12-17-2014

Accelerated data reading and analysis through high performance computing

HPC1 PROJECT REPORT

By: MOHAMMAD ATIF FAIZ AFZAL

Table of Contents

Accelerated data reading and analysis through high performance computing

HPC course project by Mohammad Atif Faiz Afzal

Course instructor: Dr. Matthew Jones

12/17/2014

1. Summary

In our research group, one of the projects is to discover trends in data derived from different quantum chemical methods. The work focuses on the analysis and comparison of results of various molecular properties derived from different flavors of Density Functional Theory (DFT). These patterns in data have significant implications for the utility of the employed approximations and the design of new quantum chemical techniques. Different molecular properties of about 10 million molecular candidates has been evaluated in the Harvard Clean Energy Project (HCEP). Each of these molecular properties are determined using 9 different methods of DFT. Correlations between these methods have been determined in the current project. Initially, linear regression models were used to connect results from different methods and it was observed that these models do not implement parallel programming and thus were taking very long times to determine a single correlation. As there are many flavors, large number of correlations are required and thus the overall time of computing is very high. As part of HPC project, I have determined these correlations using gradient descent method and implemented parallel gradient descent to increase the performance by reducing the computation time. The code developed also include parallel data reading and determines multiple correlations simultaneously. The code is written in Python programming language and the parallel computing is implemented using MPI4Py package.

2. Background

2.1. Comparison of quantum chemical methods

Computational techniques have been a tremendous help in discovering new compounds that can be particularly useful for chemical applications. For example, many new drugs that are used for medicinal purposes have sprouted from the use of computational techniques, like molecular simulations and modeling. Many scientists are using computer systems for data mining, to find in the massive amounts of data available, the main point that is relevant to one's current efforts or to recover experiments done in the past. Computational techniques based on quantum chemistry, is extensively used in our graduate group to calculate certain properties of a compound. Quantum chemistry computations can be performed by various different flavors that are available.

Density functional theory (DFT) is a quantum chemical method that has found a huge resurgence in popularity over the past 20 years and is still the state-of-the-art in the quantum chemistry community [1, 2]. The low computational cost, combined with useful accuracy, has made DFT a customary technique in most branches of chemistry and physics. Electronic structure problems in a variety of fields are currently being tackled by DFT [3]. However, DFT has many limitations in its present form: too many approximations (flavors), failures for strongly correlated systems, etc [4]. This research involves finding correlations between computational inexpensive DFT flavors, with more expensive and accurate DFT flavors. Having these correlations published and accessible by the world can dramatically decrease the amount of time spent conducting future simulations and modelling of chemical compounds. This project is tremendously helpful in accordance to future work by Dr. Hachmann's Research group.

My graduate research is currently related to using quantum chemical techniques to find the optical properties of different organic polymers for use in the optoelectronics field. These quantum chemical techniques use expensive DFT flavors that can take multiple days to run for a single organic polymer. Finding correlation between different DFT flavors could dramatically cut down the time required to run many of these calculations; by mapping expensive DFT flavor to a much cheaper flavor while maintaining comparatively accurate results. In the current project, these correlations are found out using supervised machine learning technique called gradient descent.

The huge data for the finding these correlations is obtained from Harvard Clean Energy Project (CEP).

2.2. Big Data from Harvard Clean Energy Project

The Harvard Clean Energy Project (CEP) is an initiative which started out with the aim of discovering next generation of organic solar cell materials [5]. Carbon-based solar cells are being emerged as an interesting alternative to the conventional silicon-based solar devices. Organic photovoltaics range from crystalline small molecule approaches to amorphous polymers (plastics). These organic solar cell devices promise simple, low-cost, and high-volume production and have great potential in the prospect of merging the unique mechanical stability, flexibility and versatility of plastics with electronic features. OPVs have attracting properties such as easy processability, are light-weight, and can essentially be molded into any desired shape. Further, they can also be made as transparent and/or colored for special applications such as solar windows. The resulting combination of availability and applicability make OPVs prime candidates to achieve the ubiquitous harvesting of solar energy, with building integrated and ultra-portable applications as foremost targets.

The project, CEP, is a theory-driven search for the next generation OPV materials. In this project an automated framework was developed which studies the potential molecular motifs using *first-principles* quantum chemistry. Millions of molecular candidates were characterized using 9 different DFT flavors. Different molecular properties such as HUMO energy, LUMO energy, dipole moment etc., were determined using these different DFT flavors. The results were compiled in a reference database and are made available for public use. In the current project, the data from the CEP reference database will be used to find correlations between the different DFT flavors via supervised machine learning techniques such as gradient descent. Figure 1 shows some of the correlations that were determined in this project. These plots show linear correlations between HOMO energies determined from different quantum chemical methods.

