Quantum-Espresso

Introduction

Quantum-Espresso is a software designed for electronic-calculation and material-modeling, the features it has are free and open-sourced. The software is based on Density-Functional Theory, Plane Wave Basis Set and Pseudopotentials. It is a project coordinated by the Quantum-Espresso foundation which is formed by many research groups and scientists around the world. Written mainly in Fortran-90, Quantum-Espresso also has some parts which are written in C and Fortran-77. The wide variety of use also makes the software more stronger, it can solve the following problems:

- Ground State Calculations
- Structural Optimization
- Transition States and Minimum Energy Paths
- Response Properties
- Molecular Dynamics
- Spectroscopic properties
- Generation of pseudopotentials

In other words, it is a powerful tool to study materials properties at nanoscale and fast screen potential candidates for certain applications.

Finally, in 2022-HPCAI competition, we will use Quantum-Espresso to run a SCF calculation to obtain the total energy of CeO_2. Our goal is to achieve the best parallel performance on the powerful supercomputing cluster to get the lowest execution time (CPU-Time). To achieve the goal, below are the parameters which can be tunable to make the performance of the parallelization and the calculation more efficient.

- Number of Process (-np)
- Number of Pools (-npool)
- Diagonalization (-ndiag)
- Number of OpenMP threads (OMP_NUM_THREADS)

Parameters

- Number of Processes (-np)

  The number of processes we can run on each node is 48, which means that we should limit our number of processes below 1536 since we cannot exceed the limitation of 32 nodes. Moreover, the number of processes also has effects on the number of pools and diagonalization.

- Number of Pools (-npool)

  By the definition, the number of pools must be a divisor of the total number of processes (-np). Moreover, we've known that the k-points during the calculation will be subpartitioned into several groups which are called pools. In the Training course, the lecturer had suggested us to find out the number of k-points from the input and output file so that we can start to test the parameter. Therefore we analyzed the file and finally found out the number of k-points in this task is 20. It's worth to mention that when the number of pools(-npool) becomes greater than 20, the output file has a message which warns us that "Some Nodes have no K-points".

  By our assumption, the message is telling us the -npool parameter is not the optimal one which means some nodes were idle. In order to determine the better -npool, we set it to 16, 20 and 24 which is smaller, equal and greater than the number of K-points for comparison respectively.

  Additionally, we took the number of Nodes into consideration. As shown in Figure 1.1 , no matter how many Nodes perform in the task, -npool equals the number of K-points(20) has the better result among the others. Hence, we set the -npool to 20 and start to adjust other parameters.

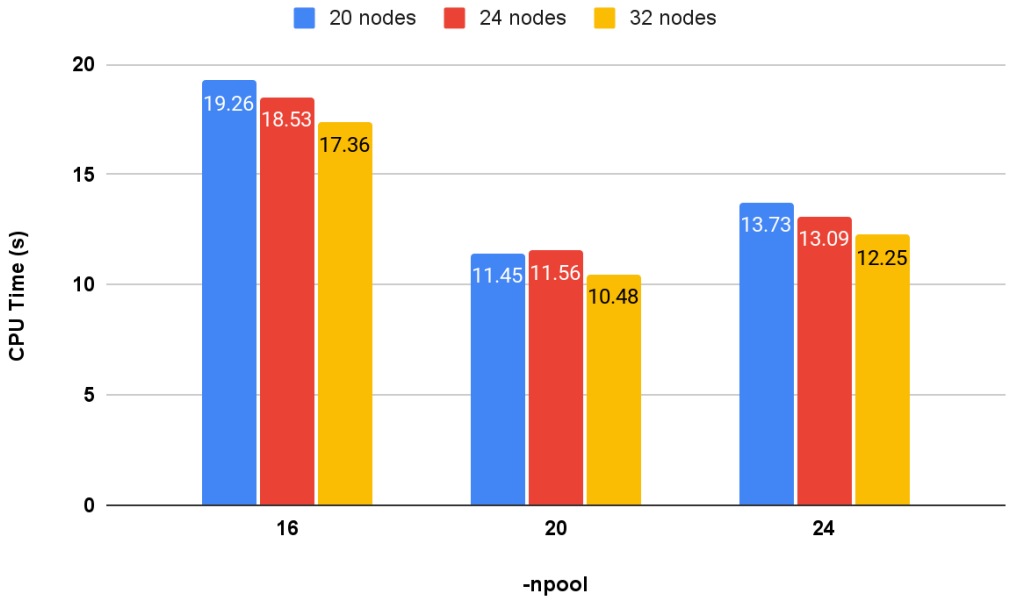

Figure 1.1

- Diagonalization (-ndiag)

