Dungeon Session Application:

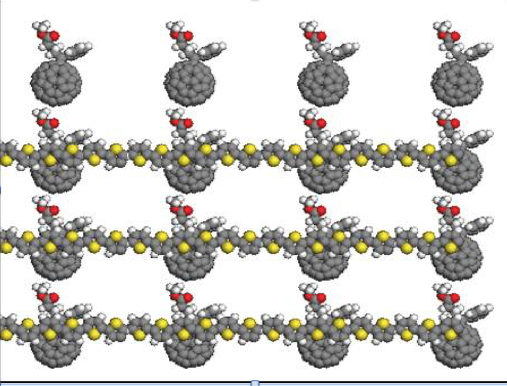
Example Answers From Quantum ESPRESSO. Erase and replace with your own answers.

**1. Target problem and sample input decks**

*1A. Design a problem that you can’t solve today but could if you have access to a machine 5-10x more powerful than Hopper or Edison and could run across the whole machine. A problem composed of multiple of smaller problems is OK.*

10,000 Atom Transition Metal Oxide System. MgO, ZnO

1000 Atom Realistic Organic Photovoltaic at Hybrid Functional Level. See below figure for example system P3HT



*1B. Create a sample input deck of a typical system you can run today on Edison (at realistic scale) that resembles a smaller version of the type of problem you want to run in 2017. This should be shareable with the NESAP team (NERSC Staff and Engineers at Intel and Cray).*

Using the QE based MiniDFT “Large” Input deck. 2000 atoms of MgO. The input file is accessible at: https://drive.google.com/file/d/0B1vuU9dbI-YnSTczMWhmanFWekE/view?usp=sharing

**2. Create A Profile Of Your Application for the Sample Input Deck**

*Use VTune on Edison. Describe the results. Where are the hotspots in your calculation? What fraction of your run is serial vs threaded on Edison. (If you prefer another profiler, that’s fine too.)*

*To do this experiment:*

1. *Build your executable using:*
   1. *-g flag throughout*
   2. *Use -dynamic on your linkline to dynamically link your application*
   3. *Do “module unload darshan” before building*
2. *Run with your code with a batch script similar to the following on Edison:*

*%cat script.pbs*

*#PBS -N MiniDFT\_lg*

*#PBS -q ccm\_queue*

*#PBS -l mppwidth=<num of nodes to use x 24>*

*#PBS -l walltime=<walltime to use>*

*#PBS -V*

*#PBS -j oe*

*cd $PBS\_O\_WORKDIR*

*export OMP\_NUM\_THREADS=<number of threads to use per MPI task>*

*module unload darshan*

*module load vtune*

*#The below runs 1 MPI task per node. This is not necessary. You may set up your job with*

*# multiple ranks per node*

*aprun -n 200 -N 1 -S 1 -cc numa\_node ./vscript*

*%cat vscript*

*#!/bin/bash*

*# The below requests a vtune report from the first MPI task, you can change the logic*

*# here to get reports from multiple MPI tasks. However, please limit this to one per node*

*# or vtune crashes*

*if [ $ALPS\_APP\_PE -eq 0 ]*

*then*

*amplxe-cl -collect hotspots -data-limit=0 -- /global/homes/j/jdeslip/Builds/Edison2/MiniDFT-1.06/src/mini\_dft -ntg 5 -in large.in*

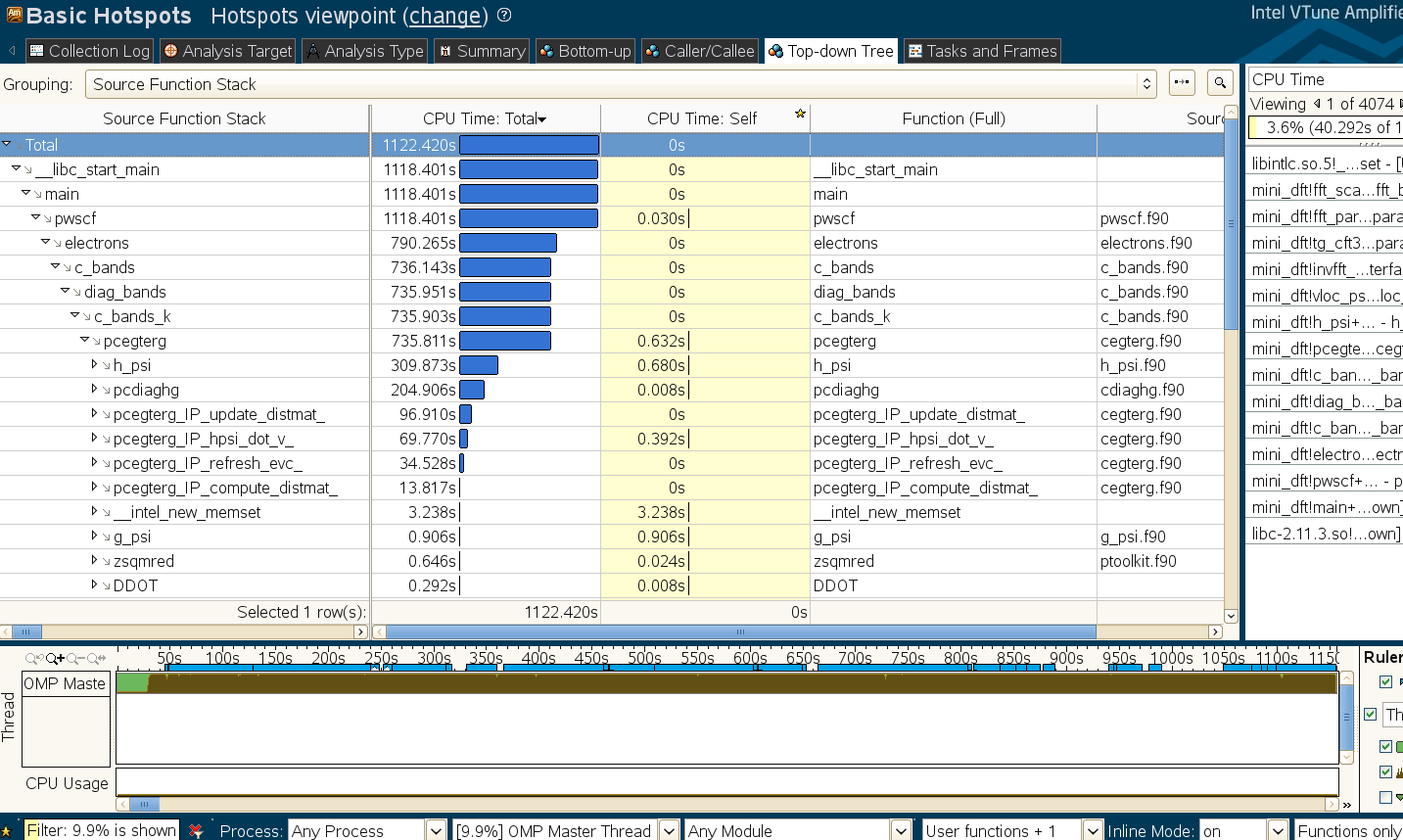
*else*

*/global/homes/j/jdeslip/Builds/Edison2/MiniDFT-1.06/src/mini\_dft -ntg 5 -in large.in*

