PRACE

PARTNERSHIP FOR ADVANCED COMPUTING IN EUROPE

2018

summerofhpc.prace-ri.eu

A long hot summer is time for a break, right? Not necessarily! PRACE Summer of HPC 2018 reports by participants are here.

HPC in the summer?

Leon Kos

There is no such thing as lazy summer. At least not for the 23 participants and their mentors at 10 PRACE HPC sites.

Summer of HPC is a PRACE programme that offers university students the opportunity to spend two months in the summer at HPC centres across Europe. The students work using HPC resources on projects that are related to PRACE work with the goal to produce a visualisation or a video.

This year, training week was in Ostrava and it seems to have been the best training week yet! From MPI to Vectorization and good food, the week was a blast! It was a great start to Summer of HPC and set us up to have an amazing summer!

At the end of the summer videos were created and are available on Youtube as PRACE Summer of HPC 2018 presentations playlist. Together with the following articles interesting code and results are available. Dozens of blog posts were created as well.

At the end of the activity, every year two projects out of the 23 participants are selected and awarded for their outstanding performance. The winners of this year, Conor O'Mara and Sukhminder Singh, presented their experience at the award ceremony in Jülich Supercomputing Centre (JSC), Germany.

Therefore, I invite you to look at the articles and visit the web pages for details and experience the fun we had this year.

What can I say at the end of this wonderful summer. Really, autumn will be wonderful too. Don't forget to smile!

Contents

PRACE SoHPC2018 Coordinator
Leon Kos, University of Ljubljana
Phone: +386 4771 436 E-mail: leon.kos@lecad.fs.uni-lj.si

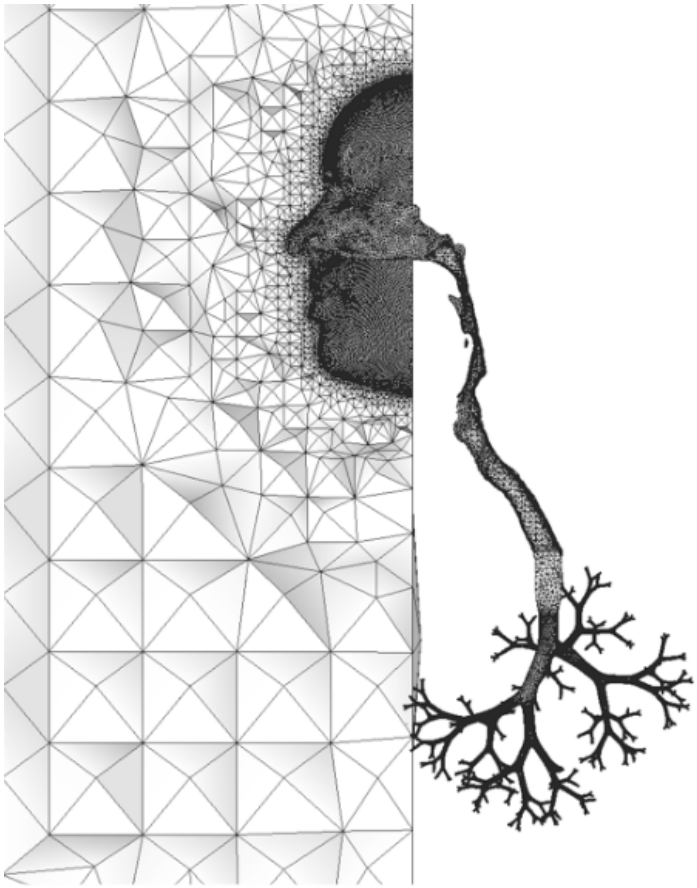
PRACE SoHPCMore Information
http://summerofhpc.prace-ri.eu


Leon Kos

Testing different types of methods for mesh partitioning for the solver of the parallel PDE systems.

Partitioning for the parallel solution of PDEs

Wojciech Laskowski



The goal of the project is to compare different mesh partitioning methods for the iterative solver and measure their performance with respect to solver efficiency, parallel efficiency and computational time.

The practical goal of the project is to speed up Finite Element Method simulations through analysis of the iterative solver performance. Only one, particular piece of the big FEM engine is analysed: influence of the type of mesh partitioning on the iterative strategy performance. The project investigates one iterative solver - the Deflated Conjugate Gradient, but the methods can be also used for other solvers that benefit from subdomain division. In total, 3 different methods have been tested. Comparison of those methods and other results of the different group number performance in large parallel environment are visually presented at the end of the report.

Before we get to the point and take a closer look at the methods, let's make a short introduction to FEM and iterative solvers.

Background theory

The Finite Element Method (FEM), in a nutshell is a numerical tool for solving a physical problem. How does it work? It's quite simple. First, we need know the mathematical formulation of the given problem, which is most likely described by a *Partial Differential Equation* (PDE). Then, we can discretize the PDE system on the problem's geometry by splitting it to a couple (or a greater number of) elements. The main graphic nicely shows a discretized geometry (which is often called a *mesh*). After we input some initial condition on the boundaries, the problem is ready to be solved. How can we solve it? Every vertex of the mesh symbolises one unknown of the linear system of equations, so we can gather all of the nodes (unknowns) and put them in the very popular formulation

$$Ax = b$$

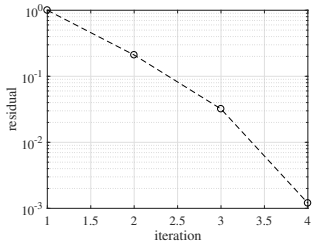
where A is a matrix of size NxN and the vectors x and b are size Nx1. N represents the number of nodes in the mesh or unknowns in the equation system.

We could solve the equations using some of the direct methods (e.g. very popular LU factorisation), but for really big systems it can take a very long time to solve. *Iterative solvers* come in handy in this situation. In short, an iterative

solver is a sequence of recurring matrix-vector or matrix-matrix operations that will lead to the final solution. Let's take a look at the equation:

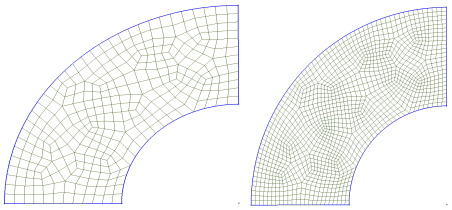
$$x^{m+1} = x^m + Br$$

By assembling a matrix B and some initial guess x^m, we can compute the next approximation x^{m+1}, which will be closer to the desired solution. $r = b - Ax^m$ is a residual. Residual or its norm is usually a very good indicator on how far we are from the exact solution and is actually known as a stopping criterion of the iterative strategy. The figure below depicts how the residual should behave with every iteration step m.



The solver

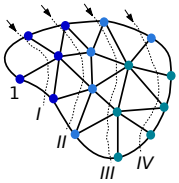
There are many iterative solvers for different applications. During the summer, only one specific solver was investigated - the Deflated Conjugate Gradient (DCG). Without getting into details, the DCG solver uses a coarse grid correction to speed up the convergence process, i.e. it solves a coarse system of equations directly and uses it to dump the low frequency errors on the fine grid (for detailed description of the method see.[1]). *Coarse grid* means the same PDE system, on the same geometry, but with lower accuracy. The mesh is divided into a fewer number, but bigger elements. This can be seen in the visualisation below, where the grid on the right represents an exemplary original mesh, and the grid on the right is a coarse grid.



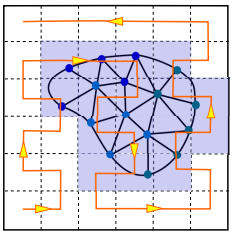
A coarse grid is chosen by the arbitrary number of groups the mesh is split into. The greater the number of groups, the finer the coarse grid becomes. For example, if we divide the mesh into a number of groups equal to the total number of elements, the coarse system would become the same as a "normal" system and it would converge in only 1 iteration, since we have already provided a direct solution to our system. Likewise, dividing the mesh using only one group means using only one element for the whole geometry. Similarly, coarse grids of the same number of groups can look completely differently through different splitting. Therefore, we reach the main objective of the project - investigating the role that the quantity of groups and the division strategy play in the solver.

Methods

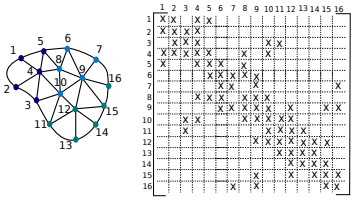
In total, 3 methods have been tested. The first method, purely base on geometry on the model is called *Frontal Approach*. We start with a first node of the first element and slowly assign each node to one group. The algorithm detects neighbours and put them in the *front* - neighbouring nodes are first in line to be assigned in the group. When we are done with one element, we go through neighbouring elements. Once we reach the required number of nodes per group, we simply close the first group and proceed with the algorithm, but now, we are picking the nodes for our second group. This goes on until all of the nodes are assigned and groups filled. The graphic below presents how the method works. Node number **1** starts, and from there, the first front of neighbouring elements is detected.



The second method is called *Space Filling Curve* (SFC). It is also based on the geometry of the model, but in a slightly different way. To do SFC partitioning, we need to take a rectangular grid, called "bin" and then, embed our geometry into the bin. We can then fill the bin with its own elements and tie them to the original mesh elements. We divide the elements on the new rectangular mesh. The idea of SFC is to explicitly coarse all the two and three-dimensional cases to 1D. We start at an arbitrary point and go through all the points, one by one, creating a continuous line throughout the mesh. The rule of how elements are assigned to a group is just the same as in the Frontal Approach - once we fill out one group, we carry on with the elements and start another.

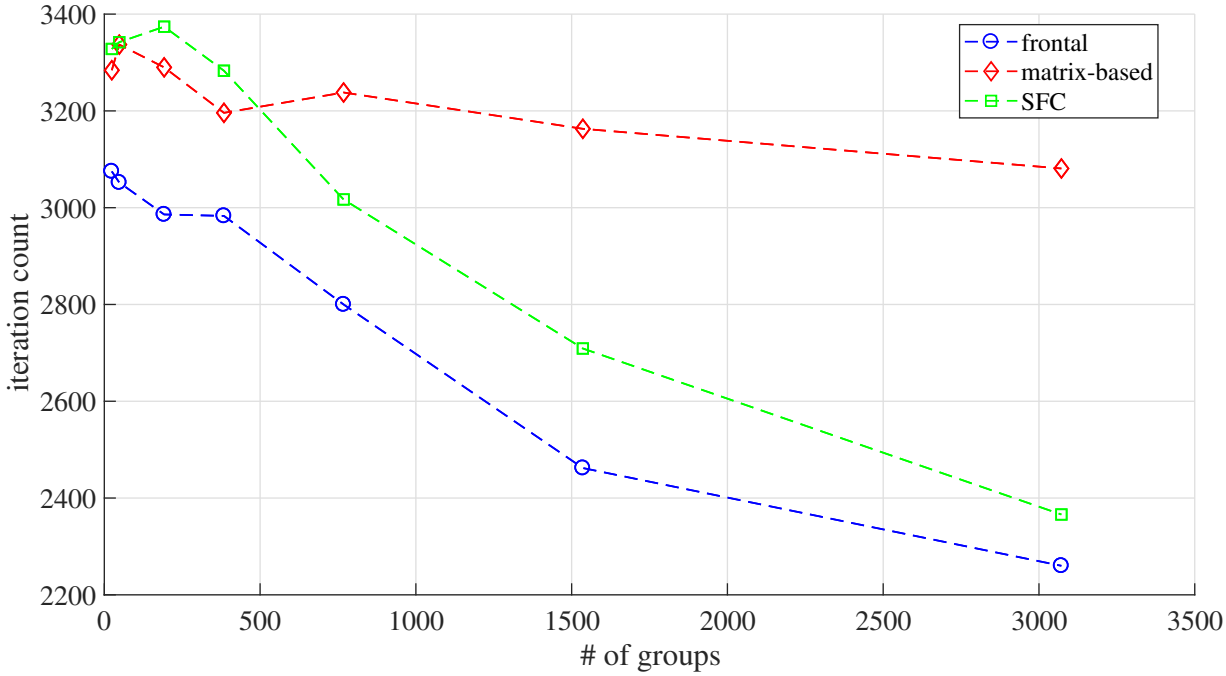


The last method, called *Matrix Based* tries to reproduce the Frontal Approach method, but without geometry. Instead, we use the matrix, more precisely, one very important property of the matrix that connects elements to each other. Each node has its own, unique number, which represents one row in the linear system of equations. And each entry in that row represents a connection between nodes. For example, if we take node number 1, with neighbours number 2, 4 and 5, it corresponds to the first row of the matrix with entries in the 1st, 2nd, 4th and 5th column of the row. Assigning the nodes to a group works exactly the same as in the Frontal Approach - we start at node 1, go trough neighbours, then through neighbours of neighbours and so on until every single group has a right number of nodes.

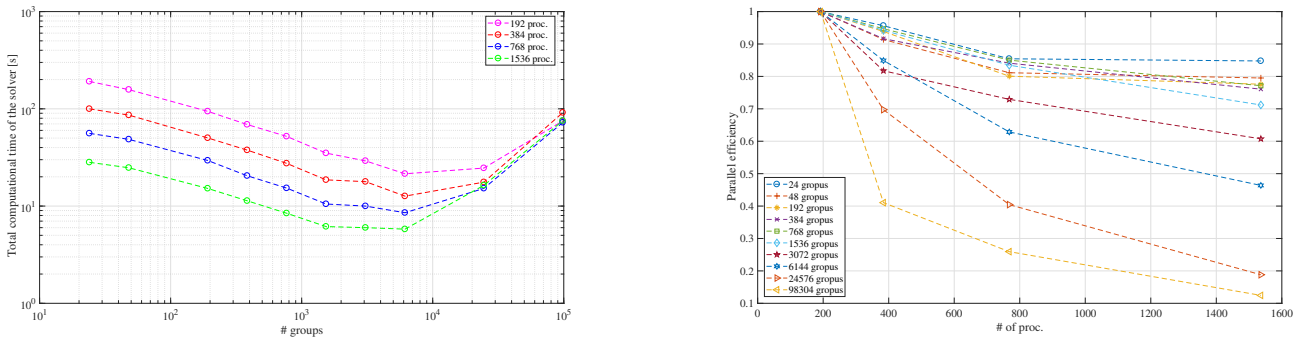


Results

All of the methods were tested on a respiratory tract case, simulated by almost 18 million elements (the leading graphics presents the case). The methods are compared with respect to the number of iterations needed to obtain the solution within a chosen tolerance. It is worth noting that number of iterations is not the most important factor from the engineering point of view. After all, the goal



Comparison of all 3 methods.



Efficiency plots of the Frontal Approach method.

of an iterative solver is to solve the problem fast, thus time would be the most representative component of all. But on the other hand, algorithmic scalability directly represents theoretical features of the solver, which is the essential part from the developer perspective and it has a direct impact on the time to solution as well. Additionally, if two methods are compared in the same environment (same solver "engine"), the number of iterations and time to solve are strictly bounded to each other.

From the methods tested, the frontal approach gives the best results. The total iteration number decreases faster with respect to number of groups, which is exactly the behaviour we wanted to see. SFC is slightly slower, but still performs quite well when compared to the Matrix Based method.

Finally, let's take a look at how the Frontal method performs on various number of processes. A computational time plot in respect to the groups clearly indicates that the "sweet spot" for fast computation is from 1000 to 10000 groups. Above that number, the influence of parallel communication becomes too big. This can also be observed on parallel efficiency plot. Lower group numbers hold on to their efficiency, but big numbers loose their efficiency almost immediately.

Outlook

The project may be a base for further study on the topic. Making the same test on different types of physical problems and geometries would give more data to analyse. Also, it is possible to enhance the Matrix Based method, which performed rather poorly in the test. Putting some more enforcing conditions in order to strengthen connection between nodes would make the groups more coherent, which will very likely result in a better convergence.

References

[1] Rainald Löhner, Fernando Mut, Juan Raul Cebral, Romain Aubry, Guillaume Houzeaux (2011). Deflated preconditioned conjugate gradient solvers for the pressure-Poisson equation: Extensions and improvements

PRACE SoHPCProject Title

Adaptive multi-partitioning for the parallel solution of PDEs

PRACE SoHPCSite

Barcelona Supercomputing Center, Spain

PRACE SoHPCAuthors

Wojciech Laskowski, [Technical University of Denmark,] Denmark

PRACE SoHPCMentor

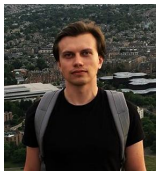
Ricard Borrell and Guillaume Houzeaux, BSC, Spain

PRACE SoHPCSoftware applied

Alya

PRACE SoHPCProject ID

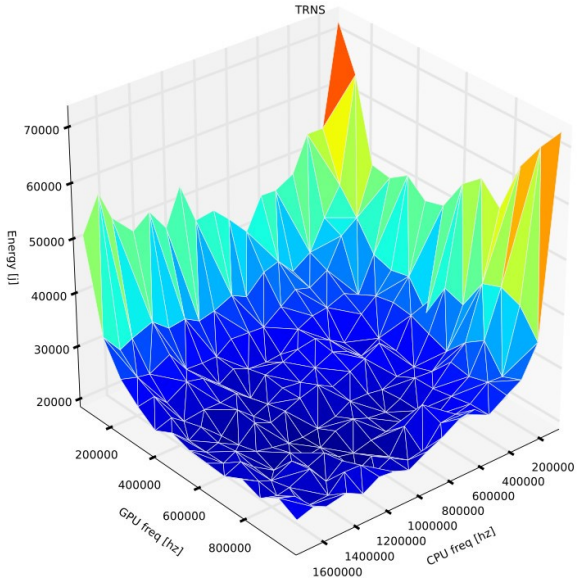
1801



Wojciech Laskowski

Automatic Frequency Scaling

Zheqi Yu



The project will design and use artificial intelligence methods to find the most appropriate performance and energy ratio in the hardware to contribute to the planet's energy saving.

This will be an optimisation attempt for low-power hardware, and efforts will be made for the energy conservation and environmental protection of the earth. The narrow concept of energy saving is outdated. For hardware, lower power consumption requires energy efficiency, because it will cause a program to run slower and need more time to finish a process. The goal is to find a balance between energy consumption and performance, and then dynamically adjust the frequency for different applications to become more energy-efficient. Today's artificial intelligence computing is becoming more and more popular, and it now has powerful analysis and logic capabilities. We will use machine learning algorithms to analyse our dynamic voltage frequency scaling data, obtained from the Nvidia Jetson TX1 chip, for different applications. We will design an operation mode that can balance the energy consumption and performance, to dynamically adjust for different applications on the Nvidia Jetson TX1 chip, thus finally completing the exploration of chip energy saving. The algorithm we designed will be validated on the Nvidia Jetson TX1 board to find the best performance and energy balance when running different programs. This work can be extended to different hardware to help them achieve better energy savings without affecting performance.

Dynamic Voltage Frequency Scaling (DVFS)

Dynamic Voltage Frequency Regulation is an emerging and generally popular approach to improve the energy efficiency of computing systems. It uses a frequency regulator to adjust the voltage and frequency levels used to complete the current task. But the mechanisms used to control frequency changes are too linear, which leads to inefficient energy consumption across the system [1]. On low-power embedded system platform such as the NVIDIA Jetson TX1, this problem becomes more and more prominent. More frequency options will cause an enormous number of CPU and GPU frequency combinations, which makes it impossible to conduct a brute force search for optimal points for energy [2].

Methodology

The method of this project are mainly divided into three parts:

- Part One: Use own power monitor functions to collect the running data of applications, and use the profiling method to analyse the internal operations of the corresponding application.
- Part Two: Use the Kmeans algorithm from unsupervised machine learning to classify the collected frequency and energy consumption data to obtain positive and negative samples as a training dataset (positive samples are the optimal balance between frequency and energy consumption).
- Part Three: Use the previously extracted dataset to train a random forest algorithm and get the classifier. After we capture the new program frequency information into the random forest algorithm, it will output whether the current running frequency is the optimal setting of the program for optimal frequency and energy consumption balance. We can use this result as a credential that decides

to modify or maintain the current frequency.

Kmeans unsupervised learning for dataset classification

The Kmeans algorithm is a process for repeating centre points of a multiple class dataset, also called the centre point of class as a centroid, and then re-dividing its containing classes to the average position [3]. The cluster data is divided into N sets of independent data. The sample is such that the variances between the N sets of clusters are equal, mathematically described as minimising inertia or the sum of squares within the cluster. N is the hyperparameter calculated by the algorithm, indicating the number of classes. The Kmeans algorithm can automatically allocate samples to different classes, but cannot decide whether to divide them into several classes. N must be a positive integer smaller than the number of samples in the training set. Sometimes the number of classes is specified by the content of the question. Therefore, the disadvantage of this algorithm is that it is necessary to determine the number of classes (N) for the data cluster in advance.

Due to our power monitor function, we can capture the following information in the hardware: timestamp (ms), total power (W), GPU power (W), CPU power (W), CPU0 freq (kHz), CPU1 freq (kHz), CPU2 freq (kHz), CPU3 freq (kHz), GPU freq (kHz). We chose 4 CPU frequencies and 13 GPU frequencies in the test and fixed the 4 CPU cores in the same frequency to avoid model errors for different energy consumption of different frequency.

Finally, we set the N parameter of the Kmeans algorithm to 13 by the GPU frequency steps, and each cluster class results are shown in the following: First, we manually extract the average location class as a positive dataset, and then choose the first and last class as a negative dataset as training data for the Random Forest algorithm.

Random Forest as a classifier to estimate currently frequency

Random forest is a non-traditional machine learning algorithm and is a classifier that uses multiple decision trees to train and predict samples. Each decision tree processes a subset of training samples. In the training phase, the features are filtered by the node splitting of the decision tree, and the samples are subdivided until the subset of each training sample is correctly classified [4].

For each decision tree, the training subset uses a back-sampled method from the whole training set, which means that some samples in the whole training set may appear in the training subset of a decision tree multiple times, or may never appear in the training subset of a decision tree. When training nodes of each decision tree, the features used are randomly extracted from whole features by a given proportion. In the test phase, the samples are classified directly based on the trained features, so the test speed is faster (by comparison that training slower than test).

Because of the fixed point calculation request of random forests, we need to classify different values as a different hierarchy before the frequency and energy data are input into the random forest algorithm. For example, the CPU and GPU are divided into 4 and 13 levels based on their different frequency steps. In this way, the algorithm will eventually get 52 different CPU and GPU frequency combinations. In the test phase, the classifier will judge whether the newly collected frequency combination output is suitable for the current program based on the trained data.

Results

We use the power monitor function to collect hardware information of Stream Compaction (SC). It performs a filtering, which is a data manipulation primitive that removes elements from an array, keeping those satisfying a certain predicate [5].

Figure 1 is the original data we collected on the energy consumption and frequency information for the program. The table drawn by the data shows a linear relationship, which cannot distinguish datasets as positive and negative samples.

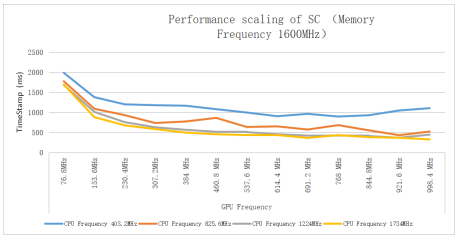


Figure 1: Line chart for original collected power monitor data

Figure 2 shows the classification information obtained after clustering analysis by Kmeans algorithm. We used N=13 for classification to get the range and corresponding samples of each class.

```
datasize=11065
0:min:16.437799 ,max:94.462097 diff:78.024300

After 14 times calculation
Randomly the K-locations are initialized as follows:
location---1:  22.672628,
location---2:  26.737795,
location---3:  31.058798,
location---4:  35.799145,
location---5:  41.507553,
location---6:  47.376507,
location---7:  52.875282,
location---8:  59.660503,
location---9:  65.947670,
location---10:  71.807251,
location---11:  78.254303,
location---12:  86.110626,
location---13:  91.487679,
```

Figure 2: Kmeans algorithm for cluster analysis of 13 classes

Figure 3 shows the result of the new power monitor information input as a test after the random forest training. "-" means the wrong frequency selection of application. "+" represents the current frequency combination is good, which ensures the program running at the balance point of optimal performance and energy consumption.

```
C:\WINDOWS\system32\cmd.exe
-:totPower:0, CPUFreq:1224000, GPUFreq:537600
-:totPower:0, CPUFreq:1224000, GPUFreq:614400
-:totPower:0, CPUFreq:1224000, GPUFreq:691200
-:totPower:0, CPUFreq:1224000, GPUFreq:768000
-:totPower:1, CPUFreq:1224000, GPUFreq:844800
-:totPower:3, CPUFreq:1224000, GPUFreq:921600
-:totPower:3, CPUFreq:1224000, GPUFreq:998400
+:totPower:0, CPUFreq:1734000, GPUFreq:76800
+:totPower:0, CPUFreq:1734000, GPUFreq:153600
+:totPower:1, CPUFreq:1734000, GPUFreq:230400
+:totPower:2, CPUFreq:1734000, GPUFreq:307200
+:totPower:1, CPUFreq:1734000, GPUFreq:384000
+:totPower:2, CPUFreq:1734000, GPUFreq:460800
+:totPower:1, CPUFreq:1734000, GPUFreq:537600
+:totPower:2, CPUFreq:1734000, GPUFreq:614400
+:totPower:1, CPUFreq:1734000, GPUFreq:691200
+:totPower:0, CPUFreq:1734000, GPUFreq:768000
+:totPower:0, CPUFreq:1734000, GPUFreq:844800
+:totPower:2, CPUFreq:1734000, GPUFreq:921600
+:totPower:1, CPUFreq:1734000, GPUFreq:998400
Train size:     1827
Test size:      52
positive:       21
negative:       31
Press any key to continue . . .
```

Figure 3: Random Forest algorithm to evaluate frequency state and output

Discussion & Conclusion

The current work has been able to judge the current frequency of the matching application, which can help the hardware to better select the appropriate frequency to a processing program, and finally to ensure energy-saving that the hardware can complete the task with the best energy consumption. We chose sufficient frequency steps to collect data, and then find the best balance point after getting their frequency combination. In the future, we will combine the completed random forest algorithm on

the power monitor function and give the kernel control permission to dynamically carry out frequency scaling, so it can get the balance between the optimal energy consumption and frequency of the current task faster. This can help for the improvement of unknown new programs that can be used for profiling data to analyse the internal operational relevance of the program. The existing training data to predict and adjust the frequency of new procedures can be used to achieve intelligent and comprehensive energy-saving purposes.

References

[1] K. Rao, J. Wang, S. Yalamanchili, Y. Wardi, and Y. Handong, (2017). Application-Specific Performance-Aware Energy Optimization on Android Mobile Devices.

[2] Baruah, T., Sun, Y., Dong, S., Kaeli, D. and Rubin, (2018). Airavat: Improving Energy Efficiency of Heterogeneous Applications.

[3] Celebi, M.E., Kingravi, H.A. and Vela, P.A., (2013). A comparative study of efficient initialization methods for the k-means clustering algorithm.

[4] Mendoza, M.R., da Fonseca, G.C., Loss-Morais, G., Alves, R., Margis, R. and Bazzan, A.L., (2013). RFMirTarget: predicting human microRNA target genes with a random forest classifier.

[5] Gómez-Luna, J., El Hajj, I., Chang, L.W., García-Floreszx, V., de Gonzalo, S.G., Jablin, T.B., Pena, A.J. and Hwu, W.M., (2017). Chai: collaborative heterogeneous applications for integrated-architectures.

PRACE SoHPCProject Title

Automatic Frequency Scaling for Embedded Co-processor Acceleration

PRACE SoHPCSite

Barcelona Supercomputing Centre, Spain

PRACE SoHPCAuthors

Zheqi Yu, [University of Wolverhampton,] UK

PRACE SoHPCMentor

Antonio J. Pena, BSC, Spain

Zheqi Yu

PRACE SoHPCContact

Zheqi, Yu, University of Wolverhampton
Phone: +44 754 749 5720
E-mail: z.yu@wlv.ac.uk

PRACE SoHPCSoftware applied

Microsoft Visual Studio

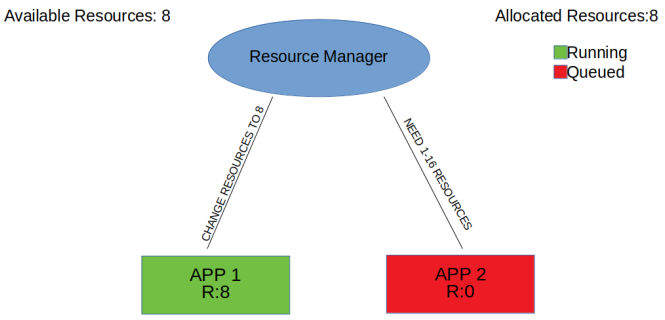
PRACE SoHPCAcknowledgement

Filippo Mantovani, BSC, Spain
Marc Jorda, BSC, Spain

PRACE SoHPCProject ID

1802

Resource Simulator for SoHPC



Klajdi Bodurri

The purpose of this project is exploring malleability in HPC by developing a simulator that executes a workload. By tuning different reconfiguration policies, we can determine the best configuration for fulfilling a given target without running the workload on an actual system.

In HPC facilities, many applications run concurrently and compete for the same resources. Applications request for resources from the resource manager in order to run. From this, we can understand that the resource manager plays a vital role in global throughput of the system and resource utilisation. The resource manager fulfills the requests of the applications by following a specific policy. Therefore, by changing the policy, we can see a total different behaviour in the throughput, resource utilisation and even in the total execution time of the system.

We should explain first what an application is. An application in our case, is nothing more than a piece of code which executes a computation. Applications also have some characteristics. One of these is malleability. Malleability is the ability to dynamically change the number of computational entities in an application. Accordingly, we can understand that even if an application requests an amount of resources that cannot be delivered by the resource manager, the application can still run with less resources instead of remaining idle. The trade-off here is that the application will run slower.

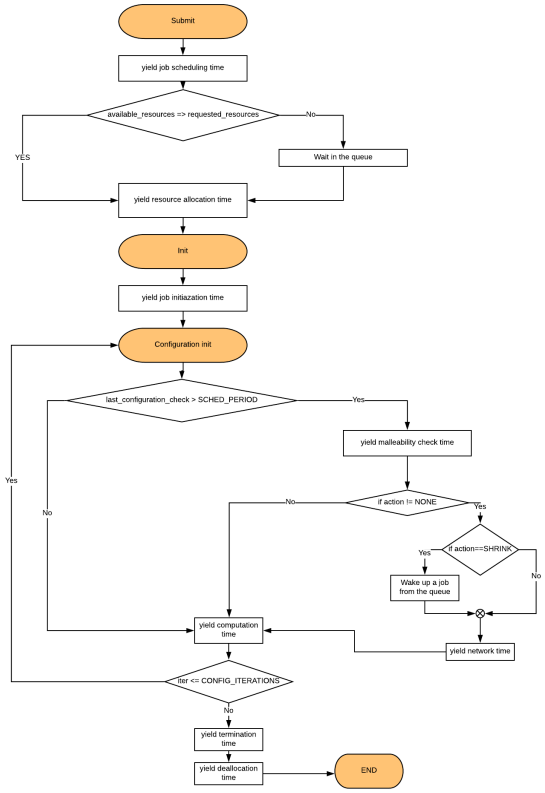
Consequently, the resource manager is very important for the distribution of resources to applications. What if the resource manager could exploit also malleability of applications by using a specific policy for malleability? This is exactly what this project is about. Giving a workload (malleable and non-malleable applications) as input to the simulator and by trying different reconfiguration policies of the resource manager, we can determine the best configuration without even running the workload in any HPC facility. We can even test different policies in a single-core system.