  By the definition, all the arrays during the calculation are distributed block-like across the "linear-algebra group", a subgroup of the pool of processors, organized in a square 2D grid. Therefore, the -ndiag parameter needs to be n^2, where n is an integer less than $(\frac{-np}{-npool})$.

- Number of OpenMP Threads (OMP_NUM_THREADS)

  In order to test the parameter, we have set OMP_NUM_THREADS parameters to 1, 2, 3 and 4 threads to compare the difference and analyze its effect on the performance. After comparing the execution time, the larger number of OMP_NUM_THREADS be given, the longer CPU Time we get. Moreover, in the case when OMP_NUM_THREADS equals 3 or 4, the CPU time will become apparently larger than 1 or 2. Therefore, we set OMP_NUM_THREADS to 1 and start to tune the other parameters.

Result

- Single-Node (OMP_NUM_THREADS=1)
  - Result of Gadi Module

#CPUs (-np)	#pools (-npool)	#linear algebra groups (-ndiag)	CPU Time [s]
48	24	4	1m53.14s
48	24	1	1m53.54s
40	20	4	1m52.79s
40	20	1	1m52.94s

  - Result of Intel Compiler + IntelMPI (Compiled by Ourselves)

#CPUs (-np)	#pools (-npool)	#linear algebra groups (-ndiag)	CPU Time [s]
48	24	4	1m58.92s
48	24	1	1m59.00s
40	20	4	1m56.94s
40	20	1	1m57.08s

- Multi-Node (OMP_NUM_THREADS=1)
  - Result of Gadi Module

#CPUs (-np)	#pools (-npool)	#linear algebra groups (-ndiag)	CPU Time [s]
1200	20	16	9.74s

Optimization

- Number of Processes (-np)

  We set the number of processes to 960, 1200, 1440 and 1536 then test for their CPU Time. As Figure 1.2 showed, when the number of processes equal to 1200 will have better performance. Since we were curious about why 1200 processes can have less CPU Time than the others, we tried to look at the output files to find some clues.