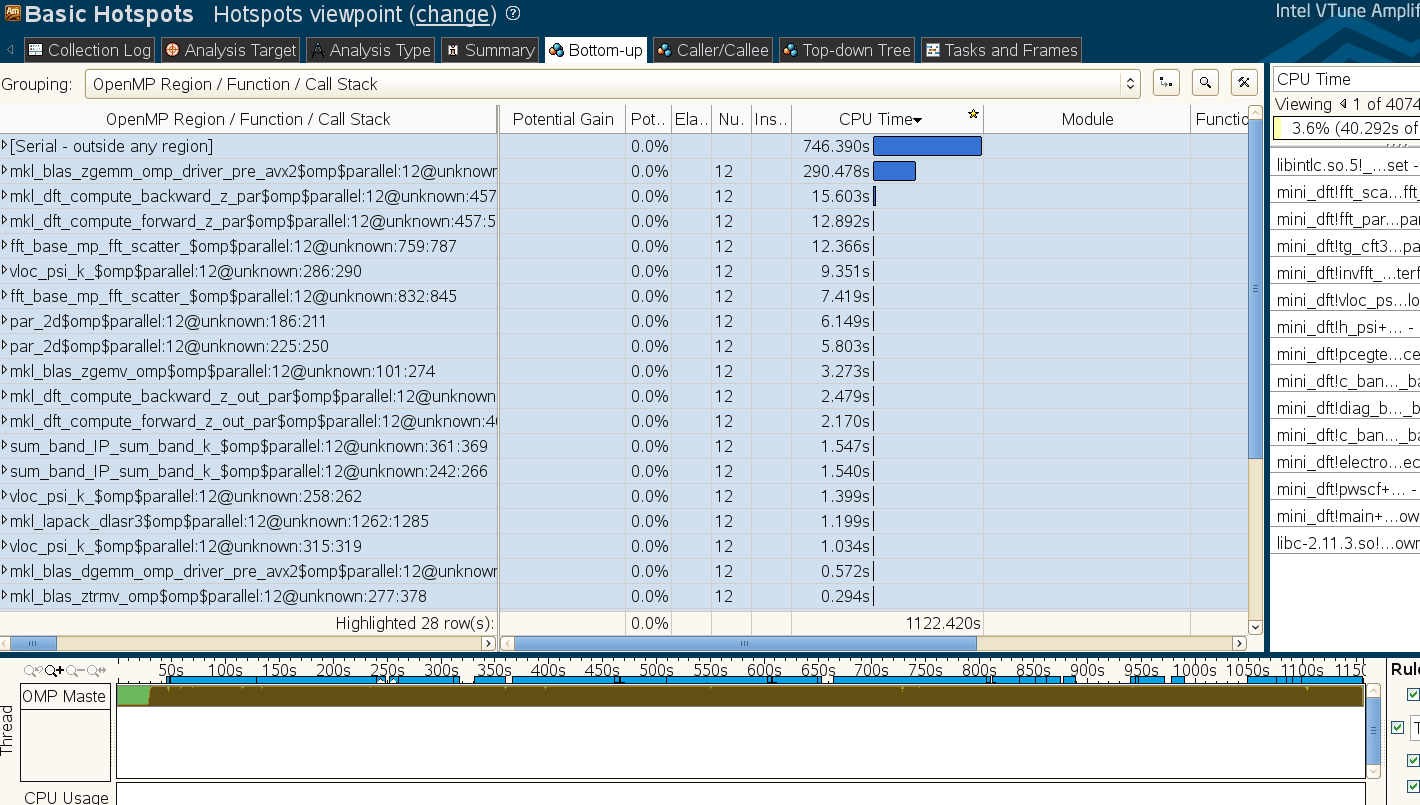
*fi*

1. *The above should create a directory like r000hs in your run directory. Run amplxe-gui on the login node and open the .amp\* file in the folder. You can use the “bottom-up” and “top-down” tabs to profile your application. If you using OpenMP, it may be helpful to filter the view by all threads and just the master thread.*

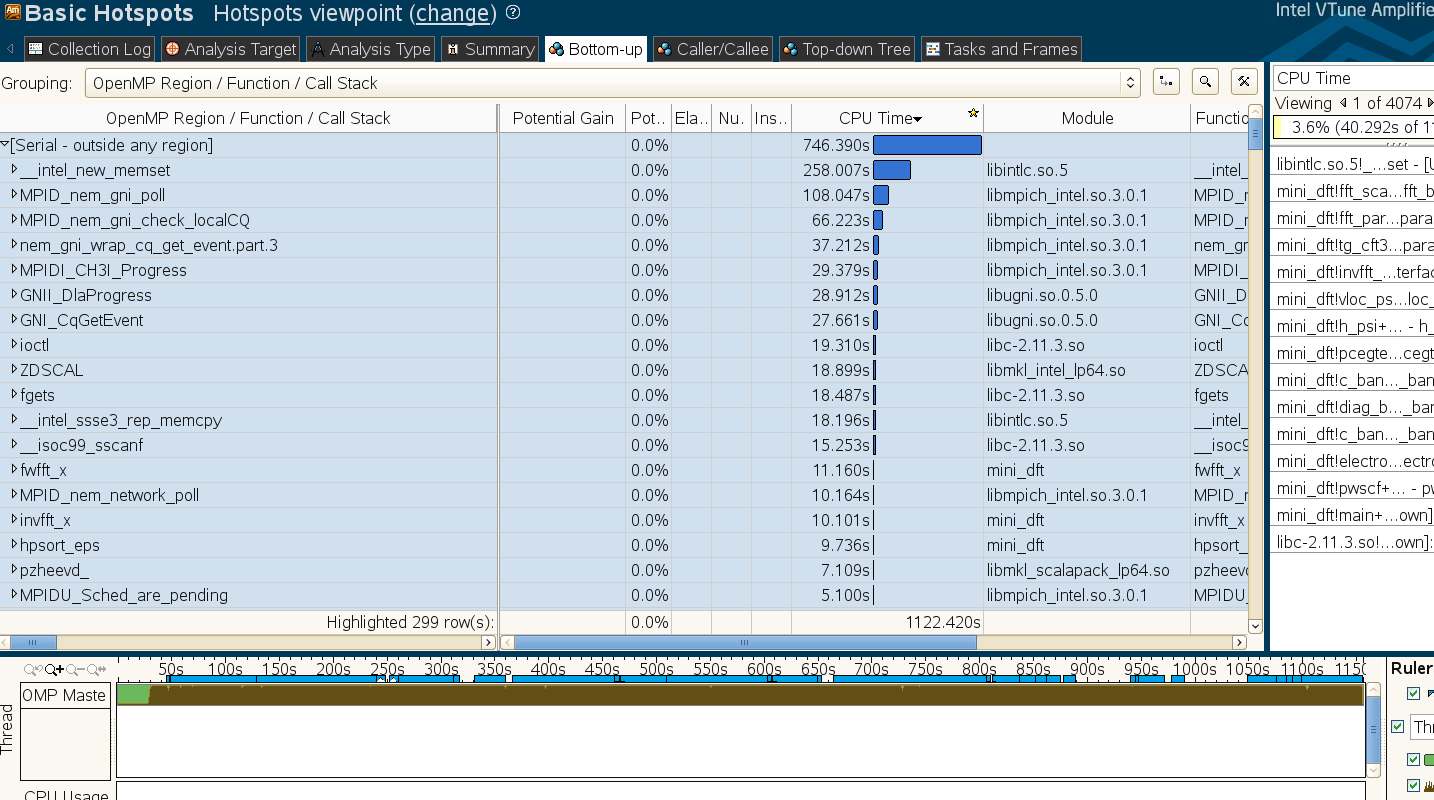
ESPRESSO VTune Top Down View:



ESPRESSO Bottom Up OpenMP Regions:

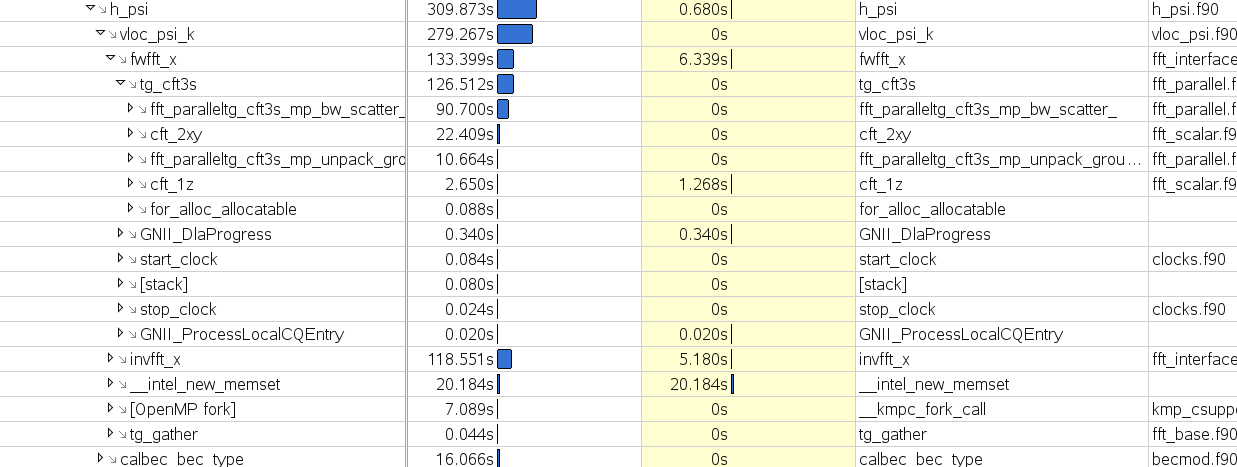


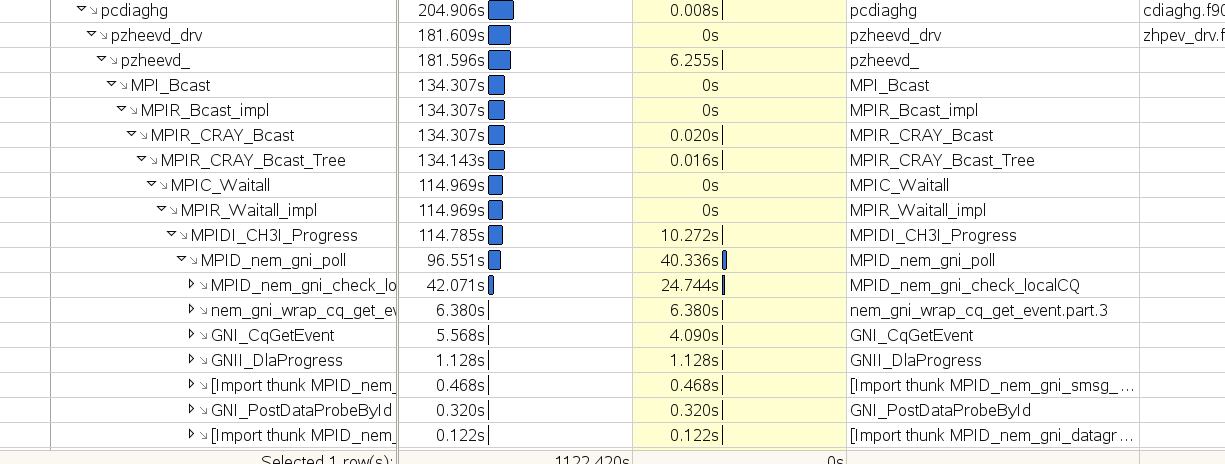
ESPRESSO OpenMP/Serial Regions Expanded



You can see that most of the time is spent in serial regions of the code. And of this time, a significant fraction is memset.