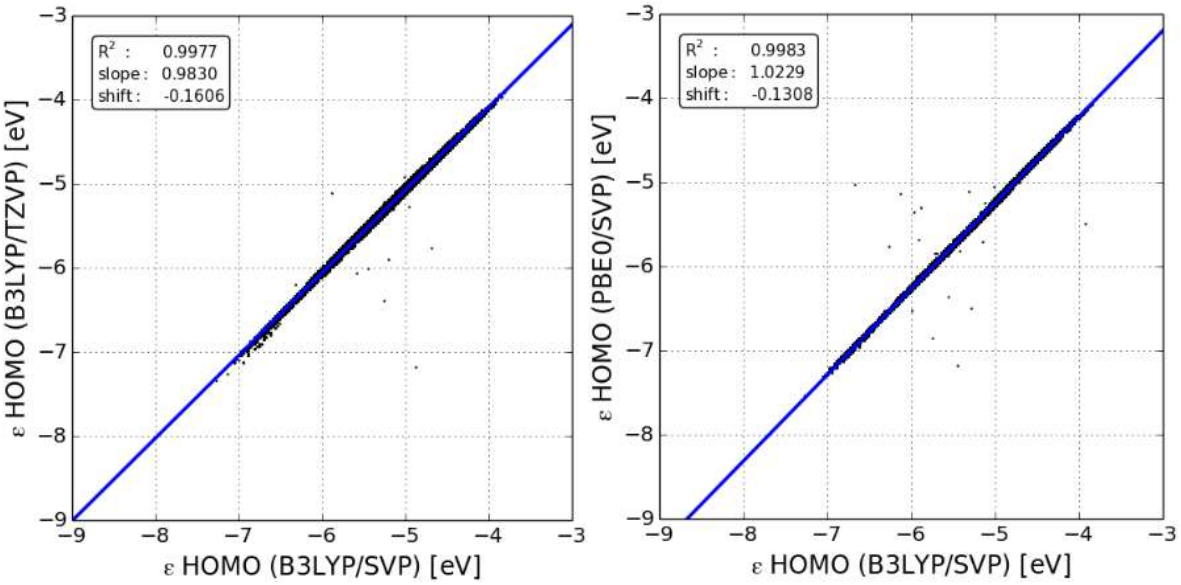

Figure 1: Linear correlation between different quantum chemical methods

2.3. Gradient descent

Gradient descent method is an algorithm to determine local minimum of functions. In this algorithm, we start with an initial guess of the solution and determine the gradient of the function. Then a step is taken in the negative direction of the gradient and the gradient is determined at the new solution. This process is iteratively repeated until the solution converges, i.e. when the gradient reaches a value of zero, which corresponds to the point of local minimum. The gradient descent is a first-order algorithm because in this method we only take first derivative of the function.

When trying to fit a set of data to a function, an error function is considered. The error function, $J(\theta)$, can be a squared error function as shown in the equation *(1)*, where $h_\theta\left(x^{(i)}\right)$ is the value of the fitting equation at $x^{(i)}$, $y^{(i)}$ is the response data point and m is total number of data points. Gradient descent method can be used to minimize this error function. In each step of the gradient descent iteration a new solution is updated as shown in the equation *(2)*, where θ^{n+1} is the updated solution, θ^n is value from iteration n and α is the learning rate. This step is repeated until the convergence of the solution is obtained. When a linear function is used for the fitting function $h_\theta\left(x^{(i)}\right)$, the equation *(2)* is reduced to the equation *(3)*.

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2 \tag{1}$$

$$\theta^{n+1} := \theta^n - \alpha\frac{\partial}{\partial\theta}\left(J(\theta)\right) \tag{2}$$

$$\theta^{n+1} := \theta^n - \alpha\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right).x^{(i)} \tag{3}$$

The steps taken while minimizing the error function can be seen qualitatively in the Figure 2. It can be seen that in each step the solution moves towards the local minima (red circle in the Figure 2). The rate at which the solution moves towards the local minima is dependent on the value of α. α (>0) is generally kept a small number so that the algorithm makes small steps and ensure that the algorithm is stable. Smaller the value of α, more stable the convergence and more likely to obtain the minima. When higher α is used, especially for a large data set, it is very probable that the solution might diverge.

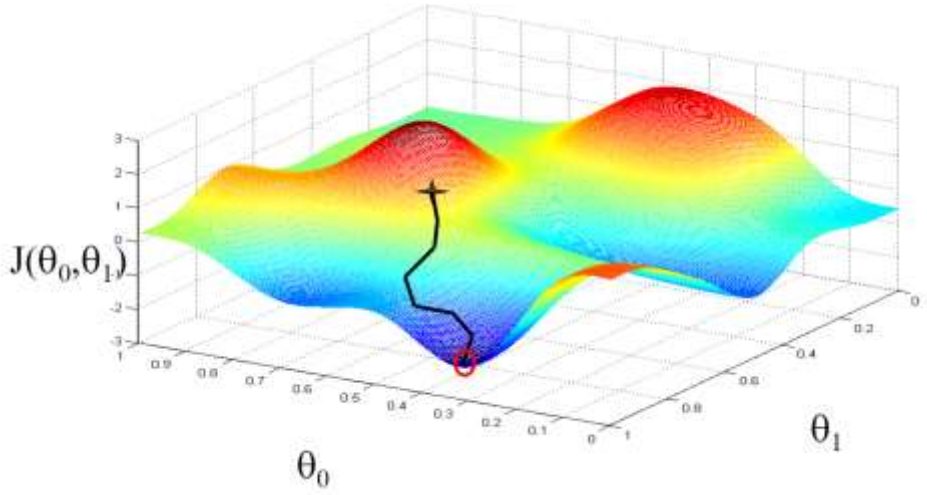

Figure 2: Illustration of the gradient descent.

3. Parallel implementation

3.1. Parallel gradient descent

For a gradient descent with t iterations, the computation time can be represented as equation *(4)*, where m is number of samples, T1 is the computation time to process a single sample data point, and T2 is the computation time to update the parameters. Normally the number of data points is more than the number of parameters. For example, if there are 10 million data points, i.e. m=10 million, then computation time is very large. So, if there are large number of data set, gradient descent becomes very expensive in terms of computation. One way to reduce to the cost of computation is to implement parallel computing in the gradient descent algorithm.

$$Time = t \times (T1 \times m + T2) \quad (4)$$

Parallel gradient descent can be obtained by diving the summation of the gradients as shown in the equation *(3)* into the number of available processors. The master processor will split the data into the available workers (processors) and send the corresponding data to each worker. The workers will then calculate the sum of the gradients and then send the sum value back to the master worker. The master worker will sum up the local gradients and update the weight for new solution and then the same process is repeated iteratively until the convergence is obtained. This can be represented in flow chart as shown in the Figure 3. The error in the gradient descent in each iteration can also be calculated parallelly in a similar fashion as the sum of gradients.