Implementation

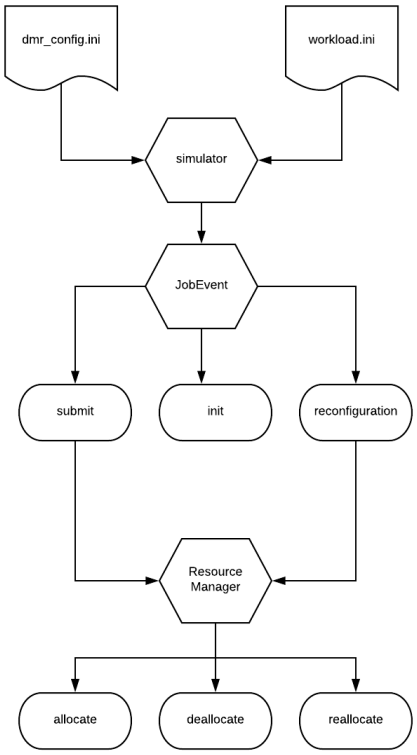
From now on, we will refer to applications as jobs and when we say resources, we mean computational entities (CPUs or GPUs). To better understand the implementation, let's break the application into states. Each application has 3 states:

- **Submit** state. In this state, the job asks for resources from resource manager ('submit' its request) and waits for the resources. The job can ask for a specific number of resources or can give the opportunity to the resource manager to decide based on the minimum and maximum resources the job needs to run.
- **Init** state. In this state, the job has some resources for itself, so it has to initialise those resources for the computation to start. This state is for simulating this initialisation.
- **Reconfiguration** state. This state is the heart of the simulation and is nothing more than a loop that runs n times based on the iterations the job needs to complete the computation. The configuration state can be broken into 2 pieces. The first one is the simulation of the computation time per iteration. The second one is every time period the job asks from the resource manager an action:
  - Action **EXPAND**: The resource manager decides to increase the number of resources for a job.
  - Action **SHRINK**: The resource manager decides to decrease the number of resources for a job.
  - Action **NONE**: The number of resources for a job remain the same.

Since the computation is distributed among the computational entities (resources), if the



(1) The workflow of a job



(2) Architecture of the simulator

job decides to shrink or to expand, it has to send data over the network in order for the computation to continue. For more information, please see Figure 1.

Architecture of the simulator

The simulator has 3 modules. The first one is called **sim** module and it is in charge of initialising the simulator by reading 2 configuration files. These two files keep the information of the infrastructure of the HPC facility that we want to simulate and of course the information of the workload. After reading these 2 files, the sim module creates the jobs.

The second module is called **JobEvent** which describes the functionality of a job. As described earlier, each job has 3 states. Submit, Init and Reconfiguration. The Submit and Reconfiguration states are connected with the third and final module which is the **Resource Manager**.

The Resource Manager is in charge of handling the requests of the jobs by providing 3 functionalities. The functionalities are Allocation, Deallocation and Reallocation of the resources. The malleability policy is hidden in the Reallocation functionality (Figure 2).

An Example

Now that we know how the simulator works, we can understand its purpose by an example. Let's assume that we have a workload of 100 jobs. We want to determine which configuration policy is the best for the current workload. The simulator supports 2 policies at the moment.

The output by running the workload with the one of the two policies.

- **Total execution time**: 4356 seconds.
- **Maximum throughput**: 16
- **Average throughput**: 15

The output by running the workload with the other policy.

- **Total execution time**: 4994 seconds.
- **Maximum throughput**: 4
- **Average throughput**: 3

Conclusion

In conclusion, for someone who wants to see the behaviour of a policy that exploits malleability or even compare different policies, it is faster and easier to get results from the simulator than running the policies in the HPC system directly. In the above example, it took 30 seconds to see which policy is better for a given workload. In an actual system, it would have taken 2-3 hours.

References

[1] Sergio Iserte, Rafael Mayo, Enrique S. Quintana-Orti, Vicenç Beltran, Antonio J. Peña. Efficient Scalable Computing through Flexible Applications and Adaptive Workloads.

PRACE SoHPCProject Title

Dynamic management of resources simulator

PRACE SoHPCSite

Universitat Jaume I, Castelló de la Plana, Spain

PRACE SoHPCAuthors

Klajdi Bodurri, [University of Thessaly,] Greece

PRACE SoHPCMentor

Sergio Iserte, Universitat Jaume I, Spain

PRACE SoHPCContact

Klajdi, Bodurri, University of Thessaly
Phone: +30 698 2946 137
E-mail: klajdibodurri@gmail.com

PRACE SoHPCSoftware applied

Python2.7 SimPy

PRACE SoHPCMore Information

http://simpy.readthedocs.io/en/latest/

PRACE SoHPCProject ID

1803

Get More Throughput Resize Me!

Sukhminder Singh

Malleable jobs can enhance the throughput of supercomputing clusters and reduce the turn-around time. In this study, we show the complexity and benefits of converting a popular molecular dynamics code LAMMPS to make it malleable.

Job malleability is a term being used in HPC which has the potential to increase the productivity of a cluster. The primary motivation for high-end supercomputers is applications requiring thousands of compute cores, with data structures distributed across a large aggregate memory, and with relatively high inter-process communication requirements. One drawback of the existing cluster schedulers is that they are 'static', i.e., once a job is allocated a set of nodes, it continues to use them until execution is finished. A malleable application can dynamically adapt its parallelism to the number of processes it has been allocated to, depending upon the cluster workload. Having malleable jobs running on a supercomputer cluster, is expected to increase the throughput of the cluster as well as decrease the turn-around time of the submitted jobs.

This project aims to study the particular case of using DMRlib, a Dynamic Management of Resources library, to generate a malleable version of LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator).

Overview of LAMMPS code

LAMMPS works by decomposing the domain and distributes the information of particles among all the processes. Neighbour lists are used for efficient computation of the forces. The information related to the particles such as positions, velocities, masses, etc. are stored in structures of arrays form. The code comes with an in-built checkpoint-restart functionality for fault tolerance.

DMRlib applied to LAMMPS

A dynamic reconfiguration system relies on resource management system (RMS) which inspects the global status of the cluster and takes decisions regarding the resizing actions. The RMS is aware of the queue of pending jobs and the current utilisation of the cluster. At the beginning of each integration iteration, the application calls the function macro DMR_RECONFIG which then informs the RMS that resizing can be performed. The RMS then responds whether a resizing action, either expand or shrink, can be taken. If there is an action, then new processes are spawned with the desired size. User defined functions are called to create a new instance of LAMMPS and transfer the data. If there is no action, then the application continues with the iteration with the same number of processes. The minimum, maximum and preferred number of processes are specified in the beginning of the simulation.

Dynamic Resizing using File-Based Check-pointing

The already available file-based checkpoint-restart functionality in LAMMPS makes the implementation quite simple. If a resize action needs to be taken, then all the processes write their restart data on disk. A new LAMMPS instance is created with the newly spawned processes, which then reads the restart files and continues executing the simulation from where it was stopped in the old processes. This way, there is no need to take care about data management as everything is done internally by LAMMPS.

Dynamic Resizing Using MPI Data Transfers

Another way of doing the dynamic reconfiguration is by transferring the data of particles between the parent group and the child group of processes through an MPI inter-communicator. Additional functions were added in LAMMPS to send the data from the old processes and receive the data in the newly spawned processes. Since there is uncertainty in knowing the exact rank to which a particle belongs in the new processes, a redistribution of particles is carried out after receiving all the data. New neighbour lists are created and the simulation continues from the point where it was stopped in the old processes.

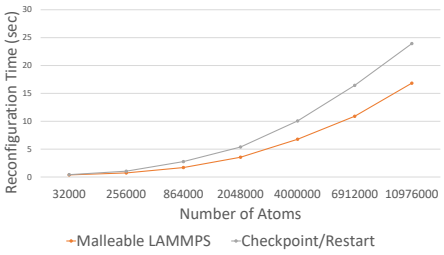


Figure 3: Reconfiguration times for job expansion from 4 to 8 processes with different number of particles.

Workload experiments were performed using the *Marenostrum IV* cluster at the Barcelona Supercomputing Center (BSC). Two workloads with 100 jobs each, one having rigid LAMMPS jobs and another having malleable LAMMPS jobs, were submitted on two sets of partitions with 32 and 40 nodes. The minimum, maximum and preferred number of nodes for a malleable job were set to 1, 16 and 4 respectively. The size of the problem was chosen to be 16k atoms and the simulations were run for 100 iterations. All 48 threads in a node were utilised using the inbuilt *USER-OMP* package in LAMMPS.

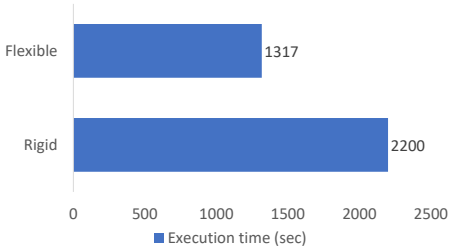


Figure 5: Workload times with cluster size of 40 nodes.

The rigid jobs were initiated with 16 processes and the flexible ones with variable number of processes from 1 to 16, depending upon the available nodes. The execution times for the workloads can be seen in Figure 4 and 5, providing speedups of 1.40 and 1.67 respectively.

Extension for Fault Tolerance

The malleable version of LAMMPS can be extended to tolerate failures of the allocated nodes. There may be a number of reasons for a node to fail during the execution of a job. In some cases, the job scheduler can predict whether a node is going to fail in the next couple of minutes. In that case, a malleable job can shrink by one node allowing itself to continue running on the cluster without needing to be manually restarted by the user.

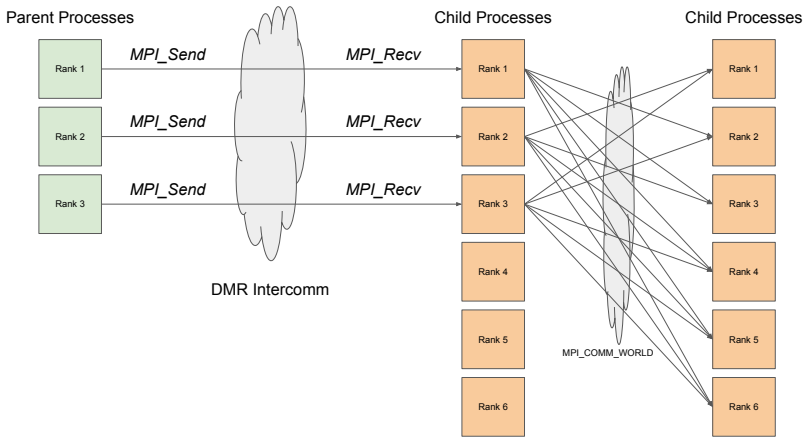


Figure 1: Flow chart describing data transfers from the parent processes to the child processes through an inter-communicator and then redistributing the data to correct processes.

Experimental Results

Experiments were done to compare the timings of MPI based dynamic reconfigurations to the file-based checkpointing.

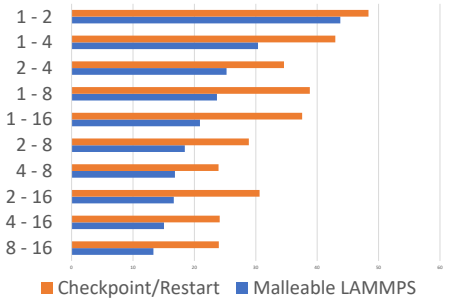


Figure 2: Reconfiguration times for job expansion with 10,976,000 atoms.

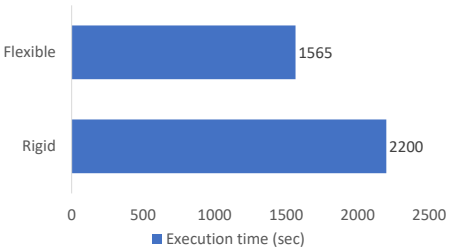


Figure 4: Workload times with cluster size of 32 nodes.

References

[1] Sergio Iserte, Rafael Mayo, Enrique S. Quintana-Ortí, Vicenç Beltran, Antonio J. Peña (2018), DMR API: Improving cluster productivity by turning applications into malleable study.

PRACE SoHPCProject Title

Get more throughput, resize me! A case of study: LAMMPS malleable

PRACE SoHPCSite

Jaume I University, Spain

PRACE SoHPCAuthors

Sukhminder Singh, [Friedrich-Alexander-Universität Erlangen-Nürnberg,] Germany

Sukhminder Singh

PRACE SoHPCMentor

Sergio Iserte, Jaume I University, Spain

PRACE SoHPCContact

Sukhminder Singh, Friedrich-Alexander-Universität Erlangen-Nürnberg
Phone: +49 15259651811
E-mail: sukhminder.singh@fau.de

PRACE SoHPCSoftware applied

LAMMPS, DMRLIB

PRACE SoHPCMore Information

lammps.sandia.gov

PRACE SoHPCAcknowledgement

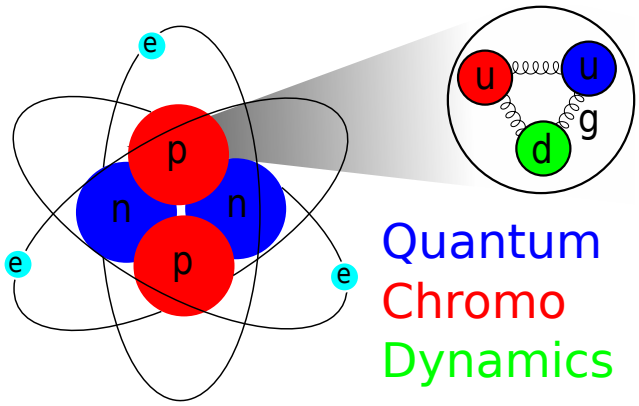
The author would like to acknowledge the invaluable support of the project mentor Sergio Iserte and the researchers of Jaume I University, Spain.

PRACE SoHPCProject ID

1804

QCD can explain the force, that keeps nucleons together. Simulations are done on a lattice and greatly benefit from being carried out on GPUs.

Lattice QCD simulations on GPUs



Marius Neumann

Lattice Quantum Chromodynamics can describe and predict experimental results from particle physics in the low energy regime, where analytical methods fail. In this project, LQCD simulations will be put on GPUs and run on high performance clusters.

Quantum Chromodynamics (QCD) is the theory of the strong nuclear force, which binds quarks together to form nucleons. Thus, knowledge of QCD is crucial to our understanding of heavy ion collision experiments in particle accelerators and the hunt for dark matter by looking into possible interactions between quarks and dark matter.

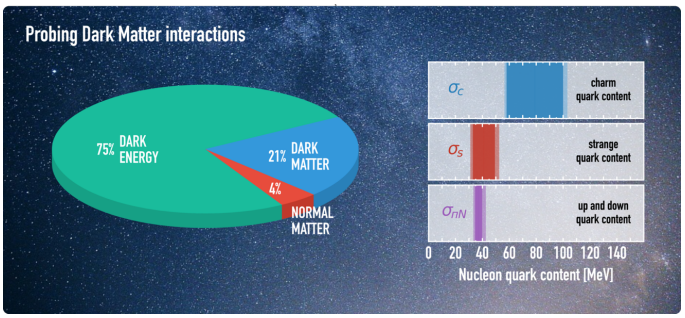


Figure 1: Of the total amount of matter that we know to exist, we can only see 4%. Thus, we simply call the rest "dark" matter and energy. The nuclear quark content[1] is a useful quantity, that can be used in the hunt for dark matter.

Unfortunately, in the low energy regime, QCD cannot be easily calculated asanalytical methods like perturbation theory do not work. So simulations must be used. This is done on a four dimensional space time lattice of billions degrees of freedom, which means that lattice QCD (LQCD) is a job for supercomputing.

Now here comes the gaming industry to give us a little boost. They have optimised GPUs to handle 3×3 rotation matrices really fast. And QCD too is described by 3×3 matrices, so some tricks made for 3D graphics acceleration may be used, too!

To run the simulation, Noether, CaSToRCs new GPU server was used for building, (lots of) debugging and finally for implementing features (like calculating the trajectory of thermalisation steps of the gauge configuration, the most expensive part of the calculation) to also be able to run on GPUs.

After the trajectory of gauge configurations has been calculated, a variety of measurements can be made, allowing us to predict crazy things like the masses of some strange particles!

In this work to be honest, we will remain a little bit less crazy and measure only the pion mass as a check of plausibility. Pions occur naturally, for example when cosmic rays hit the atmosphere.

Metropolis algorithm

To measure an observable in a quantum field theory (like QCD), we must evaluate a certain nifty path integral. In case of the QCD action, this cannot be solved analytically.

So what should we do now? Well, take out your dice and start gambling, because this integral can be solved by Monte-Carlo integration! What we do now is to start with a grid of random numbers and let them propagate according to Euler-Lagrange, until the action stabilises and we can consider the configuration as "thermalised". After each step, we randomly decide based on the total energy difference, with a probability of

$$p = \min\left[1, e^{-\Delta H}\right] \qquad (1)$$

whether to accept or ditch the new configuration. Oh, and before each step, we throw the whole thing in a heatbath, to avoid any local minima.

The total set of accepted and thermalised configuration is called a trajectory and can be used for further measurements.

Hasenbusch trick

So how do we speed up the calculation of the trajectory? Well, by looking at the most expensive part of course! In this case, this is the inversion of the Dirac Matrix D, which is done by a conjugate gradient (CG) algorithm. CG converges fastest for Matrices with large eigenvalues, which D unfortunately does not

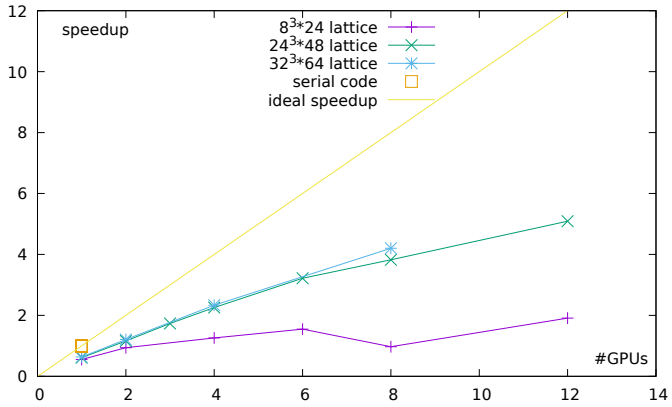


Figure 2: Speedup compared to serial code for three differently sized lattices. Notice bad performance and even dip at small lattice and good performance for larger lattices.

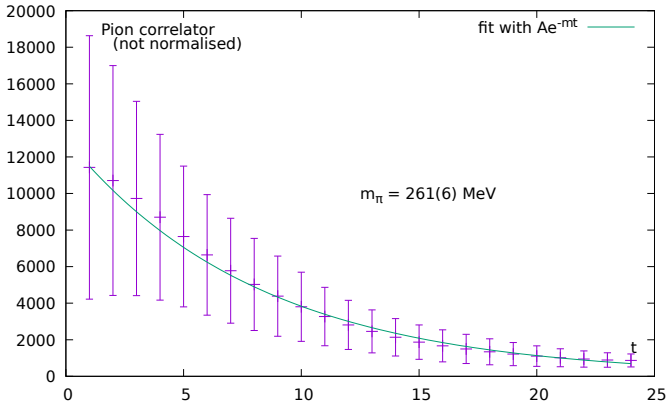


Figure 3: Unnormalised pion correlators. Only half of the time slices are shown since periodic boundary conditions apply.

have. So we have to apply a little trick here:

$$|\det D|^2 = \det\left(D^\dagger D + \mu^2\right) \times \det\left(1 + \mu^2 \left(D^\dagger D\right)^{-1}\right)^{-1} \quad (2)$$

Now this looks like quite a pedantic way to calculate the square of a determinant. But if we assume the smallest eigenvalue of D to be 10^{-4} and set μ to about 10^{-3}, then (2) becomes something like $10^{-8} = 10^{-6} \cdot 10^{-2}$. And since this has larger eigenvalues, GC can solve it faster, despite there being two determinants to calculate now! Moreover, it splits up the forces which minimises the integeration error and enhance the acceptance. And of course, you can add more than one term.

Tuning the integrator

Now that we have the Hasenbusch terms (we choose to implement two of them, so we get three in total), we can start tuning the two parameters μ_1 and μ_2 as well as the overall number of integration steps N that is used to solve the Euler-Lagrange Equations.

So when we say that we are tuning the integrator, we mean that we are trying to find the optimal set of parameters, that results in the most accepted trajectories per time. This has to be done for each lattice size to be measured.

Scaling

If we look at some speedup plots (2), we can notice quite fast, that any attempt of parallelising the code is pointless for small configurations. In the case of larger volumes instead, the speedup appears to be quite good!

The little dip at 8 GPUs on the smallest configuration can be explained by the need for a second MPI dimension here, since 24 can still be divided by 8 but this does not result in an even number, which is mandatory for the code to work. For more obvious reasons, the largest lattice cannot be run on twelve GPUs.

This is quite unfortunate, since the speedup benefits compared to the medium configuration only start kicking in at eight GPUs.

Pion correlator

One of the easiest things to calculate and thus a popular plausibility check in lattice QCD is the pion mass. To get this, the euclidean correlator

$$C(x,t) = \langle P(0)P(x,t)\rangle \approx Ae^{-mt} \quad (3)$$

has to be calculated.

To extract the mass, we can now perform a simple fit, which results in 261(6) MeV. As you may notice, this is larger than the physical pion mass at 140 MeV. But this is fine since we were using nonphysical quark masses (25.7 MeV instead of ~ 3.5 MeV for light quarks) to begin with.

Equation (3) is only approximate, since exited states also contribute, but are strongly suppressed and only have a reasonable effect on the first few time slices.

Conclusion

The implementation of the Hasenbusch for GPUs was successful (thanks to Jacob for his QUDA version, edits to QUDA were not done by myself) and appears to work properly. If further time would have been available, the heatbath could also have been carried out on the GPU.

For me, it was a valuable experience to have worked with clover fermions, twisted mass and production code, as I only had prior experience with measurements of staggered fermions.

References

[1] Alexandrou, Constantia (2016). Direct Evaluation of the Quark Content of Nucleons from Lattice QCD at the Physical Point

PRACE SoHPCProject Title

Enabling lattice QCD simulations on GPUs

PRACE SoHPCSite

Cyprus Institute, CaSToRC, Cyprus

PRACE SoHPCAuthors

Marius Neumann, Germany

Marius Neumann

PRACE SoHPCMentors

Constantia Alexandrou, CaSToRC, The Cyprus Institute
Giannis Koutsou, CaSToRC, The Cyprus Institute
Jacob Finkenrath, CaSToRC, The Cyprus Institute

PRACE SoHPCContact

Marius Neumann, Bielefeld University
Phone: +49-521-106-5315
E-mail: neumann@physik.uni-bielefeld.de

PRACE SoHPCSoftware applied

QUDA, tmLQCD

PRACE SoHPCMore Information

lattice.github.io/quda
github.com/etmc/tmLQCD

PRACE SoHPCAcknowledgement

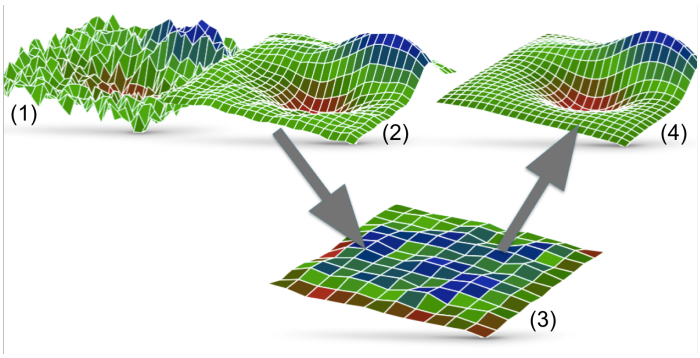
I would like to thank PRACE and the Cyprus institute for making this summer possible, as well as Constantia Alexandrou and Giannis Koutsou for giving me an opportunity to work on this interesting topic. And finally, many thanks to Jacob Finkenrath for seizing a notable amount of time to make sure everything goes well.

PRACE SoHPCProject ID

1805

Speeding up the multigrid solver by exploiting vector registers and solving for different right-hand side vectors at the same time

Vectorizing the Multigrid Solver



Marc Illa

During the last decade, physicists have performed lattice QCD simulations at the physical values of quark masses. This is possible thanks to, among others, multigrid methods, which are state-of-the-art solvers of linear systems.

Nature is governed by four fundamental interactions: gravity, electromagnetism, the weak interaction and the strong interaction. The one that we are interested in is the strong interaction, which is described by the theory of Quantum Chromodynamics (or QCD).

QCD describes the interaction between quarks and gluons and it tries to explain a wide range of phenomena: from the collisions taking place at LHC (very high energy, around the TeV scale) to how the protons and neutrons are squeezed together to form the nucleus of the atoms (very small energy, around the MeV scale). The problem is that, while for the first case we can manage to understand the process using pen and paper, for the second one it is very difficult, since all our mathematical tools break down at this energy regime.

This obstacle has not stopped physicists from making predictions at this scale, even though they can no longer be from first principles and depend on experimental data. If we want to use QCD directly to compute (for example) how the mass of the proton emerges from the interaction between the quarks and the gluons, the only way to do it is by using computers. Or, to be more precise, supercomputers.

To do that, we need to program QCD on a computer. This has been known for nearly half a century (first described by Nobel Prize winner Kenneth G. Wilson[1]): lattice QCD. It has the word lattice in it because what we do is discretise space-time in a 4-dimensional grid, placing the quarks on the nodes of this lattice and the gluons on the links.

But why is it so computationally demanding? We have to evaluate a path integral, or in other words, we have to compute every possible path that a quark can take from one point of the lattice to each and every other.

This is done by solving the following linear system:

$$Dx = b\,, \tag{1}$$

where D contains all the information about the interaction between quarks and gluons (called the Dirac matrix), x is what we are trying to get (called the quark propagator), and b is the right-hand side (rhs) vector, which tells us where we put the quark on the lattice (called the source).

Naively, one would have the temptation to solve this problem by just doing $x = D^{-1}b$. But if I tell you that the Dirac matrix is usually a $10^8 \times 10^8$ matrix with a lot of elements equal to zero, computing the inverse is not an easy task (almost impossible). Moreover, Eq. (1) needs to be solved many times, with both varying the Dirac matrix and the source, to get enough statistics and be able to extract the desired observable.

Therefore, it is crucial that we use a method that can handle this type of systems. One state-of-the-art method is the multigrid solver.

The multigrid solver

The multigrid solver can be thought of as a combination of different algorithms with the main purpose of speeding up the convergence of iterative methods, and the way it achieves it is by projecting the initial system into a smaller one, which should be faster to solve (see figure at the top).

Without going into too much detail, the solver proceeds as follows:

i) From (1) to (2) in the figure, we apply the smoother. The smoother is a linear solver which acts on the full system performing only few iterations.

ii) From (2) to (3) we restrict the linear system into a coarser one, which if small enough we solve directly, otherwise we perform again step i) and ii) on the coarse system.

iii) From (3) to (4) we prolongate the solution and check for convergence. If we didn't converge we start from i) again.

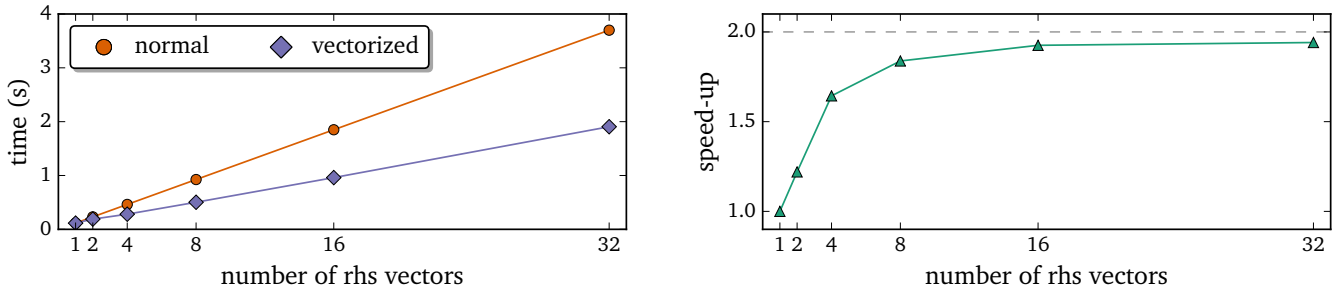


Figure 1: Time it takes for GMRES to solve Eq. (1), comparing the normal and vectorised method, with a plot of the speed-up acquired.

The algorithm that is considered here is the adaptation of DD-αAMG[2] for twisted mass fermions,[3] written in C language, where it creates 3 levels (the original lattice and two coarser lattices), and it uses a modified version of the generalised minimal residual method (FGMRES) as the iterative solver.

Vectorisation

Until now I have just explained the second part of the title, but what does "vectorising" mean?

When coding, it is very common to see the following loop:

```
for( i=0; i<n; i++ )
    c[i] = a[i] + b[i];
```

One would think that the CPU does each operation of the loop at a time, i.e. first computes `c[0]`, then `c[1]`, etc. But modern CPUs are much more intelligent.

Since CPUs are not getting any faster (their internal clocks can't go faster or they would burn up), what engineers have come up with is to increase the number of bits that CPUs can process at a time (these are called registers). If instead of having registers of 64 bits and processing one double precision float variable (single instruction single data, or SISD), we can have registers of 128 bits (or even bigger) and process two variables at a time (single instruction multiple data, or SIMD).

There are two ways of vectorising a code: an explicit and implicit way. The explicit way implies the use of special functions named "intrinsic" and defined by the constructor of the CPU. The problem of an explicit way is that the function changes depending on the register length and the CPU architecture.

The implicit way, the one we have chosen, requires to write the code in such a way that the compiler can vectorise it by its own; i.e. replacing standard operations with the intrinsic functions suitable for the CPU we are using. Modern compilers can also tell us when a loop has been or has not been vectorised and the reasons. Thanks to it we can make the right modifications for helping the compiler.

Results

So, what was my project about? The original goal was to vectorise the whole multigrid algorithm so that instead of solving the linear system for one right-hand-side vector, it could do it for 2 or more vectors at the same time (similar idea to what is done in[4]).

First, we had to change how we initialised all the vectors in the code, and instead of them being just arrays, we promoted them to a structure, which has the advantage of being able to store more than one kind of data type. Then, rather than just having the array, we also stored how many vectors there are, their size, how they are organised (their layout), etc.

As you can imagine, making this type of changes implied going through the rest of the code (+35000 lines of code!), where in some cases a simple find/replace worked, but other times we had to go through line by line. During this process, I got to understand the code structure and proceed faster with the next steps.

When we finished with this phase, we began playing with the number of vectors, looking where we could place the loops which would have to be vectorised. We started from the simpler case: the linear solver in the fine level (GMRES), changing the structure of the Dirac operator and the solver itself. Unfortunately, we didn't have time to vectorise the full multigrid, but this together with the previous step is 70% of the work needed.

The results are presented in Fig 1. On the left side, there is the time it takes to get the solution of the linear system using the normal solver or the vectorised solver, as we increase number of rhs vectors (we used a very small lattice of size 4^4). And on the right side, there is the speed-up we obtain by vectorising.

As we increase the number of rhs vectors, we reach a value of 2, which is the ideal speed-up. The reason we don't get it right away is because the compiler puts a lot of "if statements" to make sure that it uses the right amount of data inside the registers, or that it doesn't overwrite some elements which would change the final result, and they clearly slow it down.

The next step would be to solve this handicap, and then get the whole multigrid solver vectorised.

References

[1] K. G. Wilson. Confinement of Quarks. *Phys. Rev. D* **10**, (1974) 2445.

[2] A. Frommer, K. Kahl, S. Krieg, B. Leder, and M. Rottmann. Adaptive Aggregation-Based Domain Decomposition Multigrid for the Lattice Wilson–Dirac Operator. *SIAM J. Sci. Comput.*, 36(4), A1581-A1608. https://github.com/DDalphaAMG/DDalphaAMG

[3] C. Alexandrou, S. Bacchio, J. Finkenrath, A. Frommer, K. Kahl and M. Rottmann. Adaptive Aggregation-based Domain Decomposition Multigrid for Twisted Mass Fermions. *Phys. Rev. D* **94**, (2016) 114509. https://github.com/sbacchio/DDalphaAMG

[4] S. Durr, Three Dirac operators on two architectures with one piece of code and no hassle, arXiv:1808.05506 [hep-lat].

PRACE SoHPCProject Title

Vectorizing the Multigrid Solver

PRACE SoHPCSite

CaSToRC, The Cyprus Institute, Cyprus

PRACE SoHPCAuthors

Marc Illa, University of Barcelona, Spain

PRACE SoHPCMentor

Giannis Koutsou, CaSToRC, The Cyprus Institute, Cyprus

Marc Illa

PRACE SoHPCAcknowledgement

I would like to thank my mentors, Giannis Koutsou, but also Simone Bacchio and Jacob Finkerath, for their invaluable help and guidance, and Stelios Erotokritou for his support during my stay in Cyprus.

PRACE SoHPCProject ID

1806

Band structure with MPI

George Nikoulis

The calculation of the band structure of large systems is a time consuming job. Therefore the use of MPI to parallelise this work is necessary to provide a fast and efficient way for its calculation.

