
    df = extraction_pandas_frame_algo_threaded(root_dir + f'/
↳ {compiler_location}/THREADS',
    ## calc strong scaling
    return calc_speedup_based_par(df, f'{compiler_name}')

instances = [
    ('GCC_TBB', 'GCC(TBB)'),
    ('NVHPC_Multicore', 'NVC(OMP)')
]

# collect data for instances
data = [get_b4_3_set_union_cutoff_front_overhang_scaling_algo(*x) for x in
↳ instances]

b4_3_set_union_cutoff_front_overhang_strong_scaling_merged = pd.merge(*data,
↳ on='threads')

```

```
b4_3_set_union_cutoff_front_overhang_strong_scaling_merged
```

```
[69]:
```

	threads	GCC(TBB)	NVC(OMP)
0	1	1.000000	1.000000
1	2	1.376355	1.271216
2	4	1.491628	1.628908
3	8	1.443944	1.231676
4	16	1.614918	1.273868
5	32	1.518863	1.523826
6	64	1.644744	1.437845

```
[70]: # efficiency b4_3_set_union_cutoff_front_overhang_par
b4_3_set_union_cutoff_front_overhang_efficiency =
    ↪ b4_3_set_union_cutoff_front_overhang_strong_scaling_merged.copy()

instances = [
    ('GCC_TBB', 'GCC(TBB)'),
    ('NVHPC_Multicore', 'NVC(OMP)')
]

for _, compiler_name in instances:
    b4_3_set_union_cutoff_front_overhang_efficiency[f'{compiler_name}'] =
    ↪ b4_3_set_union_cutoff_front_overhang_efficiency[f'{compiler_name}'] /
    ↪ b4_3_set_union_cutoff_front_overhang_efficiency['threads']

b4_3_set_union_cutoff_front_overhang_efficiency
```

```
[70]:
```

	threads	GCC(TBB)	NVC(OMP)
0	1	1.000000	1.000000
1	2	0.688178	0.635608
2	4	0.372907	0.407227
3	8	0.180493	0.153960
4	16	0.100932	0.079617
5	32	0.047464	0.047620
6	64	0.025699	0.022466

```
[71]: # all set_union graphs

fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(16, 6))

# already sorted
ax_1 = b4_3_set_union_cutoff_one_empty_strong_scaling_merged.
    ↪ plot(kind='bar', x='threads', align='center', color=[GCC_TBB_COLOR, NVC_OMP_COLOR], ax=axes[0])
```

```

ax_1.set_ylabel('Speedup')
ax_1.set_xlabel('Input size')
ax_1.set_title('One empty')
ax_1.axhline(y=1, color='r',linestyle='--')
# decrement sorted
ax_2 = b4_3_set_union_cutoff_one_wholly_greater_strong_scaling_merged.
    ↪plot(kind='bar',x='threads',align='center',color=[GCC_TBB_COLOR,NVC_OMP_COLOR],ax=axes[1])

ax_2.set_ylabel('Speedup')
ax_2.set_xlabel('Input size')
ax_2.set_title('One wholly greater')
ax_2.axhline(y=1, color='r',linestyle='--')

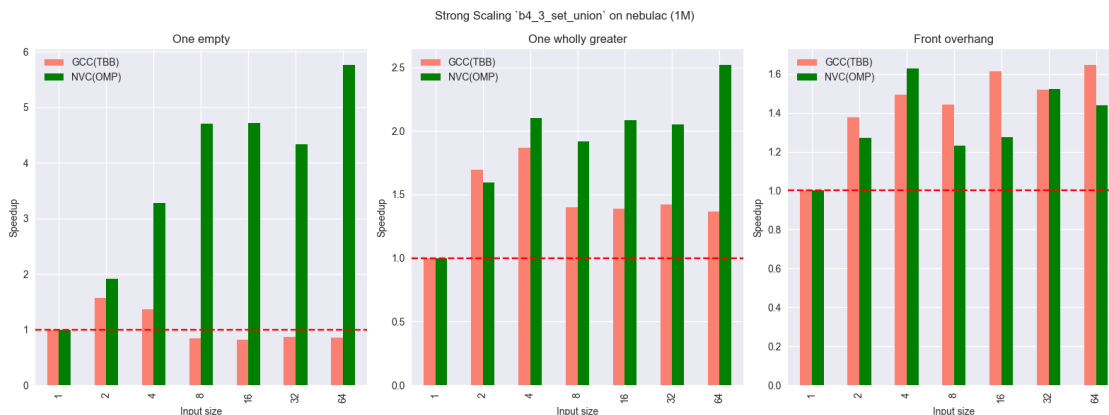
# not sorted
ax_3 = b4_3_set_union_cutoff_front_overhang_strong_scaling_merged.
    ↪plot(kind='bar',x='threads',align='center',color=[GCC_TBB_COLOR,NVC_OMP_COLOR],ax=axes[2])