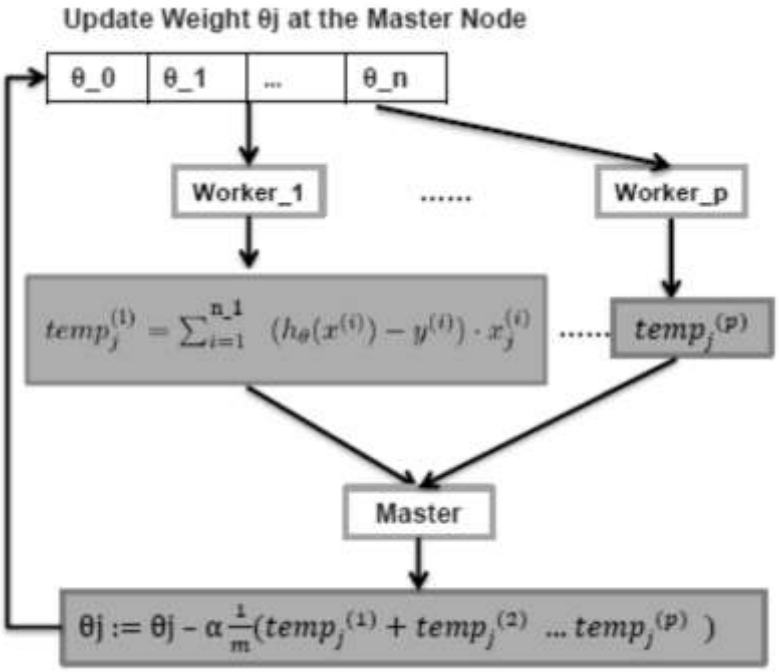

Figure 3: Flow chart showing parallel implementation of gradient descent

3.2. Parallel data reading

In the profiling of the code, it was observed that the amount of time taken to read the data in a master processor and sending it to the workers is also expensive. Therefore, I thought of reading the data in parallel fashion rather than reading the whole data from file by a single processor.

Data reading from a file is done using pointers. In the current project, the data is available in a dat file with each row representing one molecular candidate and the columns giving the properties from different DFT flavors. Because there is data in a single row in the file, reading each line will take some computation time. If there are very large number of lines, the number of lines can be split into the number of processors. Then each worker depending on its worker number jumps the pointer directly to the corresponding line in the file and the reads only the assigned number of lines. By this way of parallel reading, the computation time can be significantly reduced.

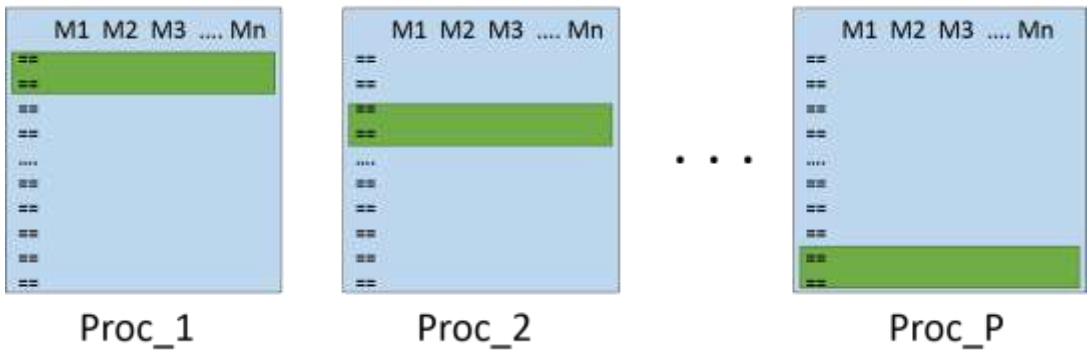

Figure 4: Parallel data reading in different processors

3.3. Synergistic model: Simultaneous data reading and gradient descent

When parallelizing gradient descent, the data has to be split and sent into individual workers. If the master workers stores and distributes data to individual workers, the time spent in reading and communicating the data would be very high for large amount of data. To avoid such scenario, a synergistic model is developed. Since, individual workers do calculations independently based on their individual assigned data, the only information that has to be communicated is the gradient sum and the updated solutions after each iteration. Exploiting this fact of the gradient descent, data can be read in parallel fashion while performing parallel gradient descent. The individual workers would extract the data simultaneously from a single data file, and perform the required gradient descent calculations. In this way, the time spent in data reading in master worker and sending data to the individual workers can be reduced.

4. Method used

All the codes in the project are written in python programming language. The MPI package for python called MPI4Py is used in all the runs. All the computations were performed on CCR nodes and the jobs were submitted using SLURM. Only one type of node that is available on CCR: 8 core node (CPU-L5520) is used in all of the computations. The computation time in all the runs was determined using MPItime.

Parallel speedup is calculated by the formula

$$Parallel\ Speedup = \frac{Time\ for\ sequential\ running}{Time\ for\ running\ on\ N\ procs}$$

Parallel efficiency is calculated by dividing the parallel speedup with the no. of processors (P) used.

$$Parallel\ Efficiency = \frac{Time\ for\ sequential\ running}{Time\ for\ running\ on\ N\ procs} \cdot \frac{1}{P}$$

The learning rate of 0.01 is selected to make sure that the gradient descent is stable and there are no divergence issues encountered. To determine the number of iterations required, initial runs were done using sequential code to determine the dependence of error on the iteration number. Figure 5 shows the error variation with the iteration number when the learning rate is 0.01. It can be seen from the plot that the error becomes constant after about 400 iterations and thus, there is no improvement in the solution after 400 iterations. This means that the local minima is obtained in about 400 iterations. To make sure the convergence is obtained for all the other data sets, 1000 iterations would be a safe number to perform. Thus, a learning rate of 0.01 and the total number of iteration of 1000 were used in all of my calculations unless otherwise mentioned.