From a topdown view, the routines h\_psi and pcdiaghg take the most time. Here is a zoomed in view of the stack starting from these routines:





See appendix for detailed code breakdown.

**3. Can kernels be created out of the hotspots in your code?**

*Kernels are self-contained code snippets intended to represent a single nodes work for a bigger problem run at scale.*

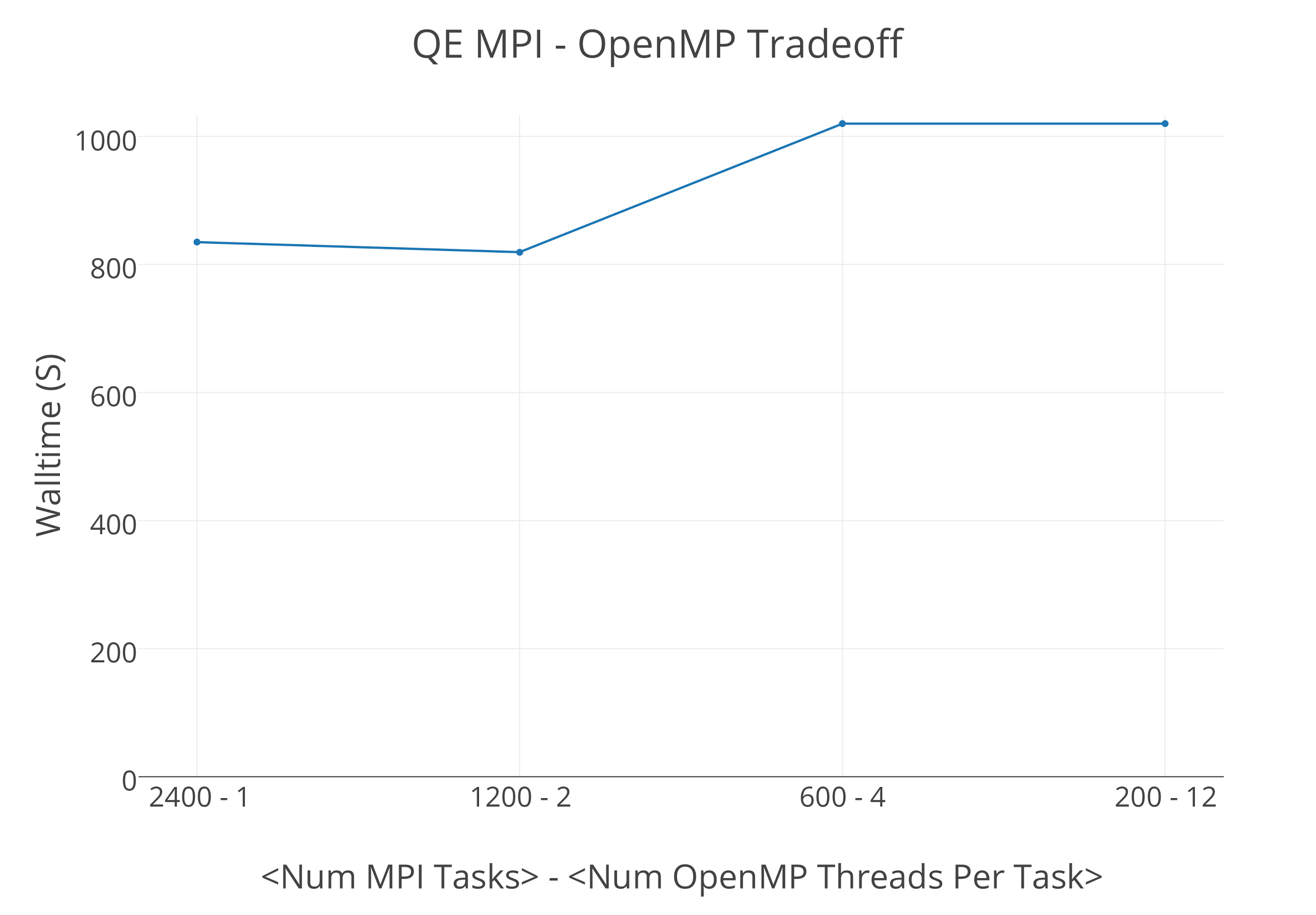
In progress for QE. Compute intensive part of code is generally in libraries (FFTW, ScaLAPACK, BLAS). Problems at this point appear to be that we are highly memory bandwidth bound and serial (see below) in the FFT dominated h\_psi routines. The vloc\_psi\_k routine will likely make a good kernel.

**4. Provide MPI vs MPI + OpenMP Scaling Curve for your sample input deck or hotspot kernels**

*4A. For a fixed number of CPU cores, run your sample calculation in pure MPI mode and mixed OpenMP-MPI mode varying from 2 to 12 threads per MPI task.*

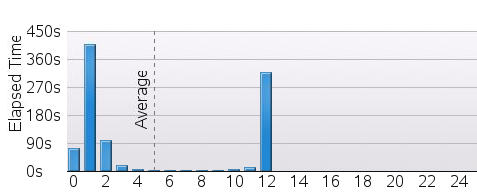
From the output of #2, you can see significant portions of the sample runtime (with 200 MPI and 12 OpenMP threads) appear in serial regions. Despite this, the MPI vs. OpenMP scaling curve below is relatively flat. The optimal performance occurring with two threads per MPI task. At 12 threads per MPI task performance is about 20% slower than than pure MPI.

For Quantum ESPRESSO, we ran our test on a fixed 2400 cores and varied the number of MPI tasks between 2400 and 200 and the number of OpenMP threads between 1 and 12.

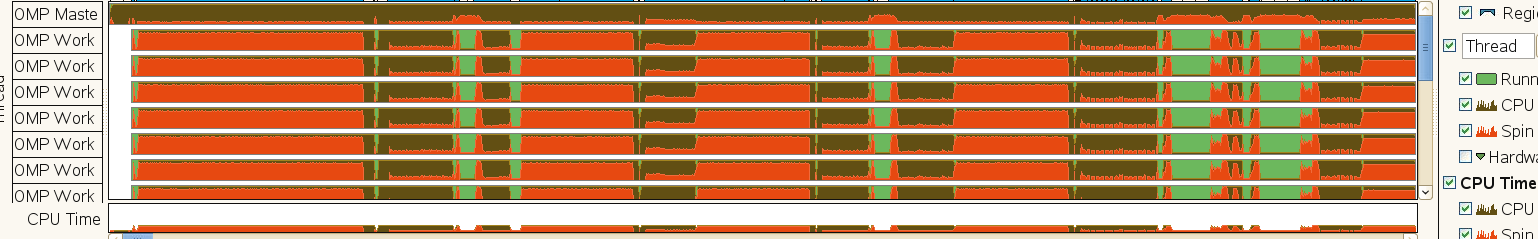


*4B. Run the vtune advanced-hotspots collection to see thread utilization and spin time information. Are threads active (and not spinning) throughout your sample calculation? Use the instructions in question 2 with “-collect hotspots” replaced by “-collect advanced-hotspots”*

We ran advanced-hotspots. Below is the CPU Usage Histogram. A majority of the time is spent with only 1 thread active.



The below shows thread utilization (brown) and spin(red) over the course of the run. You can see that it is rare when all threads are active. As shown in the appendix, green regions correspond to SCALAPACK.



**5. How much memory does your sample input run consume? How will this change between your sample input and 2017 target problem?**

*The maximum memory per task used for your job can be recovered from the MyNERSC completed jobs we portal. Browse to* [*https://my.nersc.gov/completedjobs.php*](https://my.nersc.gov/completedjobs.php)*, click on the job in question, and read off the memory value (Note this value does not include huge-pages, which may affect some users who explicitly utilize large pages).*

The “large” input above consumes 10 GB per MPI task when run with 200 MPI tasks and 12 OpenMP threads. This gives roughly **2 TB total memory**.