ax_3.set_ylabel('Speedup')
ax_3.set_xlabel('Input size')
ax_3.set_title('Front overhang')
ax_3.axhline(y=1, color='r',linestyle='--')
fig.suptitle("Strong Scaling `b4_3_set_union` on nebulac (1M)")

fig.tight_layout()

plot("Strong Scaling `b4_3_set_union` on nebulac (1M)")

```



[72]: # efficiency set\_union graphs

```
fig, axes = plt.subplots(nrows=1, ncols=3,figsize=(16, 6))
```

```

# already sorted
ax_1 = b4_3_set_union_cutoff_one_empty_efficiency.
    plot(x='threads',color=[GCC_TBB_COLOR,NVC_OMP_COLOR],ax=axes[0])

ax_1.set_ylabel('Efficiency')
ax_1.set_xlabel('Input size')
ax_1.set_title('One empty')
ax_1.axhline(y=1, color='r',linestyle='--')
ax_1.set_ylim(0,1.05)

ax_1.set_xscale('log', base=2)
current_values = ax_1.get_xticks()
ax_1.set_xticklabels(['{:,.0f}'.format(x) for x in current_values])

# decrement sorted
ax_2 = b4_3_set_union_cutoff_one_wholly_greater_efficiency.
    plot(x='threads',color=[GCC_TBB_COLOR,NVC_OMP_COLOR],ax=axes[1])

ax_2.set_ylabel('Efficiency')
ax_2.set_xlabel('Input size')
ax_2.set_title('One wholly greater')
ax_2.axhline(y=1, color='r',linestyle='--')
ax_2.set_ylim(0,1.05)

ax_2.set_xscale('log', base=2)
current_values = ax_2.get_xticks()
ax_2.set_xticklabels(['{:,.0f}'.format(x) for x in current_values])

# not sorted
ax_3 = b4_3_set_union_cutoff_front_overhang_efficiency.
    plot(x='threads',color=[GCC_TBB_COLOR,NVC_OMP_COLOR],ax=axes[2])

ax_3.set_ylabel('Efficiency')
ax_3.set_xlabel('Input size')
ax_3.set_title('Front overhang')
ax_3.axhline(y=1, color='r',linestyle='--')
ax_3.set_ylim(0,1.05)

ax_3.set_xscale('log', base=2)
current_values = ax_3.get_xticks()
ax_3.set_xticklabels(['{:,.0f}'.format(x) for x in current_values])