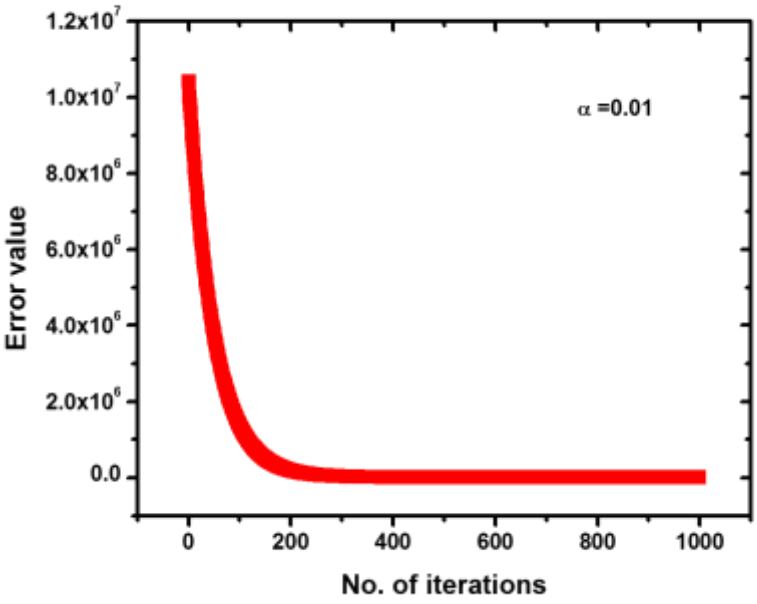

Figure 5: Error function variation with the number of iterations

5. Results

Parallel data reading

To check if we can read the data in parallel and reduce the computation time, a code was written only to see the efficiency of data reading in parallel. The code is written in python and is attached in the appendix. The performance was determined using the L5520 nodes and shown in the Figure 6. It can be seen that the computation time decreases with increasing the number of processors. The speedup keeps on increasing, however, the efficiency keeps decreasing. In this case there is no communication between the processors, therefore there are no losses and that's why we see continuous increase in the speedup.

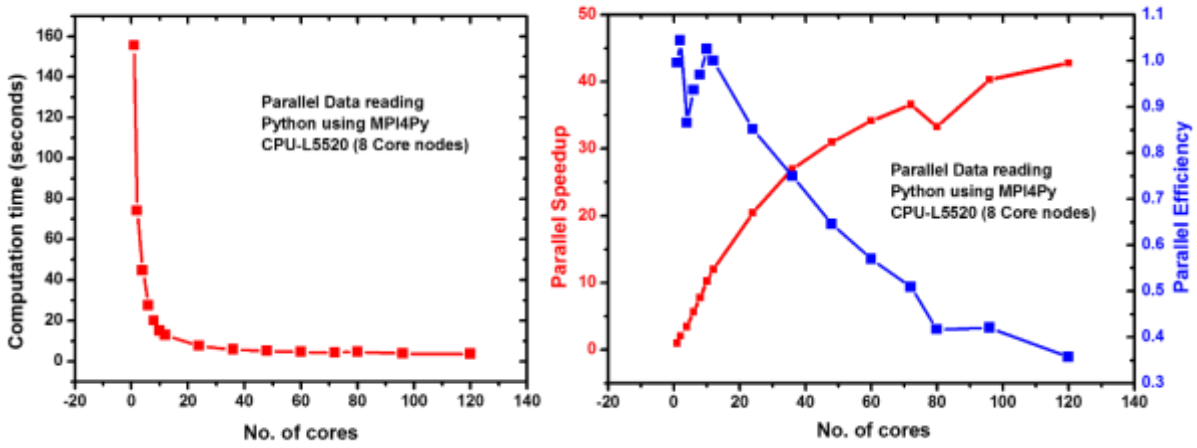

Figure 6: Parallel data reading using MPI4Py on L5520 nodes

Parallel gradient descent using the synergistic model

A parallel code that implements parallel data reading and parallel gradient descent is written in python and the performance of the code was measured with respect to the number of processors used. Figure 7 shows the performance of parallel gradient descent for 1000 iterations using 10 L5520 nodes (8 core nodes) making a total of 80 cores. To check if all the cores are being used, slurmjobvis command of CCR was used when the code was running on 96 total cores. A screenshot of the result can be seen in Figure 8. It can be seen from the Figure 7 that the computation time decreases until 12 cores and then increases with the number of cores. The increase in the computation time after 12 cores is because the communication time dominates the processing time.

This is probably because the code is written in python and the python platform on CCR do not use the infiniband connection. Whereas, Fortran and C implementation on CCR use the infiniband connections between the nodes. Python platform when run on CCR uses the ethernet connection between CCR nodes. A maximum speedup of approximately 7 is observed when 12 cores are used whereas the efficiency keeps on decreasing. The efficiency decreases very rapidly when more than 12 cores are used and approaches zero when higher number of cores are used.