This work is the parallelisation of a subroutine of the SOLID2000 program. SOLID2000 is a simulation program for systems that have symmetry in three dimensions, such as crystals. The code is written mostly in FORTRAN77 and FORTRAN95.

The project is to parallelise the calculation of the band structure using MPI, which is basically the energy of the electrons inside the solid. Such solutions are impossible to find analytically for real world systems, that is why we have to use numerical methods to sample part

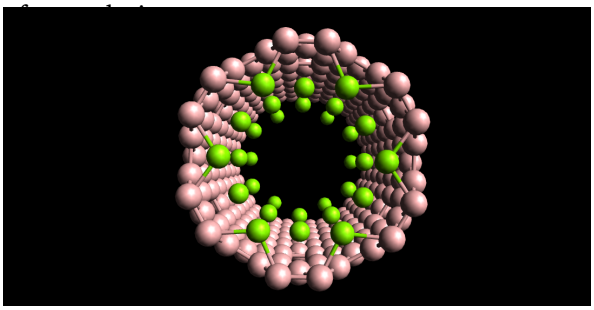


Figure 1: Nanotube from Boron and Magnesium

The finer (dense) our sampling is, the better our plot will be, and more calculation time is required. For every sample we make, we have to solve an NxN matrix, i.e to find its eigenvalues. Those eigenvalues are the solution we desire, i.e the energy of the electrons, and the matrix is called the Hamiltonian matrix which describes the whole energy of our system. This is composed of the potential energy and the kinetic energy. Normally the size of the matrix is infinite so it is up to "cut" the matrix.

The time required to calculate the eigenvalues scales as N^3, and therefore we want our matrix to be as small as possible but not too small because the accuracy of our results will be very low. The strategy that we followed to parallelize the band structure is to assign every sample to a procedure up until all the sample points are done.

That has been done by a message that the master will send to all the slaves to signal that the band structure calculation is about to start and each procedure will calculate its sample points. The work is distributed equally throughout the procedures in order to have the maximum efficiency and create an equally distributed workload. At the end of the calculation, the slaves must send their samples to the master so the master can write the band structure in a file.

Helical symmetry

The code also utilises helical symmetry, which is a special symmetry that nanotubes have. We can understand the helical symmetry properties if we take a small part of the nanotube, that is called unit cell, and duplicate it like a helical spiral. If we do that, we will end up creating the whole nanotube. This trick can decrease the computational cost of the band structure and we can avoid a big part of unnecessary work.

Difficulties

The difficulties come from the fact that we had to work with a program that consists of hundreds of files and thousands of lines of code. Of course we worked with a certain part of the code, but in order to understand the code that we had to parallelise, we had to study several files of the code to have a good overview and then start implementing the MPI coding of the band structure.

Another hard thing to overcome was that we were working on had several dependencies with other files and subroutines that we had to take into account or else the program wouldn't work properly. For this reason, we had to built

some part of the code from scratch to take care of those dependencies and make the program run smoothly.

One more problem emerges from the way that the problem itself is built. In the end of all the parallel work, we need all the data to be written in a certain order in an output file. Although the problem sounds easy, is was not that trivial. While the program is running, there is not any rule that specifies which core will take comptute part of the calculation. Therefore storing the data in order is not straight forward. To solve this problem we had to make use of the `MPI_ANY_SOURCE` and use the `tag` to control the right order of the data.

Results and Benchmark

Testing on the code showed that it works properly by changing the number of cores. We benchmarked the code using all the resources that we could from the Aurel cluster of Bratislava, up to 1024 cores.

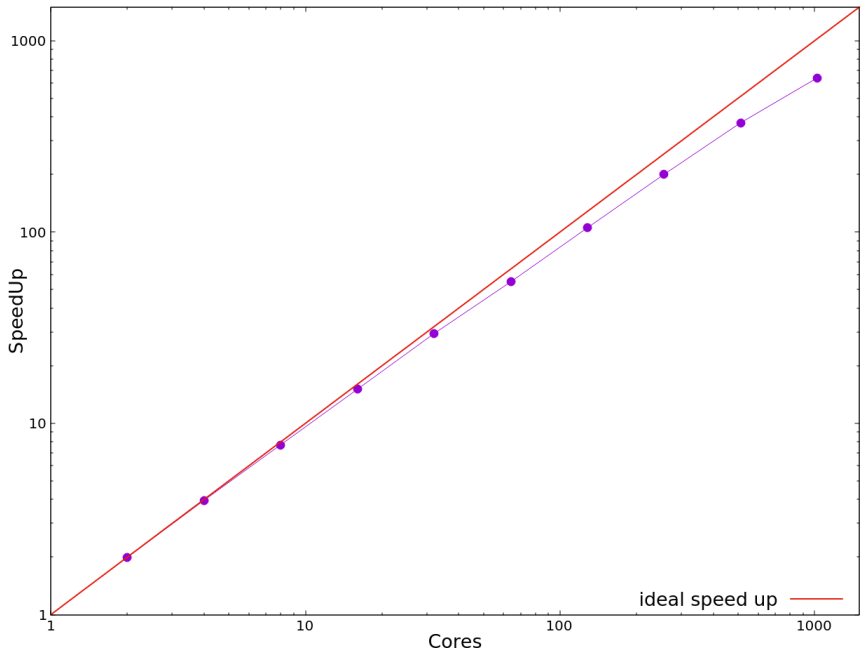


Figure 2: This is the benchmark of the band structure that we parallelised in this project.

The results show that the speed up is almost linear for the system that we used, i.e the system in the Figure 1. The code scales even better if we use a larger system. For time purposes, to have the results on time, we used this system over a bigger one.

Discussion

For the final result of the code, we have a parallel version of the band structure calculation. The efficiency of the code is very good and it is even better as the system grows larger. This product aims to treat large nanotubes where the calculations are very time demanding.

Work for the future would be to parallelise other parts of the code as well, which are also heavy on calculations, or to improve and optimise parts of the code that can be optimised.

Personal experience

The journey with the PRACE Summer of HPC was a unique opportunity to learn new and useful thing about programming. For me, as a computational physicist, that I know more in the fields of physics rather than in the programming, it was perfect to enrich my programming abilities, not only for MPI but in so many different elements. I will not refer to all of them, because the are so many, but a good example is that I familiarised myself with many different interfaces, programs and tools which they will be useful for me in the future.

Conclusion

The final product of this project is a parallel version of the band structure calculation for nanotubes by making use of the helical symmetry to further improve the performance of the code. The code uses MPI (Message Passing Interface) for the communications between the procedures in a smart way, to store the data in the correct order and take care of the dependencies that the calculation has, with other part of the program.

The band structure calculation, optimises its resources very well (Figure 2). The larger our system (nanotube) is, the bigger will be the benefit that we will see from the MPI implementation. The program can treat smaller systems without any problem, but we gave emphasis to large systems, which are very time demanding. The code can use as many cores as we want, as long as we have a big enough system to make use of them.

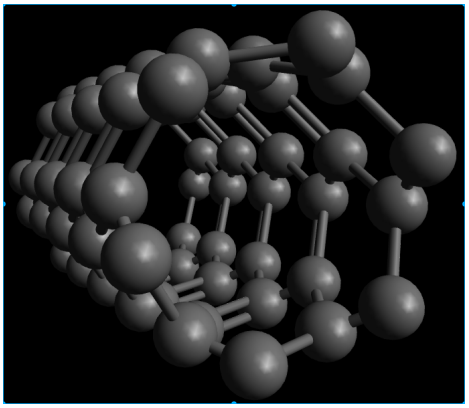


Figure 3: Carbon nanotube which we made the first runs of the program.

I don't believe there is a particular reason that I should win over my other colleagues. Everyone did a great job for the time that was given to us.

PRACE SoHPCProject Title

Electronic structure of nanotubes by utilising the helical symmetry properties: The code parallelisation

PRACE SoHPCSite

SAS, Bratislava

PRACE SoHPCAuthors

George Nikoulis, AUTH, Greece

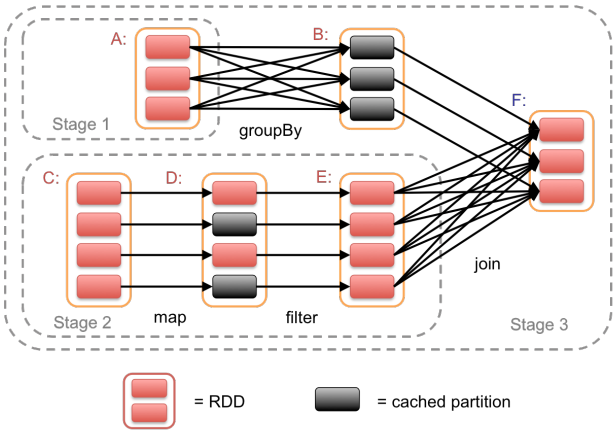
PRACE SoHPCMentor

Prof. Dr. Jozef Noga, DrSc., Comenius University, Bratislava

George Nikoulis

Comparison of HPC tools and JVM-based frameworks in terms of Machine Learning process performance on Big Data.

Machine Learning from HPC perspective



Source: Matei Zaharia slide deck on Spark at Strata conference Feb 2013

Enes Cankiri

When processing ”Big Data”, the first technologies of choice are the JVM-based frameworks, such as Apache Hadoop or Spark. Motivation behind this project is to show that HPC libraries, such as MPI or GPI-2, are at least as good or better than JVM based frameworks. We have implemented two machine learning algorithms, K-Means and Decision Tree, in order to compare the JVM and HPC approaches.

Processing Big Data is a problem by itself. Running Machine Learning algorithms on large data sets requires lots of computing time. Although runtime is important, JVM-based technologies are more popular than HPC libraries because of other reasons (runtime resilience, developers' comfort, debugging etc.), but we will focus on comparing the performance exclusively.

Big Data

Basically, Big Data have 3 properties: velocity, variety and volume. Datasets that only have large amount of samples and/or features, are not necessarily Big Data. Imagine you are receiving many emails every day. You would be happy if a computer program could mark them as spam, important, work-related, other etc. for you, and you can check them in the desired order. Emails are coming constantly (velocity) from different senders with different type of attachments (variety) and you are receiving a lots of them (volume), so it is possible to say that you have a Big Data and you need to cluster your data.

Machine Learning

Humans learn from experience, but machines learn from patterns. Machine Learning is a way to 'learn' from relationships between descriptive features of samples. There are different ways to learn from samples and different ways to find relationships between features. Learning from Big Data requires more computational power and this is where HPC comes in. We have used two different ways to learn (K-Means and Decision Tree) from samples and compared implementations of these algorithms by using different technologies.

Methods

MPI and GASPI

Basically, HPC is optimised computing in parallel environment, and the instances of the computer program running on multiple nodes need to communicate. There are some standards for the HPC communication, such as the Message Passing Interface (MPI) and/or Global Address Space Programming Interface (GASPI). MPI is a well known standard, but GASPI has some additional advantages. We have used a MPICH implementation for MPI and GPI-2 for GASPI. GPI-2 focuses on fault tolerance and efficient communication by providing non-blocking one-sided communication. This non-blocking, asynchronous approach allows overlapping of computation and communication.

For example, iterative algorithms typically need to synchronize all ranks at the end of each iteration. Every iteration thus has computation and communication phases, and during the collective communication all processes need to stop and wait for others to send/receive the data. This barrier results in idling, thus wasting of computer time.

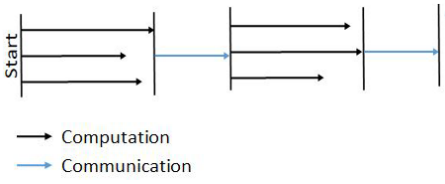


Figure 1: Phases for iterative algorithm

Data Dependency Driven

In our approach we attempted to make the iterative algorithm asynchronous, without having explicit barriers and collective communications. Doing so, we tried to maximise the overlap of computation and communications for better computer time utilisation.

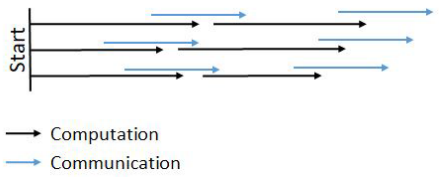


Figure 2: Phases with Data Dependency Driven approach and overlapping

Collective communication barriers are needed to make sure ranks are ready for the next iteration. If there is no general data dependency between ranks, each rank needs to know neighbours are ready. Thus, we can avoid collective communications. GPI-2 is focused on efficient communication by doing non-blocking one-sided communication which makes GPI-2 a perfect fit for this approach.

Apache Spark

Apache Spark is an open source, general purpose data processing framework. It has good relationships with HDFS and APIs in several popular languguages (Scala, Java, Python and R) make Spark good and user friendly for processing framework for Big Data. MLlib is a scalable machine learning library that utilizes Spark's API. PySpark exposes that model with Python, we have used PySpark for convenience in this work.

K-Means

K-Means is a basic clustering algorithm. Its task is to find similarities between unlabelled samples and group samples into K different clusters. Each sample is clustered to its nearest "centroid", so we have K centroid points during the program execution and at the end. The basic steps of the K-Means algorithm are:

1. Start with choosing K (random) centroids
2. Group all samples based on the distance to the nearest centroid
3. Finding mean of each group and create new centroids
4. Monitor the centroids change, and if the change is less than a threshold go step 5, else go to step 2
5. Data is labeled, finish

Every step can be implemented differently, for example we can choose K initial centroids from our sample, or we can generate random features. Also, distance and mean function should be different for numerical data or categorical data.

Data Dependency Driven approach for K-Means

We decided to spend some time on developing Data Dependency Driven approach for K-Means. K-Means have general data dependency to find next centroids, which means we can't directly apply a data dependency approach on this problem. We need to reduce this problem into a local data dependent problem, where all ranks should have different centroids to avoid synchronization phase. Every iteration will converge centroids to the center of groups, but the biggest problem is sample distributions. Centroids won't converge to the same positions as each rank will have a different distribution. We can group centroids of neighbour ranks with local centroids. So, it will form a chain communication, and after some point all ranks will converge to similar centroids.

Decision Trees

Decision Tree is the name of a structure where nodes have two types: decision or rule. Decision nodes are leafs and they come with a label. Rule nodes asks questions and branch according to an answer. There are several algorithms to build decision trees. The most important challenge in decision trees is choosing rules in the most optimum way.

- ID3 (Iterative Dichotomiser 3) uses Entropy and Information gain as the metric, it chooses an attribute for each rule node, and each category for that attribute makes a new branch.
- CART (Classification And Regression Trees) uses Gini Index as metric, it chooses a question that can be answered with yes or no. This leads to binary tree, same attribute can be used again as a rule.

We have chosen the CART algorithm for our implementation, because it is used in the Spark MLlib implementation as well. Binary trees are also implemented much easier in the C language. Gini Index requires many probability calculations over samples. So, the CART algorithm can be scaled by iterating over scattered samples.

Results

Let us start with analysis of the K-Means results. The K-Means run time can be split into three parts: **File I/O and sharing** *i.e. reading the file with samples by rank 0 and sharing to other ranks*, **main job** *main iteration procedure*, and the "**rest**", which is the overhead of memory allocation, starting/stopping of parallel environment, etc.

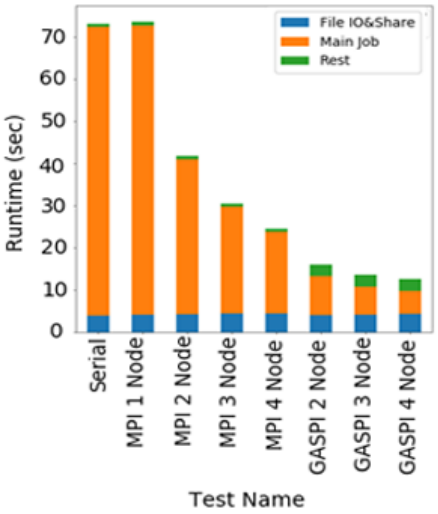


Figure 3: K-Means MPI and GASPI implementation with different sizes compared to serial code

The problem is dominated by iterating over samples. So, scalability depends on sample count per rank, so we can get the same efficiency from each rank. If we have a bigger sample size, we need to increase the rank count with the same sample count for every rank, thus we'll get same efficiency. This makes this problem weakly scalable. GASPI segment allocations takes more time than C memory allocations. The difference in "rest" part can be seen in Figure 3.

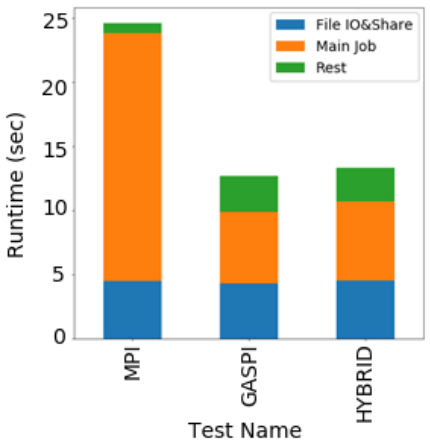


Figure 4: MPI vs GASPI vs HYBRID with 4 ranks

There is a strange result in Figure 3, GASPI has more than 4 times speedup with 2 ranks. How that can be possible? We found out that distance calculations in K-Means was the reason for this discrepancy. Both implementations use exactly the same function for this purpose. The only difference was memory accesses, MPI uses memory allocated by 'malloc' and GASPI uses GPI-2 segments. When we tried a hybrid code, using GASPI segments and MPI communication, GPI-2 segments had faster access time as can be seen in Figure 4.

Let's add PySpark and C-code serial K-means implementations to these results for full comparison. The PySpark code was nearly as slow as the serial one, see Test Environment section.

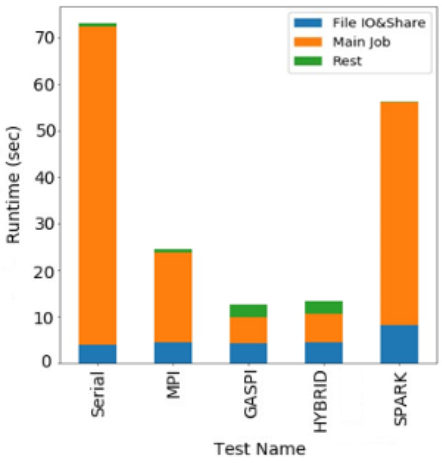


Figure 5: K-Means, Serial (single rank) vs others with 4 ranks

We also have a Data Dependency Driven approach with GASPI and a runtime comparison can be seen in Figure 6, where we can see that a Data Dependency Driven approach could be faster. This though actually depends on the distribution of data, as It may differ between datasets.

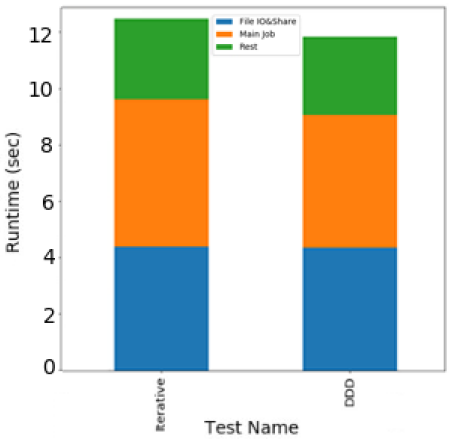


Figure 6: Data Dependency Driver approach comparison

Let's continue with benchmarks on the Decision Tree algorithm. First, the serial and GASPI results are compared in Figure 6. Unfortunately, there was not enough time for finalisisng the MPI implementation.

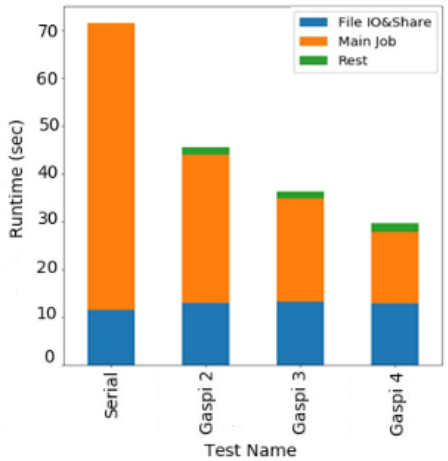


Figure 7: Decision Tree serial vs GASPI with different rank sizes

There is less memory access in Decision Trees compared to K-Means, so we do not see the "segment effect" in these results. Scalability of the Decision Tree implementation is limited by the fact, that it is iterative, just like K-Means. So, we need to keep sample count per rank while increasing the number of ranks to solve a larger problem with the same efficiency, which means implementation is weakly scalable.

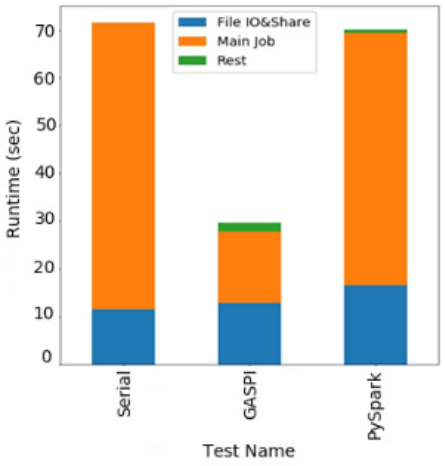


Figure 8: Decition Tree, Serial (single rank) vs others with 4 ranks

When we compare serial C-code to GASPI and PySpark runs with 4 ranks, GASPI/GPI-2 implementation is clearly faster again.

Test Environment

All tests uses ethernet connection, with infiniband installation GASPI runs could be faster.
GCC version: 6.3.0
GPI-2 version: 1.3.0
MPICH version: 3.2.1
Spark/PySpark version: 2.3.1
Hadoop version: 2.7
Operating System: Debian 9.2
Processor: Intel i5-7300HQ with 4 cores

Discussion & Conclusion

The performance benchmarks clearly demonstrates that the C/C++ compiled codes are more efficient compared to the JVM. However, if the development time, debugging options, etc. are considered, the Apache Spark is clearly the option of choice. Finally, the JVM-based technologies and HPC tools have their advantages and disadvantages, but the GPI-2 (or MPI) is a more efficient tool for processing big data, if the run time efficiency matters.

PRACE SoHPCProject Title
Machine Learning from HPC Perspective

PRACE SoHPCSite
Computing Centre of the Slovak Academy of Sciences, Slovakia

PRACE SoHPCAuthors
Enes Cankiri, DEU, Turkey

PRACE SoHPCMentor
Michal Pitonak, CCSAS, Slovakia


Enes Cankiri

PRACE SoHPCContact
Lukas, Demovic, CCSAS
E-mail: lukas.demovic@savba.sk

PRACE SoHPCSoftware applied

GPI-2

PRACE SoHPCMore Information
GPI-2 Github Repository www.gpi-site.com/gpi2/

PRACE SoHPCAcknowledgement

Write any requested acknowledgements or thanks here. Mentors should be asked for them too.

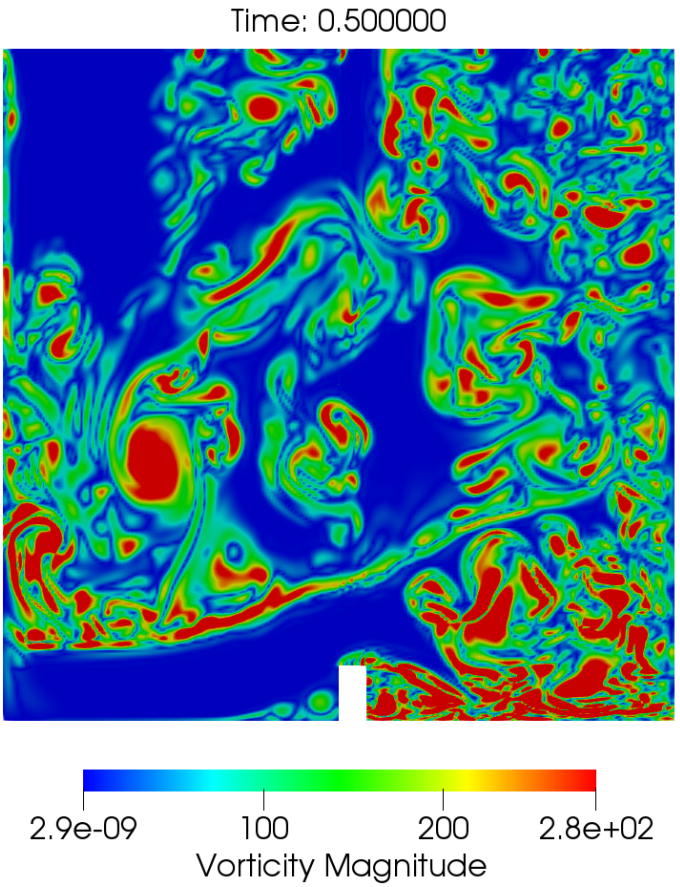
PRACE SoHPCProject ID
1808

Ever seen a postprocessing result, before completion of the run? You've come just to the right place.

Instant visualisation of CFD data with OpenFOAM

Atul Singh

CFD simulations are both memory as well as communication intensive. The situation is even grimmer when supercomputers are involved as they anyways require more communication between processors. For such a task, In-Situ visualisation offers a huge incentive as it saves a lot of memory space, by generating the visualisation while the simulation is running. This project aims to combine these In-Situ capabilities of Paraview with CFD capabilities of OpenFOAM, which is also an opensource software. Hence empowering a wide range of researchers i.e from students to companies in their CFD research.

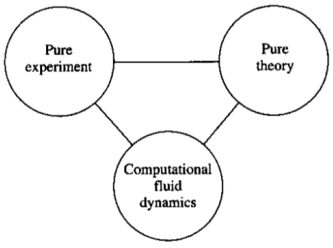


The image seen above is not a Vincent Van Gogh painting, but is a only a small result of the ginormous visualisation capabilities our modern day supercomputers have (so can this be termed as Modern Art?). The image, shows free fall of breaking of square section of water, (say, dam) under the infuelence of gravity, obstructed by a small protrusion at the bottom, at 0.5 s of the flow.

Why bother with CFD?

Consider a flight taking off from an airport. Why do you think, the next one waits a little bit after the previous one is already long gone in the air?. The answer is simple. The previous flight has left a huge whirlpool of air behind it, strong enough to make the next plane wait in line. A following good question would be how does one know when these whirlpools of air will diminish?, or rather, how weak of a whirlpool strength, would be good enough for the next plane to take off?

Questions like these underline the importance of CFD studies. Knowing the answers of which could lead to a correct take off time for your next flight. Well that is of course, if Airport support staff doesn't harass you with something else.



The three pillars

In short, one could just think of CFD simulations, or rather simulations in general, as tools that provides the means, which the complexities of theories and experiments aren't able to. And this is obviously not limited to Aerospace applications as can be seen in Fig 1.[2] These range from, but not limited to, Automotive, Health research, HVAC, Chemical processing, Sports, Marine applications, Power generation, etc.

Why is HPC involved?

When we talk about CFD simulations, they are usually a set of very long equations that are run to obtain some variables of interest, say pressure, or velocity for instance. These equations could be thought of as similar long equations that you would enter in a calculator to calculate, lets say some variable 'x'. Only slightly more complex, sometimes coupled, and quite often dependent on a previous iteration.
The calculators do it on a very small scale, while supercomputers do it on a think-as-big-as-you-possibly-could, ginormous scale. Think not only CFD simulations, but even those involving the supernova explosions and galaxy studies are performed with the help of Supercomputers. Surely they must be useful. Very very useful.

The handshake between the two

So now, we know CFD are basically a set of calculations, which HPC helps them

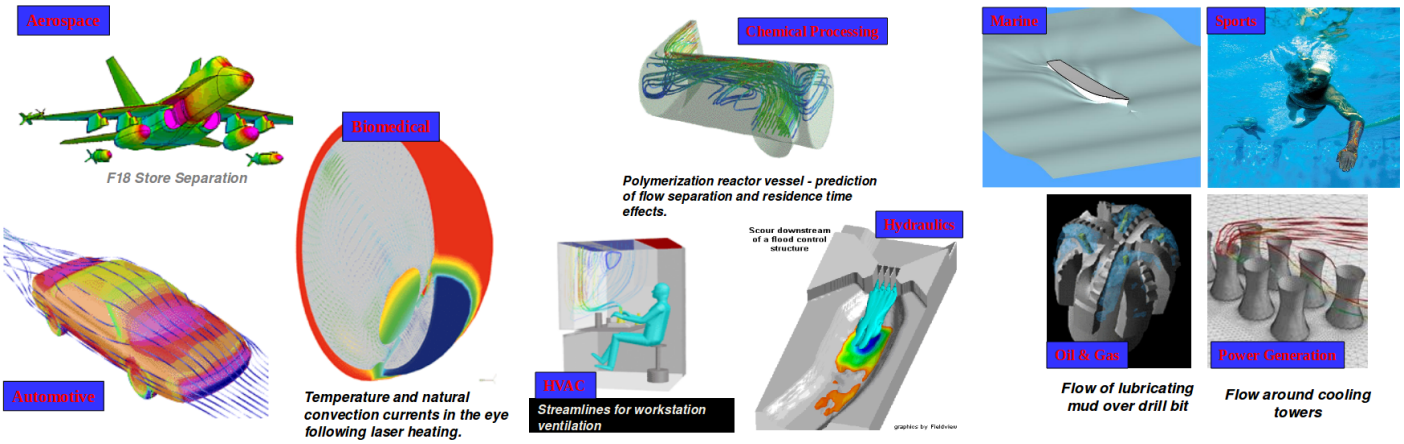
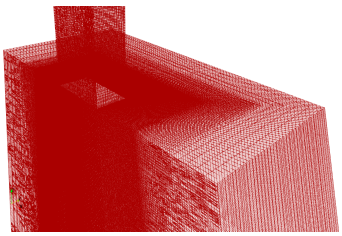


Figure 1: Some applications of CFD

calculate. The way this is done, is by employing one of the oldest trick in the book, for whatever the problem may be. "Divide and Rule". Lets see how.



The divisions[3]

The structured lines that you see above in this building like structure is called a mesh. The square or regular shaped volumes formed by these intersecting lines are called cells. These cells are the exact places where our defined equations for a variable of importance will run. The greater the number of cells, the more accurate approximations simulations will give (the denser region in the above picture). Hence, we simply divide the problem into cells, and rule it with whatever equations we want to solve.

So what's the catch?

Well, its time, another oldest ally in the book. How?, Lets just say the divisions or cells that you see from the previous image are in order of millions, and this is quite often commonplace in a CFD study, and add to the the fact that you are running a very complex mathematical equation on each of these cells, which have several unknown variables. It is bound to be time consuming to get to your results.

And when you do, you need this "post-processing" software known as Paraview that helps you see and observe the behaviour of the variable you wanted to study. Not to mention, loading the data to this software after your simulation completes in 3 days (say) will also be a slow process, because there will be loads of data and any software of course has limitations. Which is a traditional way of post processing or rather visualising the results.

What do we do then?

This is where my project comes to play. Want to observe only pressure, then use only pressure as a variable and not anything else. Seems logical, right? Don't observe all the variables, when you only need a couple of them. This is exactly what we do inside OpenFOAM (the software responsible for having those long CFD equations and solvers).

Paraview Catalyst is one such library that takes care of such a handshake between OpenFOAM and Paraview, while the OpenFOAM solver is running. The way Catalyst works, is, lets say if you have only pressure to visualise, then, catalyst lets you define only pressure as a variable.

Now if we are to combine OpenFOAM with Paraview to instantly visualise our results, it would make sense if we only took a part of features that Paraview has to provide, and not the complete software. Which is exactly what we do. Paraview has something called filters, that helps you "filter" out only specific information from the complete results that your simulations produced. But in Catalyst however, we in a way customize Paraview according to our specific needs. If, lets say, you only want to see a slice of the geometry, which shows velocity flow in 2D, you can only select those two parameters. You can then generate a python script using the Paraview-GUI, and define the parameters that saves the image in a seperate folder.

Why is this helpful?