The memory used in Quantum ESPRESSO scales as the number of atoms squared. So, a 10,000 atom calculation would require **50 TB total memory**.

**6. Is your application sensitive to changes in memory bandwidth?**

*6A. Run your example on Edison using 12 tasks/threads per node (using 2 x N nodes) versus 24 tasks/threads per node (using N nodes). If you only utilize half of the cores on a node (and half fill each socket), then the memory bandwidth available to each core will be greater (approaching factor of 2). An example comparison would be:*

*aprun -n 24 -N 12 -S 6 ...*

*vs.*

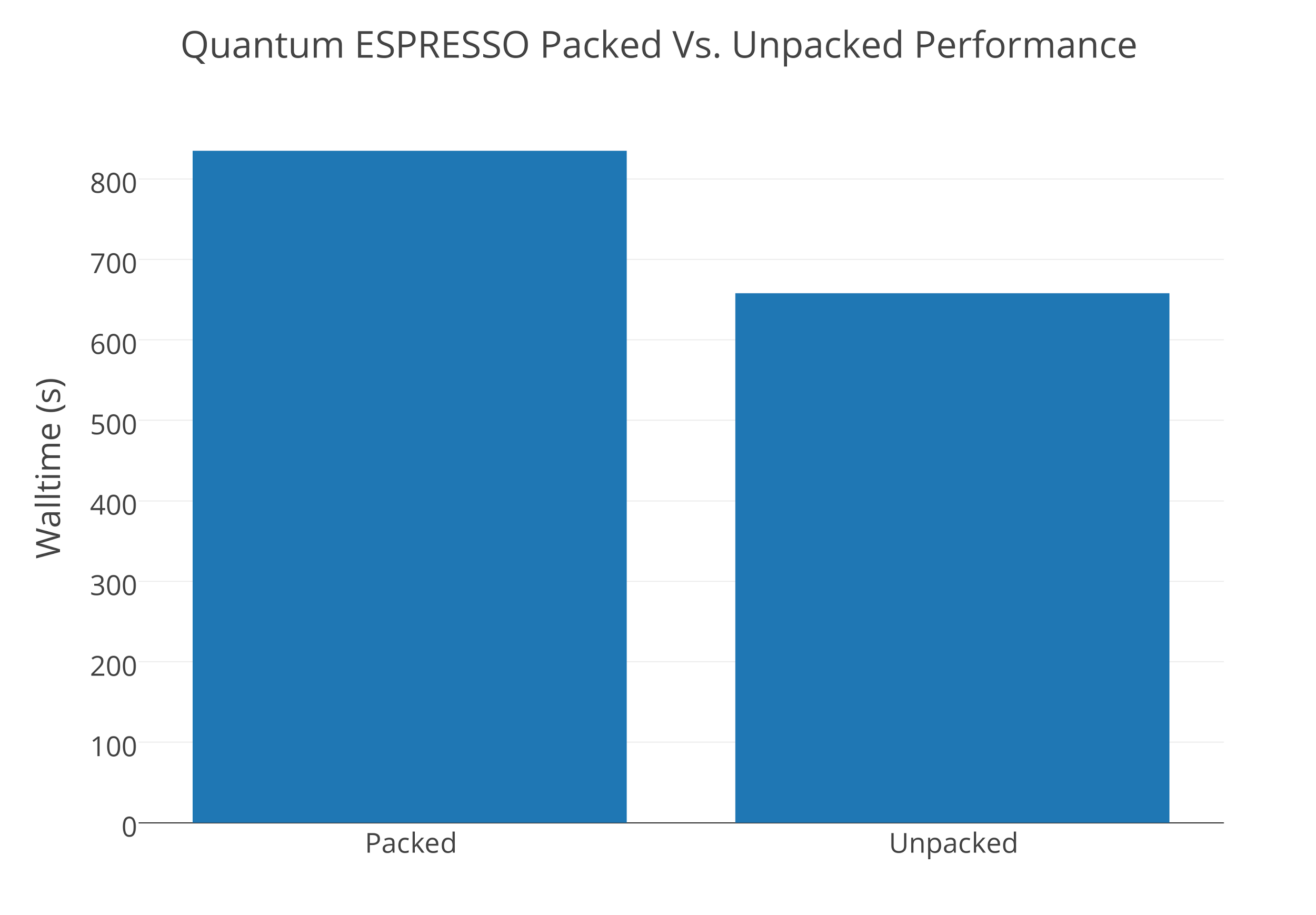
*aprun -n 24 -N 24 -S 12 ...*

*If the runtime varies significantly between these two runs, it indicates that your code or kernel is memory bandwidth bound.*

For the Quantum ESPRESSO example above, we ran the following test cases.

Packed: 2400 MPI tasks, 24 MPI tasks per node, 12 MPI tasks per socket

Unpacked: 2400 MPI tasks, 12 MPI tasks per node, 6 MPI tasks per socket



Unpacking leads to approximately 20% improvement. This suggests that our target problem in Quantum ESPRESSO is significantly bandwidth bound.

We can look at how all the top routines fared under the packed to unpacked transition:

CPU TIME (Sec)

Routine Packed Half-Packed Ratio

h\_psi 6529 2190x2 67%

-vloc\_psi\_k 5524 1738x2 63%

--invfft\_x 2652 788x2 59%

--fwfft\_x 2466 842x2 68%

distmat 2532 1051x2 83%

-zgemm 1845 855x2 93%

pcdiaghg 2235 1194x2 107%

hsi\_dot\_v 1745 749x2 86%

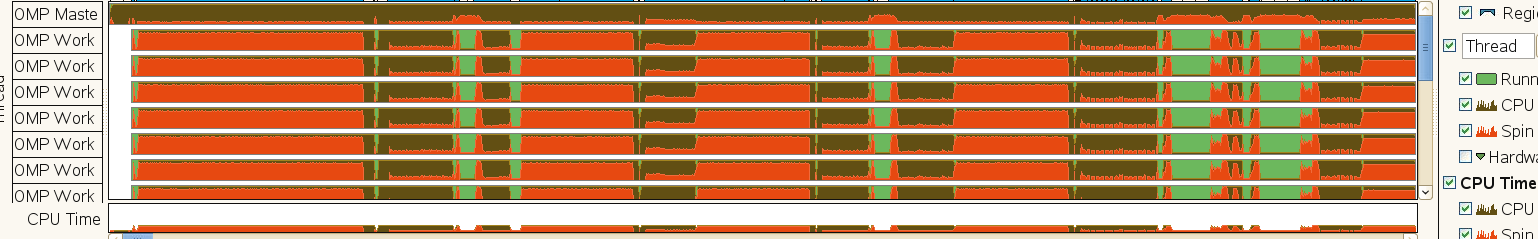
-zgemm 1401 650x2 93%

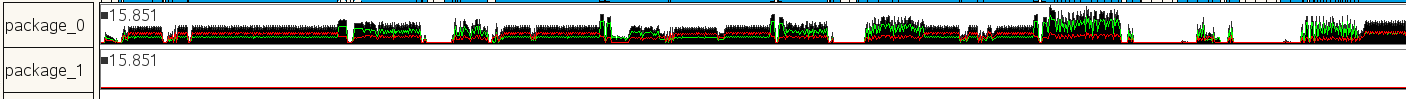
We can see from the above table that most of the gains in unpacking occur in the areas of the code (h\_psi/vloc\_psk\_k) that are dominated by the parallel FFTs. The parts of the code dominated by zgemm saw more modest gains (zgemm itself sees only a 7% boost). The pcdiaghg scalapack routine actually slows down, presumably due to more network contention.

*6B. Collect the bandwidth report from your application in vtune. Do certain areas of the code use more bandwidth than others? If you compare the vtune output for the packed and unpacked tests, do certain regions slow down more significantly than others?*

*Instructions for vtune: Repeat instructions in question 2 with “-collect bandwidth”*

We ran the the vtune bandwidth collection and show it next to the advanced hotspots thread activity for comparison. For this test we run with 200 MPI tasks and 12 OpenMP threads per task.





The bandwidth used appears to be under 16 GB a second. However, there are two issues here. One, unless you zoom in on a certain region, the graph shows an averaged value in each interval. If you zoom in on the first solid region in the plot you see a structure like the following:



Many other regions are essentially bandwidth bound because only one thread is active and using roughly the same bandwidth as a single thread of stream triad. If we were to pack 12 MPI tasks into the socket, we would require more bandwidth than available from the socket.

Need General Exploration Still

*6C What can you tell us about the memory access patterns in the sensitive regions of your code? Are the memory accesses sequential, strided, random? Do you know your cache utilization or miss rates? Does your code actively use blocking to reuse cache?*

For the ZGEMM regions the memory access patterns are sequential. For the FFT regions the access patterns are more complex - usually done in a divide and conquer approach. Need to more work here for QE.