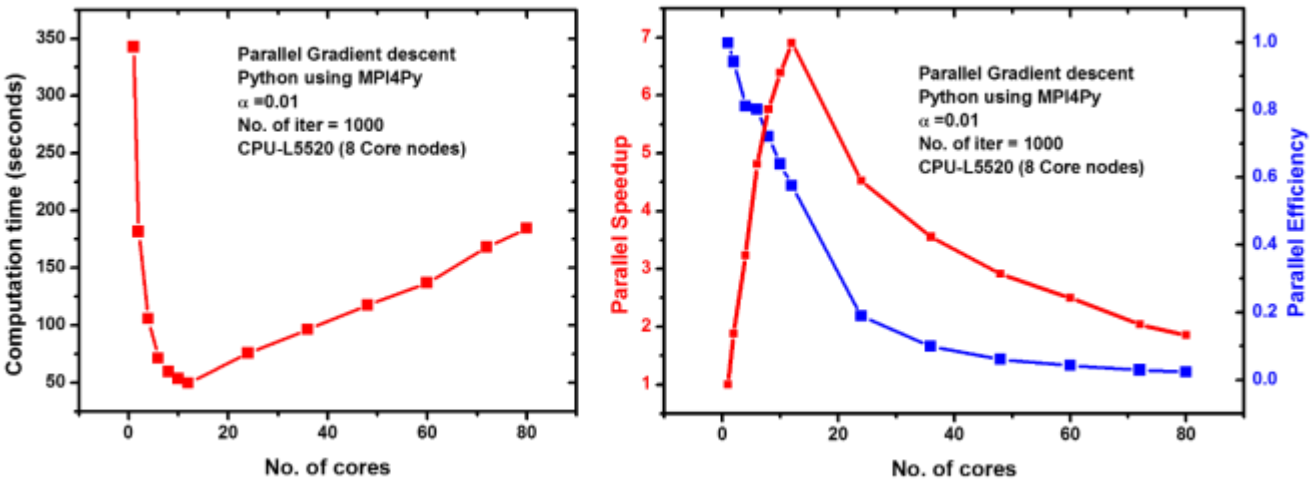

Figure 7: Parallel gradient descent (1000 iterations) performance using MPI4Py on L5520 nodes for a single correlation

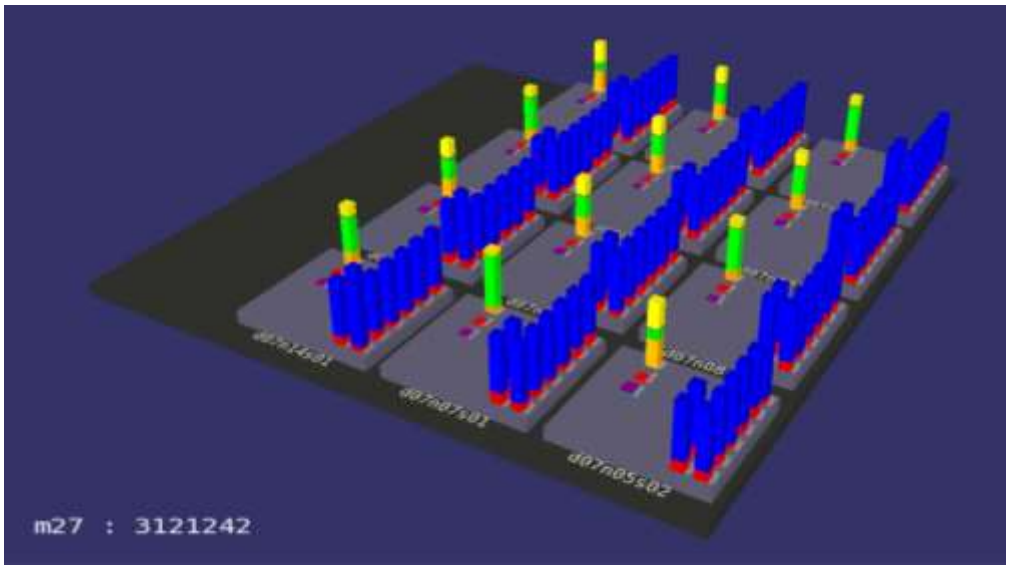

Figure 8: Work load on processors showing that all the 8 core nodes being used when MPI is implemented

To check the effect of the number of iteration on the speedup, the code was run for 500 iterations and the performance is shown in Figure 9. It can be observed that the speedup in this case increases up to approximately 8 and then decreases. This maximum speedup is observed when 12 cores are used. Thus, when higher number of iterations are required, it is probable that more speedup can be obtained when same number of cores are used.

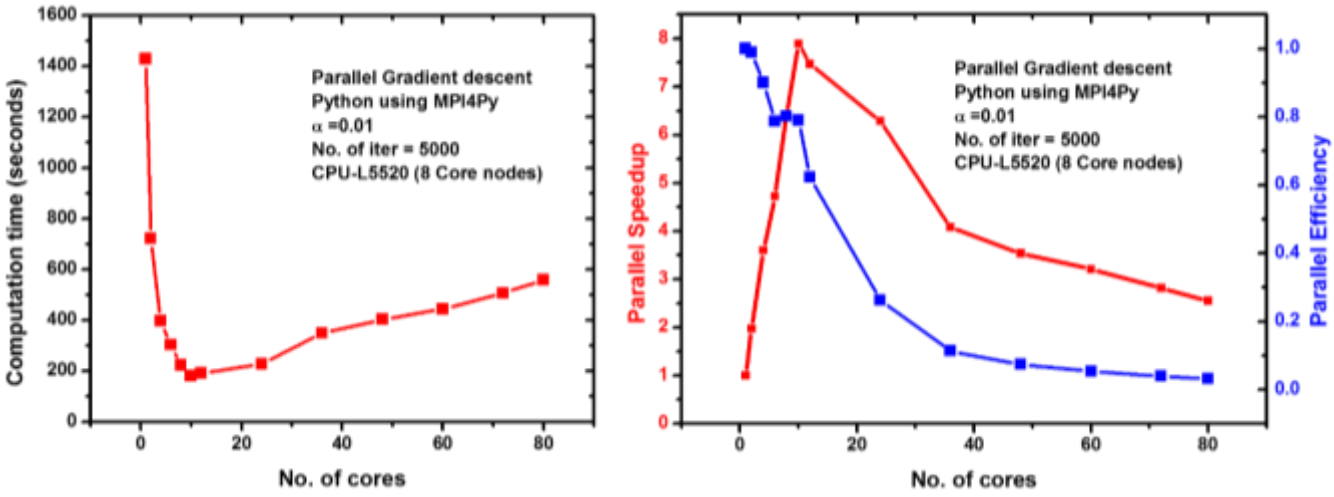

Figure 9: Parallel gradient descent (5000 iterations) performance using MPI4Py on L5520 nodes for a single correlation