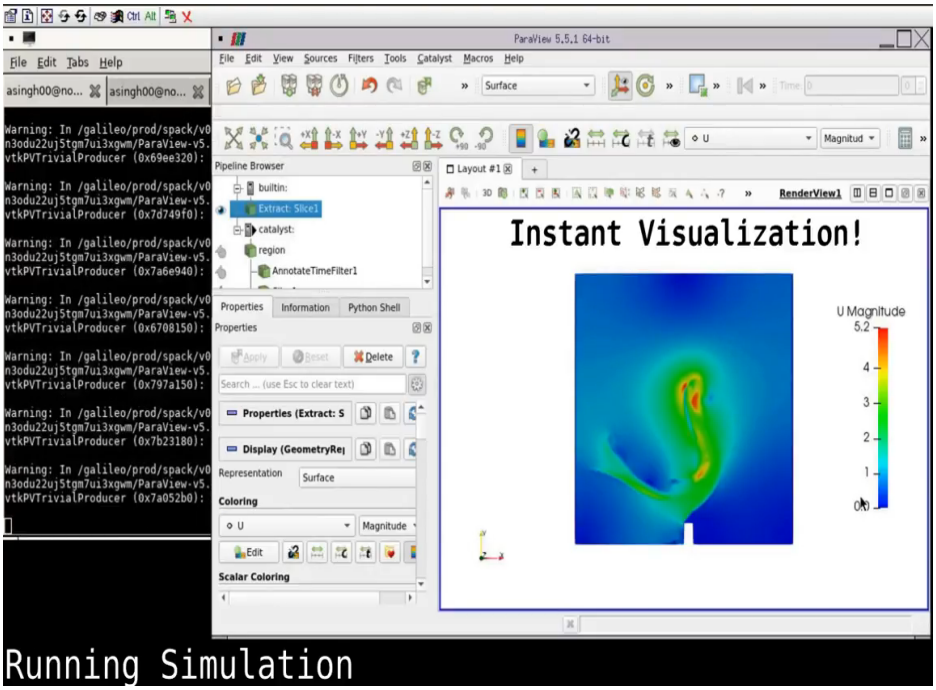
For one, It makes complete sense to use only those filters that are necessary for your results, compared to everything that Paraview has to offer. What this does is it saves memory, as every processor involved will be running the OpenFOAM solver. Having a smaller version of Paraview can make the visualisation code just like one more added calculation for OpenFOAM. Which is much more feasible. This is exactly what happens when we have included a function object as an interface in OpenFOAM to use Paraview.

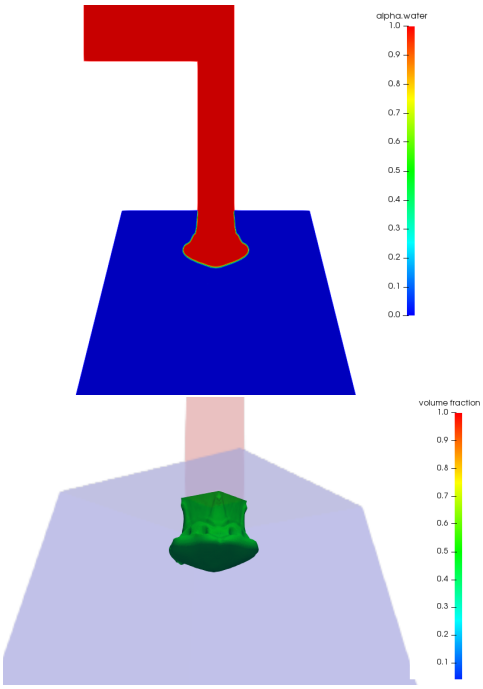
Let's test it

There are basically two steps involved

- The Script obtained from Paraview GUI[5]
- The catalyst file added with the above script

The first step after you have decided onto what visualisation would you want to see is fairly simple. All you need to do is load the Catalyst plugin. This gives a wizard of sorts that helps you generate a script specific to all the filters that you used.
The second step involves, adding a function object to OpenFOAM files, that tells it to process the above generated script





Left: The snapshot when catalyst is running with OpenFOAM. Top Right: Simplified injector tested with Catalyst,showing 2D slice of volume fraction.[3] Bottom Right: Injector with iso surface for the same volume fraction[3]

within its framework. An example can be seen from this github repository.[4]

And finally

The images seen above are the first hand results when Catalyst is being run. It can be seen when the simulations is running, the pipeline browser of Paraview (the place with the smallest blue rectangle are in above picture) has loaded the data that is defined in the script. So it is quite appealing to see the visualisation along with the running CFD simulation.

The script also defines a seperate folder where all the images can be saved. These images are helpful when one wants to make a small movie illustrating how the flow develops during the course of the flow,[6] as is often the case for a CFD simulation. Remember, this was not possible before Catalyst.

Although Catalyst is a brilliantly employed solution, it is still under development. The current testing done at CINECA Supercomputing center, under this project was the first instance where it was tested with an Exascale OpenFOAM case.

Of course, the current edition of Catalyst is not without its limitations and the documentation regarding building a custom Catalyst edition, specific to different needs, is still missing. It is also not a trivial task to build and recompile everytime an addition has been made to Catalyst. So there remains lots and lots of scope for improvement.

References

[1] Anderson, J. (2010). Computational fluid dynamics. New York, NY: McGraw-Hill.

[2] https://www.iitk.ac.in/tkic/workshop/FEM/ppt/Introduction

[3] Edelbauer, W. (2017). Numerical Simulation of cavitating injector flow and liquid spray break-up by combination of Eulerian-Eulerian and Volume-of-Fluid methods. Available at: http://www.elsevier.com/locate/compfluid [Accessed 31 Aug. 2018].

[4] https://github.com/Atulsingh92/SouCase-OpenFoam/blob/master/system/catalyst

[5] Paraview Catalyst User Guide

[6] https://www.youtube.com/watch?v=r-u9NJc-pJIlist=PLhpKvYInDmFXUyp_pWBM-h1NCD6GEUfgpindex=9

PRACE SoHPCProject Title

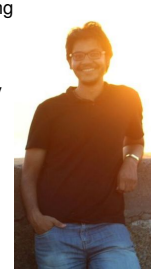
In Situ visualization of CFD data using OpenFOAM.

PRACE SoHPCSite

Cineca Supercomputing Center, Italy

PRACE SoHPCAuthor

Atul Singh, University of Rostock, Germany. contact: atulsingh92@outlook.com



Atul Singh

PRACE SoHPCMentor

Dr Fedrico Piscaglia Politecnico di Milano, Italy

PRACE SoHPCSoftware applied

OpenFOAM 5.0, Paraview 5.5.1, Catalyst, Blender 2.79, Audacity.

PRACE SoHPCMore Information

SummerofHPC website , Paraview website

PRACE SoHPCAcknowledgement

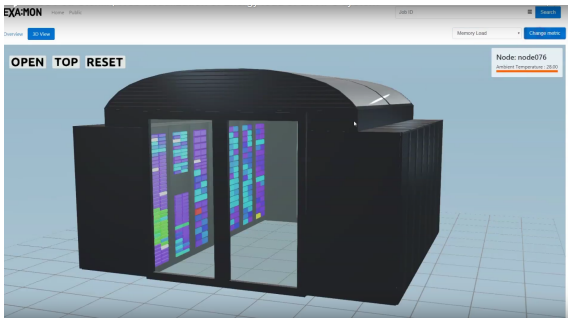
Sincere thanks to Dr Fedrico Piscaglia and Special thanks to Simone Bna. Honorable thanks also to Paola Alberigo, Francesca Delli Ponti, Giuseppa Muscianisi, Massimiliano Guarassi, Luigi Calori, Ivan Spisso, Christiano Padrin and Luis Sanchez. Default thanks to my Parents.

PRACE SoHPCProject ID

1809

Visualising HPC System's Power

Nazmiye Arslan



Energy efficiency is one of the timeliest problems in managing HPC facilities. The web page which was already developed includes some statistics about the jobs running in the selected system and a 3D view of the energy load and other observables of the GALILEO cluster. In this project we aim to realise a web interface to plot the energy efficiency observables of the D.A.V.I.D.E. cluster based on a POWER8 + Nvidia GPUs architecture.

D.A.V.I.D.E is a cluster comprised of 45 computer nodes, 1 admin node and 1 login/master node. Each compute node hosts 2 IBM POWER8+ processors, 4 NVIDIA Pascal P100 SMX2 GPUs, and innovative technologies such as the monitoring framework and the liquid cooling components.[2]

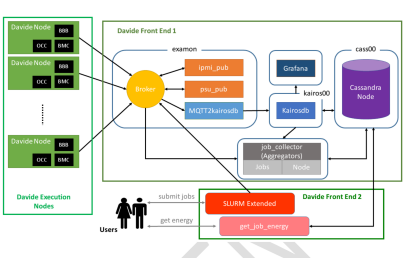


Figure 1: D.A.V.I.D.E. Architecture

The general structure of the monitoring framework based on D.A.V.I.D.E is shown in Figure 1. The following sections provide a more detailed description of the components of Examon (a framework for analysis for data collection, storage, and exascale clusters).[2]

The problems in HPC system management for energy efficiency include many technical issues such as web visualization, interaction with HPC systems and timers, large data analysis, virtual machine manipulations and authentication protocols. In the framework of SoHPC 2017, a web interface was already developed that will show the required observables.

In this project, the aim was to develop the capacity of this web tool and adapt it to the Tier-0 Marconi cluster and KNL architecture. At the same time, it is aimed to implement a web interface to draw the energy efficiency observations of D.A.V.I.D.E. This tool could help HPC system administrators to optimise the energy consumption and performance of their machines and to avoid unexpected fault and anomalies of the machines.

In the race for more performance, super computers become limited in Power and Energy. Until recent years, every new supercomputer in the world points to an increase in power consumption.[1]

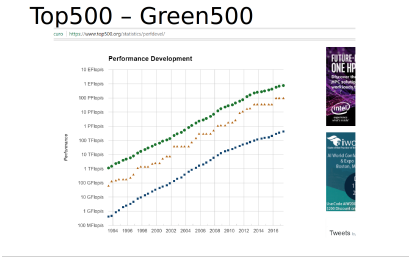


Figure 2: TOP500-GREEN500

Then so, The Power is important in HPC system due to be an important cost, limit today super computers capacity and performance.

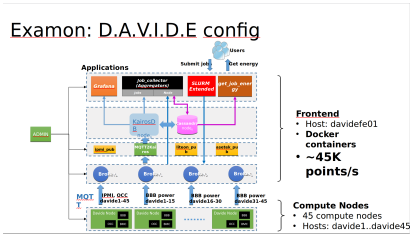


Figure 3: EXAMON-D.A.V.I.D.E.-Config

As seen in Figure 3, D.A.V.I.D.E. uses three complex database platforms for monitoring. Examon is the current framework for data collection, storage and analysis for exascale clusters, and these are described in,[2]

Sensor Collectors are low level components that read the data from several sensors. These are used by the MQTT Protocol. The specific sensors include IPMI, AMESTER, BBB and other collectors.

Storage Layer is composed of:

1- Grafana, which supports KairosDB as data source by means of a plugin and allows to visualise every stored metric.

2- Cassandra is the backend of KairosDB and it is where the fine-grain and aggregate metrics are physically stored.

3- KairosDB is a single service container that provides an instance of the KairosDB.

Methods

The current monitoring metrics are shown and one can see the job power monitoring over the last 7 days. Figure 4 shows the running grafana dashboard.

Figure 4: Jobs on Grafana-Dynamically Web Page

More meaningful values and objective viewing of job power consumption and its components are needed. Thus, the Snapshot Dashboard was developed.

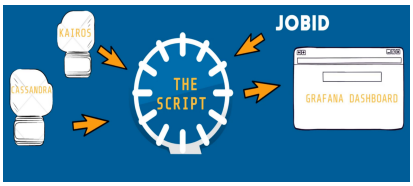


Figure 5: The project scheduler

The main method is to connect to the Cassandra database and to get all data about a specific job. Then to connect to the Kairos Database that consists of more meaningful data than Cassandra. Finally, he general dashboard for a job is generated using information from the Cassandra data source.

The project was developed on Python as requests, json, Cassandra, ConfigParser, Webbrowser main libraries of python.

The snapshot dashboard was designed using four rows.The first row is to see general information of the job as a text graph. The second row is to see all total power of each metric for the job in the form of a pie chart. The third row displays the total power of each metric for the each node which runs for a job. The last row shows the total power for each node for each metric.All information can be seen on the dashboard.

Grafana

Grafana has a HTTP API for the dashboard and the dashboard is created, updated and deleted with using get, post, delete python library requests after it connects to the Grafana server. The dashboard uses json libraries to modify and submit any new dashboard views for a job using grafana.py functions - as shown in Figure 6.

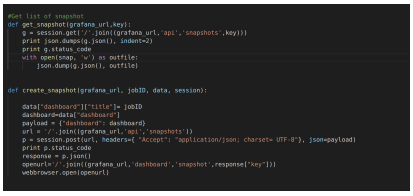


Figure 6: Get Grafana Dashboard and Create

Cassandra

Cassandra aggregates information with the purpose of long-term storage and there are 2 types of aggregated data:[2]

1- Job-aggregated information 2- Node information

It contains three distinct tables to store the physical information generated by the IPMI, BBB and OCC sensors;[2]

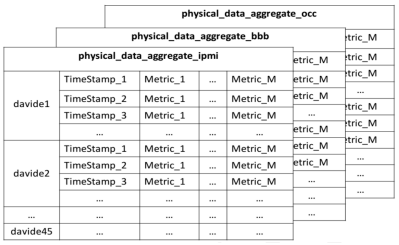


Figure 7: Cassandra Tables

The format of the data is taken from CassandraDB for the IPMI and OCC table. The BBB table contains total instantaneous power for the job.

Combining Data and Calculating Partial Total Power

Figure 8 shows the Json template dictionary created and for every new dashboard job used to collect all general power information.

Figure 8: Template Json Dict

The node information about each job is taken from CassandraDB and average power is calculated. This template with node job information is used to create job visualisation on the Grafana dashboard.

Figure 9: The Python Code That To Show General Information for First Row on Job Dashboard

In Figure 9, the general information of the job is determined.

Figure 10: The Python Code That To Calculate Total Power for Second Row on Job Dashboard

In Figure 10, the second row on the job dashboard displays the total power breakdown with each metric.

Figure 11: The Python Code That To Calculate Total Power for Third Row on Job Dashboard

In Figure 11, the third row on the job dashboard visualises total power with each node metric.

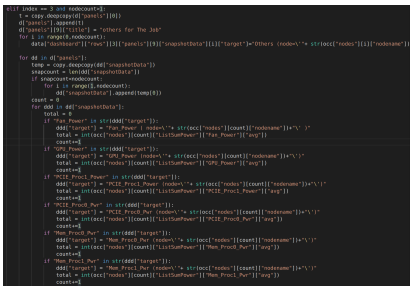


Figure 12: The Python Code That To Calculate Total Power for Second Row on Job Dashboard

In Figure 12, the last row on the job dashboard displays the total node power calculated and added for each metric.

Results

The Job dashboard is created automatically from the python script by providing the job ID. The job can then be followed from the dashboard view objectively. Results of the dashboard, are as shown in Figure 13-14-15-16-17.

Figure 13: First Row on Job Dashboard

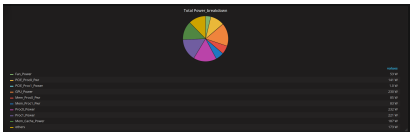


Figure 14: Second Row on Job Dashboard

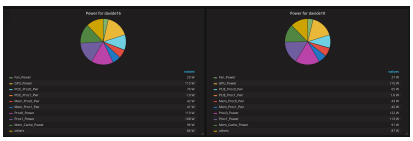


Figure 15: Third Row on Job Dashboard

Figure 16: Last Row on Job Dashboard

Figure 17: General View of Job Dashboard

Discussion & Conclusion

The Grafana Web Visualisation was developed successfully. The performance and power monitoring of super computers are more meaningful and objectively. HPC system administrators at CINECA will use this visualising to optimise energy consumption and components and to avoid unexpected faults.

References

[1] Bartolini, Andrea, et al. "The D.A.V.I.D.E. Big-Data-Powered Fine-Grain Power and Performance Monitoring Support" 2018.

[2] PRACE-3IP PCP For the award of a Pre-Commercial Procurement contract concerning R&D services on “Whole System Design for Energy Efficient HPC” SCHEDULE B - DELIVERABLES FOR PHASE III, 2017.

PRACE SoHPC Project Title

Web visualization of Energy load of an HPC system

PRACE SoHPC Site

CINECA, Italy

PRACE SoHPC Authors

Nazmiye Arslan, DEU,Turkey

PRACE SoHPC Mentor

Dr. Andrea Bartolini, UNIBO, Italy

Nazmiye Arslan

PRACE SoHPC Software applied

Grafana, Python Cassandra KairosDB

PRACE SoHPC Acknowledgement

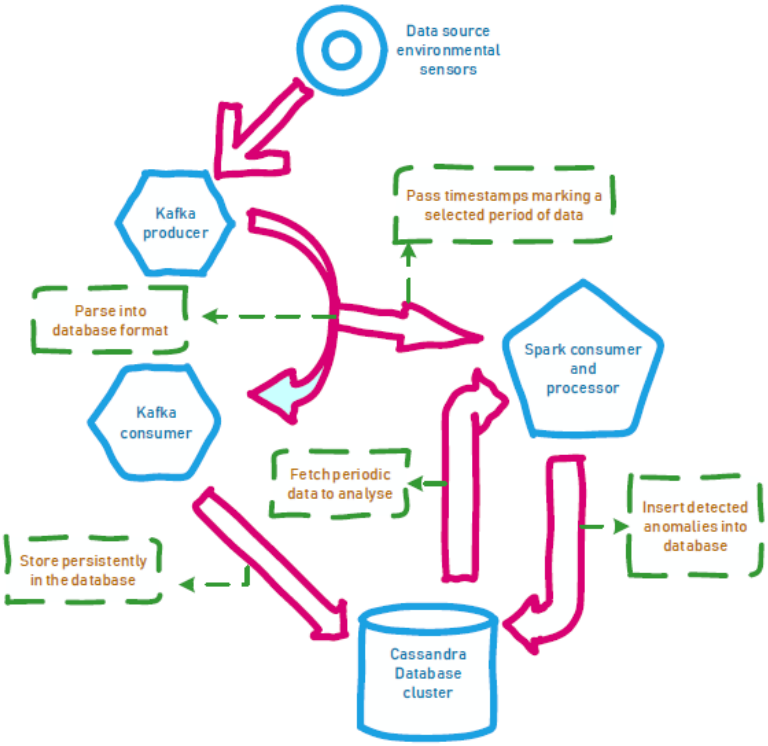
I would like to thank my site mentors at CINECA, who has hosted me on this project, and the project mentors who supported me.

PRACE SoHPC Project ID

1810

Data streaming for IoT

Jakub Mojsiejuk



The EMB project aims to highlight latest data engineering techniques. Multiple environmental sensors scattered around the UK send various types of data such as groundwater levels, water pH or seismic data that require high-efficiency online streaming and anomaly detection.

The amount of data produced by IoT devices apparently grows exponentially.[1] As a result, current machines cannot keep up with processing of the data. Therefore, data scientists, engineers and software developers must come up with even more effective streamlining architectures to compensate for the lack of power. The term "data architecture" reaches beyond the standard programming concepts and might describe a bulk, a suite of compatible solutions that combined together create a reliable platform for processing, analysing and visualising data. Despite this appearing oddly ambiguous, this is a true nature of modern data solutions. They are highly modular, tailor-made to fit a particular solution and can include machine learning models, resilient NoSQL databases or parallel processing pipelines. There is a great selection of open source software released and developed under Apache licence, including Spark, Kafka, Cassandra or Avro that are used in the project described in this article.

In the particular example of the environmental monitoring baseline (EMB) project, it provides a robust platform for analysing data which is of an uppermost concern. Many sensors suffer from an occasional failure, for instance, due to the low battery levels or a hardware fault. Such events introduce anomalies into the data sets and some of them could trigger a false alarm. As a basic example, water pH might have become unusually high not due to some pollution in the river but because one of the sensors operates on a low voltage for some long period of time.

Therefore, we need a multi-layered platform that ensures the following:

Fault-tolerance

When some major error occurs during the processing, the architecture should not crumble, but handle it *gracefully* and proceed with the analysis.

Robustness against anomalies

Singular anomalies or null data points cannot obscure the operation or affect the result of analysis.

Persistence

All of the processed data, as well as the raw data must be stored persistently, so that the accumulated database can serve as a learning material for other analysis algorithms.

High reproducibility

High degree of modularity enables the architecture to be containerised. Providing containerised applications assists in security, easy automation and reproducibility of results.

Easy scalability

A high-throughput architecture must be a scalable one. It has to operate at a standard rate, regardless of the number of data sources. Whether it is 10 or 1000 environmental sensors, the data flow must be swift, quickly processed and reliably stored.

Methods and software used

A great advantage of modern data architectures is their modularity which roughly means that instead of one huge and monumental bulk of code, we designate the particular software functionalities to the highly specialised services and applications. We can develop software in the most efficient frameworks for these specialised services. From both the design standpoint and the development approach, modularity provides much needed flexibility and workload division. It also allows for convenience when explaining the resulting framework, because we can describe each of the components separately, without reference to the actual data abstraction and all relations will still make sense. We will go through the most important constituents of the system, laying out its role and significance in the bigger picture.

Apache Kafka

Kafka is the heart of many systems. At its base, it is a communicating service - some applications send messages and other receive them. However, the way Kafka is constructed allows for a faultless communication between services. You can think of Kafka as a typical messenger application like, for example, WhatsApp. We have producers that put serialised, binary messages into *topics*, the equivalents of the conversation channels and consumers, that retrieve the data from those topics. The Zookeeper server, an integral part of every Kafka communicating system, provides the synchronisation of messages and settings across the applications. Why is Kafka is so indispensable for our architecture? Imagine that we need to write different parts of our software in different languages, frameworks, technologies, even sometimes for different operating systems - all of that to provide the aforementioned modularity. Envision further, that it needs to operate at the same time, be fault-tolerant at high rate of data transmission. Keeping in mind, the different standards for each of the used technologies, that is a textbook software engineer's nightmare. Thus, by using a specialised service, with a standardised, widely supported, almost language-agnostic, customised messaging like Kafka we have just saved ourselves much time and most importantly - our sanity.

Apache Avro schemas

In order to enhance Kafka's performance, we use Avro schemas. Schemas serve as an additional protection against invalid data formats. Since we will put our environmental readouts into the database, the proper database format of the data has to be ensured. For example, we would not like to have time stamp to be null, because in the later stage, querying the data from the database using time aggregation will not return all results due to ill-formatted values. For similar reasons, we need to know what structures represent the data that is currently being fed into a given Kafka topic - pure Kafka messaging happens solely via serialised, binary strings - no mention of integers, floats or boolean. Therefore, schemas serve as structures that hint the Kafka Consumers on how to interpret the binary data that has been fetched from the topics.

Cassandra database

We use a distributed, no-SQL database to persistently store the environmental data. The Cassandra project, although still very early at the maturity timeline, possesses a valuable advantages such as: multinode data storage which comes in handy for delocalised EMB databases in the future, quick atomic read/write that ensures that all writes and reads happen almost as transactions - when one process reads the data and the second modifies it, such a situation will not cause undefined behaviour as it sometimes happens in other no-SQL database architectures.

Singularity containers

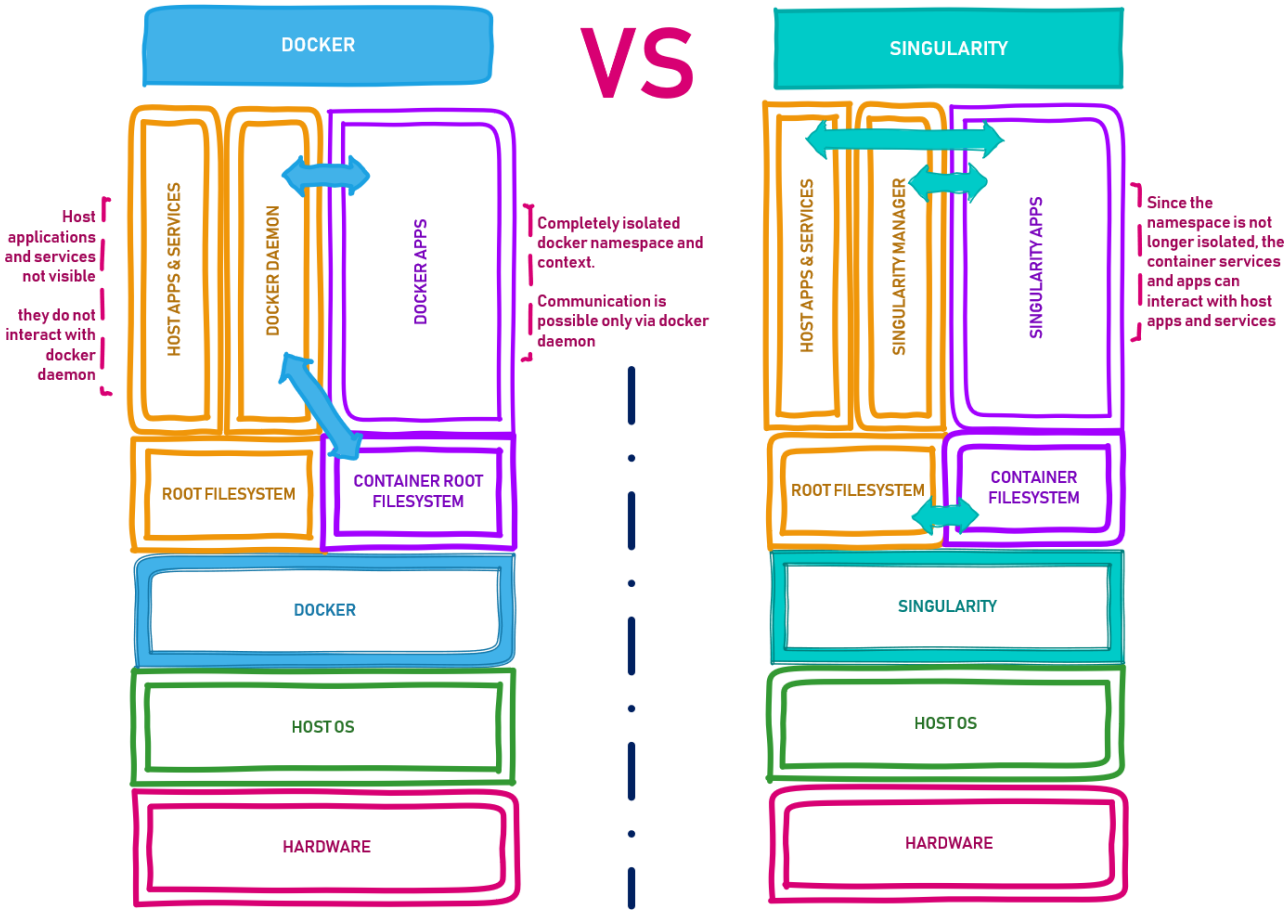
For the last several years, container services have been gradually gaining on significance. They combine both the usability of the virtual machines while maintaining lightweight distributions and a quick setup-up time. Such an impressive performance have been achieved thanks to the introduction of *c-groups* to the Linux base kernel features. In a nutshell, there was a way of separating the resources of the host machines programatically, without emulating the guest operating system. Containers run the container daemon that acts as a *membrane* for container applications to communicate with the host kernel. There is a very pronounced difference between Docker and Singularity containers. While Docker is currently the most popular container service, for HPC applications we use Singularity instead and there is a number of reasons why Singularity has an advantage over Docker in such a context. Usually, the standard resource allocation on HPC happens not via virtualisation but via direct assignment of these resources which means that the user maintains the user permissions he/she had on the login node. In other words, there is no elevation in user capabilities when the resources are assigned and hence many struggle to use the full potential of the data architectures on HPCs. This is a serious problem since most services require root privileges to run. Docker is designed to run from an administrator context - as it is a natural way of providing production environments. Here comes the great feature of Singularity - containers do not run with root privileges but can stilluse external applications that were run from the context external to the container's namespace. Furthermore, Singularity enables access to shared networking context or the root file system of the host OS. On the other hand, in Docker we have to specify the exposed ports and isolate the disk volumes in advance and even then they will exist exclusively in the container context i.e. we won't be able to reference them outside the container or communicate with the host applications. So how do we actually use the containers? We define something we call a Singularity file, equivalent to Dockerfiles in Docker. In that file we simply *define* our container by specifying: the system, installed applications, environment variables and procedures for pre- and post-installation setup.

Spark

Spark is a framework for large-scale processing. It enables parallel code execution and provides libraries to handle large chunks of data. In the EMB project we have decided to write the Spark code in Scala language that exhibits the functional nature of parallel processing. Particularly, we have implemented a SH-ESD algorithm using Spark.

SH-ESD

Anomaly detection is a really important topic in real time data analysis. Coming up with a robust algorithm that is not heavily solution specific is an extremely difficult task. It is actually quite interesting to investigate a plausible algorithm development and reasoning. Standard real time-data analysis requires the time windows to operate - since we want to predict the future or learn about the current state in a greater detail, we need to have a reasonable model - that has at least some noise removed. Then we



Comparison between Docker and Singularity. While Docker is easier to set up, Singularity allows for greater integration with host system resources. However, the future of containers is container orchestration, and currently Singularity does not support any orchestration at all.

might apply 3σ rule to filter the outliers. One way to do so is using the running average. As we move in time, we use n values to calculate the current one - and it is easily expressed mathematically:

$$z(t) = \frac{1}{n}\sum_{k=0}^{n-1} x(t-k)$$

Fair enough, but we can equally easily imagine, that even in the simplest case - an extreme value in our data set approaching infinity completely obstructs the result. We expect the moving average to reflect a pattern in our data, filtering some of the white noise. Therefore, a more robust technique is used, namely *PEWMA* that stands for Probabilistic Exponentially Weighted Moving Average and basically can be written as let $P_t = Pr(x = x(t)|x(t) \sim X)$

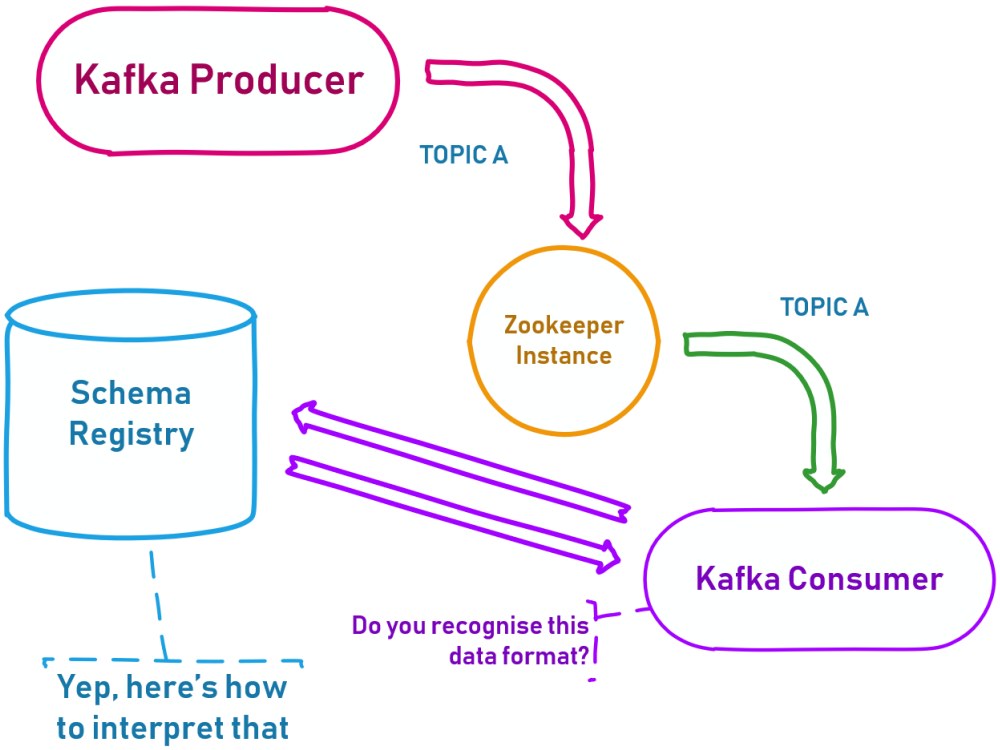
$$z(t) = \begin{cases} z(t) = x(t), t = 1 \\ z(t) = (1 - \beta P_t)x(t) \\ \quad +(\beta P_t)z(t-1), t > 1 \end{cases}$$

It roughly means that the current value of the average is dependent on the present value but also on the last one - all of that mixed with the weighted probability (assuming that $x(t)$ can be modelled by some X distribution). But then again, despite the modification of the original moving average, we are not able to provide a reliable smoothing - for example, a luxurious assumption that $x(t) \sim X$ is unlikely to take place. However, last year, engineers at Twitter have developed an advanced model called SH-ESD for detecting anomalies in any type of real-time data.[3] SH-ESD stands for Seasonal Hybrid Extreme Studentized Deviate and the name pretty much says it all. The approach assumes hypothesis testing. As in standard statistical test we have null and alternative hypothesis:
H_0: no anomalies in the data set
H_1: k anomalies in the data set
and to reject the null hypothesis we will use the t-student test. In each iteration, we check for anomalies, asking the statistics to reject the hypothesis for the anomaly $i = k, k-1, ..., 1$. Like in classical t-student test, our value of lambda is expressed as $\lambda \sim f(n, k, t_{p,n-k-1})$ - so very much with each iteration there is a greater chance of the next anomaly being not rejected since we decrease the degrees of freedom thus increasing the value of λ. We compare λ against critical value $C_k = \frac{max_k|x_k - x_{mean}|}{s}$ where s is variance. If $\lambda > C_k$ then the null hypothesis is said be rejected. Now, all of that above is the ESD part of the algorithm. There is also a problem of mean robustness - as mentioned in the moving average discussion above. We can replace the mean with MAD which stands for Median Absolute Deviation (MAD) - for it happens that median is much more robust than the standard mean in tolerating extreme values, providing we have more *non-extreme values* than we have *extreme values* (speaking precisely, at least 49 percent more). Simple definition of MAD:

$$MAD = median_i(|X_i - median_j(X_j)|)$$

Whatsmore we can further estimate standard deviation using mad:

$$\overline{\sigma} = b \times MAD$$



The schema registry provides an appropriate schema for Kafka Consumer to interpret data

where $b = 1.4826$. Hence, from now on, we use MAD instead of mean. Standard deviation estimate is of additional use, if we want to put an extra constraint on anomalies by utilising 3σ rule and also as a terminating condition to guard against false detection of monotonous data patterns. (*A hint: replace* s, x_{mean} *in* C_k for $\overline{\sigma}$ and mad). We have all necessary steps to reproduce the SH-ESD algorithm - for this particular algorithm we have to assume that the data model is additive i.e.

$$X = S + T + R$$

and also that there is fewer that $\frac{n}{2} - 1$ anomalies in the data set where n is the number of data points. To decompose the series into its constituents we use *STL* which is Seasonal-Trend Lowess and it is an algorithm that employs the use of lowess - local regression with k-means algorithm to extract trend and seasonality. For more details on models and *STL* go to the notebook study here.[2] Then for each step (number of steps equals the number of expected anomalies), we extract the seasonal element (hence seasonal in the name), compute the median of the data, then compute the residual by $R_x = X - S_x - X_{median}$. Finally, we perform the hypothesis testing on R_x using the ESD statistics that we have derived above. Notice how X_{median} serves as a sort of *trend* removal. And that's all when it comes to SH-ESD. Despite being complicated, which rarely works in the algorithm world, SH-ESD is extremely effective and robust - hence its widespread use for the big data analysis.