**7. Is your application sensitive to changes in clock speed?**

*Run your example at full vs. reduced clock speed on Edison. This test can tell you whether your code/kernel is CPU bound or not. If reducing the clock speed makes a significant performance difference in your application. Your application is at least partially CPU bound.*

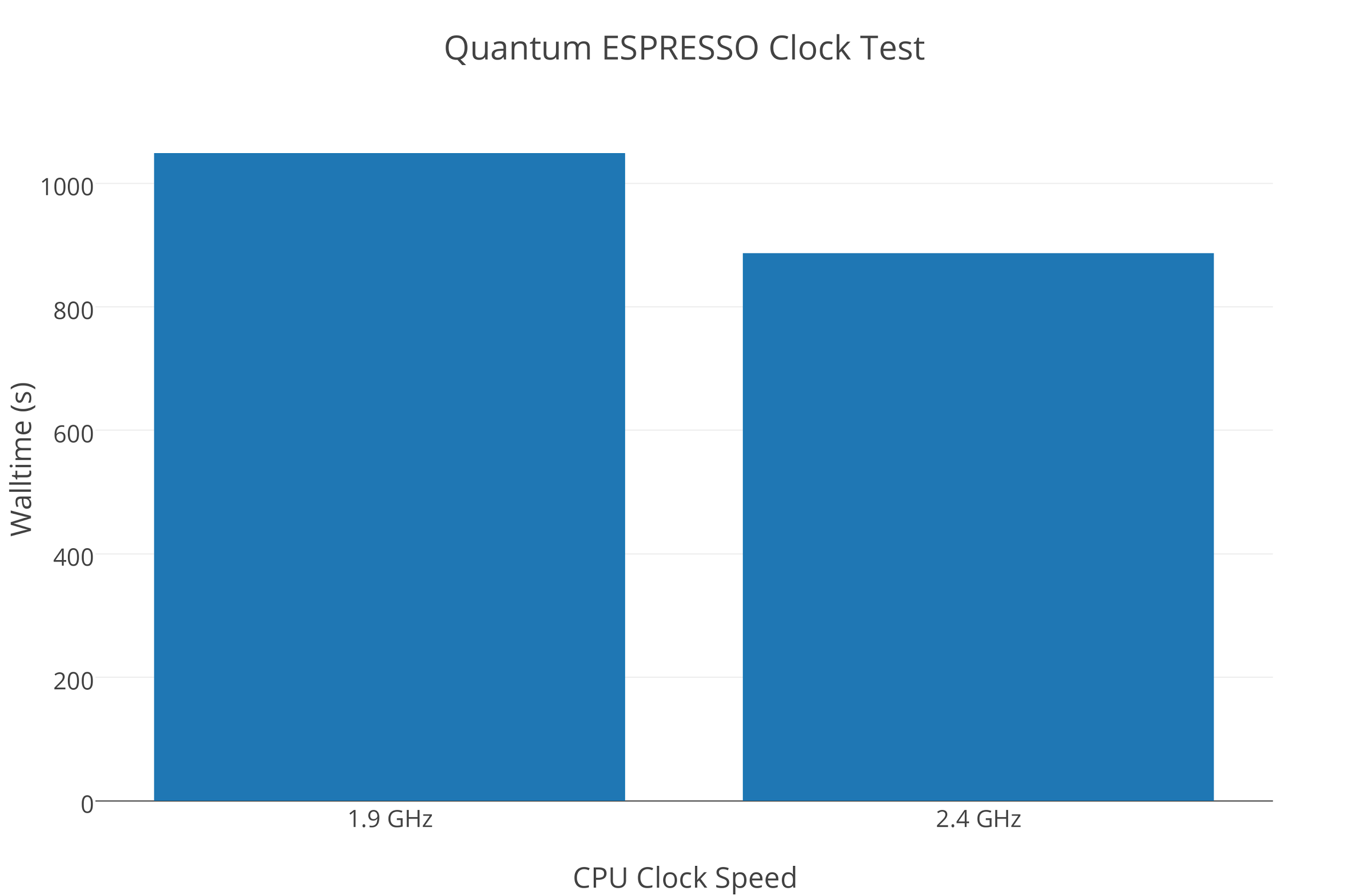
*To do this experiment:*

*To perform this test, run your calculation on Edison and specify the "-p-state" flag to aprun to specify the CPU speed in KHz (max value of 2.4 GHz, min value of 1.2 GHz). For example,*

*aprun --p-state=1900000 ...*

*specifies a clock speed of 1.9 GHz. Compare the value with 1.9 GHz and 2.4 GHz. Don’t go below 1.9 GHz or other effects come into play.*

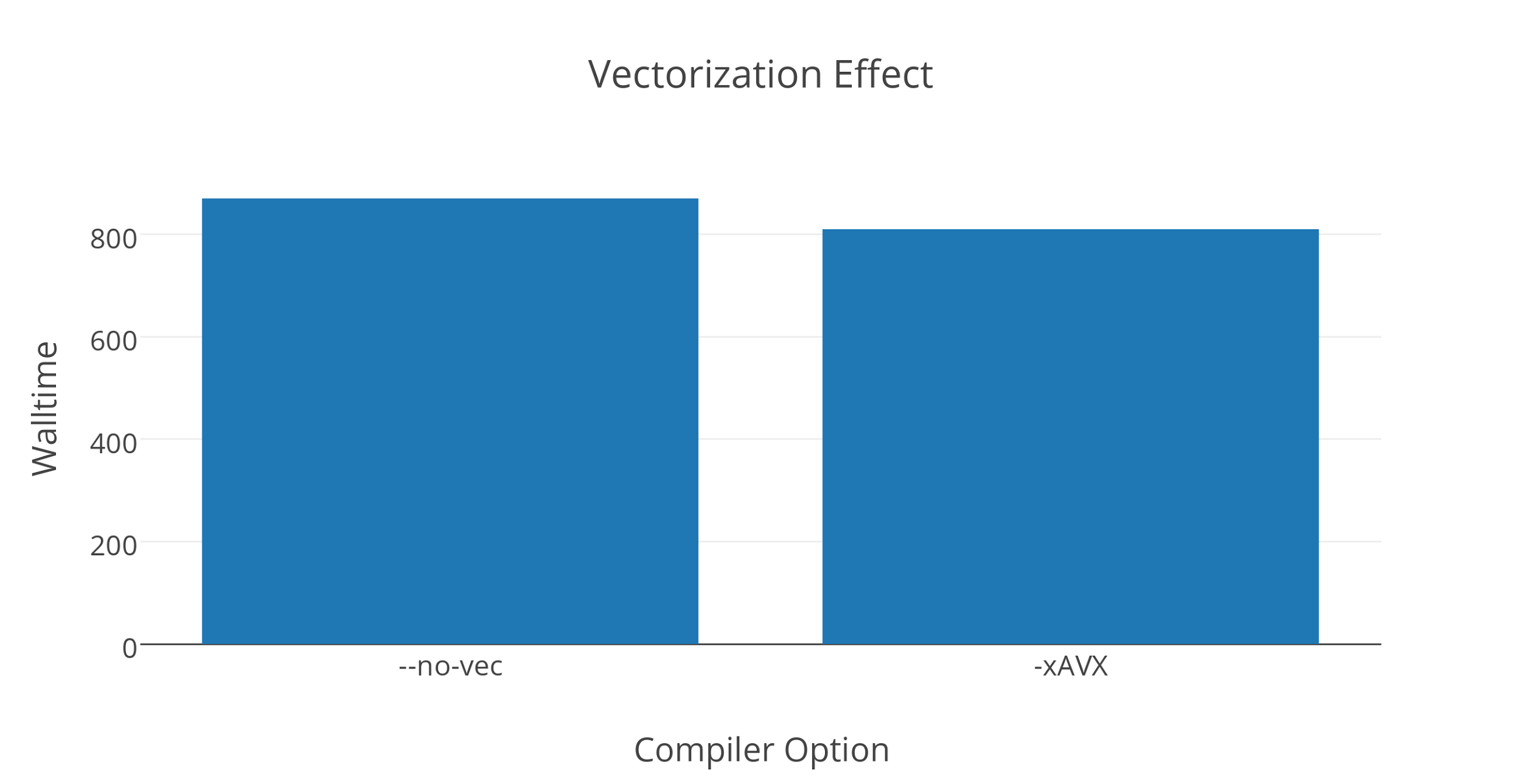
We performed the half clock speed test and found a walltime increase by approximately 18% after reducing the clock speed by 21%. This suggests while the code is bandwidth sensitive, the overall runtime is also CPU sensitive, presumably due to the dense linear algebra regions.



**8. What is your vector utilization?**

*8A. Compile and run with the flags --no-vec vs -xAVX on Edison. What is the performance difference?*

We ran with and without compiler vectorization. As shown in the plot below, the change makes little difference. This is because the code is heavily dependent on math libraries (FFTW and BLAS) and what is not is bandwidth or communication bound.