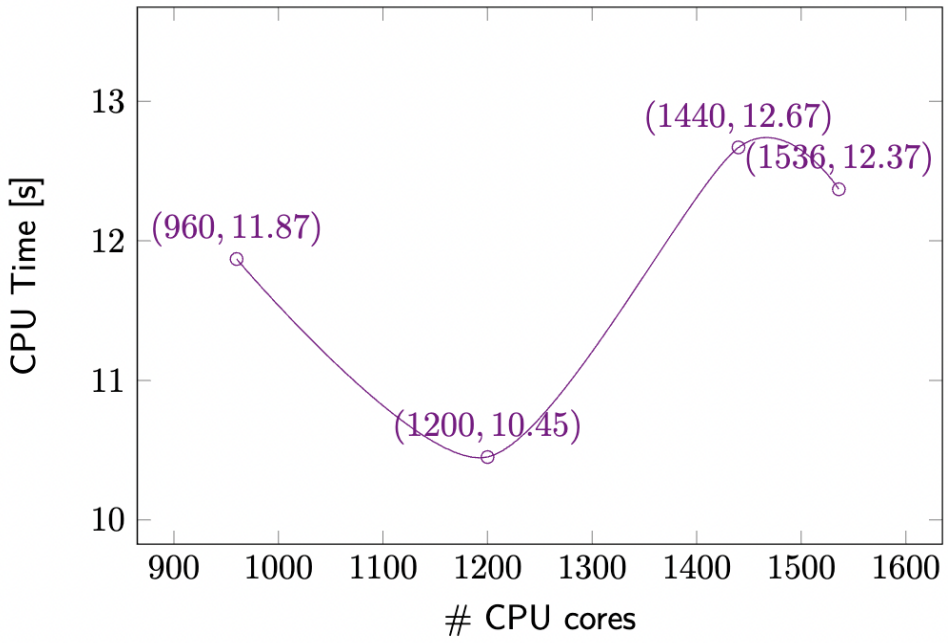

Figure 1.2

- Iterations During the Calculation

  In the output files, we found the number of iterations might have effects on the CPU Time. Hence, we compared the output file of 960, 1200, 1440 and 1536, then we made the following Figure 1.3. In Figure 1.3, since 1200 processes have fewer number of iterations than the others, we suppose that the less number of iterations might lead to lower CPU Time.

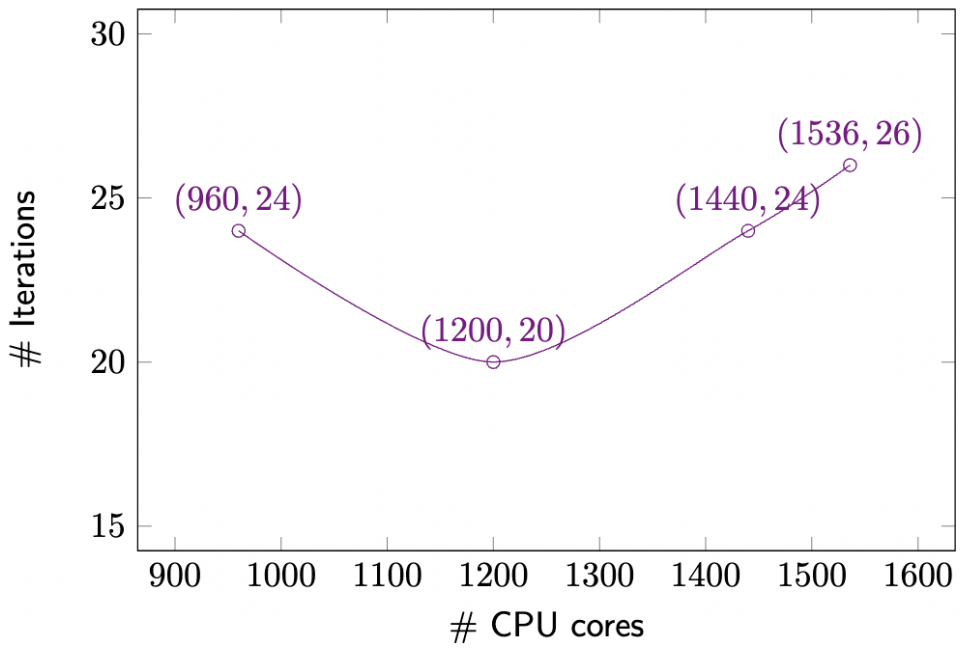

Figure 1.3

- Diagonalization (-ndiag)

  Since we thought 960, 1200 processes might generate the better result, we tried all possible -ndiag parameters among them and each took five tests then averaged the result. In Figure 1.4, we took 1440 processes as a control group. Because the data shows that the number of diagonalization between 2^2 to 6^2 had hit a trough, so we took them into the further discussion.