Discussion & Conclusion

Concluding the above outline, it can be intimidating when faced with a challenge of designing an effective processing framework. However, there are some tools available that will come handy regardless of the particular use case. Naturally, there is no good data-agnostic system, so a lot of thought has to be put in visualisation, analysis and synthesising the data sets. This is also the case with EMB project - all sorts of different data types require different management approaches. Future work can therefore focus on integrating a data analysis and visualisation system as a separate module in the framework.

Acknowledgements

I would like express gratitude to the whole EPCC staff, with special thanks for Dr Amy Krause and Dr Rosa Filgueira for supervising the project. Special thanks go to British Geological Survey for putting the environmental data for public usage.

References

[3] J. Hochenbaum, O. S. Vallis, and A. Kejariwal, 'Automatic Anomaly Detection in the Cloud Via Statistical Learning', arXiv:1704.07706 [cs], Apr. 2017.

[2] https://github.com/LemurPwned/emb_environment/blob/master/SH-EHD.ipynb

[1] Stack, T. Cisco blog: Internet of Things (IoT) Data Continues to Explode Exponentially. Who Is Using That Data and How? https://blogs.cisco.com/datacenter/internet-of-things-iot-data-continues-to-explode-exponentially-who-is-using-that-data-and-how

PRACE SoHPCProject Title
Official title of the project

PRACE SoHPCSite
University of Edinburgh, Scotland

PRACE SoHPCAuthors
Jakub Mojsiejuk, AGH University of Science and Technology, Poland

PRACE SoHPCMentor
Amy Krause, University of Edinburgh, Scotland


Jakub Mojsiejuk

PRACE SoHPCContact
Jakub, Mojsiejuk, AGH University of Science and Technology, Poland
E-mail: jakubmj@student.agh.edu.pl

PRACE SoHPCProject ID
1811

Job Scheduling Simulator for HPC

Conor O'Mara

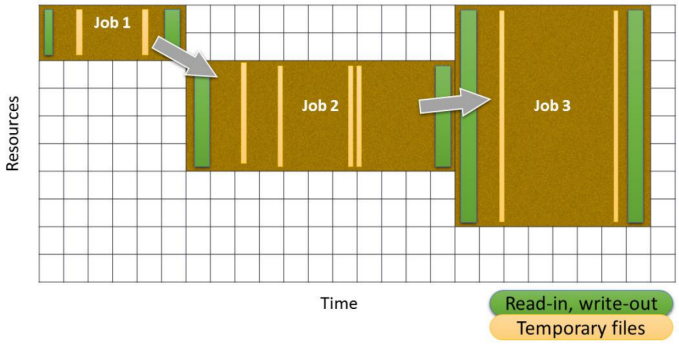


fig2

A diagram of jobs running on a HPC machine, with certain nodes left vacant while others performing multiple jobs. Note how the diagram highlights the time spent Reading and Writing, this can be reduced by using NNVRAM if the same nodes are used again. This is one of the topics of investigation of this project.

The study of how jobs get allocated to different nodes could maximise the usage of HPC resources. This project explores this by testing different job scheduling algorithms.

Introduction

High Performance Computers consist of many nodes which themselves consist of many processors. When a person wants to do some work on a HPC system they are allocated some nodes on which they can run their code for whatever their purposes may be, a person may be given 8 nodes typically. To give some scale about the capacity of the HPC system called 'Archer' at EPCC, it has 4920 nodes on which users can run simulations, calculations or whatever their desire may be. Each node has 24 processors, so that's over 100,000 processors.

There are lots of jobs running on Archer all the time, and also lots of requests for nodes to be allocated. So it is of vital importance to study how these nodes are allocated to different users so that we minimise the number of inactive nodes during a time slot and maximise the number of jobs performed.

Different users may request different numbers of nodes (depending on the size of their project) and different lengths of time for how long they will need them. Suppose Alice requests 100 nodes for 8 hours and then Bob requests 10 nodes for 2 hours. If we were to operate on a first-come first serve basis (a very näive approach), then both Alice and Bob must wait for the 100 nodes to become free before Alice can run her jobs and then Bob can run his. However, suppose 20 nodes were free the whole time and Alice must wait more than 2 hours before 100 nodes are free to begin her job. A more efficient algorithm would allow Bob to skip Alice in the queue and let him perform his job on these nodes that she can't use. He will not be delaying the time at which she begins her jobs as she will still be waiting for the other 80 of the 100 nodes to become free when Bob is finished. This is a very simple two job example to demonstrate how we can have 2 different job scheduling algorithms. Clearly the second algorithm will reduce the overall time to perform the two jobs in this situation. This algorithm is called backfilling but I will explain this in detail later. So in summary, the HPC simulator enables the study of behaviour inherent in a HPC system without using up precious node time.

HPC Simulator

In this project, we use a HPC system simulator to test different job algorithms. **Why so?** HPC systems are expensive and in high demand. So in order to avoid using up precious nodes resources, a simulator of a HPC has been developed. The goal of the project is to use the simulator to test different job scheduling algorithms to assign jobs to nodes so that we can maximise the number of jobs completed on the HPC system in a certain time period. Equivalently, this also means to minimise the time for which nodes are left vacant in the same time period.

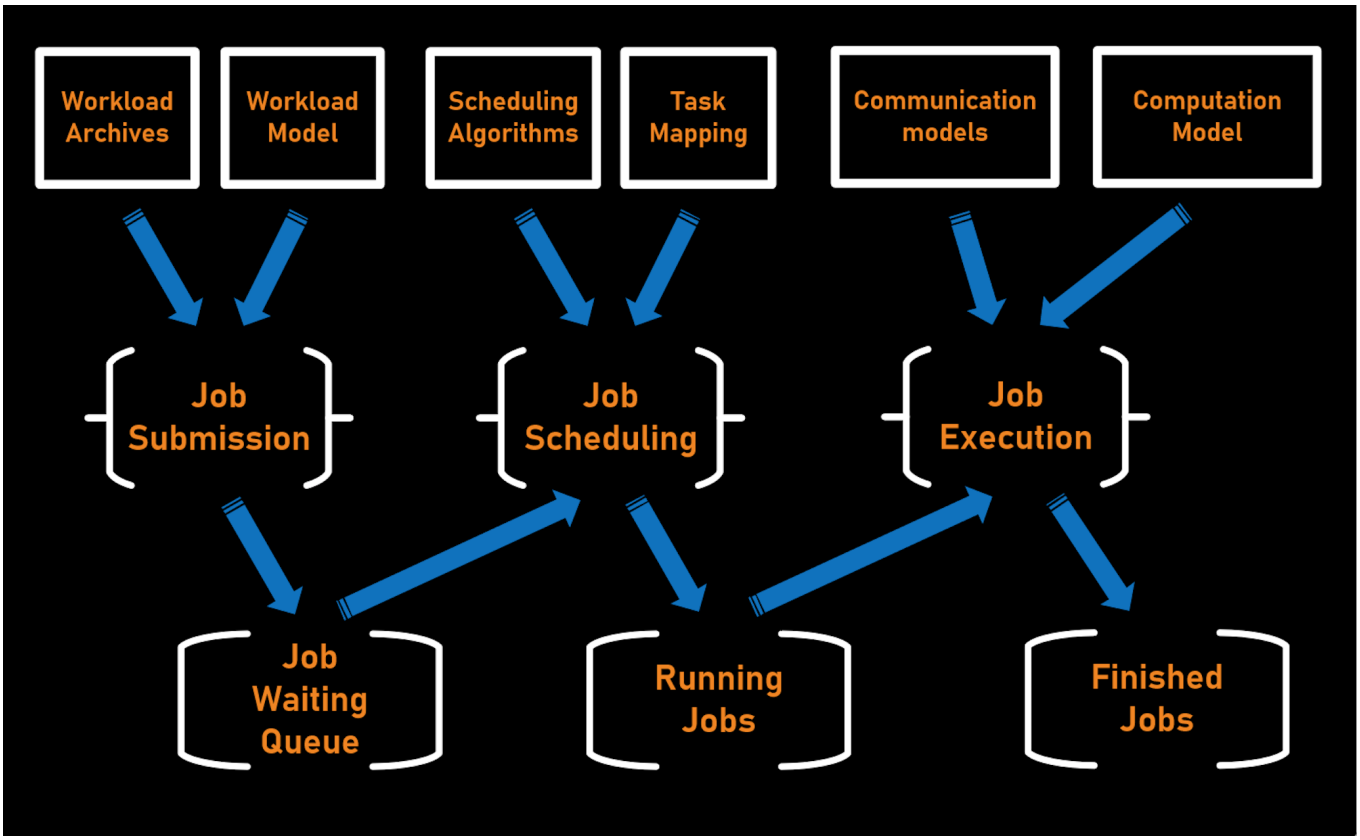


Figure 1: Design on the simulator and the main three modules and how they connect to the queues, archives and models.

Simulator design

The HPC simulator uses multi-dimensional arrays to represent the nodes and if they are busy or vacant as a function of time. This diagram conveys a nice way to visualise this. Every Job is specified by its id, size, arrival time, requested time to run, application name, application arguments, queue name and arguments related to job scheduling.
Currently, we use a fixed runtime for each job trace, and cannot simulate information on execution such as length, disk IO operations, size of input and output data, all of which help determine job execution time on the given resources. Rather here, each job has a fixed runtime, however in future the simulation of job execution would be an important addition.

There are 3 main modules to the simulator;

- Job Submission
- Job Scheduling
- Job Execution

Job Submission

The job submission module contains the workload models and workload archives. Testing different synthetic workloads gives a more rigorous approach analysing algorithms. We will discuss this more in the next section.

Job Scheduling

The job scheduling module contains the job scheduling and task mapping algorithms. Task mapping determines which compute nodes to run jobs on. The scheduling algorithms are run in a separation simulation process.

Job Execution

Finally, the job execution module contains the simulator's communication and computation models. Finished jobs are placed among all finished jobs for statistics collection.

Workload Models

Our job submission module is served by parallel workload archives from logs of job traces on ARCHER. Furthermore, more variety for testing different algorithms can be gained by using workload models. For example, one could have a synthetic workload with an exponential distribution for job inter-arrival time a normal distribution for job size and runtime. This would cause an initial bottleneck.
Another example, plotted below is one of the first such models published by Calzarosaa Serrazi (1985). The job arrival process is a function of the time of the day. The dip that occurs in this plot mimics the lunch time break in the day. The formula looks to have been generated by multiparameter regression. But this model is underspecified, it does not give job runtimes or specify the number of nodes for each job. Many current models do, and come with written code.

Job Scheduling Algorithms

Initially the simulator only had the first-come-first-serve(FCFS) algorithm. In this project we strive for better performance, so let's look at some other algorithms. First off, we have a slight adaption on FCFS, with a best fit which goes to the next job in the queue if the head of the queue can't run because of lack of resources. Next, we have shortest/longest job first (SJF/LJF). The score-based priority algorithm sorts jobs according to scores where we incorporate a fair share scheduling weight to adjust score based on the total number of compute nodes requested by the user, the number of jobs they are running, their recent history and the fraction of their jobs completed. Next, we have multi-queue priority which incorporates numerous queues with different

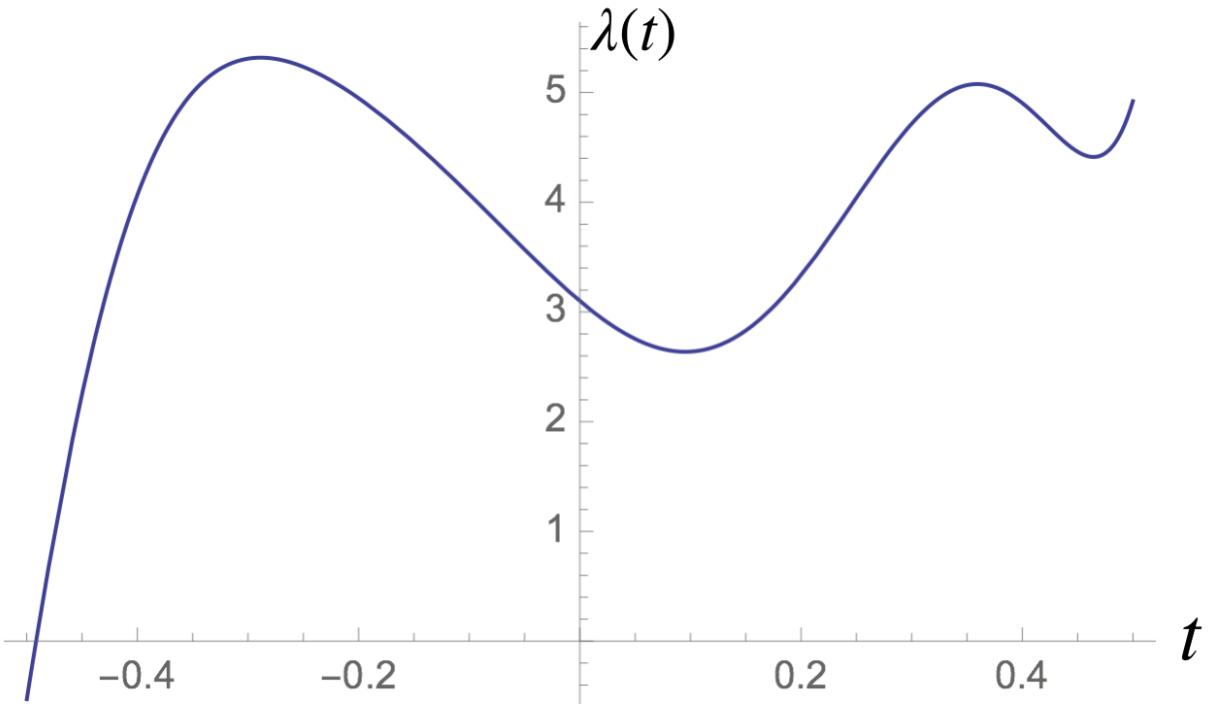


Figure 2: Plot of the function from Calzarossa Serrazi's paper that mimics job-interarrival time. Here time has been normalised by the 12 hours of the day and the origin has been translated to be at the midday which explains why time goes negative. The two drop in the plot mimic overnight inactivity and a drop around lunchtime.

levels of priority and there are certain conditions required to be in each queue. Finally, we have backfilling. Here we opportunistically run low priority jobs when insufficient resources are available for high priority jobs. That's our list of job scheduling algorithms which sort the order of the job waiting queue. The scheduling algorithm runs on job events such as when a job starts, finishes, aborts or arrives and if there are no events in the past 10 seconds it runs anyway.

Table 1: Table of algorithms

List of Algorithms	
Job Scheduling	Task Mapping
FCFS w/ best fit	Random
Shortest Job first	Round-robin
Score based priority	Dual-End
Backfilling	Cartesian Split

Task Mapping Algorithms

The task mapping algorithms determine which compute nodes to run the job on. The goal is to minimise the communication overhead and reduce cross-job interference. Random mapping is considered the worst case scenario. Round-robin keeps the nodes in an ordered list and when a job ends the node is appended to the list. Over time the fragmentation of the list becomes significant and the communication overhead drastically increases. For Dual-End we set a threshold value for time which groups every job into short or long. Short jobs search for unoccupied nodes from one end of the list, long jobs search the other end. Cartesian Splitting is a type of topological ordering algorithm which is much more complicated than the previous three and takes into account the network design of ordering the nodes.

Backfilling Algorithm

The backfilling algorithm works like so: Each job has a priority value $P(J)$ with the wait queue being ordered in terms of highest priority. The job of the highest priority $P(J1)$ gets the resource reservation but this is cancelled when a job with higher priority arrives that exceeds that of $P(J1)$ if it is still in the queue and hasn't been submitted yet.
The backfilling algorithm calculates the priority of each job J by calculating a score for 5 heuristics and summing them up. They are:

- Minimal requirements
- Ageing
- Deadline
- Licenses
- Wait minimisation

Minimal Requirements

Minimal requirement specifies details like the number of nodes and the number of software licenses on a machine.

Ageing

This heuristic is introduced to avoid job starvation. It works as in the below formula where age_factor is a multiplicative constant of choice.

$$P(J) += age\_factor \times age(J)$$

$$age(J) = wall\_clock - submit\_time(J)$$

Deadline

The goal is to maximise te number of jobs terminated by their own specified deadlines. A threshold value is set for the difference between the deadline the time it takes to complete the job. When the difference goes below this, the heuristic is calculated like so.

Licenses

License heuristics gives a higher score to jobs requiring critical resources on nodes such as NVRAM which hasn't been implemented yet.

Wait Minimisation

Minimises the wait time for jobs with the shortest execution times by a boost value, which is the backfilling component of this algorithm. In the below formula $ex\_exec(T)$ is each job's specific execution time and min_ex_t is the

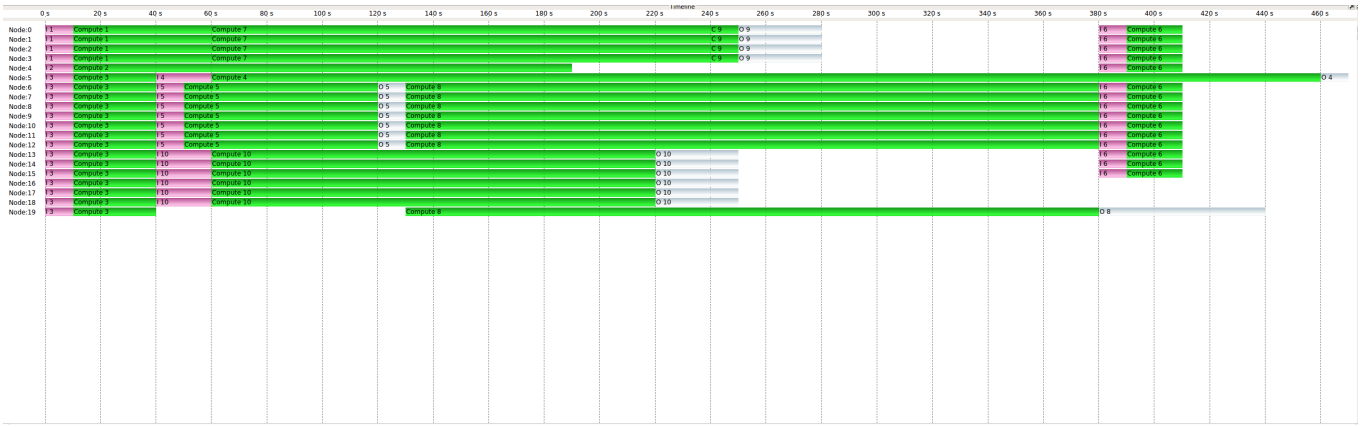


Figure 3: Visualisation of the simulator on Vampir. You can see jobs running in green and vacant nodes in white. The nodes are a function of time in this plot.

minimum of all the jobs in the scheduling queue. The user sets the value of $priority\_boost\_value$.

$$P(J) += priority\_boost\_value \times \frac{min\_ex\_t}{ex\_exec(J)}$$

Conclusions & Future Work

We covered a study of some job scheduling and task mapping algorithms and had a comprehensive look at the backfilling algorithm. Some of them have been implemented into the simulator, however due to time restraints there has been no experimentation or testing of their performances. This would be quite a significant task as it is envisaged lots of different parameters in the backfilling algorithm would have to be tweaked depending on the workload model used.

Some possible routes of future work are to:

- Create a visualisation for the simulator.
- Create experiments to test and analyse the performance of the algorithms.
- Incorporate using NVRAM (Non-volatile random access memory) technology into the algorithm. NVRAM is a new technology that retains its information when the power is switched off, this will reduce time spent writing and reading on the nodes. The algorithm would have to be adapted through such that jobs dependent on previous jobs' results are allocated the same nodes in order to make use of NVRAM. This would be implemented thorugh the use of a license heuristic where we would just make a license heuristic for jobs dependent on previous jobs and so that they could use the same nodes, perhaps the original job could would have a less restrictive license but once it has been assigned a node then subsequent jobs that are dependent on it would have to be assigned the license of that node. (this is just how I would imagine it, it has not been implemented).

References

[1] Dimitriadou, Karatza (2010). Job Scheduling in a Distributed System Using Backfilling with Inaccurate Runtime Computations

[2] A.D.Techiouba,L.Ricci (2008) Backfilling Strategies for scheduling streams of jobs on computational farms

[3] M.Abu Obaida, J.Liu (2017) Simulation of HPC job scheduling and large-scale parallel workloads

[4] H.Rajaei,M. Dadfar (2006) Comparison of backfilling algorithms for job scheduling in distributed memory parallel system

PRACE SoHPCProject Title

Scheduling on Novel and Advanced Hardware

PRACE SoHPCSite

EPCC, Scotland

PRACE SoHPCAuthors

Conor O'Mara, [Trinity College Dublin] Ireland

PRACE SoHPCMentor

Dr.Nick Johnson, EPCC, Scotland

Conor O'Mara

PRACE SoHPCContact

Conor, O'Mara, Trinity College Dublin
E-mail: omarac@tcd.ie

PRACE SoHPCSoftware applied

NEXTGENIO, Paraver, OTF2

PRACE SoHPCMore Information

http://www.nextgenio.eu/

PRACE SoHPCAcknowledgement

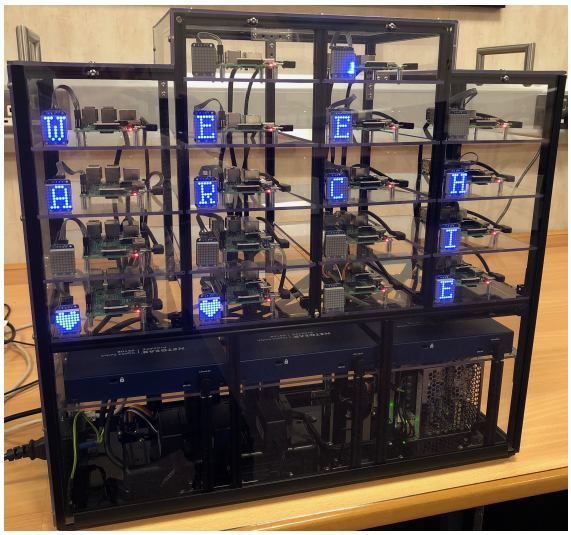
I'd like to thank my supervisor Dr. Nick Johnson for all the guidance through the two months, EPCC and the NEXTGENIO group for the use of their facilities and their friendliness. PRACE for giving me a place on the Summer of HPC programme. Many thanks to Leon Kos, Ben Morse and Stelios Erotokritou for all their work with the training week, editing blog posts, emails and the weekly web calls. Goodbye and thanks for reading.

PRACE SoHPCProject ID

1812

Visualising Computations on a mini supercomputer

Eva Havelková



Wee Archie, a suitcase-sized Raspberry Pi based cluster.

In this project new demonstrations for a mini supercomputer, Wee Archie, were developed to show explicitly how parallel programs run. These animations use LED lights connected to each node of the cluster and provide a real-time visualisation of parallel computations and communication within the cluster.

The rapid development of information technology in the past decades has brought about a great hunger for experts in the field of supercomputing not only in academia but also among commercial companies. To attract new talent to this field we need to raise awareness about supercomputers among students early in their childhood to ensure this progress continues in the future.

To bring the world of supercomputers closer to students and the public, Edinburgh Parallel Computing Center (EPCC) developed a small suitcase-sized supercomputer. It is built out of 18 Raspberry Pi nodes and it was named Wee Archie after the UK national supercomputer Archer. Wee Archie is taken to schools and science festivals to demonstrate what a supercomputer looks like and it has become extremely popular especially among kids of all ages.

A variety of programs have been developed to be run on Wee Archie, however, they do not specifically demonstrate how parallel programs work. Therefore, in this project we focused on developing new programs that employ the LED lights connected to each Raspberry Pi node on Wee Archie to provide real-time visualisation of parallel computations and communication within the cluster.

Entertain and educate

Having kids as the target audience for education is a challenging task. The goal is not only to teach them something in an understandable way but more importantly to spark their interest. Therefore, the first task of this project was to develop very simple parallel applications that would be interesting enough to draw kids attention.

As a starting point, we decided to use a database of tweets and implement a parallel search in the database. This embarrassingly parallel application can demonstrate usefulness of parallel computing while staying attractive for kids as social media in general is a very popular topic these days.

We used the Python programming language for the implementation. This was to ensure that the code will be easily readable for novice programmers, for example kids who already have some knowledge of coding.

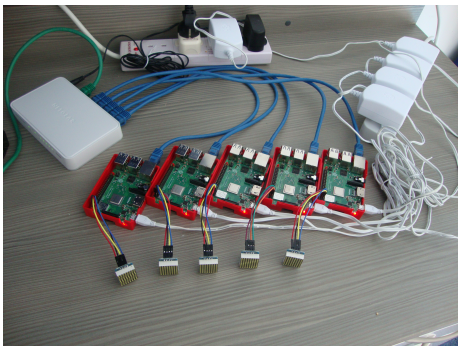
Build up a supercomputer

For development purposes one does not really need Wee Archie. Smaller Raspberry Pi based cluster can do the job and even kids can build their own. What is needed?

- 5 Raspberry Pis with power cables
- 5 micro SD cards with adaptors
- 5 Adafruit Mini 8x8 LED Matrix Backpacks
- Ethernet switch
- 6 Ethernet cables

There are many tutorials on the Internet providing instructions on how to build and set up the cluster, for example the EPCC ones.

In a nutshell, the set up includes installing the Raspbian operating system on each Raspberry Pi, setting up network infrastructure, creating a shared directory for parallel programs and installation of any additional required software on all Raspberry Pis.



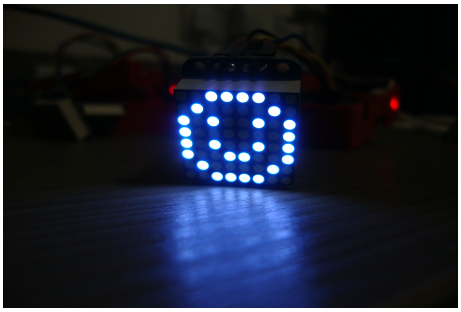
Small supercomputer with LED lights build out of 5 Raspberry Pi nodes.

Programming LED lights

The two main pieces of software needed for programming the LED lights are:

- Adafruit Python LED backpack library. It is a freely available Python library for controlling LED backpack displays on Raspberry Pis and other similar devices.
- Python PIL (or PILLOW) library, more specifically Image and ImageDraw modules.

One possibility is to program the LEDs directly with Adafruit library by setting each pixel of on the LED matrix to on or off. This might come in handy sometimes but an even easier way of doing it is to also make use of the Python PIL library. It provides functions for creating and drawing 8x8 1-bit images that can be then printed on the LEDs using the Adafruit library.



The following picture gives an example of code employing the aforementioned libraries to print a smile on a LED matrix backpack.

```
from PIL import Image, ImageDraw
from Adafruit_LED_Backpack import Matrix8x8

# Create an 8x8 pixels image with PIL
image=Image.new('1', (8,8))
draw=ImageDraw.Draw(image)
draw.line([(0,2),(0,5),(2,7),(5,7),(7,5)],fill=1)
draw.line([(7,5),(7,2),(5,0),(2,0),(0,2)],fill=1)
draw.point([(2,2),(5,2)],fill=1)
draw.line((2,4,3,5),fill=1)
draw.line((5,4,4,5),fill=1)

# Print the image to the LED with Adafruit
display=Matrix8x8.Matrix8x8()
display.begin()
display.set_image(image)
display.write_display()
```

Challenges

To make visualisations look appealing we not only want to print images but also to move them across the LEDs, for example to scroll them over a LED matrix. This can be done by creating a sequence of images that, if displayed in order one after another on the LED matrix, appear to scroll over. The Adafruit library comes with functions that create such sequence out of a given image.

Scrolling over multiple LEDs is a lot more challenging task. To animate for example point to point communication between two nodes we would like an arrow to move smoothly from one LED matrix to another. This requires synchronisations of more nodes of the cluster which means we need to also use MPI. A straightforward solution of scrolling might be sending the images that should be printed from one node to another. This however, might not result in a smooth scroll due to the latency in communication.

To achieve smooth scrolling, it is beneficial to make use of MPI communicators. Simply, we create a communicator from all nodes that should participate in the scrolling and to avoid communication we give all of them all the images that should be printed to animate scrolling. Additionally, we need to set a corresponding offset according to the order of the nodes. Using MPI Barrier function for synchronisation within the communicator will then guarantee smooth scrolling across those nodes without affecting the rest of the cluster.

Parallel search

The first application we developed was a parallel search in a database of tweets. At the beginning. the program asks the user to type in any string (a word) they want to look for in the database. Then it searches the database for all tweets containing this given string and as a result it prints some statistics about the tweets that contained the string. For example how many of them were marked to be happy tweets.

This program employs a standard master-worker model where one of the nodes acts as a master that coordinates the computation and the rest of nodes act as simple workers (slaves). The program is flow as follows:

- Master reads a string from input
- Master broadcasts the string to all worker nodes
- Each worker searches its part of the database
- Master gathers the results back and prints them to output

Two important parts of parallel computing are demonstrated with this program. These are the distribution of work among a number of working nodes and collective communication. Namely a broadcast function (master node sends identical message to all workers) and a gather function (master node collects messages from all workers).

A couple of animations were used to visualise both communication within the cluster and the search itself. For example, to visualise that a message is being sent an animation of a closing

envelope is displayed followed by arrow pointing where the message is being sent. Analogically for receiving a message. You can find a video of the program here.

Parallel search on Wee Archie. Worker nodes searching, top left master node is idle.