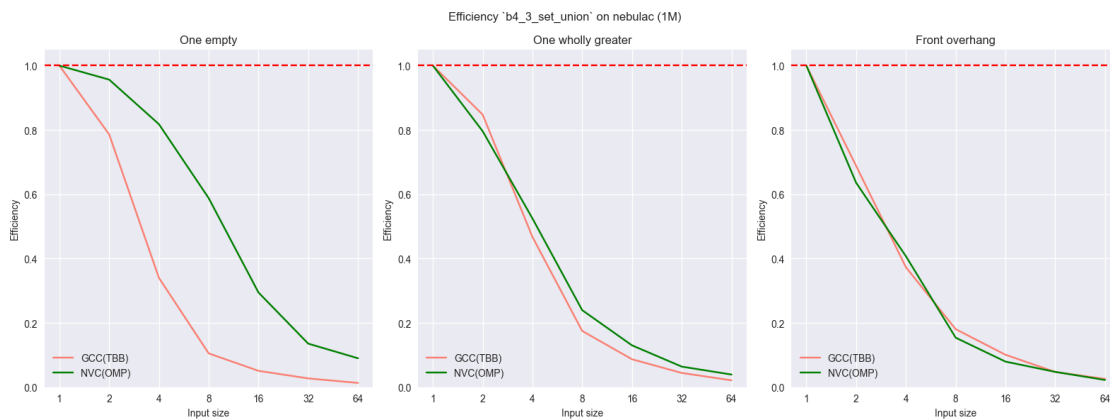
```

```
fig.suptitle("Efficiency `b4_3_set_union` on nebulac (1M)")
```

```
fig.tight_layout()
```

```
plot("Efficiency `b4_3_set_union` on nebulac (1M)")
```

```
/var/folders/42/fk0jfryd1dd1ztdldncqc1cw0000gn/T/ipykernel_33509/3955302251.py:1
8: UserWarning: FixedFormatter should only be used together with FixedLocator
  ax_1.set_xticklabels(['{:,.0f}'.format(x) for x in current_values])
/var/folders/42/fk0jfryd1dd1ztdldncqc1cw0000gn/T/ipykernel_33509/3955302251.py:3
2: UserWarning: FixedFormatter should only be used together with FixedLocator
  ax_2.set_xticklabels(['{:,.0f}'.format(x) for x in current_values])
/var/folders/42/fk0jfryd1dd1ztdldncqc1cw0000gn/T/ipykernel_33509/3955302251.py:4
6: UserWarning: FixedFormatter should only be used together with FixedLocator
  ax_3.set_xticklabels(['{:,.0f}'.format(x) for x in current_values])
```



**Strong Scaling - b4\_4\_set\_difference\_cutoff\_left\_empty\_par** 1 Million fixed input size with threads 1-64

```
[73]: def get_b4_4_set_difference_cutoff_left_empty_scaling_algo(compiler_location:
      ↪str, compiler_name: str) -> pd.DataFrame:
      ## b4_4_set_difference_cutoff_left_empty_par
      df = extraction_pandas_frame_algo_threaded(root_dir + f'/
      ↪{compiler_location}/THREADS',
      ↪'b4_4_set_difference_cutoff_left_empty_par',
      ↪[1,2,4,8,16,32,64],
      ↪COMP=compiler_name
      ↪)

      ## calc strong scaling
      return calc_speedup_based_par(df, f"{compiler_name}")
```

```

instances = [
    ('GCC_TBB', 'GCC(TBB)'),
    ('NVHPC_Multicore', 'NVC(OMP)')
]

# collect data for instances
data = [get_b4_4_set_difference_cutoff_left_empty_scaling_algo(*x) for x in instances]

b4_4_set_difference_cutoff_left_empty_strong_scaling_merged = pd.merge(*data,
    on='threads')

b4_4_set_difference_cutoff_left_empty_strong_scaling_merged

```

```

[73]:
   threads  GCC(TBB)  NVC(OMP)
0         1  1.000000  1.000000
1         2  0.993309  1.005229
2         4  1.003757  0.988039
3         8  0.996908  0.991691
4        16  0.996140  0.992383
5        32  1.005280  0.987467
6        64  0.996437  0.989760

```