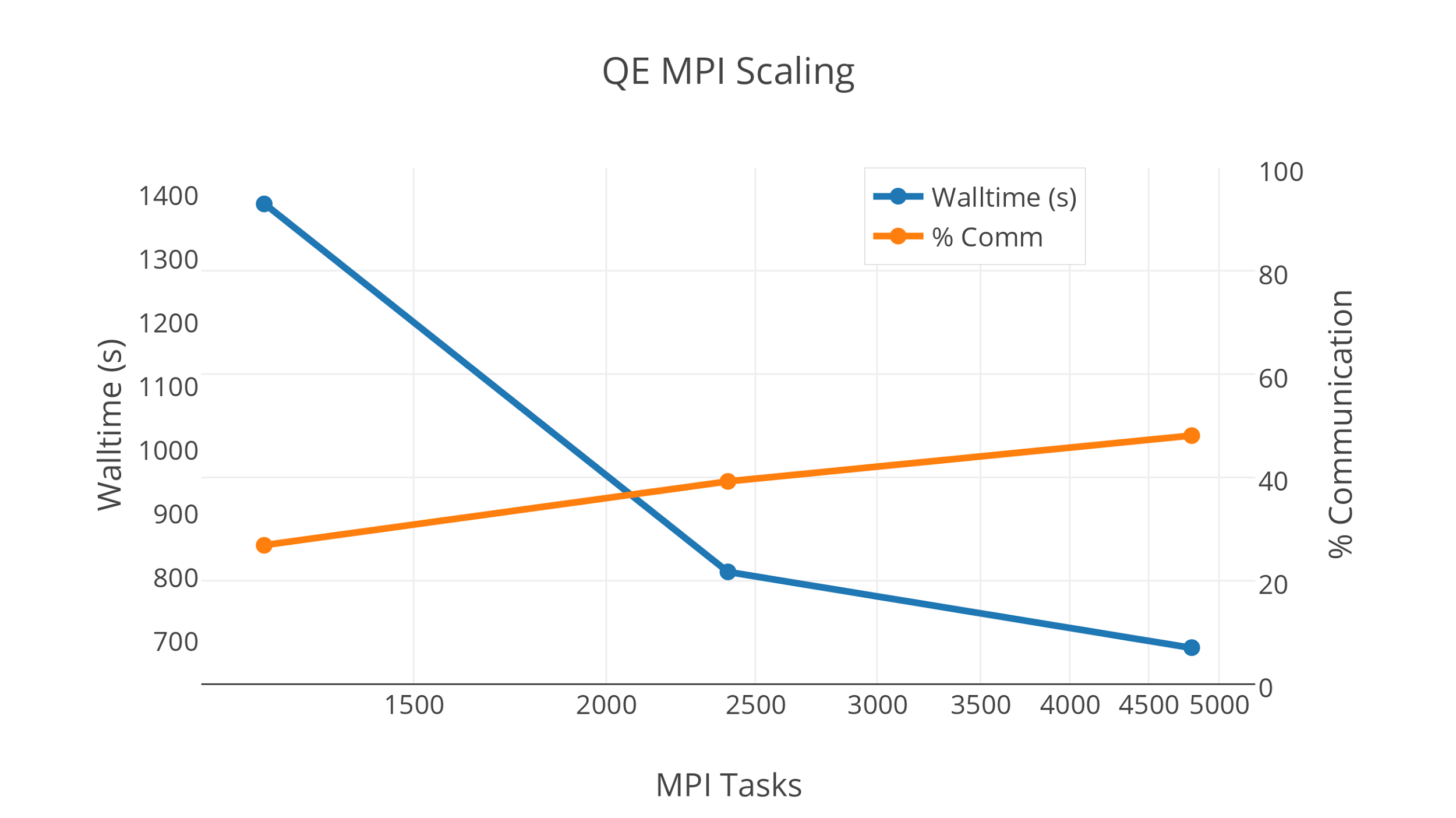
**9. IO + Communication**

*9A. Link your application with IPM by doing:*

1. *“module load ipm” before building*
2. *“module unload darshan” before building*
3. *add $IPM to the end of you link line*

*Run your application and view the IPM report near the end of standard out.*

We performed an MPI scaling study with IPM. Communication rises to about 50% with 4800 MPI tasks.

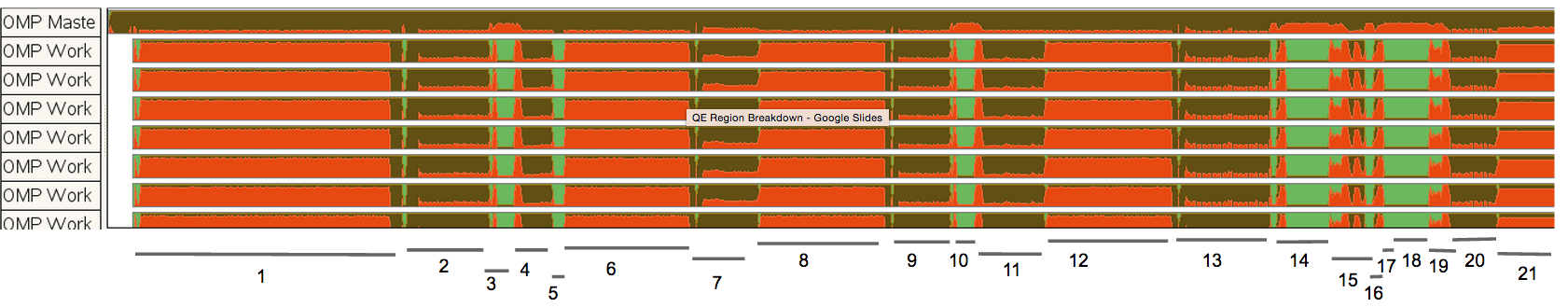


*9B. Determine the fraction of time your code spends in IO. You can use Darshan (*[*https://www.nersc.gov/users/software/debugging-and-profiling/darshan*](https://www.nersc.gov/users/software/debugging-and-profiling/darshan)*) to profile your IO usage, including average rate, total data read/written and averarage transation size.*

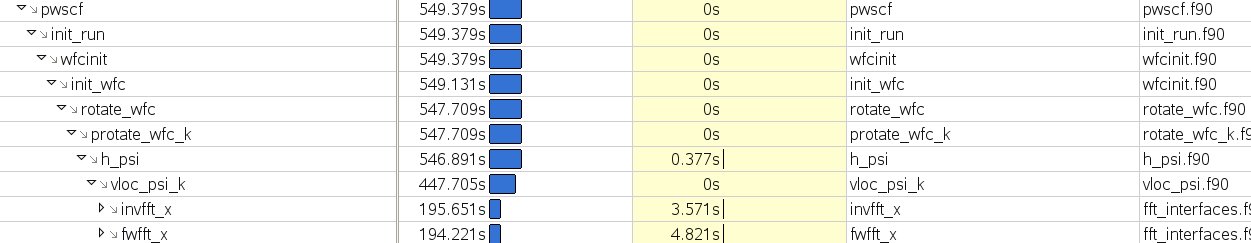
Our QE sample calculation uses a negligible amount of IO.

Appendix Detailed Code Breakdown

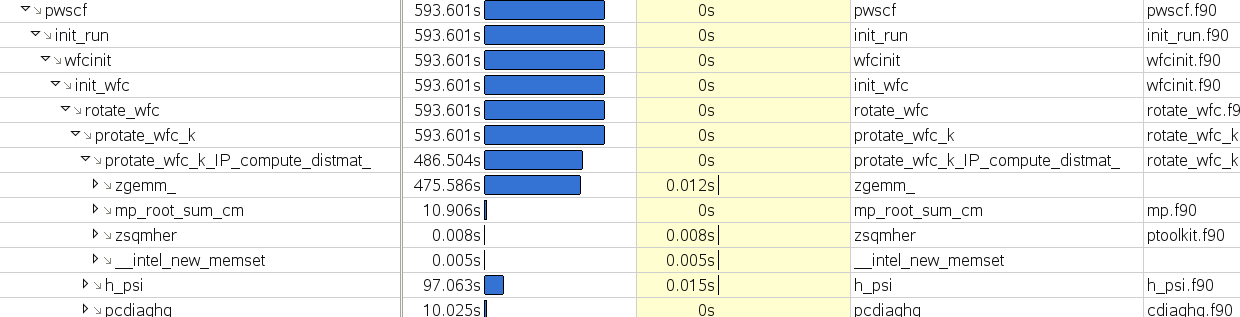
Here we breakdown the profile by code region over the sample runtime.