Outcomes

Besides the above explained Parallel Search we have also developed a little bit more advanced application that is called Bingo game. It implements a simple alphabetical bingo game on the nodes of Wee Archie and aims to visualise (among other things) point to point communication. For more details, have a look at this video presentation.

It was already mentioned that the first part of outreach is to catch peoples eye. So for science festival we also have some fun animations such as scrolling a welcome message over the LEDs of Wee Archie or a really nice-to-watch animation of a rainfall that can be used as a screen-saver.

Further more, all the animation functions for the LED lights have been wrapped up as a class of functions. These can be used in any Python program and not only contain a variety of demos for visualisation of parallel work and communication but also other additional functions that may be useful for further development.

Conclusion

The results of this project are outstanding, many expectations have been exceeded. Compared to what has been done so far the outcome is very innovative. Until now, none of the simulations developed for Wee Archie made use of the LED matrices for real-time visualisation of computations. Also, the existing simulations were usually way too complex to allow novice programmers to understand them. Now, we have simple yet attractive parallel programs that fully utilise the LEDs to visualise main building blocks of parallel computing.

For further development we provide a set of functions (demos) for controlling the LED matrix backpacks that is very straightforward to add new functions to. Some tips for follow up projects would be, for example, focusing on more advanced parts of parallel programming such as all to all communication patterns or reduction functions.

PRACE SoHPCProject Title

Parallel Computing Demonstrations on Wee Archie

PRACE SoHPCSite

Edinburgh Parallel Computing Centre, United Kingdom

Eva Havelková

PRACE SoHPCAuthors

Eva Havelková, [MFF UK, Prague] Czech Republic

PRACE SoHPCMentor

Dr. David Henty, EPCC

PRACE SoHPCContact

Dr. Leon Kos, PRACE
E-mail: leon.kos@lecad.fs.uni-lj.si

PRACE SoHPCSoftware applied

Python, MPI, Linux, Adafruit, Raspbian

PRACE SoHPCMore Information

Adafruit Python OpenMPI

PRACE SoHPCAcknowledgement

I would like to thank everyone at the Edinburgh Parallel Computing centre who helped me with this project. Namely, my mentor Dr. David Henty and co-mentor Dr. Oliver Brown for their willingness to answer all my questions. I would also like to thank Dr. Gordon Gibb and Dr. Alistair Grant for their help with both software and hardware issues I had with Wee Archie. Lastly, I would like to thank Dr. Mario Antonioletti for keeping the spirits of our office up throughout the whole summer.

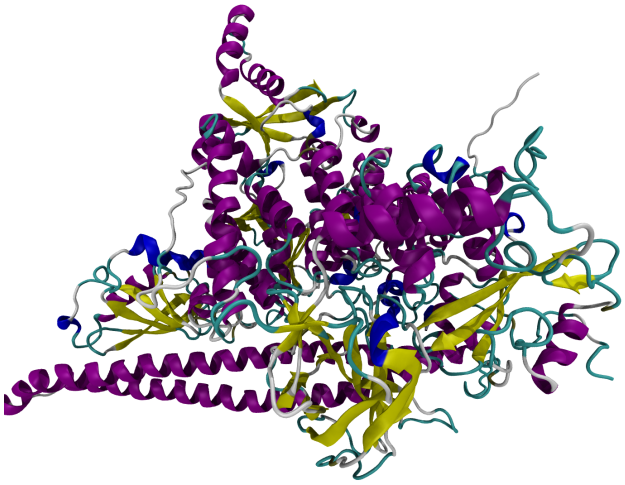
PRACE SoHPCProject ID

1813

Simulating the effects of an oncogenic mutation

Pedro Santos

PI3Kα is the most commonly mutated kinase in human cancers. The PI3Kα E545K mutation causes a spontaneous separation of PI3Kα subunits that causes its overactivation. However, the mechanism of this process is not known. In this project, metadynamics simulations and Principal Component Analysis have been employed using HPC resources in order to gain insights on the mechanism of overactivation, needed for the efficient design of new anti-cancer drugs.



Of the kinase mutations that lead to cancer, those occuring in the enzyme phosphatidylinositol 3-kinase α, PI3Kα, are the most common. This enzyme catalyses the phosphorylation of phosphatidylinositol 4,5-bisphosphate (PIP2) to phosphatidylinositol 3,4,5-trisphosphate (PIP3), which is essential for cell growth and survival.[1] Figure 1 a) shows the molecular structure and domains of PI3Kα. It is known that the wild-type protein becomes active after binding to a phosphopeptide, which causes a detachment of the nSH2 domain from the helical and kinase domains (see Figure 1 a)).

One of the most common mutations in PI3Kα is E545K (Figure 1 a)), which is known to cause overactivation of the protein. In this mutation the negatively charged glutamic acid E545 is replaced by a positively charged lysine (K545). Because E545 is in the vicinity of another positively charged lysine on the nSH2 domain (K379), the two positively charged residues K545 (helical) and K379 (nSH2) repel each other resulting in a detachment of the helical and nSH2 domains and to overactivation, similar to the effect of the phosphopeptide mentioned above. However, the exact mechanism behind this process at the molecular level is not clear, and this knowledge is of paramount importance for designing anti-cancer drugs that can target specific intermediate structures observed along the overactivation process.

Molecular dynamics (MD) simulations is a powerful tool that provides a systematic approach to investigate the dynamics of proteins in atomic-level detail. Previous simulations of PI3Kα bearing the E545K mutation have shown a spontaneous detachment of the nSH2 domain from the helical domain.[2] However, this detachment process was only observed in one instance. This is because large scale protein motions, such as the one under study here, occur on a timescale unattainable by standard MD.

In order to access the timescale of the PI3Kα detachment, specific analyses and simulation techniques have to be employed. In this work, the main objective was to gain insights on the mechanism of overactivation of PI3Kα bearing the E545K mutation, through the use of principal component analysis (PCA) and

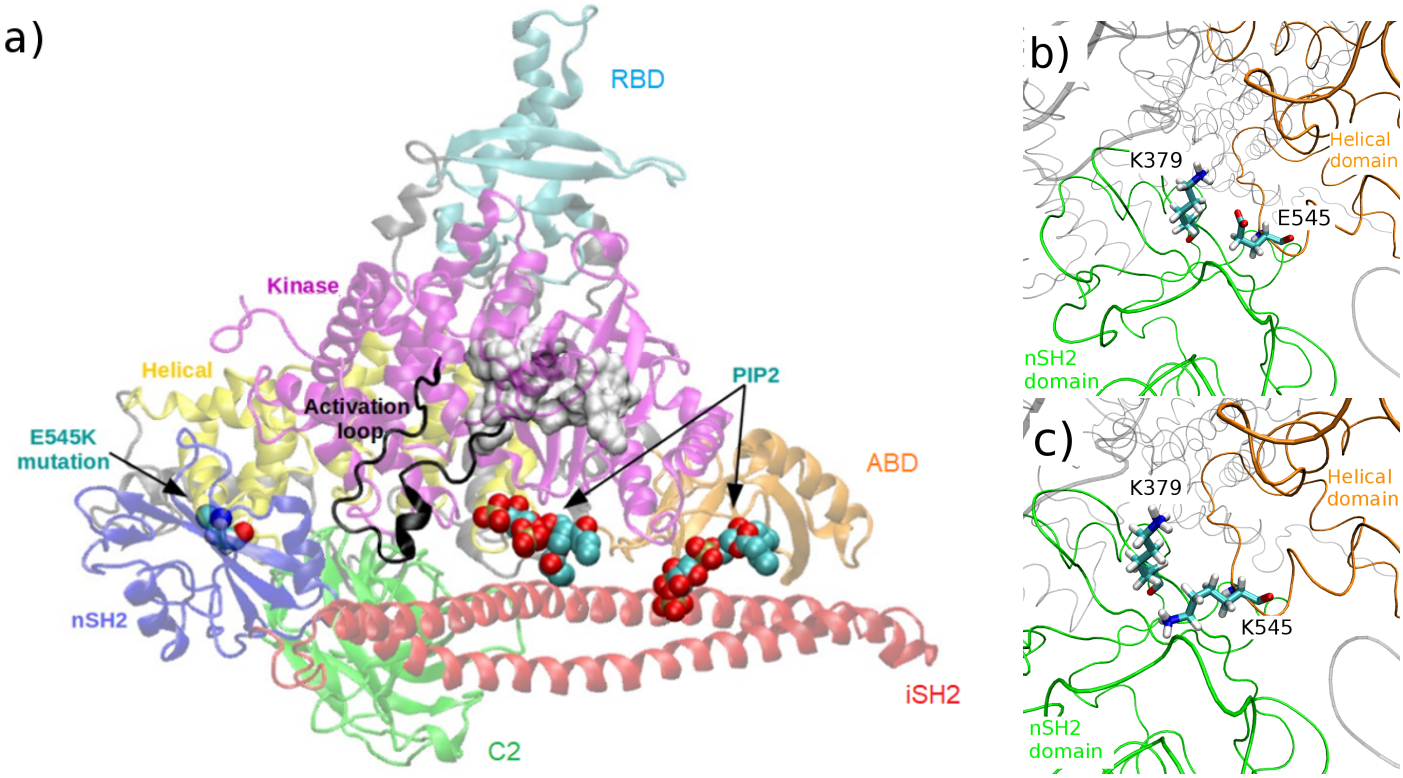


Figure 1: a) Molecular structure of PI3Kα, showing the main domains, the site where PIP2 binds for the reaction and the area where the E545K mutation is located. b) and c) show a zoom in the aminoacid 545 for the wild-type and mutant PI3Kα, respectively.

metadynamics simulations, two techniques that allow a description of large-scale protein motions, and at the same time require HPC resources.

Methods

PCA is a widely used statistical tool that allows the reduction of a large multivariate data set to only a few variables describing most of the variation, called principal components (PCs). In the context of MD, PCA can be used to simplify the description of large motions observed in the trajectory.[3] In this work, the trajectory of the MD simulation of PI3Kα with E545K mutation was analyzed by PCA, using the tools available in the software GROMACS 2016.4.
Metadynamics[4] is one of the so-called enhanced sampling techniques, whose aim is to accelerate the observation of rare events in regular MD. The basic principle is to fill the energy landscape that is being explored in MD with Gaussian functions, allowing access to different energy minima, that are separated by large energy barriers, as the simulation progresses. Two collective variables were used to describe the detachment of the nSH2 and helical domains: the distance between centers of mass of the helical and nSH2 domains (CV1) and the distance from the contact map of the reference open state of the protein (CV2). In this project, convergence analysis were performed for previously obtained multiple walkers metadynamics simulations of the wild-type and mutant PI3Kα (8 walkers) as well as the generation of a parallel tempering metadynamics simulation of the wild-type protein, using GRNET HPC resources provided by the Summer of HPC programme.

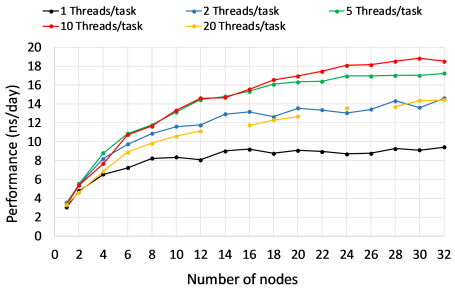


Figure 2: Scaling plot for the parallel tempering metadynamics simulation of mutated PI3Kα on ARIS (GRNET).

In addition, a parallel tempering metadynamics simulation of the E545K mutant protein was run for 44 ns with the software GROMACS 2016.4 using the PLUMED plugin, on the ARIS supercomputer. 20 replicas were used, with temperatures between 300 K and 319 K and exchanging configurations every 2000 steps. In order to determine the simulation setup that gives the best use of the available hardware, scaling tests were performed with a single replica by varying the number of nodes used and also the number of threads per MPI task. The performance results are represented in Figure 2. For a more efficient use of computational resources, 4 nodes per replica were employed, with 5 threads per task.

Results and discussion

Principal Component Analysis

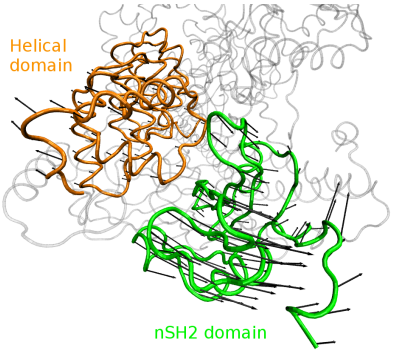


Figure 3: Vectors of the first principal component (PC1) of the trajectory of PI3Kα bearing the E545K mutation.

PCs are obtained from an eigenvalue decomposition of the covariance matrix

of our data. As such they are associated with an eigenvalue, which is related to the percentage of variance explained by the PC. From a plot of those eigenvalues we were able to see that 10 PCs are needed to describe at least 85% of the data, but if we apply the popular criteria of looking at the link in the eigenvalue plot we conclude that PC1 should already be appropriate. In fact, if we draw the vectors that constitute PC1 in space, we find that they represent a detachment of the nSH2 and helical domains, with the former having the greatest displacement (Figure 3). Therefore, we can conclude that PC1 is a good variable to represent the detachment process, and could in fact be used as a collective variable in the metadynamics simulation.

Metadynamics

In order to determine if a metadynamics simulation is converged and can be safely analysed, three criteria must be fulfiled:

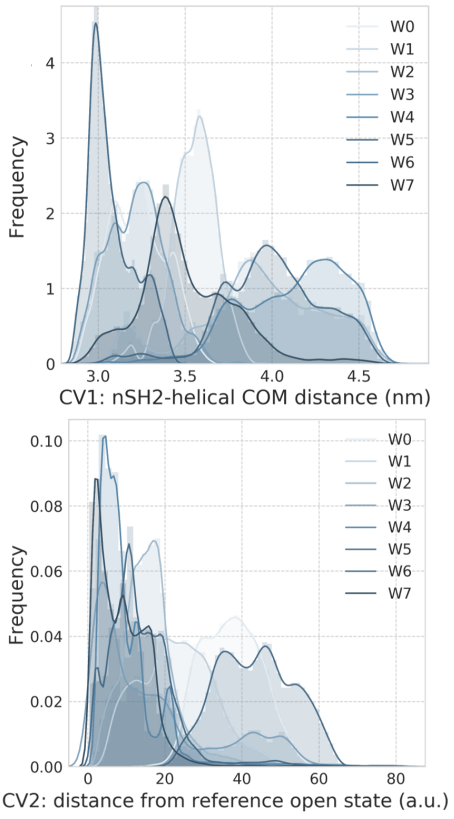


Figure 4: Distribution of CV space explored by each of the 8 walkers.

1) the entire space of each CV must be fully explored during the simulation time; 2) the height of the deposited Gaussian functions must tend to zero and 3) the free energy profile must remain consistent. In Figures 4 and 5, a representative example for the multiple walkers metadynamics simulation of the mutated PI3Kα is shown. Figure 4 shows that all CV space is fully accessed by all walkers during the metadynamics simulation.

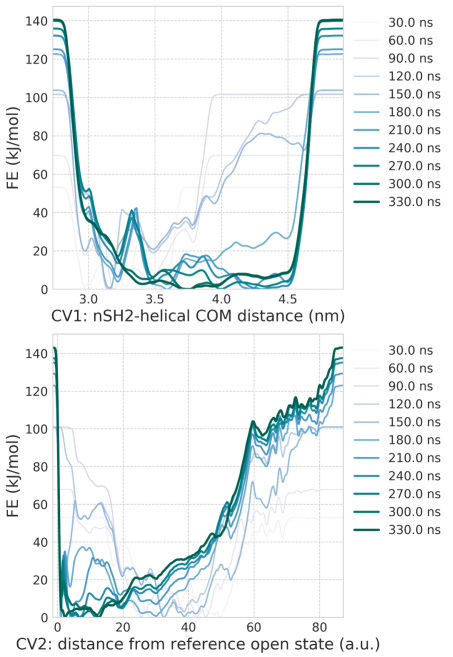


Figure 5: Free energy profile as a function of each CV and every 30 ns of simulation.

By plotting the heights of the Gaussian functions we observed that nearly all walkers have Gaussian heights close to zero at the end of the simulation. However, one of the walkers seems to present an exception, and as such the simulation should be extended to confirm that the height of all deposited Gaussians is nearly zero. Finally, Figure 5 suggests that the free energy profiles on both CV spaces remain consistent after 300 ns of simulation, another indication of convergence. A similar scenario was verified for the multiple walkers metadynamics simulation of the wild-type protein. However, on the parallel-tempering simulation of the wild-type PI3Kα the heights of the Gaussian functions did not tend towards 0, and so this simulation is not converged and should be extended for more time before further analysis.

Future directions

The PCA analysis performed in this work can serve as a platform for further metadynamics simulations, such that the identified PC1 can be used as a collective variable. Further PCA studies should be conducted for the wild-type PI3Kα as well, in order to establish the qualitative difference in dynamics between the wild-type and mutant proteins. Convergence analysis and energy analysis should also be performed for the parallel-tempering simulation run in ARIS for the mutant PI3Kα, so that meaningful conclusions can be drawn about the free energy surface of this system and how it links to its dynamics.

References

[1] Lee, J. Y., Engelman, J. A., and Cantley, L. C., PI3K Charges Ahead. Science, 2007, 317(5835), p. 206-207.

[2] Leontiadou, H., Galdadas, I., Athanasiou C., and Cournia, Z., Insights into the mechanism of the PIK3CA E545K activating mutation using MD simulations, Sci. Rep., 2018, in press

[3] Mueller Stein, S., Loccisano, A., Firestine, S., and Evanseck, J., Principal components analysis: A review of its application on molecular dynamics data, Annual Reports in Computational Chemistry, 2006, 2, p. 233-266

[4] Barducci, A., Bonomi, M. and Parrinello, M., Metadynamics, 2011, 1, p. 826-843

PRACE SoHPCProject Title

Investigating the effect of the oncogenic mutation E545K of the PI3Kα protein with enhanced sampling MD simulations

PRACE SoHPCSite

Biomedical Research Foundation of the Academy of Athens (BRFAA) and Greek Research and Technology Network (GRNET), Greece

Pedro Santos

PRACE SoHPCAuthors

Pedro Santos, University of Coimbra, Portugal

PRACE SoHPCMentor

Zoe Cournia, BFRAA, Greece

PRACE SoHPCContact

Pedro, Santos, Department of Chemical Engineering, University of Coimbra (DEQ-UC), Portugal
Phone: +351 916 132 094
E-mail: pmsantos@eq.uc.pt

PRACE SoHPCSoftware applied

GROMACS, PLUMED, VMD

PRACE SoHPCMore Information

www.gromacs.org
www.plumed.org
www.ks.uiuc.edu/Research/vmd/

PRACE SoHPCAcknowledgement

My thanks to Dr. Zoe Cournia and the Cournia Lab members, at BRFAA, for recieving me, to Ioannis Galdadas for supplying metadynamics trajectories for convergence analysis and for his support, and to Dr. Dimitris Dellis at GRNET for his support and providing access to the ARIS supercomputer.

PRACE SoHPCProject ID

1805

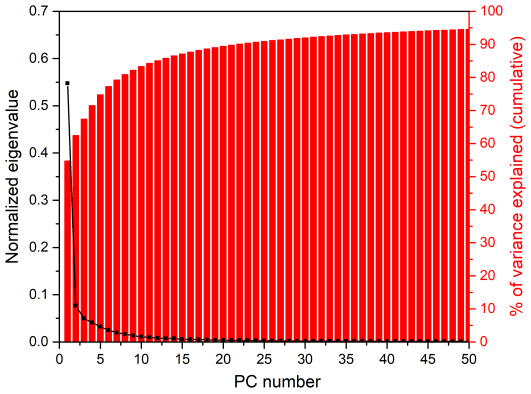


Figure 6: Scree plot of the eigenvalues for the first 50 principal components of the mutant PI3Kα simulation (black line) and respective cumulative percentage of variance explained (red bars).

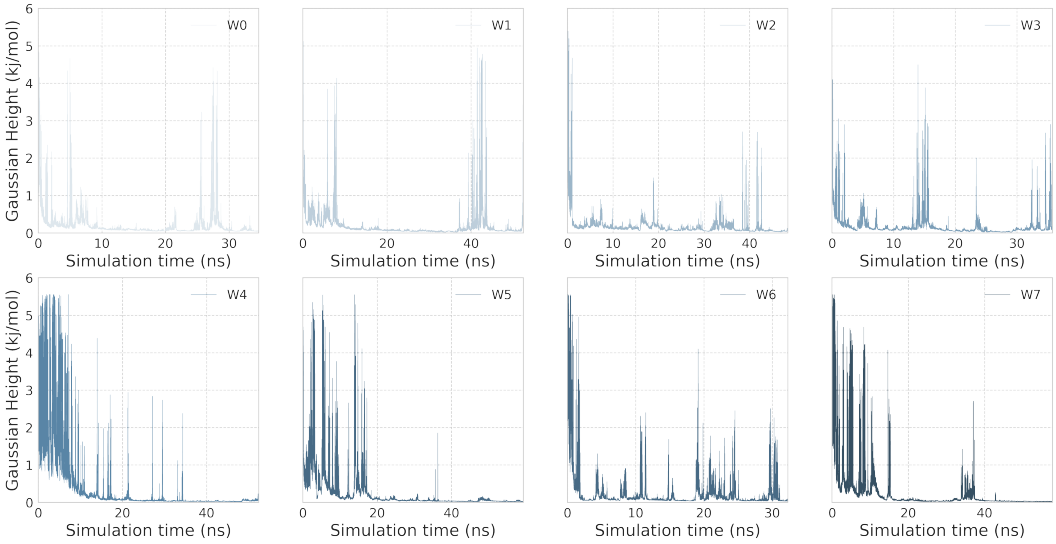


Figure 7: Evolution of the height of Gaussian functions with simulation time for all 8 walkers of the multiple walkers metadynamics simulation of PI3Kα with the E545K mutation.

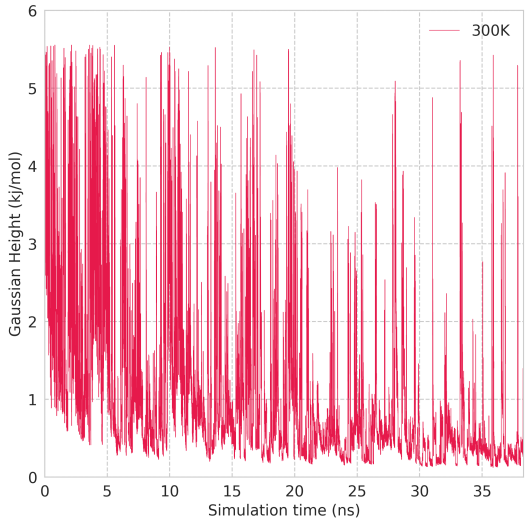
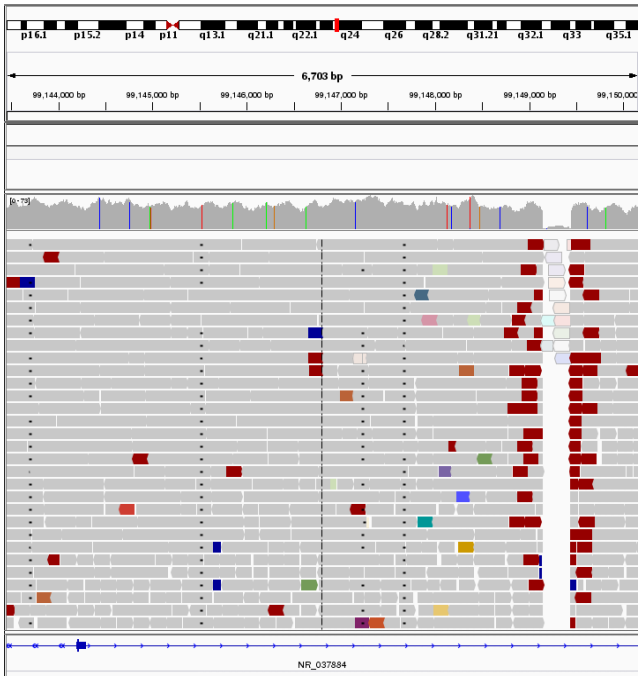


Figure 8: Evolution of the height of Gaussian functions with simulation time for the parallel tempering metadynamics simulation of wild-type PI3Kα.

ABySS Sequence Assembler

Vladimir Nikolić



ABySS is a resource efficient software for assembling DNA sequences from a large number of short fragments. The project consists of multiple stages and although the main assembly part of the project is efficient and well distributed, some stages are not performing optimally and could be optimised, which was my task!

A sequence assembler in bioinformatics takes thousands of small fragments of DNA, with, say, a hundred or two nucleobases length, and determines where to place them on the whole genome. This is a very demanding task, both algorithmically and computationally, as we can be dealing with many gigabytes of DNA information. The ABySS assembler that I was working on builds a graph of how these fragments are related and slowly tries to connect them into a contiguous sequence. As this graph is large and the process takes time, it is split up into computational nodes, each dealing with its own part of the DNA. However, as this is a multistage process, not all of it is distributed across nodes, but only the core. So the problem we are facing here is parallelising the rest of it, as each stage takes considerable amount of time.

Optimisation approaches

Here, I will describe how I have parallelised Sealer, one of the final stages of ABySS designed to fill in the missing DNA data in the final sequence.

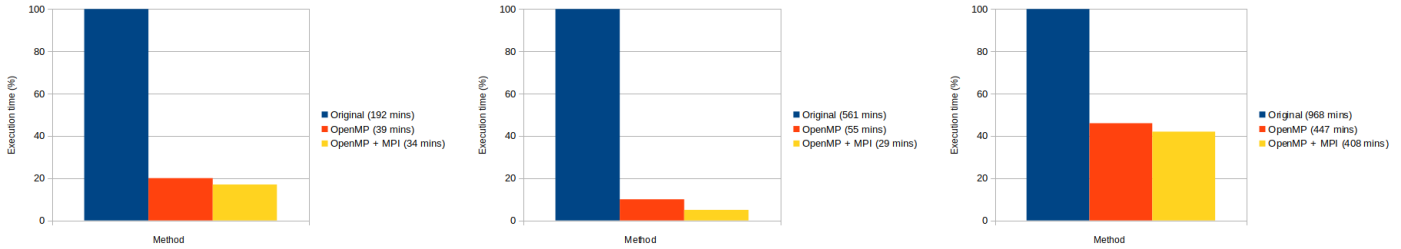
Sealer

As output of ABySS, we do not get a single contiguous DNA sequence, but instead, multiple sequences with some data missing in them, which are called scaffolds. Sealer's job is to use whatever unused DNA data we have from the input and try to extend inwards the ends of the gaps with missing data to get a better sequence. The problem is that Sealer is single threaded, and is a bottleneck to the whole pipeline. When trying to close the gaps, Sealer tries several so called k values. A k value determines the length of a kmer, which is the length of a sample that we use when dealing with DNA sequences. So the overall process consists of trying to close the gaps with one k value, then proceeding to close the leftovers with the next one, and so on.

Figure 1: Scaffold - a DNA sequence with missing information.

OpenMP

Gaps are independent one from another, and can be sealed in parallel. This is where OpenMP comes in - it automatically spawns threads in C++ code for a given loop, and distributes the iterations as equally as possible between the threads. Tests have shown that the multithreaded version of gap sealing performs up to 10 times faster on a 24 core machine, however, the main factor that determines the speedup is the longest gap that needs to be closed, as that is the bottleneck here.



1. Original: 192 mins, OpenMP: 39 mins, OpenMP + MPI: 34 mins
2. Original: 561 mins, OpenMP: 56 mins, OpenMP + MPI: 29 mins
3. Original: 968 mins, OpenMP: 447 mins, OpenMP + MPI: 408 mins

Figure 2: What we are most interested in is the difference in execution times. The charts on the figure show the comparison between the original Sealer, the OpenMP (with 24 cores), and MPI (with 8 nodes) version respectively, for three different datasets, each increasing in size, with the final one being the full fish genome.

MPI

This change distributes the workload of Sealer to multiple computing nodes. In addition to giving each node a pool of gaps to close from the total, the work is also split based on the k values - every node tries to get a different one. This gives a noticeable speed up, as every k values requires a particular data structure to be built for it, the so called Bloom filter, so instead of having one node build this for every value of k, different nodes each build for the k value they are working with.

It should be noted that MPI doesn't necessarily give much better results than the OpenMP version, as particularly long gaps determine the minimum amount of time the job will take, below which we cannot go, no matter how much we parallelise.

Other attempts

In addition to improving Sealer, there were two other unsuccessful optimisation attempts.

File reader

Because we are dealing with files up to hundreds of gigabytes in size, improving the speed of the code that reads the files seems rational. With that in mind, I integrated Heng Li's (github.com/lh3/readfq) kseq library, written in C, designed to be one of the fastest libraries for reading DNA files. Benchmarks have shown that it is twice as fast as the ABySS counterpart. However, after running the whole ABySS pipeline, the difference in execution time was only up to 10% and after discussing the results, we concluded it was not worth the additional code complexity that it introduced.

Bloom filter

This is the aforementioned required data structure, and it is a probablistic set that can tell you whether your data is in it but can give false positives. The purpose of a Bloom filter is to trade correctness (because of the false positives) for memory usage, as it requires less memory than a tradional set. When working with kmers, they are often placed into this filter instead of keeping them all in memory. Building a bloom filter is multithreaded, but it requires locking parts of it to ensure correctness. The lock contention slows things down a lot, so I implemented a lockless version, where every thread gets its own copy of the filter, builds it independently and then merges with the others in the end. This however, only provided a speed boost if the filter was not too large, as merging large filters took too long, so the change was not used.

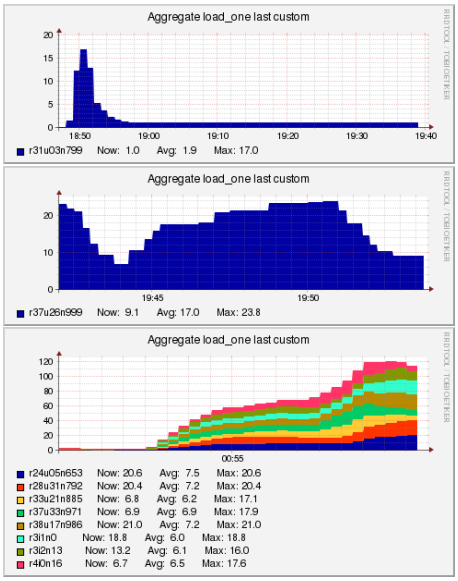


Figure 3: The graphs show the difference in how much a computing node is loaded when running Sealer. The original Sealer is single threaded (with the exception of a spike at the start, which builds a required data structure, and is multithreaded) and the node is heavily underused. The graph after it shows the OpenMP version, in which the node reaches peak usage, as all of its cores are running at one point. On the last OpenMP + MPI graph, every node gets a different color and after building a required data structure, they are all fully used as they seal the gaps.

Discussion & Conclusion

Other than the speed ups I have implemented, we have uncovered a lot of potential improvements for future work. For example, a small subset of the gaps being sealed are bottlenecks as they take most of time. This is very visible on Figure 2 on the third chart - even though Sealer is parallelised, it does not divide the execution time by the number of threads and nodes it is running on. A suggested solution here is to limit the number of nodes in the graph that are being traversed to determine the gap sequence.

Another thing we have noticed, is that there is a discrepancy between the estimated length of the gaps being sealed, and the length of actual sequences Sealer fills them with. Figure 4 shows the difference in the distances, where ideally, the charts should overlap. This means the sequences that are used for gaps are not as accurate as they should be.

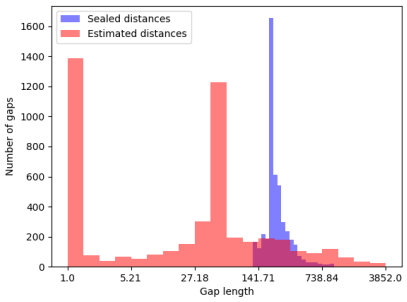


Figure 4: Histogram of estimated (red) and sealed (blue) gap distances.

Results

The outcome of the summer project is:

- Two pull requests accepted into the original project:
  - **Sealer OpenMP parallelization.**
    MPI is being phased out from ABySS, so the MPI version of Sealer was a demonstration and not suitable for merging.

  (github.com/bcgsc/abyss/pull/249)
  - **Sealed gap sequence file output.**
    This change adds a command line option to Sealer to let it write sealed gap sequences to a file during the job, letting the user inspect what is happening before everything is done.