It is very convenient when a single run can give correlations between different methods. So, a code which computes multiple correlations using gradient descent was written. The code is written in python and is attached in the appendix. This code contains all the flavors for a particular molecular property that are described in this project and is attached in the appendix. To determine the parallel performance of this code the code was run on multiple L5520 nodes and the performance is shown in Figure 10. It can be seen that computation time decreases up until 24 cores and then increases very slowly upon addition of more cores. This shows a better performance than when only one correlation was computed. Thus, increasing the size of the problem leads to better performance in parallel gradient descent. The same can be interpreted using the parallel speedup. The maximum speedup of approximately 12 was observed when 24 cores are used compared. Therefore, this code which computes 8 different correlations is computationally a better way than computing individual correlations.

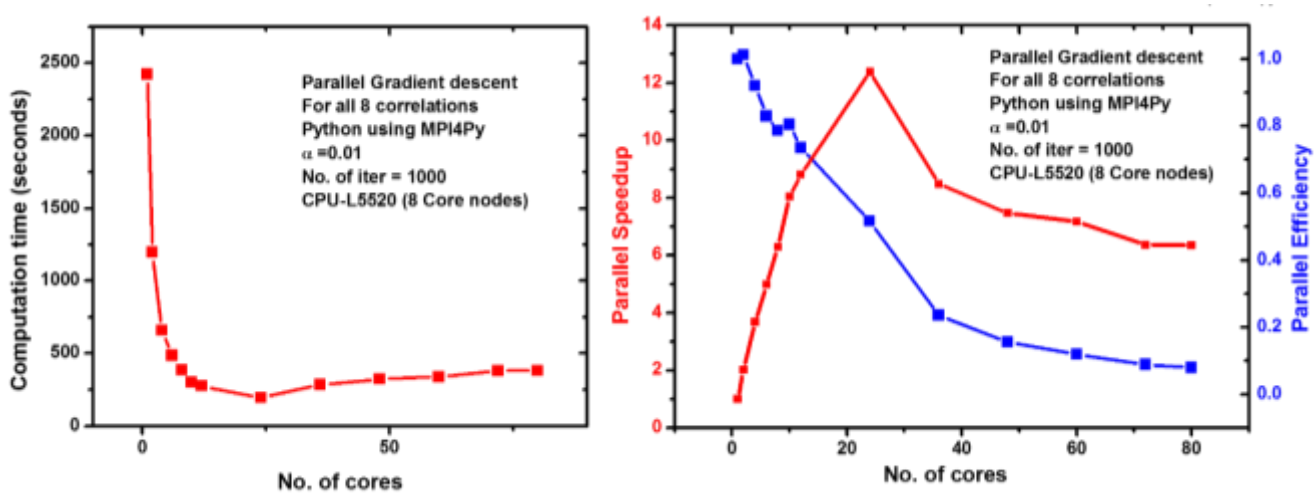

Figure 10: Parallel gradient descent (1000 iterations) performance using MPI4Py on L5520 nodes for a single correlation

6. Impact of this project and future directions

Using high performance computing, the computation time in finding correlations between millions of data can be dramatically decreased. This project shows that it is possible to perform accelerated data reading and analysis. The parallel implementation demonstrated in this project will be an important tool in our research group, because most of the research in our group is based on Big Data.

Currently, I have only implemented gradient descent to find linear correlations between different methods. However, some correlations are not linear and need more advanced tools to find correlations between these methods. In future, one aspect of my graduate research is to develop tools to find more reliable non-linear correlations between the data generated using different quantum chemical methods. I believe that this project will be a vital tool throughout my research. This project has provided me with more insight into high performance computing tools in Python and made me appreciate the power of high performance computing.

7. References

1. Dreizler, R. M.; Engel, E., *Density functional theory*. Springer: 2011.
2. Kohn, W.; Becke, A. D.; Parr, R. G., Density functional theory of electronic structure. *Journal of Physical Chemistry* 1996, *100* (31), 12974-12980.
3. Gonze, X.; Amadon, B.; Anglade, P. M.; Beuken, J. M.; Bottin, F.; Boulanger, P.; Bruneval, F.; Caliste, D.; Caracas, R.; Cote, M.; Deutsch, T.; Genovese, L.; Ghosez, P.; Giantomassi, M.; Goedecker, S.; Hamann, D. R.; Hermet, P.; Jollet, F.; Jomard, G.; Leroux, S.; Mancini, M.; Mazevet, S.; Oliveira, M. J. T.; Onida, G.; Pouillon, Y.; Rangel, T.; Rignanese, G. M.; Sangalli, D.; Shaltaf, R.; Torrent, M.; Verstraete, M. J.; Zerah, G.; Zwanziger, J. W., ABINIT: First-principles approach to material and nanosystem properties. *Computer Physics Communications* 2009, *180* (12), 2582-2615.
4. Cohen, A. J.; Mori-Sánchez, P.; Yang, W., Insights into current limitations of density functional theory. *Science* 2008, *321* (5890), 792-794.
5. Hachmann, J.; Olivares-Amaya, R.; Atahan-Evrenk, S.; Amador-Bedolla, C.; Sánchez-Carrera, R. S.; Gold-Parker, A.; Vogt, L.; Brockway, A. M.; Aspuru-Guzik, A. n., The Harvard Clean Energy Project: Large-Scale Computational Screening and Design of Organic Photovoltaics on the World Community Grid. *The Journal of Physical Chemistry Letters* 2011, *2* (17), 2241-2251.