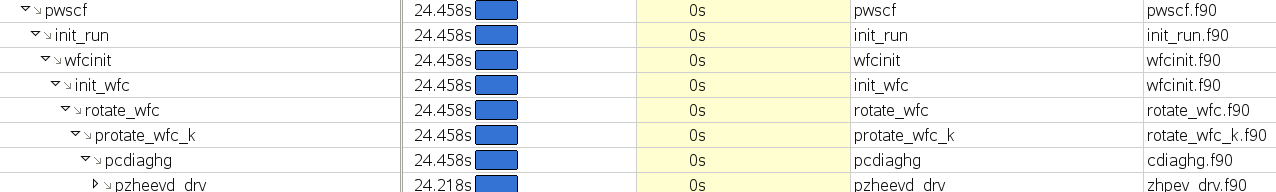
1.



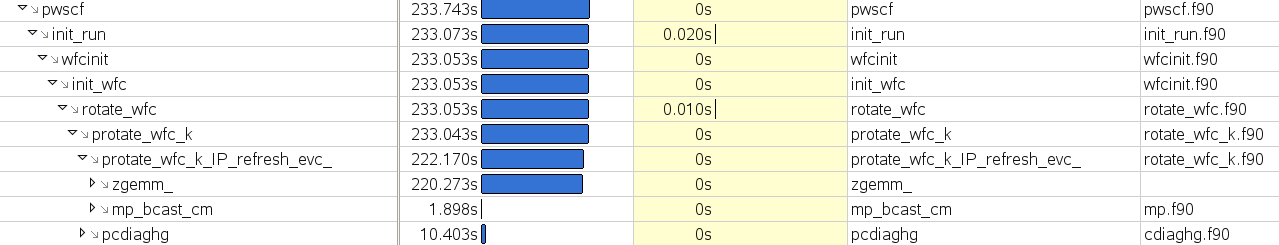
2.



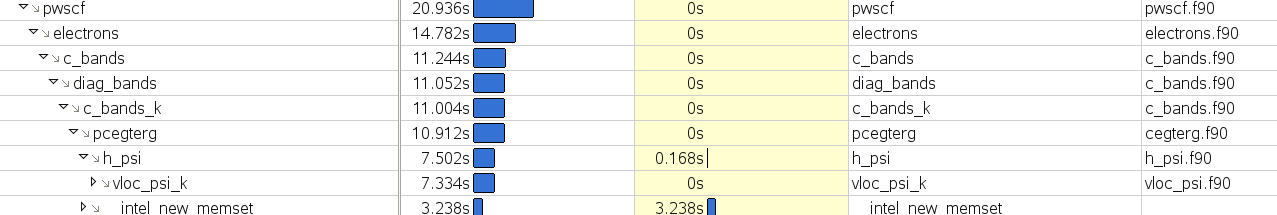
3.



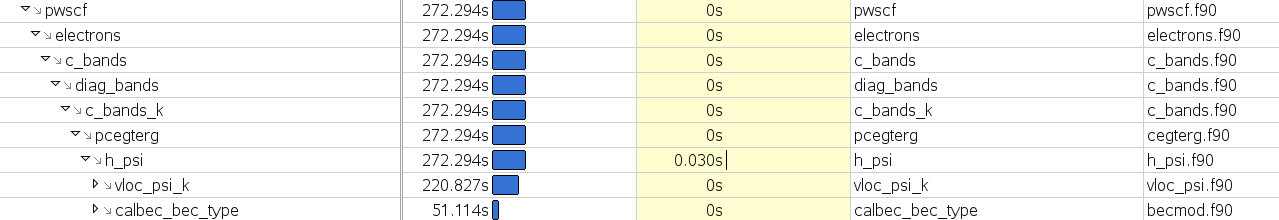
4.



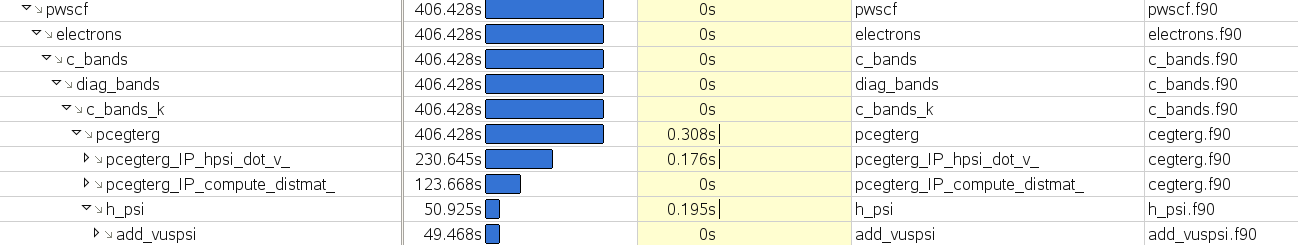
5.



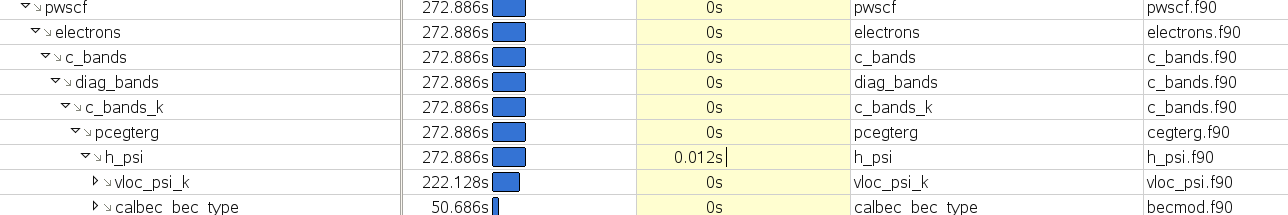
6.



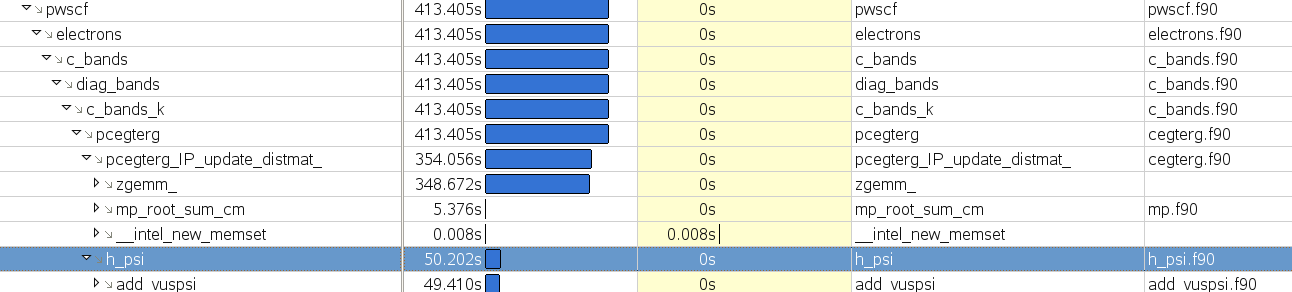
7.



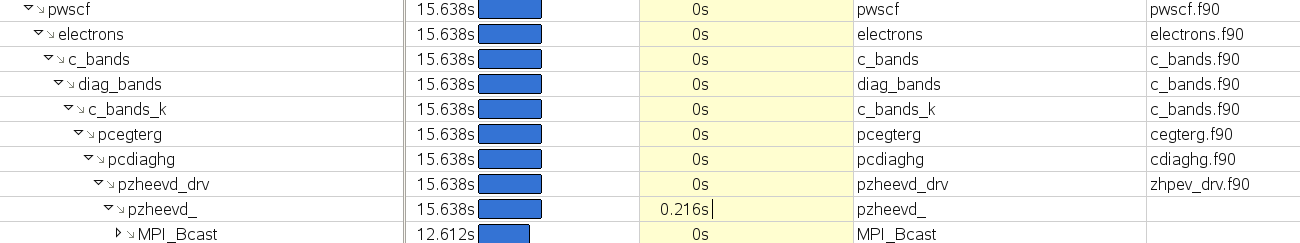
8.



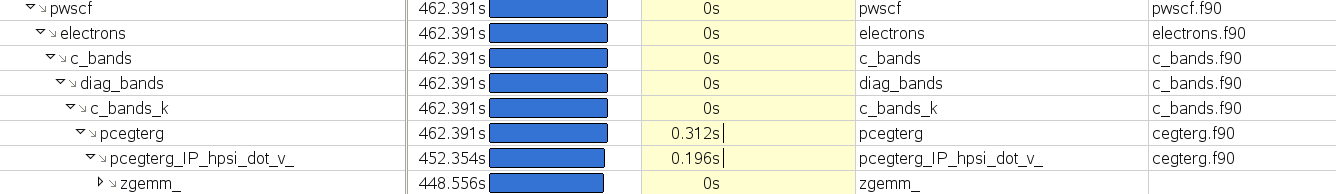
9.