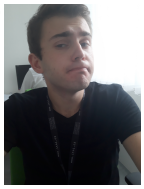
  (github.com/bcgsc/abyss/pull/247)
- **Insight into file reader performance.**
  It is not a bottleneck and further optimizations do not provide a significant improvement.
- **Insight into bloom filter performance.**
  There is some lock contention during its construction, however, on long jobs, it is not a major determining factor of the execution time.

PRACE SoHPCABySS Sequence Assembler

Improving existing genomic tools for HPC infrastructure

PRACE SoHPCSite

VŠB - Technical University of Ostrava, Czech Republic



PRACE SoHPCAuthors

Vladimir Nikolić, School of Electrical Engineering, Belgrade, Serbia

PRACE SoHPCMentor

Martin Mokrejš, VŠB - Technical University of Ostrava, Czech Republic

Vladimir Nikolićphoto

PRACE SoHPCContact

Vladimir Nikolić, School of Electrical Engineering, Belgrade
Phone: +381 61 294 2579
E-mail: nv130344d@student.etf.bg.ac.rs

PRACE SoHPCAcknowledgement

I would like to thank my mentor, Martin Mokrejš, for all the guidance on the project. I am also thankful to Ben Vandervalk and Shaun Jackman, the original creators of ABySS, for the feedback they have provided on my work. And finally, thanks to PRACE, Summer of HPC and IT4Innovations for the provided infrastructure and accomodation.

PRACE SoHPCProject ID

1817

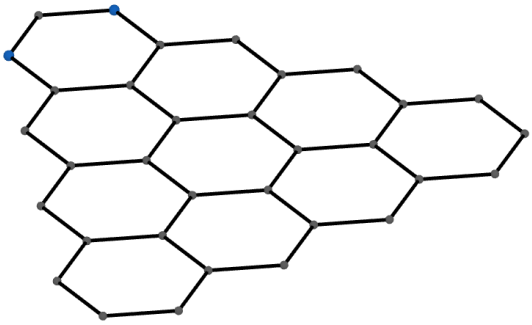
PRACE SoHPCReferences

Shaun D Jackman, Benjamin P Vandervalk, Hamid Mohamadi, Justin Chu, Sarah Yeo, S Austin Hammond, Golnaz Jahesh, Hamza Khan, Lauren Coombe, René L Warren, and Inanc Birol (2017). ABySS 2.0: Resource-efficient assembly of large genomes using a Bloom filter. Genome research, 27(5), 768-777. doi:10.1101/gr.214346.116

Adding Matlab functionality to a Hybrid Monte Carlo graphene model and implementing the Eig-CG solver.

Graphene Models in HPC

Janni Harju



Graphene, the the 2D wonder-material, can be modelled using modern HPC infrastructure. Functionality for analysis using Matlab was added, and the Eig-CG method was implemented.

Introduction

Ever since the first paper[1] on the electronic properties of graphene, a nanomaterial formed of layers of carbon atoms in a hexagonal lattice, its curious electronic properties have led to much excitement. Since methods for producing graphene on an industrial scale are still under development, computer modelling plays an important role in developing our understanding of this "wonder-material".

At the Jülich Supercomputing Center (JSC) in Germany, Smith & von Smekal,[2] amongst others, have developed Hybrid Monte Carlo (HMC) methods to calculate graphene's electronic properties using lattice-discretized quantum field theory, a model widely used to make particle physics predictions using HPC. The aim of this summer project was to provide functionality for exporting the fermion matrix to Matlab, a programming language widely used in applied mathematics. After this, the Eig-CG method, an improved version of the conjugate gradient (CG) method,[3] was implemented.

The electrons in a graphene lattice are quantum particles. This means that they can best be described by probabilistic models; the physical properties that we observe (conductance, tensile strength etc) are averages of the "fuzzy" behaviour of the particles involved. In principle, to calculate this average in order to describe graphene's electrical properties in time, we should therefore calculate every possible combination of movements of the electrons in a graphene lattice. Of course, this is not actually possible.

HMC simulations rely on generating only a small number of the most likely trajectories, and taking the average of these. These likely trajectories are constructed by generating random small "jumps" in time, corresponding to new states of the electrons at set time intervals. To ensure that we are constructing a likely trajectory, the probability of each jump is calculated, and the jump is accepted only with this probability.

To calculate the probability of a small transition in graphene, equations of type

$$MM^{\dagger}x = b \qquad (1)$$

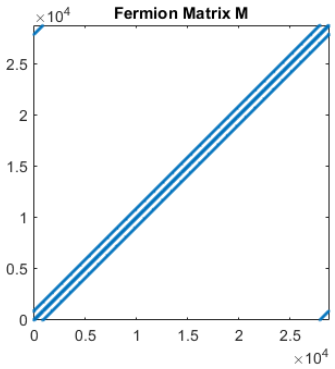
must be solved. Here $MM^{\dagger}$ is a large matrix, b is a known vector defined by the states we are considering, and x is an unknown vector we wish to find. For physical models, the matrix is very large, and supercomputers are usually needed to solve these problems repeatedly.

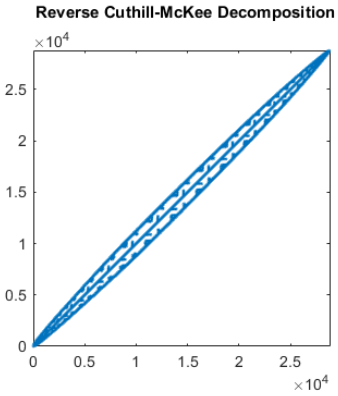
The CG method is a commonly used iterative scheme to estimate solutions x to Equation 1. Iterative schemes are numerical methods based on generating a guess x_0 for the solution. If the remainder $r = b - MM^{\dagger}x_0$ is large, this guess is improved (this counts as an iteration, and the improved guess is labelled x_1) and if it is smaller than a set value (often called the tolerance) the guess is accepted as an approximate solution.

For large scale numerical calculations like these, optimized, compiled languages such as C++/C, or FORTRAN are commonly used. They allow for fast calculations large matrices, which is a prerequisite for performing HMC simulations. However, data in such formats is not easily visualized or manipulated. Many interpreted lan-

$$M = \begin{pmatrix} S_\mp + A & D_1^+ & & & D_{N_t}^- \\ -D_1^- & S_\mp + A & D_2^+ & & \\ & -D_2^- & S_\mp + A & \ddots & \\ & & \ddots & \ddots & \\ -D_{N_t}^+ & & & -D_{N_t-1}^- & S_\mp + A \end{pmatrix}$$

$$S_\mp = \begin{pmatrix} -1 & 0 & 0 & \dots \\ 0 & 1 & 0 & \dots \\ 0 & 0 & -1 & \dots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}, \quad D_n^+ = \begin{pmatrix} e^{i\phi} - \mu & & & \\ & 0 & & \\ & & e^{i\phi} - \mu & \\ & & & \ddots \\ & & & & e^{i\phi} - \mu \\ & & & & & 0 \end{pmatrix},$$





Plots of the fermion matrix M (as defined on the left) and a decomposition using Matlab. The blue lines depict non-zero entries.

guages, such as Matlab or Python, on the other hand, often allow the programmer to perform such tasks in a single line of code.

Matlab Export

The HMC simulations conducted at the JSC are run on C++. To allow for easier and more intuitive exploration of the results, scripts were written to export the results into Matlab.

A function calculating the numerical fermion matrix for a given lattice set-up was written. Since the fermion matrix mostly consists of zeros, it was exported into Matlab using sparse matrix format. This means that a file with indices and values for non-zero values is written. Matlab features a compressed datatype sparse array which quickly loads and processes this kind of data.

Functions in Matlab for testing the integrity of the matrix, visualising non-zero elements and possible decompositions were written.

Eig-CG

Although the CG method is efficient for solving Equation 1, note that, in general, the expected number of iterations for each different right-hand side remains constant. The idea behind the Eig-CG method, as proposed by,[3] is to use the first m equations to improve later "starting point guesses". This adds overhead and hence increases the time needed to calculate the solution for the first m equations, but after this, the iteration count for later calculations can decrease dramatically.

After the algorithm was implemented, it was found that for graphene's fermion matrix the method did not appear to significantly reduce the iteration count. The reasons are currently unknown, and it is also possible that despite best efforts to find mistakes in the implementation, something might have escaped notice.

Conclusion

Over the course of this summer project, functions exporting data for HMC simulations for graphene into Matlab were written. The change of platform allows for much easier evaluation and exploration of the data.

Despite best efforts, the Eig-CG implementation did not improve performance. Checks will be done in the coming weeks to see if this is because of a mistake in the code, or because of some yet unknown qualities of the matrix under consideration.

References

[1] K. S. Novoselov et.al.. (2004). Electric Field Effect in Atomically Thin Carbon Films. *Science*, 306:666–669.

[2] D. Smith & L. von Smekal. (2014). Monte Carlo simulation of the tight-binding model of graphene with partially screened Coulomb interactions. *Phys. Rev. B*, 89:195429.

[3] A. Stathopoulos, K. Orginos. (2010). Computing and deflating eigenvalues while solving multiple right hand side linear systems in Quantum Chromodynamics. *SIAM J. Sci. Comput.*, 32:439–462.

PRACE SoHPCProject Title
GraPhine meets CudeCD

PRACE SoHPCSite
Jülich Supercomputing Center, Germany


Janni Harju

PRACE SoHPCAuthors
Janni Harju, Finland

PRACE SoHPCMentor
Stefan Krieg, JSC, Germany

PRACE SoHPCContact
Janni Harju,
E-mail: janni.harju@yahoo.com

PRACE SoHPCSoftware applied
Matlab Intel Math Kernel Library

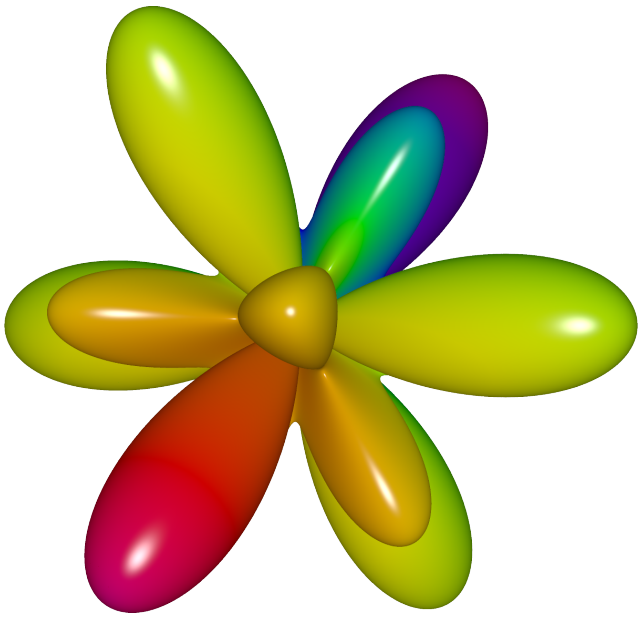
PRACE SoHPCMore Information
software.intel.com/mkl
https://www.mathworks.com/matlab

PRACE SoHPCAcknowledgement
Write any requested acknowledgements or thanks here. Mentors should be asked for them too.

PRACE SoHPCProject ID
1818

Effects of GPU abuse on FMM performance

Wojciech Nawrocki



All existing implementations of the Fast Multipole Method are either entirely CPU-based or only use acceleration for fragments of the algorithm. This work evaluates the feasibility of executing the entire algorithm on graphics processing units, utilising device-wide task queues. A preliminary implementation is evaluated.

What are the effects of electrostatic or gravitational interactions on a set of N particles? This question, also known as the N-body problem, is at the heart of simulations in chemistry, biology, materials science and other disciplines. Naive algorithms can evaluate the resulting forces and potentials in $O(N^2)$ time. For larger systems, which are what is interesting to simulate in practice, this quickly becomes computationally infeasible.

The Fast Multipole Method (FMM)[1] is an algorithm capable of solving the N-body problem in $O(N)$ time, improving upon simpler algorithms by several orders of magnitude for large systems. Unfortunately, to achieve this improvement in time complexity, the algorithm utilises several elaborate, interdependent transformation passes. The resulting workload is irregular and difficult to parallelise.

Graphics processing units (GPUs) provide floating-point processing power orders of magnitude larger than that of CPUs. However, they're designed to execute regular, structured programs – for example, squaring every element in a large array. In general, the more fine-grained (e.g. thread-wise) the control flow, the slower a GPU kernel will execute. Moreover, code running on a GPU is arguably closer to the metal and grants the programmer more control over mechanisms such as memory caching. At the same time, programmers need to be aware of many pitfalls that can result in buggy or slow execution. For example, details of thread scheduling are largely undefined and cannot be relied upon.

Given the nature of the problem and the hardware constraints, we're sure to be presented with an interesting challenge when trying to implement FMM on GPUs.

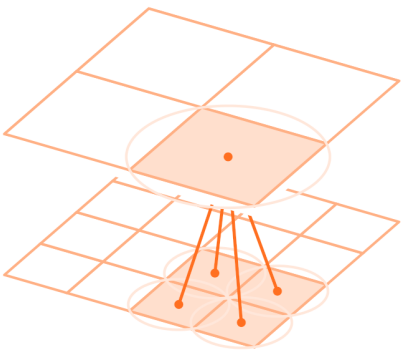


Illustration of quadtree traversal

One part of the algorithm is especially troublesome – it involves traversing a tree up and down, with computation on each level depending on results from the previous level. At the root of the tree we have to wait on a sin-

gle node, which is a serious bottleneck. Luckily, a good solution exists for CPU-based implementations.

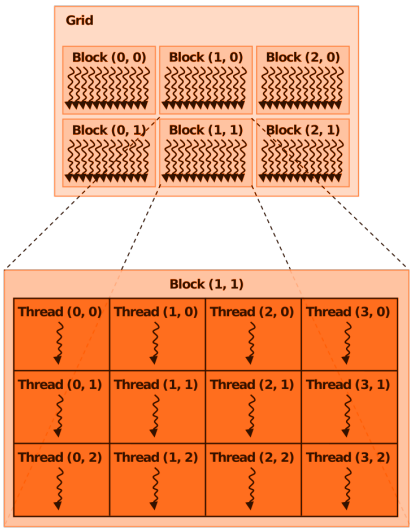
Another phase of the algorithm, known as *P2P* is completely independent from the tree traversal. This means that we can execute it concurrently with the root node computations, so that all cores of the CPU are kept occupied. After the root node has been processed, each core can either start processing lower levels of the tree or continue with *P2P*.

We do not want to control this kind of scheduling manually, so a task-based scheduling system is used. A global task queue (or several, for performance reasons) exists. A *task* is simply a function/computation definition with some associated input. Each task can also depend on others, so that the others have to finish before it can start. Computation at the root node, which depends on all of its children, is an example of this. The scheduling system automatically chooses tasks from the queue and assigns them to each available core. Thanks to this, as long as all tasks in the queue take roughly similar times, all cores are kept occupied and no parallel potential is wasted.

The implementation of a global queue requires communication between cores – we would be in trouble if two grabbed the same task or corrupted memory via a data race. On a CPU, this can be achieved in many ways by using thread-safe structures and synchronisation primitives. We choose specifically which ones we want based on their performance characteristics. With the right choices, this scheme can achieve extremely high parallel performance and scalability with the number of CPU cores.

GPU-based implementation

Implementing the queue on a GPU is a completely different story. The execution model on GPUs groups threads into *blocks* (in CUDA; called *work-groups* in OpenCL). Depending on the used framework (OpenCL 1/OpenCL 2/CUDA), the memory model is somewhere between weak and undefined. Weak means that memory operations across (and even within) blocks are out-of-order, unless special synchronisation instructions are used.[2] Undefined means that goblins might jump out of the screen and eat the programmer.[3]



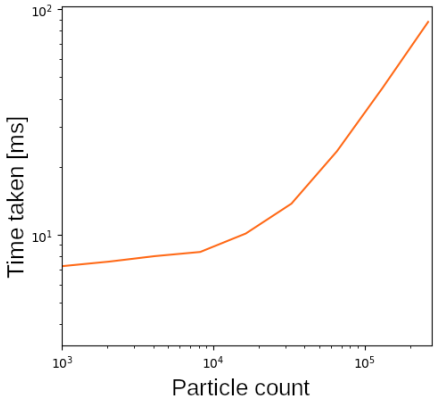
CUDA thread model

Fortunately, a tasking system implementation whose authors dealt with some of these problems already exists – Whippletree.[4] Versions for both CUDA and OpenCL 2[5] are available. So the synchronisation of the queue itself has been dealt with, but to implement FMM we still need to synchronise operations on our own data – the tree, the particle information, etc. Moreover, Whippletree does not support tasks that depend on more than one other task. For example, an operation on a parent node in the tree might depend on all its children – we need to make sure it only starts after the others are done manually.

To decrease the necessary synchronisation, I employed some memory redundancy – the same data might exist in two or more physical memory regions, so that several threads can write to it without racing. The results would later be gathered into a single array.

Where synchronisation between several tasks is required, simple spinlocks in global memory are used. A spinlock is simply a number that must reach a specified value before execution can continue. To wait on the spinlock, a thread loops and continuously fetches the lock until it obtains the desired value. For this to succeed (and not deadlock), another thread must eventually set the lock to the right value. Unfortunately, GPUs provide no forward-progress guarantees. This means that there is no guarantee of other threads ever executing before we finish, and our thread might end up waiting on the lock forever. To alleviate this problem, we launch a precisely determined number of threads, such that there are never more than there are available cores. This works, although to my best knowledge it is not guaranteed to work (goblins).

The result of my work is a megakernel implementation of the Fast Multipole Method. This means a single GPU kernel executes the whole algorithm, as opposed to splitting it into several phases with a kernel for each. It launches fine-grained tasks. For example, each node in the tree is processed by a separate task. This approach works, but proves to have poor performance. An analysis of the concurrent behaviour and emitted machine code reveals where the problems lie.



FMM computation time taken per number of input particles, using a prototype GPU-based implementation (log-log)

An important part of GPU programming is register allocation. There is a number of registers available, how many exactly depends on the specific device. Registers are used to store values of local variables. In CUDA, when a kernel uses too many local variables to fit into the registers, storage for the rest is *allocated* in local memory – a phenomenon known as *register spilling*. Local memory is drastically slower to access. When it is used many times in a loop, a large negative impact on program performance can be observed. Megakernels tend to have a lot of code by definition, so register spilling is often a problem. Whenever CPU synchronisation overhead is not a problem, it is better to split the kernel into smaller ones.

Another issue guiding us towards splitting the kernel is the difference in memory requirements between phases of the FMM algorithm. The amount of *shared* memory available to each block is allocated during the kernel launch and cannot be changed afterwards. However, after executing the first

few phases of FMM, a change in this size using a second kernel launch would result in better memory usage.

Finally, the small size of tasks presents an issue. The tasks have to communicate their results to others, and the queue has to synchronise its state for each task. A communication operation involves a global memory barrier and cache flush, which have to traverse the cache hierarchy, and hence are slow operations. Increasing the task size allows us to perform some of the operations in memory regions which are physically closer to the compute cores (L1 cache/shared memory). This turns out to substantially reduce the overhead of memory operations.

Future work

Given all these problems, a new approach is needed to achieve optimal device utilisation and parallel efficiency, utilising the redesign strategies outlined above. Moreover, a better understanding of the algorithm can be gained by restating it in purely functional/dataflow style and applying transformations that do not alter the computation result.

For example, wherever the algorithm stated this way utilises an abstract reduction algorithm, any actual implementation of reduction can be used on the GPU. This allows us to see which parts of the algorithm allow a variety of implementations, and hence the best-performing ones can be chosen.

References

[1] L. Greengard and V. Rokhlin. A Fast Algorithm for Particle Simulations. *Journal of Computational Physics*, 1997.

[2] J. Preshing. Weak vs. Strong Memory Models. *Preshing on Programming*, 2012.

[3] Undefined Behaviour. *C2 Wiki*, 2014.

[4] M. Steinberger et al. Whippletree: Task-based Scheduling of Dynamic Workloads on the GPU. *ACM Trans. Graph.*, 2014.

[5] https://github.com/apc-llc/whippletree
https://github.com/apc-llc/whippletree-opencl

PRACE SoHPCProject Title

One kernel to rule them all

PRACE SoHPCSite

Forschungszentrum Jülich, Germany

PRACE SoHPCAuthors

Wojciech Nawrocki, University of Edinburgh, United Kingdom

PRACE SoHPCMentors

Andreas Beckmann, FZJ, Germany
Ivo Kabadshow, FZJ, Germany

Wojciech Nawrocki

PRACE SoHPCContact

Ivo Kabadshow
E-mail: i.kabadshow@fz-juelich.de
Wojciech Nawrocki
Email: wjnawrocki@protonmail.com

PRACE SoHPCSoftware applied

C++, CUDA, SYCL, LLVM, Python

PRACE SoHPCAcknowledgement

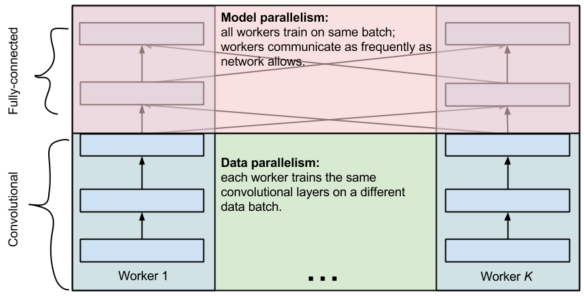
Huge thanks to Ivo, Laura, Andreas, David and everyone else at JSC for being the most friendly, welcoming groups of scientists in existence and for helping me with everything from confusing algorithms to broken bikes. My gratitude goes to the FZJ for hosting me and the SoHPC organisers for making this work possible.

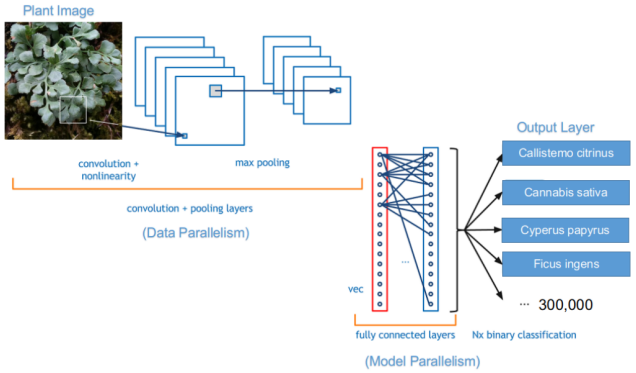
PRACE SoHPCProject ID

1819

Scaling the Neural Net

Hari Prasad Radhakrishnan





Studying the Convolutional Neural Network (CNN) architecture that can handle image recognition using deep learning techniques and optimising its compute-time performance by implementing both model and data parallelism.

Current convolutional Neural-Networking (CNN) models utilise data parallelism in order to scale up and handle large training sets. A drawback is that beyond a certain compute capacity, no further reduction in compute time is possible. The only possible scaling up happens with a larger training data-set. Model parallelism is still experimental and under heavy development. Lately, machine learning frameworks have started to adopt support for model and model-data hybrid parallelism. Training a Neural Net (NN) is compute-intensive. With increasing neurons in output layers, a lot of features should be considered for classification. Therefore, involving deeper NN architecture. To overcome this, all the possible scopes for parallelism has to be exploited. While training the model, the data is fed to the input layer in batches and the weights are updated using back propagation.

Objective

Analysing the bottle-necks due to large input data-feed and simplifying the process by splitting the model to improve the compute-time and efficiently handle the image classifier.

Description

Image recognition is the ability of a mathematical model to classify objects from the images, fed as input and labelling the content of images with meta-tags. Tasks that are easy for humans to intuitively solve are sometimes hard to describe formally, as a set of rules to automate a process. Image recognition is one such processes.

Even if provided, this methodology often fails due to the machine's inability to cognitively learn things. Therefore, deep learning techniques are used, where the mathematical model is designed to resemble the way a human brain functions.

CNN are generally used for image recognition due to its sparse connection between the layers when compared to the previous Neural Nets, requiring massive amounts of power for its compute-intensive nature.

CNN comprises of many layers stacked between the input and the output layer, known as hidden layers. The input image is read as an array of pixels along with the number of channels in the input layer. The output layer consists of the required labels in the form of neurons, which can generally store a value from zero to one.

Each layer consists of neurons that are connected to the adjacent layers with random weights. These neurons act as a placeholder, storing values based on the computations in the previous layers. These hidden layers are comprised of Convolutional and fully-connected layers.

The images are fed in batches to the network. Based on the pixel values, if greater than the set threshold value, the corresponding neurons are activated. These fired neurons send signals to the connected neurons in the next layer. These neurons accumulate the computed value based on weight and value of the previous neuron. The neurons keep firing in the same trend until the output layer. Based on the cost function, given as feedback back-propagation pass is executed and thereby the weights of the neural net is updated. The training is performed until the desired prediction accuracy is achieved. Based on the number of labels, size of the dataset, depth of the neural-net, math-operations, the training time may vary.

There is a lot of research around scaling out convolutional neural networks to a large number of compute nodes, as the computational require-

ments when training complex networks on large-scale datasets become prohibitive. However, most if not all of this work employ data-parallel training techniques, where a batch of image samples is evenly split across the number of workers. Then each worker independently processes in the forward propagation stage, with gradient communication among workers being performed in the backward propagation pass.

Although this technique proved quite successful, it has its own drawbacks. One of the most important is that scaling out to a very large number of nodes implies increasing the batch size, and this leads to more difficulty in the Stochastic Gradient Descent(SGD) optimisation. The second drawback is that data-parallel training works only if the model fits in memory. However, when using model parallelism, there is much more communication involved, particularly in the forward propagation pass.

Tensorflow, a data-flow programming and machine learning framework, is used to develop the neural net architecture. Using Horovod, a distributed library built on top of Tensorflow, hybrid model and data parallelism can be achieved. Since, the approach is still in its developing stages, there is scope on evaluating various schemes of the hybrid approach (model-parallel within node, data-parallel across nodes), and exploring its limitations and finding smart solutions. The built computational model is then tested to a real-world case, "World Flora Classification".

Data Parallelism

Across the data dimension, when the workers are allowed to train on different data samples, it is known as data parallelism. Here, the workers must synchronise the model parameters (or the parameter gradients) to ensure that they are training a consistent model. They are efficient when the amount of computation per weight is high, because the weight is the unit being communicated.

Model Parallelism

Different workers train different parts of the model. They are efficient when the amount of computation per neuron activity is high, because the neuron activity is the unit being communicated.

Implementation

The training time of the neural nets can be optimised by sharing the model across different workers. The penultimate layer is generally a fully-connected layer, where each of its neurons are completely connected with all the neurons of the output layer, making it compute-intensive for the models with larger neurons in the output layer.

For a Resnet-50 architecture, the fully connected layer consists of 1024 neurons. For a model with 300,000 neurons in the output layer, the storage space only for the last two layers is almost two Gigabytes!

Therefore, by implementing model parallelism for the last two layers, the training time can be reduced.

Results

In order to measure the run times, the Tensorflow and Horovod timelines are used. For convolutional and fully connected layers, the forward and backward computation time with respect to the input layers, and the backward computation with respect to the filters are separately tracked. For training the model, the data parallelism approach is utilised by using up to eight GPUs with a weak scaling strategy. And then compared with a hybrid model, where each of the two CPUs connects to four GPUs.

Discussion & Conclusion

Implementing data parallelism for the convolution layers and model parallelism for the fully-connected layers, reduces the train-time drastically. The further scope for parallelism is utilising the previous layers for a different mini-batch while working on the current mini-batch of images.

Table 1: Fully-connected layer(FC)

ResNet50 Architecture		
FC	Output	Memory
1024	300000	≈2 GB

Table 2: Effectiveness of Parallelism

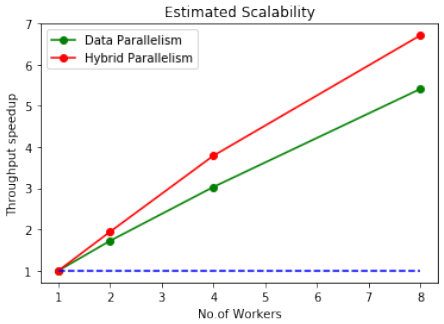
Comparison Chart		
Workers	Data	Hybrid
1	1.00×	1.00×
2	1.72×	1.94×
4	3.03×	3.79×
8	5.40×	6.70×

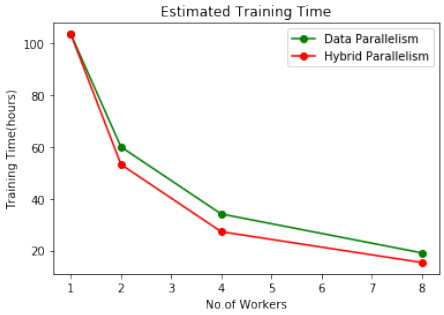
References

[1] Krizhevsky, Alex (2014). One weird trick for parallelizing convolutional neural networks

[2] Ben-Nun, Tal and Hoefler, Torsten (2018). Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis.

[3] Qi, Hang Sparks, Evan R Talwalkar, Ameet (2017). PALEO: A PERFORMANCE MODEL FOR DEEP NEURAL NETWORKS





PRACE SoHPCProject Title

Hybrid-parallel Convolutional Neural Network training

PRACE SoHPCSite

SURFsara, Netherlands

PRACE SoHPCAuthors

Hari Prasad Radhakrishnan,
TU Bergakademie Freiberg, Germany

PRACE SoHPCMentor

Valeriu Codreanu, SURFsara, Netherlands


Hari Prasad Radhakrishnan

PRACE SoHPCContact

Hari Prasad, Radhakrishnan, TU Bergakademie Freiberg, Germany
Phone: +49 176 2677 1429
E-mail: hariprasadr1hp@gmail.com

PRACE SoHPCSoftware applied

Tensorflow and Horovod frameworks (python)

PRACE SoHPCMore Information

https://summerofhpc.prace-ri.eu/hybrid-parallel-convolutional-neural-network-training/

PRACE SoHPCAcknowledgement

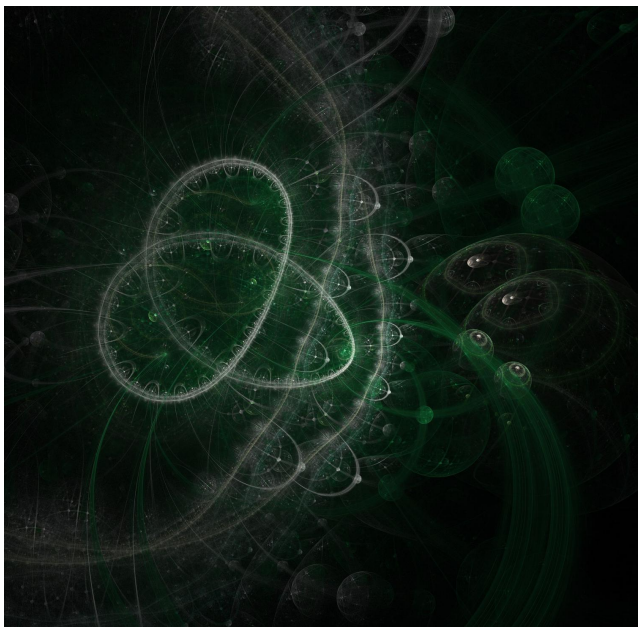
I would like to thank everybody at the SURFsara Computing Centre for their help,and for making the summer enjoyable. I would also like to thank my mentor for his help with the project

PRACE SoHPCProject ID

1820

Optimisation of quantum internet simulator

Filip Kuklis



Because of many time-consuming matrix operations and exponential data and time complexity of quantum simulation, the main idea was to find and optimise suitable operations on accelerators. But there is a bottleneck in transferring data for the used sizes of matrices. We managed to accelerate the simulation example from two to five times depending on the number of qubits and number of iterations only by algorithmic optimisation.

Although the topic of creating a scalable, efficient, quantum simulator has been actively pursued by multiple research groups in the past years, due to the complexity of the solution required (for example multiple types of gates and formalisms) it is still very much open and interesting at least for the HPC, Physics, Chemistry and Machine Learning communities. At the moment, there are multiple solutions available, varying in quality and completeness. What might be one of the more interesting debates is the suitability of GPUs for offloading the dense kernels, considering the overarching issue of steep memory requirements resulting from the use of multiple qubits (memory usage increase grows exponentially).