Appendix

Code written for HPC project
Reading the data in parallel- sample checking code

```
import numpy as np
from numpy import arange,array,ones,linalg
from mpi4py import MPI

wt1 = MPI.Wtime()
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
mpisize = comm.Get_size()

def is_number(s):
   try:
      float(s)
      return s
   except ValueError:
      return 0

infile=open('dft_vs_dft_elumo_dump.dat')

num_lines = sum(1 for line in infile)
print num_lines

start=rank*(num_lines-2)/mpisize+1
end=(rank+1)*(num_lines-2)/mpisize
lines=end-start+1
print rank, start,end,lines

x1=np.zeros(lines)
x2=np.zeros(lines)
x3=np.zeros(lines)
x4=np.zeros(lines)
x5=np.zeros(lines)
x6=np.zeros(lines)
x7=np.zeros(lines)
x8=np.zeros(lines)
x9=np.zeros(lines)

infile=open('dft_vs_dft_elumo_dump.dat')

for _ in xrange(start+1):
```

```
    next(infile)

for i,line in enumerate(infile):
    if i==(lines):
        break
    values = line.strip().split(',')
    x1[i] = is_number((values[2]))
    x2[i] = is_number((values[3]))
    x3[i] = is_number((values[4]))
    x4[i] = is_number((values[5]))
    x5[i] = is_number((values[6]))
    x6[i] = is_number((values[7]))
    x7[i] = is_number((values[8]))
    x8[i] = is_number((values[9]))
    x9[i] = is_number((values[10]))

wt2 = MPI.Wtime()

if rank==0:
    print mpisize,wt2-wt1
```

Code written for HPC project

Python code for parallel gradient descent

```
import numpy as np
from numpy import arange,array,ones,linalg
from mpi4py import MPI

wt1 = MPI.Wtime()
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
mpisize = comm.Get_size()

def is_number(s):
   try:
      float(s)
      return s
   except ValueError:
      return 0

infile=open('dft_vs_dft_elumo_dump.dat')

num_lines = sum(1 for line in infile)

start=rank*(num_lines-2)/mpisize+1
end=(rank+1)*(num_lines-2)/mpisize
lines=end-start+1

x1=np.zeros(lines)
x2=np.zeros(lines)
x3=np.zeros(lines)
x4=np.zeros(lines)
x5=np.zeros(lines)
x6=np.zeros(lines)
x7=np.zeros(lines)
x8=np.zeros(lines)
x9=np.zeros(lines)

infile=open('dft_vs_dft_elumo_dump.dat')

for _ in xrange(start+1):
   next(infile)

for i,line in enumerate(infile):
```

```
    if i==(lines):
        break
    values = line.strip().split(',')
    x1[i] = is_number((values[2]))
    x2[i] = is_number((values[3]))
    x3[i] = is_number((values[4]))
    x4[i] = is_number((values[5]))
    x5[i] = is_number((values[6]))
    x6[i] = is_number((values[7]))
    x7[i] = is_number((values[8]))
    x8[i] = is_number((values[9]))
    x9[i] = is_number((values[10]))

X_norm=np.array([np.ones(lines),x1])
X_norm=X_norm.transpose()

alpha = 0.01

theta1= np.ones(2)
theta2= np.ones(2)
theta3= np.ones(2)
theta4= np.ones(2)
theta5= np.ones(2)
theta6= np.ones(2)
theta7= np.ones(2)
theta8= np.ones(2)

def sums(x, y, theta, m):
    xTrans = x.transpose()
    hypothesis = np.dot(x, theta)
    loss = hypothesis - y
    gradient = np.dot(xTrans, loss) / m
    return gradient

def errors(x, y, theta, m):
    hypothesis = np.dot(x, theta)
    loss_sq = sum((hypothesis - y)**2)
    return loss_sq

for i in xrange(1000):

    grad_sum1= sums(X_norm, x2, theta1, num_lines)
    grad_sum2= sums(X_norm, x3, theta2, num_lines)
    grad_sum3= sums(X_norm, x4, theta3, num_lines)
```

```
grad_sum4= sums(X_norm, x5, theta4, num_lines)
grad_sum5= sums(X_norm, x6, theta5, num_lines)
grad_sum6= sums(X_norm, x7, theta6, num_lines)
grad_sum7= sums(X_norm, x8, theta7, num_lines)
grad_sum8= sums(X_norm, x9, theta8, num_lines)

error1 = errors(X_norm, x2, theta1, num_lines)
error2 = errors(X_norm, x3, theta2, num_lines)
error3 = errors(X_norm, x4, theta3, num_lines)
error4 = errors(X_norm, x5, theta4, num_lines)
error5 = errors(X_norm, x6, theta5, num_lines)
error6 = errors(X_norm, x7, theta6, num_lines)
error7 = errors(X_norm, x8, theta7, num_lines)
error8 = errors(X_norm, x9, theta8, num_lines)

if rank==0:
   totals1 = np.zeros_like(grad_sum1)
   totals2 = np.zeros_like(grad_sum2)
   totals3 = np.zeros_like(grad_sum3)
   totals4 = np.zeros_like(grad_sum4)
   totals5 = np.zeros_like(grad_sum5)
   totals6 = np.zeros_like(grad_sum6)
   totals7 = np.zeros_like(grad_sum7)
   totals8 = np.zeros_like(grad_sum8)

   total_e1=np.zeros(1)
   total_e2=np.zeros(1)
   total_e3=np.zeros(1)
   total_e4=np.zeros(1)
   total_e5=np.zeros(1)
   total_e6=np.zeros(1)
   total_e7=np.zeros(1)

else:
   totals1 = None
   totals2 = None
   totals3 = None
   totals4 = None
   totals5 = None
   totals6 = None
   totals7 = None
   totals8 = None

   total_e1=None
```

```
    total_e2=None
    total_e3=None
    total_e4=None
    total_e5=None
    total_e6=None
    total_e7=None
    total_e8=None

comm.Reduce([grad_sum1, MPI.DOUBLE],[totals1, MPI.DOUBLE],op = MPI.SUM,root = 0)
comm.Reduce([grad_sum2, MPI.DOUBLE],[totals2, MPI.DOUBLE],op = MPI.SUM,root = 0)
comm.Reduce([grad_sum3, MPI.DOUBLE],[totals3, MPI.DOUBLE],op = MPI.SUM,root = 0)
comm.Reduce([grad_sum4, MPI.DOUBLE],[totals4, MPI.DOUBLE],op = MPI.SUM,root = 0)
comm.Reduce([grad_sum5, MPI.DOUBLE],[totals5, MPI.DOUBLE],op = MPI.SUM,root = 0)
comm.Reduce([grad_sum6, MPI.DOUBLE],[totals6, MPI.DOUBLE],op = MPI.SUM,root = 0)
comm.Reduce([grad_sum7, MPI.DOUBLE],[totals7, MPI.DOUBLE],op = MPI.SUM,root = 0)
comm.Reduce([grad_sum8, MPI.DOUBLE],[totals8, MPI.DOUBLE],op = MPI.SUM,root = 0)

comm.Reduce([error1, MPI.DOUBLE],[total_e1, MPI.DOUBLE],op = MPI.SUM,root = 0)
comm.Reduce([error2, MPI.DOUBLE],[total_e2, MPI.DOUBLE],op = MPI.SUM,root = 0)
comm.Reduce([error3, MPI.DOUBLE],[total_e3, MPI.DOUBLE],op = MPI.SUM,root = 0)
comm.Reduce([error4, MPI.DOUBLE],[total_e4, MPI.DOUBLE],op = MPI.SUM,root = 0)
comm.Reduce([error5, MPI.DOUBLE],[total_e5, MPI.DOUBLE],op = MPI.SUM,root = 0)
comm.Reduce([error6, MPI.DOUBLE],[total_e6, MPI.DOUBLE],op = MPI.SUM,root = 0)
comm.Reduce([error7, MPI.DOUBLE],[total_e7, MPI.DOUBLE],op = MPI.SUM,root = 0)
comm.Reduce([error8, MPI.DOUBLE],[total_e8, MPI.DOUBLE],op = MPI.SUM,root = 0)



if rank==0:
    theta1=theta1 - alpha*totals1
    theta2=theta2 - alpha*totals2
    theta3=theta3 - alpha*totals3
    theta4=theta4 - alpha*totals4
    theta5=theta5 - alpha*totals5
    theta6=theta6 - alpha*totals6
    theta7=theta7 - alpha*totals7
    theta8=theta8 - alpha*totals8
    totals1=0
    totals2=0
    totals3=0
    totals4=0
    totals5=0
    totals6=0
    totals7=0
```

```
        totals8=0


    theta1=comm.bcast(theta1, root=0)
    theta2=comm.bcast(theta2, root=0)
    theta3=comm.bcast(theta3, root=0)
    theta4=comm.bcast(theta4, root=0)
    theta5=comm.bcast(theta5, root=0)
    theta6=comm.bcast(theta6, root=0)
    theta7=comm.bcast(theta7, root=0)
    theta8=comm.bcast(theta8, root=0)


wt2 = MPI.Wtime()

if rank==0:
    print
'C1',theta1,'C2',theta2,'C3',theta3,'C4',theta4,'C5',theta5,'C6',theta6,'C7',theta7,'C8',theta8
    print 'Elapsed wall time = ',wt2-wt1
    print mpisize, wt2-wt1, theta1
```