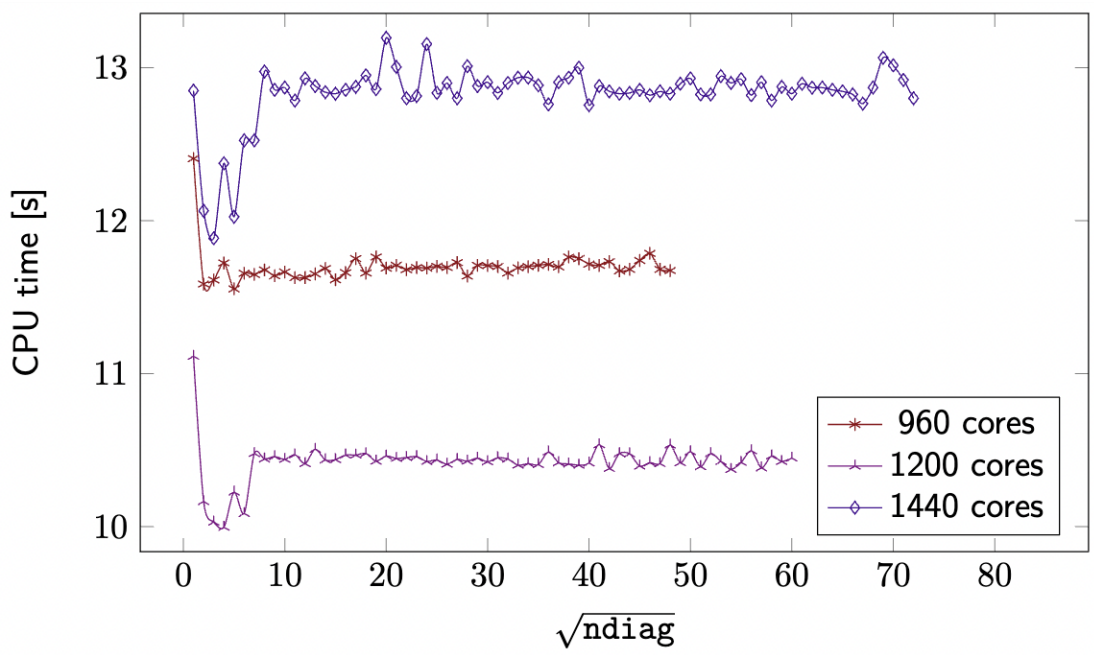

Figure 1.4

We tested each number of diagonalization among 2^2 to 6^2 for 12 times, and get the average of their CPU Time result. As shown in Figure 1.5, when the number of diagonalization equal 4^2, it will generate the best CPU Time.

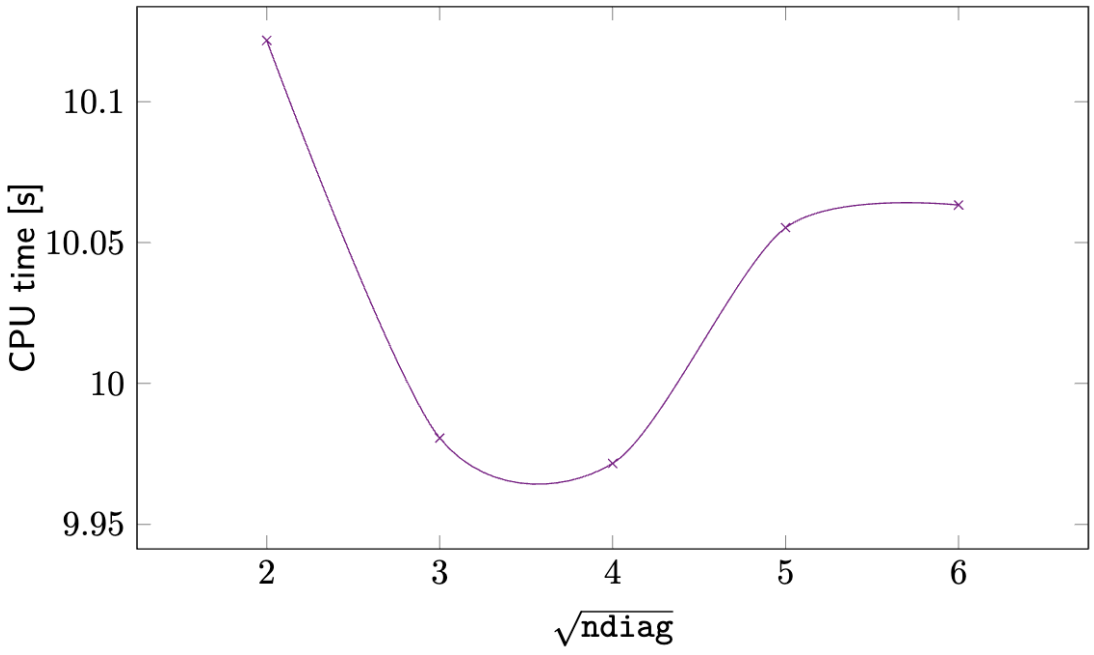

Figure 1.5

Additional Supplement

While we used the QE-7.0 module on Gadi to obtain the best performance for this task, we also compiled QE-7.0 by ourselves to compare the difference. Therefore, we cloned the QE-7.0 project from GitHub, then compiled it by using Intel Compiler and IntelMPI. Since the Intel Compiler has some parameters such as the “-fast” flag , we can use them to optimize the program.