Qubits and operations

Qubits and all operations on qubits can be represented by vectors and matrices and computed using vector-space operations. This vector-space is rises with the number of qubits exponentially. Therefore the computation time rises exponentially.

Simulator

A simulator of quantum internet was written in Cython language. Cython is an optimising static compiler for both the Python programming language and the extended Cython programming language (based on Pyrex). It makes writing C extensions for Python as easy as Python itself.[1] Cython allows:

- write Python code that calls back and forth from and to C or C++ code
- adding static type declarations
- interact efficiently with multi-dimensional NumPy arrays.
- use combined source code level debugging to find bugs in your Python, Cython and C code.

Idea

Because of big efficiency of matrix operations on GPU the idea was to write a CUDA C code called from Cython code

and offload the operations on GPU. The C code can be called from Cython without copying matrices, but to compute it on GPU the data have to be copied to and from a GPU.

Offloading operations

The two most time consuming suitable operations were identified. The matrix-vector multiplication and Kronecker product, both using complex dense or complex sparse matrices and complex dense vectors. The dense matrices are used for small number of qubits and sparse for bigger number of qubits.

Because of the fact that data has to be copied to and from GPU we began with a copy benchmark. Firstly, the simple parallel C code to compute complex matrix-vector multiplication was written for both sparse and dense matrices. Then the CUDA copy benchmark was written and the copy time of matrices with given dimensions was measured. However, the benchmark showed that only copying the data to and from GPU is much more time consuming than computation on CPU. The Kronecker product showed similar results.

In order to fully characterise the behaviour of our two operations we ran a series of benchmarks for varying sizes of dense and sparse matrices in order to identify the threshold from which it is beneficial to offload the computation to an accelerator. In the case of the Kronecker tensor product we have compared a Cupy implementation running on a Nvidia 1080TI with a Numpy and Scipy optimised one running on a 32 core Intel Boadwell CPU. The results show that when working with a dense matrix of more than 1 million elements the GPU implementation is faster than the CPU one and therefore preferable. Unfortunately, due to the limitations on the accelerator's memory this means that the class of problems that is solvable is quite limited - between approximately 20 to 25 qubits. In the case of sparse matrices, even when using cuBLAS and cuSPARSE, GPUs were outperformed by CPUs in all cases. Future work for this project will include an optimised implementation of the product that leverages more of the architecture specific features, similar to the work done.[2]

Data transfer optimisation

After this results we focused on the matrix and vector data for multiplication. The vector represents the state of the qubits and the matrix is operation on this state. The state vector changes with every iteration, however that is not the case for the operation matrices. These can be effectively reused and cached. After this, we focused only on copying the state-vector. The result is shown in Fig.1. For 20 qubits the state vector has one million elements and have 16MB(1m(num-elements)*2(complex)*8(double)). The one computation on CPU takes cca 2.25ms and copying data takes cca 3.3 ms(1.5ms to GPU 1.3ms from GPU). We are getting 10-12GB/s which is 60-75% of maximum bandwidth of PCI-e16x[3] which is normal for that small data amount.

On the other hand, after looking on Kronecker product data we found out that for outer and inner vector space padding the operation is computed always between matrix, which can be dense or sparse, and identity matrix.

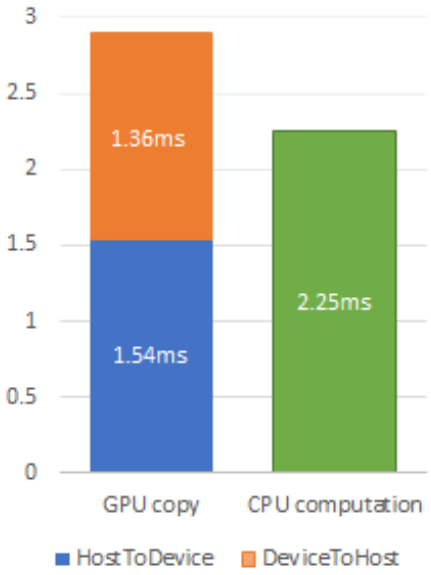


Figure 1: Copy benchmark result.

Vector space padding optimisation

Outer vector space padding

As we can see in Fig.4 for outer vector space padding – we actually don't have to compute Kronecker product because we can only copy data and create a new bigger matrix. There are two versions of outer vector space padding: before - where there is an identity matrix operated with given matrix, and after - when given matrix is operated with identity matrix. We can see some pattern and we have to compute only indexes to the new matrix. The ratio between zeros and non-zeros is quite big in the output matrices and with bigger matrices, it actually rises. Therefore the output matrices are sparse matrices and the zeros are not stored.

Inner vector space padding

On the other hand, the inner vector space padding is slightly different. There is an operation between the identity matrix and dense matrix which is split into smaller matrices and the Kronecker product is computed separately. Then these matrices have to be stuck together to one output matrix. But here, we can see nearly the same pattern and we can compute indexes from the original matrix without splitting it and then stick it together. In this case also, another optimisation was implemented. It is named as innerif because of comparing of elements in dense matrix to zero with if statement. This zeros are not stored, but the Kron operation stores this zeros. This optimisation may save time in some cases but also saves memory consumption.

Result

The code for inner and outer vector space padding was written in C. With this optimisation our simulation example runs from two to 5 times faster depending on the number of qubits and number of iterations.

Table 1: 20 qubits 10 iterations

Code op.	Comp. time	Speedup
original	32.5s	1x
outer	9.3s	3.49x
outer_inner	9s	3.61x
outer_innerif	7.5s	4.33x

Table 2: 6 qubits 5000 iterations

Code op.	Comp. time	Speedup
original	27	1x
outer	24.4	1.1x
outer_inner	10.9	2.48x
outer_innerif	11.2	2.41x

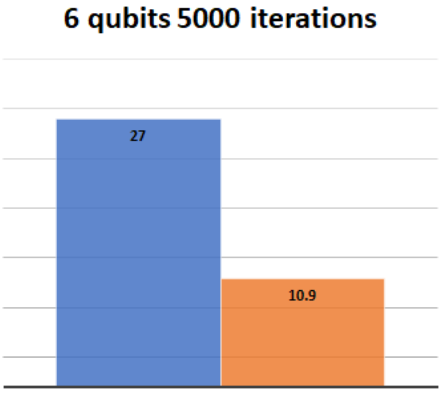


Figure 2: Computation time of simulation in seconds. Original vs optimised code.

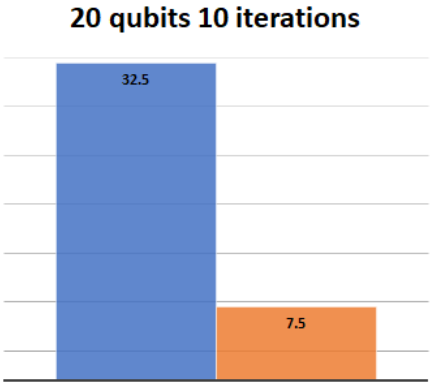


Figure 3: Computation time of simulation in seconds. Original vs optimised code.

In the tables Tab.1 and Tab.2 we can see computation time and speedup of code with different optimisations. For big matrices (20 qubits) the if optimisation saves a lot computing time, but for small matrices the if statement overhead actually slows the code a bit.

On the other hand, this optimisation saves memory in both cases.

References

[1] About Cython, https://http://cython.org/

[2] B. Liu and C. Wen and A. D. Sarwate and M. M. Dehnavi(2017) A Unified optimisation Approach for Sparse Tensor Operations on GPUs.

[3] PCI EXPRESS BASE SPECIFICATION, REV. 3.0, http://composter.com.ua/documents/PCI_Express_B

PRACE SoHPCProject Title

Large scale accelerator enabled quantum simulator

PRACE SoHPCSite

SURFSara, Netherlands

PRACE SoHPCAuthors

Filip Kuklis, [association,] country

PRACE SoHPCMentor

Damian Podareanu, SURFsara, Netherlands

Filip Kuklis

PRACE SoHPCContact

Filip, Kuklis, BUT
Phone: +421 915 745 565
E-mail: xkukli03@stud.fit.vutbr.cz

PRACE SoHPCSoftware applied

NetSQUID

PRACE SoHPCMore Information

qnetsquid.org

PRACE SoHPCAcknowledgement

Write any requested acknowledgements or thanks here. Mentors should be asked for them too.

PRACE SoHPCProject ID

1821

$$\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix}\otimes\begin{pmatrix}0.5+0.5j&1+0j\\0+1j&0.5j\end{pmatrix}=\begin{pmatrix}0.5+0.5j&1+0j&0&0&0&0\\0+1j&0+0.5j&0&0&0&0\\0&0&0.5+0.5j&1+0j&0&0\\0&0&0+1j&0+0.5j&0&0\\0&0&0&0&0.5+0.5j&1+0j\\0&0&0&0&0+1j&0+0.5j\end{pmatrix}$$

$$\begin{pmatrix}0.5+0.5j&1+0j\\0+1j&0+0.5j\end{pmatrix}\otimes\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix}=\begin{pmatrix}0.5+0.5j&0&0&1+0j&0&0\\0&0.5+0.5j&0&0&1+0j&0\\0&0&0.5+0.5j&0&0&1+0j\\0+1j&0&0&0+0.5j&0&0\\0&0+1j&0&0&0+0.5j&0\\0&0&0+1j&0&0&0+0.5j\end{pmatrix}$$

Figure 4: Example of outer vector space padding from left and from right.

RHadoop scripts for automatic storing, preprocessing, and visualising big data using clustering algorithms.

RHadoop on the trial of radiation clusters

Marcin Konieczny

Knowledge about radiation is extremely important, because it has a big influence on our lives. This project helps detecting radiation doses in your area.

Big data is all around us. Most of the people know how to send an e-mail or a text message. Everyday we produce about **2.5 quintillion** (2^{18}) bytes of data. There are many types of it: from texts through sounds and pictures.
The dataset which was the subject of my work consist of 90 million ionizing radiation measurements provided by 'Safecast'. It is a global volunteer-centered citisen science project working to empower people with data about their environments. Radiation has a big influence on our lifes. There are many units which describe radioactivity - all of them are connected with energy. In my project I used the *Sievert* unit.

$$1Sv = \frac{1J}{1Kg} \qquad (1)$$

Exposure to **100 mSv** a year is the lowest level at which any increase in cancer risk is clearly evident. A cumulative **1,000 mSv** would probably cause a fatal cancer many years later in five out of every 100 persons exposed to it[1] .
We have contact with radiation everywhere. Each year people receive about **3 mSv** (global average) of 'background radiation' which is inhalation of air, ingestion of food and water, or cosmic radiation from space. This value varies among different places. For background radiation we also include medical treatments.

Table 1: Radiation doses per treatment

Name of treatment	Radiation dose
(CT)–Colonography	6 mSv
(CT)–Head	2 mSV
Mammography	0.4 mSv
Chest X-ray	0.1 mSv
Dental X-ray	0.005 mSv

The presented data below, are a part of the background radiation and shows the doses people have received by a certain region, from the air.

1 Final product

The main goal of the project was developing RHadoop scripts for automatic storing, preprocessing, and visualising big data using clustering algorithms. Another aim was preparation of the PRACE FutureLearn MOOC tutorial.

Tutorial

PRACE FutureLearn MOOC goal is to help understand Hadoop and R. First of all, participants have a chance to download big data directly into their HDFS. Then they will become acquainted with the Hadoop environment and some R scripts which are about statistics, clustering, and visualisation. I supported my work with AWK sciprts.

RHadoop

We use RHadoop for provide concurrent processing. RHadoop is a collection of R packages that allow users to manage and analyse data with Hadoop. Hadoop's main component is a NameNode, a master server that manages the file system namespace and regulates access to files by clients. There are a number of DataNodes, usually one per node in the cluster, which manage storage. With every DataNode there are several

associated computing cores. All files are split into blocks which are stored in DataNodes.

HDFS

Hadoop Distributed File System is a system designed to run on commodity hardware. HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. It provides high throughput access to application data and is suitable for applications that have large data sets.[3]

RHadoop parallelisation

Every file is divided into 128 MB block. With every block there is a number of cores provided by the resource manager 'YARN'. The important thing in this, is a replication number which refers to the number of copies of particular block in a distributed file system. The higher the replication number is, the faster the software can work in parallel on one block of file. Each block has different cores assigned by YARN.

Map-Reduce

Map-Reduce is programming paradigm with allows to process vast amounts of data. Firstly, the master-node divides data into smaller, independent chunks. Then each worker node works on one of them. It uses **Map** function which performs filtering and sorting. The output from **Map** goes to **Reduce** function which performs join or summary operation (like searching for a minimum or maximum) .

Table 2: Number of mappers and blocks by size of the file

Size	Mapper	Block
12 MB	10	1
270 MB	223	3
10 GB	8305	77
12.5 GB	10266	96

K-means

K-means clustering is a type of unsupervised learning. The main goal is to find centroids with similar values. In my case K-means operates on coordinates. It looks for the closest neighbours on the map and make from them one point whose value is the average of its elements. It is very sensitive on the outliers influence, that's why preprocessing of data is necessary.

Described above tools enable us to process big data. Without any help from them, processing is crunching the numbers. Thanks to RHadoop we can make it much faster.

In this project, every mapper returns 100 of groups generated by K-means. Then reducer joins them all. From the biggest file, the program generates few hundred of thousand coordinate points, which are represent a neighbourhood.

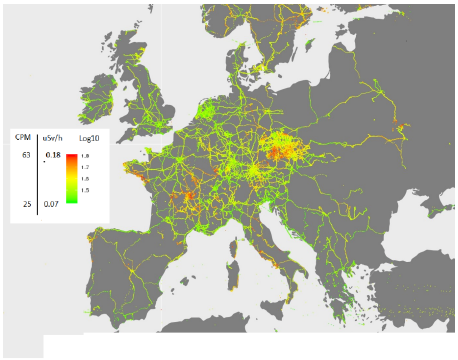


Figure 1: Radiation in Europe

HPC effectivity

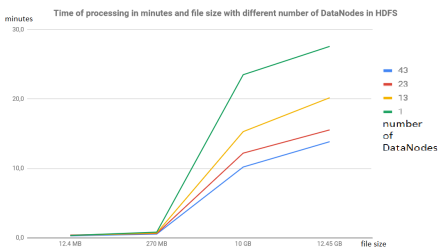


Figure 2: Time of processing in minutes and file size with different number of DataNodes in HDFS

Results

Europe is safe. European values usually oscillate around 0-150 cpm (150 cpm is about 0.45 Sv/h - yearly dose is around 4mSv). There are some places with constantly a higher radiation level than the average. This is particularly noticeable at Central Bohemian Granite Highlands in Czech Republic and in Massif Central in France. It is caused by Granite mountains. Granite contains thorium, radium and uranium 238 which makes it a little bit radioactive.

Nuclear plants

Another thing is the matter of damaged nuclear plants and their surrounding areas. We feel the effects of disaster in Chernobyl to this day. Despite the core of the nuclear plant now present under safety layer, the nearest area is still under big influence of radiation doses. Buildings nearby have devices detecting value of 40Sv/h (equal 12.000 CPM, yearly 350 mSv).

Cosmic radiation

Filtering higher values of the dataset allows to show radiation measured during flights. It is because of the influence of the cosmic radiation. Also in the high mountains areas we can see big change compare to the average value of radiation.

What Next ?

The project will continue. More data will be used: like device Id (to eliminate redundant observations) or dates to show how radiation changed over months.

References

[1] Radiation limits: https://www.reuters.com/article/us-how-much-radiation-dangerous/how-much-radiation-is-dangerous-idUSTRE72E79Z20110315

[2] Radiation doses in treatements: https://www.radiologyinfo.org/en/info.cfm?pg=safety-xray .

[3] HDFS: https://hadoop.apache.org/docs/r1.2.1/hdfs$_{design.htm}$

PRACE SoHPCProject Title

Big Data clustering with RHadoop

PRACE SoHPCSite

University of Ljubljana, Slovenia

PRACE SoHPCAuthors

Marcin Konieczny,
Poznan University of Technology, Poland

PRACE SoHPCMentor

prof. dr. Janez Povh, Faculty Of Mechanical Engineering, Slovenia


Marcin Konieczny

PRACE SoHPCContact

Marcin Konieczny
Phone: +48 695640640
E-mail: marcin.konieczny.poznan@gmail.com

PRACE SoHPCSoftware applied

Hadoop (HPC processing), R, Python

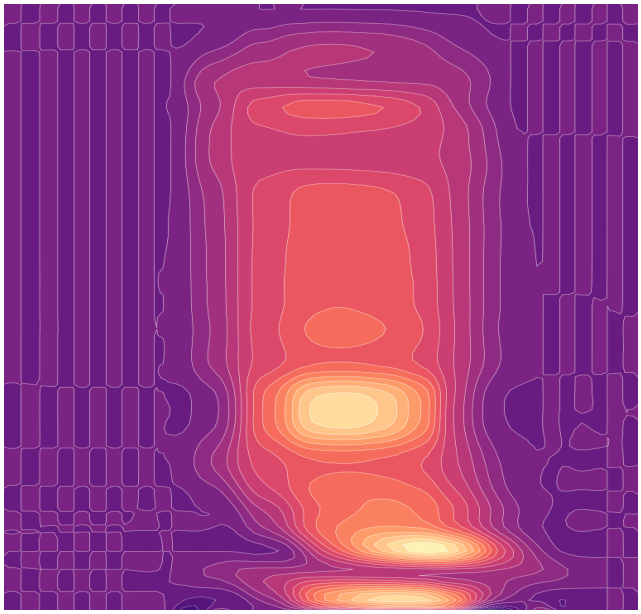
PRACE SoHPCMore Information

I would like to thank to my site coordinators, Prof. Janez Povh and Dr. Leon Kos from University of Ljubljana for their help and support

PRACE SoHPCProject ID

1822

Visualization of nuclear fusion HPC data



Mario González Carpintero

Plotting the results obtained by HPC simulations is an excellent and efficient way to extract conclusions from them. For this purpose, we have managed to create quite smart plots from the data obtained from ITER fusion simulations, by writing a custom plugin for ParaView. The data from the simulations is usually complex, but our interface provides an easy way to select what we want to plot, and to customise the plots once they are created.

A cheap and efficient way to study the processes related to nuclear fusion (which are not easy to perform experimentally) is by performing simulations by computation. This is what is being done for the plasma that eventually will burn in ITER's fusion reactor.

This reactor, so called Tokamak, has a toroidal shape (see Fig. 1), and the applied magnetic fields make the plasma spin inside, keeping it confined during each shot. In most of the simulations, one can assume that the reactor is a symmetric device around its axis (coloured green in the picture), so the angle around that axis (Ψ, orange) becomes one degree of freedom, and the outcome of the experiment becomes independent of it.

That means that we can reduce our study to a transverse cut of the toroid (like the plan coloured red). This is a huge simplification!

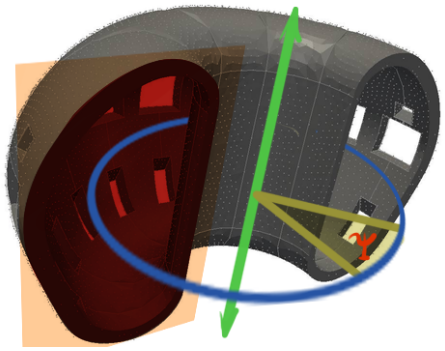


Figure 1: Scheme of a cut of the Tokamak chamber. We can notice its cylindrical symmetry.

The dataflow: from a complex database to a smart plot

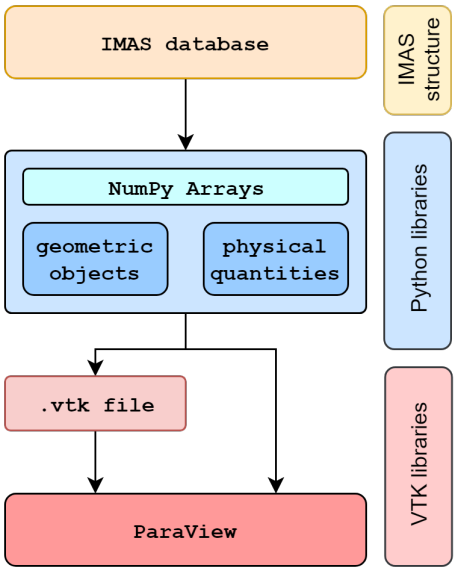
The complex framework intended for plasma computations on ITER is called Integrated Modelling & Analysis Suite (IMAS) and the IMAS databases are called Interface Data Structures (IDS). These databases contain a complex tree structure with a huge amount of leafs, although only a few amongst all of them are actually filled with data. In this database, for each stored object, both its geometry and physical quantities related to it (as scalars associated with each coordinate), are described. Every object is composed by at least some points, that may be joined together by lines, or either 2D or 3D cells.[1] Also, a physical quantity can be defined by setting its value on each point of the mesh.

These objects are complex, and the cleanest way to deal with them *outside* the IDSs is by using the VTK libraries, that allows the creation of a VTK object that not only contains all the required information, but also can be displayed in ParaView.

In fact, we have managed to use only Python code (and no C++ at all), so instead of the VTK libraries, we will import the VTK module that contains itself all the VTK functions that we need. Note that creating a `.vtk` file for its visualisation on ParaView is not necessary if we run our script inside ParaView. If we decide to create it, ParaView be-

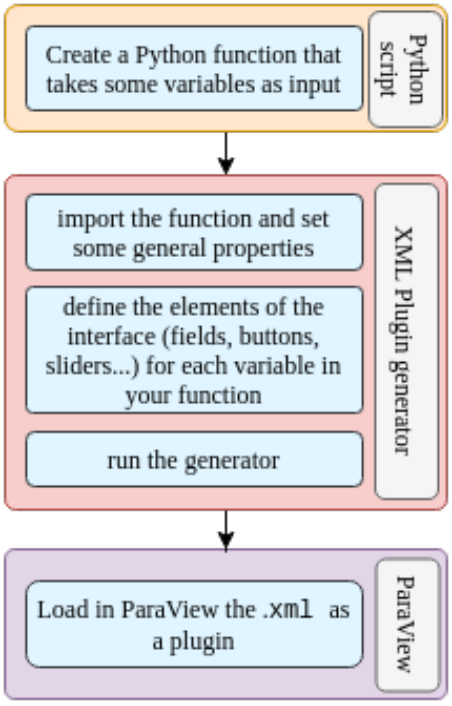
comes just a visualisation tool, that can be replaced by any other software based on VTK.

The following scheme shows the dataflow from IMAS database to the final visualisation:



Creating a custom plugin for Para-View. From a Python script to a XML plugin

Schematically, this process can be described as follows:



But that is not the only way. In fact, there are many patterns to create a plugin for ParaView. The most common one is by writing it in C++ and *compiling* it with CMake. This way requires a `.cxx` file with the source code, a `CMakeList.txt` file necessary for the compilation and an XML file describing the interface. All of them are compiled into a single `.so` file that is then loaded as a plugin into ParaView.

Our way, on the other hand, requires only an XML file that contains both the code of the plugin and the description of the interface. This XML will be loaded itself as a plugin in ParaView.

The main advantage of this method is that there is no need to recompile each time we make a change in our script, so modifying and loading the plugin is actually very easy and straightforward.

In our scheme, both the *Python script* and the *ParaView* stages work as usual. Now we will describe the usage of the XML generator.

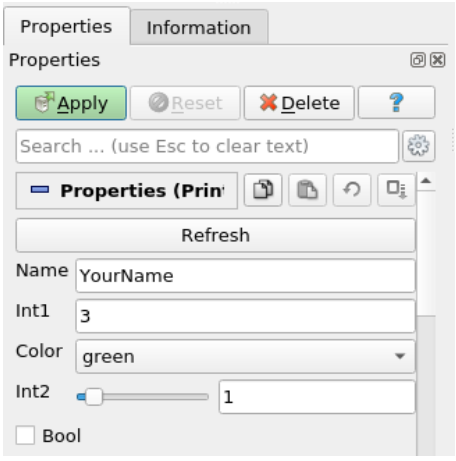
The XML generator

Writing an XML containing both the whole code as a single string, and the description of the interface, is something tedious that usually cannot be done manually. To deal with it, we have created in Python an `XML generator` that does the job.[2] Defining the interface inside is much simpler than doing it directly in the XML. The code

```
AddInt("MyInt", "3")
```

shows the simplest example. It produces a field for typing an integer, with 3 as its default value:

Then, the generator links all the variables of our plugin's code with their correspondent field on the interface and the `.xml` is generated.
The next picture illustrates a more complex example:



Here, we have added a *Refresh button*, a *string field*, an *integer field*, a *drop-down list*, a *slider* and a *boolean variable*.

Then, if we want to change the code of our plugin, we just do it and run the XML generator again. It will take less than half a second to update the plugin with the new code. Of course, considerably faster than compiling in C++!

Inside ParaView: *the TimeTools plugin*

Creating a plugin for ParaView implies that we can use its interface for setting all the input data, which is actually a good thing. Moreover, the output will be shown inside ParaView, which can be convenient in some cases.

On the other hand, our plugin has gone one step further than plotting a simple mesh with its scalars. It can also read multiple timesteps from IMAS and merge all of them into a single object inside ParaView. This object can be divided also into multiple blocks (as, for example, a car is divided into doors, tires, seats...), and each block will be coloured by the scalars defined on it.

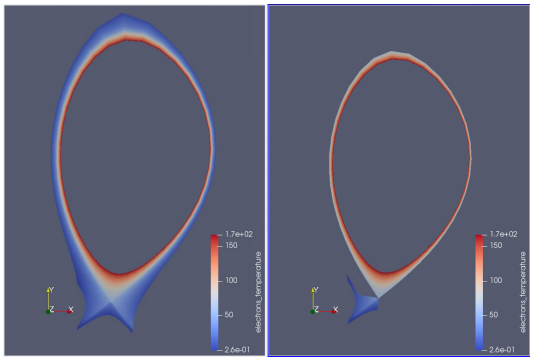
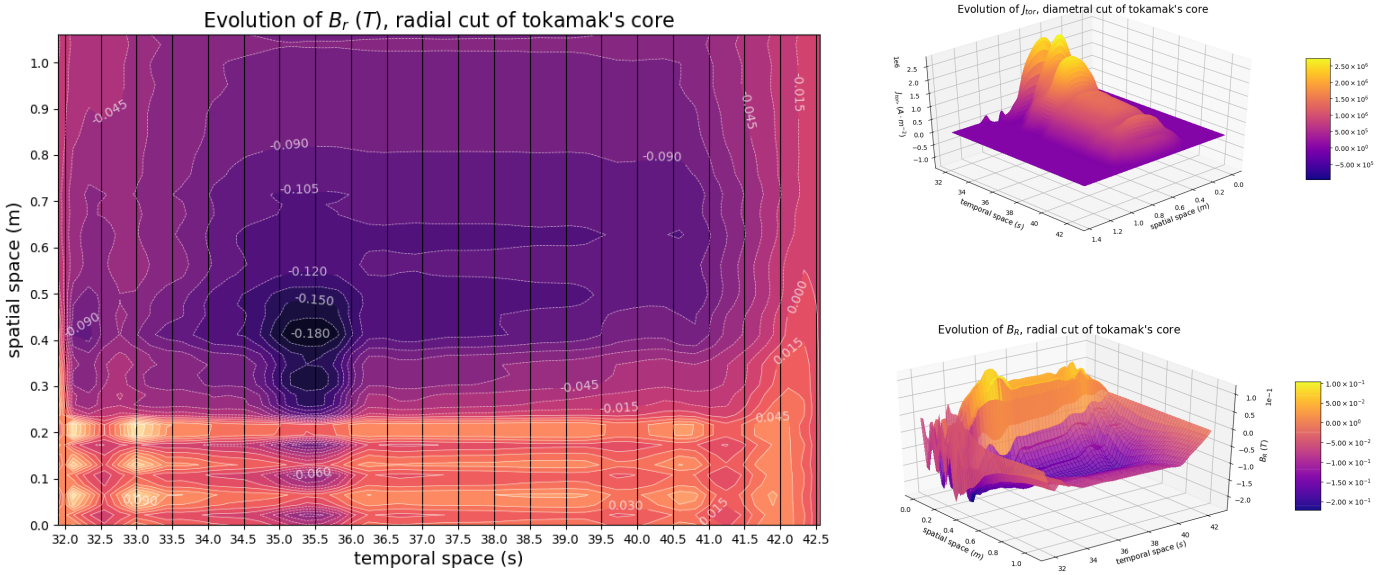


Figure 2: On the left, a transversal cut of the Tokamak, coloured by the temperature of the electrons. On the right, the same plot, restricted to the *core* and the *inner divertor* blocks.

Many of the blocks are lines defined in the object, that represent some important parts of the Tokamak. The main feature of our plugin is the possibility of taking all the points conforming these lines and tracking through time some quantities defined on them, plotting on the screen either contour or 3D plots, from which we could extract interesting conclusions just with one sight.

We can also, using the same plugin, study the behaviour of one single point, producing a smart 2D plot tracking the selected physical quantity versus time.



On the left, the value of the magnetic field is tracked over a line (vertical axis) and over time (horizontal axis), in a contour plot. On the right, down, its correspondent 3D plot rendered with low resolution and some transparency. On the top the correspondent 3D plot to the featured image, generated with large resolution. It tracks the toroidal component of the current density also in a linear subset.

Is *actually* ParaView a essential piece of our project?

The four tools inside the plugin consist in a set of Python scripts that are run inside ParaView, but there are two relevant reasons why we would like to make them standalone.

- First of all, we would like to be able to embed them in another python-based application non related with ParaView. Many of the tools used in ITER to visualise their data are, without going further, written in PyQt.
- Furthermore, ParaView uses for plotting its own *matplotlib*, which is quite more limited than the real one. This means that we can make much more customizable plots *outside* ParaView.

For these reasons, we have written our scripts in a way that the differences between the ParaView's one and the standalone are reduced to the `import paraview` statement, the paraview's equivalent to the `figure()` and `show()` of matplotlib, and some little details that depend on the script.

For the main tool (which prepares the data for the plots and displays it into ParaView as shown, for example, on Fig. 2), some code is commented and the standalone version doesn't plot anything. It just creates the `.vtk` files (in case that we want to open them later in ParaView) and the `.txt` files with the necessary data to make the plot.

The same `.txt` files are used both by our plugin and by the independent scrips, so if we load this tool from ParaView, and we want to make the plots outside (which is much smarter, as we have already remarked), it's not necessary to re-generate any data!

Results and future work

The main outcome of this project are the scripts that we have created for dealing with IMAS databases and the XML plugin creator. Unlike the rest of our work, this last tool is totally independent from IMAS or ITER. It can be used by anyone for creating any plugin for ParaView, without knowing it's internals. That's the reason why we have uploaded it to GitHub, alongside with a complete tutorial demonstrating it's features by creating a simple plugin.

All of these tools are to be included into the ITER's SOLPS-GUI project (which is also focused on the visualisation of that kind of data), alongside with the proper tutorial about using the plugins. Both files and the correspondent tutorial will be available only for ITER people, as long as the files will be uploaded to their private repository.

Acknowledgements

I would like to thank Dejan Penko, for all his time and help during the summer, and for all the resources (Cluster account, data for analysis, documentation...) that I have needed to complete my project. Also Leon Kos, for all the time that he has spent with me and for everything that I have learnt from him.

I'm also very grateful to all the people in the office, for all the help and kindness inside and outside the faculty.

References

[1] For further information about how VTK defines a mesh, refer to `www.vtk.org/wp-content/uploads/2015/04/file-formats.pdf`.

[2] All the source of the Plugin Creator can be found in our GitHub reprository: `https://github.com/mariohyls/ParaViewXMLPluginCreator`

PRACE SoHPC: Project Title
Visualization schema for HPC data

PRACE SoHPC: Site
University of Ljubljana, Slovenia

PRACE SoHPC: Authors
Mario González Carpintero, University of Oviedo, Spain

PRACE SoHPC: Mentor
Dejan Penko, University of Ljubljana, Slovenia

PRACE SoHPC: Contact
E-mail: leon.kos@lecad.fs.uni-lj.si

PRACE SoHPC: Software applied
VTK, Python, ParaView

PRACE SoHPC: More Information
video presentation:
https://www.youtube.com/watch?v=jK-YU547qzs&list=PLhpKvYInDmFXUyp_pWBM-h1NCD6GEUfgp&index=22
VTK and ParaView webpages:
www.vtk.org
www.paraview.org

PRACE SoHPC: Project ID
1823

Mario González Carpintero

www.summerofhpc.prace-ri.eu