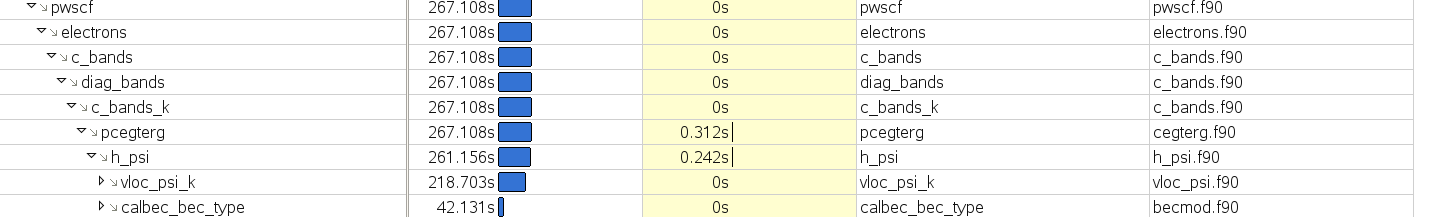


10.

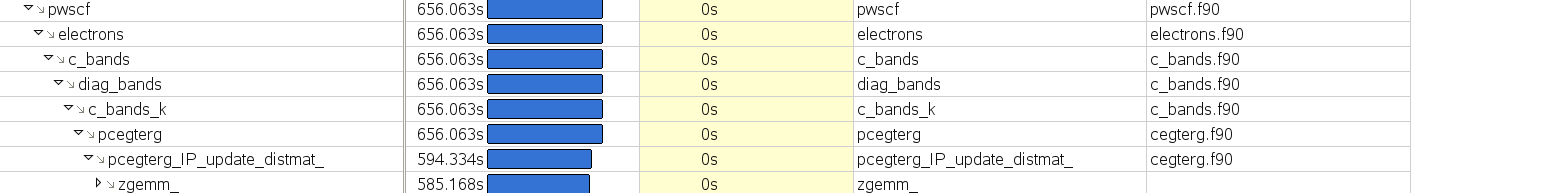


11.

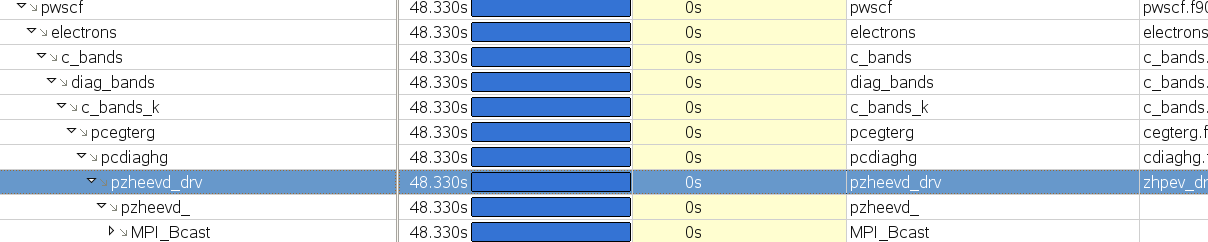


12. 

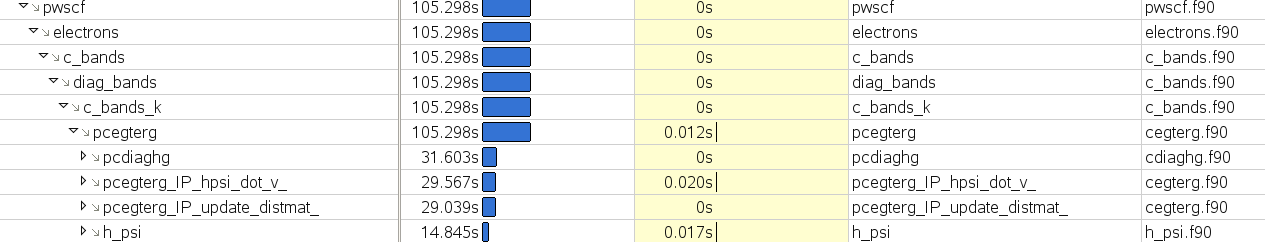
13.



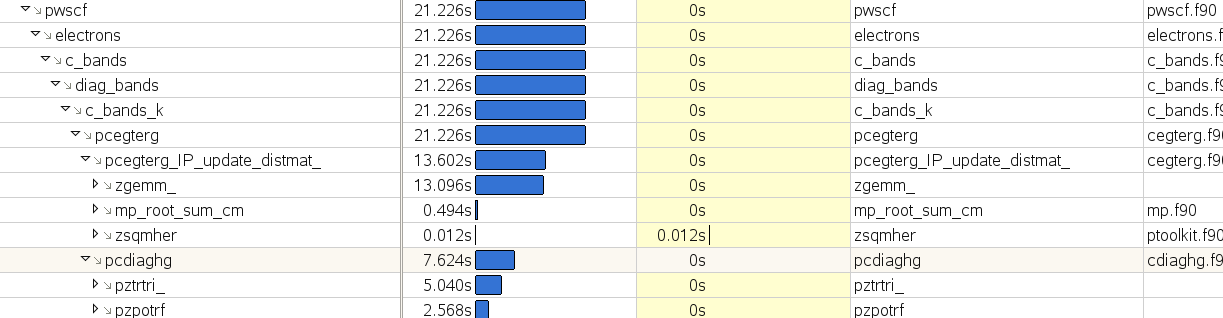
14.



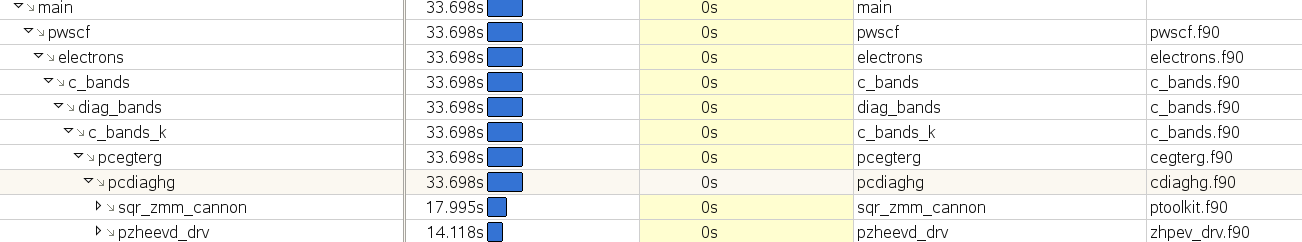
15.



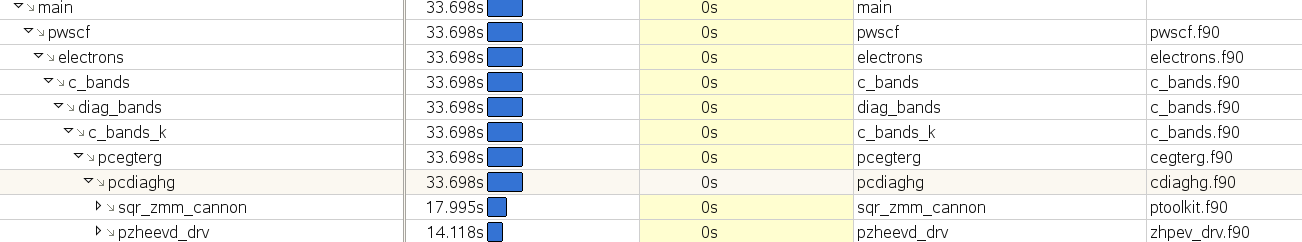
16.



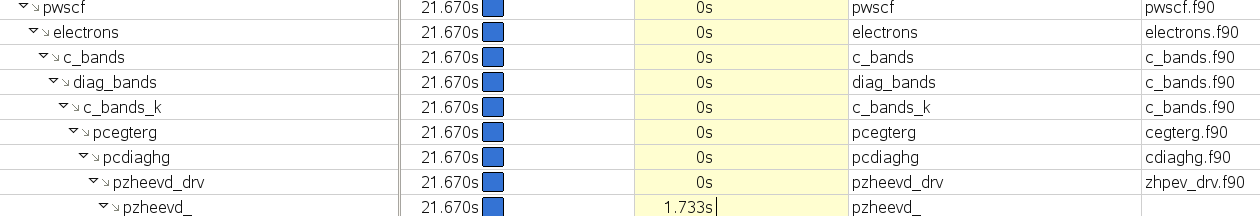
17.



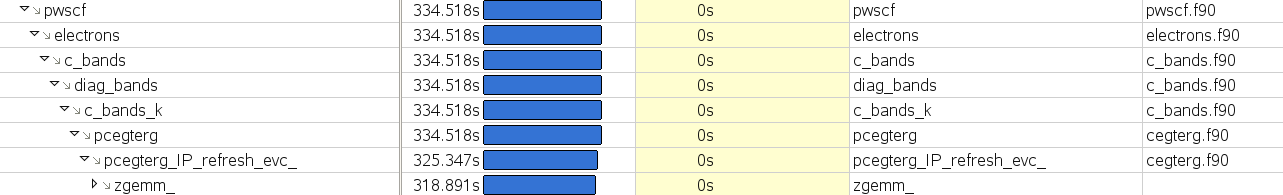
18.



19.



20.



21.