Slurm script written for running parallel code

```
#!/bin/sh
#SBATCH --partition=general-compute
#SBATCH --time=26:00:00
#SBATCH --job-name="para_grad"
#SBATCH --output=para_grad_120core.out
#SBATCH --mail-user=m27@buffalo.edu
#SBATCH --mail-type=ALL

#SBATCH --nodes=15
#SBATCH --cpus-per-task=8

# ================================================
# For 8-core nodes
# ================================================
#SBATCH --constraint=CPU-L5520


tic=`date +%s`
echo "Start Time = "`date`
echo "Loading modules ..."

module load python
ulimit -s unlimited
module list

echo "SLURM job ID        = "$SLURM_JOB_ID
echo "Working Dir         = "`pwd`
echo "Compute Nodes       = "`nodeset -e $SLURM_NODELIST`
echo "Number of Processors = "$SLURM_NPROCS
echo "Number of Nodes     = "$SLURM_NNODES

# for i in $(seq 12 12 120); do
#     echo $i
#     mpiexec -np $i python readpara.py
# done

mpiexec -np 120 python readpara3.py
mpiexec -np 96 python readpara3.py
mpiexec -np 80 python readpara3.py
mpiexec -np 72 python readpara3.py
mpiexec -np 60 python readpara3.py
```

```
mpiexec -np 48 python readpara3.py
mpiexec -np 36 python readpara3.py
mpiexec -np 24 python readpara3.py
mpiexec -np 12 python readpara3.py
mpiexec -np 10 python readpara3.py
mpiexec -np 8 python readpara3.py
mpiexec -np 6 python readpara3.py
mpiexec -np 4 python readpara3.py
mpiexec -np 2 python readpara3.py
mpiexec -np 1 python readpara3.py


echo "All Done!"

echo "End Time = "`date`
toc=`date +%s`

elapsedTime=`expr $toc - $tic`
echo "Elapsed Time = $elapsedTime seconds"
```