```

[74]: # efficiency b4_4_set_difference_cutoff_left_empty_par
b4_4_set_difference_cutoff_left_empty_efficiency =
    b4_4_set_difference_cutoff_left_empty_strong_scaling_merged.copy()

instances = [
    ('GCC_TBB', 'GCC(TBB)'),
    ('NVHPC_Multicore', 'NVC(OMP)')
]

for _, compiler_name in instances:
    b4_4_set_difference_cutoff_left_empty_efficiency[f'{compiler_name}'] =
        b4_4_set_difference_cutoff_left_empty_efficiency[f'{compiler_name}'] /
        b4_4_set_difference_cutoff_left_empty_efficiency['threads']

b4_4_set_difference_cutoff_left_empty_efficiency

```

```

[74]:
   threads  GCC(TBB)  NVC(OMP)
0         1  1.000000  1.000000
1         2  0.496655  0.502615

```

2	4	0.250939	0.247010
3	8	0.124613	0.123961
4	16	0.062259	0.062024
5	32	0.031415	0.030858
6	64	0.015569	0.015465

**Strong Scaling - b4\_4\_set\_difference\_cutoff\_right\_empty\_par** 1 Million fixed input size with threads 1-64

```
[75]: def get_b4_4_set_difference_cutoff_right_empty_scaling_algo(compiler_location:
↳str, compiler_name:str) -> pd.DataFrame:
    ## b4_4_set_difference_cutoff_right_empty_par
    df = extraction_pandas_frame_algo_threaded(root_dir + f'/
↳{compiler_location}/THREADS',
    'b4_4_set_difference_cutoff_right_empty_par',
    [1,2,4,8,16,32,64],
    COMP=compiler_name
    )

    ## calc strong scaling
    return calc_speedup_based_par(df, f"{compiler_name}")

instances = [
    ('GCC_TBB', 'GCC(TBB)'),
    ('NVHPC_Multicore', 'NVC(OMP)')
]

# collect data for instances
data = [get_b4_4_set_difference_cutoff_right_empty_scaling_algo(*x) for x in
↳instances]

b4_4_set_difference_cutoff_right_empty_strong_scaling_merged = pd.merge(*data,
↳on='threads')

b4_4_set_difference_cutoff_right_empty_strong_scaling_merged
```

```
[75]: threads  GCC(TBB)  NVC(OMP)
0         1  1.000000  1.000000
1         2  1.466047  1.917873
2         4  1.236516  3.358576
3         8  1.194825  4.920266
4        16  0.916489  5.196948
5        32  0.941898  5.170932
6        64  0.856945  5.610548
```



```
[76]: # efficiency b4_4_set_difference_cutoff_right_empty_par
b4_4_set_difference_cutoff_right_empty_efficiency = □
↳ b4_4_set_difference_cutoff_right_empty_strong_scaling_merged.copy()

instances = [
    ('GCC_TBB', 'GCC(TBB)'),
    ('NVHPC_Multicore', 'NVC(OMP)')
]

for _, compiler_name in instances:
    b4_4_set_difference_cutoff_right_empty_efficiency[f'{compiler_name}'] = □
↳ b4_4_set_difference_cutoff_right_empty_efficiency[f'{compiler_name}'] / □
↳ b4_4_set_difference_cutoff_right_empty_efficiency['threads']

b4_4_set_difference_cutoff_right_empty_efficiency
```

```
[76]:      threads  GCC(TBB)  NVC(OMP)
0         1  1.000000  1.000000
1         2  0.733023  0.958936
2         4  0.309129  0.839644
3         8  0.149353  0.615033
4        16  0.057281  0.324809
5        32  0.029434  0.161592
6        64  0.013390  0.087665
```

Strong Scaling - b4\_4\_set\_difference\_cutoff\_wholly\_greater\_par 1 Million fixed input size with threads 1-64

```
[77]: def □
↳ get_b4_4_set_difference_cutoff_wholly_greater_scaling_algo(compiler_location:
↳ str, compiler_name: str) -> pd.DataFrame:
    ## b4_4_set_difference_cutoff_wholly_greater_par
    df = extraction_pandas_frame_algo_threaded(root_dir + f'/'
↳ {compiler_location}/THREADS',
                                □
↳ 'b4_4_set_difference_cutoff_wholly_greater_par',
                                [1,2,4,8,16,32,64],
                                COMP=compiler_name
                                )

    ## calc strong scaling
    return calc_speedup_based_par(df, f'{compiler_name}')
```

```

instances = [
    ('GCC_TBB', 'GCC(TBB)'),
    ('NVHPC_Multicore', 'NVC(OMP)')
]