After building up QE, we started to compare the performance to find out the difference. Then we found out something tricky between the Gadi module and the one compiled by ourselves. As shown in Figure 1.6, QE compiled by Intel Compiler with IntelMPI will have the better performance when -np equals to 960 instead of 1200 like the QE module on Gadi. Though the performance is not bad when using Intel Compiler with IntelMPI, the QE module on Gadi is still the better one. Hence, we decided to use the QE module on Gadi to perform the task.

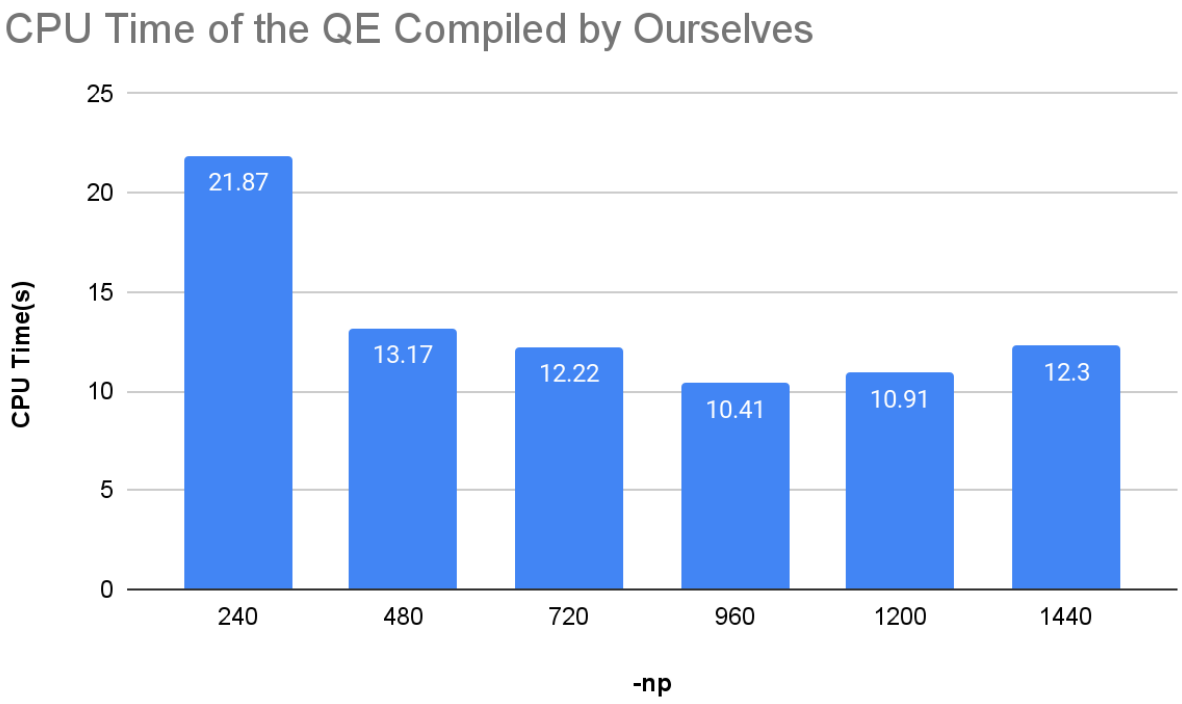

Figure 1.6

Execution

- Gadi Module version

  To reproduce the optimal result of single/multi node, you can run the following command.

```
$ qsub ./qe-single.sh
```

```
$ qsub ./qe-multi.sh
```

- Self-Compiled version

  Since we tested the QE which compiled by ourselves, we prepared a bash script to make sure you can reproduce the result.

  First, execute the *qe-script.sh* file, you will clone the project from GitHub and build it up.

```
$ bash ./qe-script.sh
```

  Secondly, write and submit the PBS file. Remember to change the OpenMPI with IntelMPI. It's also important that you need to modify the path of *pw.x* to the directory you installed the QE.

```
$ bash ./{Your Script}
```