# collect data for instances
data = [get_b4_4_set_difference_cutoff_wholly_greater_scaling_algo(*x) for x in instances]

b4_4_set_difference_cutoff_wholly_greater_strong_scaling_merged = pd.
    merge(*data, on='threads')

b4_4_set_difference_cutoff_wholly_greater_strong_scaling_merged

```

```

[77]:      threads  GCC(TBB)  NVC(OMP)
0         1  1.000000  1.000000
1         2  1.529318  1.004201
2         4  0.910725  1.008998
3         8  0.882977  1.011117
4        16  0.982963  1.009471
5        32  0.764612  0.999316
6        64  0.847159  1.010497

```

```

[78]: # efficiency b4_4_set_difference_cutoff_wholly_greater_par
b4_4_set_difference_cutoff_wholly_greater_efficiency =
    b4_4_set_difference_cutoff_wholly_greater_strong_scaling_merged.copy()

instances = [
    ('GCC_TBB', 'GCC(TBB)'),
    ('NVHPC_Multicore', 'NVC(OMP)')
]

for _, compiler_name in instances:
    b4_4_set_difference_cutoff_wholly_greater_efficiency[f'{compiler_name}'] =
        b4_4_set_difference_cutoff_wholly_greater_efficiency[f'{compiler_name}'] /
        b4_4_set_difference_cutoff_wholly_greater_efficiency['threads']

b4_4_set_difference_cutoff_wholly_greater_efficiency

```

```

[78]:      threads  GCC(TBB)  NVC(OMP)
0         1  1.000000  1.000000
1         2  0.764659  0.502100
2         4  0.227681  0.252249
3         8  0.110372  0.126390

```

4	16	0.061435	0.063092
5	32	0.023894	0.031229
6	64	0.013237	0.015789

**Strong Scaling - b4\_4\_set\_difference\_cutoff\_intersected\_par** 1 Million fixed input size with threads 1-64

```
[79]: def get_b4_4_set_difference_cutoff_intersected_scaling_algo(compiler_location:
      ↪str, compiler_name:str) -> pd.DataFrame:
      ↪    ## b4_4_set_difference_cutoff_intersected_par
      ↪    df = extraction_pandas_frame_algo_threaded(root_dir + f'/
      ↪    {compiler_location}/THREADS',
      ↪
      ↪    ↪    'b4_4_set_difference_cutoff_intersected_par',
      ↪    ↪    [1,2,4,8,16,32,64],
      ↪    ↪    COMP=compiler_name
      ↪    ↪    )

      ↪    ## calc strong scaling
      ↪    return calc_speedup_based_par(df, f"{compiler_name}")

instances = [
    ('GCC_TBB', 'GCC(TBB)'),
    ('NVHPC_Multicore', 'NVC(OMP)')
]

# collect data for instances
data = [get_b4_4_set_difference_cutoff_intersected_scaling_algo(*x) for x in ↪
      ↪instances]

b4_4_set_difference_cutoff_intersected_strong_scaling_merged = pd.merge(*data, ↪
      ↪on='threads')

b4_4_set_difference_cutoff_intersected_strong_scaling_merged
```

```
[79]:   threads  GCC(TBB)  NVC(OMP)
0         1  1.000000  1.000000
1         2  1.950886  1.003025
2         4  3.270436  1.010849
3         8  5.403732  0.999742
4        16  7.770342  0.997566
5        32  8.776868  1.018830
6        64  8.856632  1.004002
```

[80]:

```

# efficiency b4_4_set_difference_cutoff_intersected_par
b4_4_set_difference_cutoff_intersected_efficiency = 
    ↪ b4_4_set_difference_cutoff_intersected_strong_scaling_merged.copy()

instances = [
    ('GCC_TBB', 'GCC(TBB)'),
    ('NVHPC_Multicore', 'NVC(OMP)')
]

for _, compiler_name in instances:
    b4_4_set_difference_cutoff_intersected_efficiency[f'{compiler_name}'] = 
    ↪ b4_4_set_difference_cutoff_intersected_efficiency[f'{compiler_name}'] / 
    ↪ b4_4_set_difference_cutoff_intersected_efficiency['threads']

b4_4_set_difference_cutoff_intersected_efficiency

```

```

[80]:      threads  GCC(TBB)  NVC(OMP)
0         1  1.000000  1.000000
1         2  0.975443  0.501513
2         4  0.817609  0.252712
3         8  0.675467  0.124968
4        16  0.485646  0.062348
5        32  0.274277  0.031838
6        64  0.138385  0.015688

```

```

[81]: # strong scaling graphs

fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(16, 9))

# left empty
ax_1 = b4_4_set_difference_cutoff_left_empty_strong_scaling_merged.
    ↪ plot(kind='bar', x='threads', align='center', color=[GCC_TBB_COLOR, NVC_OMP_COLOR], ax=axes[0,0])

ax_1.set_ylabel('Speedup')
ax_1.set_xlabel('Input size')
ax_1.set_title('Left empty')
ax_1.axhline(y=1, color='r', linestyle='--')
ax_1.set_ylim(0,9)

# right empty
ax_2 = b4_4_set_difference_cutoff_right_empty_strong_scaling_merged.
    ↪ plot(kind='bar', x='threads', align='center', color=[GCC_TBB_COLOR, NVC_OMP_COLOR], ax=axes[0,1])

```

```

ax_2.set_ylabel('Speedup')
ax_2.set_xlabel('Input size')
ax_2.set_title('Right empty')
ax_2.axhline(y=1, color='r',linestyle='--')
ax_2.set_ylim(0,9)

# wholly greater
ax_3 = b4_4_set_difference_cutoff_wholly_greater_strong_scaling_merged.
    ↪plot(kind='bar',x='threads',align='center',color=[GCC_TBB_COLOR,NVC_OMP_COLOR],ax=axes[1,0])

ax_3.set_ylabel('Speedup')
ax_3.set_xlabel('Input size')
ax_3.set_title('Wholly greater')
ax_3.axhline(y=1, color='r',linestyle='--')
ax_3.set_ylim(0,9)

# intersected
ax_4 = b4_4_set_difference_cutoff_intersected_strong_scaling_merged.
    ↪plot(kind='bar',x='threads',align='center',color=[GCC_TBB_COLOR,NVC_OMP_COLOR],ax=axes[1,1])

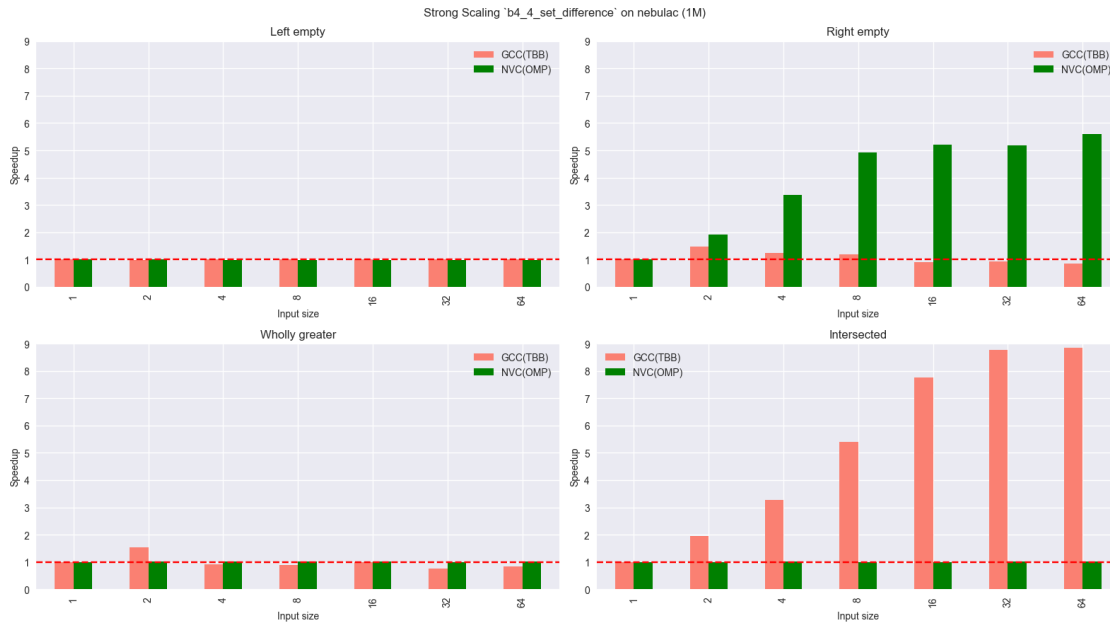
ax_4.set_ylabel('Speedup')
ax_4.set_xlabel('Input size')
ax_4.set_title('Intersected')
ax_4.axhline(y=1, color='r',linestyle='--')
ax_4.set_ylim(0,9)

fig.suptitle("Strong Scaling `b4_4_set_difference` on nebulac (1M)")

fig.tight_layout()

plot("Strong Scaling `b4_4_set_difference` on nebulac (1M)")

```



```
[82]: # efficiency graphs
```

```
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(16, 9))
```

```
# left empty
```

```
ax_1 = b4_4_set_difference_cutoff_left_empty_efficiency.
```

```
    plot(x='threads', color=[GCC_TBB_COLOR, NVC_OMP_COLOR], ax=axes[0,0])
```

```
ax_1.set_ylabel('Efficiency')
```

```
ax_1.set_xlabel('Input size')
```

```
ax_1.set_title('Left empty')
```

```
ax_1.axhline(y=1, color='r', linestyle='--')
```

```
ax_1.set_ylim(0, 1.05)
```

```
ax_1.set_xscale('log', base=2)
```

```
current_values = ax_1.get_xticks()
```

```
ax_1.set_xticklabels(['{:,.0f}'.format(x) for x in current_values])
```

```
# right empty
```

```
ax_2 = b4_4_set_difference_cutoff_right_empty_efficiency.
```

```
    plot(x='threads', color=[GCC_TBB_COLOR, NVC_OMP_COLOR], ax=axes[0,1])
```

```
ax_2.set_ylabel('Efficiency')
```

```

ax_2.set_xlabel('Input size')
ax_2.set_title('Right empty')
ax_2.axhline(y=1, color='r',linestyle='--')
ax_2.set_ylim(0,1.05)

ax_2.set_xscale('log', base=2)
current_values = ax_2.get_xticks()
ax_2.set_xticklabels(['{:,.0f}'.format(x) for x in current_values])

# wholly greater
ax_3 = b4_4_set_difference_cutoff_wholly_greater_efficiency.
    plot(x='threads',color=[GCC_TBB_COLOR,NVC_OMP_COLOR],ax=axes[1,0])

ax_3.set_ylabel('Efficiency')
ax_3.set_xlabel('Input size')
ax_3.set_title('Wholly greater')
ax_3.axhline(y=1, color='r',linestyle='--')
ax_3.set_ylim(0,1.05)

ax_3.set_xscale('log', base=2)
current_values = ax_3.get_xticks()
ax_3.set_xticklabels(['{:,.0f}'.format(x) for x in current_values])

# intersected
ax_4 = b4_4_set_difference_cutoff_intersected_efficiency.
    plot(x='threads',color=[GCC_TBB_COLOR,NVC_OMP_COLOR],ax=axes[1,1])

ax_4.set_ylabel('Efficiency')
ax_4.set_xlabel('Input size')
ax_4.set_title('Intersected')
ax_4.axhline(y=1, color='r',linestyle='--')
ax_4.set_ylim(0,1.05)

ax_4.set_xscale('log', base=2)
current_values = ax_4.get_xticks()
ax_4.set_xticklabels(['{:,.0f}'.format(x) for x in current_values])

fig.suptitle("Efficiency `b4_4_set_difference` on nebulac (1M)")

fig.tight_layout()

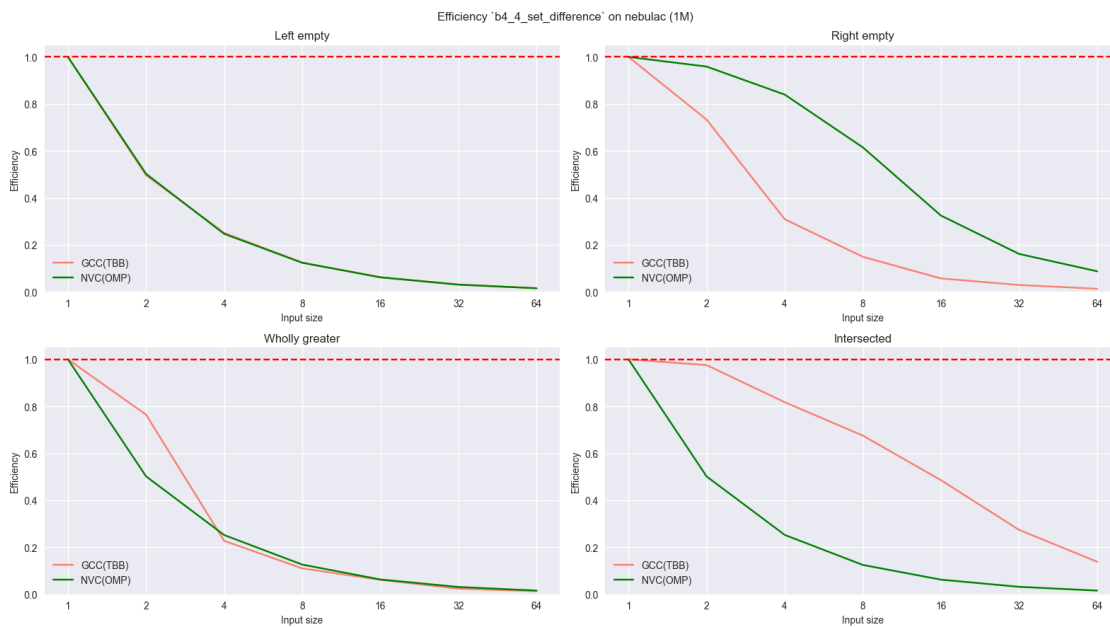
plot("Efficiency `b4_4_set_difference` on nebulac (1M)")

```

```

/var/folders/42/fk0jfryd1dd1ztdldncqc1cw0000gn/T/ipykernel_33509/848947553.py:18
: UserWarning: FixedFormatter should only be used together with FixedLocator
  ax_1.set_xticklabels(['{:,.0f}'.format(x) for x in current_values])
/var/folders/42/fk0jfryd1dd1ztdldncqc1cw0000gn/T/ipykernel_33509/848947553.py:32
: UserWarning: FixedFormatter should only be used together with FixedLocator
  ax_2.set_xticklabels(['{:,.0f}'.format(x) for x in current_values])
/var/folders/42/fk0jfryd1dd1ztdldncqc1cw0000gn/T/ipykernel_33509/848947553.py:46
: UserWarning: FixedFormatter should only be used together with FixedLocator
  ax_3.set_xticklabels(['{:,.0f}'.format(x) for x in current_values])
/var/folders/42/fk0jfryd1dd1ztdldncqc1cw0000gn/T/ipykernel_33509/848947553.py:60
: UserWarning: FixedFormatter should only be used together with FixedLocator
  ax_4.set_xticklabels(['{:,.0f}'.format(x) for x in current_values])

```



### 2.4.3 Mbytes/Sec

//TODO: reflect on this

### 2.4.4 Performance Portability Calculation (Inter Compiler)

```
[83]: # calculate pp
```

### 2.4.5 Findings for H4

Findings b4\_1\_merge

Findings b4\_2\_stable\_sort



Findings b4\_3\_set\_union

Findings b4\_4\_set\_difference

General

GPU Findings

Hypothesis Findings